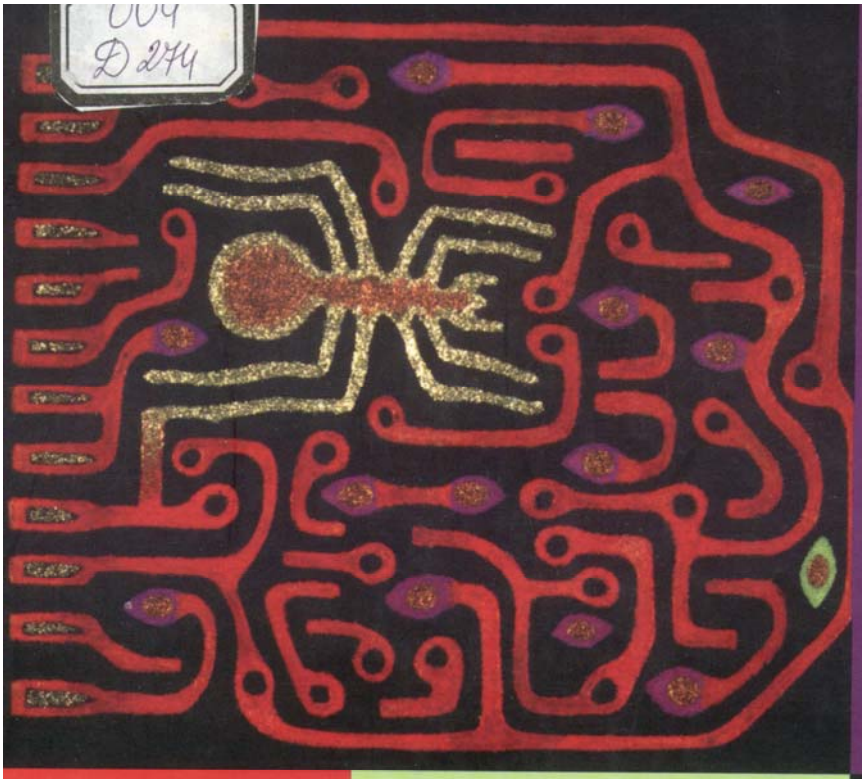


ВОСЬМОЕ ИЗДАНИЕ

Введение в системы баз данных

К. Дж. Дейт

ВИЛЬЯМС
PEARSON
Addison Wesley

ВОСЬМОЕ ИЗДАНИЕ

Введение в системы баз данных

EIGHTH EDITION

An Introduction to Database Systems

C.J.Date

Boston • San Francisco • New York
London • Toronto • Sydney • Tokyo • Singapore • Madrid
Mexico City • Munich • Paris • Cape Town • Hong Kong • Montreal

ВОСЬМОЕ ИЗДАНИЕ

Введение в системы баз данных

К. Дж. Дейт

Москва • Санкт-Петербург • Киев

2005

№.

ББК 32.973.26-018.2.75
Д27
УДК 681.3.07

Издательский дом "Вильяме"

Зав. редакцией *С.Н. Тригуб* Перевод с английского и редакция *К.А. Птицына*

По общим вопросам обращайтесь в Издательский дом "Вильяме" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

115419, Москва, а/я 783; 03150, Киев, а/я 152

Дейт, К. Дж.

Д27 Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильяме", 2005. — 1328 с.: ил. — Парал. тит. англ.

ISBN 5-8459-0788-8 (рус.)

Новое издание фундаментального труда Криса Дейта представляет собой исчерпывающее введение в очень обширную в настоящее время теорию систем баз данных. С помощью этой книги читатель сможет приобрести фундаментальные знания в области технологии баз данных, а также ознакомиться с направлениями, по которым рассматриваемая сфера деятельности, вероятно, будет развиваться в будущем. Книга предназначена для использования в основном в качестве учебника, а не справочника, и поэтому, несомненно, вызовет интерес у программистов-профессионалов, научных работников и студентов, изучающих соответствующие курсы в высших учебных заведениях. В ней сделан акцент на усвоении сути и глубоком понимании излагаемого материала, а не просто на его формальном изложении.

Книга, безусловно, будет полезна всем, кому приходится работать с базами данных или просто пользоваться ими.

ББК 32.973.26-018.2.75

ISBN 5-8459-0788-8 (рус.)
ISBN 0-321-19784-4 (англ.)

Оглавление

Содержание

Об авторе

К.Дж.Дейт (C.J.Date) — независимый публицист, лектор, ученый и консультант, специализирующийся на технологии реляционных баз данных. Он проживает в г. Хилдсбурге, штат Калифорния.

Начиная с 1967 года, Дейт несколько лет работал математиком-программистом и инструктором по программированию в компании Leo Computers Ltd. (Лондон, Великобритания). После этого он работал в лаборатории IBM (UK) Development Laboratories над интеграцией функций баз данных в язык PL/I. В 1974 году он перешел в калифорнийский центр IBM Systems Development Center, где отвечал за разработку языка баз данных, известного в настоящее время как Unified Database Language (UDL). Впоследствии Дейт принимал участие в техническом планировании и проектировании внешних интерфейсов для продуктов реляционных баз данных SQL/DS и DB2 корпорации IBM. В мае 1983 года он покинул компанию IBM.

Дейт работает с технологиями, связанными с базами данных, свыше 30 лет. Он одним из первых осознал основополагающее значение новаторской работы Э.Ф.Кодда (E.F.Codd) по реляционной модели. Дейт читал лекции по техническим вопросам (преимущественно по тематике баз данных, в частности по реляционным базам данных) во всей Северной Америке, а также в Европе, Австралии, Латинской Америке и на Дальнем Востоке. Он является автором или соавтором не только этой, но и других книг по базам данных: *Temporal Data and the Relational Model,* 2003 (издательство Morgan Kaufmann); *Foundation for Future Database Systems: The Third Manifesto* (второе издание), 2000 (издательство Addison-Wesley), в которой даны развернутые предложения по развитию данной области; *Database: Л Primer,* 1983, в которой базы данных рассматриваются с точки зрения неспециалиста; серии книг *Relational Database Writings* (1986, 1990, 1992, 1995 и 1998 годы), в которых фундаментально изложены различные вопросы реляционной технологии; а также серии книг, посвященных отдельным системам и языкам: *A Guide to DB2* (четвертое издание), 1993; *A Guide to SYBASE and SQL Server,* 1992; *A Guide to SQL/DS,* 1988; *A Guide to INGRES,* 1987; *и A Guide to the SQL Standard* (четвертое издание), 1997. Книги Дейта переведены на многие языки, в том числе голландский, греческий, испанский, итальянский, китайский, корейский, немецкий, польский, португальский, русский, французский и японский. Кроме того, его книги изданы с использованием шрифта Брайля для слепых.

Дейт опубликовал больше 300 технических статей и научных работ и внес значительный вклад в теорию баз данных. В течение многих лет он ведет постоянную колонку в журнале *Database Programming & Design.* Кроме того, Дейт регулярно пополняет материалы Web-узла http://dbdebunk.com. Профессиональные семинары по технологии баз данных, проводимые им как в Северной Америке, так и в других регионах мира, считаются непревзойденными по качеству представленного материала и ясности изложения.

Дейт с отличием окончил Кембриджский университет в Великобритании (в 1962 году он защитил диплом бакалавра гуманитарных наук, а в 1966 году— магистра гуманитарных наук), а со временем получил почетную ученую степень доктора технических наук в де Монфортском университете в Великобритании (1994).

Предисловие к восьмому изданию

Настоящая книга представляет собой исчерпывающее введение в очень широкую в настоящее время область теории систем баз данных. С ее помощью читатель сможет приобрести фундаментальные знания в области технологии баз данных, а также ознакомиться с направлениями, по которым эта область, вероятно, будет развиваться в будущем. Книга предназначена для использования в основном в качестве учебника, а не справочника (но я надеюсь, что ее в какой-то мере можно будет использовать и в качестве справочного руководства). В книге сделан акцент на усвоении сути и глубоком понимании излагаемого материала, а не просто на его формальном изложении.

ДЛЯ КОГО ПРЕДНАЗНАЧЕНА ЭТА КНИГА

В целом, книга ориентирована на читателей, которые профессионально работают с компьютерами в той или иной области и хотят получить общее представление о теории и практическом использовании систем баз данных. Предполагается, что читатель имеет, по крайней мере, базовые знания в следующих областях:

- средства управления памятью и файлами (индексация и т.д.) в современных ком
  пьютерных системах;
- средства хотя бы одного из языков программирования высокого уровня (таких
  как Java, Pascal, PL/I и т.д.).

Что касается ознакомления с первой из указанных областей, то подробное учебное руководство по относящимся к ней темам приведено в приложении Г, "Структуры хранения и методы доступа".

Структура книги

Автор хотел бы подчеркнуть свою обеспокоенность тем, что восьмое издание этой книги приобрело такой большой объем. Тем не менее, технология базы данных стала, несомненно, очень обширной областью знаний, поэтому невозможно рассмотреть всю относящуюся к ней проблематику в книге, которая занимает меньше 1000 страниц (и действительно, большинство книг, которые конкурируют на рынке компьютерной литературы с данной книгой, также имеют объем приблизительно 1000 страниц). Данная книга разделена на шесть частей.

- Часть I. Основные понятия.
- Часть II. Реляционная модель.
- Часть III. Проектирование базы данных.
- Часть IV. Управление транзакциями.

- Часть V. Дополнительные темы.
- Часть VI. Объекты, отношения и язык XML

Каждая часть в свою очередь разделена на несколько глав.

- Часть I (четыре главы) — это обширное введение в теорию систем баз данных во обще и реляционных систем в частности. Здесь также излагаются основы стан дартного языка баз данных **SQL.**
- Часть II (шесть глав) содержит подробное и весьма основательное описание реля **ционной модели,** которая является теоретической основой не только для собствен но реляционных систем, но фактически для всей сферы применения баз данных.
- Часть **III** (четыре главы) включает обсуждение основных проблем **проектирования баз данных.** Три главы посвящены теории проектирования, а в четвертой рассмат риваются семантическое моделирование и модель"сущность-связь".
- Часть IV (две главы) посвящена описанию средств **управления транзакциями** (т.е. средств восстановления и обеспечения параллельной работы).
- Часть V (восемь глав) охватывает довольно разнообразную тематику. Но в целом в ней показано, что с помощью реляционных понятий можно гораздо проще опи сать различные области применения технологии баз данных, включая **защиту дан ных, распределенные базы данных, хронологические данные, системы поддержки принятия решений** и т.д.
- Наконец, в части VI (три главы) показано, какие следствия вытекают из стремле ния применить **объектную технологию** в системах баз данных. Глава 25 посвящена исключительно описанию **объектных систем,** в главе 26 рассматривается возмож ность сближения между объектной и реляционной технологиями и обсуждаются принципы построения **объектно-реляционных** систем, а в главе 27 приведено опи сание того, как связаны между собой объектная технология и базы данныхXML.

Кроме того, в книге имеется пять приложений. В приложении А приведен краткий обзор принципиально новой, полностью отличной от существующих технологии реализации, получившей название **модель TransRelational™,** в приложении Б представлена грамматика выражений SQL в **форме Бэкуса-Наура (БНФ),** в приложении В приведен **глоссарий** наиболее важных сокращений и специальных символов, применяемых в тексте книги. Приложение Г, как уже было указано выше, включает учебное руководство по **структурам хранения и методам доступа,** а в приложении Д можно найти ответы к некоторым упражнениям, приведенным в главах книги.

ДОПОЛНИТЕЛЬНЫЕ МАТЕРИАЛЫ, ДОСТУПНЫЕ В ОПЕРАТИВНОМ РЕЖИМЕ

Представленные в этом разделе дополнительные материалы предназначены только для квалифицированных преподавателей. Для получения информации о том, как приобрести эти учебные материалы, обратитесь в местное торговое представительство издательства Addison-Wesley или отправьте по электронной почте письмо по адресу aw.cse@aw.com.

- В руководстве для преподавателей *Instructor's Manual* даны указания о том, как ис пользовать эту книгу в ходе преподавания учебного курса, посвященного базам данных. В нем приведен ряд примечаний, советов и предложений по каждой час ти, главе и приложению, а также другой вспомогательный материал.
- Ответы к упражнениям (включены в руководство *Instructor's Manual).*
- Слайды в формате PowerPoint для демонстрации в ходе лекций.

Оригиналы следующих дополнений доступны всем читателям этой книги по адресу tp: //www.aw.com/cssupport (а перевод представлен в данной книге).

- *Приложение Г, где* представлен материал по структурам хранения и методам доступа.
- Ответы к отдельным упражнениям (приложение Д).

КАК ЧИТАТЬ ЭТУ КНИГУ

В целом, книга рассчитана на последовательное чтение, но можно пропустить по-следние главы и последние разделы внутри глав по своему усмотрению. Ниже приведен рекомендуемый план первого чтения.

- Бегло прочитать главы 1 и 2.
- Очень внимательно изучить главы 3 и 4 (возможно, кроме разделов 4.6 и 4.7).
- Бегло прочитать главу 5.
- Внимательно прочитать главы 6, 7, 9 и 10, но пропустить главу 8 (возможно, кро ме раздела 8.6, посвященного языку SQL).
- Бегло прочитать главу 11.
- Внимательно прочитать главы 12 и 14[1], но пропустить главу 13.
- Внимательно прочитать главы 15 и 16 (возможно, кроме раздела 15.6 с описанием двухфазной фиксации).
- Прочитать следующие главы выборочно (но последовательно), по своему усмот рению.

Каждая глава начинается с введения и оканчивается заключением (резюме). Кроме того, в большинство глав включены упражнения. Ответы к некоторым упражнениям включены в приложение Д. Рекомендуется просматривать ответы к упражнениям, так как в них часто содержится дополнительная полезная информация по теме конкретной главы. Кроме того, в большинстве глав приведены обширные списки литературы, которые чаще всего дополнены комментариями. Такая структура книги позволяет усваивать мате-риал на нескольких уровнях, поскольку наиболее важные понятия и результаты приведены в основном тексте, а дополнительные вопросы и более сложные аспекты исследований вынесены в соответствующие упражнения, ответы к ним и комментарии к литературе.

Примечание. Ссылки обозначаются в тексте двойным номером в квадратных скобках. Например, ссылка [3.1] означает первый пункт в списке литературы к главе 3, а именно статью Э.Ф. Кодда, опубликованную в феврале 1982 года в журнале CACM, том 25, № 2. (Объяснение сокращений, которые используются в ссылках, например, "CACM", можно найти в приложении В.)

---

[1] При желании можно приступить к изучению главы 14 еще при первом чтении, сразу после главы 4.

СРАВНЕНИЕ С ПРЕДЫДУЩИМИ ИЗДАНИЯМИ

Ниже перечислены главные отличия настоящего издания от непосредственно предшествующего ему издания.

- **Часть** I. В главах 1-4 рассматриваются примерно те же темы, что и в главах 1-4 седьмого издания, но они были в значительной степени пересмотрены на уровне изложения конкретных сведений. В частности, в главе 4, в которой дано введение в SQL, вместо прежнего стандарта рассматривается современный стандарт SQL: 1999, впрочем, как и везде в данной книге, где речь идет о языке SQL. (Сам этот факт стал причиной существенного пересмотра больше чем половины глав седьмого издания.)

  Примечание. По мере необходимости упоминаются также средства, которые, по всей вероятности, будут включены в следующую версию стандарта, получившую условное название SQL:2003.

- **Часть II.** Главы 5-10, посвященные описанию реляционной модели, представляют собой полностью переработанные, значительно расширенные и существенно улучшенные версии глав 5-9 седьмого издания. В частности, сведения о типах (известных также как домены) были вынесены в отдельную главу (глава 5), а материал, который относится к проблеме целостности (глава 9), был заново продуман и полностью переработан. Автор спешит добавить, что изменения в этих главах являются следствием не пересмотра основополагающих концепций, а скорее пересмотра самого подхода, выбранного для их представления, на основании практического опыта преподавания этого материала для широкой аудитории.
  Примечание. В связи с этим необходимо привести некоторые пояснения. В предыдущих изданиях данной книги в качестве основы изложения реляционных концепций использовался язык SQL в надежде на то, что студентам будет проще вначале освоить конкретный материал, а после этого приступать к изучению абстрактных понятий. Но, к сожалению, "пропасть" между языком SQL и реляционной моделью продолжает расти, поэтому в конечном итоге автор пришел к выводу, что дальнейшее применение языка SQL для описания реляционной модели способно лишь ввести читателей в заблуждение. К сожалению, теперь язык SQL настолько далек от того, чтобы быть истинным воплощением реляционных принципов (он страдает от слишком многих недоделок и переделок), что автор выражает свое искреннее пожелание вообще не вступать в обсуждение этой темы! Тем не менее, язык SQL, безусловно, остается важным с практической точки зрения. Поэтому любой профессионал в области баз данных должен иметь определенное представление об этом языке, кроме того, его нельзя полностью игнорировать в книге такого характера. Поэтому автор принял следующее общее решение: во-первых, включить главу по основам SQL в часть I этой книги, во-вторых, по мере необходимости вводить в главы других частей отдельные разделы с описанием тех средств SQL, которые относятся к теме рассматриваемой главы. Благодаря этому в данной книге удалось привести исчерпывающее описание тематики SQL (которое действительно занимает очень большой объем), но это описание дано в таком контексте, который автор считает наиболее приемлемым.

- **Часть III.** Главы 10-13 этой части представляют собой в основном откорректиро ванные версии глав 9-12 из предыдущего, седьмого издания. Но на уровне изло жения конкретных сведений в них внесены существенные изменения.
  Примечание. И здесь необходимо дать некоторые дополнительные пояснения. В отдельных рецензиях к предыдущим изданиям были сделаны замечания, что вопросы проектирования базы данных в них рассматриваются слишком поздно. Но, по мнению автора, студенты смогут должным образом подготовиться к реше нию задач проектирования баз данных или полностью оценить важность проблем проектирования только после того, как будут иметь полное представление о том, что такое базы данных и как они используются; иными словами, автор считает важным уделить определенное внимание реляционной модели и связанным с ней вопросам и только после этого знакомить студентов с проблемами проектирова ния. Поэтому, по мнению автора, материал части III находится в должном месте. (Говоря об этом, следует признать, что многие преподаватели предпочитают из лагать сведения о сущностях и связях гораздо раньше. С учетом этого автор попы тался сделать главу 14 более или менее автономной, с тем чтобы преподаватели могли ввести ее в курс лекций, скажем, сразу же после главы 4.)
- **Часть IV.** Две главы этой части, 15 и 16, представляют собой полностью пересмот ренные, расширенные и дополненные версии глав 14 и 15 из седьмого издания. В частности, теперь глава 16 включает тщательный анализ так называемых *свойств ACID* транзакций (а также некоторые довольно неожиданные выводы по этой теме).
- **Часть V**. Глава 20 о наследовании типов и глава 23 о хронологических базах дан ных были полностью пересмотрены с учетом новейших исследований и разрабо ток в этих областях. Изменения, которые произошли в других главах, в основном представляют собой их корректировку, хотя внесены существенные уточнения в те пояснения и примеры, которые приведены в этих главах, а также повсеместно добавлен новый материал.
- **Часть VI.** Главы 25 и 26 представляют собой улучшенные и дополненные версии глав 24 и 25 из седьмого издания. Глава 27 является новой.

Наконец, новым также является приложение А, а приложения Б и В представляют собой, соответственно, пересмотренные версии приложений А и В из седьмого издания (материал из приложения Б седьмого издания включен в основную часть восьмого). Приложение Г — это существенно пересмотренная версия приложения А из шестого издания. Ответы к избранным упражнениям, которые в седьмом издании были приведены в соответствующих главах, теперь вынесены в отдельное приложение Д.

ОТЛИЧИТЕЛЬНЫЕ ОСОБЕННОСТИ ДАННОЙ КНИГИ

Каждая представленная на рынке книга по базам данных характеризуется своими собственными преимуществами и недостатками, и у каждого автора есть свой любимый конек. Одни сосредотачиваются на проблемах управления транзакциями, другие уделяют основное внимание моделям "сущность—связь", третьи рассматривают все эту проблематику через призму языка SQL, четвертые принимают чисто "объектную" точку зрения, пятые рассматривают всю эту область знаний исключительно на основе определенного

коммерческого программного продукта и т.д. И, безусловно, автор данной книги не является исключением из этого правила — у него также есть любимый конек. Условно это можно было бы назвать *основаниями науки.* Автор сохраняет твердое убеждение в том, что вначале необходимо заложить прочные основания любой науки и полностью их освоить, прежде чем пытаться что-то создавать на этом основании. Именно эта убежденность автора позволяет объяснить, почему в настоящей книге такое внимание уделено описанию реляционной модели; в частности, это позволяет понять, почему такой большой объем имеет часть II (наиболее важная часть книги), где автор излагает свое понимание реляционной модели настолько тщательно, насколько он на это способен. Автора интересуют фундаментальные знания, а не преходящие увлечения и модные направления. Технические и программные продукты постоянно изменяются, а принципы остаются.

В связи с этим автор хотел бы привлечь внимание читателя к тому, что в данной книге нескольким важным ("фундаментальным") темам посвящены целые главы (а в одном случае отдельное приложение), и в этом настоящая книга занимает совершенно особое положение среди других книг по данной тематике. Ниже перечислены основные указанные темы.

- Типы.
- Целостность.
- Представления.
- Отсутствующая информация.
- Наследование.
- Хронологические базы данных.
- Модель TransRelational™.

Развивая мысль о важности оснований науки, автор должен признать, что общая направленность этой книги с годами менялась. Первые несколько изданий по своему характеру в основном описывали текущее состояние дел; в них содержалось описание данной области в том виде, в каком она фактически реализовалась, со всеми своими сильными и слабыми сторонами. В отличие от этого, последующие издания становились все более предписывающими; в них говорилось о том, какой должна быть эта область и по какому пути она должна развиваться в будущем, если мы хотим, чтобы все в ней обстояло благополучно. А что касается настоящего издания, то в нем стремление определить правильное направление развития стало доминирующим (поэтому перед вами — не просто учебник, а книга с определенным подтекстом!). Тем не менее, добиться правильного положения дел можно лишь при условии полного понимания, в чем состоит это "правильное положение дел", поэтому автор попытался сделать все от него зависящее, чтобы это новое издание могло способствовать успеху такого важного начинания.

С этим связана еще одна важная мысль: как может быть известно читателю, автор недавно опубликовал вместе со своим коллегой Хью Дарвеном еще одну "предписывающую" книгу, *Foundation for Future Database Systems: The Third Manifesto* (в настоящей книге она указана в списке литературы как [3.3])[2]. Эта книга, сокращенно называемая ее авторами

[2] Кроме того, Третьему Манифесту посвящен Web-узел http://www.thethirc3manifesto.com. Большой объем дополнительного материала можно также найти по адресу http: / /www. dbdebunk. com.

Третьим Манифестом, или просто *Манифестом,* содержит подробные технические предложения по созданию будущих систем баз данных на основе реляционной модели; она явилась результатом многолетнего опыта преподавания и размышления на эти темы, который в дальнейшем был обобщен и изложен самим Хью и автором данной книги. Поэтому не удивительно, что идеи *Манифеста* присутствуют и в настоящей книге. Но не следует думать, что без изучения *Манифеста* нельзя приступать к чтению данной книги — это не так; но *Манифест* непосредственно касается многих вопросов, изложенных в этой книге, и в нем часто можно найти важную дополнительную информацию.

Примечание. В [3.3] в иллюстративных целях решено использовать язык Tutorial D, и то же самое принято в данной книге. Язык Tutorial D разрабатывался таким образом, чтобы его синтаксис и семантика в основном не требовали дополнительных пояснений (сам этот язык можно неформально охарактеризовать как "подобный Паскалю"). Но когда отдельные его средства используются впервые, дается их описание, если в этом есть необходимость.

ЗАКЛЮЧИТЕЛЬНОЕ ЗАМЕЧАНИЕ

В завершение своих вступительных замечаний, автор хотел бы привести следующую немного отредактированную выдержку из собственного предисловия Бертрана Рассела (Bertrand Russell) к его словарю *The Bertrand Russell Dictionary of Mind, Matter and Morals* (ed. Lester E. Denonn) — Citadel Press, 1993 (любезное разрешение на ее перепечатку получено).

> *Меня обвиняли в привычке менять свои суждения... Но разве мог бы физик, работающий с 1900 года, например, похвастаться в середине двадцатого века тем, что его суждения не изменились за последние полстолетия?... Та философия, которую я ценю и которой стараюсь следовать, научна в том смысле, что мы должны всегда стремиться получить неопровержимые знания, но новые открытия могут выявить прежние ошибки, неизбежные для любого беспристрастного разума. Когда бы и что бы я ни говорил, сейчас или в прошлом, я никогда не утверждал, что это — окончательная истина. Я утверждаю лишь то, что в свое время высказанное мной мнение было вполне обоснованным... Я был бы очень удивлен, если бы дальнейшие исследования не показали, что его необходимо пересмотреть. К тому же я никогда не высказывал свое мнение как окончательный вердикт, а просто подчеркивал, что это — лучшее, что я мог сделать в то время для достижения ясного и точного понимания. Моей целью была прежде всего полная ясность во всем.*

Если читатели предыдущих изданий будут изучать настоящее, восьмое издание, то обнаружат, что его автор также меняет свое мнение по многим вопросам (и, несомненно, будет продолжать это делать). И он надеется, что приведенная выше цитата может служить достаточным оправданием такого положения дел. Автор во всем разделяет взгляды Бертрана Рассела на то, что представляет собой любая область научных исследований, но признает, что просто не мог бы выразить эти взгляды более красноречиво.

БЛАГОДАРНОСТИ

Хочу еще раз исполнить свой приятный долг и поблагодарить всех, кто прямо или косвенно принимал участие в работе над этой книгой.

- Прежде всего, я должен поблагодарить моих друзей Дэвида Макговерна (David McGoveran) и Ника Тиндола (Nick Tindall) за их важный вклад в подготовку настоящего издания; Дэвид подготовил первый вариант главы 22 по поддержке принятия решений, а Ник — первый вариант главы 27 по XML. К тому же я очень признателен моему другу и коллеге Хью Дарвену (Hugh Darwen), который оказал мне большую помощь (в разных формах) в работе со всеми теми частями рукописи, которые касались языка SQL. Наградж Алур (Nagraj Alur) и Фабиан Паскаль (Fabian Pascal) предоставили мне большой объем технического вспомогательного материала. Особую благодарность я хочу выразить Стиву Тарену (Steve Tarin) за то, что он изобрел технологию, описанную в приложении А, и оказал мне значительную помощь в достижении ее полного понимания.
- Кроме того, текст книги был заметно улучшен благодаря замечаниям слушателей семинаров, которые я веду на протяжении последних нескольких лет. Кроме того, на нее чрезвычайно положительное влияние оказали комментарии многих друзей и рецензентов, а также мои дискуссии с ними. В их числе следует прежде всего назвать Хью Дарвена (Hugh Darwen), компания IBM; Ги де Тре (Guy de Trë), Гентский Университет; Карла Экберга (Carl Eckberg), Университет штата Сан-Диего; Ченга Хсу (Cheng Hsu), Политехнический институт Ренселлера; Абдул-Рахмана Итани (Abdul-Rahman Itani), Университет Мичигана-Дирборна; Виджая Канабара (Vijay Kanabar), Бостонский Университет; Брюса О. Ларсена (Bruce O. Larsen), Технологический институт Стевенса; Дэвида Ливингстона (David Livingstone), Нортумберлендский университет в Ньюкасле; Дэвида Макговерена (David McGoveran), компания Alternative Technologies; Стива Миллера (Steve Miller), IBM; Фабиана Паскаля (Fabian Pascal), независимая консультационная компания; Мартина К. Соломона (Martin K. Solomon), Флоридский Атлантический университет; Стива Тарена (Steve Tarin), компания Required Technologies; и Ника Тиндола (Nick Tindall), IBM. Каждый из них просмотрел, по крайней мере, часть рукописи этого издания, предоставил технический материал или как-то иначе помог мне найти ответы на многие технические вопросы, поэтому я всем им очень благодарен.
- Я хотел бы также поблагодарить мою жену Линди (Lindy) за то, что она снова участвовала в оформлении обложки книги, а в течение многих лет оказывала мне помощь при подготовке этого и других проектов, связанных с базами данных.
- И наконец, как всегда, выражаю свою признательность всем сотрудникам издательства Addison-Wesley, особенно Майте Суарес-Ривас (Maite Suarez-Rivas) и Кэтрин Харутуниан (Katherine Harutunian) за их поощрение и поддержку во время работы над проектом, а также моему редактору Элизабет Беллер (Elisabeth Beller) за ее безукоризненную работу.

К. Дж. Дейт ,
Хилдсбург, Калифорния
2003

ЖДЕМ ВАШИХ ОТЗЫВОВ!

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам обычное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг. Наши координаты:

E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com

Адреса для писем из:

России: 115419, Москва, а/я 783
Украины: 03150, Киев, а/я 152

ЧАСТЬ I

ОСНОВНЫЕ ПОНЯТИЯ

Часть I состоит из четырех вводных глав.

- В главе 1 определены основные понятия. В ней описано, что такое базы данных и
  зачем они нужны. Кроме того, в ней кратко излагаются различия между реляци
  онными и другими типами баз данных.
- В главе 2 в общих чертах представлена архитектура баз данных — так называемая
  архитектура ANSI/SPARC. На основе этого материала строятся все последующие
  главы книги.
- В главе 3 приведен обзор реляционных систем, который позволяет читателю по
  степенно подготовиться к более подробному обсуждению тех же вопросов в час
  ти II и последующих частях данной книги. Здесь также рассматривается реальный
  пример — база данных поставщиков и деталей.
- В главе 4 дано краткое введение в стандартный реляционный язык SQL (точнее,
  SQL: 1999).

ГЛАВА 1

Базы данных и управление ими

1.1. ВВОДНЫЙ ПРИМЕР

Система баз данных — это, по сути, не что иное, как *компьютеризированная система хранения однотипных записей.* Саму же **базу данных** можно рассматривать как подобие электронной картотеки, т.е. хранилище или контейнер для некоторого набора файлов данных, занесенных в компьютер. Пользователям этой системы предоставляется возможность выполнять (или передавать системе запросы на выполнение) множество различных операций над такими файлами, например:

- добавлять новые пустые файлы в базу данных;
- вставлять новые данные в существующие файлы;
- получать данные из существующих файлов;
- удалять данные из существующих файлов;
- изменять данные в существующих файлах;
- удалять существующие файлы из базы данных.

В качестве иллюстрации в табл. 1.1 приведена небольшая база данных, состоящая всего из одного файла под названием CELLAR (винный погреб). В этом файле хранятся данные о содержимом винного погреба. На рис. 1.1 показан пример выполнения операции

выборки и данные, возвращаемые с помощью этой операции. (Операции над базами данных, имена файлов и т.п. в этой книге для наглядности пишутся прописными буквами, хотя на практике часто удобнее использовать строчные. В большинстве СУБД допускаются оба варианта.) На рис. 1.2 приведены примеры операций **вставки, удаления** и **обновления** для базы данных винного погреба. Эти примеры почти не требуют пояснений. В других главах этой книги приведены примеры добавления и удаления файлов.

Таблица 1.1. Содержимое файла CELLAR

BIN#	WINE	PRODUCER	YEAR	BOTTLES	READY
2	Chardonnay	Buena Vista	2001	1	2003
3	Chardonnay	Geyser Peak	2001	5	2003
6	Chardonnay	Simi	2000	4	2002
12	Joh. Riesling	Jekel	2002	1	2003
21	Fumé Blanc	Ch. St. Jean	2001	4	2003
22	Fumé Blanc	Robt. Mondavi	2000	2	2002
30	Gewurztraminer	Ch. St. Jean	2002	3	2003
43	Cab. Sauvignon	Windsor	1995	12	2004
45	Cab. Sauvignon	Geyser Peak	1998	12	2006
48	Cab. Sauvignon	Robt. Mondavi	1997	12	2008
50	Pinot Noir	Gary Farrell	2000	3	2003
51	Pinot Noir	Fetzer	1997	3	2004
52	Pinot Noir	Dehlinger	1999	2	2002
58	Merlot	Clos du Bois	1998	9	2004
64	Zinfandel	Cline	1998	9	2007
72	Zinfandel	Rafanelli	1999	2	2007

Выборка:

```
SELECT WINE, BIN#, PRODUCER
FROM CELLAR
WHERE READY = 2004 ;
```

Результат (в том виде, в каком он отображается, например, на экране дисплея):

WINE	BIN#	PRODUCER
Cab. Sauvignon	43	Windsor
Pinot Noir	51	Fetzer
Merlot	58	Clos du Bois

Рис. 1.1. Пример выборки информации из базы данных

На основании данных, приведенных в табл. 1.1 и на рис. 1.1 и 1.2, можно сделать приведенные ниже выводы.

1. Примеры запросов на выборку, вставку, удаление и обновление, приведенных на рис. 1.1 и 1.2, представлены с помощью операторов SELECT, INSERT, DELETE и UPDATE специального языка баз данных, называемого **SQL.** Язык SQL был первоначально разработан компанией IBM, а в настоящее время поддерживается большинством коммерческих СУБД, представленных на рынке, и является официальным

стандартом языка для работы с реляционными базами данных. Поэтому в главе 4 представлен общий обзор стандарта SQL, а в большинстве последующих глав включен раздел "Средства языка SQL" с описанием тех аспектов стандарта, которые относятся к теме соответствующей главы.

Название SQL вначале было аббревиатурой, образованной от Structured Query Language (язык структурированных запросов), и его было принято произносить как "сиквел". Сейчас, когда язык стал стандартом, SQL— это уже не аббревиатура, а обычное название, которое произносится как "эс-кью-эл". В дальнейшем в этой книге предполагается именно такой вариант произношения.

Вставка новых данных:

```
INSERT
INTO   CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )
VALUES ( 53, 'Pinot Noir', 'Saintsbury', 2001, 6, 2005 ) ;
```

Удаление существующих данных:

```
DELETE
FROM   CELLAR
WHERE  BIN# = 2 ;
```

Модификация существующих данных:

```
UPDATE CELLAR
SET    BOTTLES = 4
WHERE  BIN# = 3 ;
```

Рис. 1.2. Примеры операций вставки, удаления и обновления

2. Заметим, что в языке SQL для операции обновления используется ключевое слово UPDATE, обозначающее именно операцию модификации данных (см. рис. 1.2). Это в некоторых случаях может привести к недоразумениям, поскольку термин "обновить" (update) используется также по отношению к операциям вставки (INSERT), удаления (DELETE) и обновления (UPDATE) в целом, как к группе операций. В данной книге мы будем различать эти два понятия, используя строчные буквы при записи общего понятия и прописные буквы, когда будет подразумеваться именно операция обновления UPDATE.

   Необходимо подчеркнуть, что в специальной литературе термины *оператор* и *операция* часто используются как взаимозаменяемые. Но, строго говоря, между ними есть различие (операция выполняется после вызова оператора); тем не менее, в неформальных дискуссиях эти термины часто употребляются как синонимы.

3. В языке SQL компьютерные файлы, такие как CELLAR в табл. 11, называются таблицами (по очевидным причинам). Строки (row) подобных таблиц соответствуют записям файла, а столбцы (column) можно рассматривать как поля этих записей. В дальнейшем термины *запись* и *поле* будут использоваться тогда, когда речь будет идти о базах данных вообще (в основном, в первых двух главах). Термины *таблица, строка* и *столбец* будут использоваться при рассмотрении реляционных систем, а когда мы перейдем к более формальным рассуждениям в главе 3 и следующих частях книги, то встретимся еще с одним набором терминов: *отношения, кортежи* и *атрибуты,* которые будут применяться вместо *таблиц, строк* и *столбцов.*

4. Для простоты в примере таблицы CELLAR по умолчанию предполагается, что в столбцах WINE и PRODUCER содержатся строковые данные, а во всех остальных столбцах — целочисленные данные. Но, как правило, столбцы могут содержать данные произвольной сложности. В частности, таблица CELLAR может быть до полнена для включения в нее следующих столбцов (а также столбцов многих дру гих типов).
   - LABEL. Фотография этикетки винной бутылки.
   - REVIEW. Текст отзыва о качестве вина, полученный из определенного винного магазина.
   - MAP. Карта той местности, где было изготовлено это вино.
   - NOTES. Звукозапись, содержащая комментарии о вкусе вина.

   По очевидным причинам, в большинстве примеров этой книги используются только очень простые типы данных, но следует учитывать, что всегда существует возможность применять гораздо более сложные структуры данных. Вопросы, касающиеся **типов данных** (data type) столбцов, более подробно будут обсуждаться в главах 5, 6, 26 и 27.
5. Столбец BIN# является **первичным ключом** (primary key) таблицы CELLAR (подра зумевается, что любые две строки этой таблицы никогда не будут содержать одно и то же значение поля BIN#). Для выделения в таблицах столбцов первичного ключа мы обычно будем использовать двойное подчеркивание (как в табл. 1.1).

И еще одно, последнее примечание в данном разделе: понимание материала этой и следующей главы имеет решающее значение для правильного восприятия средств и возможностей современных систем баз данных. Однако нельзя не признать, что материал этих глав кое-где носит несколько абстрактный и довольно сухой характер. Дополнительная сложность состоит в том, что здесь рассматривается много понятий и терминов, которые могут быть еще неизвестны читателю. Следующая часть книги (особенно главы 3 и 4) менее абстрактна, поэтому, возможно, она будет восприниматься лучше. Так что, может быть, стоит сначала прочитать первые две главы лишь бегло, а позже внимательно перечитывать их по мере детального ознакомления с темами, которые непосредственно связаны с материалом этих двух глав.

1.2. ОБЩЕЕ ОПРЕДЕЛЕНИЕ СИСТЕМЫ БАЗ ДАННЫХ

Как отмечалось выше, система баз данных — это компьютеризированная система хранения записей, т.е. компьютеризированная система, основное назначение которой — хранить информацию, предоставляя пользователям средства ее извлечения и модификации. К информации может относиться все, что заслуживает внимания отдельного пользователя или организации, использующей систему, иначе говоря, все необходимое для текущей работы данного пользователя или предприятия.

Следует отметить, что термины *данные* и *информация* трактуются в этой книге как синонимы. Некоторые авторы предпочитают различать эти два понятия, используя термин *данные* для ссылки на значения, которые реально сохранены в базе данных, а термин *информация* — для указания на то, что означают эти данные с точки зрения пользователя. Разница, безусловно, существенная, но предпочтительнее сделать ее более определенной там, где это уместно, вместо того, чтобы полагаться на различные толкования двух по сути одинаковых терминов.

На рис. 1.3 показана весьма упрощенная схема системы баз данных. Здесь отражено четыре главных компонента системы: **данные, аппаратное обеспечение, программное обеспечение** и **пользователи.** Каждый из этих компонентов кратко рассматривается ниже. Далее они будут обсуждаться гораздо подробнее (за исключением аппаратного обеспечения, поскольку основная часть аспектов его использования выходит за рамки этой книги).

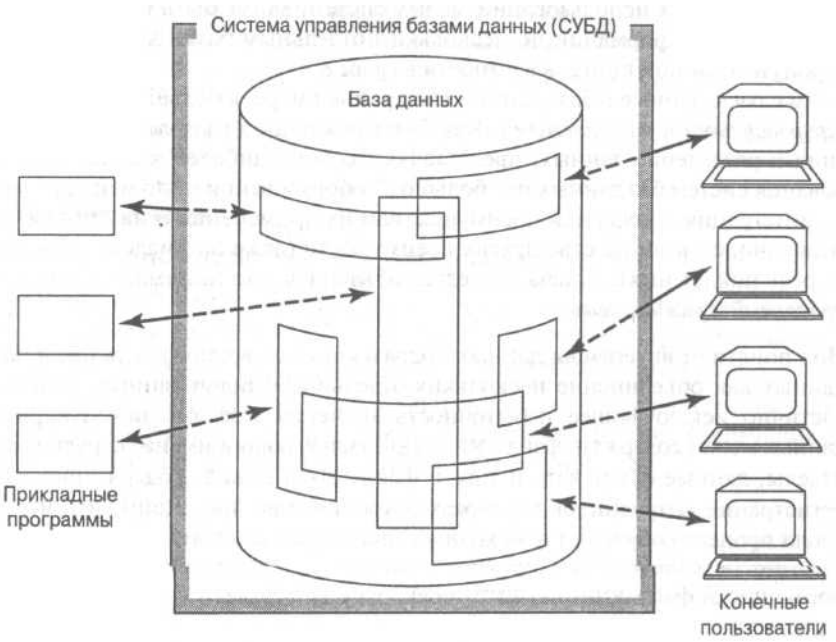


Рис. 1-3. Упрощенная схема системы баз данных

Данные

Системы с базами данных применяются как на самых малых компьютерах, так и на крупнейших мэйнфреймах. Несомненно, предоставляемые каждой конкретной системой средства в значительной мере зависят от мощности и возможностей базовой машины. В частности, на больших вычислительных машинах применяются в основном многопользовательские системы ("большие системы"), а на малых компьютерах, как правило, — однопользовательские системы ("малые системы"). Однопользовательская система (single-user system) — это система, в которой к базе данных может получить доступ одновременно только один пользователь, а многопользовательская система (multi-user system) — это такая система, в которой к базе данных могут получить доступ сразу несколько пользователей. Как показано на рис. 1.3, исходя из соображений общности, мы обычно будем изучать именно второй вид систем, хотя с точки зрения пользователей между этими системами фактически не существует большого различия, поскольку основная цель многопользовательских систем состоит в том, чтобы позволить каждому отдельному пользователю работать с ней так, как если бы он работал с *однопользовательской* системой. Различия между этими двумя видами систем проявляются

в их внутренней структуре и потому практически не видны конечному пользователю (о чем речь пойдет в части IV этой книги, особенно в главе 16).

Обычно для упрощения предполагается, что все данные в системе хранятся в одной базе данных. Мы также будем придерживаться этого предположения, поскольку количество применяемых баз данных несущественно для всех дальнейших рассуждений. Однако на практике, даже при использовании малых систем, могут быть серьезные основания для распределения информации по нескольким отдельным базам данных. Эта тема еще будет затронута в данной книге, в частности в главе 2.

В общем случае данные в базе данных (по крайней мере, в больших системах) являются *интегрированными* и *разделяемыми.* Как будет показано в разделе 1.4, эти два аспекта, интеграция и разделение данных, представляют собой наиболее важные преимущества использования систем баз данных на "большом" оборудовании и, по меньшей мере, один из них— интеграция — является преимуществом их применения и на "малом" оборудовании. Конечно, есть множество других преимуществ (даже на "малом" оборудовании), но о них речь пойдет ниже. Сначала следует объяснить, что понимается под терминами *интегрированный* и *разделяемый.*

Под понятием интеграции данных подразумевается возможность представить базу данных как объединение нескольких отдельных файлов данных, полностью или частично исключающее избыточность хранения информации. Например, база данных может содержать файл EMPLOYEE, включающий имена сотрудников, адреса, отделы, данные о зарплате и т.д., и файл ENROLLMENT, содержащий сведения о регистрации сотрудников на курсах обучения (рис. 1.4). Допустим, что для контроля процесса обучения необходимо знать отдел каждого зачисленного на курсы сотрудника. Совершенно очевидно, что нет необходимости включать такую информацию в файл ENROLLMENT, поскольку ее всегда можно получить из файла EMPLOYEE.

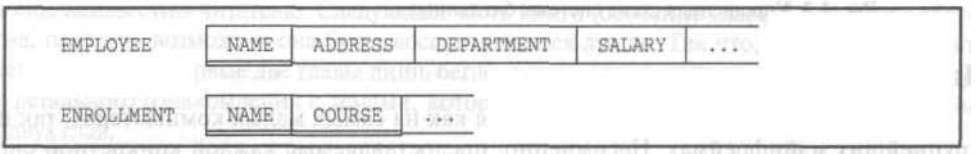


Рис. 1.4. Файлы EMPLOYEE И ENROLLMENT

- Под понятием разделяемое™ данных подразумевается возможность использования несколькими различными пользователями отдельных элементов, хранимых в базе данных. Имеется в виду, что каждый из пользователей сможет получить доступ к одним и тем же данным, возможно, даже одновременно *(параллельный доступ).* Такое разделение данных, с параллельным или последовательным доступом, частично является следствием того факта, что база данных имеет интегрированную структуру. В примере, приведенном на рис. 1.4, информация об отделе в файле EMPLOYEE может совместно использоваться сотрудниками отдела кадров и отдела обучения. (Если база данных не является разделяемой, то ее иногда называют *личной базой* или *базой данных специального назначения.)*

Одним из следствий упомянутых выше характеристик базы данных (интеграции и разделяемости) является то, что каждый конкретный пользователь обычно имеет дело лишь с небольшой частью всей базы данных, причем обрабатываемые различными пользователями части могут произвольным образом перекрываться. Иначе говоря, каждая база данных воспринимается ее различными пользователями по-разному. Фактически, даже те два пользователя базы данных, которые работают с одними и теми же частями базы данных, могут иметь значительно отличающиеся представления о них. Более подробное обсуждение этого вопроса приводится далее, в разделе 1.5 и в следующих главах (в частности, в главе 10).

В разделе 1.3 мы продолжим обсуждение свойств элементов данных, хранимых в системе баз данных.

Аппаратное обеспечение

К аппаратному обеспечению системы относится следующее:

- тома вторичной (внешней) памяти (обычно это магнитные диски), используемые для хранения информации, а также соответствующие устройства ввода—вывода (дисководы и т.п.), контроллеры устройств, каналы ввода—вывода и т.д.;
- аппаратный процессор (или процессоры) вместе с оперативной (первичной) памятью, предназначенные для поддержки работы программного обеспечения системы баз данных (подробности приведены в следующем подразделе).

Аппаратной части системы в данной книге уделяется мало внимания. Во-первых, эти вопросы составляют достаточно обширную тему, которую нужно рассматривать отдельно. Во-вторых, проблемы, которые присущи в этой области, не являются специфическими исключительно для систем баз данных. И в-третьих, эти проблемы достаточно подробно освещены в других источниках.

Программное обеспечение

Между собственно физической базой данных (т.е. данными, которые реально хранятся на компьютере) и пользователями системы располагается уровень программного обеспечения, который можно называть по-разному: **диспетчер базы данных** (database manager), **сервер базы данных** (database server) или, что более привычно, **система управления базами данных, СУБД** (DataBase Management System — DBMS). Все запросы пользователей на получение доступа к базе данных обрабатываются СУБД. Все имеющиеся средства добавления файлов (или таблиц), выборки и обновления данных в этих файлах или таблицах также предоставляет СУБД. Основная задача СУБД — дать пользователю базы данных возможность работать с ней, *не вникая во все подробности работы на уровне аппаратного обеспечения*. (Пользователь СУБД более отстранен от этих подробностей, чем прикладной программист, применяющий языковую среду программирования.) Иными словами, СУБД позволяет конечному пользователю рассматривать базу данных как объект более высокого уровня по сравнению с аппаратным обеспечением, а также предоставляет в его распоряжение набор операций, выражаемых в терминах языка высокого уровня (например, набор операций, которые можно выполнять **с** помощью языка SQL, упомянутого выше, в разделе 1.1). Далее в книге будут подробно описаны обе указанные функции СУБД.

Необходимо отметить еще две описанные ниже особенности.

- СУБД— это наиболее важный, но не единственный программный компонент системы. В числе других компонентов можно назвать утилиты, средства разра ботки приложений, средства проектирования, генераторы отчетов и *диспетчер транзакций* (transaction manager), или *диспетчер обработки транзакций* (transaction processing monitor — TP monitor). Эти компоненты описаны более подробно в гла вах 2, 3, особенно в главах 15 и 16.
- Термин *СУБД* также часто используется в отношении конкретных программных продуктов конкретных изготовителей, например, таких как *DB2 Universal Database* компании IBM. Иногда в тех случаях, когда конкретная копия подобного продукта устанавливается для работы на определенном компьютере, используется термин *экземпляр.* Безусловно, необходимо строго различать эти два понятия.

  Следует отметить, что термин *база данных* часто используется даже тогда, когда на самом деле подразумевается *СУБД* (в одном из уже упомянутых толкований). Вот типичный пример: "База данных изготовителя *X* превосходит по производитель- ности базу данных изготовителя К в два раза". Такое небрежное обращение с тер- минами предосудительно; тем не менее, оно очень широко распространено. (Проблема, естественно, заключается в том, что если СУБД называют базой дан- ных, как же тогда называть саму базу данных?) Читатель, будь внимателен!

Пользователи

Пользователей можно разделить на три большие и отчасти перекрывающиеся группы.

- Первая группа — **прикладные программисты,** которые отвечают за написание при кладных программ, использующих базу данных. Для этих целей применимы такие языки, как COBOL, PL/I, C++, Java или какой-нибудь высокоуровневый язык четвертого поколения (см. главу 2). Прикладные программы получают доступ к ба зе данных посредством выдачи соответствующего запроса к СУБД (обычно это не который оператор SQL). Подобные программы могут быть простыми пакетными приложениями или же **интерактивными** приложениями, предназначенными для поддержки работы конечных пользователей (см. следующий абзац). В последнем случае они предоставляют пользователям непосредственный оперативный доступ к базе данных через рабочую станцию, терминал или персональный компьютер. Большинство современных приложений относится именно к этой категории.
- Вторая группа — **конечные пользователи,** которые работают с системой баз данных в интерактивном режиме, как указано в предыдущем абзаце. Конечный пользова тель может получать доступ к базе данных, применяя одно из интерактивных при ложений, упомянутых выше, или же интерфейс, интегрированный в программное обеспечение самой СУБД. Безусловно, подобный интерфейс также поддержи вается интерактивными приложениями, но эти приложения не создаются пользователями-программистами, а являются **встроенными** в СУБД. Большин ство СУБД включает по крайней мере одно такое встроенное приложение, а именно — **процессор языка запросов,** позволяющий пользователю в диалоговом режиме вводить *запросы* к базе данных (их часто иначе называют *предложениями* или *командами),* например, с операторами SELECT или INSERT. Язык SQL представляет

собой типичный пример языка запросов к базе данных. (Общепринятый термин *язык запросов* не совсем точно отражает рассматриваемое понятие, поскольку слово *запрос* подразумевает лишь выборку информации, в то время как с помощью этого языка выполняются также операции обновления, вставки, удаления и др.)

Кроме языка запросов, в большинстве систем дополнительно предоставляются специализированные встроенные интерфейсы, в которых пользователь в явном виде не использует предложения, или команды с такими операторами, как SELECT и INSERT. Работа с базой данных осуществляется за счет выбора пользователем необходимых элементов меню или заполнения требуемых полей в предоставленных формах. Такие **некомандные** интерфейсы, основанные на меню и формах, облегчают работу с базами данных для тех, кто не имеет опыта работы с информационными технологиями (или ИТ; часто употребляется также сокращение ИС — информационные системы; эти понятия практически эквивалентны). **Командный** интерфейс, т.е. язык запросов, напротив, требует некоторого профессионального опыта работы с ИТ (но, безусловно, не такого большого, какой необходим для написания прикладных программ на языке программирования, подобном COBOL). Однако командный интерфейс более гибок, чем некомандный, к тому же языки запросов обычно включают определенные функции, отсутствующие в интерфейсах, основанных на использовании меню или форм.

- Третья группа (на рис. 1.3 не показана) — **администраторы *базы данных,*** *или АБД.* Обсуждение функций администраторов баз данных и связанных с ними (что очень важно) функций администрирования данных, отложим до раздела 1.4 и главы 2 (раздел 2.7).

На этом закончим предварительное описание основных аспектов систем баз данных и приступим к более детальному изучению соответствующих понятий.

1.3. ОБЩЕЕ ОПРЕДЕЛЕНИЕ БАЗЫ ДАННЫХ

Перманентные данные

Обычно данные в базе данных называют *перманентными* или *постоянно хранимыми* (хотя иногда на самом деле они недолго остаются таковыми!). Под словом *перманентные* (persistent) подразумеваются данные, которые отличаются от других, более изменчивых данных, таких как промежуточные результаты, входные и выходные данные, управляющие операторы, рабочие очереди, программные управляющие блоки и вообще все данные, *временные* (transient) по своей сути. Точнее говоря, можно утверждать, что данные в базе являются *перманентными,* поскольку после того как они были приняты средствами СУБД для помещения в базу, *их последующее удаление возможно лишь при использовании соответствующего явного запроса к базе данных,* но не в результате какого-либо побочного эффекта от выполнения некоторой программы. Подобный взгляд на понятие перманентности позволяет точнее определить *терминбаза данных.*

- **База данных** — это некоторый набор перманентных (постоянно хранимых) данных, используемых прикладными программными системами какого-либо предприятия.

Здесь слово *предприятие*— удобный общий термин для относительно независимой коммерческой, научной, технической или любой другой организации. Организация может состоять всего из одного человека (с небольшой частной базой данных), быть целой корпорацией или другой крупной организацией (с очень большой общей базой данных) либо представлять собой нечто среднее между этими крайними случаями. Ниже приведено несколько примеров.

1. Промышленная компания.
2. Банк.
3. Больница.
4. Университет.
5. Министерство.

Любое предприятие неизбежно использует большое количество данных, связанных с его деятельностью. Это и есть *перманентные данные,* о которых шла речь в приведенном выше определении. Среди перманентных данных упомянутых предприятий обычно встречаются данные, перечисленные ниже.

1. Данные о продукции.
2. Бухгалтерские данные.
3. Данные о пациентах.
4. Данные о студентах.
5. Данные о планируемой деятельности.

Примечание. В первых шести изданиях этой книги вместо термина *перманентные данные* использовался термин *операционные данные.* Старый термин первоначально акцентировал внимание на особом значении **оперативных,** или **производственных,** приложений баз данных, т.е. рутинных, часто выполняющихся приложений, предназначенных для поддержки повседневной работы предприятия (например, приложений для поддержки операций зачисления и списания средств на счетах в банковской системе). Для среды такого рода в последнее время используется термин **оперативная обработка транзакций** (On-Line Transaction Processing— OLTP). Однако теперь базы данных все чаще применяются и в приложениях другого рода, например, в приложениях **поддержки принятия решений** (decision support), и термин *операционные данные* для них уже не подходит. На практике сегодняшние предприятия используют две отдельные базы данных — с операционными данными и с данными для поддержки принятия решений; последнюю обычно называют **хранилищем данных** (data warehouse). В хранилищах данных часто содержится агрегированная информация (например, итоговые и средние значения), которая, в свою очередь, периодически (например, раз в сутки или раз в неделю) извлекается из операционной базы данных. Обсуждение баз данных и приложений, которые служат для поддержки принятия решений, будет продолжено в главе22.

Сущности и связи

Рассмотрим более подробно пример некоторой промышленной компании (допустим, она имеет название *KnowWare, Inc.).* Обычно подобному предприятию требуется записывать информацию об имеющихся *проектах* (Projects), используемых в этих проектах

деталях (Parts), *поставщиках* (Suppliers) деталей, *складах* (Warehouses), на которых хранятся детали, *служащих* (Employees), работающих над проектами и т.д. Проекты, детали, поставщики и т.д. представляют собой основные сущности (entity), о которых компании KnowWare, Inc. необходимо хранить информацию. В теории баз данных термин *сущность* обычно используется для обозначения любого различимого объекта, который может быть представлен в базе данных (рис. 1.5).

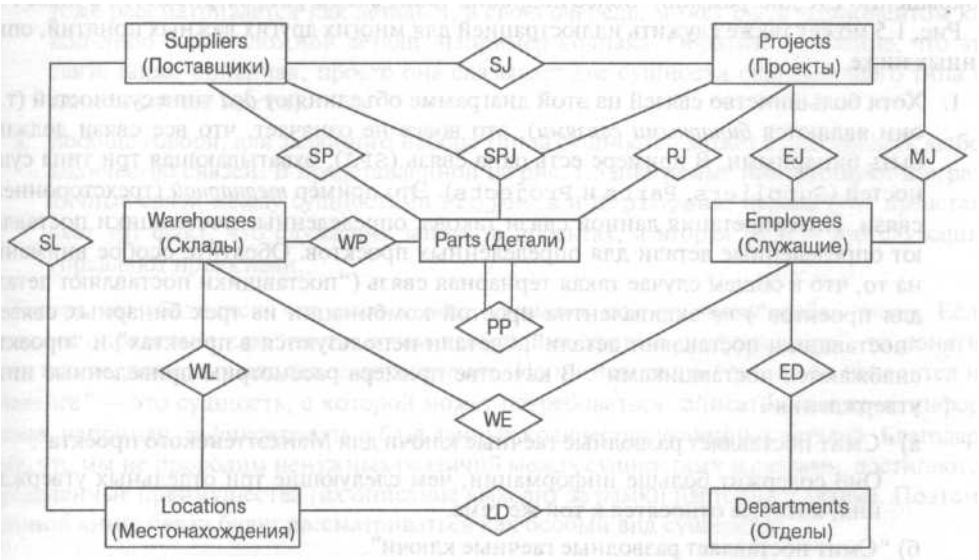


Рис. 1.5. Пример диаграммы "сущность-связь" (ER-диаграммы) для базы данных компании KnowWare, Inc.

Кроме этих основных сущностей (в данном примере это поставщики, детали и т.д.), имеются еще и **связи** (relationship) между ними, которые объединяют эти основные сущности. На рис. 1.5 связи представлены ромбами с соединительными линиями. Например, между поставщиками и деталями существует связь SP (сокращение от Shipments/Parts): каждый поставщик поставляет определенные детали, и наоборот, каждая деталь поставляется определенными поставщиками. (Точнее, каждый поставщик поставляет определенные *виды* деталей, и каждый *вид* деталей поставляется определенными поставщиками.) Аналогично, детали используются в проектах, а для реализации проектов требуются детали (связь PJ — сокращение от Parts/proJects); детали хранятся на складах, а склады хранят детали (связь WP — сокращение от Warehouses/Parts) и т.д. Обратите внимание, что эти связи — *бинарные* (или *двухсторонние),* т.е. их можно прослеживать в любом направлении. В частности, используя связь SP между поставщиками и деталями, можно ответить на следующие вопросы:

- задан поставщик, и требуется определить поставляемые им детали;
- задана деталь, и необходимо найти поставщиков, предоставляющих такую деталь.

Очень важно то, что эта связь (как и другие связи, представленные на рис. 1.5) *является такой же неотъемлемой составляющей данных предприятия, как и основные сущности.* Поэтому связи должны быть представлены в базе данных наравне с основными сущностями предметной области[1].

Следует отметить, что на рис. 1.5 приведен пример информационной структуры, которую принято называть (по очевидным причинам) **диаграммой "сущность—связь"** (сокращенно ER-диаграммой). Более подробно такие схемы рассматриваются в главе 14.

Рис. 1.5 может также служить иллюстрацией для многих других важных понятий, описанных ниже.

Хотя большинство связей на этой диаграмме объединяют *два* типа сущностей (т.е. они являются *бинарными связями),* это вовсе не означает, что все связи должны быть бинарными. В примере есть одна связь (SPJ), охватывающая три типа сущностей (Suppliers, Parts и Projects). Это пример *тернарной* (трехсторонней) связи. Интерпретация данной связи такова: определенные поставщики поставляют определенные детали для определенных проектов. Обратите особое внимание на то, что в общем случае такая тернарная связь ("поставщики поставляют детали для проектов") *не* эквивалентна простой комбинации из трех бинарных связей: "поставщики поставляют детали", "детали используются в проектах" и "проекты снабжаются поставщиками". В качестве примера рассмотрим приведенные ниже утверждения[2].

а) "Смит поставляет разводные гаечные ключи для Манхэттенского проекта".

Оно содержит больше информации, чем следующие три отдельных утвержде ния, которые относятся к той же теме.,

б) "Смит поставляет разводные гаечные ключи".

в) "Разводные гаечные ключи используются в Манхэттенском проекте".

г) "Манхэттенский проект снабжается Смитом".

Зная только утверждения *б, в* и г, мы не сможем доказать справедливость утверждения *а.* Точнее, зная, что справедливы утверждения *б, в* и г, мы можем лишь сделать заключение, что Смит поставляет разводные гаечные ключи для *какого-то* проекта (скажем, проекта J_z), что *какой-то* поставщик (скажем, поставщик S_x) поставляет разводные гаечные ключи для Манхэттенского проекта и что Смит поставляет *какую-то* деталь (скажем, деталь P_Y) для Манхэттенского проекта. Однако мы не можем точно утверждать, что поставщик S_x — это Смит, деталь P_Y — это разводной гаечный ключ, а проект J_z — это Манхэттенский проект. Ложные

---

[1] Характерной особенностью реляционных баз данных (см. раздел 1.6) является то, что основные сущности и связи между ними представлены с помощью отношений (relation) или, иными словами, с помощью таблиц, подобных показанной в табл. 1.1. Но следует учитывать, что термин *связь* (relationship), который используется в текущем разделе, и термин *отношение,* применяемый в контексте реляционных баз данных, обозначают разные понятия.

[2] Применяемый в оригинале англоязычный термин *statement* имеет два разных значения: в данном случае он указывает на утверждение, касающееся некоторого факта (которое в логике принято называть *высказыванием;* см. подраздел "Данные и модели данных" ниже в этом разделе), а в другом контексте может служить синонимом термина *command* (команда).

выводы, сделанные на основании неполной информации, называются дефектом соединения (connection trap).

2. На схеме также есть одна связь (PP), которая связывает *один* тип сущности (Parts) с самим собой. Эта связь означает, что одни детали содержат другие дета ли как собственные компоненты (так называемая связь спецификации материалов, или связь "деталь—узел"). Например, винт— это компонент шарнира, который тоже рассматривается как деталь и, в свою очередь, может быть компонентом ка кой-либо более сложной детали, например колпака. Обратите внимание, что эта связь также бинарная; просто она связывает две сущности совпадающего типа (в данном случае Parts).
3. Вообще говоря, для заданного набора типов сущностей может существовать любое количество связей. В представленной на рис. 1.5 диаграмме присутствуют две раз личные связи между сущностями Projects и Employees: первая (EJ) представ ляет тот факт, что служащие заняты в проектах, а вторая (MJ) — что служащие управляют проектами.

Теперь мы убедились, что *связь можно понимать как сущность особого типа.* Если сущность определена как "нечто, о чем необходимо хранить информацию", то понятие связи вполне подходит под такое определение. Например, связь "деталь P4 находится на складе W8" — это сущность, о которой может потребоваться записать некоторую информацию, например, зафиксировать в базе данных количество указанных деталей. Благодаря тому, что мы не проводим ненужных различий между сущностями и связями, достигаются определенные преимущества (их описание выходит за рамки настоящей главы). Поэтому в данной книге связи будут рассматриваться как особый вид сущности.

Свойства

Как было только что отмечено, сущность — это то, о чем необходимо хранить информацию. Отсюда следует, что сущности (а значит, и связи) имеют некоторые свойства (properties), соответствующие тем данным о них, которые необходимо хранить в базе. Например, поставщики имеют определенное *место расположения,* детали характеризуются *весом,* проекты — *очередностью* выполнения, закрепление служащих за проектами имеет *начальную дату* и т.д. В базе данных должны быть представлены именно эти свойства. Например, в базе данных может быть таблица S, представляющая тип сущности "поставщики", а в этой таблице может присутствовать тип поля CITY (город), представляющий свойство "место расположения".

В общем случае свойства могут быть как простыми, так и сложными, причем настолько простыми или сложными, насколько это потребуется. Например, свойство "местонахождение поставщика" — относительно простое: оно состоит только из названия города и может быть описано как простая символьная строка. В противоположность этому, сущность "склад" может иметь свойство "схема этажей" с достаточно сложной структурой, включающей архитектурный план здания, дополненный соответствующим текстовым описанием. Как отмечалось в разделе 1.1, ко времени написания этой книги лишь немногие СУБД поддерживали работу с такими сложными свойствами, как изображения с текстовым описанием. Мы еще возвратимся к данному вопросу позже в этой книге (в частности, в главах 5, 6, 26 и 27), а пока в большинстве случаев (за исключением тех,

в которых приходится явно учитывать различия между простыми и сложными свойствами) будем полагать, что все свойства являются простыми и их можно представить простыми типами данных: числами, строками, датами, отметками времени и т.п.

Данные и модели данных

Существует и другой, не менее важный, подход к трактовке данных и баз данных. Слово *данные* (data) происходит от латинского слова "дано" (напрашивается продолжение "требуется доказать"). Отсюда следует, что данные на самом деле являются заданными фактами, из которых можно логически вывести другие факты. (Получение производных фактов из заданных — это именно то, что выполняет СУБД, обслуживая запросы пользователя.) "Заданный факт", в свою очередь, соответствует тому, что в логике называется *истинным высказыванием.* Например, высказывание "Поставщик с номером S1 находится в Лондоне" вполне может оказаться истинным. *Высказыванием* в логике называется такое утверждение, которое может быть недвусмысленно определено как истинное или ложное. Например, "Вильям Шекспир написал *Гордость и предубеждение"* — это высказывание (как видно, ложное). Отсюда следует, что в действительности база данных — это **множество истинных высказываний** [1.2].

Итак, уже было отмечено, что продукты на основе языка SQL заняли доминирующее положение на рынке. Одной из причин этого является то, что они основаны на использовании формальной теории, называемой **реляционной моделью данных,** а эта теория, в свою очередь, поддерживает указанную выше интерпретацию данных и баз данных весьма просто (фактически почти тривиально). Конкретнее, реляционная модель характеризуется описанными ниже особенностями.

1. Данные представлены посредством строк в таблицах[3], и эти строки могут быть не посредственно интерпретированы как истинные высказывания. Например, строку с номером ячейки погреба (поле BIN#), равным 72 (см. табл. 1.1), можно очевид ным образом интерпретировать как следующее истинное высказывание:

   "В ячейке 72 находятся две бутылки вина Zinfandel, выпущенные компанией Rafanelli в 1999 году, которые будут готовы к употреблению в 2007 году".

2. Для обработки строк данных предоставляются операторы, которые напрямую под держивают процесс логического вывода дополнительных истинных высказываний из существующих высказываний. Например, реляционная операция *проекции* (раздел 1.6) позволяет получить из приведенного выше истинного высказывания, помимо прочих истинных высказываний, и такое:

   "Некоторые бутылки вина Zinfandel будут готовы к употреблению в 2 007 году".

   (Точнее, "Некоторые бутылки вина Zinfandel в некоторой ячейке, произведенные некоторым производителем в некотором году, будут готовы к употреблению в 2007 году".)

[3] Точнее, с помощью кортежей в отношениях, как описано в главе 3.

Однако реляционная модель — не единственная возможная модель данных. Существуют и другие модели (раздел 1.6), хотя многие из них отличаются от реляционной модели только тем, что они в определенной степени приспособлены для специальных случаев, а не строго основаны на формальной логике, в отличие от реляционной модели. Возникает вопрос: *что* же такое модель данных? Это понятие можно определить, руководствуясь [1.1] (но в ином изложении), как показано ниже.

- **Модель данных** — это абстрактное, самодостаточное, логическое определение объектов, операторов и прочих элементов, в совокупности составляющих *абстрактную машину доступа к данным,* с которой взаимодействует пользователь. Упомянутые объекты позволяют моделировать *структуру* данных, а операторы — *поведение* данных.

Используя это определение, можно эффективно разделить понятия *модели данных* и ее *реализации*.

> **Реализация** (implementation) заданной модели данных — это физическое воплощение на реальной машине компонентов абстрактной машины, которые в совокупности составляют эту модель.

Короче говоря, модель — это то, о чем пользователи должны знать, а реализация -это то, чего пользователи не должны знать.

Как следует из указанного выше, различие между моделью и ее реализацией в действительности является просто частным случаем знакомого нам различия *между логическим* и *физическим* определением данных (хотя и очень важным частным случаем). Однако, как это ни прискорбно, разработчики многих современных систем баз данных (даже систем, которые претендуют на то, чтобы называться реляционными) не проводят такого четкого различия, как требуется. Действительно, по-видимому, это весьма распространенный недостаток, состоящий в непонимании таких различий и важности их учета. И вследствие этого все чаще наблюдается расхождение между *принципами* создания баз данных (указывающими, какими должны быть системы баз данных) и *практикой* их реализации (тем, какими они являются на самом деле). В этой книге нас интересуют, в первую очередь, принципы, но честнее будет заранее предупредить читателя, что его могут поджидать сюрпризы (в основном, неприятные), когда дело дойдет до использования конкретных коммерческих продуктов.

В завершение этого раздела необходимо отметить, что в действительности термин *модель данных* используется в литературе в двух различных толкованиях. Первое определение этого термина описано выше. Второе состоит в следующем: модель данных (во втором смысле) представляет собой модель перманентных данных некоторого *конкретного предприятия* (например, промышленной компании KnowWare, Inc., упоминаемой выше в этом разделе). Таким образом, различия между этими двумя толкованиями можно охарактеризовать, как описано ниже.

- Модель данных в первом значении подобна *языку программирования* (причем достаточно абстрактному), конструкции которого могут быть использованы для решения широкого круга конкретных задач, но который сам по себе не имеет четкой связи с какой-либо из этих конкретных задач.

- Модель данных во втором значении подобна *конкретной программе,* написанной на таком языке. Иначе говоря, модель данных во втором значении использует средства, предоставляемые некоторой моделью (рассматриваемой в первом значении) и применяет их для решения конкретной проблемы. Ее можно рассматривать как некоторое *конкретное приложение* некоторой модели в первом значении.

В данной книге термин *модель данных,* начиная с этого момента, будет использоваться только в первом значении, если явно не указано обратное.

1.4. НАЗНАЧЕНИЕ БАЗ ДАННЫХ

Почему так широко используются системы с базами данных? Какие преимущества получает пользователь при работе с ними? В некоторой степени ответ зависит от того, о какой системе идет речь — однопользовательской или многопользовательской (точнее будет сказать, что многопользовательские системы предоставляют многочисленные *дополнительные* преимущества). Сначала рассмотрим случай однопользовательской системы.

Вернемся к нашей базе данных винного погреба (см. табл. 1.1), которую можно рассматривать как типичный пример однопользовательской базы данных. Она настолько мала и проста, что на ее примере сразу увидеть все потенциальные преимущества невозможно. Но представьте себе такую базу данных для большого ресторана, имеющего запас, возможно, из тысяч бутылок, который постоянно обновляется, или, скажем, для винного магазина, опять же с большим запасом, который постоянно расходуется и пополняется. Преимущества системы с базой данных по сравнению с традиционным "бумажным" методом ведения учета для этих примеров вполне очевидны. Отметим некоторые из них, которые описаны ниже.

- Компактность. Нет необходимости в создании и ведении, возможно, весьма объемистых бумажных картотек.
- Быстродействие. Компьютер может выбирать и обновлять данные гораздо быстрее человека. В частности, с его помощью можно быстро получать ответы на *произвольные* вопросы, возникающие в процессе работы (например, "Какого вина у нас сейчас больше — Zinfandel или Pinot Noir?"), не затрачивая времени на визуальный осмотр или поиск вручную.
- Низкие трудозатраты. Отпадает необходимость в утомительной работе над картотекой вручную. Механическую работу машины всегда выполняют лучше.
- Актуальность. В случае необходимости под рукой в любой момент имеется точная, свежая информация.
- Защита. Данные могут быть лучше защищены от случайной потери и несанкционированного доступа.

Эти преимущества приобретают еще большее значение в многопользовательской среде, где база данных, вероятно, больше и сложнее однопользовательской. Кроме того, многопользовательская среда имеет дополнительное преимущество: система баз данных *предоставляет предприятию средства централизованного управления его данными* (именно возможность такого управления является наиболее ценным свойством базы данных). Представьте себе противоположную ситуацию: предприятие не использует

систему баз данных, поэтому для каждого отдельного приложения создаются свои файлы, чаще всего размещаемые на отдельных магнитных лентах или дисках, в результате чего данные оказываются разрозненными. Систематически управлять такими данными очень сложно.

Администрирование данных и администрирование базы данных

Рассмотрим более подробно концепцию централизованного управления. Предполагается, что при централизованном управлении на предприятии, использующем базу данных, есть человек, который несет основную ответственность за данные предприятия. Это— **администратор данных,** или сокращенно АД, уже упоминавшийся в разделе 1.2. В связи с тем, что данные (как было отмечено выше) — это одна из главных ценностей предприятия, администратор должен разбираться в них и понимать нужды предприятия по отношению к данным *на уровне высшего управляющего звена* **в** руководстве предприятием. Сам администратор данных также должен относиться к этому звену. В его обязанности входит принятие решений о том, какие данные необходимо вносить в базу данных в первую очередь, а также выработка требований по сопровождению и обработке данных после их занесения в базу данных. Примером подобных требований может служить распоряжение о том, кто и при каких обстоятельствах имеет право выполнять конкретные операции над теми или иными данными. Другими *словами,* администратор данных должен обеспечивать *защиту данных* (подробнее об этом речь пойдет ниже).

Очень важно помнить, что администратор данных относится к управляющему звену, а не к техническим специалистам (хотя он, конечно, должен иметь хорошее представление о возможностях баз данных на техническом уровне). *Технический* специалист, ответственный за реализацию решений администратора данных, — это **администратор *базы* данных,** или АБД. Администратор базы данных, в отличие от администратора данных, должен быть профессиональным специалистом в области информационных технологий. Работа АБД заключается в создании самих баз данных и организации технического контроля, необходимого для осуществления решений, принятых администратором данных. АБД несет также ответственность за обеспечение необходимого быстродействия системы и ее техническое обслуживание. Обычно у АБД есть штат из системных программистов и технических ассистентов (т.е. на практике функции АБД часто выполняет группа из нескольких человек, а не один служащий). Однако для простоты удобнее считать, что администратор базы данных — один человек. Более подробно функции АБД описаны в главе 2.

Преимущества подхода, предусматривающего использование базы данных

Рассмотрим преимущества использования баз данных, связанные с наличием централизованного управления.

- ***Возможность совместного доступа к данным***

  Этот вопрос уже обсуждался в разделе 1.2, но для полноты проанализируем его еще раз. Совместный доступ к данным означает не только возможность доступа к ним с помощью нескольких существующих приложений базы данных, но и возможность разработки новых приложений для работы с этими же данными. Другими

словами, требования новых приложений по доступу к данным могут быть удовлетворены без необходимости добавления новых данных в базу.

- *Сокращение избыточности данных*

  В системах, не использующих базы данных, каждое приложение имеет свои файлы. Это часто приводит к избыточности хранимых данных и, следовательно, к нерациональному использованию пространства вторичной памяти. Например, и приложение, связанное с учетом персонала, и приложение, связанное с учетом результатов обучения служащих, могут иметь собственные файлы с ведомственной информацией о служащих. Но как отмечено в разделе 1.2, эти два файла можно объединить с устранением избыточной (повторяющейся) информации, при условии, что администратор данных знает о том, какие данные нужны для каждого приложения, т.е. на предприятии осуществляется необходимое общее управление.

  Примечание. В данном случае мы не имеем в виду, что избыточность данных может или должна быть устранена *полностью.* Иногда веские практические или технические причины требуют наличия нескольких копий хранимых данных. Однако такая избыточность должна строго **контролироваться,** т.е. учитываться в процессе эксплуатации СУБД. Кроме того, в подобном случае должна быть предусмотрена возможность *распространения обновлений* (подробности приводятся ниже).

- *Устранение противоречивости данных (до некоторой степени)*

  В действительности это следует из предыдущего пункта. Возьмем пример из жизни. Пусть служащий с табельным номером E3, работающий в отделе с номером D8, представлен двумя различными записями в базе данных. Предположим, что в СУБД не учтено это дублирование (т.е. избыточность данных не контролируется). Тогда рано или поздно обязательно возникнет ситуация, при которой эти две записи перестанут быть согласованными, после того как одна из них будет изменена, а другая — нет. В этом случае база данных станет *противоречивой.* Ясно, что противоречивая база данных будет предоставлять пользователю неправильную, противоречивую информацию.

  Также очевидно, что если какой-либо факт представлен только одной записью (т.е. избыточность отсутствует), то противоречия исключены. Противоречий можно также избежать, если избыточность не исключается, а *контролируется* (и это соответствующим образом предусмотрено в СУБД). Тогда СУБД сможет гарантировать, что *с точки зрения пользователя* база данных никогда не будет противоречивой. Данная гарантия обеспечивается тем, что если обновление вносится в одну запись, то оно автоматически будет распространено на все остальные. Такой процесс называется **распространением обновлений** (propagating updates).

- *Возможность поддержки транзакций*

  Транзакция (transaction) — это логическая единица работы (точнее, логическая единица работы базы данных), обычно включающая несколько операций базы данных (в частности, несколько операций модификации данных). Стандартный пример — перевод некоторой суммы денег со счета A на счет B. Очевидно, что в данном случае необходимы два изменения: списание некоторой суммы со счета

А и зачисление ее на счет В. Если пользователь укажет, что оба изменения входят в одну и ту же транзакцию, то система сможет реально гарантировать, что либо будут выполнены оба эти изменения, либо не будет выполнено ни одно из них, если до завершения процесса внесения изменений в системе произойдет сбой (скажем, из-за перерыва в подаче электроэнергии).

Примечание. Упомянутое выше свойство *неразрывности* (atomicity) транзакций — это не единственное положительное следствие поддержки транзакций. Однако в отличие от прочих, оно вполне применимо даже в однопользовательской среде. (С другой стороны, в однопользовательских системах поддержка транзакций часто совсем не предоставляется, а подобные функции просто возлагаются на пользователя.) Полное описание различных преимуществ поддержки транзакций и способов их достижения приведено в главах 15 и 16.

Обеспечение целостности данных

Задача обеспечения целостности заключается в гарантированной поддержке правильности данных в базе (насколько это возможно). Противоречие между двумя записями, представляющими один "факт", является примером утраты целостности данных (см. обсуждение этого вопроса выше в данном разделе). Конечно, эта конкретная проблема может возникнуть лишь при наличии избыточности в хранимых данных. Но даже если избыточность отсутствует, база данных может содержать неправильную информацию. Например, в базе данных может быть указано, что сотрудник отработал 400 рабочих часов в неделю вместо 40, или зафиксирована его принадлежность к отделу, которого не существует. Централизованное управление базой данных позволяет избежать подобных проблем (насколько это вообще возможно). Для этого администратор данных определяет (а администратор базы данных реализует) **ограничения целостности** (integrity constraints), которые будут применяться при любой попытке внести какие-либо изменения в соответствующие данные.

Отметим также, что целостность данных для многопользовательских систем баз данных даже более важна, чем для среды с "частными файлами", причем именно по той причине, что такая база данных поддерживает совместный доступ. При отсутствии должного контроля один пользователь вполне может неправильно обновить данные, от чего пострадают многие другие пользователи. Следует также сказать, что в большинстве существующих коммерческих СУБД поддержка ограничений целостности развита слабо, хотя в настоящее время в этом направлении наблюдаются некоторые улучшения. Приходится констатировать тот печальный факт, что, как будет показано в главе 9, ограничения целостности имеют значительно более фундаментальное и важное значение, чем это обычно признается на текущий момент.

Организация защиты данных

Благодаря полному контролю над базой данных администратор базы данных (безусловно, в соответствии с указаниями администратора данных) может обеспечить доступ к ней только через определенные каналы. Для этой цели могут устанавливаться **ограничения защиты** (security constraints), или правила, которые будут контролироваться при любой попытке доступа к конфиденциальным данным. Можно установить

различные правила для разных типов доступа (выборка, вставка, удаление и т.д.) к каждому из элементов информации в базе данных. Однако следует отметить, что при отсутствии таких правил безопасность данных подвергается большему риску, чем в обычной (разобщенной) файловой системе. Следовательно, централизованная природа системы баз данных в определенном смысле *требует* также наличия надежной системы защиты.

Возможность согласования противоречивых требований

Зная общие требования всего предприятия (а не требования каждого отдельного пользователя), администратор базы данных (опять же в соответствии с указаниями администратора данных) может структурировать базу данных таким образом, чтобы обслуживание было наилучшим для всего предприятия. Например, он может выбрать такое физическое представление данных во вторичной памяти, которое обеспечит быстрый доступ к информации для наиболее важных приложений (возможно, за счет снижения производительности некоторых других приложений).

Возможность введения стандартизации

Благодаря централизованному управлению базой данных администратор базы данных (по указанию администратора данных) может обеспечить соблюдение всех необходимых стандартов, регламентирующих представление данных в системе. Стандарты могут быть частными, корпоративными, ведомственными, промышленными, национальными и международными. Стандартизация представления данных наиболее важна с точки зрения *обмена* данными и их пересылки между системами. (Наибольшую актуальность этот вопрос приобретает в случае распределенных систем, речь о которых пойдет в главах 2, 21 и 27.) Кроме того, стандарты именования и документирования данных важны как в отношении их совместного использования, так и в отношении их описания.

Большинство перечисленных выше преимуществ достаточно очевидны. Однако есть еще одно преимущество, которое необходимо добавить к этому списку и которое не столь очевидно (хотя косвенно и охватывает несколько преимуществ). Речь идет об *обеспечении независимости от данных.* (Строго говоря, это скорее *цель* создания систем баз данных, а не обязательное их преимущество.) Концепция независимости настолько важна, что ей посвящен целый раздел, представленный ниже.

1.5. НЕЗАВИСИМОСТЬ ОТДАННЫХ

Независимость от данных может быть реализована на двух уровнях: физическом и логическом [1.3], [1.4]. Однако на данном этапе нас интересует только физическая независимость. Поэтому неуточненный термин *независимость от данных* мы пока будем понимать лишь как *физическую* независимость от данных. (Необходимо отметить, что термин *независимость от данных* не совсем подходящий — он не отражает достаточно точно сущность происходящего. Но поскольку традиционно используется именно этот термин, последуем общему правилу.)

Проще всего разобраться в понятии независимости от данных на примере той ситуации, когда независимость от данных отсутствует. Приложения, реализованные в старых системах *(дореляционные,* или созданные до появления систем баз данных), в той или иной мере *зависимы от данных.* Это означает, что способ организации данных во вторичной

памяти и способ доступа к ним диктуются требованиями приложения. Более того, *сведения об организации данных и способе доступа к ним встроены в саму логику и программный код приложения.* Например, предположим, что в некотором приложении используется файл EMPLOYEE (см. рис. 1.4). Исходя из соображений производительности решено, что этот файл необходимо проиндексировать по полю "имя служащего" (см. приложение Г). В старых системах в этом приложении учитывалось бы, что такой индекс существует и что последовательность записей в файле определена этим индексом. На основе таких сведений была бы построена вся внутренняя структура приложения. В частности, избранный способ реализации процедур доступа и обработки исключительных ситуаций в значительной степени зависел бы от особенностей интерфейса, предоставляемого программами управления данными.

Приложения, подобные описанному в этом примере, называются **зависимыми от данных,** так как невозможно изменить физическое представление (т.е. способ физического размещения данных во вторичной памяти) или метод доступа (т.е. конкретный способ доступа к данным), не изменив самого приложения (возможно, весьма радикально). Например, невозможно заменить индексированный файл в нашем примере хэшированным файлом, не внеся в приложение значительных изменений. Более того, изменению в подобных случаях подлежат те части приложения, которые взаимодействуют с программами управления данными. Трудности, возникающие при этом, не имеют никакого отношения к проблеме, для решения которой было написано данное приложение; это трудности, *порождаемые* используемой структурой интерфейса управления данными.

Однако для системы баз данных крайне нежелательно, чтобы приложение зависело от данных, и на то есть по меньшей мере две причины, описанные ниже.

1. Для разных приложений требуются разные представления одних и тех же данных. Например, предположим, что до перехода к интегрированной базе данных пред приятие имело два приложения, А и В. Каждое из них работало с собственным файлом, содержащим поле "остаток средств на счете заказчика". Предположим также, что приложение А записывает значение этого поля в десятичном формате, а приложение В — в двоичном. Эти два файла все еще можно интегрировать, а су ществующую избыточность устранить, если в СУБД есть возможность выполнить все необходимые преобразования между форматом представления данных (формат представления может быть десятичным, двоичным или любым другим) и форма том, необходимым для приложения. Например, если принято решение хранить значения этого поля в десятичном формате, то каждое обращение к приложению В потребует прямого или обратного преобразования значений из десятичного фор мата в двоичный.

   Это довольно простой пример различий, которые могут существовать в системе баз данных между формой представления данных в приложении и формой их физического хранения. Многие другие возможные различия будут рассмотрены ниже.

2. Администратор базы данных должен иметь определенные возможности (завися щие от применяемой СУБД) по изменению физического представления или метода доступа к данным в случае изменения требований, причем без необходи мости модифицировать существующие приложения. Например, к базе данных могут быть добавлены новые виды данных, на предприятии могут быть приняты новые стандарты, могут быть изменены приоритеты приложений (а следовательно, и

связанные с ними требования к производительности), могут появиться новые типы запоминающих устройств и т.д. Если приложения зависят от данных, то подобные изменения потребуют внесения корректировок в программы, а значит, дополнительных трудозатрат программистов, которые можно было бы направить на создание новых приложений. До сих пор подобные проблемы не являются исключением. И сегодня случается, что значительная часть рабочего времени программистов тратится на подобную работу (достаточно вспомнить хотя бы проблему 2000-го года!), а это, конечно, бесполезная трата дефицитных

Таким образом, обеспечение независимости от данных — важнейшая цель создания систем баз данных. Независимость от данных можно определить как **невосприимчивость приложений к изменениям в физическом представлении данных и в методах доступа к ним,** а это означает, что рассматриваемые приложения не зависят от любых конкретных способов физического представления информации или выбранных методов доступа к ним. В главе 2 будет описана архитектура систем баз данных, обеспечивающая основу для достижения этой цели. Однако прежде всего рассмотрим более подробно некоторые примеры тех видов изменений, в выполнении которых может возникнуть необходимость и к которым, следовательно, должны быть невосприимчивы приложения.

Начнем с определения трех новых терминов: *хранимое поле, хранимая запись* и *хранимый файл* (рис. 1.6).

- **Хранимое поле** — это наименьшая единица хранимых данных. Типичная база данных содержит множество **экземпляров** (occurence, или instance) каждого из нескольких описанных в ней **типов** хранимых полей. Например, база данных, содержащая информацию о деталях, может включать тип хранимого поля с именем "номер детали" и для каждого описанного в базе данных вида детали (винта, шарнира, колпака и т.д.) будет существовать отдельный экземпляр этого хранимого поля.

  Примечание. На практике часто опускают квалификаторы *тип* и *экземпляр,* полагая, что точный смысл ясен из контекста. Хотя и существует небольшая вероятность путаницы, это удобная практика, и мы будем ей время от времени следовать. (Такое примечание относится также к хранимым записям, речь о которых пойдет в следующем абзаце.)

- **Хранимая запись** — это набор взаимосвязанных хранимых полей. И снова мы различаем для них тип и экземпляр. В данном случае **экземпляр** хранимой записи состоит из группы связанных экземпляров хранимых полей. Например, экземпляр хранимой записи в базе данных деталей состоит из экземпляров каждого из следующих хранимых полей: "номер детали", "название детали", "цвет детали" и "вес детали". Мы говорим, что база данных содержит множество экземпляров хранимой записи **типа** "деталь" (опять же, по одному экземпляру для каждой конкретной детали).

- Наконец, **хранимый файл** — это набор всех существующих в настоящий момент экземпляров хранимых записей одного и того же типа. (Для упрощения предполагается, что любой заданный хранимый файл может содержать хранимые записи

только одного типа. Это упрощение не окажет существенного влияния на последующие рассуждения.)

В современных системах, отличных от баз данных, *логическая* (с точки зрения разработчика приложения) запись обычно совпадает с соответствующей *хранимой* записью. Как было показано выше, в базах данных это вовсе не обязательно, поскольку в любой момент может потребоваться внести изменения в структуру хранения данных (т.е. в хранимые поля, записи, файлы), в то время как структура данных с точки зрения приложения должна остаться неизменной. Например, поле SALARY в файле EMPLOYEE для экономии памяти может быть сохранено в двоичном формате, а в приложении, написанном на языке COBOL, это поле может рассматриваться в качестве символьной строки. В дальнейшем по каким-то причинам может понадобиться изменить двоичную форму представления этого поля на десятичную, сохранив для приложения возможность обрабатывать поле в символьном формате.

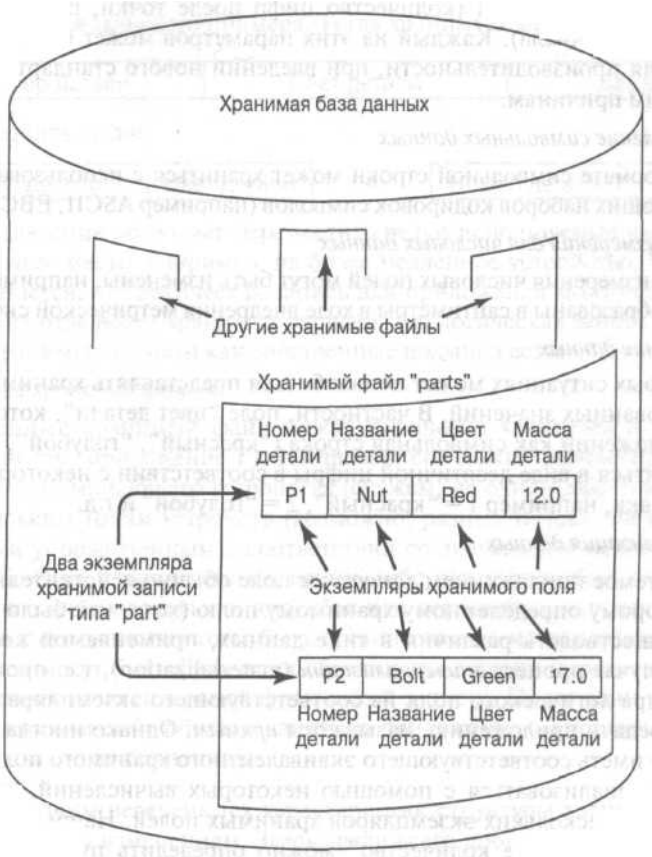


Рис. 1.6. Хранимые поля, записи и файлы

Как утверждалось ранее, такие различия (требующие преобразования типа данных некоторого поля при каждом обращении к нему) являются довольно незначительными.

Однако, в принципе, разница между теми данными, к которым получает доступ приложение, и теми, которые хранятся в базе на самом деле, может быть весьма значительной. Развивая эту мысль, перечислим те аспекты структур хранения данных в базах, которые могут подвергаться изменениям. Читателю предлагается самостоятельно подумать над тем, что должна сделать СУБД для обеспечения невосприимчивости приложения к таким изменениям (и всегда ли можно добиться подобной невосприимчивости).

- *Представление числовых данных*

  Числовое поле может храниться во внутренней арифметической форме (например, в упакованном десятичном формате) или в виде символьной строки. В каждом случае АБД должен определить, следует ли применять числа с фиксированной или плавающей точкой, выбрать подходящее основание системы счисления (например, двоичную или десятичную систему), точность (количество цифр во внутреннем представлении числа), а если это число с фиксированной точкой, то определить величину дробной части (количество цифр после точки, разделяющей целую и дробную части числа). Каждый из этих параметров может быть изменен в целях повышения производительности, при введении нового стандарта или по некоторым другим причинам.

- *Представление символьных данных*

  Поле в формате символьной строки может храниться с использованием любого из существующих наборов кодировок символов (например ASCII, EBCDIC, Unicode).

- *Единицы измерения для числовых данных*

  Единицы измерения числовых полей могут быть изменены, например, дюймы могут быть преобразованы в сантиметры в ходе внедрения метрической системы единиц.

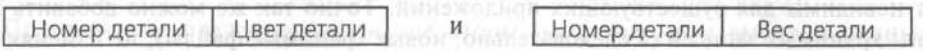
- *Кодирование данных*

  В некоторых ситуациях может понадобиться представлять хранимые данные в виде кодированных значений. В частности, поле "цвет детали", которое представлено в приложении как символьная строка ("красный", "голубой", "зеленый"), может храниться в виде десятичной цифры в соответствии с некоторой таблицей перекодировки, например 1 = "красный", 2 = "голубой" и т.д.

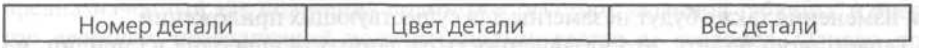
- *Материализация данных*

  Используемое приложением *логическое* поле обычно действительно соответствует некоторому определенному хранимому полю (хотя, как было показано выше, могут существовать различия в типе данных, применяемой кодировке и т.д.). В этом случае процесс *материализации* (materialization), т.е. процесс построения экземпляра логического поля из соответствующего экземпляра хранимого поля и его передачи приложению, называется *прямым.* Однако иногда логическое поле может не иметь соответствующего эквивалентного хранимого поля, а его значение будет материализоваться с помощью некоторых вычислений, выполняемых над набором из нескольких экземпляров хранимых полей. Например, значение логического поля "общее количество" можно определить путем суммирования нескольких хранимых значений поля "количество". В подобном случае процесс материализации называется *косвенным.*

Структура хранимых записей Две существующие хранимые записи можно объединить в одну. Например, хранимые записи

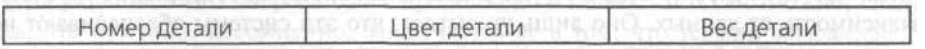


можно представить в форме:

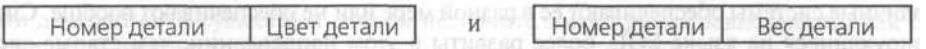


Такие изменения могут выполняться, когда в систему баз данных вводятся новые приложения. При этом предполагается, что логическая запись приложения состоит из определенного подмножества полей соответствующей хранимой записи, т.е. некоторые поля хранимой записи "невидимы" для рассматриваемого приложения.

И наоборот, одна хранимая запись может быть разделена на две. Воспользуемся записями из предыдущего примера. Тогда хранимую запись



можно разбить на две:



Такое разделение позволяет переместить редко используемые части исходной записи в другое место, например, на более медленное устройство. При этом неявно предполагается, что логическая запись для приложения может содержать поля из нескольких отдельных хранимых записей, т.е. логическая запись включает любые из этих хранимых записей как собственные подмножества.

- *Структура хранимых файлов*

  Определенный хранимый файл может физически храниться в памяти разными способами (см. приложение Г). Например, его можно разместить на одном томе внешнего запоминающего устройства (скажем, на одном диске) или распределить по нескольким томам устройств (возможно, разных типов). Он может либо быть физически упорядоченным в соответствии со значениями некоторого хранимого поля, либо быть упорядоченным каким-либо иным способом, например с помощью одного либо нескольких индексов или встроенных цепочек указателей, или же доступ к его записям может быть организован по методу хэширования. Хранимые записи могут быть физически объединены в блоки (с размещением нескольких хранимых записей в одной физической записи). Но ни один из этих факторов не должен каким-либо образом влиять на приложение (за исключением, безусловно, скорости его выполнения).

Этим мы ограничим перечень тех характеристик структуры хранения данных, которые могут подвергаться изменениям. Здесь среди всего прочего предполагается, что база данных может **расти** и развиваться, не оказывая влияния на приложения. В действительности, возможность развития базы данных без нарушения логической структуры существующих приложений является одним из наиболее важных стимулов для обеспечения независимости от данных. Например, можно было бы расширить существующий

тип хранимой записи, добавив новые хранимые поля, которые обычно представляют дополнительную информацию о некоторых возможных типах сущностей (скажем, к хранимой записи "деталь" можно добавить поле "цена за штуку"). Такие новые поля будут невидимы для существующих приложений. Точно так же можно добавить новые типы хранимых записей (а следовательно, новые хранимые файлы), не изменяя существующих приложений. Подобные записи обычно представляют собой новые типы сущностей (например, к базе данных "детали" можно добавить тип записи "поставщик"). Эти изменения также будут незаметны для существующих приложений.

Теперь легко понять, что независимость от данных— еще одна из причин, по которым следует отделить модель данных от ее реализации, как уже было показано в конце раздела 1.3. Важно также отметить, что независимость от данных нельзя обеспечить, не достигнув должной степени отделения модели данных от ее реализации. Недостаточное разделение модели и ее реализации является широко распространенным недостатком современных систем, использующих язык SQL (что особенно огорчает).

Примечание. Последнее примечание (относительно использующих язык SQL современных систем) вовсе не означает, что в подобных системах совершенно не обеспечена независимость от данных. Оно лишь указывает, что эти системы обеспечивают независимость от данных в значительно меньшей степени, чем теоретически возможно в реляционных системах[4]. Иными словами, независимость от данных — понятие относительное. Различные системы обеспечивают ее в разной мере или не обеспечивают вообще. Системы, базирующиеся на языке SQL, более развиты в этом направлении, чем старые системы, но, как будет показано в следующих главах, все они еще далеки от совершенства.

1.6. РЕЛЯЦИОННЫЕ И ДРУГИЕ СИСТЕМЫ

Как уже было сказано выше, *системы с поддержкой SQL,* или просто *системы SQL,* в настоящее время стали преобладающими на рынке баз данных, и одной из важных причин подобного состояния дел является именно то, что такие системы основаны на *реляционной модели данных.* Поэтому не удивительно, что в неформальном общении системы SQL часто называют просто реляционными системами[5]. Кроме того, подавляющее большинство научных исследований в области баз данных в течение последних 30 лет было посвящено (иногда косвенно) именно этой модели. Фактически, введение реляционной модели в 1969 и 1970 годах было, несомненно, *наиболее важным событием во всей истории развития теории баз данных.* По этим причинам, а также учитывая то, что реляционная модель основана на определенных математических и логических принципах и, следовательно, идеально подходит для изложения теоретических концепций систем баз данных, основное внимание в настоящей книге уделяется именно реляционным системам и реляционному подходу.

Что подразумевается под реляционной системой? К сожалению, на данном этапе обсуждения невозможно дать полный ответ на этот вопрос. Однако можно (и нужно!) дать хотя бы приблизительный ответ, который в дальнейшем будет существенно уточнен.

---

[4] В приложении А приведен поразительный пример того, на что реляционные системы способны в этом отношении.

[5] Даже несмотря на тот факт, что системы SQL, как будет показано ниже, во многих отношениях ха рактеризуются значительными отклонениями от реляционной модели.

Итак, кратко и не совсем точно можно определить, что реляционная система — это система, основанная на описанных ниже принципах.

Данные рассматриваются пользователем как таблицы (и никак иначе).

Пользователю предоставляются операторы (например, для выборки данных), позволяющие генерировать новые таблицы на основании уже существующих. Например, в системе обязательно должны присутствовать оператор *сокращения,* предназначенный для получения подмножества строк заданной таблицы, и оператор *проекции,* позволяющий получить подмножество ее столбцов. Однако подмножество строк и подмножество столбцов некоторой таблицы, безусловно, можно рассматривать как новые таблицы.

Причина, по которой такие системы называют *реляционными,* состоит в том, что английский термин "relation" *{отношение),* по сути, представляет собой общепринятое математическое название для *таблиц.* Поэтому на практике термины *отношение* и *таблица* в большинстве случаев можно считать синонимами, по крайней мере, для неформальных целей. (Обсуждение этого вопроса будет продолжено в главах 3 и 6.) Возможно, следует добавить, что причина, несомненно, заключается *не* в том, что термин *отношение* "по существу — просто формальное название" для связи (relationship) в терминах диаграмм "сущность-связь" (см. раздел 1.3). На самом деле между реляционными системами и подобными диаграммами существует совсем незначительная прямая зависимость, как отмечено в данном разделе.

Как уже упоминалось, в дальнейшем будет дано более точное определение, а пока мы будем использовать приведенное выше. На рис. 1.7 представлен пример структуры данных и операторов, используемых в реляционных системах. Данные, приведенные на рис. 1.7, *а,* состоят из одной таблицы CELLAR (в действительности это еще одна версия таблицы CELLAR (см. табл. 1.1), которая просто была уменьшена для того, чтобы с ней было легче работать). На рис. 1.7, *б* показаны два примера выборки данных: один— с получением подмножества строк (оператор *сокращения),* а другой — с получением подмножества столбцов (оператор *проекции).* В обоих этих примерах снова применяется язык SQL.

Теперь мы можем различать реляционные и нереляционные Системы по следующим признакам. Как уже отмечалось, пользователь реляционной системы видит данные в виде таблиц и никак иначе. Пользователь нереляционной системы, напротив, видит данные, представленные в других структурах— либо вместо таблиц реляционной системы, либо наряду с ними. Для работы с этими другими структурами применяются иные операции. В частности, в иерархической системе (например, в СУБД IMS фирмы IBM) данные представляются пользователю в форме набора древовидных структур (иерархий), а среди операций работы с иерархическими структурами есть операции *перемещения (навигации) по иерархическим указателям,* позволяющие переходить вверх и вниз по ветвям деревьев. Реляционные системы, как мы видели, не имеют таких указателей, и это очень важная их отличительная особенность (по крайней мере, в них отсутствуют указатели, видимые для пользователя, иными словами, указатели на уровне модели, но могут быть предусмотрены указатели на уровне физической реализации).

На основании изложенного можно сделать вывод, что системы баз данных могут быть легко распределены по категориям в соответствии со структурами данных и операциями, которые они предоставляют пользователю. Прежде всего, старые (дореляционные)

системы можно разделить на три большие категории[6]: системы с **инвертированными списками** (inverted list), **иерархические** (hierarchic) и **сетевые** (network). *{Примечание.* Термин *сетевая система* в данном случае не имеет ничего общего с *коммуникационной* сетью, как описано в следующей главе.) В настоящей книге эти категории подробно не рассматриваются, поскольку, по крайней мере, с точки зрения технологии, их можно считать устаревшими. Заинтересованный читатель может найти методическое описание всех трех систем в [1.5].

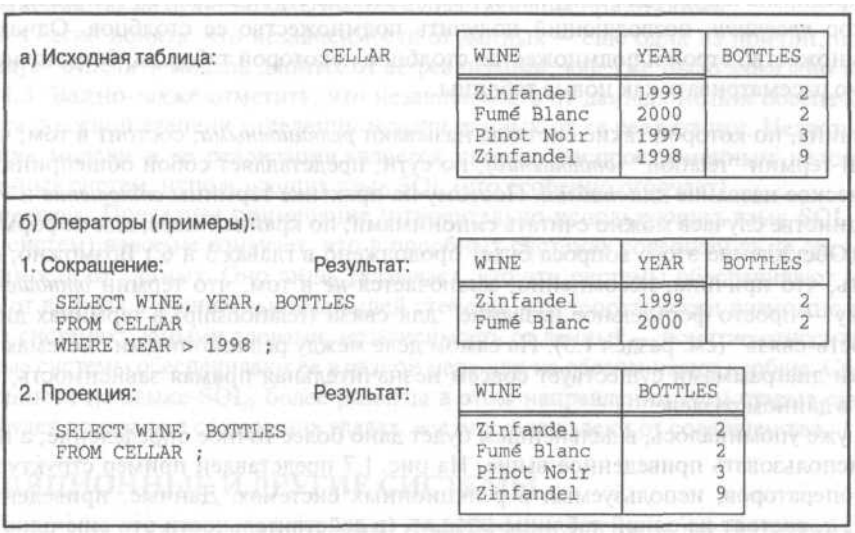


Рис. 1.7. Структура данных и операторы в реляционной системе (примеры)

Следует отметить, что сетевые системы иногда называют системами CODASYL или системами **DBTG** (Data Base Task Group) в честь предложившего их подразделения организации CODASYL (Conference on Data Systems Language). Пожалуй, наиболее известной из таких систем была IDMS корпорации Computer Associates International, Inc. Подобно иерархическим системам (но в отличие от реляционных), все подобные системы предоставляют пользователю доступ к указателям.

Первые **реляционные** продукты начали появляться в конце 1970-х и начале 1980-х годов. Ко времени написания этой книги преобладающее большинство СУБД были реляционными (по меньшей мере, они поддерживали язык SQL) и предназначались для работы практически на любой существующей программной и аппаратной компьютерной платформе. Среди них ведущими (в алфавитном порядке) являлись следующие: DB2

---

[6] По аналогии с реляционной моделью, в ранних изданиях книги использовались термины *модель инвертированных списков, иерархическая модель* и *сетевая модель* (они также использовались в других книгах). Однако это не совсем верно, поскольку по сравнению с реляционной моделью, "модели" инвертированного списка, иерархическая и сетевая были определены *после свершившегося факта,* т.е. соответствующие коммерческие продукты были реализованы *раньше,* а "модели" были определены *впоследствии* "по индукции" (в данном контексте так мы из вежливости назвали не подкрепленные теорией рассуждения) из уже существующих реализаций. Дополнительная информация по этой теме приведена в аннотации к [1.1].

(всевозможные версии) корпорации IBM; Ingres II корпорации Computer Associates International, Inc.; Informix Dynamic Server корпорации Informix Software, Inc.[7]; Microsoft SQL Server корпорации Microsoft; Oracle 8i корпорации Oracle и Sybase Adaptive Server компании Sybase, Inc.

Примечание. Если придется ссылаться на эти продукты ниже в настоящей книге, мы будем называть их (как это делается в большинстве случаев) сокращенными именами: DB2, Ingres (произносится "ингресс"), Informix, SQL Server, Oracle и Sybase.

В последнее время стали появляться **объектно-ориентированные** и **объектно-реляционные** продукты: первые объектные системы были выпущены в конце 1980-х и начале 1990-х годов, а **объектно-реляционные** системы были созданы в конце 1990-х годов. Большинство объектно-реляционных СУБД основываются на расширенных оригинальных реляционных продуктах, как в случае с DB2 или Informix. Существующие объектно-ориентированные системы иногда представляют собой попытки создать нечто совершенно отличное от других систем, как это имеет место в случае с системой GemStone корпорации GemStone Systems, Inc. и системой Versant ODBMS компании Versant Object Technology. Эти новые системы рассматриваются в части VI данной книги. (Следует отметить, что термин *объект,* применяемый в данном абзаце, имеет довольно специальное значение, как будет описано в части VI. Но в остальных частях настоящей книги, если не указано иное, этот термин используется в его обычном, универсальном смысле.)

В дополнение к различным уже упоминавшимся выше подходам, в течение нескольких лет проводились исследования множества альтернативных схем, включая **многомерный** (multi-dimensional) подход и **логический** (logic-based) подход, называемый еще *дедуктивным* или *экспертным.* Мы рассмотрим многомерные системы в главе 22, а логические — в главе 24. Кроме того, в связи с наблюдающимся в последнее время стремительным ростом World Wide Web и расширением области применения языка XML, проявляется значительный интерес к направлению научной деятельности, которое получило (не совсем удачное) название *полуструктурированный подход.* Полуструктурированные системы рассматриваются в главе 27.

1.7. РЕЗЮМЕ

Итак, подведем итог обсуждению основных вопросов. **Систему баз данных** можно рассматривать как компьютеризированную систему хранения записей. Такая система включает сами по себе **данные** (хранимые в **базе данных), аппаратное обеспечение, программное обеспечение** (в частности, **систему управления базами данных,** или СУБД), а также **пользователей** (что наиболее важно). Пользователи, в свою очередь, подразделяются на **прикладных программистов, конечных пользователей** и **администраторов базы данных,** или АБД. Последние отвечают за администрирование базы данных и всей системы баз данных в соответствии с требованиями, устанавливаемыми **администратором данных.**

Базы данных являются **интегрированными** и чаще всего **совместно используемыми.** Они применяются для хранения **перманентных данных.** Можно считать (хотя это и не совсем точно), что эти данные представляют собой **сущности** и **связи** между этими сущностями, хотя сами связи — это, по сути, просто специальный вид сущности. Очень кратко мы рассмотрели понятие **диаграмм "сущность—связь".**

---

[7] Отделение разработки СУБД компании Informix Software, Inc. было приобретено корпорацией IBM в 2001 году.

Система баз данных имеет ряд преимуществ, наиболее важным из которых является физическая **независимость от данных.** Независимость от данных может быть определена как невосприимчивость прикладных программ к изменениям способа физического хранения данных и используемых методов доступа. Среди всего прочего для обеспечения независимости от данных требуется строгое разделение между **моделью данных** и ее **реализацией.** (По ходу изложения напоминаем, что термин *модель данных,* к нашему сожалению, имеет два различных значения.)

Системы баз данных обычно поддерживают **транзакции,** или логические единицы работы. Основное преимущество транзакций заключается в том, что они гарантируют **неразрывность** выполняемых действий (по принципу "все или ничего"), несмотря на возможные сбои системы, имевшие место до завершения выполнения транзакции.

Наконец, система баз данных может быть основана на нескольких различных подходах. Реляционные системы базируются на формальной теории, называемой **реляционной моделью,** в соответствии с которой данные представляются в виде строк в таблицах (и интерпретируются как **истинные высказывания),** а пользователям предоставляется возможность использовать операции, обеспечивающие поддержку процесса **логического вывода** дополнительных истинных высказываний как следствий из существующих данных. И с экономической, и с формальной точек зрения можно считать, что реляционные системы являются наиболее важным сегментом рынка баз данных (и это положение дел, по-видимому, не изменится в обозримом будущем). Мы рассмотрели несколько примеров использования языка **SQL** — стандартного языка для работы с реляционными системами (в частности, были приведены примеры операторов **SELECT, INSERT, DELETE** и **UPDATE** этого языка). Материал данной книги в значительной мере ориентирован на реляционные системы и в меньшей мере — на сам язык SQL (по причинам, указанным в предисловии).

УПРАЖНЕНИЯ

7.1. Дайте определения следующим терминам:

АБД
администрирование данных
база данных
бинарная связь
диаграмма "сущность—связь"
защита
избыточность
интеграция
интерактивное приложение
командный интерфейс
многопользовательская система
независимость от данных
некомандный интерфейс (меню)
некомандный интерфейс (формы)
параллельный доступ
перманентные данные
свойство
связь
система баз данных
совместное использова
СУБД
сущность
транзакция
хранимая запись
хранимое поле
хранимый файл
целостность данных
язык запросов

1.2. Каковы преимущества использования системы баз данных? Каковы недостатки использования системы баз данных?

1.3. Как вы понимаете термин *реляционная система?* Укажите различия между реляционной и нереляционной системами.

1.4. Как вы понимаете термин *модель данных?* Объясните различие между моделью данных и ее реализацией. Почему так важно это различие?

1.5. Приведите результат выполнения следующих операторов SQL выборки информации из базы данных винного погреба, представленной в табл. 11.

а)
```
SELECT WINE, PRODUCER
FROM CELLAR
WHERE BIN# = 72 ;
```

б)
```
SELECT WINE, PRODUCER
FROM CELLAR
WHERE YEAR > 2 000 ;
```

В)
```
SELECT BIN#, WINE, YEAR
FROM CELLAR WHERE READY <
2003 ;
```

Г)
```
SELECT WINE, BIN#, YEAR FROM
CELLAR WHERE PRODUCER = 'Robt.
Mondavi' AND BOTTLES > б ;
```

1.6. Дайте собственную словесную интерпретацию типичной строки в одном из ответов на упр. 1.5, представив ее в виде истинного высказывания.

1.7. Приведите результат выполнения следующих операторов SQL внесения изменений в базу данных винного погреба, представленную в табл. 11.

а)
```
INSERT
INTO CELLAR (BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY )f?
VALUES (80, 'Syrah', 'Meridian', 1998, 12, 2003 );
```

б)
```
DELETE
FROM CELLAR
WHERE READY > 2004 ;
```

В)
```
UPDATE CELLAR
SET BOTTLES = 5 WHERE
BIN# = 50 ;"
```

Г)
```
UPDATE CELLAR
SET BOTTLES = BOTTLES + 2
WHERE BIN# = 50 ;
```

1.8. Напишите оператор SQL для выполнения приведенных ниже операций в базе данных винного погреба.

а) Выбрать номер ячейки, наименование вина и количество бутылок для всех вин производства Geyser Peak.

б) Выбрать номер ячейки и наименование вина для всех вин, запас которых составляет больше пяти бутылок.

в) Выбрать номер ячейки для всех красных вин.

г) Добавить три бутылки в ячейку с номером 3 0.

д) Удалить из всего запаса вин все вина производства компании Chardonnay. *%*

е) Добавить данные нового поступления (12 бутылок): производитель — Gary Farrell, сорт — Merlot, ячейка номер 55, год выпуска — 2000, будет готово к употреблению в 2005 году.

1.9. Предположим, что у вас есть коллекция записей классической музыки, содержащаяся на компакт-дисках, мини-дисках, долгоиграющих пластинках и аудиокассетах, и вы хотите создать базу данных, которая позволит находить записи определенного композитора (например, Сибелиуса), дирижера (например, Симона Ратла), солиста (например, Артура Грюмикса), произведения (например, Пятой симфонии Бетховена), оркестра (например, Нью-Йоркского филармонического), вида произведения (например, концерта для скрипки) или оркестровой группы (например, квартета Кронус). Начертите диаграмму "сущность—связь" для этой базы данных по образцу, представленному на рис. 1.5.

СПИСОК ЛИТЕРАТУРЫ

1.1. Codd E.F. Data Models in Databases Management // Proc. Workshop on Data Abstrac tion, Database and Conceptual Modelling. — Pingree Park, Colo, June 1980. Эту статью можно также найти в ACM S1GART Newsletter.— January 1981, №74; ACM SIGMOD Record 11. — February 1981, № 2 и в других источниках.

Кодд является создателем реляционной модели, которую он впервые описал в [6.1]. Но в этой работе он на самом деле не дал определения термину *модель данных* как таковому; оно было дано гораздо позже, лишь в данной работе. В ней рассматривается вопрос "Каково назначение моделей данных вообще и реляционной модели в частности?", а также приводятся факты, подтверждающие, что вопреки распространенному мнению реляционная модель фактически стала самой первой моделью данных, которая была когда-либо определена. Иначе говоря, Кодда по праву называют создателем концепции модели данных вообще, а также реляционной модели в частности.

1.2. Darwen H. What a Database Really Is: Predicates and Propositions // Date C.J., Darwen H., and McGoveran D. Relational Database Writings 1994—1997. — Reading, Mass.: Addison-Wesley, 1998.

В этой статье дается неформальное, но точное объяснение идеи (которая кратко обсуждалась в конце раздела 1.3), содержащей утверждение, что базу данных лучше представлять как набор истинных высказываний.

1.3. Date C.J. and Hopewell P. Storage Structures and Physical Data Independence // Proc. ACM SIGFIDET Workshop on Data Definition, Access, and Control. — San Diego, California. — November 1971.

1.4. Date C.J. and Hopewell P. File Definition and Logical Data Independence // Proc. ACM SIGFIDET Workshop on Data Definition, Access, and Control. — San Diego, California. — November 1971.

Статьи [1.3], [1.4] являются первыми письменными работами, в которых было определено различие между физической и логической независимостью отданных.

1.5. Date C.J. Relation Database Writtings 1991-1994. — Reading, Mass.: Addison-Wesley, 1995.

ГЛАВА 2

Архитектура системы баз данных

2.1. ВВЕДЕНИЕ

Теперь, после изучения вводной главы, мы можем ознакомиться с архитектурой системы баз данных. Наша цель — "заложить фундамент", на котором будет строиться дальнейшее изложение. Именно на него мы будем опираться при описании общих понятий и изучении структуры конкретных систем баз данных. Однако это отнюдь не означает, что каждая система баз данных обязательно должна соответствовать этому определению или что предложенная конкретная архитектура является единственно возможной. Например, "малые" системы (см. главу 1), весьма вероятно, не будут поддерживать все функции предложенной архитектуры. Тем не менее, рассматриваемая архитектура с достаточной точностью описывает большинство систем (и не только реляционных). Более того, она практически полностью согласуется с архитектурой, предложенной исследовательской

группой ANSI/SPARC (Study Group on Data Management Systems), — так называемой архитектурой ANSI/SPARC [2.1], [2.2]. Однако здесь мы не будем придерживаться терминологии ANSI/SPARC во всех деталях.

Следует отметить, что материал этой главы подобен материалу предыдущей в том смысле, что он является основой, необходимой для полного понимания структуры и возможностей современных систем баз данных. По этой причине он также носит несколько абстрактный характер, следовательно, достаточно "академичен". Учитывая это, как и в случае изучения главы 1, предварительно можно бегло просмотреть настоящую главу, а затем, по мере освоения излагаемого в книге материала, возвращаться к отдельным ее разделам, непосредственно связанным с той или иной изучаемой темой.

2.2. ТРИ УРОВНЯ АРХИТЕКТУРЫ

Архитектура ANSI/SPARC включает три уровня: внутренний, внешний и концептуальный (рис. 2.1). В общих чертах они представляют собой следующее.

Рис. 2.1. Три уровня архитектуры ANSI/SPARC

- **Внутренний уровень** (называемый также *физическим)* наиболее близок к физическому хранилищу информации, т.е. связан со способами сохранения информации на физических устройствах.
- **Внешний уровень** (называемый также *пользовательским логическим)* наиболее близок к пользователям, т.е. связан со способами представления данных для отдельных пользователей.
- **Концептуальный уровень** (называемый также *общим логическим* или просто *логическим,* без дополнительного определения) является "промежуточным" уровнем между двумя первыми.

Если внешний уровень связан с *индивидуальными* представлениями пользователей, то концептуальный уровень связан с *обобщенным* представлением пользователей. Как было отмечено в главе 1, большинству пользователей нужна не вся база данных, а только ее небольшая часть, поэтому может существовать несколько внешних представлений, каждое из которых состоит из более или менее абстрактного представления определенной части базы данных, и только одно концептуальное представление, состоящее из абстрактного представления базы данных в целом. Кроме того, и внешний, и концептуальный

уровни представляют собой уровни моделирования, а внутренний служит в качестве уровня реализации; иными словами, первые два уровня определены в терминах таких пользовательских информационных конструкций, как *записи* и *поля,* а последний — в терминах машинно-ориентированных конструкций наподобие битов и *байтов.*

Для лучшего понимания этих идей рассмотрим пример, представленный на рис. 2.2. Здесь отображено концептуальное представление простой базы данных о персонале, а также соответствующие ему внутреннее и два внешних представления (одно — для пользователя, применяющего язык PL/I, а другое — для пользователя, применяющего язык COBOL)[1]. Конечно, этот пример полностью гипотетичен и мало похож на реальные системы, поскольку в нем умышленно исключены многие не относящиеся к делу детали.

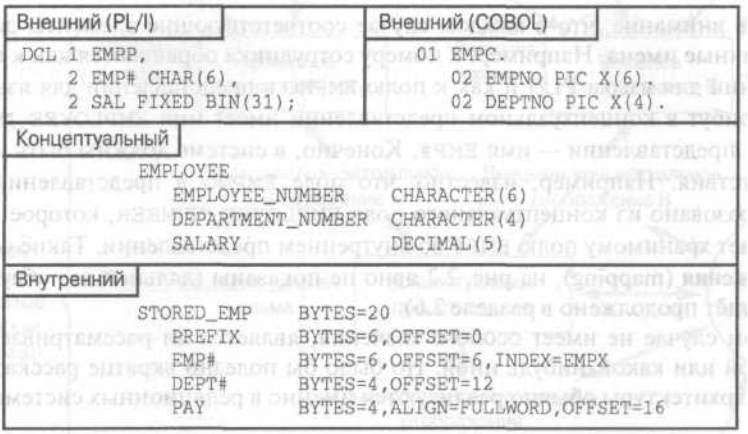


Рис. 2.2. Пример трех уровней представления базы данных

Рассматриваемый здесь пример нуждается в пояснениях.

1. На концептуальном уровне база данных содержит информацию о типе сущности с именем EMPLOYEE (служащий). Каждый экземпляр сущности EMPLOYEE включает атрибуты номера служащего EMPLOYEE_NUMBER (шесть символов), номера отдела DEPARTMENT_NUMBER (четыре символа) и зарплаты служащего SALARY (пять де сятичных цифр).
2. На внутреннем уровне служащие представлены типом хранимой записи STORED_EMP, длина которой составляет 20 байтов. Запись STORED_EMP содержит четыре храни мых поля: шестибайтовый префикс (возможно, содержащий управляющую ин формацию, такую как флажки или указатели) и три поля данных, соответствую щие трем свойствам сущности, которая представляет служащего. Кроме того, записи STORED_EMP индексированы по полю EMP# с помощью индекса EMPX, оп-

[1] Традиционные языки программирования PL/I и COBOL, послужившие основой для данного примера, все еще широко используются в программном обеспечении, установленном на многих предприятиях.

3. Пользователь, применяющий язык PL/I, имеет дело с соответствующим внешним представлением базы данных. В нем каждый сотрудник представлен записью на языке PL/I, содержащей два поля (номера отделов данному пользователю не требуются, поэтому в представлении они опущены). Тип записи определен с помощью обычной структуры, соответствующей правилам языка PL/I.
4. Аналогично, пользователь, применяющий язык COBOL, имеет дело с собственным внешним представлением базы данных, в котором каждый сотрудник представлен записью на языке COBOL, содержащей, опять же, два поля (в данном случае опущен размер оклада). Тип записи определен с помощью обычного описания на языке COBOL в соответствии с принятыми в нем стандартными правилами.

Обратите внимание, что в каждом случае соответствующие элементы данных могут иметь различные имена. Например, к номеру сотрудника обращаются как к полю EMP# в представлении для языка PL/I и как к полю EMPNO в представлении для языка COBOL. Этот же атрибут в концептуальном представлении имеет имя EMPLOYEE_NUMBER, а во внутреннем представлении — имя EMP#. Конечно, в системе должны быть известны все эти соответствия. Например, известно, что поле EMPNO в представлении для языка COBOL образовано из концептуального поля EMPLOYEE_NUMBER, которое, в свою очередь, отвечает хранимому полю EMP# во внутреннем представлении. Такие соответствия, или отображения (mapping), на рис. 2.2 явно не показаны (дальнейшее обсуждение этих вопросов будет продолжено в разделе 2.6).

В данном случае не имеет особого значения, является ли рассматриваемая система реляционной или какой-нибудь иной. Но было бы полезно вкратце рассказать, как эти три уровня архитектуры обычно реализуются именно в реляционных системах.

- Во-первых, концептуальный уровень в такой системе определенно будет реляционным в том смысле, что видимые на этом уровне объекты являются реляционными таблицами, а используемые операторы — реляционными операторами (включая операторы *выборки* строк и столбцов, кратко рассмотренные в главе 1).
- Во-вторых, каждое внешнее представление, как правило, также будет реляционным или достаточно близким к нему. Например, объявления записей в PL/I и COBOL, представленные на рис. 2.2, упрощенно можно считать аналогами объявлений таблиц в реляционной системе.

  Примечание. Следует иметь в виду, что термин *внешнее представление* (часто — просто *представление)* в реляционном контексте имеет, к сожалению, довольно специфический смысл, который *не* полностью совпадает со смыслом, приписанным ему в этой главе. Выяснению реляционного смысла данного термина и его обсуждению посвящены главы 3 и (особенно) 10.
- В-третьих, внутренний уровень *не* будет реляционным, поскольку объекты на этом уровне не будут реляционными (хранимыми) таблицами; наоборот, они будут объектами такого же типа, как и находящиеся на внутреннем уровне объекты любой другой системы (хранимые записи, указатели, индексы, хэшированные значения и т.п.). В действительности реляционная модель как таковая не позволяет узнать ничего существенного о внутреннем уровне. Она, как уже отмечалось в главе 1, имеет отношение лишь к тому, как воспринимает базу данных *пользователь.*

Теперь перейдем к более детальному исследованию трех уровней архитектуры, начиная с внешнего уровня. На рис. 2.3 показаны основные компоненты архитектуры и их взаимосвязь. На этот рисунок мы будем часто ссылаться в данной главе.

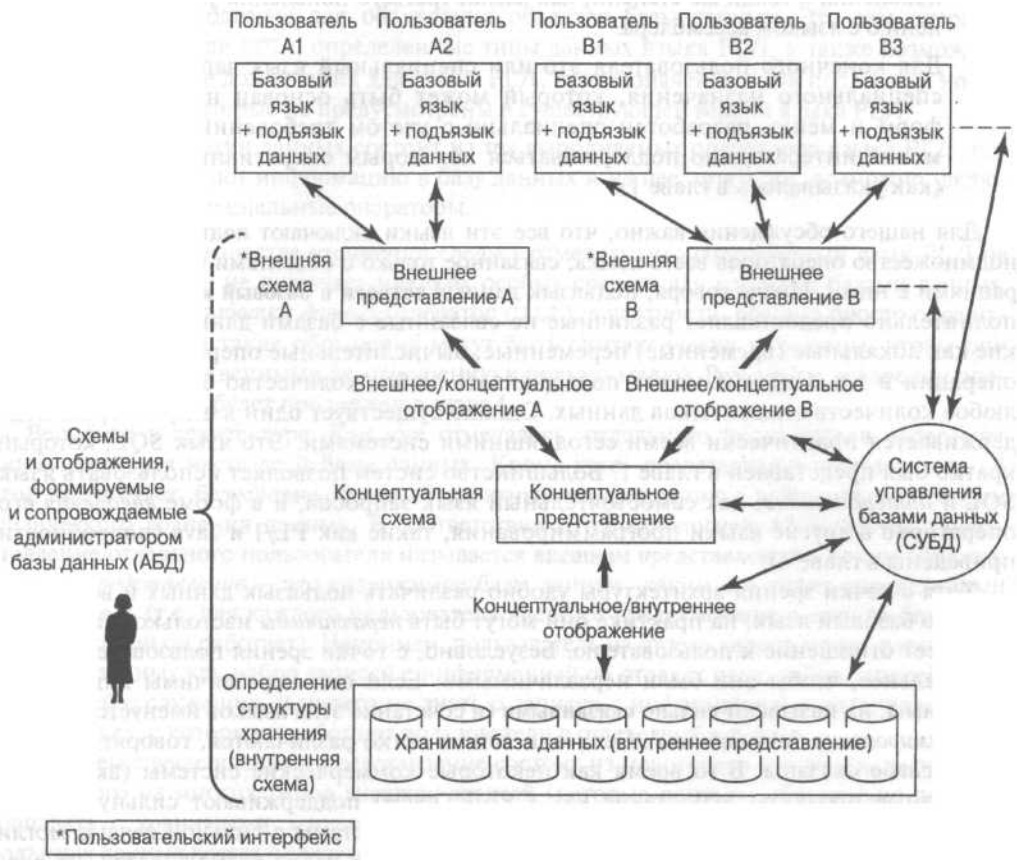


Рис. 2.3. Подробная схема архитектуры системы баз данных

2.3. ВНЕШНИЙ УРОВЕНЬ

Внешний уровень — это индивидуальный уровень пользователя. Как было сказано в главе 1, пользователь может быть прикладным программистом или конечным пользователем с любым уровнем профессиональной подготовки. Особое место среди пользователей занимает администратор базы данных (АБД). В отличие от остальных пользователей, АБД интересуют также концептуальный и внутренний уровни. Об этом еще будет сказано в следующих двух разделах.

У каждого пользователя есть свой **язык** для работы с СУБД.

- Для прикладного программиста это либо один из распространенных языков программирования (например, PL/I, C++ или Java), либо специальный язык рассматриваемой системы. Такие оригинальные языки называют *языками четвертого*

поколения на том (не вполне формальном!) основании, что машинный код, язык ассемблера и такие языки, как PL/I, можно считать языками трех первых "поколений", а оригинальные языки модернизированы по сравнению с языками третьего поколения в такой же степени, как языки третьего поколения улучшены по сравнению с языком ассемблера.

Для конечного пользователя это или специальный язык запросов, или язык специального назначения, который может быть основан на использовании форм и меню, разработан специально с учетом требований пользователя и может интерактивно поддерживаться некоторым оперативным приложением (как указывалось в главе 1).

Для нашего обсуждения важно, что все эти языки включают подъязык данных, т.е. подмножество операторов всего языка, связанное только с объектами баз данных и операциями с ними. Иначе говоря, подъязык данных встроен в базовый язык, который дополнительно предоставляет различные не связанные с базами данных средства, такие как локальные (временные) переменные, вычислительные операции, логические операции и т.д. Система может поддерживать любое количество базовых языков и любое количество подъязыков данных. Однако существует один язык, который поддерживается практически всеми сегодняшними системами. Это язык SQL, который кратко был представлен в главе 1. Большинство систем позволяет использовать язык SQL и *интерактивно,* как самостоятельный язык запросов, и в форме *внедрения* его операторов в другие языки программирования, такие как PL/I и Java (подробности приведены в главе 4).

Хотя с точки зрения архитектуры удобно различать подъязык данных и включающий его базовый язык, на практике они могут быть *неразличимы* настолько, насколько это имеет отношение к пользователю. Безусловно, с точки зрения пользователя предпочтительнее, чтобы они были неразличимыми. Если они неразличимы или трудно различимы, их называют сильно связанными (и сочетание этих языков именуется *языком программирования базы данных)*[2]. Если они ясно и легко различаются, говорят, что эти языки слабо связаны. В то время как некоторые коммерческие системы (включая и конкретные продукты SQL, такие как СУБД Oracle) поддерживают сильную связь, многие системы обеспечивают лишь слабую связь. Системы с сильной связью могли бы предоставить пользователю более унифицированный набор возможностей, но очевидно, что они требуют больше усилий со стороны системных проектировщиков и разработчиков, поэтому, вероятно, и сохраняется такое положение дел.

В принципе, любой подъязык данных является на самом деле комбинацией по крайней мере двух подчиненных языков — языка определения данных (Data Definition Language — DDL), который позволяет формулировать определения или объявления объектов базы данных, и языка манипулирования[3] данными (Data Manipulation Language — DML), который поддерживает операции с такими объектами или их обработку. Например, рассмотрим пользователя языка PL/I (см. рис. 2.2). Подъязык данных этого пользователя

---

[2] В этом смысле языком программирования базы данных вполне можно назвать язык Tutorial D, ко торый будет использоваться в следующих главах в качестве основы для примеров (см. примечания на эту тему в предисловии к данной книге).

[3] Термин "манипулирование" (буквально — выполнение действий вручную), не очень удачный, но получил широкое распространение и поэтому утвержден официально.

включает определенные средства языка PL/I, применяемые для организации взаимодействия с СУБД.

- Язык определения данных включает некоторые описательные структуры языка PL/I, необходимые для объявления объектов базы данных. Это сам оператор DECLARE (или DCL), определенные типы данных языка PL/I, а также возможные специальные дополнения для языка **PL/I,** предназначенные для поддержки новых объектов, которые не предусмотрены в существующей версии языка PL/I.
- Язык обработки данных состоит из тех выполняемых операторов языка PL/I, ко торые передают информацию в базу данных и из нее; опять же, возможно, вклю чая новые специальные операторы.

Примечание. В качестве уточнения следует отметить, что современный язык PL/I на самом деле вообще не включает никаких особых средств для работы с базами данных. Оператор *языка обработки данных* (оператор CALL), в частности, обычно просто обращается к СУБД (хотя такие обращения могут быть синтаксически упрощены, чтобы они стали более дружественными по отношению к пользователю). Разговор о внедрении операторов языка SQL будет продолжен в главе 4.

Вернемся к архитектуре. Как уже отмечалось, отдельного пользователя интересует лишь некоторая часть всей базы данных. Кроме того, представление пользователя об этой части будет, безусловно, более абстрактным по сравнению с выбранным способом физического хранения данных. В соответствии с терминологией ANSI/SPARC, представление отдельного пользователя называется **внешним представлением.** Таким образом, *внешнее представление* — это содержимое базы данных, каким его видит определенный пользователь (т.е. для каждого пользователя внешнее представление и *есть* та база данных, с которой он работает). Например, пользователь из отдела кадров может рассматривать базу данных как набор записей с информацией об отделах плюс набор записей с информацией о служащих, и ничего не знать о записях с информацией о материалах и их поставщиках, с которыми работают пользователи в отделе снабжения.

В общем случае внешнее представление состоит из некоторого множества экземпляров каждого из многих типов **внешних записей** (которые вовсе *не* обязательно должны совпадать с хранимыми записями)[4]. Предоставляемый в распоряжение пользователя подъязык данных всегда определяется в терминах внешних записей. Например, операция *выборки* языка обработки данных осуществляет выборку экземпляров внешних, а не хранимых записей. (Теперь становится очевидно, что термин *логическая запись,* употреблявшийся в главе 1, на самом деле относится к внешним записям. Поэтому в дальнейшем мы будем избегать его использования.)

Каждое внешнее представление определяется посредством **внешней схемы,** которая в основном состоит из определений записей каждого из типов, присутствующих в этом внешнем представлении (см. рис. 2.2). Внешняя схема записывается с помощью *языка*

[4] В данном случае предполагается, что вся информация на внешнем уровне представлена в форме записей. Но некоторые системы позволяют представлять информацию иначе: либо вместо записей, либо совместно с ними. Для использующих такие альтернативные методы систем все определения и пояснения этого раздела требуют соответствующих изменений. Это замечание касается также концептуального и внутреннего уровней. Детальное обсуждение подобных вопросов в этой части книги было бы преждевременным, поэтому мы вернемся к ним позднее, в главах 14 (в особенности — в разделе "Список литературы") и *25. См. также приложение А (где приведена подробная* информация о внутреннем уровне).

определения данных, являющегося подмножеством подъязыка данных пользователя. (Поэтому язык определения данных иногда называют **внешним** языком определения данных.) Например, тип внешней записи о работнике можно определить как шестисимвольное поле с номером работника, как поле из пяти десятичных цифр, предназначенное для хранения данных о его зарплате, и т.д. Кроме того, может потребоваться определить *отображение* между внешней и исходной *концептуальными* схемами (подробности — в следующем разделе). Это отображение рассматривается в разделе 2.6.

2.4. КОНЦЕПТУАЛЬНЫЙ УРОВЕНЬ

Концептуальное представление — это представление всей информации базы данных в несколько более абстрактной форме (как и в случае внешнего представления) по сравнению с описанием физического способа хранения данных. Однако концептуальное представление существенно отличается от представления данных какого-либо отдельного пользователя. Вообще говоря, концептуальное представление — это представление данных в том виде, какими они являются на самом деле, а не в том, какими их вынужден рассматривать пользователь в рамках, например, определенного языка или используемого аппаратного обеспечения.

Концептуальное представление состоит из некоторого множества экземпляров каждого из существующих типов **концептуальных записей.** Например, оно может состоять из набора экземпляров записей, содержащих информацию об отделах, набора экземпляров записей, содержащих информацию о поставщиках, набора экземпляров записей, содержащих информацию о материалах и т.д. Концептуальная запись вовсе не обязательно должна совпадать с внешней записью, с одной стороны, и с хранимой записью — с другой.

Концептуальное представление определяется с помощью **концептуальной** схемы, включающей определения для каждого существующего типа концептуальных записей (см. рис. 2.2). Концептуальная схема использует другой язык определения данных — **концептуальный.** Чтобы добиться независимости от данных, нельзя включать в определения концептуального языка какие-либо указания о структурах хранения или методах доступа. Определения концептуального языка должны относиться *только* к содержанию информации. Это означает, что в концептуальной схеме не должно быть никакого упоминания о представлении хранимого файла, упорядоченности хранимых записей, индексировании, хэш-адресации, указателях или других подробностях хранения данных или доступа к ним. Если концептуальная схема действительно обеспечивает независимость от данных в этом смысле, то внешние схемы, определенные на основе концептуальной (раздел 2.6), *заведомо* будут обеспечивать независимость от данных.

Концептуальное представление — это представление всего содержимого базы данных, а концептуальная схема — это определение такого представления. Однако было бы ошибкой полагать, что концептуальная схема представляет собой не более чем набор определений, весьма напоминающих простые определения записей в программе на языке COBOL (или каком-либо другом языке). Определения в концептуальной схеме могут характеризовать большое количество различных дополнительных аспектов обработки данных, например таких, как ограничения защиты или требования поддержки целостности данных, упомянутые в главе 1. Более того, некоторые авторитетные специалисты предлагают в качестве конечной цели создания концептуальной схемы рассматривать

описание всего предприятия — не только самих его данных, но и того, как эти данные используются, как они перемещаются внутри предприятия, для чего используются в каждом конкретном месте, какая проверка и иные типы контроля применяются к ним в каждом отдельном случае и т.д. [2.3]. Однако необходимо подчеркнуть, что ни одна сегодняшняя система реально не поддерживает такого концептуального уровня, который хотя бы немного приблизился к указанной выше степени развития[5]. В большинстве существующих систем концептуальная схема в действительности представляет собой нечто, что лишь немного больше простого объединения всех независимых внешних схем с привлечением дополнительных средств защиты и поддержкой правил обеспечения целостности. Вероятно, со временем системы станут гораздо "интеллектуальнее" с точки зрения поддержки концептуального уровня.

2.5. ВНУТРЕННИЙ УРОВЕНЬ

Третьим уровнем архитектуры является внутренний уровень. **Внутреннее представление** — это низкоуровневое представление всей базы данных как базы, состоящей из некоторого множества экземпляров каждого из существующих типов **внутренних записей.** Термин *внутренняя запись* относится к терминологии ANSI/SPARC и означает конструкцию, иначе называемую *хранимой* записью (в дальнейшем мы будем использовать именно этот термин). Внутреннее представление, так же как внешнее и концептуальное, отделено от физического уровня, поскольку в нем не рассматриваются физические записи, обычно называемые **блоками** или **страницами,** и физические области устройства хранения, такие как цилиндры и дорожки. Другими словами, внутреннее представление предполагает наличие бесконечного *линейного адресного пространства.* Особенности методов отображения этого адресного пространства на физические устройства хранения в значительной степени зависят от используемой операционной системы и по этой причине не включены в общую архитектуру. Следует отметить, что блоки (или страницы) **устройства ввода—вывода** — это количество данных, передаваемых из вторичной памяти (памяти накопителя) в основную (оперативную) память за одну операцию ввода-вывода. Обычно, страницы имеют размер от 1 Кбайт и выше, но не больше 64 Кбайт (1 Кбайт = 1024 байт).

Внутреннее представление описывается с помощью **внутренней схемы,** которая определяет не только различные типы хранимых записей, но также существующие индексы, способы представления хранимых полей, физическую упорядоченность хранимых записей и т.д. (Соответствующий простой пример также приведен на рис. 2.2; см. также приложение Г.) Внутренняя схема формируется с использованием еще одного языка определения данных — **внутреннего.**

Примечание. В этой книге вместо терминов *внутреннее представление* и *внутренняя схема* обычно будут использоваться интуитивно более понятные термины *хранимая структура* (или *хранимая база данных)* и *определение структуры хранения,* соответственно.

В заключение отметим, что в некоторых исключительных ситуациях прикладные программы, в частности те из них, которые называются *утилитами* (о чем подробнее будет рассказано в разделе 2.11), могут выполнять операции непосредственно на внутреннем, а не на внешнем уровне. Конечно, использовать такую практику не рекомендуется, поскольку она связана с определенным риском с точки зрения защиты (игнорируются

[5] Многие считают, что к этой цели ближе подошли так называемые системы, основанные на использовании делового регламента, или бизнес-правил (см. главы 9 и 14).

правила защиты) и сохранения целостности данных (правила целостности также игнорируются). К тому же такая программа будет зависеть от определения обрабатываемых данных. Однако иногда подобный подход может быть единственным способом реализации требуемой функции или достижения необходимого быстродействия (иногда по аналогичным причинам приходится обращаться к средствам языка ассемблера пользователю языка высокого уровня).

2.6. ОТОБРАЖЕНИЯ

Представленная на рис. 2.3 архитектура, кроме элементов самих трех уровней, включает определенные **отображения:** отображение концептуального уровня на внутренний и несколько отображений внешних уровней на концептуальный.

- Отображение *"концептуальный-внутренний"* устанавливает соответствие между концептуальным представлением и хранимой базой данных, т.е. описывает, как концептуальные записи и поля представлены на внутреннем уровне. При измене нии структуры хранимой базы данных (т.е. при внесении изменений в определение структуры хранения) отображение "концептуальный—внутренний" также изменя ется, с учетом того, что концептуальная схема остается неизменной. (Осуществле ние подобных изменений входит в обязанности администратора базы данных и может обеспечиваться самой СУБД.) Иначе говоря, чтобы обеспечивалась незави симость от данных, результаты внесения любых изменений в схему хранения не должны обнаруживаться на концептуальном уровне.
- Отображение *"внешний—концептуальный"* определяет соответствие между неко торым внешним представлением и концептуальным представлением. В целом, различия, которые могут существовать между этими двумя уровнями, подобны различиям между концептуальным представлением и хранимой базой данных. Например, данные полей могут относиться к разным типам, названия полей и за писей могут быть изменены, несколько концептуальных полей могут быть объе динены в одно (виртуальное) внешнее поле и т.д. В одно и то же время допустимо существование любого количества внешних представлений, причем одно и то же внешнее представление может принадлежать нескольким пользователям, а разные внешние представления — перекрываться.

  Очевидно, что отображение "концептуальный—внутренний" служит основой *физической* независимости от данных, а отображения "внешний—концептуальный" являются ключом к *логической* независимости от данных. Как было показано в главе 1, система обеспечивает **физическую** независимость от данных [1.3], если пользователи и пользовательские программы обладают невосприимчивостью к изменениям в *физической* структуре хранимой базы данных. Аналогично, система обеспечивает **логическую** независимость от данных [1.4], если пользователи и пользовательские программы обладают невосприимчивостью к изменениям в *логической* структуре базы данных (подразумеваются изменения на концептуальном или "общем логическом" уровне). Этот важный вопрос будет обсуждаться в главах 3 и 10.
- Следует отметить, что большинство систем позволяет выражать одно определение внешнего представления через другое (по существу, с помощью отображения *"внешний-внешний"),* не требуя обязательного явного определения отображения

каждого внешнего представления на концептуальный уровень. Эта возможность очень полезна, если несколько представлений подобны друг другу. В частности, аналогичная возможность предусмотрена во многих реляционных СУБД.

2.7. АДМИНИСТРАТОР БАЗЫ ДАННЫХ

Как уже отмечалось в главе 1, администратор *данных* (АД) — это человек, отвечающий за стратегию и политику принятия решений, связанных с данными предприятия, а администратор *базы данных* (АБД) — это человек, обеспечивающий необходимую техническую поддержку для реализации принятых решений. Таким образом, АБД отвечает за общее управление системой на техническом уровне. Теперь опишем функции АБД более подробно.

- *Определение концептуальной схемы*

  Администратор *данных* решает, какая именно информация должна храниться в базе данных, т.е. указывает те типы сущностей, в которых заинтересовано предприятие, а также определяет, какую информацию об этих сущностях необходимо вводить в базу данных. Этот процесс обычно называют логическим (или *концептуальным)* проектированием базы данных. После того как содержимое базы данных на абстрактном уровне будет определено администратором данных, АБД создает соответствующую концептуальную схему, используя концептуальный язык определения данных. Объектная (откомпилированная) форма этой схемы будет использоваться в СУБД для получения ответов на запросы, связанные с доступом к данным. Исходная (не откомпилированная) форма будет играть роль справочного документа для пользователей системы.

  Следует отметить, что на практике редко все происходит точно по описанной выше схеме. Иногда концептуальную схему создает сам АД, а логическим проектированием занимается АБД.

- *Определение внутренней схемы*

  Администратор базы данных должен также решить, как данные будут представлены в хранимой базе данных. Этот процесс обычно называют физическим проектированием базы данных. После завершения физического проектирования АБД создает соответствующие структуры хранения, используя внутренний язык определения данных (т.е. описывает внутреннюю схему). Кроме того, он определяет соответствующее отображение между внутренней и концептуальной схемами. На практике концептуальный и внутренний языки определения данных (особенно первый) могут включать в себя средства определения этого отображения, однако две данные функции (создание схемы и определение отображений) должны быть четко разделены. Как и концептуальная схема, внутренняя схема и соответствующее отображение будут существовать в исходной и объектной формах.

  Обратите внимание на то, что проектирование осуществляется в определенной последовательности — вначале необходимо установить, какие данные требуются, а затем решить, как следует представить их в памяти. Физическое проектирование выполняется *после* логического.

- *Взаимодействие с пользователями*

  В обязанности АБД входит также взаимодействие с пользователями для предоставления им необходимых данных и подготовки требуемых внешних схем (или оказания помощи пользователям в их подготовке) с применением прикладного внешнего языка определения данных. (Как уже отмечалось, СУБД может поддерживать несколько различных внешних языков определения данных.) Кроме того, необходимо определить отображение между каждой созданной внешней и концептуальной схемами. На практике внешний язык определения данных может включать средства формирования этого отображения, но, опять же, схемы и отображения должны быть четко разделены между собой. Каждая внешняя схема и соответствующее ей отображение будут существовать в исходной и объектной формах.

  Другие аспекты взаимодействия с пользователями включают консультации по разработке приложений, обеспечение требуемого технического обучения, предоставление помощи в выявлении и устранении возникающих проблем и прочие виды связанного с системой профессионального обслуживания.

- *Определение требований защиты и обеспечения целостности данных*

  Как уже отмечалось, требования защиты и поддержки целостности данных рассматриваются как часть определения концептуальной схемы. Концептуальный язык определения данных должен включать в себя средства описания этих требований.

- *Определение процедур резервного копирования и восстановления*

  После того как предприятие доверит хранение своих данных системе баз данных, оно станет полностью зависимым от бесперебойного функционирования этой системы. В случае повреждения какой-либо части базы данных вследствие ошибки человека, отказа оборудования или сбоя программ операционной системы очень важно иметь возможность восстановить утраченные данные с минимальной задержкой и с наименьшим воздействием на остальную часть системы. В идеале, данные, которые *не* были повреждены, совсем не должны затрагиваться в процессе восстановления. АБД должен разработать и реализовать, во-первых, подходящую схему восстановления, включающую периодическую выгрузку базы данных на устройство резервного копирования (получение *дампа),* а во-вторых, процедуры загрузки базы данных из последнего созданного дампа в случае необходимости.

  Помимо всего прочего, требование быстрого восстановления поврежденных данных является одной из тех причин, по которым желательно организовать хранение данных не в каком-либо одном месте, а распределить их по нескольким отдельным базам данных. Каждая из таких баз данных будет представлять собой оптимальный объект выгрузки или перезагрузки. В этой связи отметим, что уже существуют *терабайтовые системы*[6] (т.е., грубо говоря, коммерческие системы, хранящие больше триллиона байтов данных), а в будущем системы станут

[6] 1024 байт - 1 Кбайт (килобайт); 1024 Кбайт = 1 Мбайт (мегабайт); 1024 Мбайт = 1 Гбайт (гигабайт); 1024 Гбайт = 1 Тбайт (терабайт); 1024Тбайт = 1 Пбайт (петабайт); 1024 Пбайт = 1 Эбайт (эксабайт); 1024 Эбайт = 1 Дбайт (дзетабайт); 1024 Дбайт = 1 Йбайт (йотабайт). Кстати, вопреки распространенному мнению, первый слог в английском слове *gigabyte* является открытым и произносится с мягким начальным *g* (как в слове "gigantic").

еще более крупными. Понятно, что такие системы *очень больших баз данных* (Very Large DataBase — VLDB) требуют тщательного и продуманного администрирования, в особенности если необходимо обеспечить для пользователей постоянный доступ к базе данных (а часто именно так и бывает). Однако для простоты рассуждений будем по-прежнему подразумевать, что мы имеем дело с единственной базой данных.

Управление производительностью и реагирование на изменяющиеся требования

Как отмечалось выше, в главе 1, АБД отвечает за такую организацию системы, при которой можно поддерживать производительность, оптимальную для всего предприятия в целом, а также за корректировку работы системы (т.е. ее **настройку)** в соответствии с изменяющимися требованиями. Например, может возникнуть необходимость в периодической **реорганизации** хранимой базы данных для обеспечения того, чтобы производительность системы всегда поддерживалась на приемлемом уровне. Как уже упоминалось, любые изменения на уровне физического хранения данных (внутреннем уровне) должны сопровождаться соответствующими изменениями в определении его отображения на концептуальный уровень, что позволит сохранить концептуальную схему неизменной.

Конечно, перечисленное выше — отнюдь не исчерпывающий список обязанностей АБД, а лишь попытка высказать некоторые соображения об их существе и охвате.

2.8. СИСТЕМА УПРАВЛЕНИЯ БАЗОЙ ДАННЫХ

Система управления базой данных (СУБД) представляет собой программное обеспечение, которое управляет всем доступом к базе данных. Концептуально это происходит следующим образом (см. рис. 2.3).

1. Пользователь выдает запрос на доступ к данным, применяя определенный подъя зык данных (обычно это язык SQL).
2. СУБД перехватывает этот запрос и анализирует его.
3. СУБД просматривает внешнюю схему (ее объектную версию) для этого пользователя, соответствующее отображение "внешний—концептуальный", концептуальную схему, отображение "концептуальный—внутренний" и определения структур хранения.
4. СУБД выполняет необходимые операции в хранимой базе данных.

В качестве примера рассмотрим, как осуществляется выборка экземпляра определенной внешней записи. В общем случае поля этой записи будут выбираться из нескольких экземпляров концептуальных записей, которые в свою очередь будут запрашивать поля из нескольких экземпляров хранимых записей. Концептуально СУБД сначала должна выбрать все требуемые экземпляры хранимых записей, затем сформировать требуемые экземпляры концептуальных записей и после этого создать экземпляр внешней записи. На каждом этапе могут потребоваться преобразования типов данных или другие преобразования.

Конечно, описание предыдущего примера весьма упрощено; в частности, в данном случае предполагается, что весь процесс основан на *интерпретации* (т.е. анализ запроса, проверка различных схем и другие процедуры осуществляются непосредственно во время

выполнения). Однако интерпретация обычно характеризуется невысокой производительностью, поскольку на ее выполнение затрачивается много времени. На практике обычно существует возможность предварительно *откомпилировать* запрос на доступ к данным до начала его выполнения; в частности, в современных системах SQL применяется именно такой подход (см. аннотации к [4.131 и [4.271 в главе 4).

Теперь рассмотрим функции СУБД немного подробнее. Они обязательно будут включать поддержку работы компонентов базы данных, показанных на рис 2.4

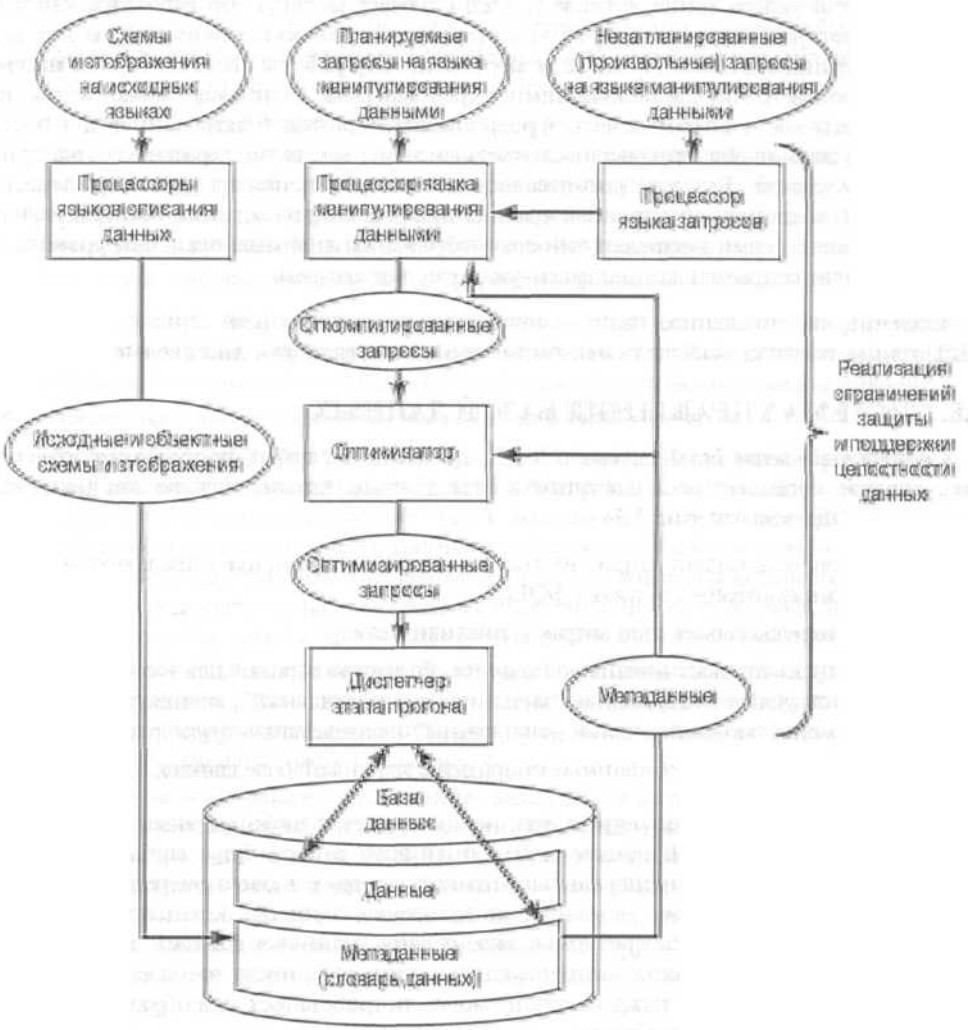


Рис. 2.4. Основные функции и компоненты типичной СУБД

- *Определение данных*

  СУБД должна предоставлять средства определения данных в виде исходной формы (внешних схем, концептуальной схемы, внутренней схемы, а также всех необходимых отображений) и преобразования этих определений в соответствующую объектную форму. Иначе говоря, СУБД должна включать в себя компоненты **процессора ЯОД** (языка определения данных) или **компилятора ЯОД** для каждого из поддерживаемых ею языков определения данных. СУБД должна также правильно трактовать синтаксис языка определения данных, чтобы ей можно было, например, сообщить, что внешние записи EMPLOYEE включают поле SALARY. Эту информацию СУБД должна использовать при анализе и выполнении запросов обработки данных (таких как "Найти всех служащих с зарплатой, составляющей меньше 50 тыс. долл".).

- *Манипулирование данными*

  СУБД должна быть способна обрабатывать запросы пользователя на выборку, изменение или удаление данных, уже существующих в базе, или на добавление в нее новых данных. Другими словами, СУБД должна включать в себя компонент **процессора ЯМД** или **компилятора ЯМД,** обеспечивающего поддержку *языка манипулирования данными (ЯМД).*

  В целом, запросы ЯМД подразделяются на планируемые и непланируемые.

  а) **Планируемый запрос** — это запрос, необходимость выполнения которого была предусмотрена заранее. Администратор базы данных, возможно, должен будет разработать физический проект базы данных таким образом, чтобы гарантиро вать достаточное быстродействие выполнения подобных запросов.

  б) **Непланируемый запрос** — это, наоборот, некоторый *произвольный* запрос на вы борку или (что менее вероятно) на обновление, необходимость выполнения ко торого не была предусмотрена заранее и возникла по какой-то особой причине. Физический проект базы данных может обеспечивать успешное выполнение кон кретного произвольного запроса, или может оказаться не совсем подходящим.

  Согласно терминологии, введенной в главе 1 (раздел 1.3), планируемые запросы более характерны для операционных, или производственных приложений, а непланируемые — для приложений поддержки принятия решений. Более того, планируемые запросы обычно поступают из заранее подготовленных приложений, а непланируемые запросы по определению вводятся интерактивно, с помощью *процессора языка запросов.* (В главе 1 уже отмечалось, что на самом деле процессор языка запросов — это встроенное интерактивное приложение, а не какая-то часть *самой* СУБД. Мы показали этот компонент на рис. 2.4 исключительно ради создания полной картины.)

- *Оптимизация и выполнение*

  Запросы ЯМД, планируемые или непланируемые, должны быть обработаны таким компонентом, как **оптимизатор,** назначение которого состоит в поиске достаточно эффективного способа выполнения каждого из запросов[7]. Оптимизация подробно

[7] Во всей данной книге термин "оптимизация" относится исключительно к оптимизации запросов ЯМД, если явно не указано иное.

обсуждается в главе 18. Затем оптимизированные запросы выполняются под управлением **диспетчера этапа прогона** (run-time manager). На практике диспетчер этапа прогона для получения доступа к хранимым данным, возможно, будет использовать что-то подобное *диспетчеру доступа к файлам.* (Последний компонент кратко обсуждается в конце данного раздела.)

- *Защита и поддержка целостности данных*

  СУБД должна контролировать пользовательские запросы и пресекать любые попытки нарушения ограничений защиты и целостности данных, определенные АБД (см. предыдущий раздел). Этот контроль может осуществляться во время компиляции, во время выполнения или на обоих этих этапах обработки запроса.

- *Восстановление данных и поддержка параллельности*

  СУБД или другой связанный с ней программный компонент, обычно называемый **диспетчером транзакций** или **диспетчером выполнения транзакций,** должен обеспечивать необходимый контроль над восстановлением данных и управление параллельной обработкой. Детали реализации этих функций системы выходят за рамки данной главы — подробное обсуждение указанных тем можно найти в части IV. Диспетчер транзакций не показан на рис. 2.4, поскольку обычно он не является частью *самой* СУБД.

- *Словарь данных*

  СУБД должна поддерживать функцию ведения **словаря данных.** Сам словарь данных вполне можно считать самостоятельной базой данных (но не пользовательской, а системной). Словарь содержит "данные о данных" (иногда называемые **метаданными** или **дескрипторами),** т.е. в нем находятся *определения* других объектов системы, а не просто обычные данные. В частности, в словаре данных должны быть записаны исходные и объектные формы всех существующих схем (внешних, концептуальной и т.д.) и отображений, а также установленные ограничения защиты и целостности данных. Расширенный словарь может также включать большой объем дополнительной информации, например, о том, какие из программ используют ту или иную часть базы данных, какие отчеты требуются каждому из пользователей и т.д. Словарь может быть (а фактически просто *должен* быть) интегрирован в определяемую им базу данных, а значит, должен содержать описание самого себя. Конечно, должна существовать возможность обращения с запросами и к словарю, как и к любой другой базе данных, например, для того, чтобы можно было узнать, какие программы и/или пользователи будут затронуты при предполагаемом внесении изменения в систему. Дальнейшее обсуждение этого вопроса приведено в главе 3.

  Следует отметить, что здесь мы коснулись области, в которой может возникнуть путаница из-за несовершенства используемой терминологии. То, что мы называем *словарем* (dictionary), иногда называют *справочником* (directory) или *каталогом* (catalog), потому что справочники и каталоги считаются в некотором смысле менее важными по сравнению с настоящими словарями, а термин *словарь* применяют для обозначения специального (более важного) вида инструментов разработки приложений. Также встречаются и другие термины — *репозитарий данных* (глава 14) и *энциклопедия данных.*

- *Производительность*

  Очевидно, что СУБД должна выполнять все указанные функции с максимально возможной эффективностью.

Итак, можно сделать вывод, что в общем назначением СУБД является предоставление **пользовательского интерфейса** к системе баз данных. Пользовательский интерфейс может быть определен как существующий в системе ограничительный уровень, ниже которого для пользователя все остается невидимым. Следовательно, по определению пользовательский интерфейс находится на *внешнем* уровне. Тем не менее, в главе 10 будет показано, что иногда внешнее представление лишь незначительно отличается от соответствующей ему части основного концептуального представления (по крайней мере, в современных коммерческих продуктах SQL).

В заключение вкратце сопоставим описанную здесь типовую СУЬД и систему управления файлами (сокращенно называемую также *диспетчером файлов* или *файловым сервером).* В своей основе **система управления файлами** является компонентом базовой операционной системы, предназначенной для управления хранимыми файлами. Проще говоря, она расположена "ближе к диску", чем СУБД. (В действительности, как описано в приложении Г, СУБД обычно строится на базе некоторого варианта диспетчера файлов.) Таким образом, пользователь системы управления файлами может создавать и уничтожать хранимые файлы, а также выполнять простые операции выборки и обновления хранимых записей в созданных им файлах. Однако, в сравнении с СУБД, системе управления файлами свойственны следующие недостатки.

- Система управления файлами не имеет никаких сведений о внутренней структуре хранимых записей и, следовательно, не способна обрабатывать запросы, требую щие знания этой структуры.
- Как правило, подобные системы предоставляют слабую поддержку ограничений защиты и поддержки целостности данных или же вообще ее не предоставляют.
- Обычно такие системы предоставляют недостаточную поддержку управления вос становлением данных и параллельным доступом к ним или же не предоставляют ее вовсе.
- На уровне диспетчера файлов не существует концепции настоящего словаря данных.
- Вообще говоря, эти системы обеспечивают независимость от данных гораздо хуже по сравнению с СУБД.
- Как правило, отдельные файлы не являются "интегрированными" или "разделяемыми" в том смысле, который вкладывается в эти понятия применительно к базам данных. (Обычно они являются собственными файлами пользователя или приложения.)

2.9. СИСТЕМА УПРАВЛЕНИЯ ПЕРЕДАЧЕЙ ДАННЫХ

В этом разделе вкратце рассматривается **передача данных.** Запрос конечного пользователя к базе данных в действительности передается от рабочей станции пользователя (которая может быть физически удалена от самой системы баз данных) к некоторому интерактивному приложению (встроенному или нет), а от него — к СУБД в форме *коммуникационных сообщений.* Более того, ответы пользователю также передаются в форме

подобных сообщений (от СУБД или оперативного приложения к станции пользователя). Передача таких сообщений происходит под управлением **еще** одного программного компонента — **диспетчера передачи данных.**

Диспетчер передачи данных не является частью СУБД; он представляет собой автономную систему со своими правами. Однако, поскольку очевидно, что от диспетчера передачи данных и СУБД требуется согласованная совместная работа, они иногда рассматриваются как равноправные компоненты программного продукта более высокого уровня, называемого **системой базы данных и передачи данных** (DataBase/Data-Communications — DB/DC). В такой системе СУБД отвечает за работу с базой данных, а диспетчер передачи данных обрабатывает все сообщения, поступающие в СУБД и из нее (точнее, поступающие в то приложение, в котором используется СУБД, и из него). Однако в этой книге мы можем рассмотреть лишь относительно небольшой объем информации об обработке сообщений (поскольку такая обширная тема вполне заслуживает самостоятельного обсуждения). В разделе 2.12 кратко рассматриваются вопросы передачи данных *между отдельными системами* (т.е. между отдельными компьютерами в сети передачи данных, подобной Internet), но это действительно отдельная тема.

2.10. АРХИТЕКТУРА "КЛИЕНТ/СЕРВЕР"

В предыдущих разделах этой главы подробно обсуждалась так называемая *архитектура ANSI/SPARC* для систем баз данных. Ее упрощенная схема приведена на рис. 2.3. В настоящем разделе мы будем изучать базы данных с несколько иной точки зрения.

Основным назначением систем баз данных, безусловно, является поддержка разработки и выполнения приложений баз данных. Поэтому на высоком уровне систему баз данных можно рассматривать как систему с очень простой структурой, состоящей из описанных ниже двух частей — **сервера** *(внутреннего компонента,* или *машины базы данных)* и **клиентов** *(внешних компонентов,* или *внешних интерфейсов),* как показано на рис. 2.5.

1. **Сервер—** это сама СУБД. Он поддерживает все основные функции СУБД, которые обсуждались в разделе 2.8, а именно: определение данных, манипулирование данными, защиту данных, поддержание целостности данных и т.д. В частности, он предоставляет полную поддержку внешнего, концептуального и внутреннего уровней, обсуждавшихся в разделе 2.8. Поэтому *сервер* в этом контексте — это просто другое название для СУБД.
2. **Клиенты** — это различные приложения, которые выполняются с помощью СУБД. Таковыми являются как приложения, написанные пользователями, так и встроенные приложения, предоставляемые поставщиками СУБД или некоторыми сторонними поставщиками программного обеспечения. Конечно, в самом сервере разница между встроенными приложениями и приложениями, написанными пользователем, не обнаруживается, поскольку все эти приложения используют один и тот же интерфейс сервера, а именно интерфейс внешнего уровня, который уже рассматривался в разделе 2.3. (Некоторые специальные "служебные" приложения могут оказаться исключениями. Как упоминалось выше, в разделе 2.5, они иногда работают непосредственно на *внутреннем* уровне системы. Подобные утилиты правильнее относить к встроенным компонентам СУБД, а не к приложениям в обычном смысле. В следующем разделе утилиты обсуждаются более подробно.)

Рис. 2.5. Архитектура "клиент/сервер"

Приложения, в свою очередь, делятся на несколько четко определенных категорий.

- **Приложения, написанные пользователями. В** основном, это обычные прикладные программы, чаще всего написанные либо на языке программирования общего назначения, таком как C++ или COBOL, либо на одном из специализированных языков четвертого поколения, хотя в обоих случаях эти языки должны как-то связываться с соответствующим подъязыком данных (см. раздел 2.3).
- **Приложения, предоставляемые поставщиками** (часто называемые **инструментальными средствами). В** целом, назначение таких средств — упрощать процесс создания и выполнения других приложений, т.е. приложений, которые разрабатываются специально для решения некоторой конкретной задачи. Часто эти приложения могут выглядеть вовсе не так, как приложения в общепринятом смысле. И это понятно, поскольку само назначение инструментальных средств состоит в предоставлении пользователям, особенно конечным, возможности создавать приложения *без* написания традиционных программ. Например, одно из предоставляемых поставщиком СУБД инструментальных средств может быть *генератором отчетов,* с помощью которого конечный пользователь сможет получить отформатированный отчет, выполнив обычный запрос к системе. Каждый такой запрос является, по существу, не чем иным, как небольшим специальным приложением, написанным на языке очень высокого уровня (с конкретным назначением), а именно — на *языке формирования отчетов.*

  Отдельно поставляемые инструментальные средства, в свою очередь, делятся на несколько самостоятельных классов.

  а) Процессоры языков запросов.

  б) Генераторы отчетов.

в) Графические бизнес-подсистемы.

г) Электронные таблицы.

д) Процессоры команд на естественных языках.

е) Статистические пакеты.

ж) Средства управления копированием или средства извлечения данных, з) Генераторы приложений (включая процессоры языков четвертого поколения).

и) Другие средства разработки приложений, включая CASE-инструменты (CASE, или Computer-Aided Software Engineering — автоматизированное проектирование и создание программ) и т.д.

к) Инструментальные средства интеллектуального анализа данных и визуализации.

Подробное обсуждение этих приложений выходит за рамки данной книги, однако следует отметить, что (как утверждалось ранее) главная задача системы баз данных — поддержка создания и выполнения приложений. Поэтому качество имеющихся клиентских инструментальных средств должно быть главным фактором при выборе СУБД, наиболее подходящей для конкретного заказчика. Другими словами, характеристики *самой* СУБД — не единственный и не важнейший фактор, который нужно учитывать в этом случае.

Завершим настоящий раздел ссылкой на последующий материал. Так как система в целом может быть четко разделена на две части (сервер и клиенты), появляется возможность организовать работу этих двух частей *на разных компьютерах.* Иначе говоря, существует возможность организации распределенной обработки.

Распределенная обработка предполагает, что отдельные компьютеры можно соединить какой-нибудь *коммуникационной сетью* так, чтобы выполнение одной задачи обработки данных можно было распределить на несколько компьютеров этой сети. Как показала практика, эта возможность настолько заманчива по различным соображениям (главным образом, экономическим), что термин "клиент/сервер" стал применяться почти исключительно в тех случаях, когда клиенты и сервер действительно находятся на разных компьютерах. Более подробно распределенная обработка данных рассматривается ниже, в разделе 2.12.

2.11. УТИЛИТЫ

Утилиты — это программы, разработанные для администратора базы данных и используемые им при решении различных административных задач. Как упоминалось выше, некоторые утилиты выполняются на внешнем уровне системы и потому представляют собой не что иное, как приложения специального назначения. Одни из них могут быть созданы даже не поставщиком данной СУБД, а какими-то сторонними разработчиками. Однако другие утилиты выполняются непосредственно на внутреннем уровне (т.е. действительно являются частью сервера) и поэтому должны предоставляться поставщиками СУБД.

Ниже приводится несколько примеров утилит различных типов, которые часто применяются на практике.

- Инструменты **загрузки,** применяемые для создания исходной версии базы данных из одного или нескольких файлов операционной системы.
- Инструменты **выгрузки—перезагрузки,** применяемые для выгрузки базы данных или ее части, создания резервных копий хранимых данных и восстановления базы данных из этих копий. (Безусловно, инструменты перезагрузки по сути идентичны упомянутым выше инструментам загрузки.)
- Инструменты **реорганизации,** применяемые для перераспределения данных в хранимой базе данных (обычно в целях повышения производительности). В частности, это может быть процедура группирования данных на диске не которым специальным образом или освобождения пространства, прежде за нятого данными, которые больше не используются.
- **Статистические** инструменты, применяемые для вычисления различных статистиче ских показателей и показателей производительности, таких как сведения о разме рах файлов или значениях данных, о количестве операций ввода-вывода и т.п.
- Инструменты **анализа,** применяемые для анализа упомянутой выше статистиче ской информации.

Этот список охватывает лишь небольшую часть функциональных возможностей, обычно предоставляемых доступными утилитами. Помимо перечисленных, существует множество других функций.

2.12. РАСПРЕДЕЛЕННАЯ ОБРАБОТКА

Как было указано в разделе 2.10, средства *распределенной обработки* позволяют соединить отдельные компьютеры в коммуникационную сеть (такую как Internet) для организации совместного решения общей задачи обработки данных на нескольких компьютерах сети. (Термин *параллельная обработка* используется практически в том же смысле, но в *параллельных* системах взаимодействующие компьютеры должны быть расположены физически достаточно близко, тогда как для распределенной системы это вовсе не обязательно, и отдельные компьютеры могут быть удалены на большие расстояния.) Взаимодействие между компьютерами осуществляется с помощью специального программного обеспечения, предназначенного для управления сетью, которое может представлять собой некоторое расширение диспетчера передачи данных (см. раздел 2.9), но чаще всего применяется в виде отдельного программного компонента.

Организация распределенной обработки может быть весьма разнообразной и осуществляется на разных уровнях. Как отмечалось в разделе 2.10, один из простейших случаев, представленный на рис. 2.6, — это случай, когда сервер СУБД запускается на одном компьютере, а клиентское приложение — на другом.

Как уже отмечалось в конце раздела 2.10, термин "клиент/сервер", несмотря на то что он, строго говоря, касается исключительно архитектуры приложений, фактически чаще всего служит для обозначения структуры, в которой клиент и сервер запускаются на разных компьютерах (см. рис. 2.6). И действительно, существует множество доводов в пользу подобной схемы.

- Главным доводом является возможность организации параллельной обработки. В этом случае для решения общей задачи применяются сразу несколько компьютеров, поэтому работа сервера (базы данных) и клиента (приложения) осуществляется

параллельно. В результате улучшаются такие показатели, как скорость обработки запросов в системе и производительность системы.

■ Кроме того, *серверный компьютер* может быть изготовлен по специальному заказу и специально приспособлен для работы с СУБД ("компьютер для базы данных"). Такое решение позволяет дополнительно повысить производительность. ■ *Клиентский компьютер* также может представлять собой персональную рабочую станцию, максимально приспособленную к потребностям конкретного конечного пользователя, что позволит предоставить ему наиболее удобный интерфейс и гарантировать высокий уровень готовности, быструю реакцию системы и другие дополнительные удобства при использовании.

Рис. 2.6. Вариант распределенной обработки, в котором клиент и сервер запускаются на разных компьютерах

■ К одному и тому же серверному компьютеру могут иметь доступ несколько разных клиентских компьютеров (что чаще всего и происходит). Поэтому одна база данных может совместно использоваться несколькими различными клиентскими системами, как показано на рис. 2.7.

К сказанному выше можно добавить, что размещение сервера и клиента (клиентов) на отдельных компьютерах соответствует принципам функционирования многих субъектов хозяйственной деятельности. Типичный способ организации работы отдельных предприятий (например, банков) заключается в том, что используется много компьютеров, причем данные для одной части предприятия сохраняются на одних компьютерах, а данные для другой части — на других. Несомненно, что пользователям одного компьютера (пусть лишь изредка) обязательно потребуется доступ к данным, хранящимся на другом

компьютере. В примере с банком можно с высокой степенью вероятности утверждать, что пользователям одного отделения банка иногда потребуется доступ к данным, хранящимся в другом его отделении. На основании этого можно сделать вывод, что на клиентских компьютерах могут храниться пользовательские данные, а на серверном компьютере могут эксплуатироваться собственные приложения. Поэтому, вообще говоря, каждый компьютер будет выступать в роли сервера для одних пользователей и в роли клиента для других, образуя систему, представленную на рис. 2.8. Иными словами, в этом случае каждый компьютер будет поддерживать *полную систему баз данных* (в том смысле, который вкладывался в это понятие в предыдущих разделах главы).

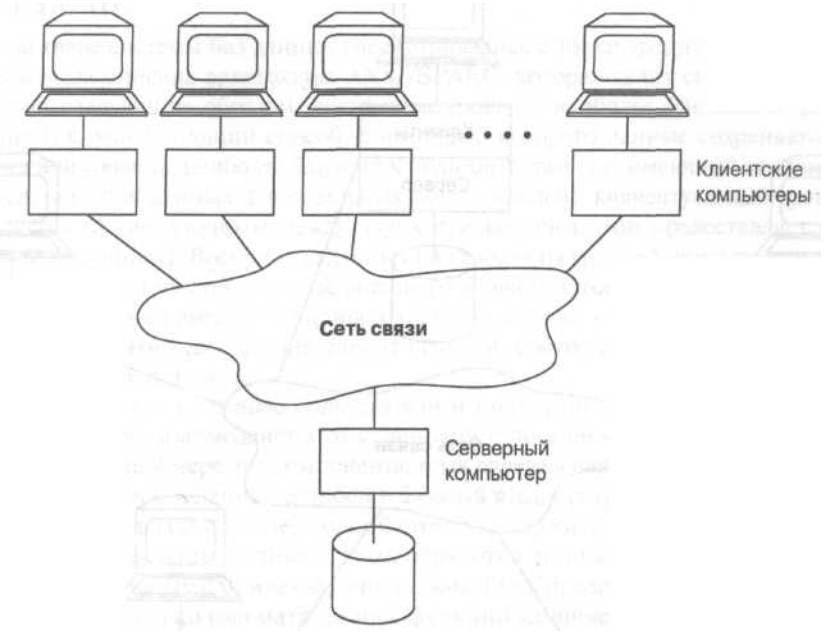


Рис. 2.7. Система с одним сервером и несколькими клиентами

Отметим последнее преимущество, которое состоит в том, что отдельный клиентский компьютер может иметь доступ к нескольким серверным компьютерам (случай, противоположный показанному на рис. 2.7). Это весьма удобная возможность, поскольку, как уже упоминалось, предприятие обычно выполняет обработку данных таким образом, что полный набор всех данных не сохраняется на одном компьютере, а распределяется по нескольким разным компьютерам, причем в приложениях иногда необходим доступ к данным нескольких компьютеров. Такой доступ предоставляется в основном двумя способами.

- Клиент может получить доступ к любому количеству серверов, но лишь к одному из них в каждый момент времени (т.е. каждый запрос к базе данных может быть направлен только к одному серверу). В такой системе невозможно за один запрос получить комбинированные данные от двух и более серверов. Кроме того, пользователь в

такой системе должен знать, на каком именно компьютере содержится конкретная часть данных.

Клиент может получить доступ к любому количеству серверов одновременно (т.е. за один запрос можно получить комбинированные данные с двух и более серверов). В этом случае серверы рассматриваются клиентом как единый сервер (с логической точки зрения) и пользователь не обязан знать, на каком именно компьютере содержится та или иная часть данных.

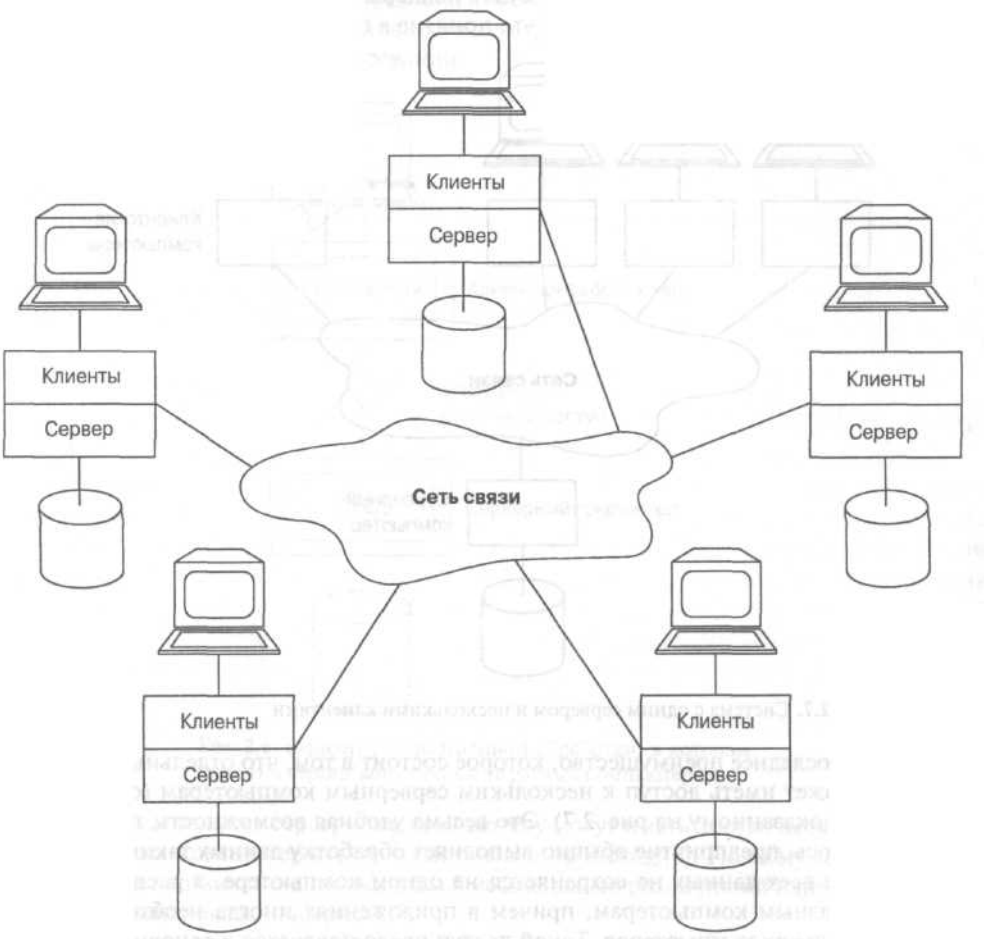


Рис. 2.8. Система, в которой каждый компьютер одновременно является и клиентом, и сервером

Второй случай — это пример системы, которую обычно называют **распределенной системой баз данных.** Тема распределенных баз данных сама по себе весьма обширна. Подходя к некоторому логическому завершению, отметим следующее: полная поддержка распределенных баз данных означает, что отдельное приложение может *прозрачно* обрабатывать данные, распределенные между множеством различных баз данных, управление

которыми осуществляют разные СУБД, работающие на соединенных коммуникационными сетями компьютерах разных типов с различными операционными системами. Здесь понятие *прозрачно* означает, что приложение выполняет обработку данных с логической точки зрения так, как будто управление данными полностью осуществляется одной СУБД, работающей на единственном компьютере. Предоставление такой возможности может оказаться чрезвычайно трудной задачей, но весьма желательной с практической точки зрения. Производители СУБД напряженно работают над созданием подобных систем. Подробнее распределенные базы данных рассматриваются в главе 21.

2.13. РЕЗЮМЕ

В этой главе системы баз данных рассматривались с точки зрения их общей архитектуры. Здесь была описана **архитектура** ANSI/SPARC, которая делит систему баз данных на три уровня следующим образом: **внутренний** уровень, наиболее близкий к физическому хранению (т.е. учитывающий способ, с помощью которого данные сохраняются физически); **внешний** уровень, наиболее близкий к пользователям (т.е. имеющий отношение к способу представления данных для отдельных пользователей); **концептуальный** уровень, который является промежуточным между двумя предыдущими (он предоставляет *обобщенное представление* данных). Восприятие данных на каждом из уровней описывается с помощью **схемы** (или нескольких схем в случае внешнего уровня). **Отображения** описывают соответствие между заданной внешней и концептуальной, а также между концептуальной и внутренней схемами. Эти отображения служат основой, соответственно, **логической** и **физической независимости от данных.**

Пользователи, т.е. конечные пользователи и прикладные программисты, работающие на внешнем уровне, взаимодействуют с данными с помощью **подъязыка данных,** который включает по меньшей мере два компонента: **язык определения данных** и **язык манипулирования данными.** Подъязык данных встроен в **базовый язык** программирования. Следует отметить, что границы, разделяющие, с одной стороны, базовый язык и подъязык данных, а с другой, язык определения данных и язык обработки данных, по своей природе в основном являются условными. В идеале, они должны быть прозрачными для пользователя.

Здесь также детально рассматривались функции **администратора базы данных** и функции СУБД. Кроме всего прочего, администратор *базы данных* несет ответственность за создание внутренней схемы **(физическое проектирование базы данных). В** противоположность этому, за создание концептуальной схемы **(логическое** или **концептуальное** проектирование базы данных) несет ответственность администратор *данных.* В функции СУБД входит также реализация запросов пользователей, написанных на языке определения данных или языке обработки данных. Функцией СУБД является и поддержка **словаря данных.**

Системы баз данных удобно рассматривать как простую структуру, состоящую из **сервера** (собственно СУБД) и набора **клиентов** (приложений). Клиент и сервер могут выполняться и зачастую выполняются на отдельных компьютерах, обеспечивая таким образом простейший вариант **распределенной обработки данных. В** общем случае каждый сервер может обслуживать много клиентов, а каждый клиент может работать со многими серверами. Если система обеспечивает полную прозрачность доступа (т.е. клиент работает так, как будто он имеет дело с одним сервером на единственном компьютере, не учитывая реального физического расположения данных), то налицо настоящая **распределенная система баз данных.**

УПРАЖНЕНИЯ

2.1. Начертите схему архитектуры системы баз данных, представленной в этой главе (архитектуры ANSI/SPARC).

2.2. Дайте определения следующим терминам:

базовый язык	планируемый запрос
внешний интерфейс	подъязык данных
внешний язык определения данных, схема, представление	пользовательский интерфейс
внутренний язык определения данных, схема, представление	распределенная база данных
выгрузка-перезагрузка данных	распределенная обработка
диспетчер передачи данных	реорганизация
загрузка данных	сервер
клиент	система базы данных и передачи данных
концептуальный язык определения данных, схема, представление	словарь данных
логический проект базы данных	утилита
машина базы данных	физический проект базы данных
непланируемый запрос	хранимое определение базы данных
отображение "внешний—концептуальный"	язык манипулирования данными
отображение "концептуальный—внутренний"	язык определения данных

2.3. Опишите последовательность шагов, применяемых при выборке определенного экземпляра внешней записи.

2.4. Перечислите главные функции, выполняемые СУБД.

2.5. Укажите различия между логической и физической независимостью от данных.

2.6. Как вы понимаете термин *метаданные?*

2.7. Перечислите главные функции, выполняемые АБД.

2.8. Укажите различия между СУБД и системой управления файлами.

2.9. Приведите несколько примеров инструментальных средств, предоставляемых различными поставщиками.

2.10. Приведите несколько примеров утилит базы данных.

2.11. Проанализируйте любую доступную вам систему баз данных. Попытайтесь представить ее в соответствии с архитектурой ANSI/SPARC, как описано в этой главе. Полностью ли она поддерживает три уровня архитектуры? Как определены отображения между уровнями? Чем подобны различные языки определения данных (внешний, концептуальный и внутренний)? Какой подъязык данных поддерживает система? Какой язык является базовым? Кто выполняет функции АБД? Имеются ли какие-нибудь средства организации защиты и поддержания целостности данных? Существует ли в системе словарь? Описывает ли он сам

себя? Какие приложения, предоставляемые поставщиками, поддерживает система? Какие утилиты входят в состав системы? Есть ли в системе отдельный компонент диспетчера передачи данных? Имеются ли какие-либо возможности распределенной обработки?

СПИСОК ЛИТЕРАТУРЫ

Хотя некоторые из перечисленных ниже изданий были выпущены давно, в них можно найти полезную информацию, которая касается понятий, представленных в этой главе.

2.1. ANSI/X3/SPARC Study Group on Data Base Management Systems. Interim Report // FDT (ACM SIGMOD bulletin). - 1975. - 7, № 2.

2.2. Tsichritzis D.C. and Klug A. (eds). The ANSI/X3/SPARC Framework: Report of the Study Group on Data Base Management Systems // Information Systems. — 1978. — 3.

Эти два документа, [2.1] и [2.2],— соответственно, предварительный и окончательный отчеты группы ANSI/X3/SPARC Study Group. Группа ANSI/X3/SPARC (полное название — Study Group on Data Base Management Systems) была организована в 1972 году комитетом Standards Planning and Requirements Committee (SPARC) института American National Standards Institute on Computers and Information Processing (ANSI/X3).

Примечание. Примерно через 25 лет название X3 перешло к комитету NCITS (National Committee on Information Technology Standards — Национальный комитет по стандартам информационной технологии). Несколько лет спустя это название было снова передано, на этот раз комитету INCITS, в названии которого буквы IN, которые обозначают *международный* (INternational), заменили букву N, обозначавшую *национальный* (National) в названии NCITS.
В задачи группы входило определение того, нуждаются ли какие-либо области технологии баз данных в стандартизации (если нуждаются, то какие именно), и выработка набора рекомендуемых действий в каждой из этих областей. В процессе работы над поставленными задачами группа пришла к выводу, что единственный подходящий объект стандартизации — *интерфейсы,* и в соответствии с этим определила общую архитектуру, или фундамент, системы баз данных, а также указала на важную роль подобных интерфейсов. В окончательном отчете представлено подробное описание архитектуры и некоторых из 42 (!) указанных интерфейсов. Предварительный отчет — это более ранний документ, который представляет определенный интерес, так как в нем отдельные вопросы рассмотрены более подробно.

2.3. Van Griethuysen J.J. (ed.). Concepts and Terminology for the Conceptual Schema and the Information Base // International Organization for Standardization (ISO) Technical Report ISO/TR9007. -July 1987.

Этот документ представляет собой отчет рабочей группы Международной организации по стандартизации (International Standard Organization — ISO), в который включено "определение понятий для языков концептуальных схем". В отчете рабочей группы предложено три альтернативных подхода (точнее, три *группы* подходов) к формализации концептуальной схемы. Каждый из подходов был применен к стандартному примеру, связанному с деятельностью гипотетического управления

регистрацией автомобилей. Три группы — это подходы "сущность—атрибут-связь", подходы с использованием *бинарных связей* и подходы, основанные на *интерпретируемой предикатной логике.* В отчете обсуждаются фундаментальные понятия, лежащие в основе понятия концептуальной схемы, а также излагаются принципы реализации системы, которая должным образом поддерживает концептуальную схему. Это достаточно сложный, однако очень важный документ для всех, кто серьезно интересуется концептуальным уровнем системы.

2.4. Kent W. Data and Reality. — Amsterdam, Netherlands: North-Holland; New York, N.Y.: Elsevier Science, 1978.

Захватывающее и заставляющее задуматься описание природы информации, в частности концептуальной схемы. "Эта книга представляет философию, согласно которой жизнь и действительность являются, в сущности, аморфными, беспорядочными, противоречивыми, непоследовательными, нерациональными и нерепрезентативными" (цитата из последней главы). Книгу можно рассматривать как краткое руководство по решению реальных проблем, с которыми трудно справиться в рамках существующего формализма баз данных, в частности из-за ограничений структур, подобных записям, которые используются в реляционном подходе. Рекомендуем ознакомиться.

2.5. Odysseas G. Tsatalos, Marvin H. Solomon, and Yannis E. Ioannidis. The GMAP: A Versatile Tool for Phisical Data Independence. Proc. 20th Int. Conf. On Very Large Data Bases. — Santiago, Chile. — September 1994.

Сокращение GMAP означает *обобщенный многоуровневый путь доступа* (Generalized Multi-Level Access Path). Авторы статьи справедливо отмечают, что современные продукты баз данных "вынуждают пользователей составлять запросы в терминах логической схемы, которая непосредственно связана с физическими структурами", и поэтому усиливают зависимость от физических данных. В этой статье предлагается язык отображения "концептуальный-внутренний" (по терминологии данной главы), который можно использовать для значительно большего количества видов отображений, чем обычно обеспечивается в современных продуктах. Предоставляются конкретная *логическая схема* и язык, основанный на реляционной алгебре (см. главу 7), и следовательно, описательный, а не процедурный по своей природе, что позволяет определить множество физических схем, которые формально следуют из такой логической схемы. Кроме всего прочего, подобные физические схемы могут включать вертикальное и горизонтальное разделения (глава 21), произвольное количество путей физического доступа, группирование и контроль избыточности.

В статье также приводится алгоритм преобразования пользовательских операций над логической схемой в эквивалентные операции над физической схемой. Прототип реализации показывает, что АБД получает возможность настроить физическую схему так, чтобы "достичь значительно более высокой производительности по сравнению с той, которой можно добиться обычными методами".

ГЛАВА 3

Введение в реляционные базы данных

3.1. ВВЕДЕНИЕ

Как отмечалось в главе 1, в этой книге основное внимание сконцентрировано на реляционных системах. В части II подробно описаны теоретические основы таких систем, а точнее — реляционная модель данных. Эта глава является лишь предварительным и весьма неформальным введением в материал, подробно изложенный в части II (и в некоторой степени в материал последующих глав), которое послужит основой для лучшего понимания последующих частей книги. Большинство затронутых здесь тем рассматривается в дальнейшем более формально и подробно.

3.2. РЕЛЯЦИОННАЯ МОДЕЛЬ

Как уже неоднократно отмечалось, реляционные системы базируются на формальных основах, или теории, которая называется *реляционной моделью данных.* Интуитивно ясно, что, кроме всего прочего, в такой системе выполняются как минимум три условия.

- **Структурный аспект.** Данные в базе воспринимаются пользователем, как таблицы (и никак иначе).
- Аспект целостности. Эти таблицы отвечают определенным условиям целостности (которые будут рассматриваться в конце раздела).
- **Аспект обработки.** В распоряжении пользователя имеются операторы манипулиро вания таблицами (например, предназначенные для поиска данных), которые гене рируют новые таблицы на основании уже имеющихся и среди которых есть, по крайней мере, операторы сокращения (restrict), проекции (project) и объединения (join).

На рис. 3.1 показан простой пример реляционной базы данных отделов (таблица DEPT) и служащих (таблица EMP). Очевидно, что эта база данных действительно "воспринимается как набор таблиц" (по-видимому, смысл этих таблиц не требует пояснений). На рис. 3.2 показаны некоторые примеры операций сокращения, проекции и соединения для этой базы данных. Ниже приведены (очень нестрогие!) определения этих операций.

DEPT

DEPT #	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M
D3	Research	5M

EMP

EMP #	ENAME	DEPT #	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

Рис. 3.1. База данных отделов и служащих (приведенные в ней данные часто используются в качестве примера)

- Операция *сокращения* извлекает указанные строки из таблицы (в названии этой операции подразумевается, что кардинальность ее результата меньше или равна кардинальности исходной таблицы). Следует отметить, что операцию сокращения иногда называют *выборкой;* мы предпочитаем термин *сокращение,* поскольку соот ветствующий оператор не полностью соответствует оператору SELECT языка SQL.
- Операция *проекции* предназначена для извлечения определенных столбцов из таб лицы.
- **а** Операция *соединения* предназначена для получения комбинации двух таблиц на основе общих значений в общих столбцах.

Из трех примеров, приведенных на рис. 3.2, в комментариях, по-видимому, нуждается только операция соединения. Прежде всего, заслуживает внимания то, что в таблицах DEPT и EMP есть общий столбец DEPT#, а следовательно, для этих таблиц можно выполнить операцию соединения на основе общих значений в этом столбце. При выполнении данной операции строка таблицы DEPT соединяется со строкой таблицы EMP и образуется более длинная строка, но подобное происходит тогда и только тогда, когда эти две

строки имеют одинаковое значение поля DEPT#. Например, можно соединить в результирующую строку (табл. 3.1) следующие строки таблиц DEPT и EMP (названия столбцов приведены для наглядности). Это возможно, так как в общем столбце рассматриваемых строк имеется одно и то же значение D1. Результирующая строка приведена в табл. 3.2. Общий результат состоит из множества всех таких соединенных строк. Обратите внимание, что столбец DEPT# в каждой результирующей строке встречается один раз, а не два. Следует также отметить, что в поле DEPT# таблицы EMP отсутствует значение D3 (т.е. нет служащих, работающих в отделе D3), поэтому в данном поле нет и результирующих строк со значением D3, хотя строка со значением D3 в таблице DEPT *присутствует.*

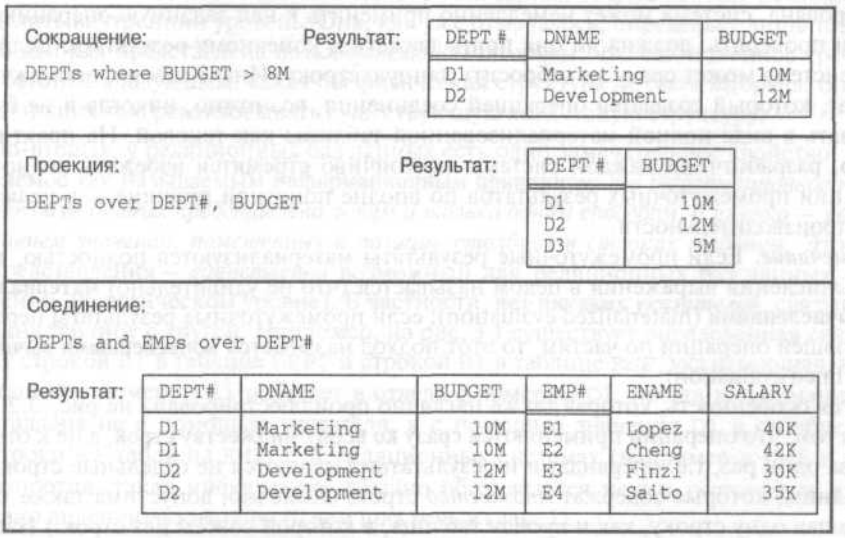


Рис. 3.2. Примеры применения операций сокращения, проекции и соединения

Таблица 3.1. Строки таблиц EMP и DEPT перед соединением

DEPT#	DNAME	BUDGET
D1	Marketing	10M

EMP#	ENAME	DEPT#	SALARY
E1	Lopez	D1	40K

Таблица 3.2. Результат соединения

DEPT#	DNAME	BUDGET	EMP#	ENAME	SALARY
D1	Marketing	10M	E1	Lopez	40K

Необходимо отметить одну важную особенность, очевидную из рис. 3.2: *результат выполнения каждой из трех представленных операций — это еще одна таблица* (другими словами, эти операторы фактически "порождают таблицы из таблиц", что и было указано ранее). Это —пример проявления реляционного свойства **замкнутости.** Оно имеет очень большое значение, главным образом потому, что результатом выполнения операции является объект того же рода, что и объект, над которым производилась операция, а именно — таблица. Но это значит, что к *результатам операций можно снова применить*

какую-либо операцию. Например, можно сформировать проекцию соединения, соединение двух сокращений, сокращение проекции и т.д. Другими словами, можно использовать *вложенные выражения,* т.е. выражения, в которых операнды представлены выражениями, а не простыми именами таблиц. В данной и последующих главах показано, что этот факт имеет, в свою очередь, целый ряд важных следствий.

Кстати, говоря о том, что результатом выполнения каждой операции является таблица, мы имеем в виду, что это — таблица, но *с концептуальной точки зрения.* Под этим не подразумевается, что система обязательно должна полностью материализовать результат каждой отдельной операции[1]. Предположим, например, что понадобилось сформировать сокращение соединения. В этом случае, как только строка требуемого соединения будет сформирована, система может немедленно применить к ней заданную операцию сокращения и проверить, должна ли она принадлежать к конечному результату. Если это не так, то система может сразу же отбросить данную строку. Иначе говоря, промежуточный результат, который создается операцией соединения, возможно, никогда и не будет существовать в виде полной материализованной таблицы как таковой. На практике, как правило, разработчики каждой системы настойчиво стремятся избежать полной материализации промежуточных результатов по вполне понятной причине — в целях повышения производительности.

Примечание. Если промежуточные результаты материализуются полностью, то стратегия вычисления выражения в целом называется (что не удивительно) **материализованными вычислениями** (materialized evaluation); если промежуточные результаты передаются последующей операции по частям, то этот подход называется **конвейерными вычислениями** (pipelined evaluation).

Другая особенность, которая также наглядно проиллюстрирована на рис. 3.2, заключается в том, что операции применяются **сразу ко всему множеству строк,** а не к отдельной строке за один раз, т.е. операндами и результатами являются не отдельные строки, а целые таблицы, которые содержат *множество* строк. (Конечно, допустима также таблица, содержащая одну строку, как и *пустая* таблица, в которой совсем нет строк.) Например, операция соединения на рис. 3.2 применяется к двум таблицам, состоящим из трех и четырех строк, соответственно, а в результате возвращает таблицу из четырех строк. В противоположность этому, операции нереляционных систем обычно за одно обращение обрабатывают только одну строку или запись. Таким образом, *возможность обработки множества строк* — главная отличительная характеристика реляционных систем (обсуждение этого вопроса будет продолжено в разделе 3.5).

Вернемся к анализу рис. 3.1. Ниже приведено несколько замечаний, которые можно сделать на основе примера базы данных, приведенного на этом рисунке.

- Во-первых, отметим, что определение реляционной системы требует, чтобы база данных только *воспринималась пользователем* как набор таблиц. Таблицы в реляционной системе являются **логическими,** а не физическими структурами. На самом деле, на физическом уровне система может использовать любую из существующих структур памяти (последовательный файл, индексирование, хэширование, цепочку указателей, сжатие и т.п.), лишь бы существовала возможность отображать эти структуры в виде таблиц на логическом уровне. Данное положение можно

[1] Другими словами, как было отмечено в главе 1, реляционная модель — это действительно только модель; в ней ничего не сказано о реализации.

сформулировать и по-другому: таблицы представляют собой *абстракцию* способа физического хранения данных, в которой все нюансы реализации на уровне физической памяти (размещение хранимых записей, упорядочение хранимых записей, кодировка хранимых данных, префиксы хранимых записей, хранимые структуры доступа, такие как индексы и т.д.) *скрыты от пользователя.*

В данном случае термин *логическая структура* в терминологии ANSI/SPARC относится как к концептуальному, так и ко внешнему уровням. Дело в том, что (как описано в главе 2) и концептуальный, и внешний уровни в реляционной системе являются одинаково реляционными, и лишь внутренний или физический уровень не является таковым. На самом деле реляционная теория вообще не может определить внутренний уровень. Она, как уже отмечалось, определяет лишь то, как база данных представлена *пользователю*[2]. Повторяем, что единственное требование состоит в следующем: какая бы физическая структура не была выбрана, она должна полностью реализовывать существующую логическую структуру.

- Во-вторых, у реляционных баз данных есть одно замечательное свойство, определяемое так называемым **информационным принципом:** *все информационное наполнение базы данных представлено одним и только одним способом, а именно — явным заданием значений, помещенных в позиции столбцов в строках таблицы.* Этот метод представления — *единственно* возможный для реляционных баз данных (естественно, на логическом уровне). В частности, **нет никаких** *указателей,* связывающих одну таблицу с другой. Например, на рис. 3.1 существует определенная связь между строкой D1 в таблице DEPT и строкой E1 в таблице EMP, указывающая, что служащий с номером E1 работает в отделе с номером D1, но эта информация представлена не с помощью указателя, а с помощью значения D1 в столбце DEPT# строки E1 таблицы EMP. В нереляционных системах (например в IMS и 1DMS), напротив, такая информация обычно обозначается неким *указателем,* который явно виден пользователю (о чем шла речь в главе 1).

  Примечание. Объяснение причин, по которым предоставление пользователю доступа к указателям является нарушением информационного принципа, приведено в главе 26. А на данный момент лишь отметим — говоря о том, что в реляционной системе нет указателей, мы не имеем в виду, что в ней не может быть указателей *на физическом уровне;* наоборот, они, конечно, могут быть предусмотрены на этом уровне и в действительности наверняка существуют. Но, как уже было сказано, сведения об организации физического хранения в реляционных системах скрыты от пользователя.

На этом завершается тема, связанная со структурой и обработкой данных в реляционной модели. Перейдем к рассмотрению аспекта целостности. Еще раз воспользуемся базой данных отделов и сотрудников, приведенной на рис. 3.1. На практике для такой

---

[2] К сожалению, приходится констатировать, что большинство современных продуктов SQL не обеспечивает должной поддержки этого аспекта теории. А точнее, они обычно в определенной степени поддерживают только отображения "концептуальный—внутренний" для операций выборки (как правило, отображают одну логическую таблицу непосредственно в один хранимый файл). И вследствие этого (как указано в главе 1) они не обеспечивают независимость от физических данных в той мере, в какой это теоретически предусмотрено в реляционной технологии (дополнительные сведения приведены в приложении А).

базы данных потребовалось бы определить несколько ограничений поддержки целостности базы. Например, допустим, что зарплата служащих не должна выходить за пределы от 25 до 95 тыс. долл. в год, а бюджет отдела должен находиться в пределах от 1 до 15 млн. долл. и т.д. Некоторые из таких правил имеют настолько важное практическое значение, что получили специальные названия. Рассмотрим их более подробно.

1. Каждая строка в таблице DEPT должна включать уникальное значение столбца DEPT#; аналогично, каждая строка в таблице EMP должна включать уникальное значение столбца EMP#. Говорят, что столбцы DEPT# В таблице DEPT И EMP# В таб лице EMP являются **первичными ключами** для своих таблиц. (Напомним, что в главе 1 мы условились отмечать столбцы первичных ключей двойным подчеркиванием.)
2. Каждое значение столбца DEPT# в таблице EMP ДОЛЖНО быть представлено и в виде значения столбца DEPT# в таблице DEPT в соответствии с тем фактом, что каждый служащий должен быть приписан к существующему отделу. Говорят, что столбец DEPT# в таблице EMP является **внешним ключом,** ссылающимся на первичный ключ таблицы DEPT.

Более формальное определение

Завершим этот раздел несколько более формальным определением реляционной модели, чтобы можно было ссылаться на него в дальнейшем (хотя на данном этапе такое определение будет несколько абстрактным и не очень четким). В первом приближении **реляционная** модель состоит из следующих пяти компонентов.

1. Неограниченный набор **скалярных типов** (включая, в частности, *логический тип* или *истинностное значение).*
2. Генератор **типов отношений** и соответствующая интерпретация для сгенерирован ных типов отношений.
3. Возможность определения **переменных отношения** для указанных сгенерированных типов отношений.
4. Операция **реляционного присваивания** для присваивания реляционных значений указанным переменным отношения.
5. Неограниченный набор общих **реляционных операторов** *{реляционная алгебра)* для получения значений отношений из других значений отношений.

Вполне очевидно, что реляционная модель — это нечто большее, чем просто "таблицы плюс операции сокращения, проекции и соединения", хотя ее неформально довольно часто характеризуют именно таким образом.

Примечание. Читателя, возможно, удивило то, что в предыдущем определении отсутствует явное упоминание об ограничениях целостности. Однако на самом деле такие ограничения представляют собой просто приложения (хотя и очень важные), в которых используются реляционные операторы. Иначе говоря, правила ограничений формулируются (во всяком случае, концептуально) в терминах реляционных операторов, как мы сможем убедиться в главе 9.

3.3. ОТНОШЕНИЯ И ПЕРЕМЕННЫЕ ОТНОШЕНИЯ

Если предположить, что реляционная база данных — это по существу просто база данных, в которой данные представлены в виде таблиц (а это так и *есть),* то возникает резонный вопрос: почему же мы называем такую базу данных именно реляционной, а не, скажем, табличной? Ответ прост (фактически он был дан еще в главе 1): термин "relation" *(отношение)* — это формальное название таблицы (точнее, определенного вида таблиц; подробности будут приведены в главе 6). Например, можно сказать, что база данных отделов и служащих, представленная на рис. 3.1, содержит два *отношения.*

В настоящее время в неформальном контексте термины *отношение* и *таблица* принято считать синонимами. На практике в подобном контексте термин *таблица* используется гораздо чаще, чем термин *отношение.* На обсуждении этого вопроса стоит остановиться, чтобы выяснить, почему все-таки термин *отношение* поставлен на первое место. Вкратце это объясняется следующим.

- Как уже отмечалось, реляционные системы основаны на реляционной модели. Ре ляционная модель, в свою очередь, — это абстрактная теория данных, основанная на некоторых областях математики (в основном на теории множеств и логике пре дикатов).
- Принципы реляционной модели были сформулированы в 1969 и 1970 годах Е.Ф. Коддом (E.F. Codd), который в то время работал в корпорации IBM. В конце 1968 года Кодд, математик по образованию, впервые пришел к выводу, что для внедрения в сферу управления базами данных строгих и точных принципов можно использовать математические дисциплины. В то время данной области недостава ло именно этих качеств. Идеи Кодда впервые подробно были изложены в статье "A Relational Model of Data for Large Shared Data Banks", ставшей классической (см. [6.1] в главе 6).
- С того времени эти идеи стали общепринятыми и оказали весьма существенное влияние на все аспекты технологии баз данных, а также на другие области, такие как искусственный интеллект, обработка естественных языков и проектирование аппаратных средств.

В теории реляционной модели, с того времени, как она была первоначально сформулирована Коддом, умышленно применялись лишь определенные, тщательно подобранные термины. Например, вначале сам термин *отношение* в сфере информационных технологий практически не использовался, хотя такое понятие иногда употреблялось в других областях. Суть заключалась в том, что многие распространенные тогда термины были очень *нечеткими* — им не хватало точности, необходимой для такой формальной теории, которую предложил Кодд. Например, рассмотрим термин *запись.* В разное время и в различных контекстах он мог означать *экземпляр* записи или *тип* записи, *логическую* запись или *физическую* запись, *хранимую* запись или *виртуальную* запись, а возможно, и еще что-нибудь. Поэтому в формальной реляционной модели термин *запись* не используется вообще — вместо него применяется термин *кортеж* (tuple), точное определение которого дал сам Кодд, когда ввел его впервые. Мы не станем приводить здесь это определение (оно дано в главе 6), а лишь отметим, что термин *кортеж* приблизительно соответствует понятию *строки* (так же, как термин *отношение* приблизительно соответствует понятию *таблицы).* Аналогичным образом, в реляционной модели не используется

термин *поле;* вместо него используется термин *атрибут,* который в рассматриваемом контексте примерно соответствует понятию столбца таблицы.

При дальнейшем изучении теоретических аспектов реляционных систем в части II мы будем использовать более формальную терминологию, однако в данной главе мы не станем придерживаться строго заданных терминов, а воспользуемся более привычными, такими как *строка* и *столбец.* Однако один формальный термин — *отношение* — мы все-таки будем использовать довольно часто.

Снова обратимся к базе данных отделов и сотрудников, представленной на рис. 3.1. Заметим, что таблицы DEPT и EMP в базе данных фактически являются *переменными отношения,* т.е. их значения — это *значения отношения* (т.е. отношения принимают различные значения в разное время). Предположим, например, что таблица EMP в данный момент имеет значение *(значение* отношения), которое показано на рис. 3.1, и далее допустим, что мы удалили строку о сотруднике с фамилией Saito (его номер — E4).

```
DELETE EMP WHERE EMP# = EMP# ('E4') ;
```

Результат выполнения этой операции показан на рис. 3.3.

EMP

EMP #	ENAME	DEPT #	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K

Рис. 3.3. Переменная отношения EMP после удаления строки сотрудника с кодом E4

Концептуально это действие можно описать таким образом: *старое значение отношения EMP было заменено* в целом *совершенно другим, новым значением отношения.* Конечно, старое значение (с четырьмя строками) и новое значение (с тремя строками) очень похожи, но в действительности они *являются* разными. По сути, данная операция удаления строки — это просто альтернативный, упрощенный способ записи для определенной операции **реляционного присваивания,** которая могла бы выглядеть примерно следующим образом.

```
EMP := EMP WHERE NOT ( EMP# = EMP# ( ' E4 ' ) ) ;
```

Как и при любом присваивании, здесь с концептуальной точки зрения сначала вычисляется выражение, расположенное справа от знака присваивания, а затем результат присваивается переменной, которая записана перед знаком присваивания (по определению перед знаком присваивания может стоять, естественно, только *переменная).* В целом, как уже отмечалось, задача заключается в том, чтобы заменить "старое" значение отношения EMP "новым".

Примечание. Как первоначальный оператор DELETE, так и равносильный ему оператор присваивания записаны на языке Tutorial D, который широко используется в данной книге.

Естественно, что операции INSERT и UPDATE также по существу являются просто сокращенной формой записи соответствующих реляционных операций присваивания (дополнительные сведения приведены в главе 6).

Но вот что огорчает: во многих публикациях используется термин *отношение,* когда в действительности подразумевается *переменная отношения* (а также тогда, когда подразумевается *само* отношение, т.е. *значение* отношения). А это, как показывает практика,

приводит к путанице. Поэтому в дальнейшем мы будем четко различать переменные отношения и *сами* отношения. Следуя примеру публикации [3.3], для переменной отношения (relation variable) будем использовать термин **переменная отношения** (сокращенный английский вариант — relvar), и только этот термин, во всех случаях, когда речь будет идти не об отношении, а о переменной отношения[3]. Обратите внимание на то, что с этого момента неуточненный термин *отношение* будет применяться именно для описания значения отношения (по такому же принципу, как, например, неуточненный термин *целое число* применяется исключительно для описания целочисленного значения), хотя время от времени мы будем также использовать уточненный термин *значение отношения,* чтобы подчеркнуть его отличие от переменной отношения.

Прежде чем перейти к дальнейшему изложению, следует предупредить читателя, что термин *переменная отношения* (relvar) не является общепринятым, хотя должен быть таковым! Мы действительно считаем, что очень важно различать *сами* отношения (т.е. значения отношений) и переменные отношения как таковые. (Автор должен признать, что такое требование не учитывалось в предыдущих изданиях настоящей книги, как и в другой литературе по базам данных. Мало того, до сих пор большинство специалистов в этой области игнорируют существенные различия между отношениями и переменными отношения.) Особое внимание следует обратить на то, что ограничения целостности и операции обновления, которые рассматриваются в главах 6 и 9, в соответствии с их определением, должны, безусловно, применяться к переменным отношения, а не к отношениям.

3.4. СМЫСЛ ОТНОШЕНИЙ

В главе 1 отмечалось, что столбцы в отношениях связаны с **типами данных.** А в конце раздела 3.2 мы говорили, что реляционная модель включает "неограниченный набор типов [данных]". Помимо всего прочего, это означает, что **пользователи могут определять собственные типы** (а также, конечно, применять определяемые системой или *встроенные* типы). Например, определять типы можно представленным ниже способом (снова воспользуемся синтаксисом языка Tutorial D, причем многоточие "..." здесь заменяет сами определения, которые для нас сейчас не важны).

```
TYPE EMP# ... ;
TYPE NAME ... ; TYPE
DEPT# . . . ; TYPE
MONEY ... ;
```

Тип EMP#, например, можно рассматривать, как *множество всевозможных табельных номеров,* тип NAME — *как множество всевозможных имен* и т.д.

Теперь обратимся к рис. 3.4, который фактически представляет собой часть представленного на рис. 3.1 отношения EMP, дополненного обозначениями типов столбцов. Как показано на этом рисунке, каждое отношение, точнее, каждое *значение* отношения, состоит из двух частей: набора пар *"имя_столбца:имя_типа"* **(заголовка)** и набора строк, согласованных с этим заголовком (тела). Следует отметить, что на практике компоненты заголовка *"имя_типа"* зачастую опускают, как это делали и мы во всех предыдущих примерах. Однако необходимо учитывать, что концептуально они всегда присутствуют.

---

[3] Различие между значениями отношения и переменными отношения фактически представляют собой особый случай различия между значениями и переменными в целом. Последнее различие подробно анализируется в главе 5.

EMP# : EMP#	ENAME : NAME	DEPT # : DEPT#	SALARY : MONEY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

Рис. 3.4. Пример значения отношения EMP с указанием типов столбцов

Рассмотрим один важный, хотя и не столь распространенный, способ представления смысла отношений.

- Во-первых, в определенном отношении r заголовок отношения r представляет со бой определенный **предикат** (под *предикатом* подразумевается просто функция с истинностными значениями, которая, как и все функции, имеет ряд *формальных параметров).*
- Во-вторых, как упоминалось в главе 1, каждая строка в теле отношения r представ ляет собой определенное **истинное высказывание,** полученное из предиката путем подстановки определенных значений *фактических параметров* соответствующего типа вместо *формальных параметров* этого предиката (т.е. путем конкретизации

Например в случае, представленном на рис. 3.4, предикат будет следующим.

```
Служащий с табельным номером EMP# и фамилией ENAME работает в
отделе с номером DEPT# и получает зарплату SALARY.
```

Здесь формальными параметрами являются EMP#, ENAME, DEPT# И SALARY, которые, конечно же, соответствуют четырем столбцам переменной отношения EMP. Ниже приведены примеры соответствующих истинных утверждений.

Служащий с табельным номером E1 и фамилией Lopez работает в отделе **с** номером D1 и получает зарплату 40 тыс. долл. в год.

(Получено путем подстановки в параметр EMP# значения E1, в параметр NAME — значения Lopez, в параметр DEPT# — значения D1 и в параметр SALARY — значения 4 ОК.)

```
Служащий с номером E2 и фамилией Cheng работает в отделе с
номером D1 и получает зарплату 42 тыс. долл. в год.
```

(Получено путем подстановки в параметр EMP# значения E2, в параметр NAME — значения Cheng, в параметр DEPT# — значения D1 и в параметр SALARY — значения 42К.)

Иными словами:

- ***типы* — это объекты (множества объектов), которые могут стать предметом обсуж**
  дения;
- ***отношения* — это факты (множества фактов), касающиеся объектов, которые могут**
  стать предметом обсуждения.

(Есть прекрасная аналогия, которая может помочь вам понять смысл и запомнить эти важные определения. *Типы связаны с отношениями точно так же, как существительные связаны с предложениями на естественном языке.)* В нашем пример предметом обсуждения являются табельные номера служащих, их имена, номера отделов и значения денежных сумм, а то, что мы можем сказать об обсуждаемом предмете, — это истинное высказывание такого вида:

"Служащий с определенным табельным номером имеет определенное имя, работает в определенном отделе и получает определенную зарплату".

Из вышесказанного следует:

- во-первых, *необходимы* и типы, и отношения (без применения типов мы не сможем определить предмет обсуждения, а без помощи отношений мы не сможем ничего о нем сказать);
- во-вторых, типы и отношения так же *необходимы,* как и *достаточны,* т.е., строго говоря, для того чтобы высказаться по поводу обсуждаемого предмета, больше ни чего не нужно;
- в-третьих, нельзя смешивать понятия типов и отношений. Достойно сожаления то, что в некоторых коммерческих продуктах (которые, безусловно, не заслужива ют названия реляционных!) не проводятся различия между типами и отношения ми. В главе 26 (раздел 26.2) приведен ряд соображений по поводу этой путаницы.

Кстати, важно понимать, что *каждое* отношение имеет связанный с ним предикат, включая отношения, полученные с помощью реляционных операторов, например оператора соединения. Отношение DEPT на рис. 3.1 и три результирующих отношения на рис. 3.2 имеют предикаты, которые определены следующим образом.

- **DEPT.** "Отдел **с** номером DEPT# называется DNAME И имеет бюджет BUDGET".
- **Сокращение отношения DEPT, где BUDGET > 8M.** "Отдел с номером DEPT# назы вается DNAME и имеет бюджет BUDGET, который превышает 8 млн. долл.".
- **Проекция, состоящая из столбцов DEPT# и BUDGET отношения DEPT.** "Отдел с номером DEPT# имеет некоторое название и бюджет BUDGET".
- **Соединение переменных отношения DEPT и EMP на основе столбца DEPT#.** "Отдел с номером DEPT# называется DNAME И имеет бюджет BUDGET, а служащий с номе ром EMP# по фамилии ENAME работает в отделе с номером DEPT# и получает зар плату SALARY" (заметим, что этот предикат имеет шесть параметров, а не семь, поскольку две ссылки на DEPT# обозначают один и тот же параметр).

Наконец, следует отметить, что переменные отношения также имеют предикаты. Таковым является предикат, общий для всех отношений, которые представляют собой возможные значения рассматриваемой переменной отношения. Например, предикат для переменной отношения EMP можно представить следующим образом.

```
Служащий с табельным номером EMP# имеет фамилию ENAME,
работает в отделе DEPT# и получает зарплату SALARY
```

3.5. ОПТИМИЗАЦИЯ

Как было описано в разделе 3.2, все реляционные операции, такие как сокращение, проекция и соединение, выполняются на *уровне множеств.* Поэтому реляционные языки часто называют *непроцедурными,* так как пользователь указывает, *что* делать, а не *как* делать. Фактически пользователь сообщает лишь, что ему нужно, без указания процедуры получения результата. Процесс *навигации* (перемещения) по хранимой базе данных в целях удовлетворения запроса пользователя выполняется системой автоматически, а не пользователем вручную. Поэтому реляционные системы иногда называют системами **автоматической навигации.** В нереляционных системах за навигацию по базе данных в основном несет ответственность сам пользователь. На рис. 3.5 приведена яркая иллюстрация преимуществ автоматической навигации — оператору языка SQL INSERT противопоставлен код навигации, подготовленный "вручную". Для получения того же результата подобный код, вероятно, должен быть подготовлен пользователем любой нереляционной системы (в данном случае — сетевой системы CODASYL; пример взят из главы по сетевым базам данных в [1.5]). Следует отметить, что здесь в качестве примера снова используется база данных деталей и поставщиков. За подробностями обратитесь к разделу 3.9.

Несмотря на предыдущие замечания, следует отметить, что *непроцедурный* — это хотя и общепринятый, но не совсем точный термин, потому что *процедурный* и *непроцедурный* — понятия относительные. Обычно можно с уверенностью определить лишь то, является ли язык А более (или менее) процедурным, чем язык Б. Поэтому точнее будет сказать, что реляционные языки, такие как SQL, характеризуются *более высоким уровнем абстракции,* чем типичные языки программирования, подобные C++ или COBOL (либо подъязыки данных, обычно принадлежащие нереляционным СУБД; см. рис. 3.5). В принципе, именно более высокий уровень абстракции способствует повышению продуктивности труда программистов, свойственному для реляционных систем.

Ответственность за организацию выполнения автоматической навигации возложена на очень важный компонент СУБД — **оптимизатор** (мы уже упоминали о нем в главе 2). Другими словами, работа оптимизатора заключается в том, чтобы выбрать самый эффективный способ выполнения для каждого запроса пользователя. Предположим, например, что пользователь сделал следующий запрос (снова воспользуемся языком Tutorial D).

```
( EMP WHERE EMP# = EMP# ('E4') ) { SALARY }
```

Пояснение. Выражение в первых скобках (EMP WHERE ...) означает, что применяется операция сокращения текущего значения переменной отношения EMP, касающаяся той строки, в которой значение столбца EMP# равно E4. Применяемая здесь языковая конструкция, представляющая собой имя столбца SALARY, заключенное в фигурные скобки, означает *проекцию* результата операции *сокращения* по столбцу SALARY. Результатом этой операции проекции становится отношение, состоящее из одного столбца и одной строки, которое содержит данные о заработке служащего E4. (Обратите внимание, что в данном случае в неявном виде используется реляционное свойство *замкнутости:* мы написали вложенное выражение, в котором результат операции сокращения применяется в качестве входных данных для операции проекции.)

```
INSERT INTO SP ( S#, P#, QTY )
VALUES ( 'S4', 'P3', 1000 ) ;
```

```
MOVE 'S4' TO S# IN S
FIND CALC S
ACCEPT S-SP-ADDR FROM S-SP CURRENCY
FIND LAST SP WITHIN S-SP
while SP found PERFORM
   ACCEPT S-SP-ADDR FROM S-SP CURRENCY
   FIND OWNER WITHIN P-SP
   GET P
   IF P# IN P < 'P3'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN S-SP
END-PERFORM
MOVE 'P3' TO P# IN P
FIND CALC P
ACCEPT P-SP-ADDR FROM P-SP CURRENCY
FIND LAST SP WITHIN P-SP
while SP found PERFORM
   ACCEPT P-SP-ADDR FROM P-SP CURRENCY
   FIND OWNER WITHIN S-SP
   GET S
   IF S# IN S < 'S4'
      leave loop
   END-IF
   FIND PRIOR SP WITHIN P-SP
END-PERFORM
MOVE 1000 TO QTY IN SP
FIND DB-KEY IS S-SP-ADDR
FIND DB-KEY IS P-SP-ADDR
STORE SP
CONNECT SP TO S-SP
CONNECT SP TO P-SP
```

Рис. 3.5. Примеры автоматической навигации и навигации вручную

Даже в этом простом примере могут применяться по крайней мере два *способа доступа* к необходимым данным.

1. Последовательный физический просмотр (хранимой версии) переменной отношения EMP, пока требуемая запись не будет найдена.
2. Если есть индекс для столбца EMP# (в хранимой версии) переменной отношения (который, вероятно, действительно существует, поскольку этот столбец является уникальным, а большинство систем фактически *требует* создания индекса для обеспечения уникальности), то переход с помощью этого индекса непосредственно к данным служащего с номером E4.

Оптимизатор определит, какую из двух возможных стратегий следует применить. В общем случае для реализации любого конкретного реляционного запроса оптимизатор будет осуществлять выбор стратегии, исходя из соображений, подобных следующим:

- на какие переменные отношения есть ссылки в запросе;
- насколько велики эти переменные отношения в настоящее время;
- какие существуют индексы;
- насколько избирательны эти индексы;
- как физически группируются данные на диске;
- какие реляционные операции используются; и т.д.

Поэтому повторяем: пользователь указывает в запросе, *какие* данные ему требуются, а не *как их получить,* тогда как стратегия доступа к данным выбирается оптимизатором (автоматическая навигация). В результате пользователи и пользовательские программы становятся независимыми от применяемых стратегий доступа, что, конечно же, совершенно необходимо, если мы хотим достичь реальной независимости от физического представления данных.

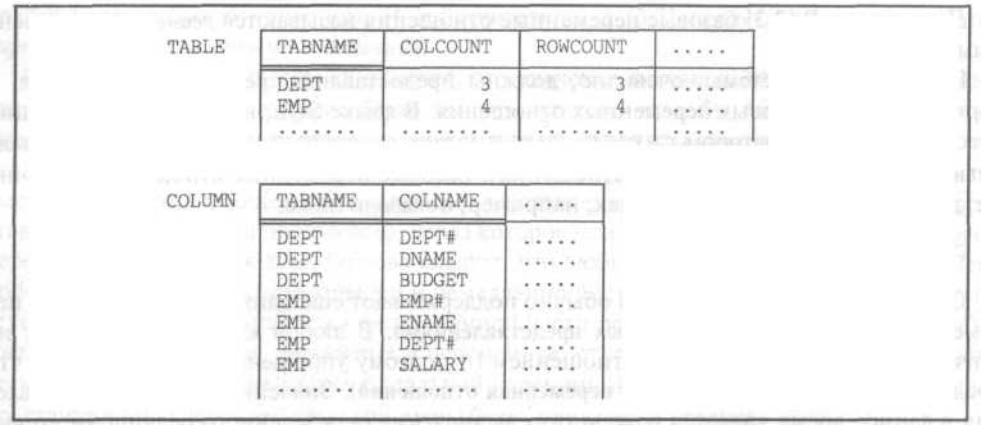
Более подробные сведения об оптимизаторе приводятся в главе 18.

3.6. КАТАЛОГ

Как отмечалось в главе 2, каждая СУБД должна поддерживать функции **каталога,** или **словаря.** Каталог обычно размещается там, где хранятся различные схемы (внешние, концептуальные, внутренние) и все, что относится к отображениям ("внешний-концептуальный", "концептуальный-внутренний", "внешний-внешний"). Иначе говоря, в каталоге содержится подробная информация (иногда называемая **описательной информацией** или **метаданными),** касающаяся различных объектов, которые имеют значение для самой системы. Примерами таких объектов могут служить переменные отношения, индексы, ограничения поддержки целостности, ограничения защиты и т.д. Описательная информация необходима для обеспечения правильной работы системы. Например, оптимизатор использует информацию каталога об индексах и других физических структурах хранения данных, а также прочую информацию, необходимую ему для принятия решения о том, как выполнить тот или иной запрос пользователя (см. главу 18). Аналогично, подсистема защиты использует информацию каталога о пользователях и установленных ограничениях защиты (глава 17), чтобы разрешить или запретить выполнение поступившего запроса.

Замечательным свойством реляционных систем является то, что их *каталог также состоит из переменных отношения* (точнее, из *системных* переменных отношения, названных так для того, чтобы отличать их от обычных пользовательских). В результате пользователь может обращаться к каталогу так же, как к своим данным. Например, в каталоге любой системы SQL обычно содержатся системные переменные отношения TABLES и COLUMNS, назначение которых — описание известных системе таблиц (т.е. переменных отношения) и столбцов этих таблиц. Для базы данных отделов и служащих переменные отношения[4] TABLES и COLUMNS могут быть схематически представлены в виде иерархической структуры так, как показано на рис. 3.6.

[4] Отметим, что исходя из наличия на этом рисунке столбца ROWCOUNT, можно сделать следующее за ключение: выполнение в базе данных операций INSERT и DELETE в качестве побочного эффекта потре бует обновления данных каталога. На практике столбец ROWCOUNT обычно обновляется только по спе циальному запросу (например, при запуске некоторой утилиты), поэтому значения этого столбца не всегда будут актуальными.



TABLE

TABNAME	COLCOUNT	ROWCOUNT	.....
DEPT	3	3	.....
EMP	4	4	.....
.....	.....	.....	.....

COLUMN

TABNAME	COLNAME	.....
DEPT	DEPT#	.....
DEPT	DNAME	.....
DEPT	BUDGET	.....
EMP	EMP#	.....
EMP	ENAME	.....
EMP	DEPT#	.....
EMP	SALARY	.....
.....	.....	.....

Рис. 3.6. Каталог базы данных отделов и служащих (изображен схематически)

Примечание. Как упоминалось в главе 2, каталог должен также описывать сам себя, т.е. включать записи, описывающие переменные отношения самого каталога (см. упр. 3.3).

Теперь предположим, что пользователю базы данных отделов и служащих понадобилось узнать, какие именно столбцы содержит переменная отношения DEPT (конечно, предполагается, что по каким-то причинам пользователь не имеет этой информации). Тогда необходимое выражение будет выглядеть следующим образом.

```
( COLUMNS WHERE TABNAME = 'DEPT' ) { COLNAME }
```

Вот еще один пример. Пусть необходимо узнать, в каких переменных отношения есть столбец EMP#.

```
( COLUMNS WHERE COLNAME = 'EMP#' ) { TABNAME }
```

Упражнение для самопроверки. Каким будет результат выполнения следующего выражения?

```
( ( TABLES JOIN COLUMNS )
          WHERE COLCOUNT < 5 ) { TABNAME, COLNAME }
```

3.7. БАЗОВЫЕ ПЕРЕМЕННЫЕ ОТНОШЕНИЯ И ПРЕДСТАВЛЕНИЯ

Мы уже видели, что на основе реляционных значений, присвоенных некоторому множеству переменных отношения, подобных DEPT и EMP, реляционные выражения позволяют получить множество других значений отношений, например, в результате соединения двух переменных отношения. Теперь необходимо ввести еще несколько новых терминов. Исходные (заданные) переменные отношения называются **базовыми переменными отношения,** а присвоенные им значения называются **базовыми** отношениями. Отношение, которое получено или может быть получено из базового отношения в результате выполнения каких-либо реляционных выражений, называется **производным** отношением.

Примечание. В [3.3] базовые переменные отношения называются *реальными* переменными отношения.

Реляционные системы, очевидно, должны предоставлять средства для создания, в первую очередь, базовых переменных отношения. В языке SQL, например, эта функция обеспечивается оператором CREATE TABLE (здесь слово TABLE используется в узком смысле, как *базовая* переменная отношения). Базовые переменные отношения, конечно же, должны быть *именованными,* как, например, показано ниже.

```
CREATE TABLE EMP ... ;
```

Однако реляционные системы обычно поддерживают еще один вид именованных переменных отношения, называемых **представлениями. В** любой конкретный момент их значение является *производным* отношением (и поэтому упрощенно можно считать, что представление — это **производная переменная отношения).** Значение данного представления в данное время является результатом вычисления определенного реляционного выражения в данный момент, а упомянутое реляционное выражение определяется в момент создания этого представления. Например, для определения представления TOPEMP можно использовать следующий оператор.

```
CREATE VIEW TOPEMP AS
                                        ( EMP WHERE SALARY > 3 3K
) { EMP#, ENAME, SALARY } ;
```

Примечание. Поскольку в данный момент это несущественно, в примере для удобства используется смешанная нотация языков SQL и Tutorial D.

При выполнении этого оператора выражение, следующее за ключевым словом AS и фактически **определяющее представление,** не вычисляется, а просто *запоминается* системой (обычно посредством сохранения в каталоге под указанным именем TOPEMP). Однако с точки зрения пользователя все выглядит так, как будто в базе данных появилась вполне реальная переменная отношения с именем TOPEMP, имеющая текущее значение, которое показано на рис. 3.7 только в незатененных участках. И пользователь должен иметь возможность оперировать этим представлением точно так, как если бы оно являлось обычной базовой переменной отношения.

TOPEMP

EMP #	ENAME	DEPT #	SALARY
E1	Lopez	D1	40K
E2	Cheng	D1	42K
E3	Finzi	D2	30K
E4	Saito	D2	35K

Рис. 3.7. Виртуальная переменная отношения TOPEMP (незатененные участки) как представление базовой переменной отношения EMP

Примечание. Выше уже было сказано, что если такие переменные отношения, как DEPT и EMP, можно считать *реальными,* то переменную отношения TOPEMP следует рассматривать как *виртуальную* переменную отношения, иначе говоря, как переменную отношения, которая внешне существует как таковая, но на самом деле ее нет (значение этой переменной отношения в любой данный момент зависит от значений некоторых

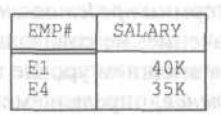
других переменных отношения). И действительно, в [3.3] представления называются *виртуальными переменными отношения*.

Однако будьте внимательны: отмечая, что значение переменной отношения TOPEMP является отношением, которое было бы результатом, если бы определяющее данное представление выражение было на самом деле вычислено, мы вовсе не хотим сказать, что существует *отдельная копия* этих данных. Иначе говоря, мы вовсе не имеем в виду, что выражение, определяющее представление, действительно вычисляется. Наоборот, представление — это по сути просто *окно,* через которое можно видеть часть значения базовой переменной отношения EMP. Отсюда следует, что любые изменения в базовой переменной отношения будут автоматически и немедленно поступать в подобное окно (конечно, если эти изменения относятся к незатененной части реальной переменной отношения). Аналогично, изменения, внесенные в переменную отношения TOPEMP, будут автоматически и немедленно применены к реальной переменной отношения EMP и, следовательно, станут видны через это окно (примеры будут приведены позднее).

Ниже показан пример запроса, использующего представление TOPEMP.

```
( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }
```

Если в качестве исходных используются данные на рис. 3.7, то результат будет иметь вид, показанный на рис. 3.8.



EMP#	SALARY
E1	40K
E4	35K

Рис. 3.8. Результат использования представления TOPEMP

С концептуальной точки зрения операции с представлениями, подобные рассмотренной выше, аналогичны операциям поиска, которые фактически реализуются посредством замены ссылки на имя представления выражением, которое *определяет* представление (т.е. выражением, сохраненным в каталоге). Поэтому в рассмотренном примере исходное выражение

```
( TOPEMP WHERE SALARY < 42K ) { EMP#, SALARY }
```

модифицируется системой следующим образом.

```
( ( ( EMP WHERE SALARY > 33K ) { EMP#, ENAME,
        SALARY } ) WHERE SALARY < 42K ) { EMP#,
        SALARY }
```

Здесь курсивом выделено имя представления в исходном выражении и заменяющий текст в модифицированной версии. После определенного количества перегруппировок это выражение может быть упрощено (глава 18) и представлено в следующем виде.

```
( EMP WHERE SALARY > 33K AND SALARY < 42K ) { EMP#, SALARY }
```

Вычисление данного выражения приводит к результату, показанному выше. Иными словами, первоначальная операция над представлением просто преобразуется в эквивалентную операцию над соответствующей базовой переменной отношения, после чего

полученная эквивалентная операция выполняется обычным образом (точнее, *оптимизируется* и выполняется обычным образом).

В качестве другого примера рассмотрим операцию удаления DELETE.

```
DELETE TOPEMP WHERE SALARY < 42K ;
```

На самом деле будет выполнена следующая операция.

```
DELETE EMP WHERE SALARY > 33K AND SALARY < 42K ;
```

Рассматриваемое здесь представление TOPEMP является очень простым, состоящим (выражаясь неформально) из подмножества строк и столбцов единственной базовой переменной отношения. Однако, в принципе, определение представления может иметь *произвольную сложность* (оно может даже ссылаться на другие представления). Например, далее приведено представление, определение которого включает соединение двух базовых переменных отношения.

```
CREATE VIEW JOINEX AS
   ( ( EMP JOIN DEPT ) WHERE BUDGET > 7M ) { EMP#, DEPT# } ;
```

К вопросу определения и обработки представлений мы еще вернемся в главе 10. Между прочим, сейчас уже можно объяснить смысл приведенного в конце раздела 2.2 замечания, касающегося того, что термин *представление* (view) в реляционном контексте имеет довольно специфическое значение, не совпадающее со значением, приписанным ему в архитектуре ANSI/SPARC. На внешнем уровне этой архитектуры база данных воспринимается как *внешнее представление,* определяемое внешней схемой (и разные пользователи могут иметь разные внешние представления). В реляционных системах, наоборот, представление, как пояснялось выше, является специально *именованной производной виртуальной переменной отношения.* Поэтому реляционным аналогом *внешнего представления* ANSI/SPARC обычно служит множество из нескольких переменных отношения, каждая из которых является представлением в реляционном смысле. *Внешняя схема* состоит из определений таких представлений. (Из этого следует, что в реляционной модели представления являются одним из способов обеспечения **логической независимости от данных,** хотя еще раз следует отметить, что современные коммерческие продукты имеют серьезные недостатки в этой части. Подробности приведены в главе 10.)

Архитектура ANSI/SPARC является достаточно общей и допускает произвольные преобразования между внешним и концептуальным уровнями. В принципе, даже *типы* структур данных, поддерживаемые на этих двух уровнях, могут быть различными: например, концептуальный уровень может быть реляционным, тогда как конкретному пользователю может быть передано внешнее представление иерархического типа[5]. Однако на практике в большинстве систем в качестве базовых на обоих уровнях используются одинаковые типы структур. Реляционные продукты не являются исключением из этого общего правила — представление по-прежнему остается переменной отношения, как и базовые переменные отношения. А поскольку на обоих уровнях поддерживаются одинаковые типы объектов, на этих уровнях применяется один и тот же подъязык данных (обычно это язык SQL). Действительно, тот факт, что представление — это переменная

[5] Пример осуществления такой возможности приведен в главе 27.

отношения, так же важен для реляционных систем, как для математики важно то, что подмножество является множеством.

Примечание. Однако, по-видимому, в документации к реальным продуктам SQL и в самом стандарте языка SQL этот нюанс часто игнорируется (глава 4), поскольку нередко можно встретить упоминания о "таблицах и представлениях" (т.е. при этом подразумева ется, что представление не является таблицей). Советуем *не* делать этой распространен ной ошибки и использовать термин *таблицы* (или *переменные отношения)* лишь для обо значения *базовых* таблиц (или *базовых переменных отношения).*

Есть еще один заслуживающий внимания вопрос, касающийся базовых переменных отношения и представлений. Различие между базовой переменной отношения и представлением часто характеризуют следующим образом.

- Базовые переменные отношения "реально существуют" в том смысле, что они во площают в себе данные, которые действительно хранятся в базе данных.
- Представления, наоборот, "реально не существуют", а просто предоставляют раз личные способы просмотра "реальных" данных.

Однако такая характеристика, хотя и полезна в неформальном смысле, неточно отражает истинное положение дел. Действительно, пользователи могут *рассматривать* базовые переменные отношения как физически хранимые, поскольку фактически главная цель создания реляционных систем состоит в том, чтобы дать возможность пользователю работать с базовыми переменными отношения как с физически существующими, не заботясь о том, как эти переменные отношения физически представлены в памяти. Но (и это весьма существенное "но"!) подобные взгляды пользователей на то, как происходит обработка данных, *нельзя* толковать так, будто базовая переменная отношения — это непосредственно физически хранимая переменная отношения (например, как единственный хранимый файл). Как пояснялось в разделе 3.2, базовые переменные отношения лучше всего представлять как *абстракцию* для некоторого набора хранимых данных — абстракцию, скрывающую все детали уровня хранения данных. В принципе, базовая переменная отношения и ее хранимый эквивалент[6] могут различаться в произвольной степени.

Этот вопрос поможет прояснить простой пример. Снова рассмотрим базу данных отделов и служащих. В большинстве современных реляционных систем она, вероятно, была бы реализована в виде двух хранимых файлов, по одному для каждой переменной отношения базы данных. Но нет абсолютно никаких доводов против создания одного хранимого файла *иерархически* хранимых записей, каждая из которых состоит, во-первых, из номера отдела, названия и бюджета для некоторого отдела, и, во-вторых, вместе с ними заданы табельный номер служащего, фамилия и зарплата каждого служащего, работающего в этом отделе. Иначе говоря, для физического хранения данных может быть использован любой подходящий способ (дополнительные сведения приведены в приложении А), но на логическом уровне данные всегда должны выглядеть одинаково.

---

[6] Следующая цитата из одной недавно вышедшей книги демонстрирует некоторые недоразумения, обсуждаемые здесь, а также недоразумения, которые обсуждались в разделе 3.3: "Важно различать хранимые отношения, которые являются *таблицами,* и виртуальные отношения, которые являются *представлениями...* [Мы] будем использовать термин *отношение* только в тех случаях, когда вместо него можно использовать термин «таблица» или «представление». Если необходимо будет подчеркнуть, что данное отношение является хранимым отношением, а не представлением, то в этом случае мы будем использовать термин *базовое отношение* или *базовая таблица."* Подобные цитаты, к сожалению, — не редкость.

3.8. ТРАНЗАКЦИИ

Примечание. Тему этого раздела нельзя отнести исключительно к теме реляционных систем. Тем не менее, она здесь уместна, поскольку понимание основной идеи необходимо для усвоения некоторых понятий и логичного перехода к части II. Однако здесь данная тема намеренно рассматривается лишь поверхностно.

В главе 1 указывалось, что транзакция — это *логическая единица работы,* обычно включающая несколько операций над базой данных. Также отмечалось, что пользователь должен иметь возможность указать системе, что отдельные операции являются частью одной транзакции. Для этого используются операции BEGIN TRANSACTION, COMMIT и ROLLBACK. Как правило, транзакция начинается при выполнении операции BEGIN TRANSACTION и прекращается при выполнении операции COMMIT ИЛИ ROLLBACK, как в следующем примере, написанном на псевдокоде.

```
BEGIN TRANSACTION ;           /* Перевод денег со счета A на счет B
*/
UPDATE account A ;              /* Списание денег со счета A */
UPDATE account B ;              /* Зачисление денег на счет B */
IF <все выполнено успешно>
   THEN COMMIT ;          /* Нормальное завершение */
   ELSE ROLLBACK ;        /* Аварийное завершение */
END IF ;
```

Отметим некоторые свойства транзакций.

1. Транзакции заведомо **неразрывны,** т.е. они гарантируют (с логической точки зрения), что будут либо выполнены полностью, либо вовсе не выполнены[7], даже если в системе до окончания процесса выполнения транзакции произойдет сбой.
2. Транзакции гарантируют **сохранность результатов** их выполнения в том смысле, что если транзакция успешно выполнила оператор COMMIT, то все внесенные ею изменения гарантированно будут записаны в базе данных, даже если позднее в системе в какой-то момент произойдет сбой.

   Примечание. По сути, благодаря именно этому свойству сохранности результатов транзакций данные в базе данных являются *перманентными {постоянными)* в том смысле, который указан в главе 1.
3. Для транзакций также гарантируется **изолированность** одной транзакции от другой. Под этим подразумевается, что изменения в базе данных, внесенные некоторой транзакцией 77, станут видимыми для любой транзакции 72 исключительно после того, как транзакцией 77 будет успешно выполнена операция COMMIT. После выполнения операции COMMIT изменения, которые были произведены некоторой транзакцией, становятся видимыми для других транзакций. О таких изменениях говорят, что они *зафиксированы,* и гарантируется, что они никогда не будут отменены. Если же, напротив, транзакцией была выполнена операция ROLLBACK, все

---

[7] Поскольку транзакция — это выполнение определенного фрагмента кода, то выражение типа "выполнение транзакции" является фактически бессмысленным (если оно что-либо и означает, это значение по сути сводится к "выполнению выполнения"). Тем не менее, подобный способ словоупотребления является общепринятым и необходимым, и из-за отсутствия лучшего, мы сами будем им руководствоваться в данной книге.

изменения, которые были внесены в базу данных в процессе выполнения этой транзакции, будут отменены (выполнен *откат* изменений). В последнем случае конечный результат будет таким, как если бы данная транзакция вообще не начинала выполняться.

4. При параллельном выполнении нескольких транзакций, операции которых чередуются между собой, гарантируется, что процесс осуществления этих операций будет **упорядочиваемым** (serializable). Иначе говоря, результат выполнения каждой из транзакций будет точно таким же, как при строго последовательном выполнении всех этих же транзакций (без чередования операций, от начала и до конца) в некотором произвольном порядке.

Развернутое обсуждение всех остальных аспектов данной темы будет продолжено в главах 15 и 16.

3.9. БАЗА ДАННЫХ ПОСТАВЩИКОВ И ДЕТАЛЕЙ

В настоящей книге в большинстве примеров используется база данных, известная под названием базы данных **поставщиков и деталей.** Назначение настоящего раздела — ознакомить читателя с этой базой данных, которая будет служить примером для ссылок в следующих главах. На рис. 3.9 показано множество значений ее данных. Именно эти конкретные значения будут фактически использоваться в дальнейшем (где это имеет смысл)[8]. На рис. 3.10 показано определение базы данных, для которого снова используется язык Tutorial D (в этом языке ключевое слово VAR означает *переменная).* Обратите внимание на то, что несколько столбцов имеют типы данных, которым присвоено название, аналогичное названию соответствующего столбца. Столбцы STATUS и CITY определены как имеющие не пользовательский, а встроенный тип данных, соответственно, INTEGER (целое) и CHAR (строка символов произвольной длины). Наконец, необходимо отметить, что применительно к значениям, показанным в столбцах на рис. 3.9, должно быть сделано одно важное замечание, однако мы еще не готовы к этому. Поэтому обсуждение упомянутого замечания будет отложено до главы 5, точнее, до конца подраздела "Возможные форматы представления, селекторы и операторы THE_" в разделе 5.3.

Предполагается, что эта база данных имеет следующее назначение.

- Переменная отношения S представляет *поставщиков* (точнее, поставщиков, рабо тающих по контракту). Каждый поставщик имеет уникальный номер (S#); имя (SNAME), не обязательно уникальное (хотя оно может быть уникальным, как в слу чае, представленном на рис. 3.9); значение рейтинга или статуса (STATUS); место расположения (CITY). Предполагается, что для каждого поставщика может быть указан только один город.
- Переменная отношения P представляет *детали* (точнее, *виды* деталей). У каждого вида детали есть номер детали (P#), который является уникальным; название дета ли (PNAME); цвет (COLOR); вес (WEIGHT); город, где находится склад с деталями этого вида (CITY). Предполагается, что если вес детали имеет значение, то он указан в фунтах (ознакомьтесь также с тем, что сказано о единицах измерения в главе 5,

[8] Для тех читателей, которые знакомились с этими образцами данных в предыдущих изданиях, отметим, что деталь P3 передана из Рима в Осло. Такое же изменение внесено на рис. 4.5 в следующей главе.

раздел 5.4). Предполагается также, что каждый отдельный вид детали имеет только один цвет и хранится на складе только в одном городе.

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Рис. 3.9. База данных поставщиков и деталей (пример значений)

```
TYPE S# ... ;
TYPE NAME ... ;
TYPE P# ... ;
TYPE COLOR ... ;
TYPE WEIGHT ...
; TYPE QTY ... ;

VAR S BASE
    RELATION { S#
    S#,
      SNAME NAME,
      STATUS INTEGER,
      CITY CHAR
      PRIMARY KEY { S# } ;

VAR P BASE RELATION
    { P# P#,
                                    PNAME NAME,
      COLOR COLOR,
      WEIGHT
      WEIGHT, CITY
      CHAR }
                                      PRIMARY KEY { p# } ;

VAR SP BASE RELATION
     { S# S#,
        P# P#,
       QTY QTY }
        PRIMARY KEY {      S#, P# }
        FOREIGN KEY {      S# } REFERENCES S
        FOREIGN KEY {      P# } REFERENCES P ;
```

Рис. 3.10. База данных поставщиков и деталей (определение данных)

- Переменная отношения SP представляет *поставки*. Она в известном смысле служит для организации логической связи двух других переменных отношения. Например, первая строка переменной отношения SP на рис. 3.9 связывает поставщика с номером S1 из переменной отношения S с соответствующей деталью, имеющей номер P1 в переменной отношения P, т.е. представляет факт поставки деталей типа P1 поставщиком с номером S1 (а также указывает количество деталей — 300 штук). Таким образом, каждая поставка характеризуется номером поставщика (S#), номером детали (P#) и количеством (QTY). Предполагается, что в одно и то же время может быть выполнено не больше одной поставки для одного поставщика и одного вида деталей, поэтому для каждой поставки комбинация значений столбцов s# и P# уникальна с точки зрения набора текущих поставок, представленных в переменной отношения SP. Обратите внимание на то, что на рис. 3.9 с одним из поставщиков (с номером S5), не связано ни одной поставки.

Как отмечалось выше (в главе 1, раздел 1.3), детали и поставщиков можно рассматривать как **сущности,** а поставку — как **связь** между определенным поставщиком и определенной деталью. Однако в том же разделе было указано, что связь можно считать особым видом сущности. Одно из преимуществ реляционных баз данных состоит именно в том, что все сущности, независимо от того, что на самом деле они могут являться связями, представляются одним универсальным способом, а именно — с помощью строк, объединенных в отношения, как показано в нашем примере.

И еще несколько заключительных замечаний.

- Во-первых, наша база данных поставщиков и деталей исключительно проста, го раздо проще любой реальной базы данных, которая может встретиться на практи ке. Большинство реальных баз данных включает значительно больше сущностей и связей (и намного больше *видов* сущностей и связей). Но несмотря на это, пред ложенный здесь простой пример вполне подходит для иллюстрации большинства вопросов, обсуждаемых в оставшейся части книги, и (как уже отмечалось) будет использоваться как основа для большинства (но не для всех) примеров в несколь ких последующих главах.
- Во-вторых, безусловно, не было бы ошибкой, если бы мы использовали более описательные названия переменных отношения, подобные SUPPLIERS (постав щики), PARTS (детали) и SHIPMENTS (поставки), вместо сокращенных названий S, P и SP. Более того, на практике рекомендуется использовать именно такие опи сательные названия. Однако в нашем конкретном случае в последующих главах названия этих переменных отношения будут употребляться так часто, что целесо образнее использовать именно короткие названия.

3.10. РЕЗЮМЕ

На этом завершается краткий обзор реляционной технологии. Конечно, мы лишь слегка коснулись темы, ставшей в наши дни весьма обширным предметом изучения, но, как уже отмечалось, назначение данной главы — введение в более расширенное обсуждение, которое проводится далее. Но несмотря на это, нам удалось охватить немалую часть материала. Подведем итог обсуждению затронутых тем.

Реляционная база данных — это такая база данных, которая воспринимается ее пользователями как множество **переменных** (т.е. *переменных отношения* — relvar), значениями которых являются **отношения** или, менее формально, **таблицы. Реляционная система** — это система, которая поддерживает реляционные базы данных и операции над ними, включая, в частности, операцию сокращения **RESTRICT** (иначе называемую *выборкой,* **SELECT),** операцию проекции **PROJECT** и операцию соединения **JOIN.** Эти и подобные им операции, известные под названием *операций реляционной алгебры*[9]*,* выполняются на **уровне множеств.** Свойство **замкнутости** реляционных систем означает, что результат выполнения операции имеет тот же тип, что и объекты, над которыми проводилась операция (все они являются отношениями). А это, в свою очередь, позволяет использовать **вложенные реляционные выражения.** Значения переменных отношения изменяются с помощью операций **реляционного присваивания,** причем привычные нам операции обновления **INSERT, UPDATE** и **DELETE** можно считать сокращенной формой записи операций реляционного присваивания определенных типов.

Формальная теория, лежащая в основе реляционных систем, называется **реляционной моделью данных.** Реляционная модель представляет материал только на логическом уровне и не затрагивает физический уровень. В модели рассматриваются три принципиальных аспекта данных — их **структура,** сохранение их **целостности** и **манипулирование** данными. *Структурный* аспект касается *собственно* отношений, аспект *целостности* распространяется (помимо всего прочего) на **первичные** и **внешние ключи,** а аспект *манипулирования* данными связан с операторами (сокращения, проекции, соединения и т.д.). **Информационный принцип** (который, как теперь становится очевидно, следовало бы назвать **принципом единообразного представления)** гласит, что все информационное содержимое реляционной базы данных должно быть представлено одним и только одним способом, а именно — явным заданием значений, помещенных в позиции столбцов в строках отношений. В соответствии с этим принципом, единственными переменными, которые могут применяться в реляционной базе данных, являются переменные отношения.

Каждое отношение имеет **заголовок** и **тело;** заголовок — это набор пар "*имя_столбца: имя_типа",* а тело отношения состоит из набора строк, которые соответствуют заголовку. Заголовок любого отношения можно рассматривать как **предикат,** а каждую строку в теле отношения — как некоторое **истинное высказывание,** образованное в результате подстановки определенных значений **фактических параметров** соответствующего типа вместо **формальных параметров** этого предиката. Это определение распространяется и на основные, и на производные отношения, а также, в определенном смысле, на переменные отношения. Другими словами, типы — это то (множество чего-то), что может стать предметом обсуждения, а отношения — это то (множество чего-то), что можно сказать об этом предмете. И типы, и отношения **необходимы** и **достаточны** для представления любых данных (на логическом уровне).

Оптимизатор — это компонент системы, определяющий, как именно будут реализованы запросы пользователей (которые указывают, *что* делать, а не *как* делать). Таким образом, на реляционные системы возложена ответственность за навигацию по хранимой базе данных для поиска требуемых данных. Подобные системы иногда называют системами с **автоматической навигацией.** Оптимизация и автоматическая навигация

---

[9] Этот термин уже упоминался в формальном определении реляционной модели в разделе 3.2. Но в полной мере он будет использоваться только начиная с главы 6.

создают предпосылки для достижения реальной **независимости от физического представления данных.**

Каталог — это набор системных переменных отношения, содержащих **метаданные** о различных элементах, важных для системы (базовых переменных отношения, представлениях, индексах, пользователях и т.д.). Пользователи могут получать информацию из каталога теми же методами, которые они применяют для доступа к собственным данным.

Исходные переменные отношения в некоторой базе данных называются **базовыми переменными отношения,** а их значения называются **базовыми отношениями.** Отношение, которое получено из таких базовых отношений путем вычисления некоторого реляционного выражения, называется **производным** (базовые и производные отношения иногда называют **представимыми** отношениями). **Представление** — это переменная отношения, значение которой в любой данный момент является некоторым производным отношением (неформально представление можно рассматривать как **производную переменную отношения).** Значение такой переменной отношения в любой данный момент представляет собой результат вычисления соответствующего реляционного **выражения, определяющего это представление.** Поэтому можно сказать, что базовые переменные отношения *существуют независимо от других отношений,* а представления — нет, поскольку они зависят от соответствующих базовых переменных отношения. (Это можно сформулировать и иначе: базовые переменные отношения **автономны,** а представления — нет.) Пользователь может оперировать представлениями практически так же, как и базовыми переменными отношения (по крайней мере, теоретически). Система выполняет операции над представлениями, заменяя ссылку на название представления определяющим его выражением, т.е. преобразуя операцию над представлением в соответствующую операцию над базовыми переменными отношения.

Транзакция — это *логическая единица работы,* которая обычно включает выполнение нескольких операций базы данных. Выполнение транзакции начинается с выполнения оператора **BEGIN TRANSACTION** и завершается выполнением оператора **COMMIT** (нормальное завершение) или **ROLLBACK** (аварийное завершение). Транзакции обладают свойствами **неразрывности, сохранности результатов** и **изолированности.** При чередующемся выполнении операций нескольких параллельно обрабатываемых транзакций обычно гарантируется, что выполнение этих операций будет **упорядочиваемым** (как если бы все транзакции выполнялись от начала и до конца, одна за другой).

И наконец, в данной главе приведен основной пример книги — **база данных поставщиков и** деталей. Рекомендуется ознакомиться с ним именно сейчас, если вы еще не сделали этого. Необходимо знать, по крайней мере, какие в этой базе данных существуют столбцы, в каких переменных отношения они заданы, а также какие внешние и первичные ключи для них определены. (Безусловно, сами конкретные значения в таблицах не имеют большого значения!)

УПРАЖНЕНИЯ

3.1. Дайте определения следующим терминам:

автоматическая навигация	предикат
базовая переменная отношения	представление
внешний ключ	проекция
высказывание	производная переменная отношения

Замкнутость	реляционная база данных
Каталог	реляционная модель
операция на уровне множества	реляционная СУБД
оптимизация	соединение
откат	сокращение
первичный ключ	фиксация транзакции

3.2. Опишите содержимое переменных отношения каталога TABLE и COLUMN для базы данных поставщиков и деталей.

3.3. Как пояснялось в разделе 3.6, каталог должен описывать самого себя, т.е. включать записи о переменных отношения самого каталога. Дополните рис. 3.6 так, чтобы он включал необходимые записи о самих переменных отношения TABLE и COLUMN.

3.4. Ниже приведен запрос к базе данных поставщиков и деталей. Что получится в результате его выполнения? Какой предикат соответствует этому результату?

```
( ( S JOIN SP ) WHERE P# = P# CP2[1]) ) { S#, CITY }
```

3.5. Предположим, что выражение, применяемое в запросе из упр. 3.4, используется для определения представления.

```
CREATE VIEW V AS
   ( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#, CITY } ;
```

Теперь рассмотрим следующий запрос.

```
( V WHERE CITY = 'London' ) { S# }
```

Что получится в результате его выполнения? Какой предикат соответствует этому результату? Поясните, какой компонент используется со стороны СУБД при выполнении запроса.

3.6. Как вы понимаете термины, характеризующие свойства транзакций: неразрывность, сохранность результатов, изолированность и упорядочиваемость операций параллельных транзакций.

3.7. Сформулируйте *информационный принцип.*

3.8. Если вы знакомы с иерархической моделью данных, укажите все известные вам различия между ней и реляционной моделью, которая кратко описана в данной главе.

СПИСОК ЛИТЕРАТУРЫ

3.1. Codd E.F. Relational Database: A Practical Foundation for Productivity // CACM. — February 1982. — 25, № 2. (Переиздано: Robert L. Ashenhurst (ed.). ACM Turing Award Lectures: The First Twenty Years 1966—1985. — Reading, Mass.: Addison-Wesley, ACM Press Anthology Series. — 1987.)

Статья была представлена Коддом по случаю получения им награды ACM Turing Award в 1981 году за его работу над реляционной моделью. В ней обсуждается хорошо известная проблема *отставания разработок приложений.* Перефразируя ее,

можно сказать: "Потребности в приложениях для компьютеров быстро возрастают — настолько быстро, что отделы информационных систем (которые несут ответственность за написание приложений) отстают от этих потребностей все больше и больше". Существует два дополнительных метода разрешения этой проблемы.

1. Предоставить специалистам по информационным технологиям новые средства для повышения продуктивности их работы.
2. Предоставить пользователям возможность доступа непосредственно к базе данных, чтобы они могли полностью обойтись без специалистов по информационным технологиям.

Оба подхода необходимо развивать, причем в предлагаемой статье Кодда приводится обоснование того, что основу для обоих этих подходов дает применение реляционной технологии.

3.2. Date C.J. Why Relational? // C.J. Date. Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

Попытка составить краткую, но достаточно основательную сводку основных преимуществ реляционного подхода. Приведем следующее характерное высказывание из статьи: "Среди многочисленных преимуществ реляционного подхода существует одно, которое следует особо *подчеркнуть: наличие солидной теоретической базы".* Цитирую:"...реляционная модель действительно является иной. Она отличается тем, что не является *случайно выбранной.* Прежние же системы, напротив, имели произвольно выбранную организацию; они предоставляли решения для определенных задач того времени, но у них не было твердой теоретической базы, тогда как у реляционных систем такая база есть... а это означает, что [они] *надежны, как скала".* "Благодаря этому твердому основанию поведение реляционных систем отличается предсказуемостью и пользователь (возможно, не осознавая этого) держит в голове простую модель этого поведения, позволяющую ему предвидеть, что сделает система в той или иной ситуации. Сюрпризов быть не может (и не должно быть). Предопределенность означает, что пользовательский интерфейс прост для понимания, документирования, обучения, изучения, использования и запоминания".

3.3. Date C.J., Darwen H. Foundation for Object/Relational Databases: The Third Manifesto (2d edition). — Reading, Mass.: Addison-Wesley, 2000. См. также узел http: //www.thethirdmanifesto.com, где представлены некоторые формальные выдержки из этой книги, список обнаруженных опечаток и другие материалы по данной теме. Кроме того, к этой теме относится [20.1].

Третий манифест — это строгое, формальное и подробное предложение для выбора будущих направлений развития СУБД. Манифест можно рассматривать как абстрактный план проектирования СУБД и языка этой СУБД. Данный план основан на классических фундаментальных понятиях **тип, значение, переменная** и **оператор.** Например, у нас может быть тип INTEGER; целое число 3 может быть значением этого типа; N может быть переменной этого типа, значение которой в каждый момент — это некоторое целое значение (т.е. некоторое значение этого типа); знак "+" **может быть оператором,** применяемым к целым значениям (т.е. к значениям

того типа). Следует отметить, что в этой книге особое внимание уделено исследованию проблемы типов. Об этом свидетельствует даже ее подзаголовок: "Подробное изучение влияния теории типов на реляционную модель данных, включая всеобъемлющую модель наследования типов". Особый оттенок указанной направленности книги придает то, что теория типов и реляционная модель являются в определенной степени независимыми друг от друга. Точнее, реляционная модель не предписывает поддержку каких-либо конкретных типов (кроме логического); в ней лишь предусмотрено, что атрибуты отношений должны иметь некоторый тип; таким образом, подразумевается, что должны поддерживаться некоторые (неопределенные) типы.

Термин *переменная отношения* (relation variable, или relvar) взят из этой книги. В связи с этим отметим, что в книге "Третий Манифест" сказано также следующее: "В первой версии этого Манифеста проводилось различие между значениями базы данных и переменными базы данных, аналогичное различию между значениями отношения и переменными отношения. Кроме того, в ней было введен термин *dbvar* как сокращение от *database variable* (переменная базы данных). Хотя мы все еще полагаем, что указанное выше различие остается в силе, мы пришли к выводу, что оно почти не имеет непосредственного значения при обсуждении других, более важных аспектов этих предложений. Поэтому мы решили, в интересах использования знакомого всем языка, возвратиться к более традиционной терминологии". Но впоследствии оказалось, что это решение было неправильным... Как указано в [23.4]: "Мы сейчас можем констатировать, что было бы намного лучше преодолеть временные сложности и принять более правильные с точки зрения логики термины *значение базы данных* (database value) и *переменная базы данных* (database variable, или dbvar), несмотря на то, что они еще не нашли достаточно широкого распространения". В настоящей книге автор все еще придерживается привычного термина *база данных,* но принял решение об этом не в (полном) согласии со своими внутренними убеждениями. Необходимо отметить также следующее. В книге "Третий Манифест" есть такие слова: "Мы [признаем], что мы действительно чувствуем определенную неловкость в связи с тем, что назвали *манифестом* документ, имеющий главным образом техническое содержание. Согласно словарю *Chambers Twentieth Century Dictionary,* манифест — это письменное объявление о намерениях, мнениях или мотивах некоторого лица или группы лиц (например политической партии). В отличие от этого, Третий Манифест... касается вопросов науки и логики, а не просто намерений, мнений или мотивов". Однако Третий Манифест был специально написан как ответ и возражение на два вышедших раньше документа: "Манифест объектно-ориентированных систем баз данных" [20.2], [25.1] и "Манифест систем баз данных третьего поколения" [26.44], поэтому выбор нами названия для своей книги был фактически предопределен чужим поступком.

3.4. C. J. Date: "Great News, The Relational Model Is Very Much Alive!", http:// www. dbdebunk. com (August 2000).

С тех пор как реляционная модель была впервые разработана в 1969 году, она подвергалась беспрецедентному количеству нападок во множестве различных публикаций. Один из последних примеров подобных публикаций имеет вполне типичное

для них название: "Great News, The Relational Model Is Dead!" (Прекрасные новости — реляционная модель наконец-то мертва!). Настоящая статья была написана в качестве опровержения такой *ПОЗИЦИИ.*

3.5. C. J. Date: "There's Only One Relational Model!", http://www.dbdebunk.com (February 2001).

Со времени своего появления в 1969 году реляционная модель стала объектом невероятного количества попыток неправильного толкования и искажения в бесчисленных публикациях. Характерным примером может служить глава одной недавно вышедшей книги, озаглавленная "Разные реляционные модели", первое предложение который гласит: "Больше не существует такого понятия, как единая реляционная модель для баз данных (именно так?!), как нет и одной лишь геометрии". Настоящая статья также была написана в качестве опровержения указанной позиции.

ГЛАВА 4

Введение в язык SQL

4.1. ВВЕДЕНИЕ

Как отмечалось в главе 1, SQL является стандартным языком для работы с реляционными базами данных и в настоящее время поддерживается практически всеми продуктами, представленными на рынке. Он был разработан в лаборатории IBM Research в начале 1970-х годов [4.9], [4.10]. Первой серьезной реализацией этого языка был продукт-прототип System R компании IBM **[4.1]—[4.3],** [4.12]—[4.14]; впоследствии он был реализован в многочисленных коммерческих продуктах как компании IBM [4.8], [4.14], [4.21], так и других изготовителей. В этой главе представлено введение в язык SQL, а дополнительные аспекты, касающиеся таких вопросов, как целостность, защита и т.п., обсуждаются в последующих главах, специально посвященных этим темам. При обсуждении языка SQL, если не указано иное[1], мы будем основываться на текущей версии стандарта (т.е. **SQL:1999).** В [4.23] приведена формальная спецификация SQL: 1999; а в [4.24] можно найти значительное количество исправлений и дополнений к этой спецификации.

Примечание. Предыдущей версией стандарта была SQL: 1992, а версия SQL: 1999 предназначалась для использования в качестве совместимого расширения этой предыдущей

[1] После утверждения новой версии стандарта ("SQL:2003"), работа над которой началась в 2002 году и должна быть завершена в 2004 году, мы в следующих изданиях будем также иногда явно ссылаться на эту версию.

версии. Однако пока можно со всей уверенностью утверждать только то, что в наши дни ни один программный продукт не поддерживает полностью даже **SQL:** 1992; вместо этого такие продукты, как правило, поддерживают то, что можно было бы назвать "надмножеством подмножества" стандарта (либо SQL: 1999, либо, с большей вероятностью, SQL: 1992). Вернее, большинство продуктов не поддерживают некоторые средства, обусловленные стандартом, и в то же время предлагают другие средства, которые не определены этим стандартом[2]. Например, программный продукт DB2 компании **IBM** не поддерживает все стандартные средства обеспечения целостности, но вместе с тем предусматривает возможность использовать некоторые операторы для переименования базовых таблиц, которые не определены в стандарте. И еще несколько предварительных замечаний.

- Язык SQL первоначально разрабатывался конкретно как подъязык данных (см. главу 2). Однако после включения в стандарт в конце 1996 года такого средства, как **постоянные хранимые модули SQL** (SQL Persistent Stored Modules — SQL/PSM, или сокращенно PSM), стандарт SQL стал полностью поддерживать все вычисли тельные конструкции (и сейчас в нем предусмотрены процедурные операторы, например CALL, RETURN, SET, CASE, IF, LOOP, LEAVE, WHILE, REPEAT, а также несколько связанных с ними функциональных возможностей, например, можно использовать переменные и обработчики исключительных ситуаций). Более под робное описание модулей PSM выходит за рамки данной книги, но подробное ин структивное руководство можно найти в [4.20].
- В языке SQL вместо терминов *отношение* и *переменная отношения* используется термин **таблица,** а вместо терминов *кортеж* и *атрибут* — **строка** и **столбец.** Именно эти термины используется в стандарте языка SQL и в поддерживающих его продуктах, поэтому в соответствии с ними мы будем использовать указанные термины в данной главе (и везде, где речь идет о языке SQL).
- Необходимо подчеркнуть, что SQL — язык очень большого объема. Его специфи кация [4.23] содержит свыше 2000 страниц, не считая больше 300 страниц исправ лений в [4.24]. Поэтому в книге, подобной этой, невозможно дать исчерпывающее описание языка. Достаточно полно мы сможем рассмотреть лишь самые важные его особенности.
- Наконец, нельзя не сказать о том (как уже неоднократно отмечалось в главах 1—3), что языку SQL еще очень далеко до совершенного реляционного языка. В нем много недостатков, появившихся в результате как недоделок, так и переделок. Од нако как бы там ни было, это — стандарт, он поддерживается практически всеми продуктами, представленными на рынке, и поэтому каждый специалист по базам данных должен быть знаком с этим языком, по крайней мере, в определенном объеме, поэтому он и рассматривается в данной книге.

---

[2] На самом деле ни один из продуктов, по-видимому, и не смог бы в полной мере поддерживать этот стандарт, поскольку в нем на сегодняшний день содержится множество расхождений, ошибок и противоречий (о чем свидетельствуют [4.23] и [4.24]). Подробно этот вопрос рассматривается в [4.20].

4.2. ОБЗОР ЯЗЫКА SQL

В языке SQL имеются операции как определения данных, так **и** манипулирования ими. Сначала мы познакомимся с операциями **определения** данных. На рис. 4.1 показано, как с помощью средств языка SQL определяется база данных поставщиков и деталей (ср. с рис. 3.09 в главе 3). Как можно видеть, определение включает по одному оператору CREATE TYPE для каждого из шести определяемых пользователем типов (User-Defined Type — **UDT) и** по одному оператору CREATE TABLE для каждой из трех базовых таблиц (как было указано в главе 3, ключевое слово TABLE в операторе CREATE TABLE обозначает именно *базовую* таблицу). Каждый оператор CREATE TABLE задает имя создаваемой базовой таблицы, имена и типы данных столбцов этой таблицы, а также первичный ключ таблицы и любые внешние ключи, присутствующие в ней (кроме того, может быть указана другая дополнительная информация, которая не показана на рис. 4.1). Приведем еще пару замечаний по синтаксису.

```
CREATE TYPE      S# ... ;
CREATE TYPE      NAME ... ;
CREATE TYPE      P# ... ;
CREATE TYPE      COLOR ... ;
CREATE TYPE      WEIGHT ... ;
CREATE TYPE      QTY ... ;

CREATE TABLE S
     ( S# S#,
     SNAME NAME, STATUS
     INTEGER, CITY
     CHAR(15), PRIMARY
     KEY ( S# ) ) ;

CREATE TABLE P
     ( P# P#,
     PNAME NAME, COLOR
     COLOR, WEIGHT
     WEIGHT, CITY CHAR(15)
     , PRIMARY KEY { P# )
     ) ;

CREATE TABLE SP
     ( S# S#,
       P# P#,
       QTY QTY,
     PRIMARY KEY ( S#, P# ), FOREIGN
     KEY ( S# ) REFERENCES S, FOREIGN
     KEY ( P# ) REFERENCES P ) ;
```

Рис. 4.1. Определение базы данных поставщиков и деталей средствами языка SQL

- Обратите внимание, что символ "#", который иногда используется в книге, например, в названиях типов и в именах столбцов, на самом деле не определен в стандарте SQL.

- В качестве признака конца оператора мы здесь используем символ ";", хотя соглас но стандарту SQL, выбор используемого для этой цели символа зависит от реализа ции. Подробное рассмотрение данного вопроса выходит за рамки настоящей книги.
- При использовании встроенного типа CHAR в определении таблицы на языке SQL необходимо указывать соответствующую длину (на рис. 4.1 это значение равно 15).

Определив базу данных, можно начинать выполнять в ней различные операции, задаваемые с помощью операторов **манипулирования данными** языка SQL: SELECT, INSERT, UPDATE и DELETE. В частности, можно выполнять с данными реляционные операции сокращения, проекции и соединения, причем во всех этих случаях следует использовать один и тот же оператор манипулирования данными языка SQL — оператор **SELECT.** Некоторые примеры операций показаны на рис. 4.2.

Примечание. Пример операции соединения на этом рисунке подтверждает, что в языке SQL иногда необходимо использовать **уточненные имена** (например s. S#, SP. S#), позволяющие устранить неоднозначность при указании столбцов. Согласно общему правилу (хотя есть и исключения), уточненные имена допустимы всегда, а неуточненные имена допустимы, только если при их использовании не возниьает неоднозначность.

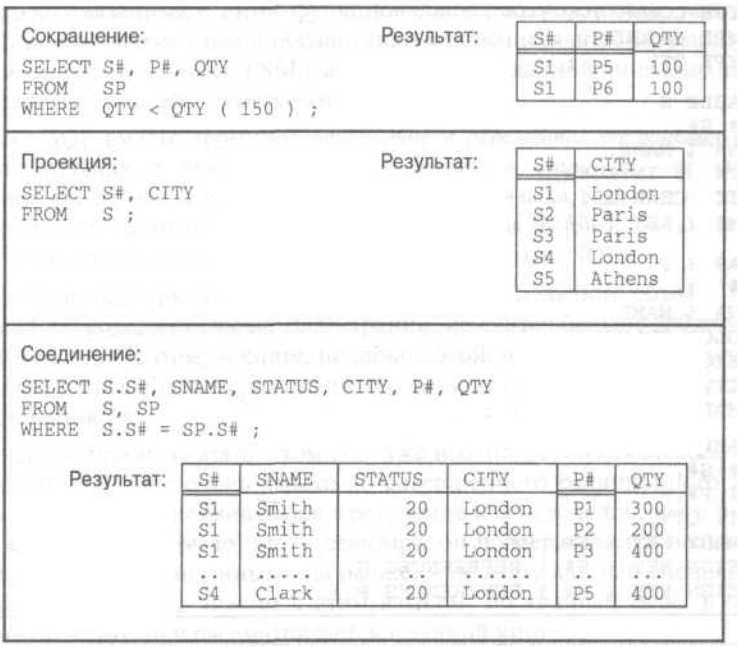


Рис. 4.2. Примеры выполнения операций выборки, проекции и соединения на языке SQL

Отметим, что в языке SQL поддерживается сокращенная форма предложения SELECT, как показано в следующем примере.

```
SELECT *;          /* или SELECT S.* (т.е. символ "*" может быть
                   уточнен с помощью */
FROM   S           /* точечного обозначения) */
```

В результате выполнения этого запроса будет получена копия всей таблицы S. Символ "*"_ это сокращенное обозначение списка имен столбцов, разделенного запятыми (формальное определение этого понятия приведено в разделе 4.6). Во-первых, подразумевается, что в этом списке содержатся заданные слева направо имена всех столбцов первой таблицы, указанной в конструкции FROM, в том порядке, в котором эти столбцы определены в указанной таблице. Во вторых, за именами столбцов первой таблицы в нем следуют заданные слева направо имена всех столбцов второй таблицы, указанной в конструкции FROM, в том порядке, в котором эти столбцы определены в указанной таблице (и т.д., применительно ко всем остальным таблицам, приведенным в конструкции FROM).

Примечание. Согласно стандарту SQL, выражение SELECT * FROM *m,* где г— это имя таблицы, может быть сокращено до простого выражения TABLE г.

Значительно подробнее оператор SELECT обсуждается в главе 8 (раздел 8.6).

Перейдем к операциям **обновления.** Примеры операций INSERT, UPDATE и DELETE языка SQL приведены в главе 1. Однако во всех примерах этой главы намеренно использовались только операции обработки отдельных строк. Тем не менее, операции **INSERT, UPDATE** и **DELETE,** как и операция SELECT, обрабатывают данные на *уровне множеств* (некоторые упражнения и ответы к ним в главе 1 действительно демонстрировали эту возможность). Вот несколько примеров обновления на уровне множеств для базы данных поставщиков и деталей.

```
INSERT
INTO   TEMP  (   P#, WEIGTH )
       SELECT P#, WEIGTH
       FROM   P
       WHERE  COLOR = COLOR('Red') ;
```

В этом примере подразумевается, что предварительно создана другая таблица TEMP с двумя столбцами, P# и WEIGTH. Оператор INSERT вставляет в нее номера деталей и соответствующие веса всех деталей с цветом ' Red' (красный).

```
DELETE FROM
SP WHERE  P#
=
```

Этот оператор DELETE удаляет из таблицы SP все строки с информацией о поставках детали с номером ' P2 '.

```
UPDATE S
SET    STATUS = 2 * STATUS ,
       CITY = 'Rome'
WHERE  CITY = 'Paris' ;
```

Приведенный выше оператор UPDATE увеличивает в два раза код статуса всех поставщиков в Париже и вносит в базу данных информацию о том, что эти поставщики переехали в Рим.

Примечание. В язык SQL не включен прямой аналог операции **реляционного присваивания.** Но эту операцию можно эмулировать, сначала удалив все строки из целевой таблицы, а затем выполнив для нее операции INSERT ... SELECT ... (как это сделано выше, в первом примере).

4.3. КАТАЛОГ

Стандарт SQL включает спецификации стандартного каталога, именуемого в нем **информационной** схемой. Знакомые нам термины *каталог* и *схема* действительно используются в языке SQL, но с особым смыслом, характерным только для языка SQL. Вообще говоря, **каталог** в языке SQL состоит из дескрипторов (метаданных) для отдельной базы данных[3], а схема состоит из дескрипторов той части базы данных, которая принадлежит отдельному пользователю. Другими словами, в системе может быть любое число каталогов (по одному для каждой базы данных), каждый из которых делится на произвольное число схем. Однако каждый каталог должен содержать одну и только одну схему с именем INFORMATION_SCHEMA (информационная схема), которая с точки зрения пользователя и является схемой (как уже указывалось), выполняющей функции обычного каталога.

Таким образом, информационная схема состоит из набора таблиц SQL, содержимое которых фактически отражает (точно заданным способом) все определения из всех остальных схем рассматриваемого каталога. Точнее говоря, информационная схема по умолчанию содержит набор представлений гипотетической "схемы: определения". Для поддержки схемы определения реализация не требуется, но она требуется, во-первых, для поддержки *некоторого* вида "схемы определения", и, во-вторых, для поддержки представлений такой "схемы определения", которые имеют вид, подобный представлениям информационной схемы. Необходимо отметить следующие особенности информационной схемы.

1. Основная причина того, что требования состоят из двух отдельных требований, сформулированных выше, заключается в том, что существующие продукты, ко нечно, поддерживают нечто близкое к "схеме определения". Однако от одного продукта к другому спектр различий таких схем будет очень широким (даже если эти продукты выпущены одним изготовителем). Поэтому имеет смысл требовать лишь то, чтобы в реализации поддерживались заранее регламентированные пред ставления "схемы определения".
2. На самом деле, речь может идти только об "одной из информационных схем", а не просто об "информационной схеме", поскольку, как мы убедились, такая схема имеется в каждом каталоге. Поэтому в общем случае все данные, которые доступны некоторому пользователю, *не* будут описаны с помощью единственной информа ционной схемы. Однако, чтобы упростить изложение материала, мы будем про должать считать, что в действительности существует только одна оема.

---

[3] Чтобы быть более точным, необходимо отметить, что в стандарте языка SQL такое понятие, как "база данных", в действительности просто отсутствует! По определению каталогом описывается то, что называется набором данных и зависит от реализации. Однако нет никаких разумных аргументов, запрещающих подразумевать под этим понятием базу данных.

Нет смысла вдаваться здесь в детали содержимого информационной схемы. Просто приведем некоторые из наиболее важных представлений информационной схемы в надежде на то, что сами названия позволят читателю получить некоторое понятие о содержимом этих представлений. Следует также отметить, что представление TABLES содержит информацию обо *всех* именованных таблицах, как базовых таблицах, так и представлениях, в то время как представление VIEWS содержит информацию только о представлениях.

- ASSERTIONS. Утверждения.
- CHECK_CONSTRAINTS. Проверочные ограничения.
- COLUMN_PRIVILEGES. Привилегии ДЛЯ столбцов.
- COLUMN__UDT_USAGE. Применение определяемого пользователем типа данных столбца.
- COLUMNS. Столбцы.
- CONSTRAINT_COLUMN_USAGE. Использование столбца ограничения.
- CONSTRAINT_TABLE_USAGE. Использование таблицы ограничений.
- KEY_COLUMN_USAGE. Использование столбца ключа.
- REFERENTIAL_CONSTRAINTS. Ссылочные Ограничения.
- SCHEMATA. Схемы.
- TABLE_CONSTRAINTS. Ограничения для таблицы.
- TABLE_PRIVILEGES. Привилегии для таблиц.
- TABLES. Таблицы.
- UDT_PRIVILEGES. Привилегии на применение типов данных, определяемых поль зователем.
- USAGE_PRIVILEGES. Привилегии на использование.
- USER_DEFINED_TYPES. Типы данных, определяемых пользователем.
- VIEW_COLUMN_USAGE. Использование столбца представления.
- VIEW_TABLE_USAGE. Использование таблицы представления.
- VIEWS. Представления.

В [4.20] этот вопрос рассмотрен более подробно, в частности в этой работе описано, как формулировать запросы для получения данных из информационной схемы (а это не так просто, как можно было бы ожидать).

4.4. ПРЕДСТАВЛЕНИЯ

Приведем пример определения представления на языке SQL.

```
CREATE VIEW GOOD SUPPLIER
    AS SELECT S#, STATUS,
       CITY FROM S WHERE
       STATUS > 15 ;
```

А ниже приведен пример запроса **SQL к** этому **представлению.**

```
SELECT S#, STATUS
FROM  GOOD SUPPLIER
WHERE  CITY = 'London'
```

Подставив определение представления вместо ссылки на имя представления, получим выражение, которое будет подобно приведенному ниже (обратите внимание на **вложенный подзапрос** в предложении FROM).

```
GOOD SUPPLIER. S# , GOOD SUPPLIER
STATUS FROM ( SELECT  S#, STATUS,
CITY
       FROM    S
          WHERE   STATUS > 15 ) AS
GOOD SUPPLIER WHERE  GOOD SUPPLIER.CITY =
'London' ;
```

Это выражение может быть затем упрощено, например, следующим образом.

```
SELECT S#, STATUS
FROM   S
WHERE  STATUS > 15
AND    CITY =
'Lond'
```

В последнем случае показан текст запроса, который фактически будет выполняться. В качестве второго примера рассмотрим следующую операциюБЕЬЕТЕ.

```
DELETE
FROM   GOOD SUPPLIER
WHERE  CITY = 'London' ;
```

Запрос на удаление, который будет фактически выполнен, выглядит следующим образом.

```
DELETE
FROM   S
WHERE  STATUS > 15
AND    CITY = 'London' ;
```

4.5. ТРАНЗАКЦИИ

SQL включает непосредственные аналоги операторов BEGIN TRANSACTION, COMMIT и ROLLBACK (см. главу 3), именуемые, соответственно, START TRANSACTION, COMMIT WORK и ROLLBACK WORK (ключевое слово WORK не является обязательным).

4.6. ВНЕДРЕНИЕ ОПЕРАТОРОВ SQL

В большинстве продуктов SQL операторы языка SQL могут выполняться как **непосредственно** (т.е. интерактивно, с подключенного терминала), так и в виде части прикладной программы (т.е. операторы SQL могут быть **внедренными,** а значит, могут смешиваться с операторами базового языка этой программы). Приложения, использующие внедренные операторы SQL, могут быть написаны на многих базовых языках; стандарт SQL включает поддержку для Ada, C, COBOL, Fortran, Java, M (прежде известного как MUMPS), Pascal и PL/I. Рассмотрим особенности технологии внедрения операторов SQL более подробно.

Фундаментальный принцип, лежащий в основе технологии внедрения операторов **SQL,** называется **принципом двухрежимности.** Он заключается в том, что *любое выражение **SQL,** которое можно использовать интерактивно, можно применять и путем внедрения **в** прикладную программу.* Конечно, существует множество различий в деталях между интерактивными операторами **SQL** и их внедренными аналогами. В частности, операции выборки требуют существенной дополнительной обработки в вычислительной среде базового языка (подробности приведены ниже в этом же разделе). Тем не менее, сам принцип двухрежимности всегда соблюдается. (Обратное, между прочим, не верно, т.е. существует несколько внедряемых операторов SQL, которые не могут использоваться интерактивно, о чем речь пойдет дальше в этой главе.)

Прежде чем начать обсуждение конкретных внедряемых операторов **SQL,** необходимо обсудить некоторые детали. Большинство из них иллюстрируется фрагментом программы, представленным на рис. 4.3. (Для закрепления наших представлений будем считать, что базовым языком является PL/I. Большинство приводимых примеров транслируется на другие базовые языки лишь с незначительными изменениями.)

Рассмотрим этот фрагмент программы.

1. *Внедренные* операторы **SQL** предваряются инструкцией **EXEC SQL,** так что их легко отличить от других операторов базового языка, и заканчиваются специальным **завершающим символом** (для языка PL/I таковым является точка с запятой "; ").

2. *Выполняемый* оператор **SQL** (далее и до конца этого раздела уточняющее слово *внедренный* обычно не будет применяться) может быть в программе везде, где могут находиться выполняемые операторы базового языка. Обратите внимание на уточняющее слово *выполняемый:* в отличие от интерактивного режима использования языка SQL, режим внедрения операторов **SQL** подразумевает включение в программу отдельных операторов **SQL,** которые являются чисто декларативными, а не выполняемыми. Например, оператор DECLARE CURSOR — это не выполняемый оператор (подробности приводятся в разделе "Операции, в которых используются курсоры"); таковыми не являются и операторы BEGIN и END DECLARE SECTION (см. п. 5 этого списка), а также оператор WHENEVER (см. п. 9).

```
EXEC SQL BEGIN DECLARE SECTION ;
   DCL SQLSTATE CHAR(5) ; DCL P#
   CHAR(6) ; DCL WEIGHT FIXED
   DECIMAL(5,1) ;

EXEC SQL END DECLARE SECTION ;

P# = ' P2 ' ;      /* Рассматривается в качестве
примера */ EXEC SQL SELECT P.WEIGHT
     INTO :WEIGHT
     FROM P
     WHERE P.P# = P# ( :P# ) ; IF SQLSTATE = '00000' THEN ...
;       /* WEIGHT - значение, полученное путем выборки */
ELSE ... ;       /* Возникло некоторое исключение */
```

Рис. 4.3. Фрагмент программы на языке PL/I с внедренными операторами языка SQL

3. Операторы SQL могут включать ссылки на **базовые переменные** *(т.е.* переменные базового языка). Подобные ссылки должны иметь **префикс в виде двоеточия,** позволяющий отличить их от имен столбцов таблиц SQL. Базовые переменные могут применяться во внедренных операторах SQL везде, где в интерактивном языке SQL могут использоваться литералы. Они могут также находиться в предложении INTO операторов SELECT (см. п. 4) и FETCH (подробности — в разделе "Операции, в которых используются курсоры"), определяющем результирующие переменные для размещения результатов выборки данных.
4. Обратите внимание на конструкцию **INTO** оператора SELECT, представленного на рис. 4.3. Назначение этой конструкции (как только что отмечалось) — указать ре зультирующие (целевые) переменные, в которых будут возвращены выбранные зна чения. Каждая /-я целевая переменная, указанная в предложении INTO, соответству ет /-му извлекаемому значению, указанному в списке выборки предложения SELECT.
5. Все базовые переменные, на которые ссылаются внедренные операторы SQL, долж ны быть определены в разделе объявлений **внедренного языка SQL,** который ограни чивается операторами **BEGIN DECLARE SECTION И END DECLARE SECTION**
(в PL/I для этого используется операторDCL).
6. Каждая программа, содержащая внедренные операторы SQL, должна включать ба зовую переменную с именем **SQLSTATE.** После выполнения любого присутствую щего в программе оператора SQL в эту переменную возвращается код состояния. В частности, код состояния 00000 означает, что оператор был выполнен успешно, а код состояния 02000 — что оператор был выполнен, но никаких удовлетворяю щих запросу данных найдено не было (другие значения указаны в [4.23]). Таким образом, выполнение в программе каждого оператора SQL должно завершаться проверкой значения переменной SQLSTATE и, если это значение будет отличаться от ожидаемого, должны предприниматься соответствующие действия. На практи ке, однако, такая проверка обычно выполняется неявно (см. п. 9).
7. Каждая переменная базового языка должна иметь **тип данных,** соответствующий значениям, для хранения которых эта переменная используется. В частности, базовая переменная, используемая в качестве целевой (например, для хранения результатов операции SELECT), должна иметь тип данных, совместимый с типом выражения, значение которого присваивается этой целевой базовой переменной. Аналогично, если базовая переменная служит источником (например, для опера ции INSERT), она должна иметь тип данных, совместимый с типом SQL того столбца, которому присваивается значение из этого источника. Но эта тема явля ется гораздо более сложной по сравнению со сведениями, приведенными в данной главе, поэтому она здесь не рассматривается (по крайней мере, достаточно под робно), а ее дальнейшее обсуждение будет продолжено в главе 5, раздел 5.7.
8. Базовые переменные для столбцов таблиц SQL могут иметь те же имена, что и имена соответствующих столбцов.
9. Как уже упоминалось, выполнение каждого оператора SQL, в принципе, должно сопровождаться проверкой значения, возвращаемого в переменной SQLSTATE. Для упрощения этого процесса предназначен оператор **WHENEVER,** который имеет следующий синтаксис.

EXEC SQL WHENEVER <*condition*> <*action*> }',

Здесь параметр <*condi tion*> (условие) может принимать значение NOT FOUND (данные не найдены), SQLWARNING (предупреждающее сообщение SQL) и SQLEXCEPTION (исключительная ситуация SQL) (другие условия включают определенные значения SQLSTATE и фиксируют нарушения заданных ограничений целостности), а параметр <*action*>— это либо оператор CONTINUE (продолжить), либо оператор GO TO (перейти к метке). Оператор WHENEVER не является выполняемым; это просто директива для компилятора SQL. Наличие в программе выражения "WHENEVER <*condition*> GO TO <*label*>*"* приведет к тому, что компилятор поместит оператор "IF <*condition*> GO TO <*label*> END IF" после каждого встретившегося ему выполняемого оператора SQL. Однако, встретив выражение "WHENEVER <*condition*> CONTINUE", компилятор SQL не вставляет в программу никаких операторов, и следовательно, программист должен будет вставить требуемые операторы вручную. Условия NOT FOUND, SQLWARNING и SQLEXCEPTION определены, как описано ниже.

- NOT FOUND показывает, что не найдены данные, соответствующие оператору; значение SQLSTATE = 02xxx;
- SQLWARNING означает, что возникла незначительная ошибка; значение SQLSTATE = Olxxx;
- SQLEXCEPTION означает, что возникла серьезная ошибка (исключительная си туация); значения SQLSTATE приведены в [4.23].

  Каждый оператор WHENEVER (для определенного условия), встреченный компилятором при последовательном просмотре текста программы, отменяет предыдущий (для этого условия).

10. Используя терминологию главы 2, отметим, что внедрение операторов SQL устанавливает *слабую связь* между средой SQL и базовым языком.

Итак, для предварительного обсуждения этого достаточно. В остальной части этого раздела главным образом рассматриваются операторы манипулирования данными. Как уже отмечалось, большинство из них можно использовать практически в неизменном виде (т.е. лишь с незначительными изменениями в синтаксисе). Однако операции выборки требуют отдельного описания. Проблема состоит в том, что такие операторы в общем случае выбирают не одну, а множество строк, в то время как процедурные базовые языки обычно не приспособлены для выборки больше одной строки за одно обращение. Следовательно, необходимо создать своего рода "мост" между предусмотренными в языке SQL средствами выборки, позволяющими получать одновременно множество строк, и применяемыми в базовом языке средствами обработки, допускающими одновременное использование только одной строки. В качестве подобного "моста" используются **курсоры.** Курсор представляет собой своего *рода логический указатель,* который может использоваться в приложении для перемещения по набору строк, указывая поочередно на каждую из них и таким образом обеспечивая возможность адресации этих строк — по одной за один раз. Однако временно отложим подробное обсуждение курсоров (до подраздела "Операции, в которых используются курсоры") и рассмотрим сначала такие операторы, для которых курсоры не требуются.

Операции, в которых не используются курсоры

Ниже перечислены операторы манипулирования данными, для которых не требуется использование курсоров.

- Однострочный оператор SELECT.
- INSERT.
- UPDATE (кроме формы CURRENT — см. следующий подраздел).
- DELETE (также кроме формы CURRENT — см. следующий подраздел).

Рассмотрим примеры каждого из этих операторов.

- **Однострочный оператор SELECT.** Получить статус и название города для поставщика, номер которого задан в базовой переменной GIVENS#.

```
EXEC SQL SELECT STATUS, CITY
     INTO :RANK, :TOWN FROM S
     WHERE S# = S# ( :GIVENS# )
     ;
```

  Термин *однострочный оператор SELECT* используется для обозначения выражения SELECT, значением которого будет таблица, содержащая не больше одной строки. В данном примере, если в таблице S существует одна и только одна строка, отвечающая заданному условию в конструкции WHERE, значения столбцов STATUS и CITY из этой строки в соответствии с запросом будут присвоены базовым переменным RANK и CITY, а переменной SQLSTATE будет присвоено значение 00000. Если в таблице S нет ни одной строки, отвечающей заданному условию WHERE, переменной SQLSTATE будет присвоено значение 02000. Если же таких строк окажется больше одной, будет зафиксирована ошибка и переменная SQLSTATE будет содержать ее код.

- **Оператор INSERT.** Вставить в таблицу P сведения о новой детали (номер детали, ее название и вес задаются содержимым базовых переменных P#, PNAME, PWT, соответственно; цвет детали и город неизвестны).

```
EXEC SQL INSERT
     INTO P ( P#, PNAME, WEIGHT )
     VALUES ( :P#, :PNAME, :PWT )
     ;
```

  Столбцам COLOR и CITY вновь добавляемой строки таблицы будут присвоены соответствующие *значения, применяемые по умолчанию.* Подробнее об этом речь пойдет в разделе 6.6 главы 6. Следует отметить, что по причинам, анализ которых выходит за рамки данной книги, применяемое по умолчанию значение для столбца, который имеет некоторый тип, определяемый пользователем, обязательно должно быть неопределенным (NULL). (Подробное обсуждение темы, касающейся неопределенных значений, откладывается до главы 19; однако нам неизбежно придется время от времени возвращаться к этому вопросу.)

- **Оператор DELETE.** Удалить сведения обо всех поставках для поставщиков из города, название которого помещено в базовую переменную CITY.

```
EXEC SQL DELETE
         FROM  SP
         WHERE :CITY =
              ( SELECT CITY FROM   S
                WHERE  S.S# = SP.S#
                ) ;
```

И снова, если нет строк, удовлетворяющих условию WHERE, переменной SQLSTATE присваивается значение 02000. Также обратите внимание на вложенный подзапрос (на этот раз в предложении WHERE).

- **Оператор UPDATE.** Увеличить статус всех поставщиков из Лондона на значение, помещенное в базовую переменную RAISE.

```
EXEC SQL UPDATE S
         SET    STATUS = STATUS +
         :RAISE WHERE  CITY = 'London[1]
         ;
```

Если в таблице поставщиков не будет найдено строк, удовлетворяющих условию WHERE, система присвоит переменной SQLSTATE значение 02000.

Операции, в которых используются курсоры

Теперь перейдем к вопросу о выборках на уровне множеств, т.е. о выборках не одной строки, как это было в случае однострочного оператора SELECT, а множества с произвольным количеством строк. Как указывалось ранее, в этой ситуации потребуется поочередный доступ к строкам выбранного множества, а механизмом такого доступа будет **курсор.** На рис. **4.4** этот процесс схематически проиллюстрирован на примере выборки информации о поставщиках (столбцы s#, SNAME и STATUS) для всех поставщиков из города, название которого задается в базовой переменной У.

```
EXEC SQL DECLARE X CURSOR FOR      /* Определить курсор   */
         SELECT  S.S#, S.SNAME, S.STATUS FROM    S
         WHERE   S.CITY = :Y
         ORDER   BY Si ASC ;

EXEC SQL OPEN X ;                   /* Выполнить запрос    */
         DO <для всех строк S, доступных через X> ;
            EXEC SQL FETCH X INTO :S#, :SNAME, :STATUS ;
                                    /* Получить данные о
                                    следующем поставщике   */

         END ;
EXEC SQL CLOSE X ;                  /* Перевести курсор X в
                                    неактивное состояние   */
```

Рис. 4.4. Выборка нескольких строк

Пояснение. Оператор DECLARE X CURSOR. . . определяет курсор x, связанный с *табличным выражением* (т.е. выражением, которое возвращает таблицу). Табличное выражение определяется оператором SELECT, который является частью всего выражения

DECLARE, но не вычисляется в этом месте программы, поскольку оператор DECLARE CURSOR— чисто декларативный. Табличное выражение *вычисляется* только при открытии курсора (оператор OPEN X). Далее для выборки строк из результирующего множества, по одной за один раз, используется оператор FETCH X INTO. . ., присваивающий извлеченные значения базовым переменным в соответствии со спецификациями в конструкции INTO. (Для простоты базовым переменным присвоены имена, совпадающие с именами соответствующих столбцов таблицы базы данных. Обратите внимание, что в операторе SELECT при определении курсора не задается собственная конструкция INTO.) Поскольку в результирующем наборе потенциально присутствует большое количество строк, оператор FETCH обычно вызывается в цикле. Цикл будет повторяться до тех пор, пока не закончатся строки в результирующем наборе. После выхода из цикла курсор x закрывается (оператор CLOSE X).

А теперь рассмотрим курсоры и операции с ними более подробно. Курсор определяется с помощью оператора **DECLARE CURSOR,** который имеет следующий общий вид.

```
EXEC SQL DECLARE <cursor name> CURSOR
        FOR <table exp> [ <ordering> ] ;
```

Для краткости несколько необязательных спецификаций в этом определении не указаны. Здесь параметр *<cursor name>* — это имя определяемого курсора. Необязательный параметр определения сортировки результата выборки *<ordering>* имеет следующий формат.

ORDER BY *<order item commalist>*

Здесь параметр *<order item commalist>* — разделенный запятыми список элементов *<order i tem>,* по которым должно быть выполнено упорядочение извлекаемых строк. Список должен содержать не меньше одного элемента *< order i tem>,* а в каждом элементе списка должно содержаться имя столбца (заметьте, неуточненное)[4], после которого может следовать необязательное служебное слово ASC (по возрастанию) или DESC (по убыванию). При отсутствии служебного слова по умолчанию принимается порядок по возрастанию (ASC). Если конструкция ORDER BY не определена, то принцип упорядочения определяется системой. (Фактически, последнее замечание остается в силе, если даже определена конструкция ORDER BY, по крайней мере, когда речь идет о строках с одним и тем же значением для указанного списка*< order item commalist>.)*

Примечание. Дадим определение удобному термину *разделенный запятыми список элементов (commalist).* Пусть *<xyz>* обозначает произвольную синтаксическую категорию (т.е. то, что находится слева от некоторого правила вывода в нотации BNF). Тогда выражение *<xyz commalist>* (или *<разделенный запятыми список xyz>)* обозначает последовательность от нуля и больше элементов *<xyz>,* в которой каждая пара элементов *<xyz>* разделена

---

[4] Фактически, имя столбца может быть уточнено, если заданное табличное выражение <table exp> соответствует довольно сложному набору правил. Эти правила были впервые введены в стандарте SQL: 1999, в котором также регламентированы правила, согласно которым элемент <order item> может иногда определять либо вычислительное выражение, например, ORDER BY A+B, либо имя столбца, который входит в состав таблицы результата, например, SELECT CITY FROM S ORDER BY STATUS. Изложение подробных сведений об этих правилах выходит за рамки настоящей книги.

запятой (и возможно также одним или несколькими пробелами). Обозначение в виде списка элементов, разделенного запятыми, будет широко использоваться в приводимых далее синтаксических правилах (причем во всех синтаксических правилах, а не только в правилах языка SQL).

Как утверждалось ранее, оператор DECLARE CURSOR — декларативный, а не выполняемый. Он предназначен для объявления курсора с определенным именем, для которого предусмотрено табличное выражение и с которым постоянно связан тип упорядочения. Табличное выражение может включать ссылки на базовые переменные. Программа может содержать любое количество операторов DECLARE CURSOR, каждый из которых должен быть, конечно, предназначен для определения разных курсоров.

Для работы с курсорами существует три выполняемых оператора: **OPEN, FETCH** и **CLOSE.**

- Оператор OPEN имеет следующий формат.

  ```
  EXEC SQL OPEN <cursor name>;
  ```

  Он предназначен для открытия или *активизации* указанного курсора (который в данный момент не должен быть открыт). В результате его выполнения вычисляется связанное с этим курсором табличное выражение (причем для всех базовых переменных, упоминаемых в этом выражении, используются текущие значения). В результате идентифицируется определенное множество строк, которое становится текущим **активным набором** для данного курсора. Курсор также устанавливает исходную *позицию* в этом активном наборе, а именно — **позицию** перед его первой строкой. Следует отметить, что активные наборы всегда рассматриваются как упорядоченные (см. приведенное выше описание конструкции ORDER BY), а значит, и понятие позиции имеет для них смысл[5].

- Оператор FETCH имеет следующий формат.

```
EXEC SQL FETCH <cursor name>
         INTO <host variable reference commalist> ;
```

  Здесь параметр *<host variable reference commalist>*—разделенный запятыми список ссылок на базовые переменные. Этот оператор служит для перемещения позиции указанного курсора (который должен быть уже открыт) к следующей строке в его активном наборе с последующим присваиванием значений столбцов этой строки базовым переменным, указанным в предложении INTO. Если после очередного вызова оператора FETCH на выполнение следующая строка отсутствует, то выборка данных не производится и переменной SQLSTATE присваивается значение 02 **000.**

- Оператор CLOSE имеет следующий формат.

  ```
  EXEC  SQL CLOSE  <cursor name>   ;
  ```

---

[5] Сами по себе множества, конечно, не являются упорядоченными (глава 6), так что "активный набор" — это на самом деле не множество как таковое. Его лучше представлять в виде *упорядоченного списка* или *массива* (строк).

Он служит для закрытия *(деактивизации)* указанного курсора (который должен быть в данный момент открытым). После его выполнения с курсором уже не будет связан активный набор. Однако в дальнейшем курсор вновь может быть открыт; при этом он снова получит активный набор — возможно, уже не такой, как раньше (в частности, если значения указанных в объявлении курсора базовых переменных к текущему моменту были изменены). Следует отметить, что изменение этих переменных при открытом курсоре не оказывает влияния на его активный набор.

Есть еще два оператора, в которых могут использоваться ссылки на курсоры, — это варианты операторов UPDATE и DELETE с конструкцией CURRENT. Если курсор (скажем, x) в данный момент позиционирован на определенную строку, то можно обновить или удалить эту "текущую строку курсора x", т.е. строку, на которую в данный момент позиционирован курсор x, например, как показано ниже.

```
EXEC SQL UPDATE S
         SET STATUS = STATUS +
         :RAISE WHERE CURRENT OF X
         ;
```

Выражения CURRENT операторов DELETE и UPDATE будут недопустимыми, если табличное выражение < *table exp*> в объявлении курсора определено с участием необновляемого представления, созданного с помощью оператора CREATE VIEW (подробности приведены в главе 10, раздел 10.6).

Динамический язык SQL и интерфейс SQL/CLI

В предыдущем разделе по умолчанию предполагалось, что данная конкретная программа (включая операторы SQL и все операторы базового языка) может быть полностью откомпилирована "заблаговременно" (т.е. до наступления этапа прогона) в том виде, в каком она реализуется на данный момент. Однако применительно к некоторым приложениям соблюдение такого условия нельзя гарантировать. Например, рассмотрим оперативное приложение. (Напомним, что, как отмечалось в главе 1, *оперативными* называются приложения, которые предоставляют пользователю доступ к базе данных с некоторого интерактивного терминала или подобного ему устройства.) Как правило, в этом приложении должны выполняться примерно такие действия, как показано ниже.

1. Принять с терминала команду пользователя.
2. Проанализировать поступившую команду.
3. Сгенерировать соответствующие операторы SQL для обращения к базе данных.
4. Возвратить сообщение и (или) полученные результаты на терминал.

Если набор команд пользователя, который программа может принять с терминала в шаге 1, достаточно мал (как, например, в случае обработки предварительных заказов мест на авиалиниях), то набор всех возможных выполняемых операторов SQL также будет невелик и его можно будет непосредственно внедрить в программу. В этом случае действия на втором и третьем этапах будут состоять в логической проверке введенной команды с последующим переходом к той части программы, которая выполняет заранее предопределенные операторы SQL. С другой стороны, если набор вводимых команд достаточно разнообразен, было бы непрактично заранее предопределять и внедрять в программу все

необходимые выражения **SQL,** соответствующие всем возможным командам. Вместо этого, вероятно, целесообразнее конструировать необходимые запросы SQL динамически, а затем динамически же компилировать и выполнять сконструированные запросы. Средства динамического языка **SQL,** описанные в этом разделе, предназначены для поддержки данного процесса.

Динамический язык SQL

Средства **динамического языка SQL** — это часть средств поддержки внедрения языка SQL. Они состоят из множества *динамических операторов* (компиляция самих этих операторов осуществляется заблаговременно), назначение которых состоит именно в том, чтобы обеспечивать поддержку трансляции и выполнения обычных операторов SQL, создаваемых на этапе прогона программы. Таким образом, существуют два основных динамических оператора SQL — PREPARE (Подготовить — по сути, *компилировать)* и EXECUTE (Выполнить). Их использование проиллюстрировано на следующем (нереальном по своей простоте, но достаточно точном) примере на языке PL/I.

```
DCL SQLSOURCE CHAR VARYING (65000) ;

SQLSOURCE = 'DELETE FROM SP WHERE QTY < QTY ( 3 00
)' ; EXEC SQL PREPARE SQLPREPPED FROM :SQLSOURCE ;
EXEC SQL EXECUTE SQLPREPPED ;
```

Пояснения

1. Имя SQLSOURCE идентифицирует переменную языка PL/I типа символьной строки переменной длины, в которой на этапе прогона программа определенным образом подготавливает исходную форму (т.е. представление в виде символьной строки) некоторого оператора SQL, в нашем конкретном примере — оператора DELETE.
2. Имя SQLPREPPED, напротив, идентифицирует переменную среды *SQL,* а не базового языка PL/I, которая будет (в принципе) использоваться для хранения компилированной формы оператора SQL (исходная форма которого представлена в переменной SQLSOURCE). Конечно, имена, подобные SQLSOURCE И SQLPREPPED, МОЖНО выбирать произвольно.
3. С помощью оператора присваивания SQLSOURCE =…; на языке PL/I переменной SQLSOURCE присваивается исходная форма оператора SQL DELETE. Конечно, на практике процесс формирования такого исходного оператора будет значительно сложнее и, возможно, в нем будут использоваться ввод и анализ некоторых элементов запросов от конечного пользователя, выраженных на естественном языке или в другой, более "дружественной" форме, чем обыкновенный язык SQL.
4. Оператор PREPARE извлекает это исходное выражение и подготавливает (т.е. компилирует) его, создавая выполняемую версию извлеченного им оператора, сохраняемую в переменной SQLPREPPED.
5. Оператор EXECUTE выполняет откомпилированную версию оператора из переменной SQLPREPPED, в результате чего осуществляется собственно операция DELETE. Информация SQLSTATE выполненного оператора DELETE возвращается так же, как при выполнении аналогичного оператора удаления, обычным сбразом.

Обратите внимание, что имя SQLPREPPED идентифицирует переменную языка SQL, а не **PL/I,** поэтому при его использовании в операторах PREPARE И EXECUTE двоеточие перед ним не указывается. Важно также, что подобные переменные SQL явно не объявляются.

Следует отметить, что описанный выше процесс в точности совпадает с процессом, который происходит, если выражения SQL вводятся интерактивно. Во многих системах имеется некоторое подобие процессора запросов SQL. Этот процессор в действительности — не что иное, как обобщенное интерактивное приложение, способное обрабатывать весьма широкий спектр вводимых команд, а именно — любой допустимый (или недопустимый!) оператор языка SQL. Для конструирования операторов SQL, соответствующих вводимым пользователем командам, для компиляции и выполнения сконструированных операторов и для возврата сообщений и результатов на терминал в нем используются именно средства динамического языка SQL.

Безусловно, средства динамической поддержки SQL не исчерпываются описанными выше операторами PREPARE И EXECUTE; например, в них предусмотрены механизмы параметризации подготавливаемых операторов и подстановки фактических параметров вместо формальных параметров перед выполнением динамически сформированных операторов; кроме того, имеются аналоги тех средств курсоров, которые описаны в предыдущем разделе. В частности, в состав этих средств входит оператор EXECUTE IMMEDIATE, который фактически позволяет объединить функции PREPARE И EXECUTE В ОДНОЙ операции.

Интерфейсы уровня вызовов SQL/CLI

Средства **интерфейса уровня вызовов SQL** (SQL Call-Level Interface — SQL/CLI, или сокращенно CLI) были введены в стандарт SQL в 1995 году. Интерфейс CLI в основном создан на базе интерфейса ODBC (Open Database Connectivity) компании Microsoft. И тот, и другой интерфейс позволяет приложениям, которые написаны на одном из базовых языков, выдавать запросы к базе данных, обращаясь к *процедурам CLI,* предоставляемым изготовителем, не прибегая к помощи внедренных операторов SQL. Затем в этих процедурах, которые предварительно должны быть обязательно связаны с данным приложением, используется динамический язык SQL для выполнения требуемых операций с базой данных от имени приложения. (Иными словами, с точки зрения СУБД процедуры CLI ничем не отличаются от обычного приложения.)

Как видим, интерфейс SQL/CLI (а также ODBC) решает ту же задачу, что и динамический язык SQL, а именно — позволяет приложению передавать на выполнение текст оператора SQL именно к тому времени, когда его непосредственно необходимо выполнять. Однако применяемый в интерфейсах CLI и ODBC подход к решению этой задачи организован лучше, чем в динамическом языке SQL. Его преимущества заключаются в следующем.

- Во-первых, динамический язык SQL — это *исходный код,* который должен соответствовать стандарту SQL. Поэтому для любого приложения, которое использует динамический язык SQL, требуется какой-то компилятор SQL, необходимый для обработки установленных стандартом операций, таких как PREPARE, EXECUTE и т.д. Интерфейсом CLI, напротив, нормированы лишь подробности *вызова процедур* (т.е. в основном вызова подпрограмм). Не требуются средства специального

компилятора; достаточно использовать обычный компилятор стандартного базового языка. Поэтому приложение может распространяться (возможно даже сторонними изготовителями программного обеспечения) в "сжатой" форме, в виде *объектного кода.*

- Во-вторых, такие приложения могут быть *независимыми от типа СУБД,* т.е. интерфейс SQL/CLI включает средства создания универсальных приложений (опять же, возможно, сторонними изготовителями программного обеспечения), которые могут использоваться для нескольких различных типов СУБД, без специализации с учетом какой-то конкретной СУБД.

Ниже в качестве иллюстрации использования интерфейса SQL/CLI приведен аналог примера динамического вызова SQL, который можно найти в предыдущем подразделе.

```
char sqlsource [65000] ;

strcpy ( sqlsource,
         "DELETE FROM SP WHERE QTY < QTY ( 300 )" ) ;
rc = SQLExecDirect ( hstmt, (SQLCHAR *)sqlsource,
SQL_NTS ) ;
```

Пояснение

1. Поскольку в реальных приложениях SQL/CLI базовым языком обычно служит С, в данном примере вместо PL/I используется С. Кроме того, в соответствии со спецификацией SQL/CLI в именах переменных, именах процедур и т.п. используются прописные буквы (или сочетание прописных и строчных букв), а не только прописные буквы, как во всех остальных именах в этой книге (кроме того, в этих пояснительных примечаниях имена выделены для наглядности моноширинным шрифтом). Необходимо также учитывать, что интерфейс SQL/CLI представляет собой набор стандартных средств для вызова подпрограмм из базового языка, поэтому его синтаксис (а также, безусловно, соответствующая семантика) изменяется в зависимости от базового языка.
2. Функция strcpy языка С вызывается для копирования исходной формы определенного оператора DELETE языка SQL в переменную sqlsource языка С.
3. Оператор присваивания С ("=") вызывает процедуру SQLExecDirect интерфейса SQL/CLI (аналог оператора EXECUTE IMMEDIATE динамического языка SQL) для выполнения оператора SQL, содержащегося в переменной sqlsource, и присваивает переменной rc программы С код возврата, полученный в результате этого вызова.

Как и можно было бы предположить, интерфейс SQL/CLI в той или иной степени включает аналоги всех средств динамического выполнения SQL, наряду с некоторыми дополнениями. Более подробное описание этого интерфейса выходит за рамки данной книги. Однако следует учитывать, что такие интерфейсы, как CLI, ODBC и JDBC (фактически вариант ODBC для языка Java), приобретают все более важное практическое значение по причинам, которые будут обсуждаться в главе21, раздел 21.6.

4.7. НЕСОВЕРШЕНСТВО ЯЗЫКА SQL

Как отмечалось в разделе 4.1 этой главы, язык SQL отнюдь нельзя назвать "совершенным" реляционным языком, поскольку он имеет много недостатков, вызванных многочисленными недоделками и переделками. Конкретные критические замечания будут представлены в следующих главах. Отметим лишь основной недостаток, который заключается в том, что в целом язык SQL, строго говоря, некорректно поддерживает реляционную модель. Поэтому возникает сомнение, действительно ли современные продукты SQL заслужили право называться реляционными. Фактически, насколько это известно автору, на сегодняшний день *на рынке нет ни одного продукта, который поддерживал бы реляционную модель в полном объеме*[6]. Мы не хотим этим сказать, что если современные продукты обходятся без каких-то элементов реляционной модели, то последние не очень важны; напротив, в модели важен *каждый элемент.* Более того, каждый из ее элементов важен исключительно по практическим соображениям. Нельзя не подчеркнуть тот непреложный факт, что назначение реляционной теории состоит не в том, чтобы быть просто "теорией ради теории". Вовсе нет, ее назначение — заложить основу для построения систем, которые *будут практически применимыми на все сто процентов.* Но, как это ни печально, со стороны изготовителей продуктов еще не сделано реальных шагов к решению проблемы реализации реляционной теории во всей ее полноте. В результате, с позволения сказать, "реляционные" продукты сегодняшнего дня все как один по тем или иным причинам оказываются неспособными реализовать преимущества, которые могут быть достигнуты в результате использования реляционной технологии в потном объеме.

4.8. РЕЗЮМЕ

На этом завершается знакомство с некоторыми важнейшими функциональными возможностями, предусмотренными стандартом языка SQL. Мы подчеркиваем тот факт, что язык SQL очень важен с коммерческой точки зрения, хотя, к сожалению, в определенной степени несовершенен с чисто реляционной точки зрения.

Язык SQL состоит из двух компонентов: языка **определения данных** (Data Definition Language — DDL) и языка **манипулирования данными** (Data Manipulation Language — DML). Язык манипулирования данными может применяться и на внешнем уровне (по отношению к представлениям), и на концептуальном уровне (по отношению к базовым таблицам). Аналогично, язык определения данных может использоваться для определения объектов на внешнем уровне (представления), концептуальном уровне (базовые таблицы), а в большинстве коммерческих систем (хотя это и не соответствует стандарту как таковому) даже на внутреннем уровне (по отношению к индексам и другим физическим структурам организации памяти). Кроме того, язык SQL предоставляет определенные возможности *управления данными,* т.е. возможности, которые нельзя отнести ни к языку DDL, ни к языку DML. Примером здесь может служить оператор GRANT, который одни пользователи могут применять для предоставления *привилегий доступа* другим пользователям системы (глава 17).

Было показано, как можно использовать язык SQL для создания базовых таблиц с помощью оператора **CREATE TABLE** (кроме того, был кратко рассмотрен оператор CREATE TYPE). Затем было приведено несколько примеров использования операторов

---

[6] Однако см. [20.1].

SELECT, INSERT, DELETE и **UPDATE** и, в частности, продемонстрировано, как можно применять оператор SELECT для реализации реляционных операций сокращения, проекции и соединения. Также некоторое внимание было уделено **информационной схеме,** состоящей из множества предопределенных представлений гипотетической "схемы определения", и возможностям языка SQL по работе с **представлениями** и **транзакциями.**

Значительная часть этой главы была посвящена **внедренным операторам SQL.** Основной замысел, лежащий в основе использования внедренных операторов SQL, называется **принципом двухрежимности,** т.е. принципом, в соответствии с которым (насколько это возможно) *любое выражение SQL, которое можно использовать интерактивно, можно внедрить и в прикладную программу.* Главное исключение из этого принципа имеет место в связи с **операциями многострочной выборки,** для которых требуется использовать **курсоры,** позволяющие преодолеть разрыв между возможностью выборки данных на уровне множеств в языке SQL и возможностью выборки данных на уровне строки в базовых языках программирования, таких, например, как PL/I.

Далее мы обсуждали главным образом вопросы синтаксиса, в том числе выяснили назначение переменной **SQLSTATE** и рассмотрели такие операторы, как **однострочный** оператор **SELECT** и операторы **INSERT, DELETE** и **UPDATE,** для которых курсор не нужен.

Затем мы обратились к операторам, для которых действительно *требуется* использование курсора, и обсудили операторы **DECLARE CURSOR, OPEN, FETCH, CLOSE,** а также операторы **DELETE** и **UPDATE** с конструкцией **CURRENT.** (В стандарте языка SQL форму этих операторов с конструкцией CURRENT называют, соответственно, *позиционным* оператором UPDATE и DELETE, а термин *поисковый* используют для других форм этих операторов, отличных от использующих CURRENT.) Наконец, мы кратко обсудили концепцию **динамического языка SQL,** в частности — операторы **PREPARE** и **EXECUTE,** а также кратко коснулись (в сравнении с ODBC и JDBC) назначения **интерфейса уровня вызовов SQL** (SQL Call-Level Interface — SQL/CLI).

УПРАЖНЕНИЯ

4.1. На рис. 4.5 показаны примеры значений данных для расширенной формы базы данных поставщиков и деталей, которая называется *базой данных поставщиков, деталей и проектов*[1]. Поставщики (S), детали (P) и проекты (J) однозначно определяются, соответственно, номером поставщика (S#), номером детали (P#) и номером проекта (J#). Значение предиката для отношения SPJ (поставки) таково: определенный поставщик S# поставляет определенную деталь P# для определенного проекта J# в определенном количестве QTY (причем комбинация значений столбцов {S#, P#, J#} представляет собой первичный ключ). Запишите соответствующие определения данных на языке SQL для этой базы данных.

Примечание. Эта база данных будет использоваться во многих упражнениях в последующих главах.

4.2. В разделе 4.2 был описан оператор CREATE TABLE так, как он определен в стандарте языка SQL. Однако многие коммерческие продукты поддерживают дополнительные опции этого оператора, обычно связанные с индексами, размещением

[7] Рис. 4.5 приведен на стр. 154 (а рис. 3.8 — на стр. 119).

на дисковом пространстве и другими вопросами реализации, что противоречит требованиям обеспечения физической независимости от данных и междусистемной совместимости. Исследуйте доступный вам продукт, поддерживающий язык SQL. Относятся ли предыдущие замечания к этому продукту? В частности, какие дополнительные опции оператора CREATE TABLE поддерживаются в этом про-

4.3. И снова исследуйте доступный вам продукт, поддерживающий язык SQL. Под держивается ли в нем информационная схема? Если нет, то каким образом под держивается каталог?

4.4. Сформулируйте на языке SQL следующие операции обновления для базы данных поставщиков, деталей и проектов.

а) Вставить данные нового поставщика S10 в таблицу S; имя поставщика — Smith, город — New York, статус еще не известен.
б) Удалить все проекты, для которых нет поставок.
в) Изменить цвет всех деталей красного цвета (red) на оранжевый (orange).

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

J

J#	JNAME	CITY
J1	Sorter	Paris
J2	Display	Rome
J3	OCR	Athens
J4	Console	Athens
J5	RAID	London
J6	EDS	Oslo
J7	Tape	London

SPJ

S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	100
S5	P3	J4	200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

Рис. 4.5. База данных поставщиков, деталей и проектов (значения приведены в качестве примера)

4.1. Используя базу данных поставщиков, деталей и проектов, напишите программу с внедренными выражениями SQL для выдачи списка всех строк поставщиков по порядку их номеров. За каждой строкой поставщика должны непосредственно

следовать строки проектов, обеспечиваемых этим поставщиком, по порядку номеров проектов.

4.2. Даны таблицы PART И PART_STRUCTURE, определенные следующим образом.

```
CREATE TABLE PART
     ( P# P#, DESCRIPTION CHAR(100),
       PRIMARY KEY ( P# ) )
; CREATE TABLE
PART_STRUCTURE
     ( MAJOR P# P#, MINOR P# P#, QTY QTY,
       PRIMARY KEY ( MAJOR P#, MINOR P# ),
       FOREIGN KEY ( MAJOR P# ) REFERENCES
       PART, FOREIGN KEY ( MINOR P# )
       REFERENCES PART ) ;
```

В таблице PART_STRUCTURE показано, какие детали (MAJOR_P#) содержат другие детали (MINOR_P#) как компоненты первого уровня. Напишите на языке SQL программу для получения списка всех компонентов данной детали на всех имеющихся уровнях (задача разузлования деталей). *Примечание.* Значения, показанные в качестве примера на рис. 4.6, могут помочь вам более наглядно представить предложенную выше задачу. Следует отметить, что таблица PART_STRUCTURE демонстрирует, как информация о *составе изделий* (см. главу 1, раздел 1.3, подраздел "Сущности и связи") обычно представляется в реляционных системах.

PART_STRUCTURE

MAJOR_P#	MINOR_P#	QTY
P1	P2	2
P1	P3	4
P2	P3	1
P2	P4	3
P3	P5	9
P4	P5	8
P5	P6	3

Рис. 4.6. Таблица PART_STRUCTURE (значения приведены в качестве примера)

СПИСОК ЛИТЕРАТУРЫ

В [3.3] (приложение H) приведены результаты подробного сравнения спецификации SQL: 1999 и предложений Третьего Манифеста. См. также приложение Б настоящей книги.

4.1. Astrahan M.M., Lorie R.A. SEQUEL-XRM: A Relational System // Proc. ACM Pacific Regional Conference. — San Francisco, Calif., April 1975.

Описан первый прототип реализации языка SEQUEL— самой ранней версии языка SQL [4.9]. См. также документы [4.2] и [4.3], которые выполняют аналогичную функцию для проекта System R.

4.2. Astrahan M.M. et al. System R: Relational Approach to Database Management // ACM TODS. - June 1976. - 1, № 2.

Система System R была реализацией основного прототипа (ранней версии — языка SEQUEL/2, см. [4.10]) языка SQL. В статье описывается архитектура System R в том виде, в каком она была изначально запланирована; см. также [4.3].

4.3. Blasgen M.W. et al. System R: An Architectural Overview // IBM Sys. J. — February 1981. - 20, №1.

Описывается архитектура System **R** на тот **момент,** когда система была полностью реализована (ср. с [4.2]).

4.4. Celko J. SQL for Smarties: Advanced SQL Programming. San Francisco, Calif.: Morgan Kaufmann, 1995.

"Это первая вышедшая из печати книга по языку SQL с глубоким и детальным освещением материала, в которой исчерпывающе представлены средства и методы, позволяющие совершенствовать свои навыки читателю, от неопытного пользователя языка SQL до высококвалифицированного программиста" (цитата с обложки книги).

4.5. Chaudhuri S., Weikum G. Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System // Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt. — September 2000.

Эта статья содержит некоторые серьезные критические замечания в адрес SQL. Приведем одну цитату: "SQL — это источник неприятностей. Одной из основных трудностей, связанных с использованием систем с базами данных, является именно язык SQL. Он представляет собой объединение всех тех средств, которые только можно себе представить (многие из которых редко используются или, во всяком случае, не должны быть рекомендованы для использования) и слишком сложен для разработчика приложений среднего уровня. Ядро этого языка, к которому можно отнести запросы с операторами выборки, проекции и соединения, а также операции агрегирования, является чрезвычайно полезным, но мы сомневаемся, что все его дополнительные возможности оправданы и достойны широкого применения. Попытка понять семантику спецификации SQL: 1992, не говоря уже о SQL: 1999, изучить все комбинации вложенных (и коррелированных) подзапросов, неопределенных значений, триггеров, функций поддержки абстрактных типов данных (Abstract Data Type — ADT) и т.д. становится буквально пыткой. Обучение языку SQL обычно ограничивается ядром этого языка, после чего преподаватели рекомендуют слушателям в качестве домашнего задания освоить необходимые дополнительные средства языка на собственном практическом опыте. В некоторых отраслевых журналах иногда публикуются задачи по языку SQL, в которых требуется выразить сложный информационный запрос в виде одного оператора SQL. Такие операторы занимают несколько страниц и едва ли постижимы".

4.6. Eisenberg A., Melton J. SQL: 1999, Formerly Known as SQL3 // ACM SIGMOD Record. — March 1999. — 28, № 4.

Краткое вводное описание новых средств, которые были добавлены в стандарт SQL в результате публикации спецификации SQL: 1999.

4.7. Eisenberg A. and Melton J. SQLJ Part 0, Now Known as SQL/OLB (Object Language Bindings) //ACM SIGMOD. — December 1998. — 27, № 4; Eisenberg A. and Melton J. SQLJ— Part 1: SQL Routines Using the Java™ Programming Language // ACM

SIGMOD. — December 1999. — 28, № 4. См. также Gray Clossman et al. Java and Relational Databases: SQLJ // Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data, Seattle, Wash. — June 1998.

Название SQLJ первоначально было присвоено проекту, в котором рассматривались возможности интеграции SQL и Java (в этом проекте совместно участвовали некоторых из самых известных поставщиков программного обеспечения SQL). В части 0 этого проекта изучались способы внедрения операторов SQL в программы на языке Java, в части 1 реализовывалась идея вызова компонентов Java из среды SQL (например, вызова хранимых процедур, написанных на языке Java, — см. главу 21); а часть 2 была посвящена анализу возможностей использования классов Java в качестве типов данных SQL (например, как основы для определения столбцов в таблицах SQL). Часть 0 была включена в спецификацию SQL: 1999, а части 1 и 2 будут почти наверняка включены в окончательную спецификацию SQL:2003 (см. аннотацию к [4.23]).

4.8. Chamberlin D. Using the New DB2. — San Francisco, Calif.: Morgan Kaufmann, 1996.

Увлекательное и всестороннее описание современного положения дел в отношении коммерческих продуктов SQL, сделанное одним из основных разработчиков первоначальной версии языка SQL [4.9]—[4.11].

Примечание. В этой книге Чемберлена обсуждаются также "некоторые спорные решения", воплощенные в проекте языка SQL (прежде всего, касающиеся поддержки неопределенных значений и допустимости появления дубликатов строк). Как писал сам автор этой книги, Дональд Чемберлен: "Я стремился ... в основном излагать исторические сведения, а не доказывать, что принятые решения были правильными; я считаю, что каждый должен решить, следует ли поддерживать неопределенные значения и дубликаты, на основании собственных убеждений...;. Разработчики [языка SQL] главным образом были практиками, а не теоретиками, и такая их направленность отразилась на многих решениях [проекта]". Подобная позиция очень отличается от той, которую представляет автор настоящей книги! Вопросы поддержки неопределенных значений и дубликатов должны решаться на основании *научных исследований, а* не собственных убеждений; они обсуждаются в настоящей книге с научной точки зрения в главах 19 и 6, соответственно. А что касается противопоставления "практиков теоретикам", то мы категорически отвергаем мнение о непрактичности теории. Мы уже заявили о своей позиции по этому вопросу (в разделе 4.8), отметив, что теория, по крайней мере, реляционная, по своей сути очень даже практична.

4.9. Chamberlin D.D. and Boyce R.F. SEQUEL: A Structured English Query Language // Proc. ACM SIGMOD Workshop on Data Description, Access, and Control. — Ann Arbor, Mich. — May 1974.

В статье впервые представлен язык SQL (или SEQUEL, как он назывался вначале; впоследствии название по правовым соображениям было изменено).

4.10. Chamberlin D.D. et al. SEQUEL/2: A Unified Approach to Data Definition, Manipulation, and Control // IBM J. R&D. — November 1976. — 20, № 6; см. также исправления и дополнения к этой статье. // IBM J. R&D. — January 1977. — 21, № 1.

Опыт реализации предыдущего прототипа языка SEQUEL, описанного в [4.1], и результаты проверок практической применимости привели к разработке новой версии языка, названной SEQUEL/2. Язык, поддерживаемый системой System R [4.2], [4.3], был в основном похож на SEQUEL/2 (с заметным отсутствием возможностей так называемых *утверждений* и *триггеров;* подробности приводятся в главе 9), плюс некоторые расширения, появившиеся в результате учета опыта пользователей [4.11].

4.11. Chamberlin D.D. A Summary of User Experience with the SQL Data Sublanguage // Proc. Int. Conf. on Database. — Aberdeen, Scotland, July 1980. (См. также IBM Research Report RJ2767. — April 1980.)

В статье обсуждается ранний опыт использования системы System R и предлагаются некоторые дополнения к языку SQL на основании этого опыта. Некоторые из этих дополнений — операторы EXISTS, LIKE, PREPARE и EXECUTE — действительно были реализованы в окончательной версии System R. Они описаны в разделе 8.6 (EXISTS), приложении Б (LIKE) и разделе 4.7 (PREPARE и EXECUTE).

4.12. Chamberlin D.D. et al. Support for Repetitive Transactions and Ad Hoc Queries in System R//ACM TODS. - March 1981. - 6, № 1.

В статье приводятся некоторые результаты оценки производительности системы System R как в среде выполнения произвольных запросов, так и в среде выполнения стандартных транзакций. *{Стандартная транзакция* — это простое приложение, которое имеет доступ лишь к небольшой части базы данных и перед выполнением предварительно компилируется. Это соответствует тому, что мы называли *планируемым запросом* в разделе 2.8 главы 2.) Эта статья, кроме всего прочего, демонстрирует, что в системе, подобной System R, компиляция почти всегда превосходит интерпретацию по итоговой производительности, даже при выполнении произвольных запросов, и что система приобретает способность выполнять несколько стандартных транзакций в секунду, если в базе данных предусмотрены соответствующие индексы. Эта статья достойна внимания, поскольку она была одной из первых статей, показавших несостоятельность заявлений (которые в то время приходилось слышать очень часто), что "реляционные системы никогда не будут иметь хорошие эксплуатационные характеристики". Безусловно, со времени этой первой публикации коммерческие реляционные продукты достигли такой производительности, что за секунду могут выполняться сотни и даже тысячи транзакций.

4.13. Chamberlin D.D. et al. A History and Evaluation of System R // CACM. — October 1981. -24, №10.

Описываются три основные фазы развития проекта System R (предварительный прототип, многопользовательский прототип и оценочный вариант); основное внимание уделяется технологиям компиляции и оптимизации, которые впервые использовались в System R. Частично в этой статье обсуждаются те же вопросы, что и в [4.14].

4.14. Chamberlin D.D., Gilbert A.M., Yost R.A. A History of System R and SQL/Data System // Proc. 7th Int. Conf. on Very Large Data Bases. — Cannes, France. — September 1981.

Обсуждаются уроки, полученные в результате использования прототипа системы System R, а также описывается эволюция этого прототипа до первого семейства реляционных продуктов DB2 компании IBM, а именно — SQL/DS (переименованного впоследствии в "DB2 for VM and VSE").

4.15. Date C.J. A Critique of the SQL Database Language // ACM SIGMOD Record. — November 1984. — 14, № 3. (Переиздано: C.J. Date. Relational Database: Selected Writings. — Reading, Mass.: Addison-Wesley, 1986.)

Как уже подчеркивалось в этой главе, язык SQL далек от совершенства. В статье представлен критический анализ его принципиальных недостатков (в основном исходя из требований к формальному компьютерному языку вообще, а не из требований к языку баз данных).

Примечание. Некоторые критические замечания из этой статьи не относятся к стандарту SQL: 1999.

4.16. Date C.J. What's Wrong with SQL? // Date C.J. Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

Обсуждаются некоторые недостатки языка SQL в дополнение к описанным в [4.15] под заголовками "Недостатки собственно языка SQL", "Недостатки стандарта SQL" и "Переносимость приложений".

Примечание. Некоторые критические замечания из этой статьи не относятся к стандарту SQL: 1999.

4.17. Date C.J. SQL Dos and Don'ts // Date C.J. Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

В статье предложены некоторые практические советы по использованию языка SQL, что позволяет избежать потенциальных ловушек, вызванных недостатками этого языка, которые описаны в [4.15], [4.16], [4.16], и получить максимальные преимущества по производительности, переносимости, обеспечению связи и т.п.

4.18. Date C.J. How We Missed the Relational Boat // Date C.J. Relational Database Writings 1991-1994. — Reading, Mass.: Addison-Wesley, 1995.

Краткое заключение о недостатках языка SQL, имеющих отношение к поддержке (или отсутствию таковой) некоторых элементов реляционной модели: структурных элементов, средств обработки и обеспечения целостности.

4.19. Date C.J. Grievous Bodily Harm (в двух частях) // DBP&D. — May 1998. — 11, № 5 и DBP&D. — June 1998. — 11, № 6. См. также Fifty Ways to Query // Web-узел DBP&D www. dbpd. com. — July 1998.

Язык SQL чрезмерно избыточен в том смысле, что если не все, то большинство простых запросов могут быть выражены различными способами. В статьях это показано на примерах; в них также обсуждаются возможные следствия избыточности языка SQL. В частности показано, что конструкции GROUP BY, HAVING и переменные области значений можно с большой пользой исключить из языка, не потеряв при этом никаких функциональных возможностей (то же самое справедливо и в отношении конструкции "IN *<subquery>*").

Примечание. Все перечисленные выше конструкции поясняются в главе 8, раздел 8.6.

4.20. Date C.J., Darwen H. A Guide to the SQL Standard (4th edition). — Reading, Mass.: Addison-Wesley, 1997.

Полное руководство по стандарту SQL (версии, выпущенной в 1992 году), включая описание средств SQL/CLI (по состоянию на 1995 год), SQL/PSM (по состоянию на 1996 год) и предварительный анализ спецификации SQL: 1999. В частности, в приложении D к этой книге приведены сведения о "многих аспектах стандарта, которые на сегодняшний день определены неадекватно или даже неверно". Большинство недостатков, указанных в этом приложении, обнаруживаются и в версии SQL: 1999.

4.21. Date C.J. and Colin J.W. A Guide to DB2 (4th edition). — Reading, Mass.: Addison-Wesley, 1993.

В книге дается обширный и детальный обзор СУБД DB2 компании IBM (по состоянию на 1993 год) и некоторых ее сопутствующих продуктов. СУБД DB2 также была основана на системе System R, хотя и не в такой степени, как SQL/DS [4.14].

4.22. Fishman N. SQL du Jour. // DBP&D. - October 1997. - 10, № 10.

Оставляющий грустное впечатление обзор некоторых несовместимостей, обнаруженных в продуктах SQL, которые были объявлены как "поддерживающие стандарт SQL".

4.23. International Organization for Standardization (ISO): Information Technology — Database Languages— SQL, Document ISO/IEC 9075:1999. Примечание: см. также [22.21].

Оригинальное определение стандарта ISO/ANSI SQL: 1999 (специалисты называют его также ISO/IEC9075 или просто ISO 9075). Первоначальный документ, состоящий из одной части, был со временем расширен до неограниченной серии, состоящей из отдельных частей (ISO/IEC 9075-1, 9075-2 и т.д.). Ко времени выпуска данной книги были определены следующие части.

- Часть 1. Структура (SQL/Framework).
- Часть 2. Основы (SQL/Foundation).
- Часть 3. Интерфейс на уровне вызова (SQL/CLI).
- Часть 4. Постоянные хранимые модули (SQL/PSM).
- Часть 5. Связь с базовым языком (SQL/Bindings).
- Часть 6. Отсутствует.
- Часть 7. Отсутствует.
- Часть 8. Отсутствует.
- Часть 9. Управление внешними данными (SQL/MED).
- Часть 10. Связь с объектным языком (SQL/OLB).

Как указано выше в этой главе, следующая версия стандарта должна была быть официально утверждена в 2003 году и предполагалось, что в состав этого документа войдут описанные ниже изменения.

- Материал из части 5 будет перенесен в часть 2, а часть 5 будет удалена.
- Материал из части 2, содержащий определение стандартного каталога базы данных ("информационной схемы"), будет перемещен в новую часть 11, "Схема: SQL".
- Появится новая часть 13, "Подпрограммы и типы Java (SQL/JRT)", которая будет использоваться для стандартизации результатов дальнейшей интеграции Java и SQL (см. аннотацию к [4.7]).
- В новой части 14, "Спецификации, касающиеся XML (SQL/XML)", будут стандартизированы средства, обеспечивающие взаимодействие SQL и XML (см. главу 27).

Примечание. Стоит упомянуть, что язык SQL часто называют международным стандартом "реляционных" баз данных, но в документе стандарта это не утверждается; в действительности в документе вообще не используется термин *relation* (отношение)! (Тем не менее, как указано в одной из приведенных выше сносок, в нем не упоминается и термин *база данных.)*

4.24. International Organization for Standardization (ISO) : (ISO Working Draft) — Database Language SQL— Technical Corrigendum 5, Document ISO/IEC JTC1/SC32/WG3 (December 2, 2001).

Содержит большое количество исправлений и дополнений к спецификации [4.22].

4.25. Raymond A.L. and Daudenarde J.J. SQL and its Applications. — Englewood Cliffs, N.J.: Prentice-Hall, 1991.

Книга о языке SQL, содержащая практические советы и инструкции (почти половина книги посвящена подробному обсуждению ряда примеров применения практических приложений).

4.26. Raymond A.L. and Nilsson J.F. An Access Specification Language for Relational Data Base System // IBM J.R&D. - May 1979. - 23, № 3.

В публикации подробно рассматриваются вопросы, связанные с техникой компиляции в системе System R [4.12], [4.27]. Для любого данного оператора SQL оптимизатор системы генерирует программу на внутреннем языке ASL (Access Specification Language — язык спецификации доступа). Этот язык используется как интерфейс между оптимизатором и *генератором объектного кода.* (Генератор объектного кода, как следует из его названия, преобразует программу на языке ASL в машинный код.) Язык ASL состоит из операторов типа scan (просмотр) и insert (вставка), применяемых к таким объектам, как индексы и хранимые файлы. Язык создавался, чтобы весь процесс трансляции можно было сделать более управляемым. Это достигается путем разбиения процесса трансляции на множество четко определенных подпроцессов.

4.27. Raymond A.L. and Bratford W.W. Compilation of High-Level Data Language. IBM Research Report RJ2598. — August 1979.

В системе System R впервые была применена идея компиляции запросов перед их выполнением с последующей автоматической рекомпиляцией запросов, если физическая структура базы данных значительно изменилась на каком-то промежуточном этапе. В этой статье подробно описывается механизм компиляции

и рекомпиляции, но не затрагиваются вопросы оптимизации (эта актуальная тема подробно рассматривается в [18.33]).

4.28. Melton J., Simon A.R.: SQL: 1999 — Understanding Relational Components. —San Francisco, Calif.: Morgan Kaufmann, 2002.

Руководство по стандарту SQL: 1999 (представлены только основы, а описание дополнительных возможностей отложено до выпуска книги [26.32]). Ко времени выхода [14.28] из печати один из авторов этого руководства (Melton) был редактором спецификации стандарта SQL.

4.29. Rozenshtein D., Abramovich A., Birger E. Optimizing Transact-SQL: Advanced Prog ramming Techniques. — Fremont, Calif.: SQL Forum Press, 1995.

Язык Transact-SQL — это диалект языка SQL, который поддерживается такими продуктами, как Sybase и SQL Server. В этой книге рассматривается ряд приемов программирования для языка Transact-SQL, которые основаны на использовании *характеристических функций* (определенных авторами как "средства, позволяющие программистам кодировать логические условия в виде... выражений в предложениях SELECT, WHERE, GROUP BY И SET"). Хотя эти идеи выражены в терминах языка Transact-SQL, на самом деле они имеют более широкую область применения.

Примечание. Необходимо отметить, что слово "optimizing", которое входит в название книги, не имеет отношения к оптимизатору СУБД. Напротив, здесь подразумевается оптимизация, которая может выполняться самими пользователями вручную.

ЧАСТЬ II

РЕЛЯЦИОННАЯ МОДЕЛЬ

Нет ни малейшего сомнения в том, что основой современной технологии баз данных является реляционная модель; именно благодаря наличию этой основы данная отрасль знаний становится наукой. Поэтому, по определению, любая книга с описанием основ технологии баз данных, которая не включает исчерпывающего описания реляционной модели, является неполной. Аналогичным образом, любые попытки представить себя как эксперта в области баз данных вряд ли будут оправданы, если претендент на это звание не понимает глубоко реляционную модель. Спешим добавить, что этот материал не слишком "сложен", даже напротив, но еще раз отметим, что он представляет собой основу и останется таковой в обозримом будущем.

Как было описано в главе 3, реляционная модель распространяется на три принципиальных аспекта организации данных: структура данных, манипулирование данными и целостность данных. В этой части книги последовательно рассматриваются все три аспекта.

- В главах 5 и 6 описана структура данных (в главе 5 речь идет о типах, а в главе 6 —
  об отношениях).
- Главы 7 и 8 посвящены вопросам манипулирования данными (в главе 7 рассмат
  ривается реляционная алгебра, а в главе 8 — реляционное исчисление).
- В главе 9 дано описание темы целостности данных.

Наконец, в главе 10 рассматривается такая важная тема, как представления.

Примечание. Следует добавить, что это деление реляционной модели на три части, хотя и очень удобное на высоком концептуальном уровне, становится менее четким с того момента, когда начинается анализ более конкретных вопросов. В действительности, как вскоре станет очевидно, отдельные компоненты этой модели тесно взаимосвязаны и во многом зависят друг от друга, поэтому в целом невозможно (даже в принципе) удалить какой-то отдельный компонент, не разрушив всю модель. Одним из следствий этого факта является то, что главы 5—10 включают многочисленные перекрестные ссылки друг на друга.

Важно также понять, что реляционная модель не остается неизменной — с годами она постоянно развивалась и расширялась и эта тенденция сохраняется[1]. Текст, приведенный в этих главах, отражает современные взгляды самого автора и других исследователей, работающих в этой области (как указано в предисловии, особое влияние на эти взгляды оказали идеи *Третьего Манифеста* [3.3]). Рассматриваемый подход представляется весьма обоснованным, даже окончательно сложившимся (ко времени выхода данной книги из печати), хотя, безусловно, в этой части книги он представлен не как описание текущих исследований, а как учебный материал, но читатель не должен думать, что эта область знаний не имеет дальнейших перспектив развития.

Еще раз отметим, что реляционную модель понять не так уж трудно, но это — теория, а большинство теорий обладает своей собственной специальной терминологией, и реляционная модель (по причинам, уже описанным в разделе 3.3) не является исключением из этого правила. Поэтому вполне естественно, что в данной части книги будет использоваться специальная терминология. Тем не менее, трудно оспаривать тот факт, что эта терминология на первых порах может оказаться немного обескураживающей и действительно способна стать барьером для понимания. (Этот факт особенно достоин сожаления, поскольку идеи, лежащие в основе реляционной модели, действительно не так уж сложны для понимания.) Таким образом, если читатель будет испытывать затруднения в понимании изложенного ниже материала, просим его сохранять терпение; ему вскоре станет очевидно, что изложенные здесь концепции вполне понятны, как только он ознакомится с терминологией.

Следует также добавить, что главы в этой части являются весьма объемными (на их основе вполне можно выпустить отдельную книгу). Но большой объем этих глав отражает лишь важность рассматриваемой темы! Было бы вполне возможно представить краткий обзор этой темы всего лишь на одной или двух страницах. И действительно, одно из важных преимуществ реляционной модели состоит в том, что ее основные идеи можно объяснить и понять очень легко. Тем не менее, краткое изложение не соответствует важности рассматриваемого предмета и не позволяет показать широкую область применения реляционной модели. Итак, значительный объем этой части книги следует рассматривать не как свидетельство значительной сложности модели, а как признак ее важности и ее успешного применения в качестве основы для многочисленных перспективных разработок. Усилия, затраченные читателем на полное понимание этого материала, многократно окупятся благодаря его дальнейшей успешной деятельности в области баз данных.

Наконец, кратко коснемся языка SQL. В части I этой книги уже было сказано, что SQL является стандартным языком *реляционных* баз данных и его поддерживает почти любой продукт категории баз данных, имеющийся на рынке (точнее, поддерживает некоторый диалект этого языка — см. [4.22]). Вследствие этого ни одна книга по современным базам данных не будет полной без исчерпывающего описания SQL. Поэтому в приведенных ниже главах, посвященных различным аспектам реляционной модели, рассматриваются также соответствующие средства SQL, в той степени, в какой они относятся к данной теме (при этом описание основано на материале главы 4, в которой приведены основные концепции языка SQL).

---

[1] В этом отношении она напоминает математику (область математических знаний также не является застывшей, а развивается со временем); в действительности, сама реляционная модель может рассматриваться как малая ветвь математики.

ГЛАВА 4

Типы

5.1.ВВЕДЕНИЕ

Примечание. При первом прочтении книги читатель может пожелать ознакомиться с этой главой лишь кратко. Сама эта глава по праву занимает место, отведенное ей в данной части, но значительный объем представленного в ней материала фактически не потребуется до главы 20 части V и глав 25-27 части VI.

Понятие *типа данных* (или сокращенно *типа)* является фундаментальным; каждое значение, каждая переменная, каждый параметр, каждый оператор, предназначенный только для чтения, и особенно каждый реляционный атрибут относится к тому или иному типу. Так что же такое тип? Кроме всего прочего, он представляет собой **множество значений.** К примерам таких типов относятся INTEGER (множество всех целых чисел), CHAR (множество всех символьных строк), s# (множество всех номеров поставщиков) и т.д. Поэтому, например, говоря о том, что переменная отношения s с данными о поставщиках имеет атрибут STATUS типа INTEGER, МЫ ПОД ЭТИМ подразумеваем, что значениями этого атрибута являются целые числа и ничего кроме целых чисел.

Примечание. Из этого непосредственно следуют два изложенных ниже вывода.

- Первым является то, что типы можно также называть *доменами,* особенно когда речь идет о реляционной модели; фактически автор сам использовал последний термин в предыдущих изданиях настоящей книги, но теперь предпочитает термин *типы.*
- Второй вывод может служить предостережением: автор пытается сохранить в этой части книги разумную точность. Поэтому он не утверждает, например, что тип INTEGER представляет собой множество всех возможных целых чисел, а должен фактически сделать оговорку, что это— множество всех целых чисел, которые могут быть представлены в рассматриваемой компьютерной системе (поскольку, безусловно, есть такие целые числа, которые превышают возможности представления в любой компьютерной системе). Вполне очевидно, что аналогичное уточнение относится ко многим последующим утверждениям и примерам, приведенным в этой главе; в связи с этим автор не будет явно подчеркивать указанную особенность типов, но просит читателя всегда учитывать данное предостережение.

Любой отдельно взятый тип является либо **определяемым системой** (т.е. встроенным), либо **определяемым пользователем. В** данной главе предполагается, что из трех типов, упомянутых выше, INTEGER И CHAR определены системой, а s# определен пользователем. Но основой для объявления реляционных атрибутов (а также переменных, параметров и операторов, предназначенных только для чтения, — см. раздел 5.2) может стать любой тип, независимо от того, определен ли он системой или пользователем.

С любым конкретным типом связано множество **операторов,** которые могут с полным правом применяться к значениям рассматриваемого типа; это означает, что операции со значениями указанного типа могут выполняться исключительно с помощью операторов, определенных для этого типа (выражение "определен для этого типа" означает именно то, что рассматриваемый оператор имеет параметр, который объявлен как относящийся к этому типу). Например, в случае определяемого системой типа INTEGER используются описанные ниже операторы.

- В системе предусмотрены операторы для сравнения целых чисел,"=", "<" и т.д.
- В ней также предусмотрены операторы для выполнения арифметических опера ций с целыми числами,"+", "*" и т.д.
- Но в системе не предусмотрены операторы для выполнения над целыми числами строковых операций " | | " (конкатенация), SUBSTR (выделение подстроки) и т.д. (иными словами, строковые операции над целыми числами не поддерживаются).

В отличие от этого, в случае использования типа, определяемого пользователем, такого как s#, может потребоваться определить операторы для сравнения номеров поставщиков — "=", "<" и т.д. Но такие операторы, как "+", "*" и т.д., скорее всего, не будут определены, а это означает, что арифметические операции над номерами поставщиков не должны поддерживаться (и действительно, для чего может вообще потребоваться складывать или умножать номера двух поставщиков?).

Теперь перейдем к более подробному изучению изложенных выше идей, используя в качестве основы теорию типов, которая представлена в [3.3].

5.2. ОПРЕДЕЛЕНИЕ ЗНАЧЕНИЙ И ПЕРЕМЕННЫХ

Прежде всего, необходимо подчеркнуть, что между значениями и переменными есть важное и фундаментальное логическое различие[1] (достойно удивления то, насколько часто в литературе встречаются ошибочные взгляды по этому вопросу). Автор следует трактовке, изложенной в [5.1], и принимает за основу приведенные ниже определения.

- **Значение** представляет собой "отдельно взятую константу", например, конкрет ную константу, которая выражается в виде целого числа 3. Для значения не опре делено место во времени или в пространстве. Но значения могут быть представле ны в памяти с помощью некоторого метода кодирования и такие *представления* или *проявления {appearances;* автор предпочитает именно этот термин) характери зуются определенным местоположением во времени и пространстве. Безусловно, различные проявления одного и того же значения могут существовать в различных позициях во времени и пространстве, количество которых является неограничен ным. Неформально говоря, из этого следует, что одно и то же значение могут иметь многие разные переменные, в одно и то же или в разное время. В частности, отметим, что по определению **значение не** может быть обновлено, поскольку если бы такое было возможно, то после подобного обновления это было бы уже другое значение.
- **Переменная** представляет собой некоторую позицию, в которой размещается кон кретное проявление определенного значения. Переменная, в отличие от значения, имеет свое положение во времени и пространстве. Кроме того, безусловно, пере менные, в отличие от значений, могут быть обновлены; это означает, что текущее значение рассматриваемой переменной может быть заменено другим значением, которое может оставаться тем же или отличаться от предыдущего. (Разумеется, и после обновления рассматриваемая переменная остается той же самой.)

Обратите особое внимание на то, что допустимыми значениями являются не только такие простые проявления, как целое число 3. Напротив, значения могут быть сколь угодно сложными; например, значение может представлять собой геометрическую точку, многоугольник, рентгеновский луч, документ XML, отпечаток пальца, массив, стек, список, отношение (и т.д. и т.п.). Аналогичное замечание, безусловно, относится также и к переменным.

Кроме того, отметим, что важно различать между собой значение как таковое, с одной стороны, и проявление этого значения в некотором определенном контексте (в частности, в качестве текущего значения некоторой переменной), с другой стороны. Как уже было описано, одно и то же значение может проявляться одновременно во многих разных контекстах. Каждое из таких проявлений, по сути, состоит из некоторого физического (преобразованного в определенный код) представления рассматриваемого значения; более того, все эти кодированные представления не обязательно должны быть одинаковыми. Например, целочисленное значение 3 во множестве целых чисел встречается только один раз ("во всей Вселенной" существует одно и только одно целое число 3 как таковое), но проявление этого целого числа как своего текущего значения могут одновременно содержать сколь угодно много переменных. Кроме того, некоторые из таких

---

[1] Определение этого важного и полезного понятия приведено в [3.3].

проявлений данного конкретного числа могут быть физически представлены с помощью (допустим) десятичного кодирования, а другие — с помощью двоичного кодирования. Итак, необходимо учитывать также логическое различие между **проявлением** некоторого значения, с одной стороны, и внутренним **кодом,** или **физическим представлением** данного проявления, с другой стороны.

Не умаляя важности приведенных выше рассуждений, отметим, что обычно намного удобнее по вполне очевидным причинам заменять формулировку "преобразованное в некоторый код проявление определенного значения" просто выражением "проявление некоторого значения" или (чаще всего) просто заменять его словом "значение", если при этом не возникает опасность неправильного толкования. Следует отметить, что выражение "проявление некоторого значения" представляет собой концепцию *уровня модели,* а формулировка "проявление некоторого значения, преобразованное в определенный код" представляет собой понятие, относящееся к *уровню реализации.* Например, пользователям, безусловно, может потребоваться знать, не содержат ли две разные переменные проявления одного и того же значения (т.е. покажет ли их сравнение то, что они "являются равными"); но пользователям не нужно знать, используется ли для этих двух проявлений одно и то же физическое преобразование в определенный код.

Типизация значений и переменных

Каждое значение имеет определенный тип (эту мысль равным образом можно выразить так, что каждое значение относится к определенному типу). Иными словами, если v представляет собой некоторое значение, то v может рассматриваться как некоторая конструкция, обозначенная своего рода флажком, на котором указано: "Я — целое число", или "Я — номер поставщика", или "Я — геометрическая точка" (и т.д.). Отметим, что по определению любое отдельно взятое значение всегда имеет один и только один тип[2], который никогда не изменяется. Из этого следует, что различные типы являются несовместимыми; это означает, что они не имеют друг с другом каких-либо общих значений. Более того, справедливы приведенные ниже утверждения.

- Каждая *переменная* должна быть явно объявлена как принадлежащая к некоторому типу; это означает, что все возможные значения рассматриваемой переменной представляют собой значения рассматриваемого типа.;
- Каждый *атрибут* каждой переменной отношения (см. главу 6), должен быть явно объявлен как относящийся к некоторому типу; это означает, что каждое возможное значение рассматриваемого атрибута представляет собой значение рассматриваемого типа.
- Каждый *оператор* (см. раздел 5.5), который возвращает некоторый результат, должен быть явно объявлен как относящийся к некоторому типу; это означает, что любой допустимый результат, который может быть получен путем вызова рассматриваемого оператора, представляет собой значение рассматриваемого типа.
- Каждый *формальный параметр* каждого оператора (см. также раздел 5.5) должен быть явно объявлен как относящийся к некоторому типу; это означает, что любой допустимый фактический параметр, который может быть подставлен вместо

[2] Если не считать той возможности, что поддерживается наследование типов; такая возможность не будет рассматриваться до главы 20.

рассматриваемого формального параметра, должен иметь значение рассматриваемого типа. (Это утверждение фактически не является достаточно точным. Операторы в целом подразделяются на два отдельных класса— операторы, допускающие только чтение, и операторы обновления; операторы, предназначенные только для чтения, возвращают некоторый результат, а операторы обновления вместо этого осуществляют обновление одного или нескольких из своих фактических параметров. Для оператора обновления любой фактический параметр, подлежащий обновлению, обязан быть не значением, а переменной, относящейся к тому же типу, что и соответствующий формальный параметр.)

- Вообще говоря, каждое *выражение* на языке программирования должно быть хотя бы неявно объявлено как относящееся к некоторому типу. А именно, тип должен быть объявлен для самого внешнего из операторов, участвующих в этом выражении, где под "самым внешним оператором" подразумевается оператор, выполняемый в последнюю очередь в процессе вычисления рассматриваемого выражения. Например, объявленным типом выражения a * (Ь + с) является объявленный тип оператора "*" (умножение).

В качестве дополнительного замечания следует отметить, что изложенные выше рассуждения, касающиеся операторов и формальных параметров операторов, требуют небольшого уточнения, если рассматриваемые операторы являются **полиморфными.** Оператор называется *полиморфным,* если он определен в терминах некоторого формального параметра P, а фактические параметры, соответствующие P, могут иметь разные типы при различных вызовах. Одним из наиболее наглядных примеров может служить оператор проверки на равенство "=". Допускается проверка на равенство любых двух значений v1 и v2 (но лишь при том условии, что vl и v2 относятся к одному и тому же типу), и поэтому оператор "=" является полиморфным — он может применяться к целым числам, к символьным строкам, к номерам поставщиков и фактически к значениям любого возможного типа. Аналогичные замечания относятся и к оператору присваивания ": =" (который также определен для каждого типа) — допустимо применять операцию присваивания любого значения v любой переменной V, но также при условии, что v и V относятся к одному и тому же типу. (Безусловно, операция присваивания оканчивается неудачей, если в ней нарушаются некоторые ограничения целостности, как описано в главе 9, но она не может окончиться неудачей из-за ошибки, связанной с несоответствием типов как таковой[3].) Другие примеры полиморфных операторов можно найти в главе 20 и в других главах этой книги.

5.3. ОПРЕДЕЛЕНИЯ ТИПОВ И ФОРМАТОВ ПРЕДСТАВЛЕНИЯ

Выше в данной главе уже было кратко указано, что существует логическое различие между типом как таковым, с одной стороны, и физическим представлением значений этого типа в системе, с другой стороны. Фактически типы относятся к уровню *модели,*

---

3 Точнее, эта операция не может окончиться неудачей из-за ошибки во время прогона программы, связанной с несоответствием типов. В этой книге принято достаточно обоснованное предположение, что в системе предусмотрена проверка типов во время компиляции (или так называемая "статическая" проверка); очевидно, что ошибка во время прогона программы не может возникнуть, если проверка во время компиляции прошла успешно.

а форматы физического представления — к уровню *реализации.* Например, номера поставщиков могут быть физически представлены в виде символьных строк, но из этого не следует, что нам дано право выполнять над номерами поставщиков такие же операции, как над символьными строками; точнее, **мы** можем выполнять подобные операции, но лишь при том условии, что для данного типа определены соответствующие операторы. Кроме того, операторы, определяемые для данного конкретного типа, должны естественным образом зависеть от намеченного назначения рассматриваемого типа, а не от способа, выбранного для физического представления значений этого типа; в действительности, эти физические форматы представления должны быть **скрытыми от пользователя.** Иными словами, различие, которое мы проводим между типом и физическим представлением, является одним из важных аспектов *независимости от данных* (см. главу 1).

Кстати, следует отметить, что типы данных (особенно определяемые пользователем) в литературе иногда называют **абстрактными типами данных,** или сокращенно ADT (Abstract Data Type), чтобы подчеркнуть такую их особенность; дело в том, что мы обязаны различать типы и форматы их физического представления. Но в данной книге этот термин не используется, поскольку он наводит на мысль, что могут существовать некоторые типы, не являющиеся в этом смысле "абстрактными", а автор считает, что следует всегда учитывать различие между типом и его физическим представлением.

Определения скалярных и нескалярных типов

Любой отдельно взятый тип может быть либо *скалярным,* либо *нескалярным.* Определения этих понятий приведены ниже.

- **Нескалярным** (nonscalar) называется тип, значения которого явно определены как имеющие множество видимых пользователю, непосредственно доступных компо нентов. В частности, в этом смысле являются нескалярными типы отношения (см. главу 6), поскольку отношения имеют такие видимые пользователю компоненты, как кортежи и атрибуты. (Кроме того, в свою очередь, типы кортежа также явля ются нескалярными, поскольку кортежи имеют такие видимые пользователю компоненты, как значения атрибутов.)
- **Скалярным** (scalar) называется тип, который не является нескалярным (!).

  Примечание. Вместо термина *скалярный* иногда используются также термины **инкапсулированный** (encapsulated) и **атомарный** (atomic); термин *атомарный* особенно часто применяется в реляционных контекстах (включая предыдущие издания этой книги). Для ознакомления с термином *инкапсулированный* обратитесь к главе 25.

Значения типа т являются *скалярными* или *нескалярными* в соответствии с тем, является ли скалярным или нескалярным сам тип т; поэтому нескалярное значение имеет множество видимых пользователю компонентов, а скалярное значение не имеет такового. Аналогичные замечания относятся к переменным, атрибутам, операторам, параметрам и выражениям в целом, но с соответствующими оговорками.

Возможные форматы представления, селекторы и операторы THE_

Допустим, что т — скалярный тип. Выше было показано, что физическое представление значений типа т скрыто от пользователя. В действительности, такие представления могут быть сколь угодно сложными (в частности, они вполне могут состоять из нескольких

компонентов), но еще раз подчеркнем, что все подобные компоненты должны быть скрыты от пользователя. Однако задача состоит в том, чтобы значения типа т имели хотя бы один **возможный формат представления**[4] (объявленный в составе определения типа т) и чтобы любой такой возможный формат представления не был скрытым от пользователя; в частности, все эти форматы должны иметь видимые пользователю компоненты. Но следует учитывать, что рассматриваемые компоненты являются не компонентами типа, а компонентами данного возможного формата представления, тогда как тип как таковой все еще остается скалярным в том смысле, какой был описан выше. В качестве иллюстрации рассмотрим определяемый пользователем тип QTY (сокращение от quantity — количество), определение которого на языке Tutorial D может выглядеть примерно следующим образом.

```
TYPE QTY POSSREP { INTEGER } ;
```

В этом определении типа, по сути, сказано, что значения количества "могут быть представлены" с помощью целых чисел. Таким образом, объявленное *возможное представление* (possible representation — possrep), безусловно, имеет видимые пользователю компоненты (фактически оно имеет точно один такой компонент, типа INTEGER), НО понятие количества как таковое не имеет подобного компонента.

Ниже приведен еще один пример, позволяющий более наглядно подчеркнуть эту же мысль.

```
TYPE POINT      /* Геометрические точки в двухмерном
    пространстве */ POSSREP CARTESIAN { X RATIONAL, Y
    RATIONAL P} POSSREP OLAR { R RATIONAL, 0 RATIONAL } ;
```

Приведенное здесь определение типа POINT включает объявления двух различных возможных форматов представления, CARTESIAN И POLAR, отражающих тот факт, что точки в двухмерном пространстве действительно "могут быть представлены" с помощью декартовых (CARTESIAN) ИЛИ полярных (POLAR) координат. Каждый из этих возможных форматов представления имеет в свою очередь два компонента, причем оба они относятся к типу RATIONAL[5]. Но важно отметить, что тип POINT как таковой (еще раз подчеркнем) все еще остается скалярным — он не имеет видимых пользователю компонентов.

Примечание. Мы принимаем такое соглашение о синтаксисе, что если некоторый заданный тип т имеет возможное представление без явно определенного имени, то это возможное представление по умолчанию принимает имя т. Мы также принимаем соглашение, что если заданное возможное представление PR имеет компонент без явного имени, то этот компонент по умолчанию получает имя PR. Кроме того, применение каждого объявления POSSREP влечет за собой автоматическое определение перечисленных ниже операторов, которые в основном не требуют пояснений.

---

[4] За исключением того случая, когда тип Т является "фиктивным" (см. главу 20).

[5] В языке Tutorial D используется тип RATIONAL, более точно определенный по сравнению с при вычным типом REAL. Следует отметить, что тип RATIONAL может также служить примером встроенного типа, имеющего больше чем одно возможное объявленное представление. Например, выражения 530.00 и 5.3E2 вполне могут соответствовать одному и тому же значению RATIONAL; это означает, что оба они могут стать следствием разных, но эквивалентных вызовов двух различных селекторов RATIONAL (более подробно об этом будет сказано ниже).

- Оператор-селектор, который позволяет пользователю определять или выбирать значение рассматриваемого типа, предоставляя некоторое значение для каждого компонента возможного представления.
- Множество операторов THE_ (по одному для каждого компонента возможного представления), которые позволяют пользователю обращаться к соответствующим компонентам возможного представления значений рассматриваемого типа.

Примечание. Под утверждением, что объявление POSSREP вызывает "автоматическое определение" этих операторов, подразумевается, что тот "деятель" (возможно система, а может быть, и какой-то пользователь), который отвечает за реализацию рассматриваемого типа, несет также ответственность за реализацию самих операторов.

Ниже в качестве примера представлены некоторые вызовы операторов-селекторов и оператора THE_ для типа POINT.

```
CARTESIAN ( 5.0, 2.5 )
/* Выбирает точку с координатами x = 5.0, y = 2.5 */

CARTESIAN ( X1, Y1 )
/* Выбирает точку с координатами x = X1, y = Y1, где */
/* X1 и Y1 - переменные типа RATIONAL                */

POLAR ( 2.7, 1.0 )
/* Выбирает точку с координатами r = 2.7, 8 = 1.0 */

THE_X ( P )
/* Обозначает координату x точки в */
/* P, где P - переменная типа POINT */

THE_R ( P )
/* Обозначает координату r точки в P */

THE_Y ( exp )
/* Обозначает координату y точки, обозначенной */
/* выражением exp (которое относится к типу POINT) */
```

Обратите внимание на то, что селекторы имеют такое же имя, как соответствующее возможное представление, а операторы THE_ имеют имена в форме THE_C, где c— имя соответствующего компонента соответствующего возможного представления. Обратите также внимание на то, что *селекторы* (или, точнее, *вызовы селекторов)* представляют собой обобщение более известного понятия литерала (все литералы являются вызовами селекторов, но не все вызовы селекторов могут рассматриваться как литералы; в действительности вызов селектора является литералом, если и только если все его фактические параметры в свою очередь являются литералами).

Для того чтобы определить, как описанные выше синтаксические конструкции могут применяться на практике, предположим, что физическое представление точек фактически является множеством декартовых координат (но в общем не требуется, чтобы физическое представление было идентично любому из объявленных возможных представлений). В таком случае в системе могут быть предусмотрены некоторые в высшей степени защищенные операторы, отличающиеся тем, что за ними следует псевдокод, обозначенный курсивом, который по сути предоставляет доступ к этому физическому представлению, а

реализатор типа (type implementer) использует эти операторы для реализации необходимых селекторов CARTESIAN и POLAR. Безусловно, что такой реализатор типа является (а фактически и должен быть) исключением из того общего правила, что пользователи не обязаны учитывать физические представления. Соответствующий пример приведен ниже.

```
OPERATOR CARTESIAN ( X RATIONAL, Y RATIONAL ) RETURNS
   POINT ; BEGIN ;
      VAR P POINT ;          /* P - переменная типа POINT */
<компонент X физического представления P := X> ;
<компонент Y физического представления P := Y> ;
RETURN ( P ) ; END ; END OPERATOR ;

OPERATOR POLAR ( R RATIONAL, Q RATIONAL ) RETURNS
   POINT ; RETURN ( CARTESIAN ( R * COS ( Q ) , R *
   SIN ( Q ) ) ) ;
END OPERATOR ;
```

Отметим, что в определении POLAR используется селектор CARTESIAN, а также операторы SIN и COS (предполагается, что они являются встроенными). Иным образом, определение POLAR может быть выражено непосредственно в терминах защищенных операторов, как показано ниже.

```
OPERATOR POLAR ( R RATIONAL, Q RATIONAL ) RETURNS
   POINT ; BEGIN ;
      VAR P POINT ;
<компонент X физического представления P : = R * COS ( Q)> ;
<компонент Y физического представления P :- R * SIN ( Q )>
; RETURN ( P ) ; END ; END OPERATOR ;
```

В конструкции реализатора типа также используются эти защищенные операторы для реализации необходимых операторов THE_ следующим образом.

```
OPERATOR THE X ( P POINT      ) RETURNS RATIONAL ;
   RETURN ( <компонент X      физического представления
P> ) ;
END OPERATOR ;

OPERATOR THE Y ( P POINT      ) RETURNS RATIONAL ;
   RETURN ( <компонент Y      физического представления
P> ) ;
END OPERATOR ;

OPERATOR THE R ( P POINT      ) RETURNS RATIONAL ;
   RETURN ( SQRT ( THE X      ( P ) ** 2 + THE Y ( P )
** 2 ) ) ;
END OPERATOR ;

OPERATOR THE Q ( P POINT      ) RETURNS RATIONAL ;
   RETURN ( ARCTAN ( THE Y ( P ) / THE X ( P ) )
) ; END OPERATOR ;
```

Обратите внимание на то, что в определениях THE_R и THE_Q используются операторы THE_X и THE_Y, а также операторы SQRT и ARCTAN (при этом подразумевается , что

они являются встроенными). Иным образом, операторы THE_R и THE_0 могут быть определены непосредственно в терминах защищенных операторов (их практическое осуществление оставляем читателю в качестве упражнения).

Очевидно, что пример реализации типа POINT является очень наглядным. Но важно понять, что все эти описанные концепции применимы также к более простым типам[6], например, к типу QTY. Ниже приведены некоторые примеры вызова селекторов для этого типа.

```
QTY ( 100 )
QTY ( N )
QTY ( N1 - N2 )
```

А ниже приведены некоторые примеры вызова операторовTHE_.

```
THE QTY ( Q )
THE_QTY ( Q1 - Q2 )
```

Примечание. В этих примерах предполагается, что N, N1 и N2 — переменные типа INTEGER, что Q, Q1 и Q2 — переменные типа QTY и что "-" представляет собой полиморфный оператор, который применяется и к целым числам, и к значениям количества.

Итак, поскольку значения всегда типизированы, высказывания, подобные тем, "что количество в определенной поставке равно 100", являются крайне недопустимыми. Количество — это значение типа QTY, а не значение типа INTEGER! Поэтому применительно к рассматриваемой поставке мы обязаны применять более правильный способ обозначения количества — QTY(IOO), а не просто указывать число 100 как таковое. Но в неформальном общении мы обычно не стремимся соблюдать такие требования к точности, поэтому используем (например) 100, как удобное сокращение для QTY(100). В частности, следует отметить, что подобные сокращения применяются также в определениях баз данных поставщиков и деталей, а также поставщиков, деталей и проектов (см. рис. 3.8 и 4.5, приведенные, соответственно, на стр. 119 и 154).

Ниже приведен еще один пример определения типа.

```
TYPE LINESEG POSSREP { BEGIN POINT, END POINT } ;
```

Тип LINESEG определяет отрезки прямых. Данный пример иллюстрирует такую особенность, что любое конкретное возможное представление может быть задано в терминах определяемых пользователем типов, а не только с помощью типов, определяемых системой, как во всех предыдущих примерах (иными словами, определяемый пользователем тип является таким же полноценным, как и определяемый системой).

Наконец, следует отметить, что во всех примерах этого подраздела, касающихся возможных представлений и связанных с ними конструкций, рассматривались только скалярные типы. Но возможные представления имеют и нескалярные типы. Мы вернемся к этому вопросу в разделе 5.6.

---

[6] Это утверждение является справедливым также и по отношению, в частности, ко встроенным типам, несмотря на то, что при определении соответствующих селекторов и операторов THE_ иногда могли применяться немного иные синтаксические и другие правила по сравнению с описанными в этом разделе (отчасти это обусловлено исторически сложившимися причинами). Дополнительные сведения приведены в [3.3].

5.4. ОПРЕДЕЛЕНИЕ ТИПА

В языке Tutorial D новые типы могут быть введены либо с помощью оператора TYPE, применение которого уже было показано в некоторых примерах предыдущего раздела, либо с помощью определенного генератора типа. Отложим обсуждение генераторов типа и связанного с этим вопроса о том, как следует определять нескалярные типы, до раздела 5.6, а в этом разделе рассмотрим более подробно оператор TYPE. Ниже в качестве примера приведено определение скалярного типа WEIGHT.

```
TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                      CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

Пояснение. Предположим, что по условиям задачи вес может быть представлен с помощью десятичных чисел с точностью пять цифр и с одной цифрой после десятичной точки, где рассматриваемое десятичное число больше нуля и меньше 5000.

Примечание. Приведенное выше предложение в целом представляет собой **ограничение типа** для типа WEIGHT. В общем *ограничение типа* для типа т представляет собой не что иное, как определение множества значений, из которых состоит тип т. Если какое-то объявление POSSREP не содержит явной спецификации CONSTRAINT, то по умолчанию предполагается использование спецификации CONSTRAINT TRUE (поэтому в данном примере пропуск спецификации CONSTRAINT означал бы, что допустимые значения WEIGHT являются именно такими, которые могут быть представлены с помощью десятичных чисел с точностью пять цифр и с одной цифрой после десятичной точки).

Но пример определения типа WEIGHT наводит еще на одну мысль. В раздел 3.9 главы 3 было указано, что веса деталей заданы в фунтах. Но было бы нецелесообразно увязывать само понятие этого типа с понятием **единиц,** которое, вообще говоря, является независимым от него (здесь под термином *единицы* подразумеваются *единицы измерения).* В действительности, согласно рекомендациям, приведенным в [3.3], мы должны предоставить пользователям возможность трактовать веса как измеряемые либо в фунтах, либо (допустим) в граммах, предусматривая отдельные возможные представления для каждого из них, как показано ниже.

```
TYPE WEIGHT
     POSSREP LBS { L DECIMAL (5,1)
                   CONSTRAINT L > 0.0 AND L < 5000.0
     } POSSREP GMS { G DECIMAL (7,1)
                   CONSTRAINT G > 0.0 AND G < 2270000.0
                  AND MOD ( G, 45.4 )= 0.0 };
```

Следует отметить, что оба объявления POSSREP включают спецификацию CONSTRAINT и эти две спецификации являются логически эквивалентными (MOD — это оператор, который принимает два числовых операнда и возвращает остаток от деления первого операнда на второй; в целях упрощения предполагается, что 1 фунт = 454 грамма). Из этого определения следуют приведенные ниже выводы.

- Если W— выражение типа WEIGHT, то оператор THE_L(W) возвратит значение DECIMAL(5,1), представляющее соответствующий вес в фунтах, а оператор THE_G(W) возвратит значение DECIMAL (7,1), представляющее тот же вес в граммах.

- Если N— выражение типа DECIMAL (5,1), то оба выражения, LBS(N) и GMS (454*N), возвращают одно и тоже значение WEIGHT.

Ниже приведены описания синтаксиса Tutorial D, применяемого для определения *скалярного типа.*

```
<type def>          ::= TYPE <type name>
<possrep def list>  ;

<possrep def>
      ::= POSSREP [ <possrep name> ]
                { <possrep component def commalist>
                          [ <possrep constraint def> ] }

<possrep component def>
      ::= [ <possrep component name> ] <type name>

<possrep constraint def>
      : : = CONSTRAINT <bool
exp>
```

На основании изучения этого синтаксиса можно сделать приведенные ниже выводы (большинство из них иллюстрируется двумя примерами для типа WEIGHT, показанными выше).

1. В этом синтаксисе используются и списки *(list),* и списки, разделенные запяты ми *(commalist).* Термин *разделенный запятыми список* был определен в главе 4 (раздел 4.6); термин *список* определяется аналогично, следующим образом. Пред положим, что <*xyz*> обозначает произвольную синтаксическую категорию (т.е. всё, что может находиться в левой части некоторого порождающего правила в форме Бэкуса-Наура). В таком случае выражение <*xyz list*> обозначает после довательность от нуля или больше категорий <*xyz*>, в которой каждая пара смеж ных категорий <*xyz*> разделена одним или несколькими пробелами.
2. Список <*possr ep def I is t*> должен содержать по меньшей мере одно определе ние возможного представления <*possrep def*>, а разделенный запятыми список <*possrep component def commalist*> должен содержать по меньшей мере одно определение компонента возможного представления <*possrep component def*>.
3. Квадратные скобки " [" и "] " показывают, что заключенный в них материал явля ется необязательным (как обычно принято в системе обозначений в форме Бэкуса-Наура). В отличие от этого, фигурные скобки "{" и "}" обозначают сами себя; иначе говоря, они являются синтаксическими символами в определяемом языке, а не (как обычно) символами метаязыка. Вернее, фигурные скобки используются для включения разделенных запятыми списков элементов, если разделенный за пятыми список предназначен для обозначения множества определенного рода (при этом, кроме всего прочего, подразумевается, что порядок, в котором элемен ты присутствуют в разделенном запятыми списке, является несущественным, а также подразумевается, что ни один элемент не может появиться в списке больше одного раза).

4. В общем выражение <*bool exp*> (сокращение от "boolean expression" — логиче ское выражение) обозначает *истинностное значение* (TRUE или FALSE). В данном контексте выражение <*bool exp*> не должно указывать на какие-либо перемен ные, но конструкции с обозначением имен компонентов возможного представле ния <*possrep component name*> из содержащего их определения <*possrep def*> могут использоваться для обозначения соответствующих компонентов при менимого возможного представления произвольного значения рассматриваемого скалярного типа.

   Примечание. Логические выражения называются также *условными, истинностными* или *булевыми* выражениями.

5. Отметим, что в определениях типа < *type def*> абсолютно ничего не сказано о физических представлениях. Вместо этого подобные представления должны быть определены в составе "концептуального-внутреннего" отображения (см. главу 2, раздел 2.6).

6. Определение нового типа вынуждает систему внести в свой каталог запись с опи санием этого типа (сведения о каталоге приведены в главе 3, раздел 3.6). Анало гичные замечания относятся также к определениям операторов (см. раздел 5.5).

Ниже приведены определения скалярных типов, используемых в базе данных поставщиков и деталей, которые будут широко использоваться в дальнейшем (за исключением типа WEIGHT, который уже рассматривался выше). В целях упрощения спецификации CONSTRAINT ОПущеНЫ.

```
TYPE S#    POSSREP { CHAR } ;
TYPE NAME  POSSREP { CHAR } ;
TYPE P#    POSSREP { CHAR } ;
TYPE COLOR         POSSREP      {       CHAR } ;
TYPE QTY   POSSREP { INTEGER } ;
```

(Как было указано в главе 3, атрибут поставщика STATUS и атрибуты CITY поставщика и детали определены в терминах встроенных типов, а не определяемых пользователем типов, поэтому определения типов, соответствующие этим атрибутам, не показаны.)

Безусловно, должна быть также предусмотрена возможность избавиться от некоторого типа, если его применение в дальнейшем не предусмотрено, как показано ниже.

```
DROP TYPE <type name> ;
```

Здесь конструкция < type *name*> должна обозначать определяемый пользователем, а не встроенный тип. Эта операция вызывает удаление из каталога записи с описанием этого типа, а это означает, что в дальнейшем рассматриваемый тип становится неизвестным для системы. В целях упрощения предполагается, что операция DROP TYPE оканчивается неудачей, если рассматриваемый тип все еще где-то используется, в частности, если на его основе определен некоторый атрибут некоторой переменной отношения в той или иной базе данных.

В завершении данного раздела укажем, что операция определения типа фактически не создает соответствующее множество значений; в принципе, считается, что эти значения уже существуют и всегда будут существовать (например, достаточно представить себе

значения типа INTEGER). Поэтому единственное, что происходит при выполнении так называемой операции "определения типа" (например, оператора TYPE языка Tutorial D), представляет собой не что иное, как введение некоторого имени, с помощью которого в дальнейшем можно будет ссылаться на это множество значений. Аналогичным образом, оператор DROP TYPE фактически уничтожает не соответствующие значения, а просто имя, введенное с помощью соответствующего оператора TYPE.

5.5. ОПЕРАТОРЫ

До сих пор все определения операторов, применявшиеся в этой главе, относились либо к селекторам, либо к операторам THE_, а теперь рассмотрим определения операторов в целом. В качестве первого примера ниже показан определяемый пользователем оператор ABS для встроенного типа RATIONAL.

```
OPERATOR ABS ( Z RATIONAL ) RETURNS
   RATIONAL ; RETURN ( CASE
WHEN Z > 0 . 0 THEN +Z WHEN Z < 0.0
THEN -Z END CASE ) ; END OPERATOR
;
```

Оператор ABS (сокращение от "absolute value" — абсолютное значение) определен в терминах только одного параметра, z, имеющего тип RATIONAL, и возвращает результат того же типа. Поэтому любой вызов оператора ABS, например ABS (AMTl + AMT2), является по определению выражением типа RATIONAL.

Рассматриваемый в следующем примере оператор DIST (сокращение от "distance between" — расстояние между двумя точками) принимает два параметра одного и того же определяемого пользователем типа (POINT) и возвращает результат другого типа (LENGTH).

```
OPERATOR DIST ( P1 POINT, P2 POINT ) RETURNS LENGTH ;
   RETURN { WITH THE X ( P1 )  AS    X1 ,
                 THE X ( P2) AS X2 ,
                 THE Y ( P1) AS Y1 ,
                 THE Y ( P2) AS Y2 :
            LENGTH ( SQRT ((X1 - X2 ) ** 2
                        + ( Y1 - Y2 ) ** 2 ) ) )
;
END OPERATOR ;
```

Предполагается, что селектор LENGTH принимает параметр типа RATIONAL. Кроме того, заслуживает внимания то, как применяется конструкция WITH ДЛЯ присваивания имен результатам некоторых подвыражений. Эта конструкция будет широко использоваться в следующих главах.

В качестве следующего примера рассматривается один из обязательных операторов сравнения, "=" (сравнение на равенство[7]), для типа POINT.

---

[7] Приведенный здесь оператор "сравнения на равенство" было бы точнее назвать оператором сравнения на идентичность, поскольку выражение vl = v2 принимает истинное значение, если и только если значения vl и v2 по сути являются полностью одинаковыми.

```
OPERATOR EQ ( P1 POINT, P2 POINT    )      RETURNS BOOLEAN ;
   RETURN ( THE X ( P1 ) = THE X    (      P2 ) AND
            THE Y ( P1 ) = THE Y {   P2 )
) ;
END OPERATOR ;
```

Следует отметить, что здесь в выражении, которое входит в состав оператора RETURN, используется встроенный оператор "=" для типа RATIONAL. В целях упрощения с этого момента будем считать, что в качестве оператора проверки на равенство может использоваться обычное инфиксное обозначение "=" (для всех типов, т.е. не только для типа POINT); здесь не приведены рассуждения на тему, касающуюся того, как инфиксные обозначения могут быть определены на практике, поскольку по сути такое решение относится просто к сфере синтаксиса.

Ниже приведен оператор ">" для типа QTY.

```
OPERATOR GT ( Q1 QTY, Q2 QTY ) RETURNS BOOLEAN ;
   RETURN ( THE QTY ( Q1 ) > THE QTY ( Q2 )
) ; END OPERATOR ;
```

Здесь в выражении в составе оператора RETURN используется встроенный оператор ">" для типа INTEGER. И в данном случае подразумевается, что начиная с этого момента в качестве данного оператора может использоваться обычное инфиксное обозначение, причем не только для типа QTY, но и для всех так называемых *порядковых типов*. (По определению порядковым типом называется тип, к которому применим оператор ">". Простым примером *непорядкового* типа является POINT.)

Наконец, ниже приведен пример определения оператора обновления (все предыдущие примеры относились к типу операторов, предназначенных только для чтения, в которых не допускается обновлять что-либо, за исключением, возможно, локальных переменных)[8]. Вполне очевидно, что в этом определении вместо спецификации RETURNS применяется спецификация UPDATES; операторы обновления не возвращают значение и должны вызываться с помощью явно заданных операторов CALL [3.3].

```
OPERATOR REFLECT ( P POINT ) UPDATES
   P ; BEGIN ;
THE X ( P ) := - THE X ( P ) ;
THE Y ( P ) := - THE Y ( P ) ;
RETURN ; END ; END OPERATOR ;
```

Оператор REFLECT по сути перемещает точку, обозначенную своими декартовыми координатами (x,y), в противоположную позицию (-x, -y); он выполняет это действие путем соответствующего обновления переданных ему фактических параметров с координатами точки. Обратите внимание на то, что в этом примере используются **псевдопеременные THE_.** Псевдопеременная THE_ представляет собой вызов оператора THE_ в целевой позиции (в частности, слева от оператора присваивания). При таком вызове

[8] Операторы, предназначенные только для чтения, и операторы обновления часто называют также, соответственно, наблюдателями и модификаторами, особенно в объектных системах (см. главу 25). Для обозначения операторов, предназначенных только для чтения, применяется еще один синоним — функция (и этот термин время от времени используется в указанном значении и в данной книге).

фактически *обозначается* заданный компонент (соответствующего возможного представления) фактического параметра (а не просто возвращается его значение). Например, в определении оператора REFLECT следующий оператор присваивания

```
THE_X ( P ) := . . . ;
```

фактически присваивает значение компоненту X (декартового возможного представления) переменной формального параметра, соответствующей параметру P. Безусловно, любой формальный параметр, который необходимо обновить с помощью оператора обновления (в частности, путем присваивания значения псевдопеременной THE_), должен быть задан именно как переменная, а не в виде некоторого более общего выражения. Псевдопеременные могут быть вложенными, как показано ниже.

```
VAR LS LINESEG ;
THE_X ( THE_BEGIN ( LS ) ) := 6.5 ;
```

Но следует отметить, что в принципе псевдопеременные THE_ не являются строго необходимыми. Еще раз рассмотрим следующий оператор присваивания.

```
THE_X ( P ) := - THE_X ( P ) ;
```

Этот оператор присваивания, в котором используется псевдопеременная, является логически эквивалентным оператору, приведенному ниже, в котором таковая не используется.

```
. P := CARTESIAN ( - THE_X ( P ) , THE_Y ( P ) ) ;
```

Аналогичным образом, следующий оператор присваивания

```
THE_X ( THE_BEGIN ( LS ) ) := б.5 ;
```

является логически эквивалентным приведенному ниже.

```
LS := LINESEG ( CARTESIAN ( 6.5,
                            THE_Y ( THE_BEGIN ( LS ) )
                ) , THE_END ( LS ) ) ;
```

Иными словами, псевдопеременные как таковые не являются строго необходимыми для поддержки той разновидности обновления на уровне компонентов, которая рассматривается в этом разделе. Но интуитивно подход с использованием псевдопеременных кажется более привлекательным, чем показанный выше альтернативный подход (к тому же он может рассматриваться как более упрощенный вариант этого альтернативного подхода); кроме того, он также обеспечивает более высокую степень невосприимчивости к изменениям в синтаксисе соответствующего селектора. (К тому же может оказаться, что его проще реализовать наиболее эффективно.)

Коснувшись темы упрощений, следует указать, что единственным оператором обновления, который в принципе необходим, является оператор присваивания (":="); все остальные операторы обновления могут быть определены исключительно в терминах присваивания (как уже фактически было показано в главе 3, особенно на примере реляционных операторов обновления). Но нужна также поддержка для множественной формы присваивания, которая позволяет "одновременно" выполнить любое количество

отдельных присваиваний [3.3]. Например, можно заменить два оператора присваивания в определении оператора REFLECT следующим оператором *множественного присваивания.*

```
THE_X ( P ) := - THE_X (P)
THE_Y ( P ) := - THE_Y ( P)
```

(В этой конструкции заслуживает особого внимания разделитель в виде запятой.) Этот оператор имеет следующую семантику: во-первых, вычисляются все исходные выражения с правых сторон операторов присваивания; во-вторых, после этого все отдельные операторы присваивания выполняются в той последовательности, в которой они записаны[9].

Примечание. Поскольку операция множественного присваивания рассматривается как одна операция, "в процессе" подобного присваивания проверка целостности не производится; в действительности этот факт является одной из основных причин, по которым мы стремимся в первую очередь обеспечить поддержку множественного присваивания. Дополнительные сведения по этой теме приведены в главах 9 и 16.

Наконец, должна быть предусмотрена возможность избавиться от некоторого оператора, если он в дальнейшем не потребуется, например, как показано ниже.

```
DROP OPERATOR REFLECT ;
```

Данный конкретный оператор должен быть определяемым пользователем, а не встро-. енным.

Преобразования типов

Еще раз рассмотрим следующее определение типа.

```
TYPE S# POSSREP { CHAR } ;
```

Здесь по умолчанию возможное представление имеет унаследованное имя S# и поэтому такое же имя имеет и соответствующий оператор-селектор. Таким образом, показанный ниже вызов селектора является допустимым.

```
S# ('S I')
```

(Этот вызов возвращает определенный номер поставщика.) Поэтому можно сделать вывод, что данный селектор s# может неформально рассматриваться как оператор **преобразования типа,** который преобразует символьные строки в номера поставщиков. Аналогичным образом, селектор P# может рассматриваться как оператор преобразования, который преобразует символьные строки в номера деталей, а селектор QTY — как оператор преобразования, преобразующий целые числа в значения количества, и т.д.

На основании таких же рассуждений операторы THE_ могут рассматриваться как операторы, осуществляющие преобразование типа в противоположном направлении. Например, еще раз вернемся к определению типа WEIGHT, приведенному в начале раздела 5.4.

---

[9] Это определение требует некоторого уточнения в том случае, если два или больше отдельных операторов присваивания относятся к одной и той же целевой переменной. Подробные сведения о такой ситуации выходят за рамки данной книги; достаточно отметить, что подобные операции должны быть тщательно определены, с тем чтобы достигался требуемый результат в тех случаях, когда (как в рассматриваемой примере) фактически различные отдельные операторы присваивания обновляют отдельные части одной и той же целевой переменной (этот конкретный случай является особенно важным).

```
TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                      CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

Если w относится к типу WEIGHT, то выражение

```
THE_D ( W )
```

фактически преобразует вес, обозначенный переменной w, в десятичное число DECIMAL(5,1).

Итак, в разделе 5.2 было указано, что источник и цель в операторе присваивания должны относиться к одному и тому же типу и что к одинаковому типу должны относиться операнды в операции сравнения на равенство. Но в некоторых системах соблюдение этих правил непосредственно не предписано, поэтому в таких системах можно, например, сформировать запрос на проведение сравнения между номером детали и символьной строкой (допустим, в конструкции WHERE), как показано ниже.

```
... WHERE P# = 'P2'
```

В данном случае левый операнд относится к типу P#, а правый — к типу CHAR, поэтому по указанным выше соображениям попытка провести такое сравнение должна окончиться неудачей из-за ошибки, связанной с **несоответствием типов** (фактически ошибка, связанная с несоответствием типов, должна обнаруживаться на этапе компиляции). Но концептуально здесь происходит именно то, что система определяет возможность использования "оператора преобразования" P# (иными словами, селектора P#) для преобразования операнда операции сравнения типа CHAR в тип P#, и поэтому фактически преобразует эту операцию сравнения в представленную ниже.

```
... WHERE P# = P# ('P2')
```

После этого операция сравнения становится допустимой.

Неявный вызов оператора преобразования подобным образом называется **приведением типа** (coercion). Но хорошо известно, что непродуманное использование приведения типа может стать причиной возникновения ошибок в программе. По этой причине мы в данной книге придерживаемся консервативной позиции, согласно которой неявно заданные операции приведения типа не допускаются: операнды всегда должны относиться к соответствующим типам, а не просто быть приводимыми к этим типам. Безусловно, автор допускает возможность определения операторов преобразования типа (или операторов CAST, как они обычно называются) и явного их вызова в случае необходимости, например, как показано ниже.

```
CAST_AS_CHAR ( 53 0.00 )
```

Как уже было сказано, селекторы (по меньшей мере, те из них, которые принимают только один параметр) могут рассматриваться как своего рода операторы явного преобразования.

Итак, читатель к этому времени вполне мог прийти к заключению, что все сказанное здесь относится к подходу, известному в кругу пользователей языков программирования как **строгая типизация.** Различные авторы приводят немного разные определения для этого понятия, но в том виде, в каком оно определено в данной книге, это понятие означает, кроме всего прочего, что каждое значение имеет тип и при каждой попытке

выполнить некоторую операцию система проверяет, имеют ли операнды типы, допустимые для использования в данной операции. Например, рассмотрим следующие выражения.

```
WEIGHT + QTY   /* Суммирование веса детали с количеством
деталей в поставке */
WEIGHT * QTY   /* Умножение веса детали на количество деталей
в поставке    */
```

Первое из этих выражений не имеет смысла, и система должна отвергнуть попытку его выполнить. С другой стороны, второе выражение вполне обосновано: оно обозначает общий вес всех деталей, которые входят в данную поставку. Поэтому операторы, которые могут быть определены для весов и количеств, применяемых в сочетании, должны, по всей видимости, включать"*", но не "+".

Ниже приведено еще несколько примеров, в которых на этот раз рассматриваются операции сравнения.

```
WEIGHT >
QTY EVEN >
ODD
```

(Во втором примере предполагается, что операнд EVEN относится к типу четного целого числа EVEN_INTEGER, а операнд ODD — к типу нечетного целого числа ODD_INTEGER, причем семантика этих типов является очевидной.) Поэтому и в данном случае первое выражение не имеет смысла, а второе действительно имеет смысл. Поэтому операторы, которые могут быть определены для сочетаний весов и количеств, по-видимому, не должны включать ">", а для четных и нечетных целых чисел применение оператора ">" является вполне допустимым[10]. (Что касается проблемы определения того, какие операторы являются допустимыми для тех или иных типов, следует отметить, что по традиции в большей части литературы по базам данных, включая первые несколько изданий настоящей книги, рассматривались только операторы сравнения и игнорировались все прочие операторы, такие как"+" и "*".)

Заключительные замечания

Изучение вопроса о полной поддержке операторов, соответствующих требованиям, изложенным в настоящем разделе, приводит к нескольким важным выводам, которые кратко изложены ниже.

- Первый и наиболее важный вывод состоит в том, что система должна иметь ин формацию о том, **какие именно выражения являются допустимыми** и каким должен
  быть **тип результата** для каждого такого допустимого выражения.
- Это также означает, что общая коллекция типов для конкретной базы данных должна представлять собой замкнутое множество; из этого следует вывод, что тип результата каждого допустимого выражения должен быть типом, известным сис теме. В частности, следует отметить, что это замкнутое множество типов должно

[10] На практике типы EVEN_INTEGER и ODD_INTEGER вполне могли быть определены как подтипы типа INTEGER, и в этом случае оператор ">", вероятно, был бы унаследован от этого последнего типа (см. главу 20).

также включать *логический тип* (или *истинностное значение),* поскольку в ином случае ни один оператор сравнения не будет допустимым!

- В частности, из того факта, что системе должен быть известен тип результата каждого допустимого выражения, следует, что системе должно быть известно, какие операции **присваивания** и какие операции **сравнения** являются допустимыми.

Завершаем этот раздел ссылкой на изложенный ниже материал. Перед этим уже было указано, что понятия, называемые по традиции в сообществе пользователей реляционных систем *доменами,* в действительности являются типами данных, определяемыми системой или пользователем, имеющими произвольную внутреннюю сложность, значениями которых можно оперировать исключительно с помощью операторов, определенных для рассматриваемого типа (и физическое представление которых по этой причине должно быть скрыто от пользователя). Итак, если мы теперь обратим наше внимание на объектные системы, то обнаружим, что наиболее фундаментальная концепция объекта, класс объекта, в действительности представляет собой тип данных, определяемый системой или пользователем, имеющий произвольную внутреннюю сложность, значениями которого можно оперировать исключительно с помощью операторов, определенных для рассматриваемого типа (и физическое представление которых по этой причине должно быть скрыто от пользователя)... . Иными словами, домены и классы объектов представляют собой одно и то же! И поэтому нам остается только соединить эти две технологии (отношения и объекты). Этот важный вопрос исследуется в главе 26.

5.6. ГЕНЕРАТОРЫ-ТИПОВ

Теперь обратимся к изучению типов, которые не определены с помощью оператора TYPE, а получены путем вызова некоторого **генератора типа** (type generator). В общем, *генератор типа* — это просто особый вид оператора; его специфика состоит в том, что он возвращает тип, скажем, вместо простого скалярного значения. Например, в обычном языке программирования можно применить следующую конструкцию.

```
VAR SALES ARRAY INTEGER [12] ;
```

Эта конструкция позволяет определить переменную SALES, допустимыми значениями которой являются одномерные массивы из 12 целых чисел. В этом примере выражение ARRAY INTEGER [12] может рассматриваться как вызов генератора типа ARRAY, который возвращает массив конкретного типа. Данный конкретный тип массива является **сгенерированным типом.** Из этого следуют приведенные ниже выводы.

1. Генераторы типов известны в литературе под многими разными названиями, включая конструкторы типов, параметризованные типы, полиморфные типы, шаблоны типов и универсальные типы. В данной книге принят термин *генератор типа.*
2. Сгенерированные типы являются вполне допустимыми типами и могут приме няться везде, где допустимо использование обычных "несгенерированных" типов. Например, может быть определена некоторая переменная отношения, имеющая определенный атрибут типа ARRAY INTEGER [12]. В отличие от этого, сами ге нераторы типов как таковые не являются типами.

3. Большинство сгенерированных типов (но не все) являются именно нескалярными типами (одним из характерных примеров могут служить типы массивов). Поэтому в разделе 5.4 мы обещали показать, а здесь показали, как могут быть определены нескалярные типы.

   Примечание. Хотя и может существовать возможность определять нескалярные типы, не вызывая непосредственно некоторый генератор типа, в данной книге такая возможность больше не рассматривается.

4. Для полной ясности отметим, что мы рассматриваем сгенерированные типы именно как типы, определяемые системой, поскольку для их получения требуется вызвать генератор типа, определяемый системой.

   Примечание. Фактически здесь допущено некоторое упрощение. В частности, мы не исключаем возможности для пользователей определять свои собственные генераторы типов. Но такая возможность в данной книге больше не рассматривается.

Итак, сгенерированные типы, безусловно, имеют возможные представления (сокращенно possrep — possible representation), которые, вполне очевидно, являются производными, во-первых, от универсального возможного представления, применяемого к рассматриваемому генератору типа, и, во-вторых, от конкретного возможного представления (представлений) видимого пользователю компонента (компонентов) конкретного рассматриваемого сгенерированного типа. Например, в случае генератора типа ARRAY INTEGER [12] имеют место описанные ниже особенности.

> Должно существовать некоторое универсальное возможное представление, определенное для одномерных массивов в целом, возможно, как непрерывная последовательность элементов массива, которые могут быть определены с помощью индексов в диапазоне от нижнего до верхнего (где нижний и верхний индексы обозначают допустимые границы; в данном примере 1 и 12).
>
> Видимые пользователю компоненты представляют собой именно те 12 элементов массива, которые были только что упомянуты, и они имеют то возможное представление (представления), которое определено для типа INTEGER.

Аналогичным образом, должны быть предусмотрены операторы, предоставляющие необходимые функциональные возможности селектора и оператора THE_. Например, выражение

```
ARRAY INTEGER ( 2, 5, 9, 9, 15, 27, 33, 32, 25, 19, 5, 1 )
```

которое по сути представляет собой литерал с определением массива, может использоваться для задания конкретного значения типа ARRAY INTEGER [12] (иными словами, "для применения функциональных средств селектора"). Аналогичным образом, следующее выражение

```
SALES [3]
```

может использоваться для доступа к третьему компоненту (т.е. к третьему элементу массива) того значения массива, которое оказалось текущим значением переменной массива SALES (иными словами, "для применения функциональных средств оператора THE_"). Это выражение может также служить в качестве псевдопеременной.

Применимы также операторы присваивания и сравнения для проверки на равенство. Например, ниже приведен допустимый оператор присваивания.

```
SALES := ARRAY INTEGER ( 2, 5, 9, 9, 15, 27, 33, 32, 25,
19, 5, 1 ) ;
```

А в данном случае определен допустимый оператор проверки на равенство.

```
SALES = ARRAY INTEGER ( 2, 5, 9, 9, 15, 27, 33, 32, 25, 19, 5,
1 )
```

Примечание. Любой конкретный генератор типа может также иметь множество связанных с ним универсальных ограничений и операторов (они являются универсальными в том смысле, что рассматриваемые ограничения и операторы применяются к любому конкретному типу, полученному с помощью вызова рассматриваемого генератора типа). Например, в случае генератора типа ARRAY имеют место описанные ниже особенности.

- Может быть предусмотрено такое универсальное ограничение, что нижний индекс *lower* не должен быть больше верхнего индекса *upper*.
- Может быть предусмотрен универсальный оператор "обращения" REVERSE, который принимает в качестве входных данных произвольный одномерный массив и возвращает в качестве выходных данных другой такой же массив, содержащий элементы первого массива в обратном порядке.

(В действительности, упомянутые выше *селекторы, операторы* THE_ , а также операторы присваивания и сравнения на равенство также фактически происходят от некоторых универсальных операторов.)

В заключение отметим, что двумя генераторами типов, которые имеют особую важность в мире реляционных баз данных, являются TUPLE И RELATION. ЭТИ операторы подробно рассматриваются в следующей главе.

5.7. СРЕДСТВА SQL

Встроенные типы

В языке SQL предусмотрены перечисленные ниже встроенные типы, имена которых в основном говорят сами за себя.

- BOOLEAN
- BIT [VARYING] (n)
- BINARY LARGE OBJECT (n)
- CHARACTER [VARYING] (n)
- CHARACTER LARGE OBJECT(n)
- NUMERIC (p,q)
- DECIMAL (p,q)
- INTEGER
- SMALLINT
- FLOAT(p)
- TIME
- DATE
- TIMESTAMP
- INTERVAL

Поддерживаются многочисленные применяемые по умолчанию параметры, сокращения и альтернативные имена, например, CHAR как сокращение от CHARACTER, CLOB— OT CHARACTER LARGE OBJECT, BLOB— OT BINARY LARGE OBJECT И Т.Д.; В
данной книге эти подробности не рассматриваются. Ниже приведена наиболее важная информация по этой теме.

1. Встроенные типы данных BIT И BIT VARYING были введены в спецификации SQL: 1992 и должны быть снова изъяты из спецификации SQL:2003 (!).
2. Несмотря на их имена, содержащие слово "объект", типы CLOB и BLOB фактиче ски являются строковыми (они не имеют никакого отношения к объектам в том смысле, какой рассматривается в главе 25; в частности, тип BLOB фактически представляет собой строковый тип, состоящий из байтов или "октетов" (он не имеет ничего общего с двоичными числами, хотя и содержит в своем полном име ни слово BINARY). Кроме того, поскольку значения этих типов могут быть очень велики (поэтому такие значения иногда неформально называют просто *длинными строками),* в языке SQL предусмотрена конструкция, называемая *локатором* (locator), которая (кроме всего прочего) позволяет обеспечить доступ к отдельным фрагментам этих значений.
3. Для всех этих типов предусмотрены операторы присваивания и сравнения на ра венство. Операция сравнения на равенство по сути выполняется очень просто (но см. пункт 5). Оператор присваивания выглядит примерно следующим образом.

SET *<target> = <source> ;*

Безусловно, операции присваивания осуществляются также неявно при выполнении операций выборки и обновления базы данных. Но реляционное присваивание как таковое не поддерживается. Кроме того, не поддерживается множественное присваивание, за исключением следующего случая": если строка г обновляется с помощью оператора в приведенной ниже форме.

```
UPDATE T SET C1 = exp1, ..., Cn = expn WHERE p ;
```

Здесь все выражения expl, ..., expn вычисляются до того, как происходит любое из отдельных присваиваний переменным C1,..., Cn (a r представляет собой строку в результатах вычисления выражения т WHERE p).

4. Поддерживается строгая типизация, но только в ограниченном объеме. Точнее, к встроенным типам можно применить определенный способ классификации, согласно которому они подразделяются на 10 отдельных категорий следующим образом:

- истинностное значение
- битовая строка
- двоичное значение
- символьная строка
- числовое значение
- дата
- время
- временная отметка
- интервал год/месяц
- интервал сутки/время суток

[11] Еще два исключения кратко описаны в главе 9, раздел 9.12, подраздел "Ограничения базовой таблицы" и в главе 10, раздел 10.6, подраздел "Обновления представлений". Если не считать этих исключений, автору не известно ни об одном программном продукте, представленном на современном рынке, который поддерживал бы множественное присваивание. Но автор считает, что такая поддержка желательна и даже необходима; и действительно, она запланирована для включения в спецификацию SQL:2003, но не для отношений.

Проверка типов происходит на основе именно таких 10 категорий (это особенно касается операций присваивания и проверки на равенство). Поэтому, например, попытка сравнить число и символьную строку является недопустимой, но попытка сравнить два числа осуществима, даже если эти числа относятся к разным числовым типам, скажем INTEGER и FLOAT (в данном примере перед выполнением сравнения значение INTEGER будет приведено к типу FLOAT).

5. Правила проверки типов являются особенно сложными применительно к символьным строковым типам, таким как CHAR (n), CHAR VARYING (П) И CLOB (n). Изложение подробных сведений по этой теме выходит за рамки данной книги, но мы должны кратко рассмотреть случай символьных строк фиксированной длины (т.е. тип CHAR (п)), как описано ниже.

   - **Сравнение.** Если сравниваются значения типа CHAR (nl) и CHAR (n2), то более короткое значение обязательно дополняется справа пробелами так, чтобы его длина стала равной длине более длинного значения, и после этого происходит сравнение[12]. Поэтому, например, строки ' P2 ' (с длиной два) и ' P2 ' (с дли ной три) рассматриваются при их сравнении как равные.
   - **Присваивание.** Если значение типа CHAR(nl) присваивается переменной типа CHAR(n2), то перед выполнением операции присваивания значение CHAR(nl) дополняется справа пробелами при nl < п2 или усекается справа при nl > n2 для того, чтобы это значение могло иметь длину п2. Считается ошибкой, если при любом таком усечении теряются какие-либо непробельные символы.

Дополнительные пояснения и описания приведены в [4.20].

Типы DISTINCT

Язык SQL поддерживает две разновидности определяемых пользователем типов — типы DISTINCT и структурированные типы; оба эти типа определяются[13] с помощью оператора CREATE TYPE. Типы DISTINCT рассматриваются в этом подразделе, а структурированные типы — в следующем (ключевое слово "DISTINCT" записано здесь прописными буквами, чтобы подчеркнуть, что данное слово не используется в этом контексте в своем обычном естественном языковом смысле — как "различный"). Ниже приведено определение SQL для типа WEIGHT типа DISTINCT (сравните и сопоставьте всевозможные определения для этого типа на языке Tutorial D в разделе 5.4).

```
CREATE TYPE WEIGHT AS DECIMAL (5,1) FINAL ;
```

В своей простейшей форме (т.е. если не рассматриваются различные необязательные спецификации) это определение имеет следующий синтаксис.

---

[12] Здесь предполагается, что к одному из сравниваемых операндов применяется операция дополне ния пробелами PAD SPACE [4.20].

[13] Этот оператор поддерживает также некие конструкции, называемые в данном контексте *доменами,* но домены SQL не имеют ничего общего с доменами в реляционном смысле. Домены SQL подробно рассматриваются в [4.20].

```
CREATE TYPE <type name> AS <representation> FINAL ;
```

Ниже приведены дополнительные сведения по этой теме.

1. Обязательная спецификация FINAL рассматривается в главе 20.
2. Параметр <*representation*> представляет собой имя другого типа (и этот рас сматриваемый тип не должен быть определен пользователем или сгенерирован). В частности, следует отметить, что с учетом этих правил, касающихся применения параметра <*representation*>, мы не могли бы определить тип POINT из раздела 5.3 как тип DISTINCT языка SQL.
3. Следует также учитывать, что параметр <*representation*> задает не возможное представление, как было описано выше в этой главе, а скорее фактическое физи ческое представление рассматриваемого типа DISTINCT. В действительности, язык SQL вообще не поддерживает понятие "возможного представления possrep". Одним из следствий такого упущения является то, что невозможно определить тип DISTINCT (или структурированный тип, когда речь пойдет о нем) с помощью двух или нескольких отдельных возможных представлений.
4. В языке SQL нет никаких конструкций, аналогичных спецификации CONSTRAINT языка Tutorial D. Например, в случае типа WEIGHT не существует способа, позво ляющего указать, что для каждого значения WEIGHT соответствующее значение DECIMAL (5 , 1) должно быть больше нуля (чтобы вес не мог стать отрицатель ным!) или, скажем, меньше 5000.
5. Применимые к типу DISTINCT операторы сравнения определяются именно как такие операторы, которые могут применяться к соответствующему физическому представлению.

   Примечание. Если не считать оператора присваивания (см. пункт 8), другие операторы, применимые к физическому представлению, не могут применяться к типу DISTINCT. Например, ни одно из следующих выражений не является допустимым, даже если WT относится к типу WEIGHT.

```
WT  +
14.7 WT  *
2 WT  +
WT
```

6. Селекторы и операторы THE_ поддерживаются. Например, если NW— переменная базового языка типа DECIMAL (5 ,1), то выражение WEIGHT (: NW) возвращает соответствующее значение веса, а если WT — столбец типа WEIGHT, то выражение DECIMAL(WT) возвращает[14] соответствующее значение DECIMAL(5,1). Поэтому следующие операции SQL являются допустимыми.

[14] Фактически в синтаксисе, который определен в спецификации SQL: 1999, выражение DECIMAL (WT) не является допустимым, но предполагается, что оно станет допустимым в спецификации SQL:2003. Однако следует отметить, что (в отличие от операторов THE_ языка Tutorial D) его нельзя будет использовать в качестве псевдопеременной.

```
DELETE FROM P WHERE WEIGHT =
WEIGHT ( 14.7 ) ;

EXEC SQL DELETE FROM P WHERE WEIGHT
        = WEIGHT ( :NW ) ;

EXEC SQL DECLARE Z CURSOR FOR
         SELECT DECIMAL ( WEIGHT ) AS DWT
         FROM P
         WHERE WEIGHT > WEIGHT ( :NW ) ;
```

7. За одним важным исключением (см. пункт 8), к типам DISTINCT применяются правила строгой типизации. Необходимо обратить особое внимание на то, что операции сравнения значений типа DISTINCT и значений типа, лежащего в основе представления типа DISTINCT, являются недопустимыми. Поэтому следующие операторы SQL не считаются допустимыми, даже несмотря на то, что переменная NW (как и прежде) относится к типу DECIMAL(5,1).

```
DELETE FROM P WHERE WEIGHT = 14.7 ; /* Предостережение:
недопустимый оператор!!! */

EXEC SQL DELETE FROM P WHERE WEIGHT = :NW ; /* Предостережение:
        недопустимый оператор!!! */

EXEC SQL DECLARE Z CURSOR FOR
         SELECT DECIMAL ( WEIGHT ) AS DWT
         FROM P
         WHERE WEIGHT > :NW ; /* Предостережение: недопустимый
         оператор!!! */
```

8. Исключение, упомянутое в пункте 7, касается операций присваивания. Например, если требуется выполнить выборку некоторого значения WEIGHT в определенную переменную типа DECIMAL(5,1), то должно быть выполнено преобразование типов. Безусловно, что такое преобразование может быть выполнено явно, как показано ниже.

```
SELECT DECIMAL ( WEIGHT ) AS DWT
INTO :NW
FROM P
WHERE P# = P# ('P1') ;
```

Но следующий оператор также является допустимым (и при его выполнении происходит соответствующее приведение типа).

```
SELECT WEIGHT
INTO :NW
FROM P
WHERE P# = P# ('P1') ;
```

Аналогичные замечания относятся к операциям INSERT и UPDATE.

9. Для преобразования в типы, из типов или между типами DISTINCT могут быть также явно определены операторы CAST. Подробные сведения об этом здесь не рассматриваются.
10. В случае необходимости могут быть определены дополнительные операторы (а в последующем они могут быть уничтожены).

    Примечание. В языке SQL операторы называются *программными модулями* и подразделяются на три типа: функции, процедуры и методы. (Функции и процедуры могут рассматриваться в определенной степени как аналогичные описанным выше операторам, предназначенным только для чтения, и операторам обновления, соответственно; методы действуют подобно функциям, но вызываются[15] с использованием другого синтаксического стиля.) Поэтому мы можем определить функцию (по сути представляющую собой полиморфную функцию), называемую ADDWT (сокращение от "add weight" — сложение весов), которая позволяет складывать два значения независимо от того, являются ли они значениями типа WEIGHT, или значениями типа DECIMAL(5 , 1), или сочетанием того и другого. В таком случае все приведенные ниже выражения становятся допустимыми.

```
ADDWT ( WT, 14.7 )
ADDWT ( 14.7, WT )
ADDWT ( WT, WT )
ADDWT ( 14.7, 3.0 )
```

    Дополнительную информацию, касающуюся программных модулей SQL, можно найти в [4.20] и [4.28]. Более подробные сведения о них выходят за рамки данной книги.
11. Для уничтожения определяемого пользователем типа используется следующий оператор.

DROP TYPE *<type name>* *<behavior>* ;

Здесь параметр *<behavior>* с обозначением способа выполнения может иметь значение RESTRICT ИЛИ CASCADE; неформально RESTRICT означает, что операция DROP должна окончиться неудачей, если данный тип в настоящее время где-либо используется в базе данных, а ключевое слово CASCADE означает, что операция DROP всегда оканчивается успешно и вызывает неявное применение операций DROP. . .CASCADE ко всем объектам, в которых в настоящее время используется этот тип (именно так!).

Структурированные типы

Перейдем к изучению структурированных типов Ниже приведен ряд примеров.

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT FINAL ;
CREATE TYPE LINESEG AS ( BEGIN POINT, END POINT ) NOT FINAL }_
```

[15] Кроме того, для методов, в отличие от функций и процедур, предусматривается определенное связывание во время прогона (см. главу 20).

Примечание. Термин *метод* и немного странный смысл, который должен быть предписан ему в контекстах, подобных рассматриваемому в данном разделе, происходят из области объектно-ориентированных средств (см. главу 25).

(В действительности попытка выполнить второй оператор окончится неудачей, поскольку BEGIN и END в языке SQL являются зарезервированными словами, но автор ввел эти идентификаторы просто для удобства.) Поэтому в своей простейшей форме (т.е. если не рассматриваются различные необязательные спецификации) оператор создания структурированного типа имеет следующий синтаксис.

```
CREATE TYPE <type name> AS <representation> NOT FINAL ;
```

Ниже приведены дополнительные пояснения.

1. Обязательная спецификация NOT FINAL рассматривается в главе 20.

   Примечание. Предполагается, что в версии SQL:2003 вместо нее будет разрешено задавать альтернативную конструкцию FINAL.

2. Параметр <*representation*> представляет по своей структуре разделенный запятыми список определений атрибутов <*at tribute defini tion commalist*>, заключенный в круглые скобки; в этом списке параметр <*attribute*> состоит из имени атрибута <*attribute name*>, за которым следует имя типа < *type name*>. Но обратите особое внимание на то, что эти так называемые "атрибуты" не являются атрибутами в реляционном смысле, отчасти потому, что структурированные типы не являются реляционными типами (см. главу 6). Более того, применяемый здесь параметр представления <*representation*> обозначает фактическое физическое представление, а не просто некоторое возможное представление рассматриваемого структурированного типа.

   Примечание. Но проектировщик типа имеет возможность эффективно скрыть этот факт (т.е. действительно скрыть, что представление является физическим) благодаря продуманному выбору и тщательному проектированию операторов. Например, если иметь в виду приведенное выше определение типа POINT, то система автоматически предоставляет операторы для получения доступа к представлению точки в виде декартовых координат (см. пункты 3 и 6), но проектировщик типа может "вручную" предоставить операторы для получения доступа также и к представлению в виде полярных координат.

3. Каждое определение атрибута вызывает автоматическое определение двух соответствующих операторов (по сути аналогичных "методам"), причем один из них является *наблюдателем* (observer) и один — *модификатором* (mutator), которые предоставляют функциональные возможности, аналогичные[16] операторам THE_ языка Tutorial D. Например, если переменные LS, P и z, соответственно, относятся к типам LINESEG, POINT и FLOAT, то следующие операторы присваивания являются допустимыми.

---

[16] Ради достижения максимальной точности следует отметить, что в языке SQL модификаторы не являются в действительности модификаторами в обычном смысле этого термина (т.е. они не являются операторами обновления), но могут использоваться таким образом, чтобы с их помощью предоставлялись обычные функциональные средства модификаторов. Например, оператор "SET P.X = Z" (который фактически не содержит явного вызова модификатора!) определен как сокращенный вариант оператора "SET P = P.X(Z)" (который содержит такой вызов).

```
/* Действует как наблюдатель применительно к атрибуту X переменной P
                                                                  */
   SET Z = P.X ;
   /* Действует как модификатор применительно к атрибуту X
   переменной P */
   SET P.X = Z ;
   /* Действует как наблюдатель применительно к атрибуту BEGIN
   переменной LS */
   SET X = LS.BEGIN.X ;

   SET LS.BEGIN.X = Z ;
```

4. В рассматриваемом операторе не предусмотрена какая-либо конструкция, анало гичная спецификации CONSTRAINT языка Tutorial D.
5. Операторы сравнения, которые могут применяться к определяемым структуриро ванным типам, задаются с помощью отдельного оператора CREATE ORDERING. Ниже приведены два примера.

```
CREATE ORDERING FOR POINT EQUALS ONLY BY STATE
; CREATE ORDERING FOR LINESEG EQUALS ONLY BY
STATE ;
```

   Ключевые слова EQUALS ONLY означают, что для значений рассматриваемого типа единственными допустимыми операторами сравнения являются "=" и "/=" (или, скорее, "о", поскольку в языке SQL для обозначения операции сравнения на неравенство используется именно такой знак операции). Ключевые слова BY STATE указывают, что два значения рассматриваемого типа являются равными, если и только если их i-е атрибуты равны при всех значениях i. Описание других возможных спецификаций CREATE ORDERING выходит за рамки данной книги; достаточно сказать, что, например, при желании для структурированного типа может быть также определена семантика оператора ">". Но следует отметить, что если данный структурированный тип не имеет связанного с ним "упорядочения", то со значениями этого типа не могут выполняться никакие операции сравнения, даже сравнения на равенство; вполне очевидно, что такое состояние дел имеет далеко идущие последствия.
6. Ни один из селекторов не предоставляется автоматически, но эффект, аналогичный действию этих операторов, может быть достигнут следующим образом. Прежде всего, язык SQL автоматически предоставляет то, что именуется в нем *функциями-конструкторами* (constructor function), но такие функции при каждом вызове возвращают одно и то же значение, а именно то значение рассматриваемого типа, все атрибуты которого имеют применимое заданное по умолчанию значение[17]. Например, при следующем вызове функции-конструктора

   POINT ()

---

[17] Применяемое по умолчанию значение для определенного атрибута можно указать в составе соответствующего определения атрибута. Если ни одно подобное значение не указано явно, то применяемое по умолчанию значение (своего рода "значение, заданное по умолчанию для использования по умолчанию") будет пустым.

Примечание. По причинам, которые не рассматриваются в этой книге, применяемое по умолчанию значение обязательно должно быть пустым, если типом атрибута является либо тип строки таблицы, либо определяемый пользователем тип (такой как POINT), и он обязательно должен быть задан либо с помощью пустого значения, либо пустого массива (обозначенного как ARRAY [ ]), если это - тип массива. Поэтому, например, вызов функции-конструктора LINESEG () должен обязательно возвратить отрезок прямой, оба компонента которого, BEGIN и END, должны быть пустыми.

возвращается точка с заданными по умолчанию значениями y и X. Но после этого сразу же появляется возможность вызывать модификаторы X и Y (см. пункт 3) для получения любой такой точки, которая должна быть определена с помощью результатов этого вызова функции-конструктора. Более того, существует возможность связать первоначальную операцию "конструирования" и последующую операцию "модификации" в одно выражение, как показано в следующем примере.

```
POINT   ()    .X(5.0)    .Y(2.5)
```

Ниже приведен более сложный пример.

```
LINESEG () . BEGIN ( POINT ()
          . X ( 5 . 0 ) . Y ( 2 . 5 ) )  . END   (
          POINT () . X ( 7 . 3 ) . Y ( 0 . 8 ) )
```

Примечание. При желании перед вызовами функции-конструктора можно ставить необязательное ключевое слово NEW без изменения их семантики, например, следующим образом.

```
NEW LINESEG () . BEGIN ( NEW POINT
             ( ) . X ( 5 . 0 ) . Y ( 2 . 5 ) )  . END   (
             NEW POINT ( ) . X ( 7 . 3 ) . Y ( 0 . 8 ) )
```

7. К структурированным типам применяются правила строгой типизации, за исключением, возможно, тех случаев, которые описаны в главе 6, раздел 6.6 (подраздел "Структурированные типы").
8. Кроме уже упомянутых операторов, в случае необходимости могут быть определены (а в последующем могут быть удалены) дополнительные операторы.
9. Предусмотрена возможность удаления структурированных типов и упорядочений ORDERING. Кроме того, предусмотрена возможность "изменять" определения таких типов с помощью оператора ALTER TYPE, например, можно добавлять новые атрибуты или удалять существующие (иными словами, изменять представление типа).

Дополнительная информация о структурированных типах SQL приведена в следующей главе (раздел 6.6), а также в главах 20 и 26.

Генераторы типов

В языке SQL поддерживаются три генератора типов[18] (в терминологии SQL они называются *конструкторами типов):* REF, ROW и ARRAY. В данной главе рассматриваются только ROW и ARRAY, а описание REF откладывается до главы 26. Ниже приведен пример, иллюстрирующий использование генератора типа строки ROW.

```
   CREATE TABLE CUST
              ( CUST# CHAR(3),
ADDR ROW ( STREET CHAR(50), CITY CHAR(25)
, STATE CHAR(2), ZIP CHAR(5) )   PRIMARY
KEY ( CUST# ) ) ;
```

[18] Существует вероятность того, что в версии SQL:2003 будет введен еще один генератор типа - MULTISET.

Здесь STREET, CITY, STATE и ZIP— поля сгенерированного строкового типа. В общем, такие поля могут иметь любой тип, включая другие строковые типы. В ссылках на уровни поля используется способ уточнения с помощью точки, как в следующем примере (в нем применяется синтаксис <exp>.< *field name*>, где выражение <*exp*> должно иметь значение строкового типа).

```
SELECT    CX.CUST#    FROM
CUST    AS    CX    WHERE
CX.ADDR.STATE = 'CA' ;
```

Примечание. Здесь CX — *имя корреляции.* Имена корреляций подробно рассматриваются в главе 8 (раздел 8.6), а пока просто отметим, что язык SQL требует использования явных имен корреляций в ссылках на имена полей для того, чтобы можно было избежать определенной синтаксической неоднозначности, которая могла бы возникнуть в ином случае.

А ниже приведен пример оператора INSERT.

```
INSERT INTO CUST ( CUST#, ADDR )
VALUES ( ' 666 ' , ROW ( '1600 Pennsylvania Ave.',
                    'Washington', 'DC, '20500' ) ) ;
```

Обратите внимание на использование в данном примере "литерала строки" (фактически он должен был бы называться "литералом строки" и поэтому заключен в кавычки, но формально в языке SQL нет такого понятия, как *литерам строки,* и выражение, применяемое в данном примере, представляет *собой конструктор значения строки).*

Ниже приведен еще один пример.

```
UPDATE CUST AS CX
SET CX.ADDR.STATE = 'TX'
WHERE CUST# = '999 ' ;
```

Примечание. Фактически в стандарте в настоящее время не допускается обновление на уровне поля, как в этом примере, но это упущение можно считать просто недосмотром, который обязательно должен быть устранен.

Генератор типа ARRAY является во многом аналогичным. Ниже приведен соответствующий пример.

```
CREATE TABLE
     ITEM SALES (
     ITEM# CHAR(5),
       SALES INTEGER ARRAY
       [12], PRIMARY KEY (
       ITEM# ) ) ;
```

Типы, сгенерированные с помощью генератора типа ARRAY, всегда являются одномерными; в качестве заданного типа элемента (в данном примере INTEGER) можно использовать все, что угодно, кроме еще одного типа массива[19]. Допустим, что a — значение некоторого типа массива. В таком случае a может содержать любое количество n элементов (n > 0) вплоть до, но не больше указанной верхней границы (в данном примере 12). Если a содержит точно n элементов (n > 0), то этими элементами являются именно a[1], a[2], ..., a[n], и на них можно ссылаться именно таким образом. Выражение CARDINALITY (a) возвращает значение n.

Ниже приведены некоторые примеры, в которых используется таблица ITEM_SALES. Обратите внимание на то, что во втором примере применяется литерал массива (или, скорее, "литерал массива", в кавычках, поскольку формально он называется *конструктором значения массива).*

```
SELECT ITEM#
FROM ITEM SALES
WHERE SALES [3] > 10 ;

INSERT INTO ITEM SALES ( ITEM#, SALES
) VALUES ( 'X4320',
         ARRAY t 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 ] ) ;

UPDATE ITEM SALES SET
SALES [3] = 10 WHERE
ITEM# = 'Z0564' ;
```

Завершаем этот раздел замечанием, что к типам ROW и ARRAY применимы операторы присваивания и сравнения на равенство, при условии, что операторы присваивания и сравнения на равенство применимы к типам элементов рассматриваемого типа ROW ИЛИ ARRAY; если же последние операторы не определены, то нельзя определить и соответствующие операторы для рассматриваемого типа ROW или ARRAY.

5.8. РЕЗЮМЕ

В данной главе приведено всестороннее описание важного понятия **типов данных** (называемых также *доменами* или просто *типами).* Тип представляет собой множество значений, а именно, множество всех значений, которые соответствуют некоторому **ограничению типа** (определяемому в языке Tutorial D с помощью конструкции POSSREP, включающей необязательную спецификацию CONSTRAINT). Каждый тип имеет связанное с ним множество **операторов** (к нему относятся и операторы, предназначенные только для чтения, и операторы обновления), позволяющие оперировать со значениями и переменными рассматриваемого типа. Типы могут быть сколь угодно простыми или

---

[19] Существует вероятность того, что это ограничение в версии SQL:2003 будет снято. В любом случае, типом элемента может быть тип строки, а этот тип строки может включать поле некоторого типа массива. Поэтому (например) ниже приведено допустимое определение переменной.

VX ROW (FX INTEGER ARRAY [12]) ARRAY [12]

И в таком случае (например) VX [3] . FX [5] ссылается на пятый элемент массива, который представляет собой единственное значение поля в строке, которая является третьим элементом массива, представляющим собой значение переменной VX.

сложными, поэтому могут быть предусмотрены типы, значениями которых являются числа, строки, даты, отметки времени, звукозаписи, географические карты, видеозаписи или геометрические точки и т.д. Допустимое множество операторов **зависит от типов** в том смысле, что операнды любой конкретной операции обязательно должны иметь тип, определенный для этой операции **(строгая типизация).** Строгая типизация весьма желательна, поскольку она позволяет своевременно выявлять определенные логические ошибки, которые к тому же обнаруживаются на этапе компиляции, а не на этапе прогона. Следует отметить, что применение строгой типизации влечет за собой особо важные последствия по отношению к реляционным операциям (соединение, объединение и т.д.), как будет описано в главе 7.

Кроме того, в этой главе рассматривалось важное логическое различие между **значениями** и **переменными** и было указано, что существенным свойством значения является то, что его нельзя обновлять. Значения и переменные всегда являются типизированными; таковыми являются также (реляционные) атрибуты, (предназначенные только для чтения) операторы, параметры и, вообще говоря, выражения произвольной сложности.

Типы могут быть **определяемыми системой** или **пользователем;** кроме того, они могут быть **скалярными** или **нескалярными.** Скалярный тип не имеет компонентов, видимых пользователю. (Наиболее важными нескалярными типами в реляционной модели являются типы отношения, которые рассматриваются в следующей главе.) Автор стремится показать важное различие между типом и его **физическим представлением** (типы относятся к уровню модели, а физические представления — к уровню реализации). Но требуется также, чтобы каждый тип имел по меньшей мере одно объявленное **возможное представление** (допустимо также наличие больше одного такого представления). Каждое подобное возможное представление вызывает автоматическое определение одного оператора селектора, а также определение по одному оператору **THE_** для каждого компонента этого возможного представления (включая **псевдопеременную** THE_). Автор поддерживает идею явного **преобразования** типов, но не поддерживает неявные способы **приведения** типов. Автор является также сторонником подхода, предусматривающего определение любого количества дополнительных операторов для скалярных типов, и выдвигает требование, чтобы для каждого типа был предусмотрен оператор **сравнения на равенство** и оператор **присваивания** (возможно, даже множественного).

В настоящей главе рассматривались также **генераторы типов,** представляющие собой операторы, которые возвращают типы (такие как ARRAY). Ограничения и операторы, применимые к сгенерированным типам, происходят от универсальных ограничений и операторов, которые связаны с применимым генератором типа.

Наконец, в этой главе были кратко описаны средства определения типов языка SQL. В языке SQL предусмотрен целый ряд **встроенных типов,** таких как BOOLEAN, INTEGER, DATE, TIME и т.д. (с каждым из которых, безусловно, связано определенное множество операторов), но в связи с этим типами поддерживается лишь ограниченная форма строгой типизации. Этот язык позволяет также пользователям определять свои собственные типы, которые подразделяются на типы **DISTINCT** и **структурированные** типы; кроме того, он поддерживает некоторые генераторы типов (ARRAY и ROW, а также REF). В настоящей главе приведен анализ всех этих функциональных средств SQL с точки зрения тех идей, которые были представлены в ней выше.

УПРАЖНЕНИЯ

5.1.Сформулируйте правила определения типов для операторов присваивания (":=") и сравнения на равенство ("=").

5.2.Покажите, в чем состоят различия:

- значения и переменной;
- типа и представления;
- физического представления и возможного представления;
- скалярной и нескалярной переменной;
- оператора, предназначенного только для чтения, и оператора обновления.

5.3.Дайте своими словами определения следующих терминов:

генератор типа литерал оператор THE_ перечислимый тип полиморфная операция приведение типа псевдопеременная сгенерированный тип селектор строгий контроль типов

5.4.Почему с формальной точки зрения псевдопеременные не являются строго необходимыми?

5.5.Определите оператор, который, получив в качестве параметра рациональное число, возвращает куб этого числа.

5.6.Определите оператор, предназначенный только для чтения, который, получив в качестве параметра точку с декартовыми координатами x и y, возвращает точку с декартовыми координатами f(x) и g (y), где f и g — заранее определенные операторы.

5.7.Повторите упр. 5.6, но примените вместо оператора, предназначенного только для чтения, оператор обновления.

5.8.Дайте определение типа для скалярного типа CIRCLE. Какие селекторы и операторы THE_ применимы к этому типу? Кроме этого, выполните перечисленные ниже задания.

а) Определите множество операторов, предназначенных только для чтения, которые позволяют определить диаметр, периметр и площадь данного круга.

б) Определите оператор обновления, позволяющий удвоить радиус данного круга (точнее, модифицировать заданную переменную CIRCLE таким образом, чтобы представленные с ее помощью параметры круга оставались неизменными, за исключением радиуса, — его новое значение должно увеличиваться вдвое по сравнению с предыдущим).

5.9. Приведите некоторые примеры типов, для которых было бы удобно определить два или больше различных возможных представлений. Если возможно, приведите такой пример, в котором различные возможные представления для одного и того же типа имеют разное количество компонентов.

5.10. Рассмотрим каталог для базы данных отделов и служащих, схематически показанный на рис. 3.6 в главе 3. Как можно расширить этот каталог, чтобы в нем учитывались определяемые пользователем типы и операторы?

5.11. На основе каких типов определены переменные отношения самого каталога?

5.12. Приведите соответствующее множество определений скалярных типов для базы данных деталей, поставщиков и проектов (см. рис. 4.5 на стр. 154). Выполните это упражнение, не предпринимая попыток предварительно записать определения переменной отношения.

5.13. Как было указано в разделе 5.3, строго говоря, недопустимо использовать такую формулировку, что (например) количество предметов в некоторой поставке равно 100. ("Количество— это значение типа QTY, а не значение типа INTEGER!"). Вследствие этого рис. 4.5 является довольно сомнительным, поскольку он наводит на мысль, что, например, значения количества могут рассматриваться как целые числа. С учетом полученного вами ответа на упр. 5.12, покажите правильный способ задания различных скалярных значений на рис. 4.5.

5.14. С учетом полученного вами ответа на упр. 5.12, укажите, какие из следующих скалярных выражений являются допустимыми. Для допустимых выражений укажите тип результата, а для недопустимых составьте соответствующее допустимое выражение, позволяющее получить результат, который, по-видимому, должен быть достигнут в данном случае.

а) `J.CITY = P. CITY.`

б) `JNAME | | PNAME.`

в) `QTY * 100.`

г) `QTY + 100.`

д) `STATUS + 5.`

е) `J.CITY < S.CITY.`

ж) `COLOR = P. CITY.`

з) `J.CITY = P. CITY | | 'burg'.`

5.15. Иногда можно встретить такие утверждения, что типы в действительности также являются переменными, как и переменные отношения. Например, может потребоваться увеличить количество знаков в представлении табельных номеров с трех цифр до четырех в связи с увеличением количества служащих на предприятии. Тогда соответствующую операцию можно было бы назвать обновлением типа "множества всех возможных табельных номеров". Изложите свое мнение по поводу таких утверждений.

5.16. Приведите аналоги всех определений типов, приведенных в разделах 5.3 и 5.4, в контексте языка SQL.

5.17. Приведите ответ к упр. 5.12 на языке SQL.

5.18. В контексте языка SQL ответьте на следующие вопросы.

а) Что такое тип DISTINCT? Какое общее название применяется к значениям типа DISTINCT? Может ли существовать тип, отличный от DISTINCT?

б) Что такое структурированный тип? Какое общее название применяется к значениям структурированного типа? Может ли существовать тип, отличный от структурированного?

5.19. Дайте определения терминов *наблюдатель, модификатор* и *функция-конструктор,* которые используются в языке SQL.

5.20. Каковыми будут последствия того, что для некоторого конкретного типа не будет определен оператор "="?

5.21. Тип — это множество значений, поэтому может быть определен пустой тип, представляющий собой (обязательно уникальный) тип, в котором рассматриваемое множество является пустым. Укажите возможные области применения такого типа.

5.22. Строго говоря, в языке SQL отсутствуют литералы строк или массивов. Объясните смысл этого утверждения и обоснуйте его.

5.23. Рассмотрите тип POINT языка SQL, который определен в подразделе "Структурированные типы" раздела 5.7. Этот тип имеет представление, в котором используются декартовы координаты X и Y. Что произойдет, если этот тип будет заменен пересмотренным типом POINT, имеющим представление, в котором вместо этого используются полярные координаты R и 6?

5.24. В чем состоит различие между операторами CARDINALITY и COUNT языка SQL? *Примечание.* Оператор COUNT рассматривается в разделе 8.6 главы 8.

СПИСОК ЛИТЕРАТУРЫ

5.1. Cleaveland J.C. An Introduction to Data Types // Reading, Mass.: Addison-Wesley, 1986.

ГЛАВА 6

Отношения

6.1. ВВЕДЕНИЕ

В предыдущей главе в целом рассматривались типы, значения и переменные, а в этой главе приведены конкретные сведения о типах, значениях и переменных отношения. Кроме того, поскольку отношения формируются из кортежей (выражаясь довольно неформально), необходимо также рассмотреть типы, значения и переменные кортежей. Но следует сразу же отметить, что кортежи сами по себе не являются столь важными, по крайней мере, с точки зрения реляционной теории; их значимость обусловлена прежде всего тем фактом, что они представляют собой промежуточную ступень на пути к отношениям.

6.2. КОРТЕЖИ

Вначале рассмотрим точное определение термина *кортеж.* Если дана коллекция типов Ti (i = 1, 2, . . ., n), которые не обязательно все должны быть разными, то значением кортежа (или кратко *кортежем),* определенным с помощью этих типов (назовем его t), является множество упорядоченных троек в форме <Ai,Ti,vi>, где Ai — имя атрибута, Ti — имя типа и vi — значение типа Ti. Кроме того, кортеж t должен соответствовать приведенным ниже требованиям.

- Значение n имеет **степень,** или **арность** t.
- Упорядоченная тройка <Ai, Ti, vi> является **компонентом** t.
- Упорядоченная пара <Ai, Ti> представляет собой **атрибут** t и однозначно определяется именем атрибута Ai (имена атрибутов Ai и Aj совпадают, только если i=j). Значение vi — это **значение атрибута,** соответствующее атрибуту[1] Ai кортежа t. Тип Ti — это соответствующий **тип атрибута.**
- Полное множество атрибутов составляет **заголовок t.**
- **Тип кортежа** t определен заголовком t, а сам заголовок и этот тип кортежа имеют такие же атрибуты (и поэтому такие же имена и типы атрибутов) и такую же степень, как t. Ниже показано точное определение **имени типа кортежа.**

```
TUPLE { A1 T1, A2 T2, ..., An Tn }
```

А ниже показан пример кортежа.

MAJOR_P# : P#	MINOR_P# : P#	QTY : QTY
P2	P4	7

В этом кортеже атрибуты имеют имена MAJOR_P#, MINOR_P# и QTY; соответствующими именами типов являются: p#, еще один экземпляр p# и QTY, а соответствующими значениями — P# { ' P2 '), P# (' P4 ') и QTY (7) (для упрощения в этой таблице указанные значения были заменены, соответственно, сокращенными обозначениями P2, P4 и 7). Приведенный выше кортеж имеет степень три, а его заголовок имеет следующий вид.

MAJOR_P# : P#	MINOR_P# : P#	QTY : QTY

Ниже приведено определение типа этого кортежа.

```
TUPLE { MAJOR_P# P#, MINOR_P# P#, QTY QTY }
```

Примечание. В неформальном изложении принято исключать имена типов из заголовка кортежа и показывать только имена атрибутов. Поэтому показанный выше кортеж может быть неформально представлен следующим образом.

MAJOR_P#	MINOR_P#	QTY
P2	P4	7

В языке Tutorial D для обозначения рассматриваемого кортежа может использоваться следующее выражение.

```
TUPLE { MAJOR_P# P#('P2'), MINOR_P# P#('P4'), QTY QTY(7) }
```

(Это выражение может служить примером вызова *селектора кортежа;* см. раздел "Генератор типа TUPLE".) В данном выражении заслуживает особого внимания то, что

---

[1] Безусловно, существует логическое различие между именем атрибута и атрибутом как таковым. Но несмотря на этот факт, в неформальном общении часто используется выражения типа "атрибут Ai" для обозначения атрибута, имеющего имя Ai (в действительности подобные выражения уже встречались несколько раз в предыдущей главе).

типы атрибутов кортежа определены однозначно с использованием заданных значений атрибутов (например, атрибут MINOR_P# ОТНОСИТСЯ К типу p#, поскольку к типу P# относится соответствующее значение атрибута).

Свойства кортежей

Для описания кортежей используется целый ряд важных свойств, которые являются непосредственным следствием определений, приведенных выше в данном разделе. Некоторые из этих свойств перечислены ниже.

- Каждый кортеж содержит точно одно значение (соответствующего типа) для каждого из своих атрибутов.
- Для компонентов кортежа не предусмотрено упорядочение, допустим, слева направо. Это свойство следует из того, что понятие кортежа определено на основании множества компонентов, а в математике множества не характеризуются упорядочением своих элементов.
- Каждое подмножество кортежа представляет собой кортеж (а каждое подмножество заголовка является заголовком). Более того, эти свойства являются истинными применительно, в частности, к пустым подмножествам! (см. следующий абзац).

Дополнительные определения. Кортеж степени один называется одноэлементным, кортеж степени два — двухэлементным, кортеж степени три — трехэлементным (и т.д.); в общем кортеж степени n называют n-элементным[2]. Кортеж нулевой степени (т.е. кортеж без компонентов) называют нуль-элементным или нуль-арным. Кратко проиллюстрируем последнее определение. Ниже приведен нуль-арный кортеж в системе обозначений языка Tutorial D.

```
TUPLE { }
```

Иногда для обозначения кортежа нулевой степени применяется более наглядное представление, такое как "0-элементный кортеж". Это позволяет подчеркнуть тот факт, что данный кортеж не имеет компонентов. На первый взгляд может показаться, что нуль-элементные кортежи почти не имеют практического значения, но фактически оказывается, что это понятие является исключительно важным. Дополнительная информация по этой теме приведена в разделе 6.4.

Генератор типа TUPLE

В языке Tutorial D предусмотрен генератор типа TUPLE, который может быть вызван в определении (например) некоторого атрибута переменной отношения или некоторой переменной кортежа[3]. Ниже приведен пример применения последнего варианта.

---

[2] Вместо термина кортеж иногда используется термин n-элементный кортеж (и по этому принципу формируются, например, выражения четырехэлементный кортеж, двухэлементный кортеж и т.д.). Но обычно принято опускать прилагательное с обозначением количества элементов.

[3] Переменные кортежа не входят в состав реляционной модели, поэтому их применение в реляционной базе данных не предусмотрено. Но такая система, которая полностью поддерживает реляционную модель, по-видимому, могла бы поддерживать возможность использования переменных кортежа в отдельных приложениях (т.е. переменные кортежа представляют собой понятия, "локальные по отношению к отдельным приложениям").

```
VAR ADDR TUPLE { STREET
                 CHAR, CITY
                 CHAR, STATE
                 CHAR, ZIP
                 CHAR } ;
```

Вызов генератора типа TUPLE имеет следующую общую форму.

TUPLE { *<attribute commalist>*)

Здесь каждый параметр *<attribute>* с обозначением атрибута состоит из имени атрибута *<attribute name>,* за которым следует имя типа *<type name>*. Тип кортежа вырабатывается в результате конкретного вызова генератора типа TUPLE; например, тип, только что показанный в определении переменной ADDR, безусловно, является сгенерированным типом.

Каждый тип кортежа имеет связанный с ним оператор *селектора кортежа.* Ниже приведен пример вызова селектора для типа кортежа, показанного в определении переменной ADDR.

```
TUPLE { STREET '1600 Pennsylvania Ave.', CITY 'Washington',
        STATE 'DC', ZIP '20500' }
```

Кортеж, определяемый этим выражением, может быть присвоен переменной кортежа ADDR или проверен на равенство другому кортежу того же типа. В частности, следует отметить, что необходимым и достаточным условием того, чтобы два кортежа имели одинаковый тип, является наличие в них одинаковых атрибутов. Обратите также внимание на то, что сами атрибуты определенного типа кортежа могут относиться к любому типу (они могут даже принадлежать к некоторому типу отношения или некоторому другому типу кортежа).

Операции с кортежами

Выше в данной главе уже кратко упоминались операции применения селектора кортежа, присваивания и проверки на равенство. А теперь необходимо подробно изучить смысл операции проверки кортежей на равенство, поскольку значительная часть материала, приведенного в следующих главах, основана на использовании таких операций. А именно, в терминах этих операций определены все перечисленные далее понятия.

- По сути, все операции реляционной алгебры (см. главу 7).
- Потенциальные ключи (см. главу 9).
- Внешние ключи (см. также главу 9).
- Функциональные и другие зависимости (см. главы 11—13).

Кроме того, на этих операциях основаны определения многих других понятий. Ниже приведено точное определение операции проверки кортежей на равенство.

- Операция проверки кортежа на равенство. Кортежи tl и t2 являются равными (т.е. выражение tl = t2 принимает истинное значение) тогда и только тогда, когда они имеют одинаковые атрибуты A1, A2, . . . , An и для всех i (i = 1, 2, . . ., п) значение vl атрибута Ai в кортеже tl равно значению v2 атрибута Ai в кортеже t2.

- Кроме того (это утверждение может показаться очевидным, но должно быть выражено явно), кортежи t1 и t2 являются **дубликатами** по отношению друг к другу тогда и только тогда, когда они равны в указанном выше смысле (это означает, что они действительно ничем не отличаются друг от друга).

Следует отметить, что из приведенного выше определения непосредственно следует, что все нуль-элементные кортежи являются дубликатами друг друга! По этой причине мы имеем право непосредственно использовать термин *нуль-элементный кортеж,* как мы обычно и делаем (а не рассуждать *о некотором* нуль-элементном кортеже).

Следует также учитывать, что к кортежам не могут применяться операторы сравнения на неравенство, "<" и ">" (т.е. типы кортежа не относятся к категории "порядковых типов").

В дополнение к сказанному выше, в [3.3] предложены аналоги некоторых широко известных реляционных операций (которые будут рассматриваться в главе 7) — проекция кортежа, соединение кортежей и т.д. Эти операции в основном не требуют объяснения, поэтому остановимся на том, что рассмотрим здесь только один пример — проекцию кортежа (эта операция, вероятно, имеет наибольшее практическое значение). Предположим, что переменная ADDR имеет определение, указанное в предыдущем подразделе, а ее текущее значение можно представить следующим образом.

```
TUPLE { STREET 1600 Pennsylvania Ave.',
        CITY 'Washington', STATE 'DC', ZIP '20500' }
```

В этом случае **проекция кортежа**

```
ADDR { CITY, ZIP }
```

определяет следующий кортеж.

```
TUPLE { CITY 'Washington', ZIP '20500' }
```

Нам необходимо также иметь возможность извлекать значение указанного атрибута из определенного кортежа. Например, если переменная ADDR определена, как указано выше, то выражение

```
ZIP FROM ADDR
```

позволяет получить следующее значение.

```
'20500'
```

Ссылка на тип кортежа

Одним из важных преимуществ схемы именования типа кортежа, которая определена в начале данного раздела, является то, что она упрощает задачу выявления типа результата произвольного выражения с участием кортежа. Например, еще раз рассмотрим следующую проекцию кортежа.

```
ADDR { CITY, ZIP }
```

Как уже было сказано, это выражение позволяет получить кортеж, образующийся из текущего значения ADDR путем "выделения проекции" по атрибутам STREET и STATE, а тип кортежа для этого производного кортежа имеет следующее точное определение.

```
TUPLE { CITY CHAR, ZIP CHAR }
```

Аналогичные замечания относятся ко всем возможным выражениям с кортежами.

Операции WRAP и UNWRAP

Рассмотрим следующие типы кортежей.

```
TUPLE { NAME NAME, ADDR TUPLE { STREET CHAR, CITY CHAR,
                                STATE CHAR, ZIP CHAR } }
TUPLE { NAME NAME, STREET CHAR, CITY
                  CHAR, STATE CHAR, ZIP
                  CHAR }
```

В дальнейшем будем использовать для этих типов кортежей, соответственно, обозначения TT1 и TT2. Следует, в частности, отметить, что тип TT1 включает атрибут, который сам относится к определенному типу кортежа (но степень TT1 равна двум, а не пяти). Теперь предположим, что NADDR1 и NADDR2 — переменные кортежа типов TT1 и TT2, соответственно. В таком случае могут быть выполнены описанные ниже операции.

- Выражение

```
NADDR2 WRAP { STREET, CITY, STATE, ZIP } AS ADDR
```

  принимает текущее значение переменной NADDR2 и сворачивает компоненты STREET, CITY, STATE и ZIP этого значения для получения компонента ADDR, значением которого является один кортеж. Поэтому результатом такого выражения является переменная типа TT1, и в связи с этим (например) следующий оператор присваивания становится допустимым.

```
NADDR1 := NADDR2 WRAP { STREET, CITY, STATE, ZIP } AS ADDR ;
```

- Выражение

```
NADDR1 UNWRAP ADDR
```

  принимает текущее значение переменной NADDR1 и разворачивает компоненты значения переменной ADDR (значением которой является кортеж) для получения четырех отдельных компонентов STREET, CITY, STATE и ZIP, поэтому результат данного выражения относится к типу TT2, и в связи с этим (например) следующий оператор присваивания становится допустимым.

```
NADDR2 := NADDR1 UNWRAP ADDR ;
```

Сравнение типов кортежей и возможных представлений

Внимательный читатель мог заметить определенную аналогию между синтаксисом генератора типа TUPLE, который описан в этом разделе, и синтаксисом объявленного возможного представления, который рассматривается в главе 5 (в обоих синтаксических определениях используются разделенные запятыми списки элементов, а каждый элемент определяет имя некоторого объекта и имя соответствующего типа), поэтому у читателя может возникнуть вопрос о том, имеем ли мы дело с двумя понятиями или только с одним. В действительности это два отдельных понятия (и синтаксическое сходство определений не имеет значения). Например, если X относится к типу кортежа, то на практике вполне может потребоваться получить проекцию некоторого значения этого типа, как описано в предыдущем подразделе. Но если X — возможное представление для некоторого скалярного типа т, то задача получения проекции значения этого скалярного типа т не возникает. Дополнительные сведения по этой теме приведены в [3.3].

6.3. ТИПЫ ОТНОШЕНИЙ

Теперь перейдем к изучению отношений. В этом описании часто будут рассматриваться аналогии с определениями, касающимися кортежей, которые были сформулированы в предыдущем разделе, но применительно к отношениям должно быть приведено намного больше информации по сравнению с кортежами, поэтому соответствующий материал был разбит на несколько разделов: в разделе 6.3 рассматриваются типы отношения, в разделе 6.4 — значения отношений, а в разделе 6.5 — *переменные отношения* (relation variable, или сокращенно relvar).

Вначале рассмотрим точное определение термина *отношение.* Значение отношения (или просто *отношение),* допустим r, состоит из заголовка и тела[4], которые соответствуют определениям, приведенным ниже.

- Заголовок отношения r представляет собой заголовок кортежа, определение кото рого приведено в разделе 6.2. Отношение r имеет такие же атрибуты (следова тельно, такие же имена и типы атрибутов) и такую же степень, как заголовок.
- Тело отношения r представляет собой множество кортежей, имеющих один и тот же заголовок; кардинальность отношения r определяется как кардинальность этого множества. (Вообще говоря, *кардинальностью множества* называется коли чество элементов множества.)

Тип отношения r определяется заголовком r и имеет такие же атрибуты (следовательно, имена и типы атрибутов) и степень, как и сам заголовок. Ниже приведено точное определение имени типа отношения.

```
RELATION { A1 T1, A2 T2, ..., An Tn }
```

А здесь показан пример отношения (он аналогичен, но не идентичен отношению, показанному на рис. 4.6 в главе 4).

MAJOR_P# : P#	MINOR_P# : P#	QTY : QTY
P1	P2	5
P1	P3	3
P2	P3	2
P2	P4	7
P3	P5	4
P4	P6	8

Это отношение имеет следующий тип.

```
RELATION { MAJOR_P# P#, MINOR_P# P#, QTY QTY }
```

[4] Если рассматривается обычное изображение отношения в виде таблицы, то заголовок соответствует строке имен столбцов, а тело — множеству строк с данными. В литературе заголовок именуется также схемой (отношения) или иногда просто схемой. Кроме того, заголовок иногда называют содержанием (intension), и в таком случае для обозначения тела применяется термин расширение (extension).

В неформальном изложении принято исключать из заголовка отношения имена типов и показывать только имена атрибутов. Поэтому приведенное выше отношение можно неформально представить следующим образом.

MAJOR_P#	MINOR_P#	QTY
P1	P2	5
P1	P3	3
P2	P3	2
P2	P4	7
P3	P5	4
P4	P6	8

В языке Tutorial D для обозначения этого отношения можно использовать следующее выражение.

```
RELATION {
  TUPLE{ MAJOR P# P#('P1'),MINOR P# P#('P2'), QTY QTY(5) }      ,
  TUPLE{ MAJOR P# P#('P1'),MINOR P# P#('P3'), QTY QTY(3) }      ,
  TUPLE{ MAJOR P# P#('P2'),MINOR P# P#('P3'), QTY QTY(2) }      ,
  TUPLE{ MAJOR P# P#('P2'),MINOR P# P#('P4'), QTY QTY(7) }      ,
  TUPLE{ MAJOR P# P#('P3'),MINOR P# P#('P5'), QTY QTY(4) }      ,
  TUPLE{ MAJOR_P# P#('P4'),MINOR_P# P#('P6'), QTY QTY(8)       } }
```

Это выражение представляет собой пример вызова селектора отношения, который имеет следующий общий формат.

RELATION [*<heading>] { <tuple exp commalist> }*

Здесь необязательный параметр *<heading>,* представляющий собой разделенный запятыми список атрибутов <attribute>, заключенный в фигурные скобки, требуется, только если не задан параметр с обозначением списка выражении кортежа < tuple exp *coimalist*>. Безусловно, все выражения кортежа < tuple exp> должны относиться к одному и тому же типу кортежа, а этот тип кортежа должен быть точно таким же, как и тот, который определен параметром *<heading>,* если задан этот параметр.

Следует отметить что отношение, строго говоря, не содержит кортежей - оно содержит тело а тело в свою очередь включает в себя кортежи. Но при неформальном общении удобно формулировать свои высказывания так, как если бы отношения непосредственно включали кортежи, и мы во всей этой книге будем следовать этим соглашениям, позволяющим упростить изложение материала.

Как и в случае кортежей, отношение степени один называется *унарным,* отношение степени два - *бинарным,* отношение степени три - *тернарным* (и т.д.); в общем, отношение степени п называется *п-арным.* Отношение нулевой степени (т.е. отношение без атрибутов) принято называть *нуль-арным* (последний вариант отношения подробно рассматривается в следующем разделе). Кроме того, следует отметить, что справедливы приведенные ниже утверждения.

- Каждое подмножество заголовка является заголовком (как и в случае кортежей).
- Каждое подмножество тела является телом.

В обоих случаях рассматриваемое подмножество может, в частности, быть пустым подмножеством.

Генератор типа RELATION

В языке Tutorial D предусмотрен генератор типа RELATION, который может быть вызван (например) в определении некоторой переменной отношения. Соответствующий пример приведен ниже.

```
VAR PART_STRUCTURE ...
     RELATION { MAJOR_P# P#, MINOR_P# P#, QTY QTY } ... ;
```

(Здесь для упрощения исключены те части определения, которые не относятся к данной теме.) В общем, генератор типа RELATION имеет такую же форму, как и генератор типа TUPLE, за исключением того, что вместо ключевого слова TUPLE применяется ключевое слово RELATION. Тип отношения, выработанный в результате конкретного вызова генератора типа RELATION (например, только что показанный в определении переменной отношения PART_STRUCTURE), безусловно, является сгенерированным типом.

Каждый тип отношения имеет соответствующий оператор селектора отношения. Выше уже был приведен пример вызова селектора для только что показанного типа отношения. Отношение, полученное с помощью этого вызова селектора, может быть присвоено переменной отношения PART_STRUCTURE или проверено на равенство с другим отношением такого же типа. В частности, следует отметить, что необходимым и достаточным условием того, чтобы два отношения принадлежали к одному и тому же типу, является наличие в них одинаковых атрибутов. Необходимо также учитывать, что атрибуты любого заданного типа отношения сами могут принадлежать к любому типу (они могут даже принадлежать к некоторому типу кортежа или к некоторому другому типу отношения).

6.4. ЗНАЧЕНИЯ ОТНОШЕНИЙ

В этом разделе мы приступим к более подробному изучению отношений (т.е. значений отношений) как таковых. Прежде всего, необходимо отметить, что отношения обладают определенными свойствами; все эти свойства непосредственно следуют из определения отношения, приведенного в предыдущем разделе, и все они являются очень важными. Вначале сформулируем рассматриваемые свойства, а затем подробно их опишем. Любое конкретное отношение обладает свойствами, указанными ниже.

1. Каждый кортеж содержит точно одно значение (соответствующего типа) для каждого атрибута.
2. Атрибуты не характеризуются каким-либо упорядочением (например, слева направо).
3. Кортежи не характеризуются каким-либо упорядочением (например, сверху вниз).
4. В отношении отсутствуют дубликаты кортежей.

Для иллюстрации этих свойств воспользуемся отношением с данными о поставщиках, приведенным на рис. 3.8 (см. стр. 119). Для удобства это отношение снова показано на рис. 6.1, но заголовок в нем расширен для включения имен типов.

Примечание. Формально мы должны были бы расширить также тело для включения имен атрибутов и типов. Например, запись S# для поставщика si должна была бы фактически выглядеть следующим образом.

S# S# S#('S1')

S# : S#	SNAME : NAME	STATUS : INTEGER	CITY : CHAR
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Рис. 6.1. Отношение с данными о поставщиках, приведенное на рис. 3.8

Но для упрощения оставим тело этого отношения в таком же виде, в каком оно было первоначально представлено на рис. 3.8.

1. *Отношения нормализованы*

   Как было указано в разделе 6.2, каждый кортеж содержит точно одно значение для каждого из своих атрибутов, поэтому можно сделать неоспоримое заключение, что каждый кортеж в каждом отношении содержит точно одно значение для каждого из своих атрибутов. Отношение, которое соответствует этому свойству, называется **нормализованным;** иначе мысль можно выразить таким образом, что отношение находится в **первой нормальной форме[5], 1НФ.** Согласно этому определению, отношение, показанное на рис. 6.1, является нормализованным.

   Примечание. Это первое свойство может показаться вполне очевидным и оно действительно является таковым, особенно в связи с тем, что (как мог уже заметить читатель) все отношения нормализованы по определению! Тем не менее, из этого свойства вытекают некоторые важные следствия. См. подраздел "Атрибуты со значениями в виде отношений" ниже в этом разделе, а также главу 19 (где речь идет о недостающей информации).

2. *Атрибуты не характеризуются каким-либо упорядочением (например, слева направо)*

   Как уже было сказано, компоненты кортежа не характеризуются каким-либо упорядочением (например, слева направо), и аналогичное утверждение является справедливым применительно к атрибутам отношения (фактически по той же причине, а именно, в связи с тем, что заголовок отношения представляет собой множество атрибутов, а множества в математике не характеризуются упорядочением своих элементов). Однако если мы изображаем отношение на бумаге в виде таблицы, то мы, естественно, вынуждены показывать столбцы этой таблицы в определенном порядке расположения слева направо, но читатель должен всегда учитывать, что этот порядок не имеет значения. Например, на рис. 6.1 столбцы отношения вполне могли быть показаны (допустим) в порядке слева направо, SNAME, CITY, STATUS, S#, и этот рисунок все равно представлял бы то же отношение,

[5] Эта форма получила название "первой" в связи с тем, что могут быть также определены некоторые "более высокие" нормальные формы — вторая, третья и т.д. (см. главы 12 и 13).

по меньшей мере, если оно рассматривается как объект реляционной модели[6]. Поэтому не может быть такого понятия, как "первый атрибут" или "второй атрибут" (и т.д.), а также не может идти речь о "следующем атрибуте" (т.е. в отношении не определено понятие "следования"); атрибуты всегда должны быть указаны по именам, а не по их позициям. Благодаря этому устраняется значительная часть предпосылок для возникновения ошибок и появления крайне запутанных программ. Например, отсутствует возможность определять методы, не предусмотренные в системе и позволяющие каким-то образом "перебирать" один атрибут за другим, просто наращивая указатель. Эта ситуация отличается в лучшую сторону по сравнению со многими системами программирования, часто допускающими возможность вольно или невольно пользоваться тем, что логически не связанные элементы физически расположены близко друг от друга, несмотря на то, что это приводит к созданию программ, чреватых многочисленными ошибками.

3. *Кортежи не характеризуются каким-либо упорядочением (например, сверху вниз)* Это свойство следует из того факта, что тело отношения также представляет собой множество (кортежей); еще раз подчеркнем, что множества в математике не упорядочены. Но изображая отношения на бумаге в виде таблицы, мы вынуждены показывать строки этой таблицы в некотором порядке расположения сверху вниз; тем не менее, читатель должен учитывать, что этот порядок не имеет значения. Например, на рис. 6.1 строки вполне можно было бы также показать в обратном порядке, притом что сам рисунок по-прежнему представлял бы то же отношение. Поэтому не существует такого понятия, как "первый кортеж", "пятый кортеж" или "97-й кортеж" отношения, кроме того, не существует такого понятия, как "следующий кортеж"; иными словами, в отношениях не определена позиционная адресация и нет понятия "следования". Заслуживает внимание то, что если бы указанные понятия были определены, потребовались бы также некоторые дополнительные операции, например, "выполнить выборку л-го кортежа", "вставить этот новый кортеж здесь", "переместить этот кортеж отсюда сюда" и т.д. А поскольку существует один и только один способ представления информации в реляционной модели (что является непосредственным следствием информационного принципа Кодда), очень мощным средством реляционной модели является то, что для обработки этой информации достаточно иметь одно и только одно множество операторов.

   Рассмотрим последнее замечание более подробно. Фактически не вызывает сомнения такое утверждение, что если существует N разных способов представления информации, то требуется N различных множеств операторов, а если N > 1, то приходится реализовывать, документировать, объяснять студентам, учить самим, запоминать и использовать больше операторов. Но введение каждого дополнительного оператора приводит лишь к возрастанию сложности, а не выразительной мощи! Не существует каких-либо полезных операторов, которые могут применяться, только если N > 1,а не N= 1. Эта тема будет более подробно рассматриваться в главе 26 (см. **[26.12]—[26.14]** и [26.17]), а затем снова появится в главе 27.

---

[6] По причинам, которые нас здесь не интересуют, отношения в математике, в отличие от их аналогов в реляционной модели, характеризуются упорядочением своих атрибутов слева направо (и это замечание, безусловно, относится и к кортежам).

Вернемся к описанию самих отношений. Безусловно, определенные требования по упорядочению кортежей сверху вниз (а также по упорядочению атрибутов слева направо) выдвигаются в связи с созданием интерфейса между базой данных и базовым языком, таким как C или COBOL (см. сведения о курсорах SQL и конструкции ORDER BY, приведенные в главе 4). Но эти требования налагает базовый язык, а не реляционная модель; в действительности базовый язык требует преобразования неупорядоченных отношений в упорядоченные списки, или массивы (кортежи), именно для того, чтобы приобрели смысл операции, подобные "выборке п-го кортежа". Аналогичным образом, определенные требования по упорядочению кортежей должны соблюдаться при представлении результатов запросов для конечного пользователя. Но эти требования не входят в состав реляционной модели как таковой; скорее, они составляют часть той среды, в которой определена сама реализация этой реляционной модели. **4.** *Отсутствие в отношении дубликатов кортежей*

Это свойство также следует из того факта, что тело отношения представляет собой множество; множества в математике не содержат дубликатов элементов (иначе эту мысль можно выразить таким образом, что все элементы множества являются различными).

Примечание. Это свойство также служит для иллюстрации идеи о том, что понятия отношения и таблицы являются разными, поскольку таблица может включать дубликаты строк (если не соблюдаются некоторые требования, позволяющие предотвратить такую возможность), а отношение, по определению, никогда не содержит каких-либо дубликатов кортежей.

И действительно, это является (или должно быть) очевидным, что само понятие "дубликатов кортежей" лишено смысла. Для упрощения примем предположение, что отношение, приведенное на рис. 6.1, имеет только два атрибута, s# и CITY (как показано в разделе 6.5, интерпретация этих атрибутов соответствует их назначению— "Поставщик s# находится в городе CITY"), а также предположим, что отношение содержит кортеж, согласно которому имеет место "истинный факт", что поставщик S1 находится в Лондоне. В таком случае, если бы отношение содержало дубликат указанного кортежа (при условии, что это вообще возможно), оно просто информировало бы нас о том же "истинном факте" во второй раз. Но если нечто является истинным, то при многократном повторении оно не становится более истинным!

Развернутое описание проблем, которые могут быть вызваны наличием дубликатов кортежей в отношении, можно найти в [6.3] и [6.6].

Сравнение отношений и таблиц

Для использования в дальнейшем представим в этом подразделе приведенный ниже список некоторых основных различий между формальным объектом, который является отношением как таковым, и неформальным объектом (таблицей), представляющим собой неформальное изображение этого формального объекта на бумаге.

1. Для каждого атрибута в заголовке отношения предусмотрено имя типа, но эти имена типов обычно не показаны на изображениях в виде таблицы.

2. Для каждого компонента каждого кортежа в теле отношения предусмотрено имя типа и имя атрибута, но эти имена типа и атрибута обычно не показаны на изображениях в виде таблицы.
3. Каждое значение атрибута в каждом кортеже в теле отношения является значением соответствующего типа, но эти значения на изображениях в виде таблицы обычно показаны в сокращенной форме, например, s1 вместо s# (' S1').
4. Столбцы в таблице характеризуются упорядочением слева направо, а атрибуты отношения — нет.

   Примечание. Важным следствием из этого различия является то, что столбцы могут иметь повторяющиеся имена или вообще не иметь имен. Например, рассмотрим следующий запрос SQL.

```
SELECT S.CITY, S.STATUS * 2,
P.CITY FROM S, P ;
```

   Какими будут имена столбцов в результатах этого запроса?

5. Строки таблицы характеризуются упорядочением сверху вниз, а кортежи отношения — нет.
6. Таблица может содержать дубликаты строк, а отношение не содержит дубликаты кортежей.

Приведенный список наиболее важных различий не является исчерпывающим. Ниже перечислены некоторые другие различия.

- Безусловно, что таблицы обычно рассматриваются как имеющие по меньшей мере один столбец, а отношения не обязательно должны иметь хотя бы один атрибут (см. подраздел "Отношения без атрибутов" ниже в этом разделе).
- Безусловно, допускается (по крайней мере, в языке SQL), чтобы таблицы включали неопределенные значения (NULL), в то время как для отношений это ни в коем случае не допускается (см. главу 19).
- Безусловно, что таблицы являются "плоскими" (или двухмерными), а отношения — n-мерными (см. главу 22).

Из всего сказанного выше следует, что мы должны принять определенное соглашение о том, как "читать" подобные изображения в виде таблицы, чтобы иметь право рассматривать эти изображения как приемлемые средства представления отношений; иными словами, необходимо принять определенные правила интерпретации для таких изображений. Точнее, необходимо принять соглашение о том, что каждый столбец характеризуется соответствующим типом, каждое значение атрибута представляет собой значение определенного типа, упорядочение строк и столбцов не имеет значения и дубликаты строк не допускаются. Таблица может рассматриваться как приемлемое изображение отношения тогда и только тогда, когда соблюдаются все эти правила ее интерпретации.

Итак, теперь вполне очевидно, что в действительности между таблицей и отношением существуют значительные различия (но часто бывает удобно игнорировать эти различия). Отношение скорее следует считать таким, как сказано в его определении, — вполне абстрактным объектом, а таблица является конкретным изображением такого абстрактного объекта, обычно на бумаге. Следует еще раз подчеркнуть, что таблица и отношение

представляют собой разные объекты. Безусловно, между этими объектами есть много общего, поэтому обычно принято и вполне приемлемо рассматривать их как одинаковые, по меньшей мере, в неформальном изложении. Но если требуется полная ясность (и мы как раз хотим добиться полной ясности), то следует учитывать, что эти два понятия не являются полностью идентичными.

В заключение следует также отметить, что фактически одним из важных преимуществ реляционной модели является то, что ее основной абстрактный объект, отношение, имеет такое простое представление на бумаге. Именно благодаря этому простому представлению реляционные системы становятся удобными для использования и изучения, к тому же намного упрощается само обсуждение особенностей функционирования реляционных систем. Но, к сожалению, следует отметить, что этот простой способ представления может навести также на мысль, которая не соответствует действительности (например, о том, что отношение характеризуется упорядочением кортежей сверху вниз).

Атрибуты со значениями в виде отношения

Как было отмечено в разделе 6.3, в общем случае в качестве основы для определения реляционных атрибутов может использоваться любой тип. Из этого, в частности, следует, что основой для определения атрибутов отношений могут явиться типы отношений, поскольку они, безусловно, представляют собой типы. Иными словами, атрибуты могут иметь **значения в виде отношений,** а это означает, что в отношениях могут применяться атрибуты, значениями которых, в свою очередь, являются отношения. Иначе говоря, допустимо существование таких отношений, в которые вложены другие отношения. Пример такого отношения показан на рис. 6.2. Применительно к этому отношению можно отметить, что в данном отношении, во-первых, атрибут PQ имеет значение в виде отношения, во-вторых, его кардинальность и степень равны пяти, и в частности, в-третьих, пустое множество деталей, поставляемых поставщиком S5, представлено в виде значения PQ, которое является пустым множеством (точнее, пустым отношением).

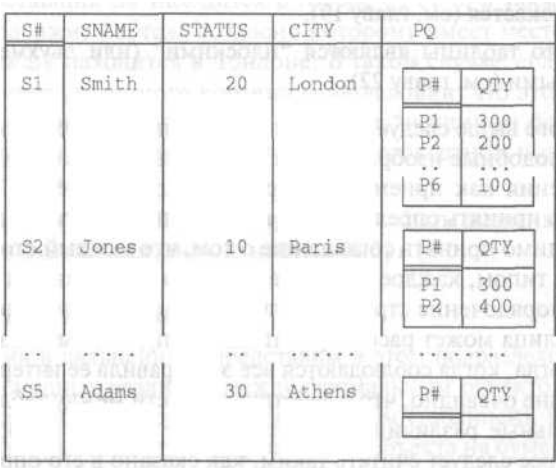


Рис. 6.2. Отношение с атрибутом, значением которого является отношение

Основная причина, по которой мы здесь подчеркиваем возможность применения атрибутов со значениями в виде отношений, состоит в том, что в свое время такая возможность обычно считалась недопустимой. В действительности она рассматривалась как таковая и в предыдущих изданиях настоящей книги. Например, ниже приведена немного отредактированная выдержка из шестого издания.

Следует отметить, что все значения столбцов являются атомарными... Это означает, что в каждой позиции на пересечении строки и столбца (именно так!) в каждой таблице (именно так!) должно всегда находиться одно и только одно значение данных, а не группа из нескольких значений. Поэтому, например, в таблице EMP должны присутствовать такие строки

DEPT#	EMP#
D1	E1
D1	E2
...	...

вместо следующей одной строки.

DEPT#	EMP#
D1	E1,E2
...	...

Во втором варианте этой таблицы столбец EMP# представляет собой пример того, что обычно называют повторяющейся группой. *Повторяющаяся группа — это столбец..., который содержит несколько значений в каждой строке (и, в общем, разное количество значений в различных строках). В реляционных базах данных повторяющиеся группы не допускаются; поэтому второй вариант приведенной выше таблицы не должен использоваться в реляционной системе.*

И ниже в том же шестом издании мы находим следующее утверждение.

Домены (т.е. типы) содержат только атомарные значения... [Поэтому] отношения не должны включать повторяющиеся группы. Отношение, удовлетворяющее этому условию, называется нормализованным *или (равным образом) находящимся в первой нормальной форме... В контексте реляционной модели термин* отношение *всегда принято трактовать как нормализованное отношение.*

Но эти замечания были неправильными (по меньшей мере, не совсем правильными). Они явились следствием того, что в свое время сам автор не понимал истинного характера типов (доменов). По причинам, которые будут обсуждаться в главе 12 (раздел 12.6), маловероятно, чтобы это заблуждение могло привести на практике к каким-либо очень серьезным ошибкам; тем не менее, автор считает свои долгом принести извинения всем, кого он ввел в заблуждение. По крайней мере, этот текст из шестого издания был правильным в тех местах, где в нем говорилось, что отношения в реляционной модели всегда нормализованы! Дополнительная информация по этой теме приведена также в главе 12.

Отношения без атрибутов

Каждое отношение имеет множество атрибутов, а поскольку пустое множество также, безусловно, представляет собой множество, то из этого следует, что возможно наличие такого отношения, которое имеет пустое множество атрибутов или, иными словами, вообще не имеет атрибутов. (Постарайтесь избежать путаницы: мы часто говорим о "пустых отношениях", подразумевая под этим отношения, телом которых является пустое множество кортежей, но здесь, напротив, речь идет об отношениях, заголовками которых служит пустое множество атрибутов.) Поэтому отношения без атрибутов являются, по меньшей мере, вполне допустимыми с математической точки зрения. Но что, возможно, покажется читателю наиболее удивительным, так это то, что эти отношения оказались чрезвычайно важными также и с практической точки зрения!

Для того чтобы перейти к более подробному изучению понятия *отношения без атрибутов,* вначале необходимо рассмотреть вопрос о том, может ли отношение без атрибутов содержать какие-либо кортежи. Ответ на этот вопрос (что также может показаться удивительным) является положительным. Точнее, подобное отношение может содержать, самое большее, один кортеж, а именно, нуль-арный кортеж (т.е. кортеж без компонентов), причем оно не может содержать больше одного такого кортежа, поскольку все нуль-арные кортежи являются дубликатами по отношению друг к другу. Таким образом, существуют два и только два отношения нулевой степени — содержащие лишь один кортеж и не содержащие вообще ни одного кортежа. Эти два отношения являются настолько важными, что, следуя Дарвену (Darwen) [6.5], автор предусмотрел для них псевдонимы. Первое отношение будет именоваться TABLE_DEE, а второе — TABLE_DUM или сокращенно DEE и DUM (DEE — это отношение с одним кортежем, aDUM — пустое).

Примечание. Для этих отношений нелегко изобразить таблицу! Принятый способ трактовки отношений как обычных таблиц начинает немного буксовать в случае отношений DEE И DUM.

Почему отношении DEE и DUM ЯВЛЯЮТСЯ такими важными? На этот вопрос есть несколько ответов, более или менее связанных между собой. Одним из этих ответов является то, что эти отношения в реляционной алгебре играют такую роль (см. главу 7), которая немного напоминает роль, принадлежащую пустому множеству в теории множеств или нулю в обычной арифметике. Другой ответ касается того, что вообще подразумевается под отношениями (см [6.5]); по сути DEE означает TRUE, или да, а DUM означает FALSE, или нет. Иными словами, эти отношения имеют наиболее фундаментальный смысл по сравнению со всеми другими отношениями. (Удобный способ запомнить, какое из этих отношений что означает, состоит в том, что в отношении DEE есть буквы "e", как и в слове yes — да.)

В языке Tutorial D выражения TABLE_DEE и TABLE_DUM могут использоваться в качестве сокращений, соответственно, для следующих вызовов селектора отношений.

```
RELATION { } { TUPLE { } }
```

И

```
RELATION { } { }
```

Пока мы не имеем возможности более подробно углубиться в эту тему; достаточно сказать, что в последующем изложении отношения DEE и DUM встретятся еще много раз. Дополнительную информацию можно найти в [6.5].

Операции с отношениями

В разделе 6.3 кратко рассматривались реляционные операции применения селектора, присваивания и сравнения на равенство. Вне всякого сомнения, операторы сравнения на неравенство "<" и ">" не могут применяться к отношениям, но отношения, безусловно, могут стать предметом сравнений другого рода в дополнение к простому сравнению на равенство, как будет показано в этом разделе.

Реляционные операции сравнения

Начнем с определения логического выражения <*bool exp*> нового типа, с оператором сравнения отношений <*relation comp*>, которое имеет следующий синтаксис.

<*relation exp*> <*relation comp op*> <*relation exp*>

Отношения, обозначенные двумя параметрами <*relation exp*> (реляционное выражение), должны принадлежать к одному и тому же типу, а оператором сравнения отношений <*relation comp op*> должен быть один из следующих операторов.

- =. Равно.
- $\neq$ Не равно.
- $\subseteq$ Подмножество.
- $\subset$ Строгое подмножество.
- $\supseteq$ Надмножество.
- $\supset$ Строгое надмножество.

После этого можно разрешить использовать операцию <*relation comp*> в любом месте, где допускается применение логического выражения <*boo1* exp>, например, в конструкции WHERE. Ниже приведен ряд примеров.

1. S { CITY } = P { CITY }

   Это выражение имеет такой смысл: "Является ли проекция отношения поставщиков по атрибуту CITY равной проекции отношения деталей по атрибуту CITY?"

2. S { S# } $\supset$ SP { S# }

   Это выражение имеет такой смысл (представленный в нем вопрос немного перефразирован): "Есть ли такие поставщики, которые вообще не поставляют детали?"

В частности, одной из операций реляционного сравнения, которая особенно часто применяется на практике, является операция проверки в целях определения того, равно ли указанное отношение пустому отношению того же типа (т.е. отношению, не содержащему кортежей). Для данного конкретного случая удобно иметь некоторый сокращенный способ проверки. Поэтому определим следующее выражение.

IS_EMPTY (<*relation exp*>)

Это выражение возвращает истинное значение TRUE, если отношение, обозначенное как реляционное выражение <*relation exp*> является пустым, а в противном случае возвращает ложное значение FALSE.

Другие операторы

Еще одним широко распространенным требованием является обеспечение возможности проверить, присутствует ли указанный кортеж t в указанном отношении r.

t Є r

Это выражение возвращает TRUE, если t присутствует в r, а в противном случае возвращает FALSE (" Є " является оператором проверки принадлежности к множеству; выражение t Є r можно читать как "t принадлежит к r", или "t является элементом r", или, проще всего, "t [имеется] в r").

Необходимо также иметь возможность извлечь один кортеж из отношения с кардинальностью один, следующим образом.

TUPLE FROM r

Это выражение активизирует исключение, если r не содержит точно один кортеж; в противном случае оно возвращает лишь этот один кортеж.

Кроме операторов, описанных выше, предусмотрены также все универсальные операторы (соединение, сокращение, проекция и т.д.), которые определены в реляционной алгебре. Отложим подробное описание этих операторов до следующей главы.

Ссылка на тип отношения

Точно так же, как схема именования типа кортежа, описанная в разделе 6.2, упрощает задачу определения типа результата выражения, в котором используется произвольный кортеж, так и схема именования типа отношения, описанная в разделе 6.3, упрощает задачу определения типа результата произвольного реляционного выражения. Более подробные сведения по этой теме приведены в главе 7, а в этом разделе остановимся на одном простом примере. Если дана переменная отношения поставщиков S, то выражение

S { S#, CITY }

приводит к получению результата (отношения), который имеет следующий вид.

RELATION { S# S#, CITY CHAR }

Аналогичные замечания относятся ко всем возможным реляционным выражениям.

Оператор ORDER BY

Для удобного представления результатов весьма желательно поддерживать оператор ORDER BY, как и в языке SQL (см. главу 3). Подробное определение этого оператора здесь не приведено, поскольку его семантика, безусловно, должна быть очевидной. Но следует обратить внимание на перечисленные ниже особенности этого оператора.

- Оператор ORDER BY действует (по сути), сортируя кортежи в некоторой указанной последовательности, несмотря на то, что для кортежей не определены операторы "<" и ">" (именно так!).

- Оператор ORDER BY не является реляционным оператором, поскольку возвращаемый им результат не представляет собой отношение.
- Оператор ORDER BY не является также функцией, поскольку в общем с его помощью может быть получено много разных наборов выходных данных для одного и того же входного набора.

В качестве примера следствия из последнего замечания рассмотрим результат применения операции ORDER BY CITY к отношению поставщиков, показанному на рис. 6.1. Безусловно, эта операция способна возвратить любой из четырех различных результатов. В отличие от этого, операторы реляционной алгебры несомненно являются функциями, поскольку они при получении любого заданного набора входных данных всегда возвращают только один соответствующий ему возможный набор выходных данных.

6.5. ПЕРЕМЕННЫЕ ОТНОШЕНИЯ

Теперь обратимся к переменным отношения (relation variable, или сокращенно relvar). Как было отмечено в главе 3, переменные отношения имеют две разновидности — базовые переменные отношения и представления (называемые также, соответственно, реальными и виртуальными переменными отношения). В данном разделе нас в основном интересует именно базовые переменные отношения (представления подробно рассматриваются в главе 10), но следует отметить, что все сказанное здесь применительно к переменным отношения без дополнительного уточнения распространяется на все переменные отношения в целом, включая представления.

Определение базовой переменной отношения

Ниже показан синтаксис определения базовой переменной отношения.

```
VAR <relvar name> BASE <relation type>
                    <candidate key def list>
                  [ <foreign key def list> ] ;
```

Параметр с обозначением типа отношения *<relation type>* принимает следующую форму.

```
RELATION { <attribute commalist> }
```

Здесь параметр *<attribute commalist>* — разделенный запятыми список атрибутов. Это выражение фактически представляет собой вызов генератора типа RELATION, как описано в разделе 6.3). Параметр *<candidate key def list>* и необязательный параметр *<foreign key def list>* описаны в одном из следующих абзацев. Ниже в качестве примера приведены определения базовых переменных отношения для базы данных поставщиков и деталей (которые были также показаны на рис. 3.9).

```
VAR S BASE RELATION
    { S# S#,
    SNAME NAME, STATUS
    INTEGER, CITY CHAR
    } PRIMARY KEY { S#
    } ;
```

```
VAR P BASE RELATION { P# P#,
     PNAME NAME,
     COLOR COLOR,
     WEIGHT WEIGHT,
     CITY CHAR }
     PRIMARY KEY { P#};

VAR SP BASE
     RELATION { S#
     S#, P# P#, QTY
     QTY }
     PRIMARY KEY { S#, P# }
     FOREIGN KEY { S# }
     REFERENCES S FOREIGN KEY { P#
     } REFERENCES P ;
```

Пояснения

1. Эти три базовые переменные отношения имеют следующие типы (отношения).

```
RELATION { S# S#, SNAME NAME, STATUS INTEGER, CITY
CHAR } RELATION { P# P#, PNAME NAME, COLOR COLOR,
          WEIGHT WEIGHT, CITY
CHAR } RELATION { S# S#, P# P#,
QTY QTY }
```

2. Все термины *заголовок, тело, атрибут, кортеж, степень* (и т.д.), которые были пе ред этим определены для значений отношения, интерпретируются также очевид ным образом применительно к переменным отношения.
3. Все возможные значения любой отдельно взятой переменной отношения принад лежат к одному и тому же типу отношения, а именно к типу отношения, указан ному в определении переменной отношения (указанному косвенно, если данная переменная отношения является представлением), и поэтому имеют одинаковый заголовок.
4. При определении базовой переменной отношения ей присваивается *начальное зна чение,* которое является пустым отношением соответствующего типа.
5. Определения потенциальных ключей подробно рассматриваются в главе 9, а на данный момент мы будем просто предполагать, что определение каждой базовой переменной отношения включает одно и только одно определение потенциаль ного ключа,<*candidate key def*> в следующей конкретной форме.

PRIMARY KEY { <*attribute name commalist*> }

6. Определения внешнего ключа также рассматриваются в главе 9.
7. Ввод определения новой переменной отношения вынуждает систему внести в свой каталог записи с описанием этой переменной отношения.
8. Как было отмечено в главе 3, переменные отношения, как и отношения, имеют соответствующий *предикат.* Таковым является предикат, общий для всех отноше ний, которые представляют собой возможные значения рассматриваемой пере менной отношения. Например, в случае переменной отношения поставщиков S предикат выглядит примерно следующим образом.

Поставщик с номером поставщика S# работает по контракту, имеет имя SNAME и статус STATUS, а также находится в городе CITY

9. Предполагается, что предусмотрены средства определения **применяемых по умолчанию значений** для атрибутов базовых переменных отношения. *Применяемым по умолчанию значением* заданного атрибута является такое значение, которое должно быть помещено в позицию соответствующего атрибута, если пользователь явно не предоставит нужное значение при вставке некоторого кортежа. Подходящая синтаксическая конструкция языка Tutorial D для определения применяемых по умолчанию значений может принимать форму новой конструкции в определении базовой переменной отношения, скажем, DEFAULT { <d*efault spec commalist*> }, где каждая спецификация применяемого по умолчанию значения <*default speo* принимает форму <af*ctribute name*> <*default*> *(<имя атрибута> <применяемое по умолчанию значение>).* Например, в определении переменной отношения поставщиков s может быть задана конструкция DEFAULT { STATUS 0, CITY ' ' }.

   Примечание. Атрибуты потенциальных ключей обычно не имеют заданных по умолчанию значений, хотя из этого правила есть исключения (см. главу 19).

Ниже приведен синтаксис операции удаления существующей базовой переменной отношения.

DROP VAR <*relvar name*> ;

Эта операция присваивает указанной переменной отношения "пустое" значение (т.е., выражаясь неформально, удаляет все кортежи в этой переменной отношения); затем она удаляет все записи каталога для этой переменной отношения. После этого переменная отношения становится неизвестной для системы.

Примечание. Для упрощения предполагается, что операция DROP оканчивается неудачей, если рассматриваемая переменная отношения все еще используется в другом месте, например, если на нее имеется ссылка в каком-то определении представления, заданном где-то в базе данных.

Обновление переменных отношения

Реляционная модель включает операцию реляционного присваивания, позволяющую присваивать значения переменным отношения (в частности базовым переменным отношения), т.е. обновлять их. Ниже приведено немного упрощенное определение синтаксиса этой операции на языке Tutorial D.

<*relation assignment*>
    ::= <*relation assign cormalist*>
; <*relation assign*>
    ::= <*relvar name*> := <*relation exp*>

Эта операция имеет такую семантику[7]: во-первых, вычисляются все реляционные выражения <*relation exp*> в правых частях операций реляционного присваивания <*relation assign*>, во-вторых, выполняются операции <relation *assign*> в той

[7] За исключением той ситуации, которая отмечена в сноске 9 в главе 5.

последовательности, в которой они записаны. Выполнение отдельной операции <*relation assign*> сводится к тому, что отношение, полученное в результате вычисления выражения <*relation exp*> в правой части, присваивается переменной отношения, обозначенной именем <*relvar name*> в левой части (с заменой предыдущего значения этой переменной отношения). Безусловно, само отношение и переменная отношения должны принадлежать к одному и тому же типу.

В качестве примера предположим, что даны еще две базовые переменные отношения, S' и SP', имеющие, соответственно, такие же типы, как и переменная отношения поставщиков S и переменная отношения поставок SP, как показано ниже.

```
VAR S' BASE RELATION
    { S# S#, SNAME NAME, STATUS INTEGER, CITY CHAR }
... ; VAR SP1 BASE RELATION
    { S# S#, P# P#, QTY QTY } ... ;
```

Ниже приведены некоторые допустимые примеры применения операции присваивания отношения<relation *assignment*>.

```
1. S' := S , SP1 := SP ;
2. S' := S WHERE CITY = 'London' ;
3. S' := S WHERE NOT ( CITY = 'Paris' ) ;
```

Следует отметить, что каждая отдельная операция <*relation assign*> может одновременно рассматриваться, во-первых, как выборка отношения, заданного в правой части, и, во-вторых, как обновление переменной отношения, указанной в левой части.

Теперь предположим, что во второй и в третий примеры внесены такие изменения — в каждом случае переменная отношения S ' в левой части заменена переменной отношения S, как показано ниже.

```
2.  S := S WHERE CITY = 'London' ;
3.  S := S WHERE NOT ( CITY = 'Paris' ) ;
```

Обратите внимание на то, что обе эти операции присваивания фактически обновляют переменную отношения S — одна из них по сути удаляет всех поставщиков, не находящихся в Лондоне, а другая фактически удаляет всех поставщиков, не находящихся в Париже. Для удобства в языке Tutorial D поддерживаются явно заданные операции INSERT, DELETE и UPDATE, но каждая из этих операций определена как сокращение для некоторого присваивания отношения <*relation assign*>. Далее приведено несколько примеров.

```
1. INSERT S RELATION { TUPLE { S# S# ('S6'),
                               SNAME NAME ('Smith'),
                               STATUS 50,
                               CITY 'Rome' } } ;
```

Эквивалентная операция присваивания представлена следующим образом.

```
S := S UNION RELATION { TUPLE { S# S# ('S6'),
                                SNAME NAME ('Smith'),
                                STATUS 50, CITY 'Rome' } }
                                ;
```

Кстати, следует отметить, что эта операция присваивания завершится успешно, если указанный кортеж с данными о поставщике S6 уже существует в переменной отношения S. На практике желательно было бы уточнить семантику операции INSERT таким образом, чтобы она активизировала исключение, если предпринимается попытка "вставить уже существующий кортеж". Но для упрощения мы здесь это уточнение игнорируем. Аналогичные замечания относятся также к операторам DELETE И UPDATE.

```
2. DELETE S WHERE CITY = 'Paris' ;
```

Эквивалентная операция присваивания представлена следующим образом.

```
S := S WHERE NOT ( CITY = 'Paris' ) ;
```

```
3. UPDATE S WHERE CITY = 'Paris'
                { STATUS := 2 *
                  STATUS, CITY :=
                  'Rome' } ;
```

Эквивалентная операция присваивания представлена следующим образом.

```
S := WITH ( S WHERE CITY = 'Paris' ) AS T1 ,
        ( EXTEND T1 ADD ( 2 * STATUS AS NEW STATUS,
                          'Rome' AS NEW CITY ) ) AS
         T2 T2 { ALL BUT STATUS, CITY } AS T3 , ( T3
         RENAME ( NEW STATUS AS STATUS,
                  NEW CITY AS CITY ) ) AS
     T4 : ( S MINUS T1 ) UNION T4 ;
```

Вполне очевидно, что в последнем случае эквивалентный оператор присваивания становится довольно сложным; фактически в нем используются некоторые средства, которые будут подробно рассматриваться только в следующей главе. По этой причине в данном разделе не дано более подробное описание данной темы.

Для использования в дальнейшем ниже приведены немного упрощенные сведения о синтаксисе операций INSERT, DELETE И UPDATE.

INSERT *<relvar name>* *<relation* **exp>;**
DELETE *<relvar name>* **[WHERE *<bool exp>*] ;**
UPDATE *<relvar name>* **[WHERE *<bool exp>***
]
{ <attribute update commalist> } ;

Параметр с определением операции обновления атрибута *<attribute update>* в свою очередь принимает следующую форму.

<attribute name> := *<exp>*

Кроме того, логическое выражение *<boo1 exp>* в операторах DELETE и UPDATE может включать ссылки на атрибуты целевой переменной отношения, имеющие вполне очевидную семантику.

В завершение этого подраздела следует подчеркнуть, что все операции реляционного присваивании, и следовательно, операции INSERT, DELETE и UPDATE ОТНОСЯТСЯ К уровню множества[8]. Например, операция UPDATE, неформально выражаясь, обновляет множество

[8] Между прочим, следует отметить, что по определению формы CURRENT операторов DELETE и UPDATE в языке SQL (см. раздел 4.6) относятся к уровню кортежа (или, скорее, к уровню строки) и поэтому их применять не рекомендуется.

кортежей в целевой переменной отношения. В неформальном изложении часто можно встретить такое высказывание, что (например) обновляется некоторый отдельный кортеж, но следует четко понимать, что фактически при этом должны учитываться перечисленные ниже соображения.

1. Речь фактически может идти только об обновлении множества кортежей, которое просто оказалось множеством с кардинальностью один.
2. Иногда обновление множества кортежей с кардинальностью один является невозможным!

Предположим, например, что на переменную отношения поставщиков распространяется такое ограничение целостности (см. главу 9), что поставщики si и S4 должны иметь одинаковый статус. В таком случае любая "однокортежная" операция UPDATE, в которой предпринимается попытка изменить статус только одного из этих двух поставщиков, должна закончиться неудачей. Вместо этого должна быть предусмотрена операция, в которой данные обоих поставщиков обновляются одновременно, как показано ниже.

```
UPDATE S WHERE S# = S# ('S1') OR Si = S#
       ('S4') { STATUS := <некоторое
       значение> } ;
```

В дополнение к сказанному выше следует отметить, что выражение "обновление кортежа, или множества кортежей" в переменной отношения, которое мы только что привели, является фактически довольно сомнительным. Как и отношения, кортежи представляют собой значения и не могут быть обновлены по определению. Поэтому когда речь идет (например) об "обновлении кортежа t с преобразованием его в t' ", фактически под этим подразумевается, что происходит **замена** кортежа t (т.е. значения кортежа t) другим кортежем t' (который, в свою очередь, также представляет собой значение кортежа)[9]. Аналогичные поправки относятся и к таким выражениям, как "обновление атрибута А" в некотором кортеже. В настоящей книге будут по-прежнему использоваться выражения "обновление кортежей" и "обновление атрибутов кортежей", поскольку такая практика очень удобна, но следует учитывать, что такое словоупотребление является лишь сокращением, причем достаточно сомнительным.

Переменные отношения и их интерпретация

В завершение этого раздела напомним, что в соответствии с указанным в разделе 3.4 главы 3, заголовок любой заданной переменной отношения может рассматриваться как **предикат,** а кортежи, присутствующие в этой переменной отношения в любой конкретный момент времени, могут рассматриваться как **истинные высказывания,** полученные из этого предиката путем подстановки фактических параметров соответствующего типа вместо формальных параметров предиката (или "путем конкретизации предиката"). Поэтому можно считать, что предикат, соответствующий заданной переменной отношения, представляет собой **намеченную интерпретацию,** или смысл этой переменной отношения, а высказывания, соответствующие кортежам данной переменной отношения, рассматриваются как истинные по определению. В действительности, **предположение о**

[9] Безусловно, сказанное здесь отнюдь не означает, что нельзя обновлять переменные-кортежи. Но как было описано в разделе 6.2, понятие переменной-кортежа в реляционной модели не определено и реляционные базы данных не включают такие переменные.

замкнутости мира (называемое также **интерпретацией замкнутого мира)** гласит, что если кортеж, допустимый согласно всем иным признакам (т.е. кортеж, соответствующий заголовку переменной отношения), не присутствует в теле переменной отношения, то можно рассматривать соответствующее ему высказывание как ложное. Иными словами, тело переменной отношения в любой конкретный момент времени содержит те и только те кортежи, которые соответствуют высказываниям, являющимся истинными **на** этот момент времени. Эта тема рассматривается гораздо более подробно в главе 9.

6.6. СРЕДСТВА SQL

Строки

В языке SQL вообще не поддерживаются кортежи как таковые; вместо этого в нем поддерживаются *строки,* которые характеризуются упорядочением своих компонентов слева направо. Поэтому в каждой конкретной строке значения компонентов (которые называются *значениями столбцов,* если строка непосредственно содержится в таблице, или *значениями полей* в противном случае) идентифицируются прежде всего по своей порядковой позиции (даже если они имеют также и имена, что не всегда бывает на практике). Типы строк не имеют явных имен типа строки. Значение строки может быть "выбрано" (в языке SQL для обозначения этой операции используется термин *сконструировано*— constructed) с помощью некоторого выражения (фактически являющегося конструктором значения строки <*row value constructor*>) в следующей форме.

[ROW] (<*exp commalist*>)

Круглые скобки могут быть опущены, если разделенный запятыми список *commalist* содержит только одно выражение <exp>; в таком случае должно быть также опущено ключевое слово ROW, которое в ином случае является необязательным. Список, разделенный запятыми, не должен быть пустым (язык SQL не поддерживает "нуль-арные строки"). Ниже приведен пример применения конструктора строки.

```
ROW ( P#('P2'), P#('P4'), QTY(7) )
```

В этом выражении создается строка степени три.

Как было показано в главе 5, в языке SQL поддерживается также конструктор типа ROW (в языке Tutorial D ему соответствует генератор типа TUPLE), который может быть вызван, например, в определении некоторого столбца таблицы или некоторой переменной[10]. Ниже приведен пример определения переменной.

```
DECLARE ADDR ROW ( STREET
                   CHAR(50), CITY
                   CHAR(25), STATE
                   CHAR(2), ZIP
                   CHAR(5) ) ;
```

Поддерживаются операции присваивания и сравнения строк с учетом того, что используются лишь те правила строгой типизации в ограниченной форме, которые описаны в разделе 5.7 главы 5. Поэтому следует обратить особое внимание на то, что если выражение

[10] Следует учитывать, что "конструктор значения строки языка SQL" является фактически селектором кортежа, а "конструктор типа строки", предусмотренный в этом языке, является по сути генератором типа TUPLE (выражаясь весьма неформально!).

r1 = r2 является истинным, это отнюдь не означает, что строки rl и г2 одинаковы. Более того, в языке SQL операторы сравнения строк "<" и ">" являются допустимыми! Но подробные сведения об этих операторах сравнения являются весьма объемными и здесь не приведены; дополнительную информацию по этой теме можно найти в [4.20].

В языке SQL не поддерживаются строковые аналоги любой из обычных реляционных операций (такие как "проекция строки", "соединение строки" и т.д.), а также не предусмотрены непосредственные аналоги для операций WRAP и UNWRAP. Кроме того, в нем не поддерживаются какие-либо операции "формирования ссылок на тип строки", но это последнее свойство языка, по-видимому, не имеет особого значения, поскольку SQL почти вообще не поддерживает какие-либо строковые операции.

Типы таблиц

Язык SQL вообще не поддерживает отношения как таковые; вместо этого он поддерживает таблицы. В языке SQL тело таблицы является не множеством кортежей, а мультимножеством строк (мультимножество — multiset — представляет собой коллекцию, которая, как и множество, не имеет упорядочения, но в отличие от множества, допускает наличие дубликатов элементов); поэтому столбцы такой таблицы имеют упорядочение слева направо и в ней могут присутствовать дубликаты строк. (Но в настоящей книге мы придерживаемся определенных требований, гарантирующих то, что дубликаты строк никогда не появятся, даже в контексте SQL.) В языке SQL не используются такие термины, как *заголовок* или *тело.*

Типы таблиц не имеют явного имени типа таблицы. Значение таблицы может быть "выбрано" (еще раз отметим, что в языке SQL применяется термин *сконструировано* — constructed) с помощью некоторого выражения (фактически конструктора значения таблицы *<table value constructor>)* в следующей форме.

VALUES *<row value constructor commalist>*

Здесь разделенный запятыми список *commalist* не должен быть пустым. Поэтому, например, следующее выражение

```
VALUES   (          P# ( ' P1' ) ,        P# ( ' P2 ' ) ,        QTY ( 5 )        )
         ,
         ( P#('P 1 '), P#('P 3 '), QTY(3) ) ,
         ( P#('P 2 '), P#('P 3 '),         QTY(2)     )        ,
         ( P#('P 2 '), P#('P 4 '),         QTY(7)     )        ;
         ( P#('P 3 '), P#('P 5 '), QTY(4) ) ,
         ( P#('P 4 '), P#('P 6 '), QTY(8) )
```

имеет своим результатом таблицу, которая внешне немного напоминает отношение, приведенное в разделе 6.3, за исключением того, что в ней нет явно заданных имен столбцов.

В языке SQL фактически вообще не поддерживаются явные аналоги генератора типа RELATION. В нем также явно не поддерживается оператор присваивания таблицы (но явно поддерживаются операторы INSERT, DELETE И UPDATE). Кроме того, в этом языке не поддерживаются какие-либо операторы сравнения таблиц (даже оператор "="). Но в SQL предусмотрен оператор, позволяющий проверить, присутствует ли указанная строка в указанной таблице, как показано ниже.

```
<row value constructor> IN <table exp>
```

Кроме того, в этом языке предусмотрен следующий аналог оператора TUPLE FROM.

(*<table exp>*)

Если подобное выражение появляется там, где требуется отдельная строка, и если выражение таблицы *<table exp>* обозначает таблицу, содержащую одну и только одну строку, то возвращается эта строка; в противном случае активизируется исключение.

Примечание. Кстати, следует отметить, что имя таблицы *< table name>* не является допустимым выражением таблицы *< table exp>* (!?).

Значения и переменные таблицы

К сожалению, в языке SQL применительно к *значению таблицы* и к *переменной таблицы* используется один и тот же термин — *таблица.* Поэтому в данном разделе термин *таблица* следует рассматривать как относящийся либо к значению таблицы, либо к переменной таблицы, в зависимости от того, что требует контекст. Поэтому здесь вначале приведен синтаксис SQL для определения базовой таблицы.

CREATE TABLE *<base table name>*
(*<base table element cornmalxst>*) ;

Каждое выражение с обозначение элемента базовой таблицы *<base table element>* представляет собой определение столбца *<column definition>* или ограничение *<constraint>,* как описано ниже[11].

- Выражения *<constraint>* определяют некоторые ограничения целостности, которые относятся к рассматриваемой базовой таблице. Отложим подробное об суждение таких ограничений до главы 9 и отметим лишь то, что таблицы SQL не обязательно должны иметь первичный ключ (или, что еще более важно, вообще какие-либо потенциальные ключи), поскольку в них допускается наличие дубли катов строк.
- Выражения *<column definition>* (должно быть предусмотрено по меньшей мере одно такое выражение) имеют следующую общую форму.

<column name> <type name> [*<default speo*]

Необязательное выражение *«defaul t speo* определяет заданное по умолчанию значение (или просто *значение по умолчанию),* которое должно быть помещено в соответствующий столбец, если пользователь явно не задаст никакого значения в операторе INSERT (подходящий пример приведен в главе 4, раздел 4.6, подраздел "Операции, в которых не используются курсоры"). Оно принимает форму DEFAULT *<default>,* где *<default>* — литерал, имя встроенного безоперандного оператора[12] или ключевое слово NULL (см. главу 19). Если для некоторого столбца явно не предусмотрено значение по умолчанию, то неявно предполагается,

---

[11] Выражение <base table element> может также принимать форму LIKE T, которая позволяет копировать часть или все объявления столбцов для определяемой базовой таблицы из некоторой существующей именованной таблицы T.

[12] *Безоперандным оператором* называется такой оператор, который не имеет явно заданных операндов. В качестве примера можно указать CURRENT_DATE.

что он по умолчанию должен иметь неопределенное значение (NULL-значение); т.е. неопределенное значение "применяется по умолчанию в качестве заданного по умолчанию значения" (фактически это правило всегда соблюдается в языке SQL).

Примечание. По причинам, описание которых выходит за рамки этой книги, применяемое по умолчанию значение обязательно должно быть неопределенным, если данный столбец имеет тип, определяемый пользователем (как уже было указано в главе 4). Оно должно быть также неопределенным, если столбец относится к некоторому строковому типу, и должно представлять собой либо неопределенное значение, либо пустой массив (определяемый как ARRAY [ ]), если столбец относится к типу массива.

Для ознакомления с некоторыми примерами применения операции CREATE TABLE можно обратиться к рис. 4.1 в главе 4. Следует отметить, что (как уже было сказано) язык SQL не поддерживает столбцы со значениями в виде таблицы, а также не поддерживает таблицы вообще без столбцов. Кроме того, в нем предусмотрена возможность использовать оператор ORDER BY наряду с аналогами большинства операторов реляционной алгебры (см. главы 7 и 8). Но применяемые в нем правила для "формирования ссылок на тип таблицы" (хотя они, безусловно, существуют) заданы неявно, по крайней мере, частично; кроме того, они являются довольно сложными и в этом разделе более подробные сведения по этой теме не будут приведены.

Предусмотрена возможность уничтожения существующей базовой таблицы. Синтаксис соответствующей операции приведен далее.

'DROP TABLE <*base table name*> <*behavior*> ;

Здесь (как и в случае оператора DROP TYPE, описанного в главе 5) выражение <*behavior*> может принимать значение RESTRICT или CASCADE. Неформально говоря, ключевое слово RESTRICT означает, что операция DROP должна окончиться неудачей, если таблица в настоящее время где-либо используется, а ключевое слово CASCADE означает, что операция DROP всегда завершается успешно и вызывает неявное применение оператора DROP. . .CASCADE KO всем объектам, в которых в настоящее время используется эта таблица. Кроме того, с помощью оператора ALTER TABLE может быть изменено определение любой существующей базовой таблицы. Поддерживаются описанные ниже разновидности такого изменения.

- Добавление нового столбца.
- Определение нового значения, применяемого по умолчанию, для существующего столбца (или замена предыдущего, если оно было задано).
- Удаление существующего значения, применяемого по умолчанию для столбца.
- Удаление существующего столбца.
- Определение нового ограничения целостности.
- Удаление существующего ограничения целостности.

Ниже приведен пример применения только первой разновидности такой операции.

ALTER TABLE S ADD COLUMN DISCOUNT INTEGER DEFAULT -1 ;

В этом операторе происходит добавление столбца DISCOUNT (типа INTEGER) к базовой таблице поставщиков. Все существующие строки в этой таблице расширяются с четырех столбцов до пяти; во всех случаях первоначальное значение нового, пятого столбца становится равным -1.

Наконец отметим, что операторы INSERT, DELETE и UPDATE языка SQL уже были описаны в главе 4.

Структурированные типы

Предостережение. Те части стандарта SQL, которые относятся к этому подразделу, являются сложными для понимания. Поэтому автор постарался сделать все от него зависящее, чтобы этот материал стал более легким для восприятия.

Поэтому, прежде всего, в этом разделе повторно приведен следующий пример определения структурированного типа из главы 5 (раздел 5.7).

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT FINAL ;
```

Теперь тип POINT может использоваться в определениях переменных и столбцов, например, как показано ниже.

```
CREATE TABLE NADDR (
     NAME ... , ADDR ...
     , LOCATION POINT
     ... ,
```

Кроме того, хотя об этом не было достаточно ясно сказано в главе 5, но, по меньшей мере, подразумевалось, что *структурированные типы SQL* являются именно *скалярными типами,* точно так же, как скалярным типом является аналог указанного выше типа POINT на языке Tutorial D. Но в некоторых отношениях структурированные типы SQL ближе[13] к типам кортежей языка Tutorial D. Безусловно, это верно, что мы так же можем иметь доступ к компонентам ("атрибутам") любого конкретного значения POINT, как если бы это был кортеж. Для такой цели используется синтаксис с уточнителями, разделенными точками, как показано в следующих примерах (следует учитывать, что требуются явно заданные имена упоминаемых компонентов).

```
SELECT NT.LOCATION.X,
NT.LOCATION.Y FROM NADDR AS NT
WHERE NAME = ... ;

UPDATE NADDR AS NT
SET NT.LOCATION.X =5.0
WHERE NAME = ... ;
```

При таком его использовании, как это показано в приведенном выше примере, структурированный тип SQL фактически действует так, как если бы он был простым строковым типом (рекомендуем еще раз обратиться к разделу 5.7 главы 5), за исключением перечисленных ниже особенностей.

---

[13] Если не считать того, что структурированные типы характеризуются упорядочением своих атрибу-тов слева направо, а типы кортежей — нет.

- Его компоненты именуются атрибутами, а не полями.
- Что еще более важно, структурированные типы, в отличие от строковых типов, имеют имена (мы еще раз вернемся к этой теме в самом конце данного раздела).

Поэтому до сих пор структурированные типы SQL на первый взгляд выглядели так, как будто в них нет ничего слишком сложного для понимания. Но (и это очень важно!) на этом относящиеся к ним сведения не заканчиваются[14]. В дополнение к изложенному выше, в языке SQL разрешено также определять базовую таблицу как принадлежащую к некоторому структурированному типу, с использованием ключевого слова "OF", и в этом случае необходимо учитывать целый ряд дополнительных соображений. Для того чтобы было проще анализировать некоторые из этих соображений, вначале расширим определение типа POINT следующим образом.

```
CREATE TYPE POINT AS ( X FLOAT, Y FLOAT ) NOT
       FINAL REF IS SYSTEM GENERATED ;
```

Теперь можно определить базовую таблицу как относящуюся к этому типу с помощью ключевого слова "OF", например, следующим образом.

```
CREATE TABLE POINTS OF POINT
     ( REF IS POINT # SYSTEM GENERATED ... ) ;
```

Пояснения

1. При определении структурированного типа т система автоматически определяет связанный с ним ссылочный тип (тип REF) с именем REF{T). Значения типа REF(T) являются *ссылками на строки* в той же базовой таблице[15], которая была определена как относящаяся к типу т с помощью ключевого слова "OF" (см. п. 3). Поэтому в данном примере система автоматически определяет тип с именем REF (POINT), значениями которого являются ссылки на строки в базовой табли це. А эта таблица, в свою очередь, определена как относящаяся к типу POINT с помощью ключевого слова "OF".
2. Спецификация REF IS SYSTEM GENERATED В операторе CREATE TYPE означа ет, что фактические значения соответствующего типа REF ДОЛЖНЫ быть преду смотрены системой (возможны и другие варианты, например, REF IS USER GENERATED, но в этом разделе они не рассматриваются).

   Примечание. В действительности конструкция REF IS SYSTEM GENERATED применяется по умолчанию, поэтому в данном примере при желании можно было бы оставить неизменным первоначальное определение типаPOINT.
3. Базовая таблица POINTS определена как относящаяся к структурированному типу POINT с помощью ключевого слова "OF". Но фактически ключевое слово OF здесь не очень хорошо подходит, поскольку таблица в действительности не состоит *из*

---

[14] То же самое относится и к приведенному ниже обсуждению! Дополнительная информация по этой теме приведена в главах 20 и 26.

[15] Или, возможно, некоторого представления. Изложение подробных сведений о том варианте при менения этой конструкции, который относится к представлениям, выходит за рамки этой книги.

(что буквально означает слово "of) рассматриваемого типа и тем более из него не состоят ее строки! Дополнительные сведения по этой теме приведены ниже[16].

- Прежде всего, если бы таблица имела только один столбец и этот столбец относился к рассматриваемому структурированному типу, скажем, ST, то можно было бы утверждать (но не применительно к конструкциям языка SQL!) нечто вроде того, что таблица относится к типу TABLE (ST), а ее строки — к типу ROW(ST).
- Однако в общем таблица не имеет только один столбец; вместо этого в ней предусмотрено по одному столбцу для каждого атрибута ST. Поэтому в данном примере базовая таблица POINTS имеет два столбца, X и Y; но она явно не имеет столбца типа POINT.
- Более того, данная таблица имеет также один дополнительный столбец, а именно столбец соответствующего типа REF. НО синтаксис определения этого столбца отличается от обычного синтаксиса определения столбца и выглядит примерно следующим образом.

```
REF IS <column name> SYSTEM GENERATED
```

Этот дополнительный столбец <*column name*> называется *столбцом, ссылающимся на самого себя;* он используется для хранения уникальных идентификаторов или "ссылок" для строк рассматриваемой базовой таблицы. Идентификатор для некоторой строки присваивается при вставке этой строки и остается связанным с ней до момента ее удаления. Поэтому в данном примере базовая таблица POINTS фактически имеет три столбца (POINT#, X и Y, в указанном порядке), а не просто два столбца.

Примечание. Не совсем ясно, почему должно соблюдаться требование вначале определять с помощью ключевого слова "OF" таблицу как относящуюся к некоторому структурированному типу, а не просто обычным образом объявлять соответствующий столбец, чтобы иметь возможность использовать это функциональное средство "уникального идентификатора", но приведенное здесь описание построено в соответствии с теми принципами, которые заложены в языке SQL.

Кстати, следует отметить (хотя это может показаться удивительным), что столбец, созданный с помощью конструкции SYSTEM GENERATED, т.е. сгенерированный системой, может быть целевым столбцом в операции INSERT или UPDATE, хотя и с учетом некоторых специальных оговорок. Здесь подробные сведения об этом не приведены.

4. Таблица POINTS может служить примером того, что в стандарте SQL именуется (не очень удачно) и *типизированной таблицей* (typed table), и *таблицей, на которую может быть сделана ссылка* (referenceable table). Приведем цитату из стандарта, в

---

[16] Поэтому следует, в частности, отметить, что если объявленный тип некоторого формального параметра P некоторого оператора Op относится к определенному структурированному типу ST, то строка из базовой таблицы, которая была определена как относящаяся к типу ST с помощью ключевого слова "OF", не может быть передана в качестве соответствующего фактического параметра при вызове этого оператора Op.

которой упоминается таблица такого типа: "Таблица ..., тип строки которой происходит от структурированного типа, называется типизированной таблицей. Типизированной таблицей может быть только базовая таблица или представление". И еще одна цитата: "Таблица, на которую может быть сделана ссылка, обязательно является также типизированной таблицей... . Типизированной таблицей называется таблица, на которую может быть сделана ссылка".

На данный момент складывается впечатление, что описанные выше средства были введены в стандарте SQL: 1999 в основном для использования в качестве основы для внедрения в язык SQL своего рода "объектных функциональных средств"[17], которые будут подробно рассматриваться в главе 26. Но в стандарте ничего не сказано о том, что рассматриваемые средства могут использоваться только в сочетании с этими функциональными возможностями, и поэтому автор решил привести их описание именно в данной главе.

Приведем еще одно заключительное замечание. Как было указано в главе 5, в языке Tutorial D нет никакого явного оператора "определения типа кортежа"; вместо этого в указанном языке предусмотрен генератор типа TUPLE, который может быть вызван (например) в определении переменной кортежа. Вследствие этого, единственными именами, которые могут иметь в языке Tutorial D типы кортежей, являются имена в следующей форме.

```
TUPLE { A1 T1, A2 T2, . . . , An Tn }
```

Из этого следует важный вывод, что можно сразу же проверить, являются ли два типа кортежа фактически одинаковыми и относятся ли два кортежа к одному и тому же типу.

Итак, типы строк в языке SQL аналогичны типам кортежей в языке Tutorial D в указанном отношении. Но структурированные типы являются иными; предусмотрена явная операция "определения структурированного типа", а структурированные типы к тому же имеют явно заданные имена. В качестве примера рассмотрим следующие определения SQL.

```
CREATE TYPE P0INT1 AS ( X FLOAT, Y FLOAT ) NOT
FINAL ; CREATE TYPE P0INT2 AS ( X FLOAT, Y FLOAT
) NOT FINAL ; DECLARE V1 P0INT1 ; DECLARE V2
P0INT2 ;
```

Здесь заслуживает особого внимания то, что переменные V1 и V2 относятся к разным типам. Поэтому они не могут сравниваться друг с другом, а значения одной переменной не могут присваиваться другой переменной.

6.7. РЕЗЮМЕ

В данной главе приведено подробное описание отношений и других связанных с ними объектов. Вначале было приведено точное определение понятия **кортежей** и особенно отмечены те их свойства, что каждый кортеж содержит одно и только одно значение каждого

---

[17] Этот вывод автора полностью подтверждается тем фактом, что структурированные типы SQL всегда имеют связанный с ними тип REF, даже если этот тип REF не выполняет никаких функций, за исключением того, что рассматриваемый структурированный тип используется в качестве основы для определения так называемой "типизированной таблицы".

из своих атриоутов, атриоуты не характеризуются упорядочением слева направо, каждое подмножество кортежа является кортежем, а каждое подмножество заголовка — заголовком. В ней также описаны **генератор типа TUPLE, селекторы** кортежей, операции **присваивании** и **проверки кортежей на равенство,** а также другие общие операции с кортежами.

Затем изложение перешло к **отношениям** (точнее, под этим подразумевались значения отношений). Было дано точное определение и указано, что каждое подмножество тела отношения является телом и (как и в случае кортежей) каждое подмножество заголовка отношения является заголовком. В этой связи были описаны такие понятия, как **генератор типа RELATION** и **селекторы** отношения, и отмечено, что в общем атрибуты любого типа отношения могут принадлежать **к любому типу.**

Примечание. Целесообразно кратко остановиться на этом последнем пункте, поскольку в области информационной индустрии, связанной с использованием баз данных, его сопровождает слишком много путаницы. Читателю, вероятно, приходилось часто сталкиваться с такими утверждениями, что реляционные атрибуты могут иметь только очень простые типы (числа, строки и т.д.). Но истина заключается в том, что в реляционной модели нет абсолютно ничего, что могло бы свидетельствовать в пользу таких утверждений. В действительности, как было отмечено в главе 5, типы могут быть сколь угодно простыми или сложными и поэтому вполне допустимо применение таких атрибутов, значениями которых являются числа, строки, даты, отметки времени, звукозаписи, географические карты, видеозаписи, геометрические точки и т.д.

Приведенное выше сообщение является настолько важным (и настолько неправильно трактуемым в широких кругах), что повторим его еще раз в другой формулировке.

Вопрос о том, какие типы данных являются поддерживаемыми, ортогонален вопросу поддержки реляционной модели.

Вернемся к данному резюме. Теперь необходимо рассмотреть определенные свойства, которым удовлетворяют все отношения.

1. Они всегда нормализованы.
2. Они не характеризуются упорядочением своих атрибутов слева направо.
3. Они не характеризуются упорядочением своих кортежей сверху вниз.
4. Они никогда не содержат каких-либо дубликатов кортежей.

В этой главе были также указаны некоторые основные различия между **отношениями** и **таблицами;** в ней описаны **атрибуты со значениями в виде отношений** и кратко рассмотрены отношения **TABLE_DEE** и **TABLE_DUM,** которые являются единственными возможными отношениями вообще без атрибутов. Кроме того, достаточно подробно были представлены реляционные операции сравнения и приведен краткий обзор некоторых других операций с отношениями (включая, в частности,**ORDER BY).**

Кстати, если речь идет об операциях с отношениями, то читатель мог заметить, что в главе 5 в определенной степени обсуждался вопрос об определяемых пользователем операторах для скалярных типов, но этого не было сделано для типов отношений. Причина состоит в том, что большинство необходимых реляционных операций (сокращение, проекция, соединение, реляционные сравнения и т.д.) фактически встроены в саму реляционную модель и не требуют каких-либо "пользовательских определений". (Мало того,

эти операции являются *универсальными* в том смысле, что они, неформально выражаясь, применимы к отношениям любых типов.) Тем не менее, нет никаких причин, препятствующих тому, чтобы эти встроенные операции были дополнены множеством операций, определяемых пользователем, если в системе предусмотрены средства для их определения.

Напомним, что заголовок любого конкретного отношения может рассматриваться как предикат, а кортежи этого отношения — как истинные высказывания, полученные на основе этого предиката путем подстановки значений фактических параметров соответствующих типов вместо формальных параметров предиката.

После этого мы перешли к рассмотрению базовых переменных отношения и отметили, что переменные отношения, как и сами отношения, имеют предикаты. Согласно предположению о замкнутости мира, можно считать, что если действительный по всем иным признакам кортеж не присутствует в теле переменной отношения, то соответствующее ему высказывание является ложным.

Затем были более подробно описаны реляционные операции присваивания (и их сокращения **INSERT,** **DELETE** и **UPDATE).** Кроме того, был подчеркнут тот пункт, что реляционные операции присваивания являются операциями уровня множества, а также отмечено, что фактически выражения "обновление кортежей" или "обновление атрибутов" являются неправильными.

Наконец, были кратко описаны аналоги перечисленных выше понятий в языке SQL, если есть такие аналоги. Таблица SQL — это не множество кортежей, а мультимножество строк (кроме того, в языке SQL для указания на значение таблицы и переменную таблицы используется один и тот же термин — *таблица).* Определения базовых таблиц можно "изменить" с помощью операции **ALTER TABLE.** Кроме того, таблицы SQL могут быть определены в терминах структурированных типов; такая возможность рассматривается более подробно далее в этой книге (в главе 26).

УПРАЖНЕНИЯ

6.1.Дайте определение термина *кардинальность.*

6.2.Определите так можно точнее термины *кортеж* и *отношение.*

6.3.Сформулируйте с наибольшей возможной точностью, что означают следующие выражения:

а) равны два кортежа;

б) равны два типа кортежа;

в) равны два отношения;

г) равны два типа отношения.

6.4.Для базы данных поставщиков, проектов и деталей, показанных на рис. 4.5 (см. стр. 154), запишите на языке Tutorial D следующее:

а) множество предикатов и

б) множество определений переменной отношения.

6.5. Запишите выражения с вызовами селектора кортежа для типичного кортежа каждой из переменных отношения в базе данных поставщиков, проектов и деталей.

6.6.Определите локальную переменную кортежа, в которую можно выполнить выборку отдельного кортежа из переменной отношения поставок, которая определена в базе данных поставщиков, проектов и деталей.

6.7.Что означают следующие выражения на языке Tutorial D?

а) `RELATION { S# S#, P# P#, J# J#, QTY QTY } { }`

б)
```
RELATION { TUPLE { S# S#('S1'), P# P#('P1'),
   J# J#('J1'), QTY QTY (200) } }
```

в) `RELATION { TUPLE { } }`

г) `RELATION { } { TUPLE { } }`

д) `RELATION { } { }`

6.8.Дайте определение *термина первая нормальная форма.*

6.9.Перечислите все известные вам различия между отношениями и таблицами.

6.10.Подготовьте самостоятельно следующие примеры отношений:

а) с одним атрибутом, значением которого является отношение и

б) с двумя такими атрибутами.

Кроме того, приведите еще два примера отношений, которые представляют ту самую информацию, что и упомянутые выше отношения, но не включают атрибуты, значением которых является отношение.

6.11.Запишите выражение, которое возвращает TRUE, если текущее значение переменной отношения деталей p является неопределенным, и FALSE в противном случае. Не используйте сокращение IS_EMPTY.

6.12.Почему ORDER BY считается довольно необычным оператором?

6.13.Сформулируйте **предположение о замкнутости мира.**

6.14.Иногда приходится слышать утверждение, что переменная отношения в действительности представляет собой просто обычный компьютерный файл, "кортежами" которого являются записи, а "атрибутами" — поля. Выскажите свое мнение по этому поводу.

6.15.Приведите формулировки на языке Tutorial D следующих операций обновления базы данных поставщиков, проектов и деталей.

a) Вставить данные о новой поставке с номером поставщика si, номером детали P1, номером проекта J2, количеством 500.

b) Вставить в таблицу s данные о новом поставщике, с номером S10 и именем Smith, который находится в городе New York; статус еще не известен.

c) Удалить все детали синего цвета.

г) Удалить все проекты, для которых нет поставок.

Д) Сменить цвет всех красных деталей на оранжевый.

е) Заменить все вхождения номера поставщика S1 вхождениями номера поставщика S9.

6.16. Как было указано выше, операции определения данных вызывают внесение обновлений в каталог. Но каталог — это лишь коллекция переменных отношения, как и остальная часть базы данных. Почему же нельзя использовать для обновления каталога обычные операции INSERT, DELETE и UPDATE? Выскажите свое мнение по этому вопросу.

6.17. В настоящей главе было указано, что в общем в качестве основы для определения реляционных атрибутов может использоваться любой тип. Но в этом утверждении не зря было употреблено выражение "в общем". Известны ли вам какие-либо исключения из этого общего правила?

6.18. Дайте определения терминов *столбец, поле* и *атрибут,* применяемых в спецификации языка SQL?

6.19. *(Модифицированная версия упр. 5.23.)* Рассмотрите определения типа POINT и таблицы POINTS на языке SQL, которые приведены в подразделе "Структурированные типы" раздела 6.6. Тип POINT имеет представление, в котором используются декартовы координаты X and Y. Что произойдет, если этот тип будет заменен пересмотренным типом POINT с представлением, в котором используются полярные координаты R and 8?

СПИСОК ЛИТЕРАТУРЫ

Большинство перечисленных ниже ссылок касается всех аспектов реляционной модели, а не только отношений как таковых.

6.1. Codd E.F. A Relational Model of Data for Large Shared Data Banks // CACM. — June 1970. — 13, № 6. (Переиздано: Milestones of Research. — Selected Papers 1958-1982 (CACM 25th Anniversary Issue) // CACM. — January 1983. — 26, № 1. См. также более раннее издание: Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks // IBM Research Report RJ599. — August 1969. — № 19.

Примечание. Это раннее издание является первой публикацией Кодда (Codd) о реляционной модели.)

С этой статьи все и началось. И хотя прошло более 30 лет, она остается актуальной и стоит того, чтобы ее по-прежнему перечитывали. Безусловно, многие идеи со времени ее публикации были в определенной степени уточнены, но по своей природе это были эволюционные, а не революционные изменения. Кроме того, в статье есть идеи, все следствия которых до сих пор еще полностью не исследованы. Необходимо сделать несколько замечаний относительно терминологии. В своей статье вместо термина "переменная отношения" Кодд использует термин "отношение, изменяющееся во времени". Однако этот термин не очень удачен. Во-первых, отношения как таковые являются *значениями* и просто не могут изменяться во времени

(в математике не известны отношения, принимающие разные значения в разное время). Во-вторых, если на каком-либо языке программирования мы пишем

```
DECLARE N INTEGER ;
```

то мы не называем N целым числом, изменяющимся во времени; мы называем ее целочисленной переменной. В этой книге используется собственный термин "переменная отношения" и не используется термин "изменяющееся во времени", однако необходимо просто помнить о существовании этого устаревшего термина.

6.2. Codd E.F. The Relational Model for Database Management Version 2. — Reading, Mass.: Addison-Wesley, 1990.

Длительное время в конце 1980-х годов Кодд пересматривал и расширял свою первоначальную модель (которую он переименовал в реляционную модель первой версии — "the Relational Model Version 1" или RM/V1), в результате чего появилась эта книга. В ней описывается так называемая реляционная модель второй версии — "the Relational Model Version 2" или RM/V2. Основное различие между версиями состоит в следующем. Первая версия создавалась как абстрактный план для определенных аспектов общей проблематики баз данных (по большей части, ее наиболее фундаментальных аспектов), а вторая — как абстрактный план для *всей системы*. Поэтому, в то время как первая версия разделена только на 3 части (структура, поддержка целостности и манипулирование данными), модель RM/V2 содержит 18 частей, которые включают не только 3 исходные части, что само собой разумеется, но и части, касающиеся каталогов, прав доступа, именования, распределенных баз данных и других аспектов управления базами данных. Для удобства ссылок ниже приводится полный перечень этих 18 частей.

A Права доступа
B Основные операторы
C Каталог
D Принципы разработки СУБД
E Команды администратора базы данных
F Функции
I Поддержка целостности данных
J Индикаторы
L Принципы разработки языка
M Манипулирование данными
N Именование
P Защита
Q Кванторы
S Структуры
T Типы данных
V Представления
X Распределенная база данных
Z Дополнительные операторы

Тем не менее, идеи, изложенные в этой книге, не нашли широкого распространения. (См., в частности, статьи [6.7] и [6.8].) Приведем небольшой комментарий к данной статье. Как уже говорилось в главе 5, домены (т.е. типы) служат ограничениями для операций сравнения. Например, для базы данных поставщиков и деталей сравнение S. S# = P. P# будет недопустимым, поскольку сравниваемые величины принадлежат к разным типам. Следовательно, связать поставщиков и детали путем сопоставления номеров поставщиков и деталей не удастся. Поэтому Кодд предложил свой вариант некоторых операторов реляционной алгебры — так называемые операторы DCO **(Domain Check Override** — с отменой проверки домена). Они выполняются, даже если при этом сравниваются значения разных типов. Так,

например, версия DCO оператора соединения, рассмотренного выше, позволяет связать поставщиков и детали, несмотря на то, что атрибуты S. S# и P. P# принадлежат к разным типам (предполагается, что соединение производится путем сопоставления не *типов,* а их *представлений).* Однако именно в этом и состоит проблема. *Вся идея DCO основана на путанице между типами и их представлениями.* Считая домены тем, чем они являются на самом деле (т.е. типами), со всеми вытекающими отсюда последствиями, мы получаем возможность контроля доменов, *а также* обеспечиваем своего рода совместимость с DCO. В частности, следующее выражение служит примером сравнения номера поставщика с номером детали на уровне допустимых представлений.

```
THE_S#  (  S#  )  =  THE_P#  (  P#  )
```

Здесь обе части равенства принадлежат к типу CHAR. Таким образом, мы утверждаем, что предложенный в главе 5 механизм обеспечивает нас всеми необходимыми средствами очевидным и систематизированным образом (т.е. непроизвольным образом), к тому же полностью ортогональным. В частности, отпадает необходимость загромождать реляционную модель новыми операторами наподобие соединения DCO и т.д.

6.3. Darwen H. The Duplicity of Duplicate Rows // C. J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

Эта статья была написана для подтверждения доводов, уже представленных в [6.6] (первая версия), в пользу известного требования реляционной модели, согласно которому в таблицах не должно содержаться одинаковых строк. В статье не только представлены новые версии этих доводов, но и выдвинуты дополнительные. Основной вопрос состоит в следующем: для того чтобы конструктивно обсуждать совпадение двух объектов, необходим *критерий равенства* (называемый в этой статье *критерием идентичности)* для рассматриваемого класса объектов (иначе говоря, определение того, что означает понятие "быть одним и тем же" для двух объектов, будь это строки в таблице или что-нибудь еще).

6.4. Darwen H. Relation-Valued Attributes // C. J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.'

6.5. Darwen H. The Nullologist in Relationland // C J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

Нуллология— это согласно Дарвену "наука вообще ни о чем" или, иначе говоря, наука о пустом множестве. (Пустые множества не имеют ничего общего с "пустыми", неопределенными NULL-значениями в стиле языка SQL!) Множества в реляционной теории являются вездесущими, поэтому вопрос о том, что произойдет, если подобное множество окажется пустым, рассматривается не ради простого любопытства. На самом деле в некоторых случаях пустые множества играют фундаментальную роль. ***Примечание.*** Что касается темы настоящей главы, то к данной статье непосредственное отношение имеют разделы 2 ("Таблицы без строк") и 3 ("Таблицы без столбцов").

6.6. Date C.J. Double Trouble, Double Trouble // http://www.dbdebunk.com, April 2002. Первоначальная версия этой статьи: Date C.J. Why Duplicate Rows Are Prohibited // Relational Database Writings 1985-1989.— Reading, Mass.: Addison-Wesley, 1990.

Представлены многочисленные доводы (с примерами) в поддержку требования реляционной модели в части того, чтобы в таблицах не было дублирующихся строк. В частности, в статье показано, что дублирующиеся строки являются главным *препятствием для оптимизации* (см. главу 18). См. также [6.3].

6.7. Date C.J. Notes Toward a Reconstituted Definition of the Relational Model Version 1 (RM/V1) // C J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

Приводится критика и резюме к реляционной модели Кодда RM/V1 [6.2] и дается альтернативное определение. Подразумевается, что крайне важно получить верную "версию 1", прежде чем обсуждать переход к какой-то "версии 2". *Примечание.* Версия реляционной модели, описанная в настоящей книге, основана на "пересмотренной" версии, кратко описанной в данной статье (а затем уточненной и описанной в [3.3]).

6.8. Date C.J. A Critical Review of Relational Model Version 2 (RM/V2) // C J. Date and Hugh Darwen. Relational Database Writings 1989—1991.— Reading, Mass.: Addison-Wesley, 1992.

Приводятся критика и резюме к реляционной модели Кодда RM/V2 [6.2].

6.9. Date C.J. The Database Relational Model: A Retrospective Review and Analysis // Reading, Mass.: Addison-Wesley. 2001.

В этой небольшой книге (объемом 160 страниц) приведен подробный и беспристрастный ретроспективный обзор, а также дана оценка вклада Кодда в реляционную модель на основании его публикаций в 1970-х годах. В частности, в ней подробно исследованы следующие статьи (а также несколько других).

- *Derivability, Redundancy, and Consistency of Relations Stored in Large Data Banks* (первая версия статьи [6.1]).
- *A Relational Model of Data for Large Shared Data Banks* [6.1].
- *Relational Completeness of Data Base Sublanguages* [7.1].
- *A Data Base Sublanguage Founded on the Relational Calculus* [8.1].
- *Further Normalization of the Data Base Relational Model* [11.6].
- *Extending the Relational Database Model to Capture More Meaning [* 14.7].
- *Interactive Support for Nonprogrammers: The Relational and Network Approaches* [26.12].

6.10. Roth M.A., Korth H.F., Silberschatz A. Extended Algebra and Calculus for Nested Relational Databases //ACM TODS 13. — December 1988. — № 4.

В течение многих лет исследователи предлагали использовать поддержку атрибутов со значениями в виде отношений; эта статья как раз и касается данной темы. Обычно подобные предложения рассматриваются под названием "отношений

NF^2" (произносится NF квадрат), служат сокращением для NFNF и обозначают нормальную форму, отличную от первой (NFNF— Non First Normal Form). Но существует по меньшей мере два важных различия между подобными предложениями и поддержкой атрибутов со значением в виде отношений, как описано в данной главе.

- Во-первых, сторонники подхода, основанного на использовании отношения NF^2, предполагают, что атрибуты со значением в виде отношений в реляционной модели запрещены и поэтому рекламируют свои предложения как "расширения" данной модели (в этой связи заслуживает внимания даже само название статьи [6.10]).
- Во-вторых, сторонники подхода, основанного на использовании отношений NF^2, действительно правы — они расширяют реляционную модель. Например, Рот (Roth) со своими сторонниками предложил расширенную форму объединения, в которой (используя нашу терминологию) рекурсивно разгруппировываются оба операнда до тех пор, пока в них вообще не остается ни одного атрибута со значением в виде отношения (эта операция выполняется прямо или косвенно), осуществляется обычная операция объединения по этим разгруппированным операндам, и наконец, полученный результат снова рекурсивно группируется. А расширением модели является именно то, что при этом предусмотрено применение рекурсии. Таким образом, хотя любое конкретное расширенное объединение является сокращением для некоторой конкретной комбинации существующих реляционных операций, в общем, нельзя утверждать, что это расширенное объединение служит просто сокращением для некоторой комбинации существующих операций.

ГЛАВА7

Реляционная алгебра

7.1. ВВЕДЕНИЕ

Реляционная алгебра — это коллекция операций, которые принимают отношения в качестве операндов и возвращают отношение в качестве результата. Первая версия этой алгебры была определена Коддом в [5.1] и [7.1]; основным источником сведений об этой "оригинальной" алгебре принято считать [7.1]. Эта "оригинальная" алгебра включала восемь операций, которые подразделялись на описанные ниже две группы с четырьмя операциями каждая.

1. Традиционные операции с множествами — *объединение, пересечение, разность* и *де*
   картово произведение (все они были немного модифицированы с учетом того факта,
   что их операндами являются именно отношения, а не произвольные множества).
2. Специальные реляционные операции, такие как *сокращение* (известное также под
   названием *выборки), проекция, соединение* и *деление.*

На рис. 7.1 приведена неформальная иллюстрация к описанию принципов действия этих операций.

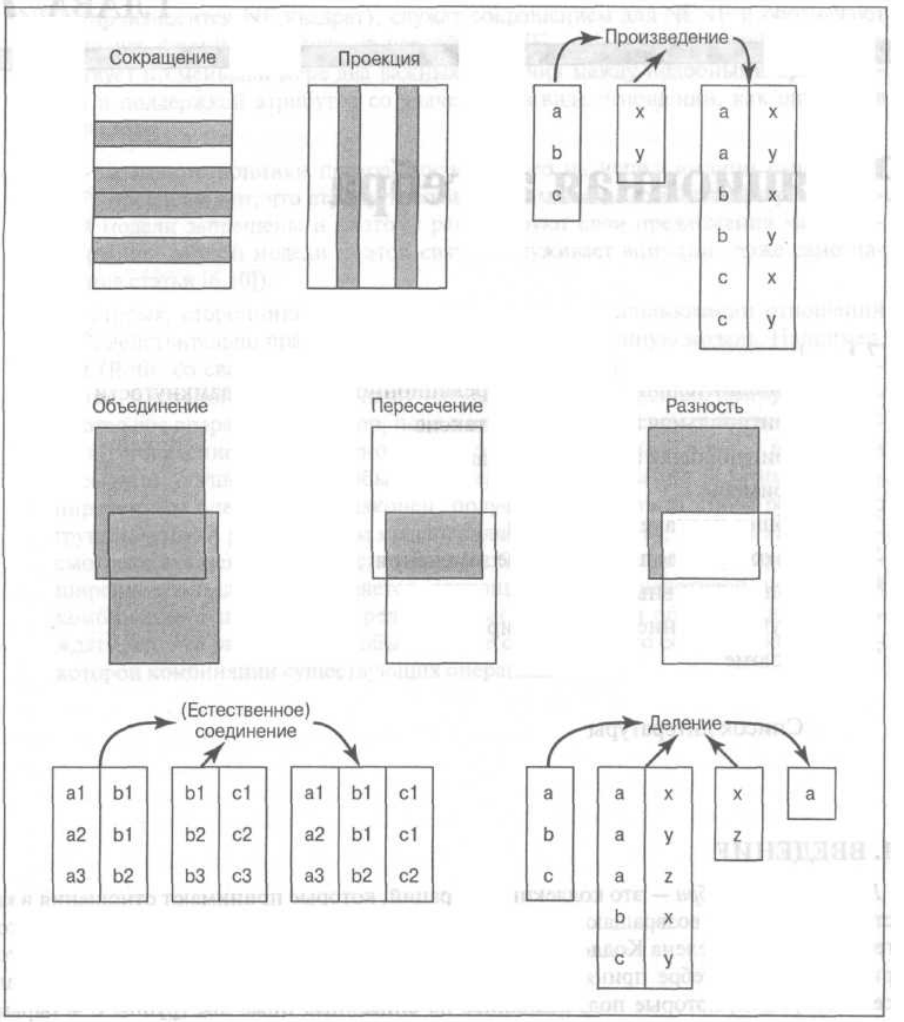


Рис. 7.1. Восемь первоначальных операций (краткая иллюстрация)

Прежде всего, отметим, что Кодд, определяя только эти восемь операций, руководствовался весьма конкретным замыслом, как будет описано в следующей главе. Но на этих восьми операциях все не заканчивается; в действительности, может быть определено *любое количество* операций, которые удовлетворяют простому требованию, чтобы "на входе и выходе были отношения", и такие дополнительные операции были фактически определены многими разными авторами. В данной главе вначале рассматриваются оригинальные восемь операций (не совсем в том виде, в каком они были определены первоначально,

а какими стали со временем) и эти операции используются как основа для обсуждения различных алгебраических идей; затем мы перейдем к описанию многих других полезных операций, которые впоследствии были добавлены к оригинальному набору.

Но прежде чем перейти к подробному обсуждению алгебры, необходимо сделать еще несколько предварительных замечаний, которые приведены ниже.

- Прежде всего, выражаясь неформально, рассматриваемые операции применимы ко всем отношениям; фактически они являются в полном смысле слова *универсальными* операциями, которые связаны с генератором типа RELATION И поэтому могут применяться к любому конкретному типу отношения, полученному путем вызова этого генератора типа.
- Кроме того, почти все операции, которые будут рассматриваться в этой главе, в действительности представляют собой просто сокращенную запись! Дополнительная информация по этой важной теме приведена в разделе 7.10.
- Далее отметим, что все эти операции предназначены *только для чтения* (т.е. они "читают", но не обновляют свои операнды). Таким образом, они применяются именно к *значениям* (разумеется, к значениям отношений) и поэтому, естественно, к тем значениям отношений, которые оказались во время их применения текущими значениями переменных отношения.
- Наконец, из предыдущего замечания следует, что имеет смысл, например, рассуждать о "проекции переменной отношения R по атрибуту A"; это выражение является определением отношения, полученного путем выполнения операции проекции по атрибуту A над текущим значением указанной переменной отношения R. Но иногда удобнее использовать выражения наподобие "проекции переменной отношения R по атрибуту A" немного в ином смысле. Например, предположим, что определено представление SC переменной отношения поставщиков S, которое состоит только из атрибутов S# и CITY этой переменной отношения. В таком случае можно применять неформальную, но очень удобную формулировку, что переменная отношения SC является "проекцией переменной отношения S по атрибутам S# и CITY"; в более строгом смысле это выражение означает, что значение SC в любой указанный момент времени является проекцией значения переменной отношения S по атрибутам S# и CITY В ТОТ же момент времени. Поэтому в данном смысле можно вести речь о *проекциях переменных отношения* как таковых, а не просто о *проекциях текущих значений переменных отношения.* Автор надеется, что такое двухцелевое использование им терминологии не вызовет каких-либо недоразумений.

План этой главы состоит в следующем. За этим вступительным разделом повторно рассматривается и излагается намного более подробно тема реляционного свойства замкнутости (раздел 7.2). Затем в разделах 7.3 и 7.4 подробно описаны оригинальные восемь операций Кодда, а в разделе 7.5 приведены примеры того, как могут использоваться эти операции для формулировки запросов. После этого в разделе 7.6 обсуждается более общий вопрос о том, для чего предназначена эта алгебра. В разделе 7.7 автор поднимает еще целый ряд различных тем. Далее, в разделе 7.8 приведено описание некоторых полезных дополнений к оригинальной алгебре Кодда, включая, в частности, такие важные

операции, как EXTEND И SUMMARIZE. В разделе 7.9 рассматриваются операции для прямого и обратного преобразования отношений с атрибутами в виде отношений в отношения без таких атрибутов. Наконец, в разделе 7.10 приведено краткое резюме данной главы.

Примечание. Отложим обсуждение соответствующих средств SQL до главы 8 по причинам, которые будут описаны в настоящей главе.

7.2. ДОПОЛНИТЕЛЬНЫЕ СВЕДЕНИЯ О РЕЛЯЦИОННОМ СВОЙСТВЕ ЗАМКНУТОСТИ

Как было описано в главе 3, тот факт, что результатом любой конкретной реляционной операции с отношениями является другое отношение, получил название *реляционного свойства замкнутости.* Из свойства замкнутости следует, что могут применяться вложенные реляционные выражения; таковыми являются реляционные выражения, операнды которых, в свою очередь, представлены реляционными выражениями произвольной сложности. (В этом проявляется очевидная аналогия между возможностью применять вложенные реляционные выражения в реляционной алгебре и возможностью создавать вложенные арифметические выражения в обычной арифметике; в действительности, тот факт, что отношения в алгебре подчиняются свойству замкнутости, важен именно по тем же причинам, по каким имеет важное значение то, что числа в обычной арифметике также подчиняются свойству замкнутости.)

Теперь отметим, что при обсуждении свойства замкнутости в главе 3 был намеренно пропущен один очень важный пункт. Напомним, что каждое отношение состоит из двух частей — заголовка и тела; выражаясь неформально, заголовок — это атрибуты, а тело — кортежи. Структура заголовка для базового отношения (где базовым отношением, как указано в главе 5, является значение базовой переменной отношения), безусловно, известна в системе, поскольку она указана в составе определения соответствующей базовой переменной отношения. А как насчет производных отношений? Например, рассмотрим следующее выражение.

```
S JOIN P
```

Это выражение представляет собой соединение отношений поставщиков и деталей по совпадающим данным о городах— по атрибуту CITY, который является единственным общим атрибутом в двух отношениях. Нам известно, как будет выглядеть тело результирующего отношения, а что можно сказать о заголовке? Из свойства замкнутости следует, что это отношение должно иметь заголовок, а системе необходимо иметь сведения о структуре этого заголовка (фактически, как указано ниже, пользователь также должен иметь эти сведения). Иными словами, результирующее отношение (безусловно!) должно принадлежать к некоторому вполне определенному типу отношения. Поэтому для полной поддержки свойства замкнутости следует определять реляционные операции таким образом, чтобы можно было гарантировать формирование любой операцией такого результата, который имеет допустимый тип отношения, в частности, имеет допустимые имена атрибутов. (Кстати, следует отметить, что именно этот аспект алгебры часто не получает достаточного освещения в литературе, а также, к сожалению, в языке SQL и поэтому в продуктах SQL; одним из заметных исключений является описание этой темы, приведенное в [7.2] и [7.10]. Трактовка алгебры, представленная в данной главе, была подготовлена под очень большим влиянием этих двух работ.)

Одна из причин, по которым требуется, чтобы каждое результирующее отношение имело допустимые имена атрибутов, состоит в том, что благодаря этому появляется возможность ссылаться на данные атрибуты в последующих операциях, в частности, в операциях, вызываемых на выполнение в любом другом месте всего вложенного выражения. Например, нет резонных оснований даже записывать выражение, подобное следующему, если не известно, что результат вычисления выражения S JOIN P имеет атрибут с именем CITY.

```
( S JOIN P ) WHERE CITY = 'Athens'
```

Поэтому необходимо, чтобы в алгебру было встроено множество **правил вывода типа отношения,** такое, что если известен тип (типы) входного отношения (отношений) для любой конкретной реляционной операции, то на основании этой операции можно определить, к какому типу будет относиться ее результат. Из этого следует, что при наличии подобных правил любое произвольное реляционное выражение, независимо от его сложности, будет вырабатывать результат, который также имеет вполне определенный тип и, в частности, вполне определенное множество имен атрибутов.

В качестве предварительного шага к достижению этой цели введем новый оператор, RENAME, назначение которого (неформально выражаясь) состоит в *переименовании* атрибутов указанного отношения. Точнее, оператор RENAME принимает заданное отношение и возвращает другое, идентичное заданному, за исключением того, что один из его атрибутов имеет другое имя. (Заданное отношение должно быть указано с использованием некоторого реляционного выражения, которое может включать иные реляционные операции.) Например, можно записать следующее выражение.

```
S RENAME CITY AS SCITY
```

Это выражение (следует подчеркнуть что это — *выражение, а.* не "команда" или оператор и поэтому может вкладываться в другие выражения) приводит к получению отношения с тем же заголовком и телом, что и отношение, которое является текущим значением переменной отношения S, за исключением того, что атрибут с указанием города в нем называется SCITY, а не CITY, как показано ниже.

S#	SNAME	STATUS	SCITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Примечание. Следует отметить тот важный факт, что это выражение с оператором RENAME не изменяет переменную отношения поставщиков в базе данных! Оно является просто выражением (точно так же, как, например, S JOIN SP представляет собой только выражение) и, подобно любому другому выражению, определяет лишь некоторое значение (которое в данном конкретном случае оказалось во многом подобным текущему значению переменной отношения поставщиков).

Ниже приведен еще один пример (в котором на сей раз применяется *множественное переименование).*

Следует отметить, что благодаря наличию оператора RENAME в реляционной алгебре,

```
P RENAME ( PNAME AS PN, WEIGHT AS WT )
```

Это выражение является сокращением для следующего выражения.

```
( P RENAME PNAME AS PN ) RENAME WEIGHT AS WT
```

Результат его применения может выглядеть так, как показано ниже.

P#	PN	COLOR	WT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

в отличие от языка SQL, нет необходимости применять имена атрибутов, уточненные с помощью точки, такие как S. S# (и фактически такие имена не поддерживаются).

7.3. ОРИГИНАЛЬНАЯ АЛГЕБРА- СИНТАКСИС

В данном разделе приведен конкретный синтаксис выражений реляционной алгебры, в которых используются оригинальные восемь операций наряду с операцией переименования RENAME. Этот синтаксис основан на применении языка Tutorial D. Он показан здесь в основном для использования в последующем изложении. Включено также несколько примечаний о семантике.

Примечание. В большинстве работ по базам данных для обозначения реляционных операторов применяются математические символы или греческие буквы: σ обозначает сокращение ("выборку"), я — проекцию, п— пересечение, х ("галстук-бабочка") — соединение и т.д. Как показывает приведенный здесь материал, автор предпочитает использовать ключевые слова наподобие JOIN и WHERE. Ключевые слова требуют больше места, но автор считает, что наряду с этим они облегчают восприятие материала.

```
<relation exp>
    ::= RELATION { <tuple exp
      commalist> } | <relvar name> |
      <relation op inv> j <with exp>
      | <introduced name> | (
      <relation exp> )
```

Здесь <*relation exp*> — это выражение, которое обозначает отношение (т.е. значение отношения). Первый формат— это вызов *селектора отношения* (см. главу 6); в этом разделе синтаксис выражения кортежа < *tuple exp*> подробно не рассматривается, поскольку для получения общего представления о нем достаточно ознакомиться с несколькими примерами. Форматы с применением имени отношения <*relvar name*> и

выражения отношения *{<relation exp>)* говорят сами за себя; остальные форматы описаны ниже.

```
<relation op inv>
    ::= <project> | <nonproject>
```

Вызов реляционного оператора *<relation op inv>* представляет собой либо оператор проекции, *<project>,* либо оператор, отличный от проекции, *<nonproject>*.

Примечание. Эти два случая различаются в синтаксисе просто для учета приоритета операторов (оператору проекции удобно назначить более высокий приоритет).

```
<project>
    ::= <relation exp>
                { [ ALL BUT ] <attribute name comnalist> }
```

Здесь выражение *<relation exp>* не должно принадлежать к типу *<nonproject>*.

```
<nonproject>
    ::= <rename> | <union> \ <intersect> \ <minus> \
    <times>
      | <where> | <join> \
<divide> <rename>
    ::- <relation exp> RENAME ( <renaming cowmalist> )
```

Здесь *<relation exp>* также не должно принадлежать к типу *<nonproject>*. Отдельные операции переименования *<renaming>* выполняются в той последовательности, в какой они записаны (для ознакомления с синтаксисом переименования *<renaming>* просмотрите примеры, приведенные в предыдущем разделе). Если разделенный запятыми список *commalist* содержит только один оператор *<renaming>,* круглые скобки могут быть опущены.

```
<union>
    ::= <relation exp> UNION <relation exp>
```

Здесь выражения *<relation exp>* также не должны принадлежать к типу *<nonproject>,* за исключением того случая, когда одно из них или оба относятся к другому выражению *<un±on>*.

```
<intersect>
    ::= <relation exp> INTERSECT <relation exp>
```

Здесь выражения *<relation exp>* также не должны принадлежать к типу *<nonproject>,* за исключением того случая, когда одно из них или оба относятся к другому выражению *<intersect>*.

```
<minus>
    ::= <relation exp> MINUS <relation exp>
```

Здесь выражения *<relation exp>* также не должны принадлежать к типу *<nonproject>*.

```
<times>
    ::= <relation exp> TIMES <relation exp>
```

Здесь выражения <relation exp> также не должны принадлежать к типу <nonproject>, за исключением того случая, когда одно из них или оба относятся к другому выражению < times>.

```
<where>
    ::= <relation exp> WHERE <bool exp>
```

Здесь выражение <relation exp> не должно принадлежать к типу <nonproject>. Выражение <Ъоо1 exp> может включать ссылки на атрибуты отношения, обозначенного как <relation exp>, с очевидной семантикой.

```
<join>
    ::- <relation exp> JOIN <relation exp>
```

Выражения <relation exp> не должны принадлежать к типу <nonproject>, если не считать того, что одно или оба из них могут представлять собой еще одно выражение типа <joi.n>.

```
<divide>
    ::= <relation exp> DIVIDEBY <relation exp> PER <per>
```

Выражение <relation exp> не должно принадлежать к типу<nonproject>.

```
<per>
    ::=  <relation exp> | ( <relation exp>, <relation exp> )
```

Выражение <relation exp> не должно принадлежать к типу<nonproject>.

```
<with exp>
    ::= WITH <name intro commalist> : <exp> ;
```

Выражения с ключевым словом WITH, <wi th exp>, которые в основном интересуют нас в этой книге, являются именно реляционными выражениями и поэтому они обсуждаются в данной главе. Но поддерживаются также скалярные выражения <wi th exp> и выражения с кортежами; фактически любая конкретная конструкция <wi th exp> представляет собой <relation exp>, <tuple exp> или <scalar exp> в соответствии с тем, относится ли, в свою очередь, выражение <exp>, стоящее после двоеточия, к типу <relation exp>, < tuple exp> или <scalar exp>. **Во всех случаях** отдельные операторы введения имен <name intro> выполняются в той последовательности, в какой они записаны, и семантика выражения <with exp> определяется как совпадающая с той версией <exp>, где каждое вхождение каждого введенного имени заменяется ссылкой на переменную, значение которой является результатом вычисления соответствующего выражения.

Примечание. WITH в действительности не является оператором реляционной алгебры как таковым; это ключевое слово служит просто средством формирования выражений, которые без его использования становились бы довольно сложными (особенно если они включают общие подвыражения). Несколько примеров приведено в разделе 7.5.

```
<name intrc»
    ::= <exp> AS <introduced name>
```

Здесь введенное имя <*introduced name*> может использоваться в содержащем его выражении <*wi th exp*> везде, где допускается применение выражения <exp> (в случае необходимости заключенного в круглые скобки).

7.4. ОРИГИНАЛЬНАЯ АЛГЕБРА - СЕМАНТИКА

Объединение

В математике объединение двух множеств представляет собой множество всех элементов, принадлежащих либо к одному из них, либо к обоим заданным множествам. Поскольку любое отношение представляет собой (или, скорее, содержит) множество (а именно множество кортежей), оно, безусловно, позволяет формировать объединение двух таких множеств; результатом является множество, состоящее из всех кортежей, присутствующих либо в одном, либо в обоих из заданных отношений. Например, объединение множества кортежей поставщиков, которые в настоящее время присутствуют в переменной отношения S, и множества кортежей деталей, присутствующих в настоящее время в переменной отношения P, безусловно, представляет собой множество.

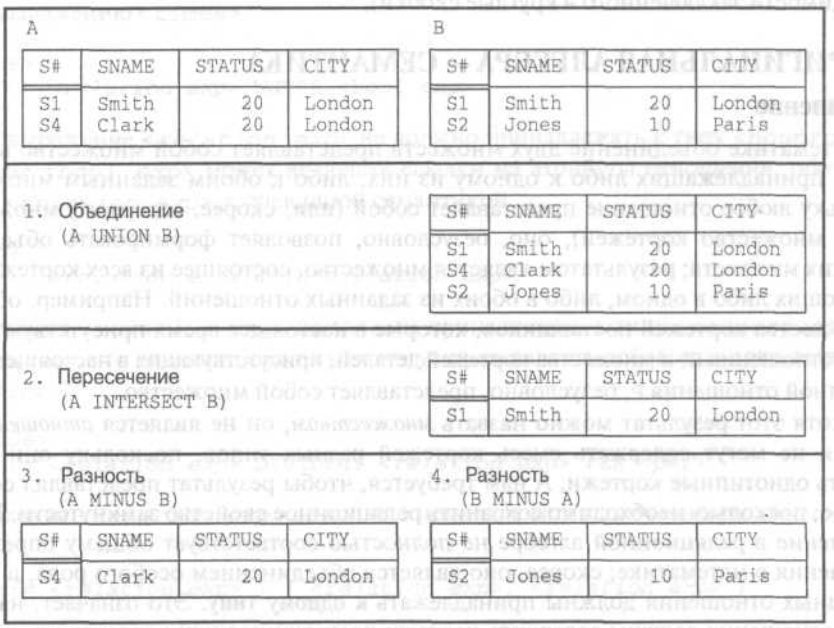
Но хотя этот результат можно назвать *множеством,* он не является *отношением;* отношения не могут содержать смесь кортежей разных типов, поскольку они должны включать однотипные кортежи. А нам требуется, чтобы результат представлял собой отношение, поскольку необходимо сохранить реляционное свойство замкнутости. Поэтому объединение в реляционной алгебре не полностью соответствует общему определению объединения в математике; скорее, оно является объединением особого рода, в котором два входных отношения должны принадлежать **к одному типу.** Это означает, например, что оба отношения должны содержать кортежи поставщиков или кортежи деталей, но не смесь кортежей этих двух типов. Если же два отношения принадлежат к одному и тому же типу, то может быть получено их объединение, а результат также будет представлять собой отношение того же типа; иными словами, сохраняется свойство замкнутости.

Примечание. По традиции в большинстве литературных источников по базам данных (включая первые издания этой книги) для обозначения того требования, что оба отношения должны принадлежать к одному и тому же типу, применялся термин *совместимость по объединению* (union compatibility). Но этот термин нельзя назвать очень удачным по целому ряду причин; наиболее важной из них является то, что указанное требование не распространяется только на объединение.

Поэтому определение операции реляционного объединения должно быть таким: если даны отношения a и b одного и того же типа, то **объединение** этих отношений a UNION b является отношением того же типа с телом, которое состоит из всех кортежей t, присутствующих в a или b или в обоих отношениях.

Пример. Предположим, что отношения A и B имеют вид, показанный на рис. 7.2 (оба они получены из текущего значения переменной отношения поставщиков S; вообще говоря, в A приведены данные о поставщиках из Лондона, а в в — данные о поставщиках, которые поставляют деталь P1). В таком случае в объединение A UNION в (см. рис. 7.2, *а)* входят поставщики, которые либо находятся в Лондоне, либо поставляют деталь P1, либо соответствуют обоим этим условиям. Обратите внимание на то, что в результате содержатся три кортежа, а не четыре; по определению отношения никогда не содержат дубликаты кортежей (поэтому, неформально выражаясь, из результатов операции объединения "устраняются дубликаты"). Кстати, следует отметить, что единственной не рассмотренной

нами операцией из первоначальных восьми, при использовании которой возникает необходимость удаления дубликатов, является проекция (как описано ниже в этом разделе).



S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London
S2	Jones	10	Paris

S#	SNAME	STATUS	CITY
S1	Smith	20	London

S#	SNAME	STATUS	CITY
S4	Clark	20	London

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris

Рис. 7.2. Примеры операций объединения, пересечения и разности

Между прочим, обратите внимание на то, что определение операции объединения базируется на понятии *равенства кортежей.* Приведем еще одно, но эквивалентное определение, в котором этот нюанс подчеркивается очень ясно (измененная формулировка отмечена курсивным шрифтом): если даны отношения a и b одного и того же типа, то объединение этих отношений a UNION b является отношением того же типа с телом, состоящим из всех кортежей t, таких, что t *равен некоторому кортежу* из a или b или из обоих отношений *{т.е. является его дубликатом).* Как вскоре станет очевидно, аналогичные замечания касаются непосредственно операций пересечения и разности.

Пересечение

Как и для объединения, и фактически по той же причине, для реляционной операции пересечения требуется, чтобы ее операнды принадлежали к одному и тому же типу. Если даны отношения a и b одного и того же типа, то **пересечением** этих отношений a INTERSECT b является отношение того же типа с телом, состоящим из всех кортежей t, таких, что t присутствует одновременно в a и b.

Пример. Снова предположим, что отношения A и B показаны на рис. 7.2. Тогда пересечение A INTERSECT B (рис. 7.2, *б)* включает всех поставщиков, которые находятся в Лондоне и поставляют деталь P1.

Разность

Как и для объединения и пересечения, для реляционной операции разности требуется, чтобы ее операнды принадлежали к одному и тому же типу. Если даны отношения a и b одного и того же типа, то **разностью** этих отношений a MINUS b (в указанном порядке), является отношение того же типа с телом, состоящим из всех кортежей t, таких, что t присутствует в a, но не в b.

Пример. Снова предположим, что отношения A и в показаны на рис. 7.2. Тогда результат операции разности A MINUS в (рис. 7.2, *в)* включает поставщиков, которые находятся в Лондоне и не поставляют деталь P1, а результат операции разности в MINUS A (рис. 7.2, *г)* включает поставщиков, которые поставляют деталь P1 и не находятся в Лондоне. Обратите внимание на то, что оператор MINUS характеризуется направленностью (некоммутативностью), так же, как вычитание в обычной арифметике (например, "5 - 2" и "2 - 5" не являются одним и тем же).

Произведение

В математике *декартовым произведением* (или сокращенно *произведением)* двух множеств является множество всех таких упорядоченных пар, что в каждой паре первый элемент берется из первого множества, а второй — из второго множества. Поэтому декартово произведение двух отношений, неформально выражаясь, представляет собой множество упорядоченных пар кортежей. Но мы снова должны сохранить свойство замкнутости; иными словами, необходимо, чтобы результат содержал кортежи как таковые, а не упорядоченные пары кортежей. Поэтому реляционной версией декартова произведения служит расширенная форма этой операции, в которой каждая упорядоченная пара кортежей заменяется одним кортежем, являющимся *объединением* двух рассматриваемых кортежей (в данном случае термин *объединение* используется в своем обычном смысле теории множеств, а не в его особом реляционном смысле). Это означает, что если даны следующие кортежи[1]

```
{ A1 a1, A2 a2, . . . , Am am }
```

И

```
{ B1 b1, B2 b2, ..., Bn bn }
```

то теоретико-множественное объединение этих двух кортежей представляет собой приведенный ниже единственный кортеж.

```
{ A1 a1, A2 a2, ..., Am am, B1 b1, B2 b2, ..., Bn bn }
```

Примечание. Здесь для упрощения предполагается, что эти два кортежа не имеют общих имен атрибутов. Данная тема более подробно рассматривается в следующем абзаце.

Еще одна проблема, которая возникает в связи с выполнением операции декартова произведения, безусловно, состоит в том, чтобы результирующее отношение имело правильно сформированный заголовок (т.е. чтобы оно принадлежало к правильному типу отношения). Итак, очевидно, что заголовок результирующего отношения состоит из всех

[1] В языке Tutorial D требуется, чтобы перед каждым из приведенных ниже выражений стояло ключевое СЛОВО TUPLE.

атрибутов, которые встречаются в заголовках обоих входных отношений. Поэтому, если два входных заголовка имеют какие-либо общие имена атрибутов, возникает проблема; если бы мы в таком случае разрешили сформировать произведение, то результирующий заголовок имел бы два атрибута с одинаковыми именами и поэтому не был бы правильно сформирован. Таким образом, если необходимо сформировать декартово произведение двух отношений, имеющих такие общие имена атрибутов, то следует вначале воспользоваться оператором RENAME, чтобы переименовать атрибуты должным образом.

Итак, определим (реляционное) **декартово произведение** a TIMES b отношений a и b, не имеющих общих атрибутов, как отношение, заголовок которого представляет собой (теоретико-множественное) объединение заголовков отношений a и b, а тело состоит из всех кортежей t, таких, что t является (теоретико-множественным) объединением кортежа, принадлежащего к отношению a, и кортежа, принадлежащего к отношению b. Следует отметить, что кардинальность результата равна произведению кардинальностей входных отношений, а и b, а степень результата — сумме степеней входных отношений.

Пример. Предположим, что отношения A и B показаны на рис. 7.3 (неформально выражаясь, отношение A содержит все текущие номера поставщиков, а в — все текущие номера деталей). В таком случае декартово произведение A TIMES B (которое показано в нижней части данного рисунка) включает в себя все текущие пары номеров поставщиков и номеров деталей.

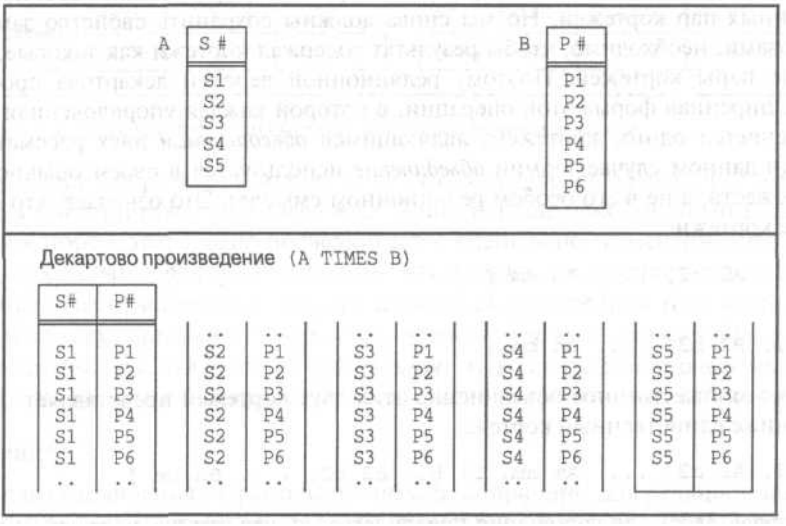


Рис. 7.3. Пример декартова произведения

Сокращение

Допустим, что отношение a имеет атрибуты X и Y (и, возможно, другие атрибуты), а 0 является таким оператором (как правило, "=", "≠", ">", "<" и т.д.), что логическое выражение X θ Y является правильно построенным и при определенных значениях X и Y приобретает истинностное значение (TRUE или FALSE). В таком случае 6-сокращением (или просто *сокращением)* отношения **a** по атрибутам X и Y (в указанном порядке), которое

задано с помощью приведенного ниже выражения, является отношение с тем же заголовком, что и в отношении a, и с телом, состоящим из всех кортежей отношения a, таких что выражение X θ Y для рассматриваемого кортежа принимает значение TRUE.

```
a WHERE X θ Y
```

Примечание. Выше фактически приведено определение сокращения, которое чаще всего встречается в литературе (включая первые издания этой книги). Но это определение можно обобщить следующим образом. Предположим, что отношение a имеет атрибуты X, Y, . . ., Z (и, возможно, другие атрибуты), а p является функцией с истинностными значениями, формальные параметры которой представляют собой строгое подмножество атрибутов X, Y, . . ., Z. В таком случае сокращение a в соответствии с p, которое определяется с помощью приведенной ниже функции, является отношением с тем же заголовком, что и в a, и с телом, состоящим из всех кортежей отношения a, таких, что функция p принимает значение TRUE для рассматриваемого кортежа.

```
a WHERE p
```

Оператор сокращения по сути позволяет получить "горизонтальное" подмножество заданного отношения, т.е. подмножество кортежей заданного отношения, для которых удовлетворяется некоторое указанное условие. На рис. 7.4 приведено несколько примеров (все эти примеры иллюстрируют только что определенную обобщенную версию сокращения).

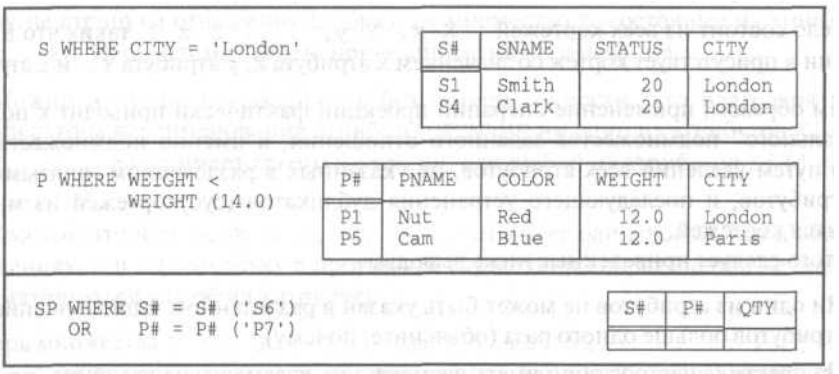


Рис. 7.4. Примеры применения операции сокращения

Из этого следуют приведенные ниже выводы.

1. Выражение p, которое следует за ключевым словом WHERE, безусловно, является логическим выражением; в действительности, это — *предикат,* в том смысле, ко торый подробно рассматривается в главе 9.
2. Данный предикат принято называть **условием сокращения.** Если это условие таково, что его проверка может выполняться для определенного кортежа t без исследования любых других кортежей, кроме t (и поэтому, в силу самого этого требования, без исследования каких-либо иных отношений, отличных от a), оно представляет собой **простое** условие сокращения. В этом смысле все условия сокращения, показанные

на рис. 7.4, являются простыми. А ниже для сравнения приведен пример, в котором используется непростое условие сокращения.

```
S WHERE ( ( SP RENAME S# AS X ) WHERE X = S# ) { P# } = P { P# }
```

Этот пример более подробно рассматривается ниже в данном разделе, вслед за описанием оператора деления.

3. Заслуживают внимания следующие эквивалентные выражения.

```
a WHERE p1 OR p2 ≡ (a WHERE p1 ) UNION ( a WHERE p2 )
a WHERE p1 AND p2 ≡ ( a WHERE p1 ) INTERSECT ( a WHERE p2 )
a WHERE NOT ( p ) ≡ a MINUS ( a WHERE p )
```

Проекция

Предположим, что отношение a имеет атрибуты x, Y, . . ., Z (и, возможно, другие атрибуты). В таком случае **проекция** отношения a по атрибутам X, y, . . . , z, которая определяется с помощью следующего выражения

```
a { X, Y, . . . , Z }
```

является отношением, соответствующим описанным ниже требованиям.

Его заголовок формируется из заголовка отношения a путем удаления всех атрибутов, не указанных в множестве { X, Y, . . . , z }.

Тело состоит из всех кортежей { X x , Y y, . . . , z z }, таких что в отношении a присутствует кортеж со значением x атрибута X, y атрибута Y... и z атрибута Z.

Таким образом, применение операции проекции фактически приводит к получению "вертикального" подмножества заданного отношения, а именно подмножества, полученного путем удаления всех атрибутов, не указанных в разделенном запятыми списке имен атрибутов, и последующего устранения дубликатов (суб)кортежей из множества оставшихся кортежей.

Из этого следует приведенные ниже выводы.

1. Ни один из атрибутов не может быть указан в разделенном запятыми списке имен атрибутов больше одного раза (объясните, почему).
2. На практике часто удобно иметь возможность указывать не атрибуты, по которым должна быть сформирована проекция, а скорее те атрибуты, которые "должны быть отброшены" (т.е. удалены) после выполнения операции проекции. Например, вместо использования формулировки "получить проекцию отношения P по атри бутам P#, PNAME, COLOR И CITY", МОЖНО ИСПОЛЬЗОВаТЬ формулировку "ИСКЛЮЧИТЬ с помощью операции проекции атрибут WEIGHT ИЗ отношения p" как показано ниже.

```
P { ALL BUT WEIGHT }
```

Другие примеры приведены на рис. 7.5. Обратите внимание на то, что в первом примере (проекция отношения поставщиков по атрибуту CITY) в результате содержатся только три кортежа (в соответствии с требованием об "устранении дубликатов"), хотя

переменная отношения S в настоящее время содержит пять кортежей. Аналогичные замечания, безусловно, относятся и к другим примерам. Следует также отметить, что и на сей раз в основе данной операции лежит операция проверки кортежей на равенство.

Соединение

Операция соединения имеет несколько разных вариантов. Но вполне очевидно, что наиболее важным из них является так называемое *естественное соединение;* в действительности, этот вариант настолько важен, что неуточненный термин *соединение* почти всегда рассматривается как обозначающий именно естественное соединение. Такое употребление данного термина принято и в настоящей книге. Поэтому сразу же перейдем к определению этой операции (оно является довольно абстрактным, но читатель должен был уже ознакомиться с естественным соединением на уровне интуитивного понимания по описаниям, приведенным в главе 3). Предположим, что отношения a и b, соответственно, имеют следующие атрибуты.

X1, X2, . . . , Xm, Y1, Y2, . . . , Yn

Y1, Y2, . . . , Yn, Z1, Z2, . . . , Zp

Это означает, что два рассматриваемых отношения имеют общее множество атрибутов Y, состоящее из атрибутов Yl, Y2 , . . . , Yn (и только из этих атрибутов), другие атрибуты отношения a образуют множество x, состоящее из атрибутов X1, *X2,* Xm, а другие атрибуты отношения b образуют множество z, состоящее из атрибутов Z1, Z2, . . . , Zp. Необходимо сделать приведенные ниже замечания.

- Можно и нужно предположить без потери точности, что благодаря наличию оператора переименования атрибутов RENAME НИ ОДИН ИЗ атрибутов xi (i = 1, 2, . . . , m) не имеет такого же имени, как любой из атрибутов z j (j = 1, 2, . . . , P).
- Каждый атрибут Yk (к = 1, 2, . . ., n) имеет одинаковый тип в обоих отношениях, а и Ь (поскольку в противном случае он по определению не должен рассматриваться как общий атрибут).

Теперь множества { XI, X2, . . . , Xm }, { Y1, Y2, . . . , Yn } и { Z1, Z2, . . ., Zp } могут рассматриваться, соответственно, как три составных атрибута x, Y и z. В таком случае (естественное) **соединение** a и b выражается следующим образом.

a JOIN b

Оно представляет собой отношение с заголовком { X, Y, Z } и телом, состоящим из всех таких кортежей { X x, Y y, z z }, что любой из этих кортежей присутствует и в отношении a, со значением x атрибута x и значением y атрибута Y, и в отношении b, со значением y атрибута Y и значением z атрибута Z.

Пример естественного соединения (естественное соединение s JOIN P по общему атрибуту CITY) приведен на рис. 7.6.

Примечание. В данной книге эта мысль была подчеркнута несколько раз (и фактически она проиллюстрирована на рис. 7.6), но необходимо снова явно указать, что соединения не

всегда устанавливаются между внешним ключом и соответствующим ему первичным (или потенциальным) ключом, даже несмотря на то, что такие соединения представляют собой широко распространенный и важный частный случай.

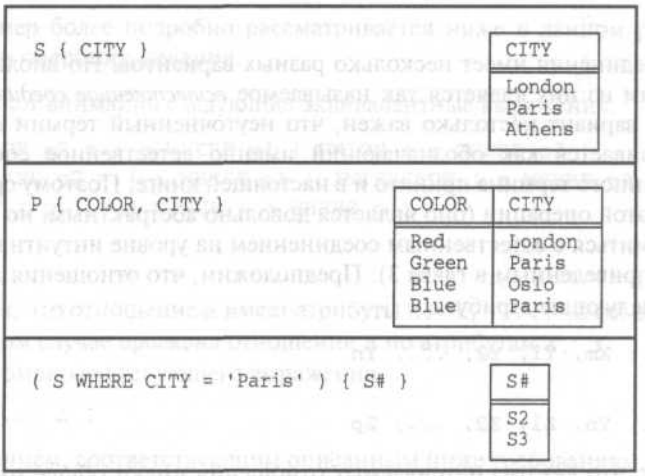


Рис. 7.5. Примеры применения операции проекции

S#	SNAME	STATUS	CITY	P#	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	Nut	Red	12.0
S1	Smith	20	London	P4	Screw	Red	14.0
S1	Smith	20	London	P6	Cog	Red	19.0
S2	Jones	10	Paris	P2	Bolt	Green	17.0
S2	Jones	10	Paris	P5	Cam	Blue	12.0
S3	Blake	30	Paris	P2	Bolt	Green	17.0
S3	Blake	30	Paris	P5	Cam	Blue	12.0
S4	Clark	20	London	P1	Nut	Red	12.0
S4	Clark	20	London	P4	Screw	Red	14.0
S4	Clark	20	London	P6	Cog	Red	19.0

Рис. 7.6. Естественное соединение S JOIN P

Кстати, следует отметить, что определение естественного соединения снова базируется на понятии равенства кортежей. Применительно к данному определению, необходимо также сделать приведенные ниже замечания.

- Если n = 0 (а это означает, что отношения a и b не имеют общих атрибутов), то операция a JOIN b вырождается[2] в операцию a TIMES b.
- Если m = p = 0 (а это означает, что отношения a и b относятся к одинаковому типу), то операция a JOIN b вырождается в операцию a INTERSECT b.

[2] Именно по этой причине версия языка Tutorial D, которая определена в [3.3], не включает непосредственной поддержки для оператора TIMES.

Теперь перейдем к изучению операции 9-соединения. Эта операция предназначена для тех случаев (сравнительно редких, но тем не менее достаточно важных), когда возникает необходимость соединить два отношения на основе некоторого оператора сравнения, отличного от сравнения на равенство. Предположим, что отношения a и b удовлетворяют требованиям для декартова произведения (т.е. не имеют общих имен атрибутов); пусть a имеет атрибут X и b — атрибут У, а x, y и θ удовлетворяют требованиям для 8-соединения. В таком случае операция **6-соединения** отношения a по атрибуту X с отношением b по атрибуту y определена как результат вычисления следующего выражения.

```
( a TIMES b ) WHERE X 9 У
```

Иными словами, результатом становится отношение с тем же заголовком, как и у декартова произведения a и b, и с телом, состоящим из множества всех кортежей t, таких что t присутствует в этом декартовом произведении, и выражение X θ У принимает значение TRUE для данного кортежа t.

В качестве примера предположим, что необходимо вычислить соединение по оператору "больше" отношения s по атрибуту CITY с отношением p по атрибуту CITY (итак, в данном случае 0 имеет вид ">", а поскольку атрибуты CITY определены как принадлежащие к типу CHAR, то оператор ">" просто означает "больше в алфавитном порядке"). Соответствующее реляционное выражение приведено ниже.

```
( ( S RENAME CITY AS SCITY ) TIMES
  ( P RENAME CITY AS PCITY )
) WHERE SCITY > PCITY
```

Обратите внимание на то, что в данном примере выполняется переименование атрибутов. (Безусловно, было бы достаточно переименовать только один из двух атрибутов CITY; единственная причина переименования обоих состоит в стремлении обеспечить равнозначность этих атрибутов.) Результат применения всего этого выражения приведен на рис. 7.7.

S#	SNAME	STATUS	SCITY	P#	PNAME	COLOR	WEIGHT	PCITY
S2	Jones	10	Paris	P1	Nut	Red	12.0	London
S2	Jones	10	Paris	P3	Screw	Blue	17.0	Oslo
S2	Jones	10	Paris	P4	Screw	Red	14.0	London
S2	Jones	10	Paris	P6	Cog	Red	19.0	London
S3	Blake	30	Paris	P1	Nut	Red	12.0	London
S3	Blake	30	Paris	P3	Screw	Blue	17.0	Oslo
S3	Blake	30	Paris	P4	Screw	Red	14.0	London
S3	Blake	30	Paris	P6	Cog	Red	19.0	London

Рис. 7.7. Соединение по оператору "больше" отношений поставщиков и деталей по городам

Если 0 представляет собой операцию проверки на равенство ("="), то 0-соединение называется **соединением по равенству,** или **соединением по эквивалентности.** Из этого определения следует, что результат любого соединения по эквивалентности должен включать два атрибута, характеризующихся тем свойством, что значения этих двух атрибутов равны в каждом кортеже данного отношения. А если один из этих двух атрибутов будет отброшен с помощью операции проекции, а другой соответствующим образом переименован

(в случае необходимости), то результатом становится естественное соединение! Например, следующее выражение представляет естественное соединение отношений поставщиков и деталей (по городам).

```
S JOIN P
```

Оно эквивалентно следующему более сложному выражению.

```
  { ( S TIMES ( P RENAME CITY AS PCITY ) ) WHERE CITY =
PCITY )
     { ALL BUT PCITY }
```

Примечание. В языке Tutorial D не предусмотрена непосредственная поддержка операции 8-соединения, поскольку на практике необходимость в ней не возникает достаточно часто и она так или иначе не относится к типу примитивных операций (т.е. может быть определена в терминах других операций, как было показано выше).

Деление

В [7.4] определены две разные операции "деления", которые именуются, соответственно, *малым делением* (Small Divide) и *большим делением* (Great Divide). В языке Tutorial D малым делением является операция <*divide*>, в которой выражение <*per*> состоит только из одного реляционного выражения <*relation exp*>, а большим делением — операция <*divide*>, в которой <*per*> состоит из заключенного в круглые скобки и разделенного запятыми списка из двух выражений <*relation exp*>. Приведенное ниже описание относится только к малому делению, причем лишь к конкретной ограниченной форме малого деления; подробное описание операции большого деления и дополнительные сведения об операции малого деления приведены в [7.4].

Следует также отметить, что рассматриваемая здесь версия малого деления отличается от первоначального определения этой операции Кондом; фактически она представляет собой усовершенствованную версию, в которой преодолены определенные сложности, возникавшие при использовании первоначальной версии операции в сочетании с пустыми отношениями. Она также отличается от версий, описанных в первых нескольких изданиях этой книги.

Перейдем к рассмотрению определения этой операции. Предположим, что отноше ния a и b, соответственно, имеют следующие атрибуты.

```
   XI,  X2,  . . . ,
Xm И
   Yl,  Y2,  ...,  Yn
```

Здесь ни один из атрибутов xi (i = 1, 2, . . ., m) не имеет одинакового имени с любым из атрибутов Yj (j = 1, 2, . . ., п). Пусть отношение с имеет следующие атрибуты:

```
XI,  X2,  . . . ,  Xm,  Yl,  Y2,  . . . ,  Yn
```

Это означает, что с имеет заголовок, представляющий собой (теоретико-множественное) объединение заголовков a и Ь. Будем рассматривать множества { XI, X2, ..., Хт } и { Yl, Y2, . . ., Yn }, соответственно, как составные атрибуты x и Y. В таком случае операция деления a на b по c (где a — *делимое,* Ъ — *делитель,* а c — *посредник)* может быть представлена с помощью следующего выражения.

```
a DIVIDEBY b PER c
```

Оно представляет собой отношение с заголовком { X } и телом, состоящим из всех кортежей { X x }, присутствующих в a, причем таких, что кортеж { x x, Y y } присутствует в c для всех кортежей { Y y }, присутствующих в b. Иными словами, неформально выражаясь, данный результат состоит из тех значений X, присутствующих в a, для которых соответствующие значения Y в c включают все значения Y из b. Обратите внимание на то, что и в этом определении применяется понятие равенства кортежей!

На рис. 7.8 приведены некоторые примеры деления. В каждом случае делимое (DEND) представляет собой проекцию текущего значения переменной отношения S по атрибуту S#; посредник (MED) в каждом случае является проекцией текущего значения переменной отношения SP по атрибутам S# и P#; а три делителя (DOR) являются такими, как указано на этом рисунке. В частности, заслуживает особого внимания последний пример, в котором делителем служит отношение, содержащее номера всех деталей, известных в настоящее время; результат (что вполне очевидно) показывает номера тех поставщиков, которые поставляют все эти детали. На основании данного примера можно сделать вывод, что оператор DIVIDEBY предназначен для запросов подобного общего характера; в действительности, каждый раз, когда версия запроса на естественном языке содержит в условной части слово "все" (например, "Определить поставщиков, которые поставляют все детали"), весьма велика вероятность того, что потребуется деление. (И действительно, Кодд специально предназначил операцию деления для использования в качестве алгебраического аналога *квантора всеобщности, во МНОРОМ подобно тому, что операция* проекции была предназначена для использования в качестве алгебраического аналога *квантора существования.* Дополнительные сведения по этой теме приведены в главе 8.)

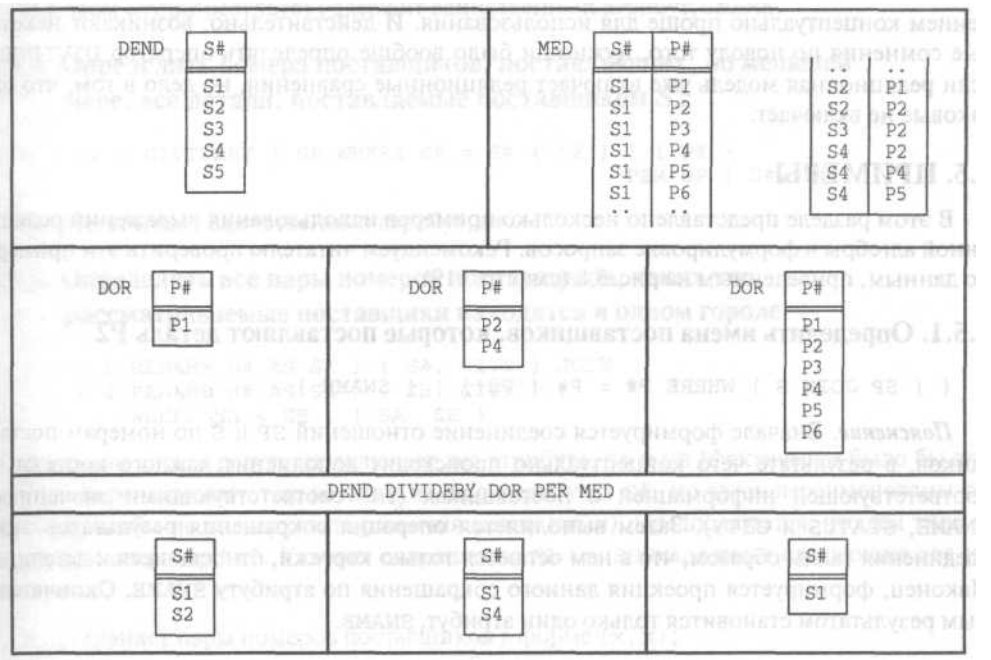


Рис. 7.8. Примеры деления

Но в связи с этим последним примером следует отметить, что запросы такого общего характера часто можно проще выразить в терминах реляционных сравнений, например, как показано ниже.

```
     S WHERE ( ( SP RENAME S# AS X ) WHERE X = S# ) {
p# } =            P { P# }
```

Результатом вычисления данного выражения становится отношение, содержащее все кортежи и только такие кортежи, которые относятся к поставщикам, поставляющим в настоящее время все известные детали. Объяснение этого выражения приведено ниже.

1. Для данного поставщика выражение

   ((SP RENAME S# AS X) WHERE X = S#) { P# }

   позволяет получить множество номеров деталей, поставляемых этим поставщиком.
2. Затем это множество сравнивается с множеством всех известных в настоящее время номеров деталей.
3. Соответствующий кортеж поставщика появляется в результате, если и только если эти два множества равны.

Ниже для сравнения приведен вариант такого выражения с использованием ключевого слова DIVIDEBY, которое на этот раз сформировано без сокращений.

S JOIN (S { S# } DIVIDEBY P { P# } PER SP { S#, P# })

У читателя вполне может сложиться впечатление, что вариант с реляционным сравнением концептуально проще для использования. И действительно, возникают некоторые сомнения по поводу того, нужно ли было вообще определять оператор DIVIDEBY, если реляционная модель уже включает реляционные сравнения, но дело в том, что она таковые не включает.

7.5. ПРИМЕРЫ

В этом разделе представлено несколько примеров использования выражений реляционной алгебры в формулировке запросов. Рекомендуем читателю проверить эти примеры поданным, приведенным на рис. 3.8 (см. стр. 119).

7.5.1. Определить имена поставщиков, которые поставляют деталь P2

■ `{ ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }`

Пояснение. Вначале формируется соединение отношений SP и S по номерам поставщиков, в результате чего концептуально происходит дополнение каждого кортежа SP соответствующей информацией о поставщиках (т.е. соответствующими значениями SNAME, STATUS и CITY). Затем выполняется операция сокращения результатов этого соединения таким образом, что в нем остаются только кортежи, относящиеся к детали P2. Наконец, формируется проекция данного сокращения по атрибуту SNAME. Окончательным результатом становится только один атрибут, SNAME.*

7.5.2. Определить имена поставщиков, которые поставляют по меньшей мере одну деталь красного цвета

```
{ ( ( P WHERE COLOR = COLOR ('Red') )
                    JOIN SP ) { S# } JOIN S ) { SNAME }
```

Единственным атрибутом результата снова становится SNAME. Кстати сказать, ниже приведена еще одна, эквивалентная формулировка того же запроса.

```
( (  ( P WHERE COLOR = COLOR ('Red') ) { P# }
                    JOIN SP ) JOIN S ) { SNAME }
```

Итак, данный пример иллюстрирует такую важную особенность, что часто существует несколько разных способов формулировки некоторого запроса. Обсуждение определенных следствий из этого факта приведено в главе 18.

7.5.3. Определить имена поставщиков, которые поставляют все детали

```
( ( S { S# } DIVIDEBY P { P# } PER SP { S#, P# } )
                                          JOIN S ) { SNAME }
```

Еще одна формулировка этого запроса приведена ниже.

```
( S WHERE
( ( SP RENAME S# AS X ) WHERE X = S# ) { P# } = P {
P# } )    { SNAME }
```

И в этом случае результат содержит единственный атрибут, SNAME.

7.5.4. Определить номера поставщиков, поставляющих, по меньшей мере, все детали, поставляемые поставщиком S2

```
S { S# } DIVIDEBY ( SP WHERE S# = S# ('S2') ) { P# }
                                            PER SP { S#, P# }
```

Результат имеет единственный атрибут, s#

7.5.5. Определить все пары номеров поставщиков, таких что рассматриваемые поставщики находятся в одном городе

```
( ( ( S RENAME S# AS SA ) { SA, CITY }
    JOIN ( S RENAME S# AS SB ) {SB,
    CITY } ) WHERE SA < SB ) { SA, SB }
```

В данном случае результат включает два атрибута, SA и SB (фактически было бы достаточно переименовать только один из двух атрибутов S#, но здесь переименованы оба атрибута, чтобы не подчеркивать различия между ними). Предполагается, что для типа s# определен оператор "<". Условие сокращения SA < SB выполняет следующие два назначения:

- устраняет пары номеров поставщиков в форме (x, x);
- гарантирует, что в результате не появятся обе пары, и (x , y) , и (y,x).

Ниже приведена еще одна формулировка этого запроса, позволяющая показать, как можно использовать ключевое слово WITH для упрощения задачи составления подобных выражений, которые могли бы в ином случае оказаться весьма сложными[3].

```
WITH ( S RENAME S#     AS SA ) { SA, CITY } AS T1,
     ( S RENAME S#     AS SB ) { SB, CITY } AS T2,
     T1 JOIN T2 AS     T3,
      T3 WHERE SA <    SB
     AS T4 :
     T4 { SA, SB }
```

Оператор WITH позволяет формировать большие, сложные выражения и при этом никоим образом не нарушает непроцедурный характер реляционной алгебры. Эта тема будет продолжена в описании, которое следует за очередным примером.

7.5.6. Определить имена поставщиков, которые не поставляют деталь P2

```
( ( S { S# } MINUS ( SP WHERE P# = P# ('P2') ) { S# } )
                                          JOIN S ) { SNAME }
```

Результат содержит единственный атрибут, SNAME.

Как было обещано, воспользуемся данным примером для иллюстрации еще одной идеи. Не всегда легко сразу представить себе, как сформулировать данный запрос в виде одного вложенного выражения. Но не обязательно проводить разработку именно в такой манере. Ниже показано, как можно поэтапно сформулировать запрос, рассматриваемый в данном примере. Здесь T6 обозначает желаемый результат.

```
WITH S { S# } AS T1,
     SP WHERE P# = P# ('P2') AS
T2,        T2 { S# } AS T3,
T1 MINUS T3 AS T4,
     T4 JOIN S AS T5,
     T5 { SNAME } AS T6 :
T6
```

Пояснение. Предполагается, что имена, введенные с помощью конструкции WITH (т.е. в данном примере имена в форме Ti), являются локальными по отношению к оператору, содержащему эту конструкцию. Итак, если система поддерживает режим "отложенного вычисления" (как, например, система PRTV [7.9]), то разбивка всего запроса на последовательность шагов в такой форме не оказывает отрицательного влияния на производительность. Данный запрос фактически обрабатывается, как описано ниже.

- Выражения, предшествующие двоеточию, не требуют немедленного вычисления системой; все, что должна сделать система, — это запомнить их наряду с именами, введенными с помощью соответствующих конструкций AS.
- Выражение, следующее за последним двоеточием, обозначает окончательный результат запроса (в данном примере этим выражением является просто "Тб").

---

[3] В действительности, скалярная форма оператора WITH уже использовалась в определении оператора DIST (см. раздел 5.5 главы 5), а в разделе 6.5 главы 6 была показана его реляционная форма в расширении сокращения UPDATE.

После достижения этой точки система больше не может откладывать вычисление и должна каким-то образом рассчитать требуемое значение (т.е. значение T6).

- Для вычисления значения T6, представляющего собой проекцию значения T5 по атрибуту SNAME, система должна вначале вычислить T5; для того чтобы вычислить T5, представляющее собой соединение T4 и S, система должна вначале вычислить T4 и т.д. Иными словами, система фактически должна вычислить первоначальное вложенное выражение точно так же, как если бы пользователь сразу подал это вложенное выражение на вход системы.

Краткое обсуждение общего вопроса о вычислении подобных вложенных выражений приведено в следующем разделе, а в главе 18 даны более подробные сведения по той же теме.

7.6. ОБЩЕЕ НАЗНАЧЕНИЕ АЛГЕБРЫ

Подведем итог изложенного выше в данной главе. В ней определена *реляционная алгебра,* т.е. коллекция операций на отношениях. В число рассматриваемых операций входят объединение, пересечение, разность, произведение, сокращение, проекция, соединение и деление, а также операция переименования атрибута, RENAME (по сути, именно это множество операций, кроме RENAME, было первоначально определено Коддом [7.1]). Здесь также представлен синтаксис данных операций и показано применение этого синтаксиса на многих примерах и иллюстрациях.

Но, как было отмечено в этом описании, восемь операций Кодда не составляют минимально возможного множества (и не были даже задуманы как таковые), поскольку некоторые из них не являются примитивными и могут быть определены в терминах других операций. Например, операции соединения, пересечения и деления могут быть определены в терминах остальных пяти операций (см. упражнение 7.6), поэтому их можно исключить без потери каких-либо функциональных возможностей. Но ни одна из оставшихся пяти операций не может быть определена в терминах остальных четырех, поэтому данные пять операций могут рассматриваться как составляющие множество примитивных операций или *минимальное множество* (тем не менее, следует отметить, что это — не единственное возможное минимальное множество)[4]. Но на практике эти дополнительные операции (особенно соединение) являются настолько полезными, что нужно сделать все возможное по обеспечению их непосредственной поддержки.

На данном этапе необходимо внести ясность в один важный вопрос. Хотя об этом явно не было сказано, но приведенное выше содержание данной главы со всей очевидностью наводит на мысль, что основное назначение рассматриваемой здесь алгебры состоит

[4] Это утверждение требует определенного уточнения. Во-первых, поскольку было показано, что произведение — это частный случай соединения, в указанном множестве примитивов произведение может быть заменено соединением. Во-вторых, фактически необходимо включить операцию RENAME, поскольку рассматриваемая здесь алгебра (в отличие от описанной в [7.1]) основана на использовании имен атрибутов, а не их порядкового расположения. В-третьих, в [3.3] описана своего рода версия "с сокращенным множеством команд" данной алгебры, называемая "алгеброй A", которая позволяет реализовать все функциональные возможности первоначальной алгебры Кодда (а также операции RENAME и нескольких других полезных операций) с использованием всего лишь двух примитивов, называемых remove (удаление) и nor ("исключительное ИЛИ").

просто в обеспечении выборки данных. Но это не соответствует истине. Основное назначение этой алгебры состоит в обеспечении возможности **составления реляционных выражений.** Эти выражения, в свою очередь, могут использоваться во многих разных операциях, включая выборку, но не ограничиваясь только выборкой. Некоторые возможные области применения таких выражений показаны в приведенном ниже списке (который не следует считать исчерпывающим).

- Определение области действия операции **выборки,** т.е. определение данных, которые должны быть получены в некоторой операции выборки (как уже было подробно описано).
- Определение области действия операции **обновления,** т.е. определение данных, которые должны быть вставлены, изменены или удалены в некоторой операции обновления (см. главу 6).
- Определение **ограничений целостности,** т.е. определение некоторых ограничений, которым должна удовлетворять база данных (см. главу 9).
- Определение **производных переменных отношения,** т.е. определение данных, которые должны быть включены в представление или снимок (см. главу 10).
- Определение **требований к стабильности,** т.е. определение данных, которые должны входить в сферу действия некоторой операции управления параллельным выполнением (см. главу 16).
- Определение **ограничений защиты,** т.е. определение данных, права доступа к которым должны предоставляться с помощью средств защиты того или иного типа (см. главу 17).

Вообще говоря, фактически эти выражения служат в качестве высокоуровневого символического описания намерений пользователя (например, применительно к некоторому конкретному запросу). А именно потому, что они являются высокоуровневыми и символическими, сами эти выражения могут стать предметом действия различных высокоуровневых, символических **правил преобразования.** Например, следующее выражение

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

имеющее смысл "Определить имена поставщиков, которые поставляют деталь P2" (см. пример 7.5.1), может быть преобразовано в логически эквивалентное, но, возможно, более эффективное выражение, представленное ниже.

```
( ( SP WHERE P# = P# ('P2') ) JOIN S ) { SNAME }
```

(Упражнение. В каком смысле второе выражение является лишь "возможно, более эффективным"? Почему только "возможно"?)

Таким образом, рассматриваемая алгебра служит удобной основой для **оптимизации** (дополнительная информация по этой теме приведена в разделе 3.5 главы 3). Поэтому, даже если пользователь формулирует запрос в первой из двух только что приведенных форм, *оптимизатор* системы должен преобразовать его перед выполнением в системе во вторую форму (в принципе производительность выполнения любого запроса не должна зависеть от того, в какой конкретной форме он был фактически сформулирован пользователем). Дополнительная информация по этой теме приведена в главе 18.

В завершение данного раздела сделаем одно замечание. Именно благодаря своему фундаментальному характеру описанная здесь алгебра часто используется как своего рода *эталон,* позволяющий оценить выразительную мощь некоторого конкретного языка. По сути, язык называется **реляционно полным** [7.1], если он является, по меньшей мере, таким же мощным, как и эта алгебра, т.е. если его выражения позволяют составить определение любого отношения, которое может быть определено с помощью выражений алгебры (под этим подразумевается оригинальная алгебра, которая описана в предыдущих разделах). Понятие реляционной полноты рассматривается более подробно в следующей главе.

7.7. НЕКОТОРЫЕ ДОПОЛНИТЕЛЬНЫЕ ЗАМЕЧАНИЯ

В данном разделе рассматриваются некоторые дополнительные темы, касающиеся восьми оригинальных операций.

Ассоциативность и коммутативность

Можно легко убедиться в том, что операция UNION является **ассоциативной.** Это означает, что если a, b и c — произвольные отношения одного и того же типа, то следующие два выражения логически эквиваленты.

```
   ( a UNION b ) UNION
c И       a UNION ( b
UNION c )
```

Поэтому для удобства разрешается записывать ряд операций UNION без круглых скобок; вследствие этого каждое из приведенных выше выражений можно без нарушения однозначности сокращенно просто записать следующим образом.

```
a UNION b UNION  c
```

Аналогичные замечания относятся к операциям INTERSECT, TIMES и JOIN (но не к операции MINUS). *Примечание.* На практике по этой причине, а также по некоторым другим причинам может оказаться более предпочтительной префиксная система обозначений, как, например, UNION (a,b, c), а не инфиксный стиль, применяемый в языке Tutorial D. Но в данной книге автор придерживается инфиксного стиля.

Кроме того, операции UNION, INTERSECT, TIMES и JOIN (но не операция MINUS) являются **коммутативными.** Это означает, что следующие выражения также являются логически эквивалентными.

```
   a  UNION
b И
   b UNION a
```

Аналогичное замечание относится и к операциям INTERSECT, TIMES и JOIN.

Мы снова вернемся к общей теме ассоциативности и коммутативности в главе 18. А что касается операции TIMES, необходимо отметить, что версия декартова произведения, которая определена в теории множеств, не является ни ассоциативной, ни коммутативной, а реляционная версия этой операции (как уже было показано) обладает обоими этими свойствами.

Некоторые эквивалентности

В этом подразделе просто перечислены некоторые важные эквивалентности, которые в основном предназначены для использования в дальнейшем изложении. В приведенных ниже формулах r обозначает произвольное отношение, а empty— пустое отношение такого же типа, как и r.

- r WHERE TRUE = r (сокращение тождества).
- r WHERE FALSE = empty.
- r { X, Y, . . . , z } = *r*, если x,Y,..., z— все атрибуты r (проекция тождества).
- r { } = TABLE_DUM, если r = empty, в противном случае TABLE_DEE (нульарная проекция).
- r JOIN r s r UNION r s r INTERSECT r = r.
- r JOIN TABLE_DEE = TABLE_DEE JOIN r = r (Т.е. TABLE_DEE ЯВЛЯеТСЯ еДИничным элементом применительно к операции соединения, так же, как в обычной арифметике нуль — единичный элемент применительно к сложению, а единица — единичный элемент применительно к умножению).
- rTIMES TABLE_DEE = TABLE_DEE TIMES r = r (эта эквикалентность представляет собой частный случай предыдущей).
- `r UNION empty = r MINUS empty = r.`
- `empty INTERSECT r = empty MINUS r = empty.`

Некоторые обобщения

Все операции JOIN, UNION и INTERSECT были определены первоначально как *бинарные* (т.е. каждая из них принимает в качестве операндов два и только два отношения)[5]; но, как было показано выше, эти операции могут быть однозначно обобщены для преобразования их в n-арные для произвольного n > 1. А что можно сказать по поводу таких операций с n = 1 или n = 0? Оказалось, что желательно иметь возможность определить, по крайней мере, с концептуальной точки зрения, операции "соединения", "объединения" и "пересечения" только с одним отношением и вообще без отношений (даже несмотря на то, что в языке Tutorial D не предусмотрена непосредственная синтаксическая поддержка ни для одной подобной операции). Определения таких операций приведены ниже. Допустим, что s — множество отношений (в случае объединения и пересечения все они принадлежат к одному и тому же типу отношения RT). В таком случае справедливы приведенные ниже утверждения.

- Если s содержит только одно отношение r, то результат всех операций соединения, объединения и пересечения всех отношений BS определен просто как r.
- Если s вообще не содержит отношений, то справедливы следующие утверждения:
  - соединение всех отношений в s определено как отношение TABLE_DEE (С тем же единичным элементом, который соответствует соединению);

[5] Операция MINUS также является бинарной. В отличие от нее, операции ограничения и проекции являются унарными.

- объединение всех отношений в s определено как пустое отношение типа RT;
- пересечение всех отношений в s определено как "универсальное" отношение[6] типа RT, иными словами, как то уникальное отношение типа RT, которое содержит все возможные кортежи с заголовком н, где н — заголовок отношения типа RT.

7.8. ДОПОЛНИТЕЛЬНЫЕ ОПЕРАЦИИ

С тех пор, как Кодд определил свои восемь оригинальных операций, многочисленные авторы предложили новые алгебраические операции. В данном разделе достаточно подробно рассматриваются несколько таких операций — SEMIJOIN, SEMIMINUS, EXTEND, SUMMARIZE и TCLOSE. В терминах применяемого в данной книге синтаксиса языка Tutorial D эти операции охватывают пять новых форм выражения *<nonproject>,* которые определены, как показано ниже.

```
   <semijoin>        ::= <relation exp>
SEMIJOIN <relation exp>

   <semiminus>
       ::= <relation exp> SEMIMINUS <relation exp>

   <extend>       ::= EXTEND <relation exp> ADD ( <extend
  add commalist> )
```

Если разделенный запятыми список *commalist* содержит только одно выражение *<extend add>,* круглые скобки можно опустить.

```
   <extend add>
        <exp> AS  <attribute
name>

   <summarize>
       ::=  SUMMARIZE <relation  exp>  PER
              <relation exp> ADD  (  <summarize
              add commalist>  )
```

Если разделенный запятыми список *commalist* содержит только одно выражение *<summarize add>,* круглые скобки можно опустить.

```
   <summarize add>
       ::= <summary type> [ ( <scalar exp>
                        ) ] AS <attribute
                        name>

   < summary type>
       ::= COUNT | SUM | AVG | MAX | MIN | ALL |
         ANY | COUNTD | SUMD | AVGD | ...

   <tclose>
       ::=  TCLOSE <relation  exp>
```

Различные реляционные выражения <*relation* exp>, упомянутые в приведенных выше правилах вывода в форме Бэкуса—Наура, не должны относиться к типу *<nonproject>*.

---

[6] Кстати, следует отметить, что термин "универсальное отношение" обычно используется в литературе во многих разных смыслах (см., например, [13.20]).

Полусоединение

Допустим, что a, b, X и Y соответствуют требованиям, которые указаны в подразделе "Соединение" раздела 7.4. В таком случае операция полусоединения отношений a и b (в указанном порядке), a SEMIJOIN b, определена как эквивалентная следующей операции.

```
( a JOIN b ) { X, Y }
```

Иными словами, полусоединение a и b представляет собой соединение a и b, к которому применена операция проекции по атрибутам a. Поэтому, неформально выражаясь, тело результата содержит те кортежи отношения a, которые имеют аналоги в отношении b.

Пример. Получить атрибуты S#, SNAME, STATUS И CITY поставщиков, которые поставляют деталь P2.

```
S SEMIJOIN ( SP WHERE P# = P# ('P2') )
```

Кстати, следует отметить, что многие практически применяемые запросы, которые основаны на использовании операции соединения, фактически требуют применения полусоединения; это означает, что на практике было бы желательно обеспечить непосредственную поддержку оператора SEMIJOIN. Аналогичное замечание относится и к оператору SEMIMINUS (см. следующий подраздел).

Полуразность

Операция полуразности между a **и** b (в указанном порядке), a SEMIMINUS b, определена как эквивалентная следующей операции.

```
a MINUS ( a SEMIJOIN b )
```

Поэтому, неформально выражаясь, телом результата являются те кортежи отношения a, которые не имеют аналогов в отношении b.

Пример. Получить атрибуты S#, SNAME, STATUS и CITY поставщиков, которые не поставляют деталь P2.

```
S SEMIMINUS ( SP WHERE P# = P# ('P2') )
```

Расширение

Читатель мог заметить, что описанная до сих пор алгебра не имеет вычислительных возможностей в том смысле, который под этим обычно подразумевается. Но на практике, безусловно, желательно иметь такие возможности. Например, может потребоваться, чтобы была возможность выполнить выборку значения арифметического выражения, такого как WEIGHT * 454, или обратиться к подобному значению в конструкции WHERE (здесь подразумевается[7], без учета сведений об единицах измерения, приведенных в разделе 5.4, что вес детали задан в фунтах, а 1 фунт = 454 грамма). Назначение операции **расширения** EXTEND состоит в поддержке таких возможностей. Если быть точнее, то операция EXTEND принимает одно отношение и возвращает другое, идентичное заданному, если не

---

[7] Здесь также предполагается, что "*" представляет собой допустимую операцию между весами и целыми числами.

Контрольный вопрос. Каков тип результата подобной операции?

считать того, что оно включает дополнительный атрибут, значения которого получены путем вычисления некоторого специального вычислимого выражения. Например, можно записать следующее выражение.

```
EXTEND P ADD ( WEIGHT * 454 ) AS GMWT
```

Данное выражение (следует подчеркнуть, что это — выражение, а не команда или оператор и поэтому может вкладываться в другие выражения) приводит к получению отношения с таким же заголовком, как P, не считая того, что оно содержит дополнительный атрибут с именем GMWT. Каждый кортеж этого отношения совпадает с соответствующим кортежем отношения P, за исключением того, что он дополнительно содержит значение веса в граммах GMWT, вычисленного с помощью заданного арифметического выражения WEIGHT * 454 (рис. 7.9).

P#	PNAME	COLOR	WEIGHT	CITY	GMWT
P1	Nut	Red	12.0	London	5448.0
P2	Bolt	Green	17.0	Paris	7718.0
P3	Screw	Blue	17.0	Oslo	7718.0
P4	Screw	Red	14.0	London	6356.0
P5	Cam	Blue	12.0	Paris	5448.0
P6	Cog	Red	19.0	London	8626.0

Рис. 7.9. Пример применения операции EXTEND

Важное замечание. Следует учитывать, что это выражение EXTEND не изменило в базе данных переменную отношения деталей; это — просто выражение и, как любое другое выражение, оно просто представляет некоторое значение. Как оказалось, это значение в данном конкретном случае весьма напоминает текущее значение переменной отношения деталей. (Иными словами, операция EXTEND не может рассматриваться как аналог операции ALTER TABLE ... ADD COLUMN языка SQL в реляционной алгебре.)

Теперь атрибут GMWT может использоваться в проекциях, сокращениях и т.д., например, как показано ниже.

```
( ( EXTEND P ADD ( WEIGHT * 454 ) AS GMWT )
          WHERE GMWT > WEIGHT ( 10000.0 ) ) { ALL BUT GMWT }
```

Примечание. Безусловно, в более дружественном языке может быть разрешено включать такие вычислительные выражения непосредственно в конструкцию WHERE примерно следующим образом.

```
P WHERE ( WEIGHT * 454 ) > WEIGHT ( 10000.0 )
```

(См. описание операции сокращения, приведенное в разделе 7.4.) Но данное средство фактически представляет собой просто удобное синтаксическое дополнение. В таком случае, вообще говоря, значением следующей операции расширения

```
EXTEND a ADD exp AS Z
```

является отношение, которое определено, как описано ниже.

- Заголовок результата состоит из заголовка a, дополненного атрибутом Z.
- Тело результата состоит из всех кортежей t, таких что t представляет собой кортеж отношения a, дополненный значением атрибута z, которое получено путем вычисления выражения exp на данном кортеже отношения a.

Отношение a не должно иметь атрибута с именем Z, а выражение exp не должно ссылаться на z. Следует отметить, что результат имеет кардинальность, равную кардинальности отношения a, и степень, равную степени отношения a плюс один. В этом результате типом атрибута z является тип выражения exp.

Ниже приведены дополнительные примеры.

1. `EXTEND S ADD 'Supplier' AS TAG`

   Это выражение фактически ставит отметку на каждом кортеже текущего значения переменной отношения S в виде символьной строки "Supplier" (литерал, или, вообще говоря, вызов селектора, — это, безусловно, допустимое вычислительное выражение).

2. `EXTEND ( P JOIN SP ) ADD ( WEIGHT * QTY ) AS SHIPWT`

   В данном примере иллюстрируется применение операции EXTEND к результату реляционного выражения, которое является более сложным, чем просто упоминание имени переменной отношения.

3. `( EXTEND S ADD CITY AS SCITY ) { ALL BUT CITY }`

   Имя атрибута, такое как CITY, также является допустимым вычислительным выражением. Следует отметить, что данный конкретный пример эквивалентен следующему.

   S RENAME CITY AS SCITY

   Иными словами, операция RENAME не является примитивной! Как оказалось, она может быть определена в терминах операции EXTEND (и операции проекции). Безусловно, мы не собираемся отбрасывать такую полезную операцию, как RENAME, просто любопытно отметить, что фактически она представляет собой всего лишь сокращенную форму.

4. `EXTEND P ADD {WEIGHT * 454 AS GMWT, WEIGHT * 16 AS OZWT)`

   В данном примере иллюстрируется применение "множественной операции EXTEND".

5. EXTEND S ADD COUNT ((SP RENAME Si AS X) WHERE X = S#) AS NP

   Результат этого выражения показан на рис. 7.10.

   Пояснение к этому выражению приведено ниже.

   а) Для указанного поставщика выражение ((SP RENAME s# AS X) WHERE x = s#) позволяет определить множество поставок, выполненных этим поставщиком.

   б) Затем к этому множеству поставок применяется агрегирующая операция COUNT и возвращает соответствующие данные о кардинальности множества (скалярное значение).

Таким образом, атрибут NP в полученных результатах обозначает количество деталей, поставляемых поставщиком, который указан с помощью соответствующего значения S#. Здесь заслуживает особого внимания значение NP для поставщика S5; множество поставок для поставщика S5 является пустым и поэтому вызов оператора COUNT возвращает нуль.

S#	SNAME	STATUS	CITY	NP
S1	Smith	20	London	6
S2	Jones	10	Paris	2
S3	Blake	30	Paris	1
S4	Clark	20	London	3
S5	Adams	30	Athens	0

Рис. 7.10. Еще один пример применения операции EXTEND

В этой связи кратко рассмотрим **агрегирующие операции.** Назначение каждой такой операции, вообще говоря, состоит в получении единственного скалярного значения из всех значений, присутствующих в некотором указанном атрибуте некоторого указанного отношения (обычно производного отношения). Типичными примерами являются операции COUNT, SUM, AVG, MAX, MIN, ALL и ANY. В языке Tutorial D вызов агрегирующего оператора *<agg op inv>* представляет собой особый вид скалярного выражения *<scalar* exp>, поскольку он возвращает скалярное значение. Этот вызов принимает следующую общую форму.

<agg op name> (*<relation exp>* [, *<attribute name>*])

Если в качестве имени агрегирующей операции *<add op name>* указано COUNT, то параметр с обозначением имени атрибута *<attribute name>* не имеет смысла и должен быть исключен; в противном случае он может быть исключен тогда и только тогда, когда реляционное выражение *<relation exp>* обозначает отношение степени один, и в этом случае по умолчанию подразумевается использование единственного атрибута результата этого выражения *<relation exp>*. Ниже приведено несколько примеров.

```
SUM ( SP WHERE S# = S# ('S1'), QTY SUM
( ( SP WHERE S# = S# ('S1') ) { QTY }
)
```

Обратите внимание на различие между этими двумя выражениями: первое из них позволяет получить сумму количеств всех поставок поставщика S1, а второе — сумму количеств всех различных поставок поставщика S1.

Если оказалось, что фактический параметр агрегирующего оператора представляет собой пустое множество, то операция COUNT (как уже было сказано) возвращает нуль и такое же значение возвращает операция SUM; операции MAX и MIN возвращают, соответственно, наименьшее и наибольшее значение применимого типа; операции ALL и ANY, соответственно, возвращают TRUE и FALSE, а операция AVG активизирует исключение.

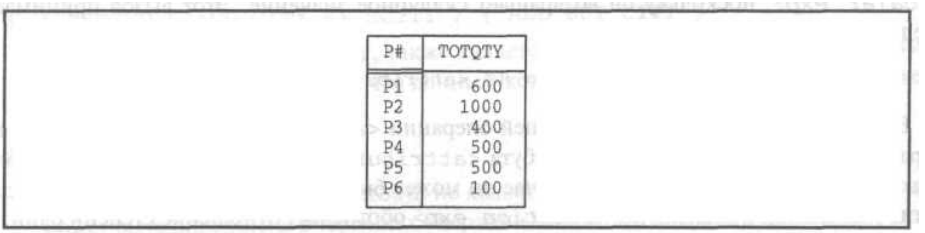
Агрегирование

Изложение материала данного подраздела необходимо начать с разъяснения того, что версия операции SUMMARIZE, описанная здесь, не совпадает с той, которая рассматривалась в предыдущих изданиях этой книги. В действительности, это — усовершенствованная версия, в которой преодолены некоторые сложности, возникавшие в предыдущей версии в связи с обработкой пустых отношений.

Как было показано выше, операция **расширения** предоставляет определенный способ включения в реляционную алгебру так называемых *горизонтальных вычислений,* или "вычислений, осуществляемых в пределах кортежа". Операция **агрегирования** выполняет аналогичную функцию для так называемых *вертикальных вычислений,* или "вычислений, осуществляемых в пределах атрибута". Например, в результате вычисления следующего выражения

```
SUMMARIZE SP PER P { P# } ADD SUM ( QTY ) AS TOTQTY
```

формируется отношение с атрибутами P# и TOTQTY, в котором имеется по одному кортежу для каждого значения P# в проекции отношения P по атрибуту P#, содержащему это значение P# и соответствующее общее количество (рис. 7.11). Иными словами, отношение SP концептуально подразделяется на группы, или множества кортежей (где имеется по одной группе для каждого номера детали в отношении P), после чего каждая такая группа используется для выработки одного кортежа в составе общего результата.



P#	TOTQTY
P1	600
P2	1000
P3	400
P4	500
P5	500
P6	100

Рис. 7.11. Пример применения операции SUMMARIZE

Вообще говоря, значение следующей **операции агрегирования**

```
SUMMARIZE a PER b ADD summary AS Z
```

представляет собой отношение, которое определено, как описано ниже.

- Прежде всего, отношение b должно принадлежать к такому же типу, так и некото рая проекция отношения a (т.е. каждый атрибут отношения b должен быть атри бутом отношения a). Допустим, что атрибутами этой проекции (равным образом и атрибутами отношения b) являются A1, A2, . . ., An.
- Заголовок результата состоит из заголовка отношения b, дополненного атрибутом Z.
- Тело результата состоит из всех кортежей t, таких что t является кортежем отно шения b, дополненным значением атрибута z. Это значение Z вычисляется путем выполнения операции агрегирования summary по всем кортежам отношения a,

которые имеют такие же значения атрибутов { A1, A2, . . ., An }, как и сам кортеж t. (Безусловно, если ни один кортеж из a не имеет такие же значения атрибутов { A1, A2, . . ., An }, как и сам кортеж t, то вычисление операции summary происходит на пустом множестве.)

Отношение b не должно иметь атрибута Z, а операция summary не должна ссылаться на Z. Обратите внимание на то, что результат имеет кардинальность, равную кардинальности отношения b, и степень, равную степени отношения b плюс один. Типом атрибута Z в этом результате является тип операции агрегирования summary.

Ниже приведен еще один пример.

```
SUMMARIZE ( P JOIN SP ) PER P { CITY } ADD COUNT AS NSP
```

Полученный при этот результат выглядит примерно следующим образом.

CITY	NSP
London	5
Oslo	1
Paris	6

Иными словами, этот результат включает по одному кортежу для каждого из трех городов, где хранятся детали (Лондона, Осло и Парижа); для каждого города в нем показано количество поставок деталей, хранящихся в этом городе.

Из сказанного следуют приведенные ниже выводы.

1. Еще раз следует отметить, что здесь мы снова встречаем пример операции, определение которой основано на понятии равенства кортежей.
2. Показанный здесь синтаксис допускает использование "множественных операций SUMMARIZE", например, следующим образом.

```
SUMMARIZE SP PER P { P# } ADD ( SUM ( QTY ) AS TOTQTY,
                               AVG ( QTY ) AS AVGQTY )
```

3. Общая форма операции агрегирования <*sumnarize*> (повторим ее еще раз) состоит в следующем.

SUMMARIZE <*relation exp*> PER <*relation exp*>
ADD (<*summarize add commalist*>)

Каждое выражение с добавлением результата агрегирования <*summarize add*> принимает, в свою очередь, следующую форму.

<*summary type*> [(<*scalar exp*>)] AS <*attribute name*>

К типичным разновидностям операции агрегирования <*summary type*> относятся COUNT, SUM, AVG, MAX, MIN, ALL, ANY, COUNTD, SUMD И AVGD. Суффикс "D" (сокращение от "distinct" — различный) в операциях COUNTD, SUMD и AVGD означает "устранить избыточные дублирующиеся значения перед выполнением операции агрегирования". Скалярное выражение <*scalar exp*> может включать ссылки на атрибуты в отношении, указанном с помощью реляционного выражения <*relation exp*>, которое непосредственно следует за ключевым словом

SUMMARIZE. Скалярное выражение <*scalar exp*> и окружающие его круглые скобки могут и должны быть опущены, только если типом операции агрегирования <*summary type*> является COUNT.

Кстати, обратите внимание на то, что выражение с добавлением результата агрегирования <*summarize add*> не равнозначно вызову оператора агрегирования <*agg op inv*>. Выражение <*agg op inv*> является скалярным и может находиться везде, где допускается применение литерала соответствующего типа. В отличие от этого, выражение <*summarize add*> представляет собой просто операнд операции SUMMARIZE; оно не является скалярным выражением, не имеет смысла за пределами контекста операции SUMMARIZE и фактически не должно появляться вне этого контекста.

4. Как читатель мог уже заметить, операция SUMMARIZE не является примитивной, поскольку она может быть промоделирована с помощью операции EXTEND. Например, следующее выражение

```
SUMMARIZE SP PER S { S# } ADD COUNT AS NP
```

определено как краткая форма выражения, приведенного ниже.

```
 ( EXTEND S { S# }
ADD ( ( ( SP RENAME S# AS X ) WHERE X = S# ) AS Y,
COUNT ( Y ) AS NP ) ) { S#, NP }
```

Указанное выражение может быть равным образом представлено в следующем виде.

```
WITH ( S { S# } ) AS T1,
     ( SP RENAME S# AS X ) AS T2,
     ( EXTEND T1 ADD ( T2 WHERE X = S# ) AS Y ) AS T3,
     ( EXTEND T3 ADD COUNT ( Y ) AS NP ) AS T4
: T4 { S#, NP }
```

Кстати, в этом случае атрибут Y имеет значение в виде отношения. Дополнительная информация по этой теме приведена в разделе 6.4..

5. Ниже приведен еще один пример.

```
SUMMARIZE S PER S { CITY } ADD AVG ( STATUS ) AS AVG_STATUS
```

В данном случае отношение, указанное вместе с ключевым словом PER, не только "относится к такому же типу, как" некоторая проекция отношения, к которому применяется операция агрегирования, но фактически является такой проекцией. В подобном случае допускается использование следующей сокращенной формы.

```
SUMMARIZE S BY { CITY } ADD AVG ( STATUS ) AS AVG_STATUS
```

Здесь конструкция PER <*relation exp*> заменена конструкцией BY <*attribute name commalist*>. Все имена атрибутов, указанные в этом списке, разделенном запятыми, должны быть атрибутами отношения, к которому применяется операция агрегирования.

6. Рассмотрим следующий пример.

```
SUMMARIZE SP PER SP { } ADD SUM ( QTY ) AS GRANDTOTAL
```

В соответствии с предыдущим пунктом, это выражение можно записать иначе, как показано ниже.

```
SUMMARIZE SP BY { } ADD SUM ( QTY ) AS GRANDTOTAL
```

В обоих случаях операции группирования и агрегирования выполняются на основе отношения, которое вообще не имеет атрибутов. Допустим, что sp — текущее значение переменной отношения SP, а также на время предположим, что отношение sp содержит по меньшей мере один кортеж. В таком случае все эти кортежи отношения sp имеют одно и то же значение, но для атрибутов, которые вообще не существуют (а именно, это один нуль-арный кортеж), поэтому в общем результате имеется лишь одна группа и, следовательно, только один кортеж (иными словами, вычисление результатов операции агрегирования выполняется точно один раз для всего отношения sp). Итак, в результате вычисления этого выражения будет получено отношение с одним атрибутом и одним кортежем; данный атрибут называется GRANDTOTAL, и единственное скалярное значение в единственном результирующем кортеже представляет собой сумму всех значений QTY в первоначальном отношении sp.

Если, с другой стороны, первоначальное отношение sp вообще не содержит кортежей, то не должно быть и групп, и поэтому не должно быть результирующих кортежей (т.е. результирующее отношение также является пустым). В отличие от этого, следующее выражение должно "действовать правильно" (т.е. возвращать правильный результат, нуль), даже если отношение sp будет пустым.

```
SUMMARIZE SP PER TABLE_DEE ADD SUM ( QTY ) AS GRANDTOTAL
```

Точнее, оно возвратит отношение с одним атрибутом, называемым GRANDTOTAL, и с одним кортежем, содержащим значение нуль. Поэтому, по мнению автора, должна быть предусмотрена возможность вообще опускать конструкцию PER, как в следующем примере.

```
SUMMARIZE SP ADD SUM ( QTY ) AS GRANDTOTAL
```

По определению, исключение из выражения конструкции PER должно быть эквивалентно применению конструкции PER TABLE_DEE.

Транзитивное замыкание

Операция транзитивного замыкания обозначается как "Tclose" (transitive closure). Здесь она упоминается в основном для полноты изложения; ее подробное описание выходит за рамки данной главы. Но в этом разделе будет по меньшей мере дано определение этой операции. Предположим, что a — бинарное отношение с атрибутами X и Y, из которых оба принадлежат к одному и тому же типу т. В таком случае транзитивное замыкание a, TCLOSE a, представляет собой отношение a^+ с таким же заголовком, как и у отношения a, и телом, представляющим собой надмножество отношения a, которое имеет приведенное ниже определение.

Следующий кортеж

{ X x, Y y }

появляется в отношении a^+, *если и только если он присутствует в отношении a или существует такая последовательность значений z1, z2, . . ., zn, относящихся к типу T, что все приведенные ниже кортежи появляются в* a.

{ X x, Y z1 }, { X z1, Y z2 }, . . . , { X zn, Y y }

Иными словами, если отношение a рассматривается как граф, то кортеж (x, y) появляется в a^+, только если в этом графе имеется путь от узла x к узлу y. Следует отметить, что тело отношения a^+ обязательно включает тело отношения a в качестве подмножества.

Дополнительные сведения о транзитивном замыкании приведены в главе 24.

7.9. ГРУППИРОВАНИЕ И РАЗГРУППИРОВАНИЕ

Тот факт, что могут существовать отношения с атрибутами, значениями которых являются отношения, в свою очередь приводит к необходимости иметь операторы, называемые здесь GROUP (группирование) и UNGROUP (разгруппирование), для прямого и обратного преобразования отношений, содержащих такие атрибуты, в отношения, которые их не содержат, например, как показано ниже.

SP GROUP { P#, QTY } AS PQ

Если рассматриваются данные, обычно применяемые в этой книге в качестве примера, то выполнение этого выражения приведет к получению результата, показанного на рис. 7.12.

Примечание. Рекомендуем читателю следить по данному рисунку за приведенными ниже пояснениями, поскольку они являются довольно отвлеченными (автор выражает свои сожаления, но ничего не может сделать).

Начнем это описание с такого замечания, что первоначальное выражение

SP GROUP { P#, QTY } AS PQ

можно читать как "группировать SP по s#", где s# — единственный атрибут SP, не упомянутый в спецификации GROUP. Результатом является отношение, которое определено следующим образом.

Во-первых, его заголовок выглядит так, как показано ниже.

{ S# S#, PQ RELATION { P# P#, QTY QTY } }

Иными словами, он состоит из атрибута PQ со значением в виде отношения (где значения PQ, в свою очередь, имеют атрибуты p# и QTY) наряду со всеми другими атрибутами SP (безусловно, выражение "все другие атрибуты SP" здесь означает просто атрибут s#). Во-вторых, его тело содержит точно по одному кортежу для каждого отдельного значения s# в отношении SP (и никаких других кортежей). Каждый кортеж в этом теле состоит из соответствующего значения s# (скажем, s) наряду со значением PQ (скажем, pq), которые получены, как описано ниже.

- Каждый кортеж SP заменяется кортежем (скажем, x), в котором компоненты P# и QTY "свернуты" в один компонент со значением в виде кортежа (скажем,y).

- Все компоненты у всех таких кортежей x, в которых значение S# равно s, "группируются" в одно отношение, pq, и тем самым образуется результирующий кортеж со значением s#, равным s, и значением PQ, равным pq.

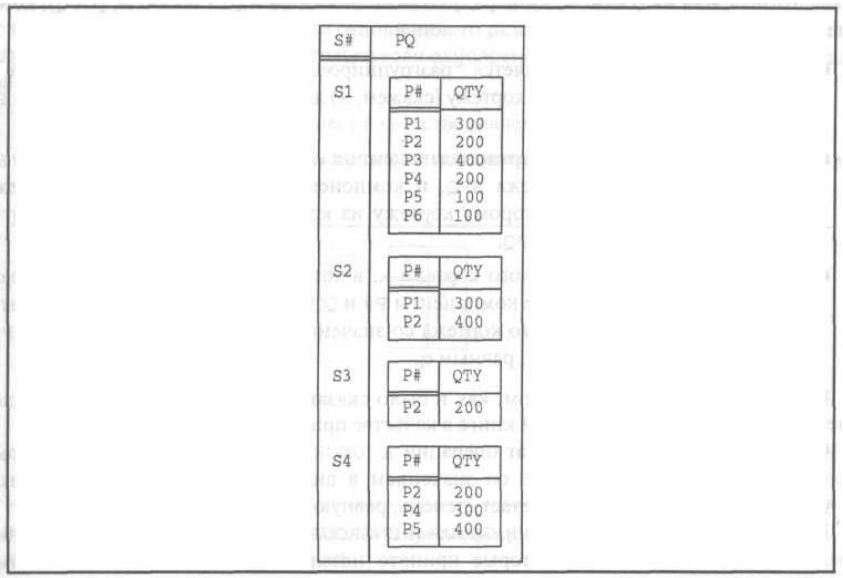


Рис. 7.12. Группирование отношения SP по атрибуту S#

Тем самым итоговый результат действительно становится таким, как показано на рис. 7.12. Следует особо подчеркнуть, что этот результат не включает ни одного кортежа для поставщика S5 (поскольку такие кортежи в настоящее время не включает и переменная отношения SP). Отметим, что результат выражения R GROUP { A1, A2, . . ., An } AS в имеет степень, равную nR - n + 1, где nR —степень отношения R. Теперь перейдем к рассмотрению *операции разгруппирования.* Допустим, что SPQ — отношение, показанное на рис. 7.12. В таком случае следующее выражение (что вполне естественно) позволяет снова получить обычное отношение SP, применяемое в данной книге в качестве примера.

```
SPQ UNGROUP PQ
```

А именно, это выражение приводит к получению отношения, которое определено следующим образом.

Во-первых, его заголовок выглядит так, как показано ниже.

```
{ S# S#, P# P#, QTY QTY }
```

Иными словами, заголовок состоит из атрибутов P# и QTY (производных от атрибута PQ), наряду со всеми другими атрибутами SPQ (т.е. в данном примере просто наряду

с атрибутом S#). Во-вторых, тело содержит точно по одному кортежу для каждого сочетания кортежа в SPQ и кортежа в значении PQ, которое входит в данный кортеж SPQ (и больше никаких других кортежей). Каждый кортеж в этом теле состоит из соответствующего значения S# (скажем, s), наряду со значениями P# и QTY (скажем, p и q), которые получены, как описано ниже.

- Каждый кортеж SPQ заменяется "разгруппированным" множеством кортежей, в котором имеется по одному кортежу (скажем, x) для каждого кортежа в значении PQ данного кортежа SPQ.
- Каждый такой кортеж x содержит компонент S# (скажем, s), равный компоненту S# из рассматриваемого кортежа SPQ, и компонент со значением в виде кортежа (скажем, y), равный некоторому кортежу из компонента PQ, который входит в рассматриваемый кортеж SPQ.
- Компоненты y каждого такого кортежа x, в котором значение S# равно s, "разворачиваются" в отдельные компоненты P# и QTY (скажем, p и q), что приводит к получению результирующего кортежа со значением s#, равным s, значением P#, равным p, и значением QTY, равным q.

Поэтому конечным результатом, как и было сказано, становится обычное отношение SP, рассматриваемое в данной книге в качестве примера.

Следует отметить, что результат операции R UNGROUP в (где отношения, которые являются значениями атрибута в со значением в виде отношения, имеют заголовок { A1, A2, . . . , An }) приобретает степень, равную nR+n-1, где nR — степень R.

Вполне очевидно, что операции GROUP и UNGROUP при совместном использовании предоставляют возможности, которые принято называть реляционными операциями "nest" (преобразовать обычную структуру во вложенную) и "unnest" (преобразовать вложенную структуру в обычную). Но автор предпочитает введенную здесь терминологию (группировать/разгруппировать), поскольку терминология nest/unnest вызывает ассоциации с понятием отношений NF^2 [6.10], а это понятие автор не поддерживает.

Для полноты описания в завершение этого раздела приведены некоторые замечания, касающиеся обратимости операций GROUP и UNGROUP (хотя автор признает, что данные замечания при первом чтении могут оказаться не совсем понятными). Если выполняется операция группирования некоторого отношения r определенным образом, то всегда существует обратная операция разгруппирования, позволяющая снова получить прежнее отношение r. Но если каким-то образом выполняется разгруппирование некоторого отношения r, то обратная операция группирования, позволяющая снова получить отношение r, может существовать или не существовать. Ниже приведен один пример (основанный на том примере, который можно найти в [6.4]). Предположим, что мы начинаем с отношения TWO (рис. 7.13) и разгруппировываем его для получения отношения THREE. Если же теперь будет выполнено группирование отношения THREE по атрибуту A (и результирующему атрибуту RVX со значением в виде отношения снова будет присвоено такое же имя), то будет получено отношение ONE, а не TWO.

Если после этого будет выполнено разгруппирование отношения ONE, мы вернемся к отношению THREE, а как уже было показано, отношение THREE может быть сгруппировано для получения отношения ONE; таким образом, операции группирования и разгруппирования

действительно являются обратными друг другу для данной конкретной пары отношений. Следует отметить, что в отношении ONE атрибут RVX является функционально зависимым от атрибута A (это действительно так, поскольку отношение ONE имеет кардинальность один)[8]. Вообще говоря, фактически можно утверждать, что если отношение r имеет атрибут RVX со значением в виде отношения, то разгруппирование r применительно к RVX является обратимым, если и только если одновременно удовлетворяются следующие два условия:

- ни один из кортежей r не имеет в качестве значения RVX пустое отношение;
- атрибут RVX является функционально зависимым от комбинации всех других атрибутов r.

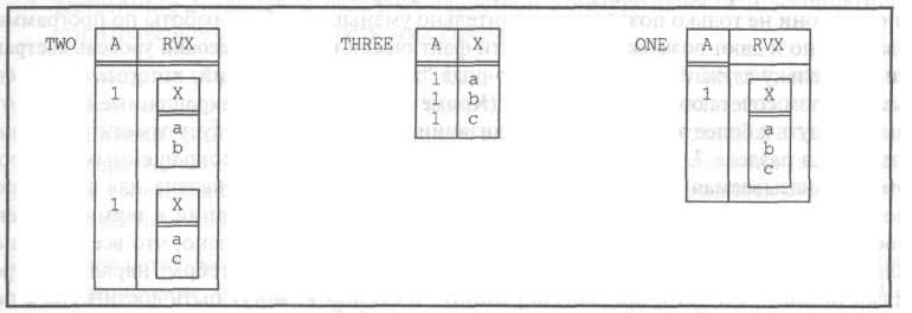


Рис. 7.13. Пример, который показывает, что операции разгруппирования и (повторного) группирования не всегда являются обратимыми

7.10. РЕЗЮМЕ

В данной главе рассматривалась **реляционная алгебра. В** начале этой главы была еще раз подчеркнута важность **реляционного свойства замкнутости** и **вложенных реляционных выражений** и показано, что для обоснованной трактовки понятия замкнутости необходимо ввести некоторые правила вывода типа отношения. На основании этих соображений был предусмотрен оператор **RENAME.**

Оригинальная алгебра состояла из восьми операций — традиционных операций с множествами, таких как **объединение, пересечение, разность** и **произведение** (все они были немного модифицированы с учетом того, что их операндами являются именно отношения, а не произвольные множества), а также специальных реляционных операций **сокращения, проекции, соединения** и **деления.** (Но в случае деления было отмечено, что запросы, требующие применения этой операции, могут быть всегда сформулированы в терминах реляционных операций сравнения, и многие находят, что такие формулировки интуитивно проще для понимания.) К этому первоначальному набору операций были добавлены операции **RENAME** (как уже было сказано), **SEMI JOIN, SEMIMINUS, EXTEND** и

[8] Рекомендуем читателю обратиться к главе 11 и, в частности, отметим, что в этой формулировке речь идет о той форме функциональной зависимости, которая касается значений отношений (а не о более обычной форме, которая касается переменных отношения).

SUMMARIZE, а также была упомянута операция **TCLOSE** и описаны операции **GROUP** и **UNGROUP.** В частности, следует отметить чрезвычайную важность описанной в этой главе операции **EXTEND** (в определенном смысле она является такой же важной, как и соединение).

Затем было отмечено, что не все рассматриваемые алгебраические операции являются **примитивными** (т.е. многие из них могут быть определены в терминах других); по мнению автора, такая ситуация является исключительно благоприятной. По этому поводу в [7.3] сказано: "Процесс определения языка должен начинаться с немногих тщательно выбранных примитивных операций ... Последующее его развитие должно по мере возможности осуществляться путем определения новых операций в терминах ... операций, определенных ранее"; иными словами, развитие языка происходит по пути определения полезных **сокращенных форм.** Если рассматриваемые сокращенные формы выбраны успешно, они не только позволяют значительно уменьшить объем работы по программированию, но и дают возможность перейти фактически **на более высокий уровень абстракции,** поскольку служат в качестве своего рода "наборов" концепций, которые естественным образом сочетаются друг с другом. (Кроме того, подобные сокращенные формы открывают путь к более эффективной реализации.) В этой связи следует отметить, что (как отмечено в разделе 7.6) в [3.3] описана своего рода алгебра "с сокращенным набором команд", называемая алгеброй A, которая специально предназначена для поддержки процесса систематического определения все более мощных операций в терминах очень небольшого набора примитивов; фактически в этой работе показано, что все функциональные возможности описанной в этой главе оригинальной алгебры, наряду с операциями RENAME, EXTEND, SUMMARIZE, GROUP, И UNGROUP, МОГуТ быть ДОСТИГНУТЫ С ПОМОЩЬЮ всего лишь двух примитивов, называемых remove и nor.

Вернемся к нашему резюме. Далее в настоящей главе было показано, как можно объединять алгебраические операции в выражения, которые служат для многих целей, в том числе для **выборки, обновления** и некоторых других. Кроме того, очень кратко была описана идея **преобразования** таких выражений в целях **оптимизации** (но эта идея рассматривается гораздо более подробно в главе 18).

Кроме того, было показано, что использование конструкции **WITH** позволяет упростить процесс составления сложных выражений; эта конструкция, по сути, дает возможность вводить имена для подвыражений и тем самым формулировать сложные выражения поэтапно, не нарушая при этом фундаментального свойства непроцедурности рассматриваемой здесь алгебры.

В настоящей главе было также указано, что некоторые операции являются **ассоциативными** и **коммутативными,** и приведены некоторые формулы **эквивалентности** (например, было показано, что любое отношение R эквивалентно некоторому сокращению R и некоторой проекции R). Кроме того, был рассмотрен вопрос о том, что означает выполнение операций соединения, объединения и пересечения только на одном отношении и вообще ни на одном отношении.

УПРАЖНЕНИЯ

7.1. Какие из реляционных операторов, описанных в этой главе, имеют определение, не основанное на понятии *равенства кортежей?*

7.2. Пусть дана обычная база данных поставщиков и деталей. Чему в этом случае будет равно значение выражения S JOIN SP JOIN P? Каков соответствующий предикат? *(Предостережение.* В этом упражнении есть ловушка!)

7.3. Пусть r — отношение степени n. Сколько существует различных проекций отношения r?

7.4. Выше упоминалось, что операции объединения, пересечения, произведения и естественного соединения обладают свойством коммутативности и ассоциативности. Проверьте справедливость этих утверждений.

7.5. Рассмотрим выражение a JOIN b. Если отношения a и b имеют непересекающиеся заголовки, это выражение эквивалентно выражению a TIMES b, а если они имеют один и тот же заголовок, оно эквивалентно a INTERSECT b. Проверьте справедливость этих утверждений. Какое выражение становится эквивалентным выражению a JOIN b, если заголовок отношения a является строгим подмножеством заголовка отношения b?

7.6. В предложенном Коддом первоначальном наборе из восьми операций пять (объединение, разность, произведение, сокращение и проекция) можно рассматривать как примитивные. Дайте определение операций естественного соединения, пересечения и (что значительно сложнее) деления в терминах этих примитивных операций.

7.7. В обычной арифметике операции умножения и деления — обратные по отношению друг к другу. Являются ли взаимно обратными операции TIMES И DIVIDEBY В реляционной алгебре?

7.8. В обычной арифметике существует два особых числа, 1 и 0, обладающее таким свойством, что для любого числа n справедливы формулы

```
n * 1   = 1 * n   = n и
n * 0   = 0 * n = 0
```

Какие отношения (если они существуют) выполняют аналогичные функции в реляционной алгебре? Изучите вопрос о том, какие воздействия оказывают на эти отношения алгебраические операции, рассматриваемые в данной главе.

7.9. В разделе 7.2 было указано, что реляционное свойство замкнутости является важным по той же причине, что и арифметическое свойство замкнутости. Но в арифметике встречается одна неприятная ситуация, когда свойство замкнутости нарушается — это деление на нуль. Возникает ли какая-либо аналогичная ситуация в реляционной алгебре?

7.10. Как известно, операция пересечения представляет собой частный случай операции соединения. Почему же обе эти операции не дают один и тот же результат, когда они выполняются вообще ни на одном отношении (т.е. применяются к пустому списку отношений)?

7.11. Какие из следующих выражений являются эквивалентными (и есть ли здесь такие выражения)?

а) `SUMMARIZE r PER r { } ADD COUNT AS CT.`

б) `SUMMARIZE r ADD COUNT AS CT.`

в) `SUMMARIZE r BY { } ADD COUNT AS CT.`

г) `EXTEND TABLE_DEE ADD COUNT ( r ) AS CT.`

7.12. Допустим, что r — отношение, заданное следующим выражением.

```
SP GROUP { } AS X
```

Покажите, какой вид имеет отношение r, взяв за основу обычный пример значения переменной отношения SP. Кроме того, покажите результат выполнения следующего выражения.

```
r UNGROUP X
```

Упражнения по составлению запросов

В основу всех остальных упражнений положена база данных поставщиков, деталей и проектов. В каждом упражнении читателю предлагается составить соответствующее алгебраическое выражение по словесной формулировке запроса. (В качестве интересного варианта можно предварительно рассмотреть ответы на некоторые упражнения, приведенные в приложении Д, и попытаться рассказать, что означает приведенное выражение на естественном языке.) Для удобства ниже повторно приведена (в общих чертах) структура используемой базы данных.

```
S    { S#, SNAME, STATUS, CITY
       } PRIMARY KEY { S# }
P    { P#, PNAME, COLOR, WEIGHT, CITY
       } PRIMARY KEY { P# }
J    { J#, JNAME, CITY
       } PRIMARY KEY {
       J# }
SPJ  { S#, P#, J#, QTY }
       PRIMARY KEY { S#, P#,j# }
       FOREIGN   KEY   {   S#   }
       REFERENCES S FOREIGN KEY {
       P# } REFERENCES P FOREIGN
       KEY { J# } REFERENCES J
```

7.13. Получить полные сведения обо всех проектах. '

7.14. Получить полные сведения обо всех проектах в Лондоне.

7.15. Определить номера поставщиков деталей для проекта с номером J1.

7.16. Определить все поставки, в которых количество деталей находится в диапазоне от 300 до 750 штук включительно.

7.17. Найти все существующие сочетания вида "цвет детали-город, из которого поставляются детали".

Примечание. Здесь и в последующих упражнениях слово "все" используется в значении "все, представленные в настоящий момент в базе данных", а не "все возможные".

7.18. Найти все такие тройки значений "номер поставщика—номер детали—номер проекта", для которых указанные поставщик, деталь и проект находятся в одном городе.

7.19. Найти все такие тройки значений "номер поставщика-номер детали—номер проекта", для которых указанные поставщик, деталь и проект не находятся в одном городе.

7.20. Найти все такие тройки значений "номер поставщика—номер детали-номер проекта", для которых никакие из двух поставщиков, деталей и проектов не находятся в одном городе.

7.21. Получить полные сведения о деталях, поставляемых поставщиком из Лондона.

7.22. Определить номера деталей, поставляемых поставщиком из Лондона для проекта в Лондоне.

7.23. Найти все пары названий городов, для которых поставщик из первого города поставляет детали для проекта во втором городе.

7.24. Определить номера деталей, поставляемых для всех проектов поставщиком из того же города, в котором разрабатывается проект.

7.25. Найти все номера проектов, детали для которых поставляются по крайней мере одним поставщиком из другого города.

7.26. Определить все пары номеров деталей, в которых обе детали поставляются одним и тем же поставщиком.

7.27. Определить общее количество проектов, детали для которых поставляются поставщиком с номером S1.

7.28. Определить общее количество деталей с номером P1, поставляемых поставщиком с номером S1.

7.29. Для каждой детали, поставляемой для проекта, определить номер детали, номер проекта и соответствующее общее количество.

7.30. Определить номера деталей, поставляемых для некоторого проекта, со средним количеством, составляющим больше 350 штук.

7.31. Определить названия проектов, детали для которых поставляются поставщиком с номером S1.

7.32. Определить цвета деталей, поставляемых поставщиком с номером si.

7.33. Установить номера деталей, поставляемых для любого проекта, разрабатываемого в Лондоне.

7.34. Определить номера проектов, в которых используется по крайней мере одна деталь, имеющаяся у поставщика с номером si.

7.35. Определить номера поставщиков по крайней мере одной детали, поставляемой по крайней мере одним поставщиком, который поставляет хотя бы одну деталь красного цвета.

7.36. Определить номера поставщиков со статусом, меньшим, чем статус поставщика с номером s1.

7.37. Определить номера проектов, разрабатываемых в городе, который находится на первом месте в алфавитном списке таких городов.

7.38. Определить номера проектов, для которых среднее количество поставляемых деталей с номером P1 больше, чем наибольшее количество любых деталей, поставляемых для проекта с номером Л.

7.39. Определить номера поставщиков детали с номером P1 для некоторого проекта в количестве, большем среднего количества деталей с номером P1, поставляемых для этого проекта.

7.40. Найти номера проектов, для которых поставщиками из Лондона не поставляется ни одна деталь красного цвета.

7.41. Определить номера проектов, детали для которых полностью поставляются поставщиком с номером S1.

7.42. Определить номера деталей, поставляемых для лондонских проектов.

7.43. Установить номера поставщиков одной и той же детали для всех проектов.

7.44. Определить номера проектов, в состав которых входят, по меньшей мере, все типы деталей, поставляемых поставщиком с номером S1.

7.45. Установить все города, в которых находится по крайней мере один поставщик, одна деталь или один проект.

7.46. Определить номера деталей, поставляемых либо лондонским поставщиком, либо для лондонского проекта.

7.47. Найти все пары "номер поставщика—номер детали", причем только такие, в которых данный поставщик не поставляет данную деталь.

7.48. Определить все пары номеров поставщиков (скажем, Sx и Sy), причем такие, что оба эти поставщика поставляют в точности одно и то же множество деталей.

Примечание. Для упрощения в данном упражнении рекомендуется использовать первоначальную версию базы данных поставщиков и деталей, а не расширенную версию базы данных поставщиков, деталей и проектов.

7.49. Подготовить в виде бинарного отношения "сгруппированную" версию всех поставок, в которой для каждой пары "номер поставщика-номер детали" показан соответствующий номер проекта и количество поставленных деталей.

7.50. Получить "разгруппированную" версию отношения, полученного в результате выполнения упражнения 7.49.

СПИСОК ЛИТЕРАТУРЫ

7.1. Codd. E.F. Relational Completeness of Data Base Sublanguages // Randall J. Rustin (ed.). Data Base Systems, Courant Computer Science Symposia Series 6. — Englewood Cliffs, N.J.: Prentice-Hall, 1972.

В этой статье Кодд впервые привел формальное определение операторов исходной версии реляционной алгебры (определения аналогичных операторов были опубликованы также в [6.1], однако они были неполными и менее формальными). ***Примечание.*** Эта статья обладает одним недостатком. В ней "для удобства обозначения и описания" предполагается, что атрибуты отношений упорядочены слева направо и, следовательно, могут быть идентифицированы по своему порядковому номеру. (Тем не менее, Кодд указал, что "с точки зрения хранения и выборки информации ... лучше использовать имена, а не порядковые номера". То же самое он утверждал и в [6.1].) Поэтому в данной статье ничего не сказано об операторе RENAME и не рассмотрен вопрос о выводе типа результата. Возможно, вследствие этих упущений те же критические замечания применимы сегодня ко многим рассуждениям в литературе по реляционной алгебре, к современным продуктам SQL и (в меньшей степени) к стандартам языка SQL.

Другие комментарии к этой статье можно найти в главе 8, в частности в разделе 8.4.

7.2. Darven H. (под псевдонимом Andrew Warden). Adventures in Relationland // Date C.J. Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990. Это серия коротких статей, в которых в оригинальном, развлекательном и содер жательном стиле исследуются различные стороны реляционной модели и реляци онных СУБД.

7.3. Darven H. Valid Time and Transaction Time Proposals: Language Design Aspects // Etzion O., Jajodia S., Sripada S. (eds.). Temporal Databases: Research and Practice. New York, N.Y.: Springer-Verlag, 1998.

7.4. Darwen H. and Date C.J. Into the Great Divide // Darwen H. and Date C.J. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

В данной статье анализируется оригинальная операция деления, определенная Коддом в [7.1], а также обобщение этой операции, сделанное Холлом (Hall), Хитчкоком (Hitchcock) и Тоддом (Todd) [7.10], которое, в отличие от определенной Коддом оригинальной операции деления, позволяет делить любые отношения на любые другие отношения. (Оригинальная операция деления Кодда была определена только для таких делимых и делителей, заголовок делителя которых является подмножеством заголовка делимого.) В этой статье описаны трудности, возникающие при использовании данных операций для пустых отношений, и указано,

что ни одна из них не позволяет решить задачу, для решения которой она первоначально была предназначена (т.е. ни одна из них не смогла стать аналогом квантора всеобщности, как того хотелось бы). Для преодоления трудностей предложены пересмотренные версии операций деления ("Small Divide", малое деление, и "Great Divide", большое деление).

Примечание. Как видно из синтаксиса языка Tutorial D, это действительно две разные операции. В частности, операция большого деления представляет собой (к сожалению) не вполне совместимое сверху вниз расширение операции малого деления. В данной статье предложено также не называть пересмотренные операции "делением" (в связи с последним примечанием см. упр. 7.7).

Для справок приведем оригинальное определение операции деления, предложенное Коддом. Пусть даны отношения A и B, соответственно, с заголовками {x, Y} и {Y} (атрибуты X и Y могут быть составными). Тогда результатом вычисления выражения A DIVIDEBY в будет отношение с заголовком {X} и телом, состоящим из всех кортежей {x x}, таких, что кортеж {X x, Y y} содержится в теле отношения A для всех кортежей {Y y}, содержащихся в отношении в. Другими словами, неформально выражаясь, результат состоит из тех значений атрибута X из отношения A, для которых соответствующие значения атрибута Y (в отношении A) содержат все значения атрибута Y в отношении в.

7.5. C.J. Date. Quota Queries (в трех частях) // Date C.J., Darven H., and McGoveran D. Relational Database Writings 1994-1997. — Reading, Mass.: Addison-Wesley, 1998.

Лимитирующий запрос (quota query) — это такой запрос, в котором указывается желаемый предел кардинальности результирующего отношения. Например, это может быть запрос "Найти три самые тяжелые детали". На языке Tutorial D его можно записать следующим образом.

```
P QUOTA ( 3, DESC WEIGHT )
```

Это выражение является сокращенной записью следующего оператора.

```
     ( EXTEND P
ADD COUNT ( ( P RENAME WEIGHT AS WT ) WHERE WT > WEIGHT
) AS # HEAVIER ) WHERE # HEAVIER < 3 ) { ALL BUT
#_HEAVIER }
```

Здесь имена WT и #_HEAVIER выбраны произвольным образом. Если взять за основу наши обычные данные, то результат будет состоять из деталей с номерами P2, P3 и P6. В этой статье глубоко анализируются лимитирующие запросы, а также предлагается несколько сокращенных синтаксических форм как для записи этих запросов, так и для других целей.

Примечание. В [3.3] описан альтернативный подход к формулировке лимитирующих запросов, в котором предусмотрено использование нового реляционного оператора, RANK.

7.6. Goldstein R.C. and Strnad A.J. The MacAIMS Data Management System // Proc. 1970 ACM SICFIDET Workshop on Data Description and Access. — November 1970.

См. аннотацию к [7.7].

7.7. Strnad A.J. The Relational Approach to the Management of Data Bases // Proc. IFIP Congress. — Ljubljana, Yugoslavia. — August 1971.

Мы упоминаем систему MacAIMS [7.6], [7.7] в основном ради исторического интереса. Она является наиболее ранним примером системы, поддерживающей n-арные отношения и алгебраический язык. Интересным также является то, что эта система разрабатывалась параллельно с работой Кодда над реляционной моделью (и, до определенной степени, независимо). Однако, в отличие от работы Кодда, система MacAIMS не получила существенного развития.

7.8. Notley M.G. The Peterlee IS/1 System // IBM UK Scientific Centre Report UKSC-0018.-March 1972.

См. аннотацию к [7.9].

7.9. Todd S.J.P. The Peterlee Relational Test Vehicle — A System Overview // IBM Sys. J. — 1976. -15, №4.

Peterlee Relational Test Vehicle (PRTV) — это экспериментальная система, разработанная в научном центре компании IBM UK, в городе Питерли, Англия. Она была разработана на основе более ранней системы IS/1 [7.8], которая, вероятнее всего, явилась первой реализацией идей Кодда. В ней поддерживались n-арные отношения и версия алгебры под названием ISBL (Information System Base Language — базовый язык информационных систем). Эта версия реляционной алгебры основывалась на предложениях, изложенных в [7.10]. Первоисточником приведенных в этой главе идей, касающихся вывода типов отношений, являются алгебра ISBL и предложения из [7.10]. Система PRTV обладала следующими важными свойствами.

- Поддерживала операторы RENAME, EXTEND и SUMMARIZE.
- Включала в себя сложные средства для преобразования выражений (см. гла ву 18).
- Включала средства отложенного вычисления, что важно как для оптимизации, так и для поддержки представлений (см. обсуждение конструкции WITH в дан ной главе).
- Предусматривала возможность добавлять определенные пользователем опера торы.

7.10. Hall P.A.V., Hitchcock P., and Todd S.J.P. An Algebra of Relations for Machine Computation // Conf. Record of the 2nd ACM Symposium on Principles of Programming Languages. — Palo Alto, Calif. — January 1975.

7.11. Klug A. Equivalence of Relational Algebra and Relational Calculus Query Languages Having Aggregate Functions // JACM 29. — July 1982. — № 3.

В статье определен ряд дополнений к реляционной алгебре и реляционному исчислению (см. главу 8), предназначенных для включения поддержки операций агрегирования, и демонстрируется эквивалентность полученных при этом двух расширенных формальных языков.

ГЛАВА 8

Реляционное исчисление

8.1. ВВЕДЕНИЕ

Реляционная алгебра и реляционное исчисление представляют собой два альтернативных подхода. Принципиальное различие между ними состоит в следующем. *Реляционная алгебра* определяет в явном виде набор операций (соединение, объединение, проекция и т.д.), которые можно использовать, чтобы сообщить системе, как в базе данных из определенных отношений *сформировать* некоторое требуемое отношение, а *реляционное исчисление* просто задает систему обозначений для *определения* требуемого отношения в терминах существующих отношений. Например, рассмотрим запрос: "Выбрать номера поставщиков и названия городов, в которых находятся поставщики детали с номером P2". Алгебраическая версия этого запроса может быть составлена примерно так (мы умышленно не используем формальный синтаксис, приведенный в главе7).

- Выполнить соединение отношений поставщиков и поставок SP по атрибуту S#.
- С помощью операции сокращения выделить из результатов этого соединения кор
  тежи, которые относятся к детали с номером P2.

- Сформировать проекцию результатов этой операции сокращения по атрибутам s# И CITY.

Этот же запрос в терминах реляционного исчисления формулируется приблизительно следующим образом.

- Получить атрибуты s# и CITY для таких поставщиков, для которых в отношении SP существует запись о поставке с тем же значением атрибута s# и со значением атрибута P#, равным P2.

В этой формулировке пользователь лишь указывает определенные характеристики требуемого результата, предоставляя системе решать, что именно и в какой последовательности соединять, проецировать и т.д., чтобы получить необходимый результат. Итак, можно сказать, что, по крайней мере, внешне формулировка запроса в терминах реляционного исчисления носит характер описания, а в терминах реляционной алгебры — *предписания.* В реляционном исчислении просто указывается, в чем *заключается* проблема, тогда как в реляционной алгебре задается *процедура решения* этой проблемы. Или, говоря *очень* неформально, алгебра имеет процедурный характер (пусть на высоком уровне, но все же процедурный, поскольку задает необходимые для выполнения процедуры), а исчисление — непроцедурный.

Подчеркнем, однако, что упомянутые различия существуют только внешне. На самом деле *реляционная алгебра и реляционное исчисление логически эквивалентны.* Каждому выражению в алгебре соответствует эквивалентное выражение в исчислении, и точно также каждому выражению в исчислении соответствует эквивалентное выражение в алгебре. Это означает, что между ними существует взаимно-однозначное соответствие, а различия связаны лишь с разными *стилями* выражения: исчисление ближе к естественному языку, а алгебра — к языку программирования. Но, повторим еще раз, эти различия только кажущиеся, а не реальные. В частности, ни один из этих подходов нельзя назвать "более непроцедурным" по сравнению с другим. Подробнее вопрос эквивалентности этих двух подходов будет рассматриваться в разделе 8.4 настоящей главы.

Реляционное исчисление основано на разделе математической логики, который называется исчислением предикатов. Идея использования исчисления предикатов в качестве основы языка баз данных впервые была высказана в статье Кунса (Kuhns) [8.6]. Понятие *реляционного исчисления,* т.е. специального метода применения исчисления предикатов в реляционных базах данных, впервые было сформулировано Коддом в [6.1], а в [8.1] Кода представил язык, основанный непосредственно на реляционном исчислении и названный "подъязыком данных ALPHA". Сам язык ALPHA никогда не был реализован, однако язык QUEL [8.5], [8.10]—[8.12], который действительно был реализован и некоторое время серьезно конкурировал с языком SQL, очень походил на язык ALPHA, оказавший заметное влияние на построение языка QUEL.

Основным средством реляционного исчисления является понятие переменной области значений. Согласно краткому определению, переменная области значений — это переменная, *принимающая значения* из некоторого заданного отношения, т.е. переменная, допустимыми значениями которой являются кортежи заданного отношения. Другими словами, если переменная области значений v изменяется в пределах отношения r, то в любой конкретный момент выражение "v" представляет некоторый кортеж t отношения r. Например, запрос "Получить номера поставщиков, находящихся в Лондоне" может быть представлен на языке QUEL следующим образом.

```
RANGE OF SX IS S ;
RETRIEVE ( SX.S# ) WHERE SX.CITY = "London" ;
```

Единственной переменной области значений здесь является переменная SX, которая принимает значения из отношения, представляющего собой текущее значение переменной отношения S (оператор RANGE — *оператор определения* этой переменной области значений). Оператор RETRIEVE означает следующее: "Для каждого возможного значения переменной SX выбирать компонент S# этого значения тогда и только тогда, когда компонентом CITY этого значения является London".

В связи с тем, что первоначальная версия реляционного исчисления основана на переменных области значений (в отличие от исчисления *доменов,* речь о котором пойдет ниже), ее называют также исчислением кортежей. Исчисление кортежей подробно описано в разделе 8.2.

Примечание. Для удобства примем следующее соглашение: далее в этой книге термины *исчисление* и *реляционное исчисление,* приведенные без такого уточнения, как *кортежей* или *доменов,* будут означать именно исчисление кортежей (там, где это играет какую-то роль).

Впоследствии Лакруа (Lacroix) и Пиротт (Pirotte) [8.7] предложили альтернативную версию исчисления, называемую исчислением доменов, в которой переменные области значений принимают значения из доменов, т.е. являются переменными, изменяемыми на доменах, а не на отношениях. (Данный термин нелогичен, ведь если исчисление доменов называется так по указанной причине, то по той же причине исчисление кортежей следовало бы назвать исчислением *отношений.)* В литературе предлагается множество языков исчисления доменов. Наиболее известным из них, пожалуй, является QBE (Query-By-Example — язык запросов по образцу), который впервые описан в [8.14]; в действительности язык QBE является смешанным, поскольку в нем присутствуют и элементы исчисления кортежей. Существует несколько коммерческих реализаций языка QBE, или "QBE-подобного" языка. В общих чертах исчисление доменов будет описано в разделе 8.7, а сам язык QBE вкратце рассматривается в разделе 8.8.

Примечание. Стремясь сделать эту главу короче, мы умышленно опускаем подробное описание некоторых вопросов, касающихся реляционного исчисления (таких как группирование и разгруппирование), которые имеют свои аналоги в реляционной алгебре (см. главу 7). Мы также опускаем рассмотрение соответствующих версий реляционных операторов обновления. Все эти сведения представлены в [3.3].

8.2. ИСЧИСЛЕНИЕ КОРТЕЖЕЙ

Как и при описании реляционной алгебры в главе 7, сначала введем для реляционного исчисления конкретный синтаксис, взяв за образец (хотя умышленно не совсем точный) версию исчисления языка Tutorial D, определенного в приложении A книги [3.3], а затем перейдем к обсуждению семантики. В следующем подразделе обсуждается синтаксис, а в остальных — семантика.

Синтаксис

Примечание. Многие из приведенных здесь синтаксических правил, сформулированных на условном языке, не будут понятны до тех пор, пока вы не изучите семантический материал, который следует далее. Однако автор все же решил собрать все правила вместе для удобства использования в дальнейшей работе.

Начнем с повторения синтаксиса параметра *<rela tion exp>,* приведенного в главе 7.

```
<relation exp>
    ::= RELATION { <tuple exp
      commalist> } | <relvar name> |
      <relation op inv> j <with exp> |
      <introduced name> j ( <relation
      exp> )
```

Иными словами, синтаксис параметра *<relation exp>* остается прежним, однако один из наиболее важных его подпараметров, *<relation op inv>,* который является единственным рассматриваемым в данной главе во всех подробностях, теперь будет иметь совершенно иное определение.

```
<range var def>
  ::= RANGEVAR <range var name>
      RANGES OVER <relation exp commalist> ;
```

Параметр *<range var name>* может использоваться[1] как < tuple *exp>,* но лишь в определенном контексте, а именно:

- перед точкой в уточняющем выражении *<range attribute ref>;*
- сразу после квантора в параметре *<quantified bool exp>;*
- как операнд в параметре *<Ьоо1 exp>;*
- как параметр *<proto tuple>* или как выражение <exp> (или операнд с выражением <exp>) в параметре *<proto tuple>*.

```
<range attribute ref>
    ::= <range var name> . <attribute
name>
      AS <ttribute name> ]
```

Параметр *<range attribute ref>* может использоваться как параметр <exp>, но только в определенных контекстах, а именно:

- как операнд параметра *<bool exp>;*
- как параметр *<proto tuple>* или как выражение *<exp>* (или операнд с выражением <exp>) в параметре *<proto tuple>*.

```
<bool exp>
    ::= ... все обычные варианты,
      наряду с | <quantified bool exp>
```

Ссылки на переменные области значений в значении параметра *<bool exp>* могут быть свободными в пределах этого параметра *<bool exp>* тогда и только тогда, когда выполнены два следующих условия.

---

[1] Мы не приводим здесь подробного описания параметра < tuple *exp>,* полагая, что общее представление о нем можно получить, изучая примеры. Но по причинам, которые в данном случае не имеют большого значения, здесь используется немного иной синтаксис по сравнению с предыдущими главами.

- Параметр <*bool exp*> присутствует непосредственно в выражении <*relation op inv*> (т.е. параметр <*boo1 exp*> следует сразу за ключевым словом WHERE).
- Ссылка (обязательно свободная) именно на ту же самую переменную области значений непосредственно присутствует в значении выражения <*proto tuple*>, непосредственно содержащегося в том же выражении <*relation op inv*> (т.е. параметр <*proto tuple*> находится непосредственно перед ключевым словом WHERE).

Примечание по терминологии. **В** контексте реляционного исчисления (в версии исчисления доменов или исчисления кортежей) логические выражения <*Ъоо1 exp*> часто называют **правильно построенными формулами** (Well-Formed Formula — WFF, что произносится как "вэфф"). Далее мы также будем часто пользоваться этой терминологией.

```
<quantified bool exp>
    ::= <quantifier> <range var name> ( <bool exp> )

<quantifier>
    ::= EXISTS | FORALL

<relation op inv>
    ::= <proto tuple> [ WHERE <bool exp> ] !
```

В реляционной алгебре, рассмотренной в главе 7, параметр <*relation op inv*> представлял собой одну из форм параметра <*relation exp*>, однако здесь, как уже было указано, он определяется иначе.

```
<proto tuple>
    ::- ... определение см. в тексте данной главы
```

Все ссылки на переменные области значений, помещенные непосредственно в значение параметра <*proto tuple*>, должны быть свободными в пределах данного параметра <*proto tuple*>.

Примечание. Выражение <*proto tuple*> является сокращением от "prototype tuple" *{кортеж-прототип);* Этот термин — удачный, но не стандартный.

Переменные области значений

Приведем несколько примеров определения переменных области значений (которые выражены, как обычно, в контексте базы данных поставщиков и деталей).

```
RANGEVAR SX RANGES OVER S
; RANGEVAR SY RANGES OVER
S ; RANGEVAR SPX RANGES
OVER SP ; RANGEVAR SPY
RANGES OVER SP ; RANGEVAR
PX RANGES OVER P ;

RANGEVAR SU RANGES OVER
       ( SX WHERE SX.CITY = 'London' ) ,
       ( SX WHERE EXISTS SPX ( SPX.S# = SX.S# AND
                               SPX.P# = P# ('P1') ) ) ;
```

В последнем примере переменная области значений SU принимает значения из объединения множества кортежей поставщиков, находящихся в Лондоне, и множества кортежей поставщиков детали с номером P1. Обратите внимание, что в определении переменной области значений SU используются переменные области значений sx и SPX. Следует также отметить, что в подобных определениях переменных, основанных на объединении отношений, объединяемые отношения, безусловно, должны быть совместимыми по типу.

Примечание. Переменные области значений не являются переменными в обычном смысле (как в языках программирования); они являются переменными в *логическом* смысле. В действительности они в значительной мере аналогичны параметрам предикатов, которые рассматривались в главе 3. Различие состоит в том, что параметры, описанные в главе 3, представляют значения из некоторого домена (независимо от определения этого домена), а переменные области значений, применяемые в исчислении кортежей, представляют именно кортежи.

Далее в этой главе предполагается, что приведенные выше формулировки с описанием переменных области значений остаются в силе. Следует отметить, что в реальном языке должны применяться конкретные правила, касающиеся пределов действия таких описаний. В настоящей главе этот вопрос в основном не рассматривается (за исключением раздела, посвященного языку SQL).

Свободные и связанные переменные области значений

Каждая ссылка на переменную области значений (в некотором контексте, в частности в некоторой правильно построенной формуле) является либо **свободной,** либо **связанной.** Сначала поясним это утверждение в чисто синтаксических терминах в данном подразделе, затем продолжим обсуждение его семантического значения в следующих подразделах.

Пусть v — переменная области значений, а p и q — правильно построенные формулы. Тогда имеем следующее.

- Ссылки на переменную v в правильно построенной формуле типа NOT p свобод ны или связаны в зависимости от того, свободны или связаны они в формуле p. Ссылки на переменную V в правильно построенной формуле типа (p AND q) и (p OR q) свободны или связаны, соответственно, в зависимости от того, свобод ны или связаны они в формулахр и q.
- Ссылки на переменную V, которые являются свободными в правильно построен ной формуле p, связаны в правильно построенных формулах типа EXISTS V (p) и FORALL V(p). Другие ссылки на переменные области значений в формуле p яв ляются свободными или связанными в правильно построенных формулах типа EXISTS V(p) и FORALL v(p) в соответствии с тем, свободны или связаны они в формуле p.

Для полноты необходимо добавить следующие замечания.

- Единственная ссылка на переменную V в значении параметра <*range var name*> V является свободной в пределах этого параметра <*range var name*>.
- Единственная ссылка на переменную V в значении параметра <*range attribute ref*> V. A является свободной в пределах этого параметра <*range a t tribu te ref*>.

- Если ссылка на переменную V является свободной в некотором выражении exp, то эта ссылка будет также свободной в любом выражении exp", непосредственно содержащем выражение exp как подвыражение, если только в выражении exp' не введен квантор, под действием которого эта ссылка становится связанной.

Приведем несколько примеров правильно построенных формул, содержащих ссылки на переменные области значений.

- *Простые сравнения*

```
SX.S# = S# {'S1')
SX.S# = SPX.S#
SPX.P# ≠ PX.P#
```

Здесь все ссылки на переменные SX, PX и SPX являются свободными.

- *Простые операции сравнения, объединенные с помощью логических выражений*

```
PX.WEIGHT < WEIGHT ( 15.5 ) AND PX.CITY = 'Oslo'
NOT ( SX.CITY = 'London' )
SX.S# = SPX.S# AND SPX.P# ≠ PX.P#

PX.COLOR = COLOR ('Red') OR PX.CITY = 'London'
```

Здесь также все ссылки на переменные sx, PX и SPX являются свободными.

- *Правильно построенные формулы с кванторами*

```
EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# =
P# ('P2') ) FORALL PX ( PX.COLOR = COLOR
('Red') )
```

В этих примерах ссылки на переменные SPX и PX являются связанными, а ссылка на переменную SX остается свободной. Подробнее данные примеры описаны ниже, в подразделе "Кванторы".

Кванторы

Существует два квантора[2]: EXISTS и FORALL. Квантор EXISTS является квантором **существования,** а FORALL — квантором **всеобщности.** По сути, если выражение p — правильно построенная формула, в которой переменная v свободна, то выражения

```
EXISTS V ( p
) И
FORALL V ( p )
```

также являются допустимыми правильно построенными формулами, но переменная V в них обеих связана. Первая формула означает следующее: **"Существует по меньшей мере одно значение** переменной V, при котором формула p становится *истинной".* Вторая формула означает следующее: "Формула p является *истинной* **при всех значениях** переменной V". Предположим, например, что переменная V принимает значения из множества "Члены

[2] Термин "квантор" происходит от латинского слова "quantum", которое, вообще говоря, означает "количество". Вместо ключевых слов EXISTS и FORALL, соответственно, часто используются символы 3 ("обратное E") и V ("перевернутое A").

сената США в 2003 году", и предположим также, что выражение p — следующая правильно построенная формула: "V— женщина" (разумеется, мы не пытаемся использовать здесь формальный синтаксис). Тогда выражение EXISTS V(p) будет допустимой правильно построенной формулой, имеющей значение TRUE *{истина);* выражение FORALL v(p) также будет допустимой правильно построенной формулой, но ее значение будет равно FALSE *{ложь),* поскольку не все члены сената— женщины.

Теперь рассмотрим квантор существования EXISTS более внимательно. Еще раз обратимся к примеру, приведенному в конце предыдущего раздела.

```
EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
```

Из приведенных выше рассуждений следует, что эта правильно построенная формула может быть прочитана следующим образом.

> *В текущем значении переменной отношения SP существует кортеж (скажем, SPX), такой, что значение атрибута S# в этом кортеже равно значению атрибута SX. S# (каким бы оно ни было), а значение атрибута P# в кортеже SPX равно P2.*

Каждая ссылка на переменную SPX в этом примере является связанной. Единственная ссылка на переменную SX свободна.

Формально квантор существования EXISTS определяется как **повторно применяемая операция OR** (ИЛИ). Другими словами, если, во-первых, r — это отношение с кортежами tl, t2, …, tm, во-вторых, V—это переменная области значений, принимающая значения из данного отношения, и, в третьих, p(V) — это правильно построенная формула, в которой переменная V используется как свободная переменная, то правильно построенная формула вида

```
EXISTS V ( p ( V ) )
```

эквивалентна следующей правильно построенной формуле.

```
FALSE OR p ( tl ) OR ... OR p ( tm )
```

В частности, следует отметить, что если отношение R пустое (т.е. т=0), то данное выражения принимает значение FALSE.

Рассмотрим в качестве примера отношение r, содержащее только следующие кортежи (для упрощения здесь не соблюдаются наши обычные требования к синтаксису).

```
( 1, 2,
3 ) ( 1,
2, 4 ) (
1, 3, 4
)
```

Предположим, что три атрибута, идущие по порядку слева направо, как показано выше, имеют имена, соответственно, A, в и с и каждый из этих атрибутов имеет тип INTEGER. Тогда приведенные ниже правильно построенные формулы будут иметь указанные значения.

```
EXISTS V ( V.C > 1 )          : TRUE
EXISTS V ( V.B > 3 )          : FALSE
EXISTS V ( V.A > 1 OR V.C = 4 ) : TRUE
```

Теперь рассмотрим квантор всеобщности FORALL, ДЛЯ чего вернемся к соответствующему примеру, приведенному в конце предыдущего раздела.

```
FORALL PX ( PX.COLOR = COLOR ('Red') )
```

Эта правильно построенная формула может быть прочитана следующим образом.

Для всех кортежей PX, скажем, в текущем значении переменной отношения P, значение атрибута COLOR в кортеже PX равно Red.

Обе ссылки на переменную PX в этом примере связаны.

Подобно тому, что квантор EXISTS был определен как результат многократного применения операции OR, квантор существования FORALL определяется как результат **многократного применения операции AND** (И). Другими словами, если обозначения r, V и p (V) имеют тот же смысл, что и в приведенном выше определении квантора EXISTS, то правильно построенная формула вида

FORALL V (p (V))

равносильна следующей правильно построенной формуле.

TRUE AND p (t1) AND . . . AND p (tm)

В частности, следует отметить, что если отношение r пустое, то данное выражение принимает значение TRUE. В качестве примера рассмотрим отношение R, содержащее те же кортежи, что и в предыдущем примере, касающемся квантора EXISTS. Тогда приведенные ниже выражения будут иметь указанные значения.

```
FORALL V ( V.A > 1 )           : FALSE
FORALL V ( V.B > 1 )           : TRUE
FORALL V ( V.A = 1 AND V.C > 2 ) : TRUE
```

Примечание. Оба квантора поддерживаются просто для удобства. С точки зрения логики нет необходимости поддерживать оба квантора, так как приведенное ниже тождество показывает, что любой из них может быть определен в терминах другого.

```
FORALL V ( p ) = NOT EXISTS V ( NOT p )
```

Неформально говоря, выражение "все значения V удовлетворяют формуле p" есть не что иное, как и выражение "нет таких значений V, которые бы не удовлетворяли формуле p". Например, (истинное) утверждение "Для любого целого x существует целое y, такое, что y > x" равносильно утверждению "Не существует целого x, такого, что не существует целого y, такого, что y > x" (иначе говоря, не существует наибольшего целого числа). Однако одни утверждения проще выразить в терминах квантора FORALL, а другие — в терминах квантора EXISTS; вернее, если один из кванторов недоступен, то иногда возникает необходимость использовать двойное отрицание (как показано в предыдущем примере). Поэтому с точки зрения практики желательно, чтобы поддерживались оба квантора.

Дополнительные сведения о свободных и связанных переменных

Предположим, что переменная x принимает значения из множества всех целых чисел, и рассмотрим следующую правильно построенную формулу.

EXISTS x (x > 3.)

Обратите внимание на то, что связанная переменная x в этой правильно построенной формуле является в определенном смысле *фиктивной.* Она применяется лишь для того, чтобы связать логическое выражение в круглых скобках с внешним квантором. В этой правильно построенной формуле просто утверждается, что существует целое число (скажем, x), которое больше 3. *Следовательно, значение этой правильно построенной формулы осталось бы полностью неизменным, если бы все ссылки на* x *были заменены ссылками на некоторую другую переменную (скажем, y).* Другими словами, правильно построенная формула

```
EXISTS y (y > 3)
```

семантически идентична формуле, приведенной ранее.

Теперь рассмотрим другую формулу WFF.

```
EXISTS x ( x > 3 ) AND x < O
```

Здесь есть три ссылки на переменную x, *обозначающие две различные переменные.* Первые две ссылки связаны и могли бы быть заменены ссылкой на другую переменную y без изменения общего смысла формулы. Третья ссылка на переменную x свободна и не может быть беспрепятственно изменена. Таким образом, из двух приведенных ниже формул WFF первая эквивалентна рассматриваемой выше формуле, а вторая— нет.

```
EXISTS У ( У > 3 ) AND x
< O EXISTS y ( y > 3 )
AND y < O
```

Кроме того, заслуживает внимания то, что окончательное значение первоначальной правильно построенной формулы не может быть определено, если не известно значение, указанное с помощью ссылки на свободную переменную x. В отличие от этого, правильно построенная формула, в которой все ссылки на переменные являются связанными, всегда имеет определенное значение, TRUE ИЛИ FALSE.

Дополнительная терминология. Правильно построенная формула, в которой все переменные связаны, называется **закрытой правильно построенной формулой. Открытая правильно построенная формула—** это формула, которая не является закрытой, т.е. такая формула, которая содержит по крайней мере одну ссылку на свободную переменную. Другими словами, используя терминологию, введенную в главе 3, закрытая правильно построенная формула — это высказывание, а открытая правильно построенная — это предикат, который не является высказыванием. (Кстати, следует отметить, что *высказывание*— тоже предикат, вернее, вырожденный частный случай предиката, в котором множество формальных параметров является пустым.)

Реляционные операции

Параметр *<relation op inv>* не совсем уместен в контексте исчисления — более подходящим вариантом был бы параметр *<relation def>*. Однако мы будем использовать именно первый вариант для согласованности с обозначениями в главе 7. В качестве напоминания приведем его синтаксис.

```
<relation op inv>
    ::= <proto tuple> [ WHERE <bool exp> ]

<proto tuple>
    ::= ... определение см. в тексте данной главы
```

Напоминаем также, что следующие синтаксические правила теперь несколько упрощены.

- Все ссылки на переменные области значений в параметре *<proto tuple>* должны быть свободными в пределах значения этого параметра с обозначением кортежа-прототипа.
- Ссылка на переменную области значений в конструкции WHERE может быть свободной только в том случае, если ссылка на эту же переменную (обязательно свободная) присутствует в соответствующем значении параметра *<proto tuple>*.

Например, следующее выражение является допустимым значением параметра *<relation op inv>* ("Определить номера поставщиков, находящихся в Лондоне").

```
SX.S# WHERE SX.CITY = 'London'
```

Здесь ссылка на переменную SX в кортеже-прототипе является свободной. Ссылка на переменную SX в конструкции WHERE также свободна, поскольку ссылка на ту же переменную области значений (обязательно свободную) имеется и в значении параметра *<proto tuple>* этого выражения.

Приведем другой пример ("Получить имена поставщиков детали с номером P2"; см. обсуждение квантора существования EXISTS в подразделе "Кванторы" этого раздела).

```
SX.SNAME WHERE EXISTS SPX ( SPX.S# = SX.S# AND
                            SPX.P# = P# ('P2') )
```

Здесь все ссылки на переменную SX являются свободными, тогда как все ссылки на переменную SPX (в конструкции WHERE) являются связанными, как и должно быть, поскольку на них нет ссылок в кортеже-прототипе.

Интуитивно ясно, что результатом выполнения операции, заданной параметром *<relation op inv>,* будет отношение, содержащее все возможные значения кортежей, определяемых параметром *<proto tuple>,* для которых результат вычисления логического выражения, заданного в конструкции WHERE параметром *<bool exp>,* принимает значение TRUE. (ЕСЛИ конструкция WHERE опущена, это эквивалентно указанию выражения WHERE TRUE.) Сделаем некоторые уточнения.

- Прежде всего, кортеж-прототип — это разделенный запятыми список элементов, заключенный в фигурные скобки (но фигурные скобки могут быть опущены, если разделенный запятыми список содержит только один элемент), каждый элемент которого является либо ссылкой на атрибут области значений (которая может включать конструкцию AS для введения нового имени атрибута), либо просто именем переменной области значений. Есть и другие варианты, но для упрощения ограничимся только этими двумя, по крайней мере, до тех пор, пока не будет указано иное. Тем не менее, отметим следующее.

  а) В этом контексте имя переменной области значений чаще всего является просто сокращенным обозначением разделенного запятыми списка ссылок на атрибуты, по одной для каждого атрибута того отношения, из которого принимает значения данная переменная области значений.

  б) Ссылка на атрибут области значений без конструкции AS по сути является сокращенным обозначением ссылки, включающей такую конструкцию, в которой новое имя атрибута совпадает со старым.

Следовательно, без потери общности кортеж-прототип можно рассматривать как разделенный запятыми список ссылок на атрибуты в виде vi . Aj AS Bj, заключенный в фигурные скобки. Обратите внимание, что не все ссылки Vi, по-видимому, будут различными, и не обязательно должны отличаться друг от друга все ссылки Aj, а ссылки в j *должны* быть разными.

- Пусть V1, V2, ..., Vm — различные переменные области значений, присутствующие в кортеже-прототипе, областями значений которых являются, соответственно, отношения rl, r2, ..., rm. Допустим, что r 1', r2', …, rm' — это новые отношения, полученные после переименования атрибутов в конструкции AS, а r' — это декартово произведение отношений r 1', r2', …, rm'.
- Пусть отношение r — это сокращение отношения r', удовлетворяющее правильно построенной формуле в конструкции WHERE.

  Примечание. Здесь предполагается, что на предыдущем шаге были также переименованы атрибуты, упоминающиеся в конструкции WHERE; в противном случае правильно построенная формула в конструкции WHERE может не иметь смысла. Но, как показано в следующем разделе, в предложенном здесь конкретном синтаксисе неоднозначность устраняется не на основании этого предположения, а с использованием уточняющей точечной системы обозначений.
- Конечное значение реляционной операции, заданной параметром <*relation op inv*>, определяется как проекция отношения r по всем заданным атрибутам Bj.

В следующем разделе будет приведено несколько примеров подобных выражений.

8.3. ПРИМЕРЫ

Представляем несколько примеров использования реляционного исчисления кортежей для формулирования запросов. В качестве упражнения рекомендуется решить эти задачи средствами реляционной алгебры (для сравнения и сопоставления). В некоторых случаях примеры являются повторением примеров из главы 7 (эти случаи обозначены специально).

8.3.1. Определить номера поставщиков из Парижа со статусом, большим 20

```
{ SX.S#, SX.STATUS }
WHERE SX.CITY = 'Paris' AND SX.STATUS > 20
```

8.3.2. Найти все пары номеров таких поставщиков, которые находятся в одном городе (повторение примера 7.5.5)

```
{ SX.S# AS SA, SY.S# AS SB }
              WHERE SX.CITY = SY.CITY AND SX.S# < SY.S#
```

Обратите внимание, что конструкции AS в кортеже-прототипе используются для присваивания имен атрибутам *результата.* Следовательно, эти имена недоступны для использования в конструкции WHERE, и потому вторая операция сравнения в конструкции WHERE записана как sx.s# < SY.S#, а не в виде SA < SB.

8.3.3. Получить полную информацию о поставщиках детали с номером P2 (модифицированная версия примера 7.5.1)

```
SX WHERE EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
```

Обратите внимание на использование имени переменной области значений в кортеже-прототипе. Этот пример является сокращенной записью следующего выражения.

```
{ SX.S#, SX.SNAME, SX.STATUS, SX.CITY }
  WHERE EXISTS SPX ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
```

8.3.4. Определить имена поставщиков по крайней мере одной детали красного цвета (повторение примера 7.5.2)

```
SX.SNAME
WHERE EXISTS SPX ( SX.S# = SPX.S# AND .
                EXISTS PX ( PX.P# = SPX.P# AND
                            PX.COLOR = COLOR ('Red') ) )
```

Эквивалентная формула (но записанная в **предваренной нормальной форме,** в которой все кванторы помещаются в начало правильно построенной формулы) имеет следующий вид.

```
SX.SNAME
WHERE EXISTS SPX ( EXISTS PX ( SX.S# = SPX.S# AND
                               SPX.P#   =   PX.P#   AND
                               PX.COLOR =  COLOR  ('Red')
                               ) )
```

Предваренная (или *пренексная)* нормальная форма по своей сути не является более (или менее) правильной по сравнению с другими формами, но после ее усвоения можно убедиться, что в большинстве случаев она представляет собой наиболее естественный метод формирования запросов. Кроме того, она позволяет уменьшить количество используемых скобок, как показано ниже. Например, рассмотрим следующую правильно построенную формулу.

```
Q1  V1  ( Q2  V2  ( wff ) )
```

Здесь каждый из кванторов Q1 и Q2 представляет собой или квантор EXISTS, или квантор FORALL. При необходимости эту формулу всегда можно однозначно привести к следующему виду.

```
Q1  V1  Q2  V2  ( wff )
```

Таким образом, приведенное выражение реляционного исчисления можно переписать следующим образом.

```
SX.SNAME
WHERE EXISTS SPX EXISTS PX ( SX.S# = SPX.S# AND
                             SPX.P#   =   PX.P#   AND
                             PX.COLOR     =     COLOR
                             ('Red') )
```

Однако для ясности во всех остальных примерах будем по-прежнему показывать все скобки.

8.3.5. Найти имена поставщиков по крайней мере одной детали, поставляемой поставщиком с номером S2

```
SX.SNAME
WHERE EXISTS SPX ( EXISTS SPY ( SX.S# = SPX.S# AND
                                SPX.P# = SPY.P# AND
                                SPY.S# = S# ('S2') )
                                )
```

8.3.6. Получить имена поставщиков всех типов деталей (повторение примера 7.5.3)

```
SX.SNAME WHERE FORALL PX ( EXISTS SPX ( SPX.S# = SX.S# AND
                                       SPX.P# = PX.P# ) )
```

Эквивалентное выражение можно записать без использования КВаНТораFORALL.

```
SX.SNAME WHERE NOT EXISTS PX ( NOT EXISTS SPX
                                    ( SPX.S# = SX.S# AND
                                      SPX.P# = PX.P# ) )
```

8.3.7. Определить имена поставщиков, которые не поставляют деталь с номером P2 (повторение примера 7.5.6)

```
SX.SNAME WHERE NOT EXISTS SPX
                       ( SPX.S# = SX.S# AND SPX.P# = P# ('P2') )
```

Обратите внимание, как просто это решение можно получить из решения примера 8.3.3.

8.3.8. Определить номера поставщиков, по крайней мере, тех деталей, которые поставляет поставщик с номером S2 (повторение примера 7.5.4)

```
SX.S# WHERE FORALL SPX ( SPX.S# ≠ S# ('S2') OR
                         EXISTS SPY ( SPY.S# = SX.S#
                                      AND SPY.P# =
                                      SPX.P# ) )
```

Переформулируем этот запрос в соответствии со следующим выражением: "Получить номера таких поставщиков sx, что для всех поставок детали SPX, независимо то того, выполнена ли эта поставка поставщиком S2 или нет, существует поставка SPY детали SPX поставщиком sx". Чтобы упростить формулировку таких сложных запросов, как этот, введем другое синтаксическое соглашение, называемое явной синтаксической формой для оператора **логической импликации.** Если p и q— правильно построенные формулы, то выражение логической импликации вида

IF p THEN q END IF

также будет правильно построенной формулой с семантикой, идентичной семантике следующей формулы.

(NOT p) OR q

Таким образом, приведенное выше выражение может быть переписано следующим образом.

```
SX.S# WHERE FORALL SPX ( IF SPX.S# = S# ('S2') THEN
                         EXISTS SPY ( SPY.S# = SX.S# AND
                                      SPY.P# = SPX.P#
                       ) END IF )
```

Дадим словесную формулировку этого запроса: "Получить номера таких поставщиков SX, что для всех поставок SPX, если существует поставка SPX поставщиком с номером S2, то существует поставка SPY всех типов деталей, входящих в поставку SPX, поставщиком sx".

8.3.9. Получить номера деталей, которые весят более 16 фунтов, поставляются поставщиком с номером S2 или соответствуют обоим условиям

```
RANGEVAR PU RANGES OVER
       ( PX.P# WHERE PX.WEIGHT > WEIGHT ( 16.0 ) ),
       ( SPX.P# WHERE SPX.S# = S# ('S2') )
; PU.P#
```

В эквивалентном выражении реляционной алгебры здесь могло бы использоваться явное объединение.

Ради интереса покажем альтернативную формулировку этого запроса. Однако эта вторая формулировка (в отличие от первой) опирается на тот факт, что каждый номер детали из переменной отношения SP появляется также в переменной отношения P.

```
PX.P# WHERE PX.WEIGHT > WEIGHT ( 16.0 )
 OR EXISTS SPX ( SPX.P# = PX.P# AND
 SPX.S# = S# ('S2') )
```

8.4. СРАВНИТЕЛЬНЫЙ АНАЛИЗ РЕЛЯЦИОННОГО ИСЧИСЛЕНИЯ И РЕЛЯЦИОННОЙ АЛГЕБРЫ

В начале этой главы утверждалось, что реляционная алгебра и реляционное исчисление в своей основе эквивалентны. Обсудим это утверждение более подробно. Вначале Кодд в [7.1] показал, что алгебра является, по меньшей мере, столь же мощной, как и исчисление. Для этой цели он предложил алгоритм, получивший название *алгоритма редукции Кодда,* с помощью которого любое выражение исчисления можно преобразовать в семантически эквивалентное выражение алгебры. Мы не станем приводить здесь этот алгоритм полностью, а ограничимся довольно сложным примером, иллюстрирующим в общих чертах, как он функционирует[3].

В качестве основы для нашего примера используется не привычная база данных поставщиков и деталей, а ее расширенная версия, упоминавшаяся в упражнениях главы 4 и в других главах. Для удобства на рис. 8.1 приведен пример возможных значений для этой базы данных (это — копия рис. 4.5 из главы 4).

---

[3] В действительности алгоритм, представленный в [7.1], содержит небольшую ошибку [8.2]. Более того, определенная в этой статье версия реляционного исчисления не включает аналог оператора объединения, следовательно, исчисление Кодда является строго менее мощным, чем его алгебра. Как бы там ни было, но утверждение о том, что алгебра и исчисление, включающее аналог операции объединения, эквивалентны, является истинным, и это доказано многими авторами (см., например, [7.11]).

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P

P#	PNAME	COLOR	WEIGHT	CITY
P1	Nut	Red	12.0	London
P2	Bolt	Green	17.0	Paris
P3	Screw	Blue	17.0	Oslo
P4	Screw	Red	14.0	London
P5	Cam	Blue	12.0	Paris
P6	Cog	Red	19.0	London

J

J#	JNAME	CITY
J1	Sorter	Paris
J2	Display	Rome
J3	OCR	Athens
J4	Console	Athens
J5	RAID	London
J6	EDS	Oslo
J7	Tape	London

SPJ

S#	P#	J#	QTY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J1	400
S2	P3	J2	200
S2	P3	J3	200
S2	P3	J4	500
S2	P3	J5	600
S2	P3	J6	400
S2	P3	J7	800
S2	P5	J2	100
S3	P3	J1	200
S3	P4	J2	500
S4	P6	J3	300
S4	P6	J7	300
S5	P2	J2	200
S5	P2	J4	100
S5	P5	J5	500
S5	P5	J7	100
S5	P6	J2	200
S5	P1	J4	100
S5	P3	J4	200
S5	P4	J4	800
S5	P5	J4	400
S5	P6	J4	500

Рис. 8.1. База данных поставщиков, деталей и проектов (значения даны для примера)

Рассмотрим теперь следующий запрос: "Получить имена поставщиков и названия городов, в которых находятся поставщики деталей, поставляющие детали по крайней мере для одного проекта в Афинах (Athens) в количестве не меньше 50 деталей каждого типа". Выражение реляционного исчисления для этого запроса приведено ниже.

```
{ SX.SNAME, SX.CITY } WHERE EXISTS JX FORALL PX EXISTS SPJX
                                ( JX.CITY = 'Athens'
                                 AND JX.J# = SPJX.J#
                                 AND PX.P# = SPJX.P#
                                 AND SX.S# = SPJX.S#
                                 AND SPJX.QTY > QTY (
                                 50 ) )
```

Здесь SX, PX, JX и SPJX — переменные области значений, принимающие значения, соответственно, из переменных отношения S, P, J и SPJ. Теперь покажем, как можно вычислить это выражение, чтобы достичь требуемого результата.

Этап 1. Для каждой переменной области значений выполним выборку ее области значений (т.е. множества всех значений переменной), по возможности, с помощью операции сокращения. Под выражением "по возможности, с помощью операции сокращения" подразумевается, что может существовать простое условие операции сокращения (определение этого термина приведено в главе 7), встроенное в конструкцию WHERE, которую можно использовать, чтобы сразу исключить из рассмотрения некоторые кортежи. В нашем случае осуществляется выборка следующих множеств кортежей.

- SX. Все кортежи отношения s — 5 кортежей.
- PX. Все кортежи отношения p — 6 кортежей.
- JX. Кортежи отношения J, в которых CITY = ' Athens'—2 кортежа. .
- SPJX. Кортежи отношения SPJ, в которых QTY ≥ QTY (50) — 24 кортежа.

Этап 2. Строим декартово произведение диапазонов, выбранных на первом этапе. Результат представлен ниже.

S#	SN	STATUS	CITY	P#	PN	COLOR	WEIGHT	CITY	J#	JN	CITY	S#	P#	J#	QTY
S1	Sm	20	Lon	P1	Nt	Red	12.0	Lon	J3	OR	Ath	S1	P1	J1	200
S1	Sm	20	Lon	P1	Nt	Red	12.0	Lon	J3	OR	Ath	S1	P1	J4	700
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...

(И т.д.) Все произведение содержит 5*6*2*24 = 1 440 кортежей.

Примечание. Для экономии места здесь это отношение полностью не приводится. Мы также не переименовывали атрибуты (хотя это следовало бы сделать во избежание двусмысленности), а просто расположили их в таком порядке, чтобы было видно, какой атрибут s# относится, например, к отношению S, а какой — к отношению SPJ. Это также сделано для сокращения изложения.

Этап 3. Применяем операцию сокращения к сформированному на этапе 2 произведению в соответствии с "частью конструкции WHERE, относящейся к соединению". В нашем примере эта часть выглядит следующим образом.

```
JX.J# = SPJX.J# AND PX.P# = SPJX.P# AND SX.S# = SPJX.S#
```

Поэтому из произведения исключаются кортежи, для которых значение атрибута s# из отношения поставщиков не равно значению атрибута s# из отношения поставок, или значение атрибута P# из отношения деталей не равно значению атрибута P# из отношения поставок, или значение атрибута J# из отношения проектов не равно значению атрибута J# из отношения поставок. Затем получаем подмножество декартова произведения, состоящее (как оказалось) только из десяти кортежей.

S#	SN	STATUS	CITY	P#	PN	COLOR	WEIGHT	CITY	J#	JN	CITY	S#	P#	J#	QTY
S1	Sm	20	Lon	P1	Nt	Red	12.0	Lon	J4	Cn	Ath	S1	P1	J4	700
S2	Jo	10	Par	P3	Sc	Blue	17.0	Osl	J3	OR	Ath	S2	P3	J3	200
S2	Jo	10	Par	P3	Sc	Blue	17.0	Osl	J4	Cn	Ath	S2	P3	J4	200
S4	Cl	20	Lon	P6	Cg	Red	19.0	Lon	J3	OR	Ath	S4	P6	J3	300
S5	Ad	30	Ath	P2	Bt	Green	17.0	Par	J4	Cn	Ath	S5	P2	J4	100
S5	Ad	30	Ath	P1	Nt	Red	12.0	Lon	J4	Cn	Ath	S5	P1	J4	100
S5	Ad	30	Ath	P3	Sc	Blue	17.0	Osl	J4	Cn	Ath	S5	P3	J4	200
S5	Ad	30	Ath	P4	Sc	Red	14.0	Lon	J4	Cn	Ath	S5	P4	J4	800
S5	Ad	30	Ath	P5	Cm	Blue	12.0	Par	J4	Cn	Ath	S5	P5	J4	400
S5	Ad	30	Ath	P6	Cg	Red	19.0	Lon	J4	Cn	Ath	S5	P6	J4	500

(Это отношение представляет собой наглядный пример соединения по эквивалентности.)

Этап 4. Применяем кванторы в порядке справа налево следующим образом.

- Для квантора EXISTS V (где v — переменная области значений, принимает значения из некоторого отношения r) формируем *проекцию* текущего промежуточного результата, чтобы исключить все атрибуты отношения r.
- Для квантора FORALL V *делим* текущий промежуточный результат на отношение "с областью значений, полученной с помощью операции сокращения", соответствующее отношению V, которое было получено выше. При выполнении этой операции также будут исключены все атрибуты отношения r.j
  Примечание. Под *делением* здесь подразумевается оригинальная операция деления
  Кодда (см. аннотацию к [7.4]).
  В нашем примере имеем следующие кванторы.

```
EXISTS JX FORALL PX EXISTS  SPJX
```

Таким образом, затем выполняются следующие операции.

- EXISTS SPJX. Исключение с помощью операции проекции атрибутов переменной отношения SPJ (SPJ.S#, SPJ.P#, SPJ. J# и SPJ.QTY). В результате получаем следующее,

S#	SN	STATUS	CITY	P#	PN	COLOR	WEIGHT	CITY	J#	JN	CITY
S1	Sm	20	Lon	P1	Nt	Red	12.0	Lon	J4	Cn	Ath
S2	Jo	10	Par	P3	Sc	Blue	17.0	Osl	J3	OR	Ath
S2	Jo	10	Par	P3	Sc	Blue	17.0	Osl	J4	Cn	Ath
S4	Cl	20	Lon	P6	Cg	Red	19.0	Lon	J3	OR	Ath
S5	Ad	30	Ath	P2	Bt	Green	17.0	Par	J4	Cn	Ath
S5	Ad	30	Ath	P1	Nt	Red	12.0	Lon	J4	Cn	Ath
S5	Ad	30	Ath	P3	Sc	Blue	17.0	Osl	J4	Cn	Ath
S5	Ad	30	Ath	P4	Sc	Red	14.0	Lon	J4	Cn	Ath
S5	Ad	30	Ath	P5	Cm	Blue	12.0	Par	J4	Cn	Ath
S5	Ad	30	Ath	P6	Cg	Red	19.0	Lon	J4	Cn	Ath

- FORALL PX. Деление полученного результата на отношение P. В результате имеем следующее.

S#	SNAME	STATUS	CITY	J#	JNAME	CITY
S5	Adams	30	Athens	J4	Console	Athens

- EXISTS JX. Исключение с помощью операции проекции атрибутов отношения J (J. J#, J. JNAME и J. CITY). В результате получаем следующее.

S#	SNAME	STATUS	CITY
S5	Adams	30	Athens

Этап 5. Получение проекции результата этапа 4 в соответствии со спецификациями в кортеже-прототипе. В нашем примере кортеж-прототип имеет следующий вид.

(SX.SNAME, SX.CITY)

Значит, конечный результат вычислений будет таков.

SNAME	CITY
Adams	Athens

Из сказанного выше следует, что начальное выражение исчисления семантически эквивалентно определенному вложенному алгебраическому выражению, и, если быть более точным, это проекция от проекции результата деления проекции сокращения, которое применено к произведению четырех сокращений (!).

Этим мы завершаем обсуждение данного примера. Конечно, можно намного улучшить используемый алгоритм (см. главу 18, в частности, аннотацию к [18.4]). И хотя многие подробности в пояснениях были опущены, этот пример вполне адекватно отражает общую идею работы алгоритма редукции.

Кстати, теперь можно объяснить одну (не единственную) из причин того, почему Кодд определил ровно восемь алгебраических операторов. Эти восемь операторов образуют удобный **целевой язык,** который может служить средством возможной реализации реляционного исчисления. Другими словами, для заданного языка, построенного на основе реляционного исчисления (подобно языку QUEL), один из возможных подходов к реализации заключается в том, что организуется получение запроса в том виде, в каком он предоставлен пользователем. По существу, он будет являться просто выражением реляционного исчисления, к которому затем можно будет применить определенный алгоритм редукции, чтобы получить эквивалентное алгебраическое выражение. Это алгебраическое выражение, разумеется, будет включать набор алгебраических операций, которые, безусловно, будут реализуемыми по определению. (Следующий этап состоит в оптимизации полученного алгебраического выражения, о чем речь пойдет в главе 18.)

Также следует отметить, что восемь алгебраических операторов Кодда являются мерой оценки выразительной мощи любого языка баз данных. Это обстоятельство уже кратко упоминалось в главе 7, в конце раздела 7.6, а сейчас пришло время обсудить его подробнее.

Прежде всего, отметим, что некоторый язык принято называть **реляционно полным,** если он по своим возможностям, по крайней мере, не уступает реляционному исчислению. Иначе говоря, для этого необходимо, чтобы любое отношение, которое можно определить с помощью реляционного исчисления, можно было определить и с помощью некоторого выражения рассматриваемого языка [7.1]. (В главе 7 отмечалось, что "реляционно полный" означает "не уступающий по возможностям реляционной алгебре", а не исчислению, но, как читатель вскоре убедится, это одно и то же. По сути, из самого существования алгоритма редукции Кодда немедленно следует, что реляционная алгебра обладает реляционной полнотой.)

Реляционную полноту можно рассматривать как основную меру выразительной мощи языков баз данных в самом общем случае. В частности, так как реляционное исчисление и реляционная алгебра обладают реляционной полнотой, они могут служить основой для проектирования не уступающих им по выразительности языков *без необходимости явно*

прибегать к использованию циклов. Это замечание особенно важно, если язык предназначается для конечных пользователей, хотя оно также существенно, если язык предназначается для использования прикладными программистами.

Далее, поскольку алгебра обладает реляционной полнотой, для доказательства того, что некоторый язык L также обладает реляционной полнотой, достаточно показать, что в языке L есть аналоги всех восьми алгебраических операций (на самом деле достаточно показать, что в нем есть аналоги пяти примитивных операций) и что операнды любой операции языка L могут быть представлены произвольными выражениями языка L (соответствующего типа). Язык SQL — это пример языка, реляционную полноту которого можно доказать описанным способом (упр. 8.9). Язык QUEL — еще один пример подобного языка. В действительности на практике часто проще показать то, что в языке есть эквиваленты операций реляционной алгебры, чем то, что в нем существуют эквиваленты выражений реляционного исчисления. Именно поэтому реляционная полнота обычно определяется в терминах алгебраических выражений, а не в терминах выражений реляционного исчисления.

При этом важно понимать, что реляционная полнота не обязательно влечет за собой полноту какого-либо другого рода. Например, желательно, чтобы язык также обеспечивал *вычислительную полноту,* т.е. позволял использовать все вычислимые функции. Вычислительная полнота — это один из факторов, побудивших ввести в реляционную алгебру операции EXTEND И SUMMARIZE (обсуждавшиеся в главе 7). В следующем разделе описано, как можно расширить реляционное исчисление, чтобы обеспечить в нем наличие аналогов этих операций.

Вернемся к вопросу об эквивалентности алгебры и исчисления. Мы на примере показали, что любое выражение исчисления можно преобразовать в его некоторый алгебраический эквивалент, а значит, алгебра, по меньшей мере, не уступает по своей мощности исчислению. Можно показать обратное: каждое выражение реляционной алгебры можно преобразовать в эквивалентное выражение реляционного исчисления, а значит, исчисление, по меньшей мере, не уступает по своей мощности реляционной алгебре. Полное доказательство этих утверждений можно найти, например, в книге Ульмана (Ullman) [8.13]. Отсюда следует, что реляционная алгебра и реляционное исчисление логически эквивалентны.

8.5. ВЫЧИСЛИТЕЛЬНЫЕ ВОЗМОЖНОСТИ

Несмотря на то, что ранее об этом не упоминалось, в определенном нами реляционном исчислении уже есть аналоги алгебраических операторов EXTEND и SUMMARIZE ПО перечисленными ниже причинам.

- Одной из допустимых форм кортежа-прототипа является параметр < *tuple se lector inv*> ("вызов селектора кортежа"), компонентами которого могут быть произвольные выражения.
- В параметре <*Ъоо1 exp*> сравниваемыми элементами могут быть произвольные выражения.
- Как было показано в главе 7, первым или единственным параметром <*agg op inv*> ("вызов агрегирующего оператора") является реляционное выражение <*relation exp*>.

Мы считаем, что здесь не следует приводить все возможные синтаксические и семантические сведения; достаточно лишь рассмотреть несколько типичных примеров (сами эти примеры также несколько упрощены).

8.5.1. Определить номера и вес в граммах всех типов деталей, вес которых превышает 10 000 г

```
{ PX.P#, PX.WEIGHT * 454 AS GMWT }
                  WHERE PX.WEIGHT * 454 > WEIGHT ( 10000.0 )
```

Обратите внимание, что спецификация AS в кортеже-прототипе (как и в примере 8.3.2) дает имя соответствующему атрибуту *результата.* Поэтому такое имя недоступно для использования в конструкции WHERE и выражение PX. WEIGHT * 454 должно быть указано в двух местах.

8.5.2. Выбрать сведения обо всех поставщиках и обозначить каждого из них литеральным значением "Supplier"

```
{ SX, 'Supplier' AS TAG }
```

8.5.3. Получить полные сведения о каждой поставке, включая общий вес поставки

```
{ SPX, PX.WEIGHT * SPX.QTY AS SHIPWT } WHERE PX.P# = SPX.P#
```

8.5.4. Для каждой детали получить номер детали и общий объем поставки в штуках

```
{ PX.P#, SIM ( SPX WHERE SPX.P# = PX.P#, QTY ) AS TOTQTY }
```

8.5.5. Определить общее количество поставляемых деталей

```
SUM ( SPX, QTY ) AS GRANDTOTAL
```

8.5.6. Для каждого поставщика получить номер поставщика и общий объем поставки в штуках

```
{ SX.S#, COUNT ( SPX WHERE SPX.S# = SX.S# ) AS #_OF_PARTS }
```

8.5.7. Указать названия таких городов, в которых хранятся детали, что в них находится больше пяти деталей красного цвета

```
RANGEVAR PY RANGES OVER P ;
PX.CITY WHERE COUNT ( PY WHERE PY.CITY = PX.CITY
                        AND PY.COLOR = COLOR ('Red') ) > 5
```

8.6. СРЕДСТВА ЯЗЫКА SQL

Как уже говорилось в разделе 8.4, реляционный язык может быть основан как на реляционной алгебре, так и на реляционном исчислении. Что же лежит в основе языка SQL? К сожалению, ответом будет "частично и то, и другое, а частично *ни то, ни другое...".*

Когда язык SQL только разрабатывался, предполагалось что он будет отличаться как от реляционной алгебры, так и от реляционного исчисления [4.8]. Действительно, именно этим мотивировалось введение в язык конструкции IN <*subquery*> (см. пример 8.6.10, приведенный ниже). Однако со временем выяснилось, что язык SQL нуждается в определенных средствах как реляционной алгебры, так и исчисления, поэтому он был расширен для включения этих функций[4]. На сегодняшний день ситуация складывается таким образом, что язык SQL в чем-то похож на реляционную алгебру, в чем-то на реляционное исчисление, а в чем-то отличается от них обоих. Таким положением дел объясняется, почему в главе 7 мы отложили обсуждение средств обработки данных языка SQL до настоящей главы. (Мы предоставляем читателю в качестве упражнения определить, какая часть языка SQL основана на алгебре, какая на исчислении, а какая ни на том, ни на другом.)

Запросы в языке SQL формулируются в виде **табличных выражений** < *table exp>,* которые в принципе могут иметь очень высокую степень сложности. Здесь мы не будем углубляться во все эти нюансы, а просто рассмотрим несколько примеров, раскрывающих наиболее важные моменты. В качестве основы для примеров взяты определения таблиц SQL для базы данных поставщиков и деталей, представленные в главе 4 (см. рис. 4.1).

8.6.1. Указать цвета деталей и названия городов для деталей, которые имеют вес свыше 10 фунтов и хранятся в городах, отличных от Парижа

```
SELECT PX.COLOR, PX.CITY
FROM   P AS PX
WHERE  PX.CITY <> 'Paris'
AND    PX.WEIGHT > WEIGHT ( 10.0 ) ;
```

Необходимо отметить следующее.

1. Как было указано в главе 5, в языке SQL в качестве оператора проверки на нера венство используется символ "о". Операторы "меньше или равно" и "больше или равно", соответственно, записываются как"< = " и ">=".
2. Спецификация P AS PX в конструкции FROM по сути представляет собой опреде ление (в стиле исчисления кортежей) переменной области значений с именем PX, областью значений которой является текущее значение таблицы P. Само имя (а не переменную!) PX принято называть **именем корреляции,** а областью его определе ния, неформально говоря, является табличное выражение, в котором появляется его определение, исключая любое внутреннее выражение, в котором определена другая переменная области значений с тем же именем (см. пример 8.6.12).

---

[4] Вследствие этого, как отмечается в аннотации к [4.19], конструкцию IN <subquery> можно полностью удалить из языка без потери его функциональности! В этом есть некоторая ирония, поскольку благодаря именно указанной конструкции в названии данного языка, в переводе означающего "язык структурированных запросов" (Structured Query Language), появилось слово "структурированный" (Structured). В действительности именно эта конструкция способствовала тому, что в первую очередь для работы с базами данных был применен язык SQL, и лишь затем реляционная алгебра и реляционное исчисление.

3. В языке SQL допускается также применение *неявно заданных* переменных области значений, что позволяет переписать приведенный выше запрос в следующем виде.

```
SELECT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > WEIGHT { 10.0 ) ;
```

   Основная идея состоит в том, что должно быть разрешено использовать имя таблицы для обозначения неявно заданной переменной области значений, которая принимает значения из рассматриваемой таблицы (разумеется, при том условии, что результаты не допускают неоднозначного толкования). Например, конструкцию FROM P в данном примере можно рассматривать как сокращенную запись конструкция FROM P AS P. Другими словами, необходимо четко понимать, что уточняющее имя P (например) в выражении P.COLOR в конструкциях WHERE и SELECT обозначает не саму таблицу P, а переменную области значений P, которая принимает свои значения из одноименной таблицы.

4. Как было отмечено в главе 4, в этом примере можно было бы вполне обойтись и без уточнителя, следующим образом.

```
SELECT COLOR, CITY
FROM P
WHERE CITY <> 'Paris'
AND WEIGHT > WEIGHT ( 10.0 ) ;
```

   Согласно общему правилу языка SQL, неутонченные имена допускаются во всех случаях, когда это не вызывает неоднозначности. Однако в наших примерах спецификаторы будут обычно (но не всегда!) использоваться и в тех случаях, когда формально они излишни. К сожалению, в определенных контекстах явно требуется, чтобы имена столбцов были *не* уточнены! Например, это требуется[5] в конструкции ORDER BY (см. следующий пример).

5. В интерактивных запросах SQL может также использоваться конструкция **ORDER BY,** уже упоминавшаяся в главе 4 в связи с объявлением DECLARE CURSOR, как показано ниже.

```
SELECT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > WEIGHT ( 10.0 )
ORDER BY CITY DESC ;  /* Следует о
```

6. Напоминаем, что допускается использование сокращения SELECT *, о котором также упоминалось в главе 4.

```
SELECT *
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > WEIGHT ( 10.0 )
```

[5] За исключением тех случаев, которые указаны в разделе 4.6 главы 4.

Символ "*" в выражении SELECT * представляет собой сокращенное обозначение разделенного запятыми списка имен всех столбцов таблицы (или таблиц), указанной в конструкции FROM. В этом списке имена столбцов находятся в том же порядке, в котором они расположены в соответствующей таблице (или таблицах). Следует отметить, что такую сокращенную запись особенно удобно использовать в интерактивных запросах, поскольку при этом уменьшается количество нажатий клавиш. Однако при использовании этой конструкции во внедренных операторах SQL (т.е. в операторах языка SQL, внедренных в программу на другом языке) существует скрытая опасность, поскольку в подобных случаях символ "*" может приобрести совсем другое значение (например, после добавления в таблицу столбца или его удаления с помощью оператора ALTER TABLE).

7. *{Более важная информация по сравнению с приведенной в предыдущих пунктах!)* Обратите внимание, что для используемого нами в примерах набора данных этот запрос будет возвращать *четыре* строки, а не две, несмотря на то, что три из них будут совершенно идентичны. В языке SQL не предусмотрено удаление излишних дублирующихся строк из результата оператора SELECT, пока пользователь явно не потребует этого с помощью ключевого слова **DISTINCT,** как показано ниже.

```
SELECT DISTINCT P.COLOR, P.CITY
FROM   P
WHERE  P.CITY <> 'Paris'
AND    P.WEIGHT > WEIGHT ( 10.0 ) ;
```

Данный вариант запроса будет возвращать уже две строки, а не четыре.

Из всего вышесказанного следует (как уже было фактически указано в главе 6), что *фундаментальным объектом данных в языке SQL является не отношение,* а скорее таблица, и таблицы SQL содержат (вообще говоря) *не множества, а мультимножества* строк (в мультимножествах допускаются повторения элементов). Таким образом, в языке SQL нарушается *информационный принцип.* Одно из следствий этого факта состоит в том, что основные операторы SQL являются не реляционными операторами в полном смысле этого слова, а аналогами реляционных операторов, предназначенных для работы с мультимножествами. Другим следствием является то, что теоремы и их побочные результаты, которые являются справедливыми в реляционной модели (например, о преобразовании выражений, [6.6]), не обязательно выполняются в языке SQL.

8.6.2. Для всех деталей указать номер детали и вес в граммах (упрощенная версия примера 8.5.1)

```
SELECT P.P#, P.WEIGHT * 454 AS
GMWT FROM   P ;
```

Спецификация AS GMWT вводит соответствующее имя столбца результата для "вычисленного столбца". Таким образом, два столбца результирующей таблицы будут называться p# и GMWT. Если бы спецификация AS GMWT была опущена, то соответствующий ; столбец остался бы фактически безымянным. Отметим, что хотя в подобных случаях правила языка SQL в действительности не требуют от пользователя указания имени результирующего столбца, в наших примерах будем их задавать всегда.

8.6.3. Получить все комбинации данных о поставщиках и деталях, находящихся в одном городе

В языке SQL существует несколько способов формулирования этого запроса. Приведем три самых простых.

```
1. SELECT S.*, P.P#, P.PNAME, P.COLOR, P.WEIGHT
   FROM   S, P
   WHERE  S.CITY = P.CITY ;
2. S JOIN P USING CITY ;
3. S NATURAL JOIN P ;
```

Результатом в каждом случае[6] будет **естественное соединение** таблиц S и P (по атрибуту города CITY).

Первая формулировка заслуживает более подробного обсуждения. Именно она, единственная из трех предложенных вариантов, допустима в первоначальной версии языка SQL (явная операция JOIN была введена в стандарте SQL: 1992). Концептуально можно рассматривать реализацию этой версии запроса следующим образом.

- Во-первых, после выполнения конструкция FROM мы получаем **декартово произведение** S TIMES P. (Строго говоря, перед вычислением произведения следовало бы позаботиться о переименовании столбцов. Для простоты мы этот вопрос не рассматриваем. Напоминаем также, что, как следует из сказанного в разделе 7.7, декартовым произведением единственной таблицы t является сама таблица t.)
- Во-вторых, после выполнения конструкция WHERE мы получаем **сокращение** этого произведения, в котором два значения атрибута CITY в каждой строке равны (иначе говоря, выполнено *соединение* таблиц поставщиков и деталей *по эквивалентности* атрибутов с обозначением города).
- В-третьих, после выполнения конструкция SELECT мы получаем **проекцию** выборки по столбцам, указанным в конструкции SELECT. Конечным результатом становится естественное соединение указанных таблиц.

Следовательно, неформально говоря, в языке SQL конструкция FROM соответствует декартову произведению, конструкция WHERE — операции сокращения, а конструкции SELECT-FROM-WHERE, вместе взятые, — проекции сокращения произведения (но, как было указано выше, рассматриваемая "проекция" не обязательно устраняет дубликаты).

8.6.4. Найти все пары названий городов, таких что поставщик, находящийся в первом городе, поставляет деталь, хранящуюся во втором городе

```
SELECT DISTINCT S.CITY AS SCITY, P.CITY AS
PCITY FROM   S JOIN SP USING S# JOIN P USING
P# ;
```

Обратите внимание, что приведенный ниже оператор является *неправильным* (объясните, почему).

---

[6] В стандарте SQL:2003, по-видимому, будет предусмотрено требование, чтобы вторая и третья формулировки включали префикс "SELECT * FROM".

```
SELECT DISTINCT S.CITY AS SCITY, P.CITY AS PCITY FROM   S
NATURAL JOIN SP NATURAL JOIN P ;
```

Ответ. Поскольку во втором соединении он включает столбец CITY как столбец, по которому выполняется соединение.

8.6.5. Получить все пары номеров поставщиков, таких что оба поставщика в каждой паре находятся в одном городе (см. пример 8.3.2)

```
SELECT A.S# AS SA, B.S# AS
SB FROM   S AS A, S AS B
WHERE  A.CITY = B.CITY AND
A.S# < B.S# ;
```

В этом примере требуется явно указывать переменные области значений. Также следует отметить, что вводимые имена столбцов SA и SB относятся к столбцам *результирующей таблицы,* и потому не могут использоваться в конструкции WHERE.

8.6.6. Определить общее количество поставщиков

```
SELECT COUNT(*) AS
N FROM   S ;
```

Результатом будет таблица с одним столбцом, которому присвоено имя N, и одной строкой, содержащей значение 5. Язык SQL поддерживает типичный набор агрегирующих функций[7]: COUNT, SUM, AVG, MAX, MIN, EVERY и ANY. Однако есть еще несколько специфических особенностей языка SQL, о которых необходимо знать пользователю.

. ■ В общем случае фактическому параметру агрегирующей функции может предшествовать необязательное ключевое слово DISTINCT (например, SUM (DISTINCT QTY)), которое указывает, что перед применением этой функции дублирующиеся строки должны быть удалены. Для функций MAX, MIN, EVERY и ANY ключевое слово DISTINCT является излишним и не вызывает никакого действия.

- Специальная агрегирующая функция COUNT(*) не допускает использования ключевого слова DISTINCT и предназначена для подсчета всех строк в таблице без предварительного удаления дублирующихся строк.
- Любые неопределенные значения в столбце фактического параметра (глава 19) удаляются перед применением агрегирующей функции независимо от того, указа но ли ключевое слово DISTINCT, кроме случая использования агрегирующей функции COUNT (*), когда неопределенные значения обрабатываются так же, как обычные значения.
- Если после удаления неопределенных значений должно остаться пустое множество, то агрегирующая функция COUNT возвращает нуль, а все остальные операторы возвращают неопределенное значение.

---

[7] EVERY - это аналог конструкции ALL в языке SQL (в этом языке ключевое слово ALL не поддерживается). Вместо ключевого слова ANY можно применить SOME. Кроме того, было введено несколько дополнительных агрегирующих операторов в связи с созданием версии SQL/OLAP, позволяющей использовать язык SQL в области так называемой "оперативной аналитической обработки" (см. главу 22).

Примечание. Этот результат является логически правильным для COUNT, но не для других операторов. Например, как было показано в разделе 8.2, оператор EVERY с точки зрения логики должен возвращать значение TRUE, если он применяется к пустому множеству.

8.6.7. Определить максимальное и минимальное количество деталей с номером P2

```
SELECT MAX ( SP.QTY ) AS MAXQ, MIN ( SP.QTY ) AS MINQ
FROM   SP
WHERE  SP.P# = P# ('P2') ;
```

Здесь обе конструкции, FROM и WHERE, фактически предоставляют часть формальных параметров для двух агрегирующих функций. Следовательно, с точки зрения логики они должны были бы записываться в скобках, заключающих аргументы. Тем не менее, данный запрос действительно должен записываться именно так, как показано выше. Этот неортодоксальный подход к синтаксису оказывает существенное отрицательное влияние на структуру, удобство использования и ортогональность[8] языка SQL. Например, одно из следствий состоит в том, что агрегирующие функции не могут быть вложенными, в результате чего такой запрос, как "Вычислить среднее значение итоговых значений количества деталей", нельзя сформулировать без громоздких выражений. Если быть точным, то следующий запрос *является неправильным.*

```
SELECT AVG ( SUM ( SP.QTY ) ) /* Предостережение! Это
неправильный запрос! */ FROM   SP ;
```

Вместо этого данный запрос следовало бы сформулировать, например, таким образом.

```
SELECT AVG ( X )
FROM ( SELECT SUM ( SP.QTY ) AS X
       FROM   SP
       GROUP  BY SP.S# ) AS POINTLESS ;
```

Назначение конструкции GROUP BY разъясняется в следующем примере, а использование вложенных подзапросов, подобных приведенным здесь в конструкции FROM, — несколько ниже.

Примечание. Спецификация AS POINTLESS бессмысленна, однако ее наличия требуют синтаксические правила языка SQL (дополнительная информация приведена в [4.20]).

8.6.8. Для каждой поставляемой детали указать номер детали и общий объем поставки в штуках (модифицированная версия примера 8.5.4)

```
SELECT SP.P#, SUM ( SP.QTY ) AS TOTQTY
FROM   SP
GROUP  BY SP.P# ;
```

---

[8] Ортогональность означает *независимость.* Язык является ортогональным, если независимые понятия сохраняют в нем свою независимость и не смешиваются между собой непонятным образом. Ортогональность весьма желательна, поскольку чем менее ортогонален язык, тем он более сложен и, как это ни парадоксально, менее мощен.

Приведенное выше выражение на языке SQL является аналогом следующего выражения реляционной алгебры.

```
SUMMARIZE SP BY { P# } ADD SUM ( QTY ) AS TOTQTY
```

Оно также является аналогом следующего выражения реляционного исчисления кортежей.

```
( SPX.P#, SUM ( SPY WHERE SPY.P# = SPX.P#, QTY ) AS TOTQTY )
```

В частности, следует отметить, что если в запросе указана конструкция GROUP BY, то выражения в конструкции SELECT должны быть **однозначными для заданной группы.** Ниже приведена альтернативная (а также в определенной степени более предпочтительная) формулировка того же запроса.

```
SELECT P.P#,        ( SELECT SUM ( SP.QTY )
                             FROM    SP
           WHERE  SP.P# = P.P# ) AS TOTQTY
FROM P ;
```

Возможность использовать подзапрос таким образом позволяет нам получить результат, который включает строки с данными о деталях, не входящих ни в одну поставку. Этого нельзя было обеспечить с помощью предыдущей формулировки, в которой использовалась конструкция GROUP BY. (Но, к сожалению, значение TOTQTY для таких деталей будет представлено в виде пустого значения, а не нуля.)

8.6.9. Определить номера всех деталей, поставляемых больше чем одним поставщиком

```
SELECT SP.P#
FROM   SP
GROUP  BY SP.P#
HAVING COUNT ( SP.S# ) > 1 ;
```

Конструкция HAVING применительно к группам выполняет такие же функции, как и конструкция WHERE применительно к строкам. Другими словами, конструкция HAVING используется для исключения групп аналогично тому, как конструкция WHERE используется для исключения отдельных строк. Выражение в конструкции HAVING должно быть однозначным для каждой группы.

8.6.10. Определить имена поставщиков детали с номером P2 (см. пример 7.5.1)

```
SELECT DISTINCT S.SNAME
FROM S
WHERE S.S# IN
    ( SELECT SP.S#
      FROM   SP     . . .
      WHERE  SP.P# = P# ('P2') ) ;
```

Пояснения. В этом примере в конструкции WHERE используется так называемый *подзапрос*. Неформально говоря, подзапрос — это выражение из конструкций SELECT-FROM-WHERE-GROUP BY-HAVING, которое вложено в другое такое же выражение. Подзапрос чаще всего используется для представления множества значений, поиск которых

осуществляется с помощью **конструкция IN с условием,** что и представлено в данном в примере. Система вычисляет запрос в целом, предварительно вычислив указанный подзапрос (по крайней мере, концептуально). Подзапрос в данном примере возвращает множество *номеров* поставщиков детали с номером P2, а именно, {S1,S2,S3,S4}. Таким образом, первоначальное выражение эквивалентно следующему, более простому.

```
SELECT   DISTINCT   S.SNAME
FROM       S
WHERE   S.S   IN   (   S#('S1)',   S#('S2'),   S#('S3'),   S#('S4')   )   ;
```

Следует отметить, что первоначальную задачу — "Получить имена поставщиков детали с номером P2" — можно столь же успешно выразить **с** помощью операции *соединения,* например, таким образом.

```
SELECT DISTINCT S.SNAME
FROM   S, SP
WHERE  S.S# = SP.S#
AND    SP.P# = P# ('P2') ;
```

8.6.11. Определить имена поставщиков по крайней мере одной детали красного цвета (см. пример 8.3.4)

```
SELECT DISTINCT
S.SNAME FROM   S WHERE
S.S# IN
    (  SELECT SP.S#
       FROM   SP
       WHERE  SP.P# IN   ( SELECT P.P# FROM   P
            WHERE  P. COLOR = COLOR ( ' Red' ) ) )
            ;
```

Подзапросы могут иметь произвольную глубину вложения. *Упражнение.* Приведите эквивалентную формулировку этого запроса с использованием операции соединения.

8.6.12. Определить номера поставщиков, имеющих статус меньше того, который в данное время является максимальным в таблице S

```
SELECT S.S#
FROM   S
WHERE  S.STATUS <
     ( SELECT MAX (
       S.STATUS ) FROM   S )
       ;
```

В этом примере используются две отдельные неявные переменные области значений, обозначенные тем же именем S и принимающие значения из той же таблицы S.

8.6.13. Определить имена поставщиков детали с номером P2

Примечание. Этот пример повторяет пример 8.6.10. Ниже приведено другое решение, позволяющее представить еще одно средство языка SQL.

```
SELECT DISTINCT
S.SNAME FROM   S
WHERE  EXISTS
      ( SELECT *
        FROM   SP
       WHERE   SP.S# = S.S#
        AND    SP.P# = P# ('P2') ) ;
```

Пояснение. Выражение EXISTS (SELECT ... FROM …) на языке SQL принимает значение TRUE тогда и только тогда, когда результат вычисления выражения SELECT ... FROM ... будет непустым. Другими словами, в языке SQL функция EXISTS соответствует квантору существования реляционного исчисления (в большей или меньшей степени; см. [19.6]).

Примечание. **В** спецификации SQL ссылки на подзапрос, аналогичные показанным в данном примере, называются ссылками на **коррелированный** подзапрос (correlated subquery), поскольку в данном подзапросе содержится ссылка на переменную области значений (а именно — на неявную переменную области значений S), которая определена во внешнем запросе. Еще одним примером коррелированного подзапроса может служить пример 8.6.8.

8.6.14. Определить имена поставщиков, которые не поставляют деталь с номером P2 (пример 8.3.7)

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  NOT EXISTS
         ( SELECT *
           FROM   SP
           WHERE  SP.S# = S.S# AND
           SP.P# = P# ('P2') ) ;
```

Этот же запрос можно также представить в альтернативной формулировке.

```
SELECT DISTINCT
S.SNAME FROM   S
WHERE  S.S# NOT IN
     ( SELECT SP.S#
       FROM   SP       WHERE  SP.P#
= P# ( ' P2 ' ) ) ;
```

8.6.15. Определить имена поставщиков, которые поставляют детали всех типов (см. пример 8.3.6)

```
SELECT DISTINCT
S.SNAME FROM   S
WHERE  NOT EXISTS    (
        SELECT * FROM   P
        WHERE  NOT EXISTS
                   ( SELECT *    FROM   SP
                  WHERE  SP.S# = S.S#    AND
                  SP.P# = P.P# ) ) ;
```

Язык SQL не включает какой-либо непосредственной поддержки квантора всеобщности FORALL; следовательно, запросы "с кванторами FORALL" обычно выражаются с помощью квантора существования и двойного отрицания, как и в этом примере.

Следует отметить, что выражения, подобные показанному выше, хотя на первый взгляд и кажутся довольно устрашающими, легко формируются пользователями, знакомыми с реляционным исчислением, как отмечается в [8.4]. В ином случае, если подобные примеры все еще кажутся слишком сложными, существует несколько "обходных" путей, позволяющих избежать использования отрицаемых кванторов. В данном случае, например, можно записать запрос следующим образом.

```
SELECT DISTINCT S.SNAME
FROM   S
WHERE  ( SELECT COUNT ( SP.P# )
         FROM   SP
         WHERE  SP.S# = S.S# )
     = ( SELECT COUNT ( P.P#
     )
         FROM   P ) ;
```

(Расшифровка: "Получить имена поставщиков, для которых количество поставляемых деталей равно количеству всех деталей".) Однако следует отметить, что в последней формулировке (в отличие от формулировки с выражением NOT EXISTS) используется тот факт, что номер каждой поставляемой детали является номером существующей детали. Другими словами, эти две формулировки эквивалентны, но вторая является правильной только благодаря поддержке некоторого *ограничения целостности* (подробности приводятся в следующей главе).

Примечание. В действительности в приведенном выше примере выполняется задача сравнения двух таблиц, поэтому данный запрос можно было бы представить следующим образом.

```
SELECT DISTINCT S.SNAME /* Предостережение! Недопустимый
запрос! */
FROM   S
WHERE  { SELECT SP.P#
         FROM   SP
         WHERE  SP.S# = S.S#
     ) = ( SELECT P.P#
         FROM   P ) ;
```

Но в языке SQL непосредственно не поддерживается операция сравнения таблиц, поэтому приходится прибегать к уловке, используя сравнение *кардинальностей* таблиц вместо сравнения таблиц (опираясь на практический опыт, который свидетельствует о том, что если кардинальности таблиц равны, то и таблицы одинаковы, по крайней мере, в обсуждаемом случае). Дополнительный материал по данной теме приведен в упр. 8.11.

8.6.16. Определить номера деталей, которые либо весят более 16 фунтов, либо поставляются поставщиком с номером S2, либо соответствуют и тому, и другому условию (см. пример 8.3.9)

```
SELECT P.P#
FROM  P
WHERE P.WEIGHT > WEIGHT ( 16.0 )

UNION
```

```
SELECT SP.P#
FROM   SP
WHERE  SP.S# = S# ('S2') ;
```

Лишние повторяющиеся строки всегда исключаются из результата выполнения неуточненных операторов **UNION, INTERSECT** и **EXCEPT** (в языке SQL оператор **EXCEPT** служит аналогом операции **MINUS** реляционной алгебры). Однако язык SQL также поддерживает уточненные варианты этих операторов **(UNION ALL, INTERSECT ALL** и **EXCEPT ALL),** при которых повторяющиеся строки (если они есть) сохраняются. Примеры с этими вариантами операторов умышленно не показаны.

8.6.17. Определить номер детали и вес в граммах для каждой детали с весом > 10 000 г (см. пример 8.5.1)

```
SELECT P.P#, P.WEIGHT * 454 AS GMWT
FROM   P
WHERE  P.WEIGHT * 454 > WEIGHT ( 10000.0 ) ;
```

Теперь необходимо вспомнить определение конструкция WITH, которая была впервые представлена в главе 5 и использовалась при описании реляционной алгебры[9] в главе 7. Неформально выражаясь, конструкция WITH предназначена для присваивания имен выражениям. В языке SQL также имеется конструкция WITH, но ее применение ограничивается только выражениями с таблицами. В данном примере с помощью этой конструкции можно избежать необходимости дважды записывать выражение P.WEIGHT * 454, как показано ниже.

```
WITH T1 AS ( SELECT P.P#, P.WEIGHT * 454 AS GMWT
             FROM   P )
     SELECT T1.P#, T1.GMWT FROM   T1
     WHERE  T1.GMWT > WEIGHT ( 10000.0
     ) ;
```

Кстати, следует отметить, что записи в конструкции WITH (которые в предыдущей главе именовались как <*name intro*> — определение имени) в языке SQL принимают вид<name> AS (<exp>), а в языке Tutorial D имеют форму <*exp*> AS <*name*>. Необходимо также учитывать, что конструкция WITH приобретает важное значение, когда требуется сформулировать на языке SQL аналог алгебраического оператора TCLOSE. Дополнительные сведения здесь не представлены, но соответствующий пример можно найти в ответе на упр. 4.6, который представлен в приложении Д.

На этом список примеров применения языка SQL для выборки данных завершается. Хотя этот список был достаточно большим, о многих возможностях языка SQL здесь даже не упоминалось. Язык SQL в действительности является чрезвычайно избыточным [4.19] в том смысле, что почти всегда существует множество способов представления одного и того же запроса, и нам не хватит места, чтобы описать все возможные формулировки и все возможные опции даже для сравнительно небольшого числа примеров, которые рассматривались в этой главе. (Дополнительные сведения на эту тему приведены в приложении Б.)

---

[9] Безусловно, эта конструкция может также использоваться в реляционном исчислении.

8.7. ИСЧИСЛЕНИЕ ДОМЕНОВ

Как было указано в разделе 8.1, исчисление доменов отличается от исчисления кортежей тем, что в нем переменные области значений определены на *доменах* (типах), а не на отношениях. С точки зрения синтаксиса наиболее очевидное различие между исчислением доменов и исчислением кортежей состоит в том, что первое поддерживает дополнительную форму параметра *<bool exp>,* который мы будем называть **условием принадлежности** (membership condition). В общем виде условие принадлежности можно записать следующим образом.

R { *<pair commalist>* }

Здесь R — имя переменной отношения, а каждый параметр *<pair>* имеет вид A x, где A — имя атрибута переменной отношения R, а x — имя переменной области значений или вызов селектора (чаще всего литерал). В целом, это условие принимает значение TRUE тогда и только тогда, когда в текущем значении переменной отношения R существует такой кортеж, что для каждого заданного выражения *<pair>* A x сравнение A = x имеет значение TRUE для данного кортежа. Например, рассмотрим результат вычисления следующего выражения.

```
SP { S# S#('S1'), P# P#('P1') }
```

Это выражение является условием принадлежности, которое принимает значение TRUE тогда и только тогда, когда обнаруживается, что в настоящее время существует кортеж с данными о поставке, в котором значение s# равно si, а значение P# равно P1. Аналогичным образом, условие принадлежности

```
SP { S# SX, P# PX }
```

принимает значение TRUE тогда и только тогда, когда обнаруживается, что в настоящее время существует кортеж с данными о поставке, в котором значение атрибута S# равно текущему значению переменной области значений SX (каким бы оно ни было), а значение атрибута p# равно текущему значению переменной домена PX (опять же, каким бы оно ни было).

До конца данного раздела подразумевается, что существуют переменные области значений, показанные в табл. 8.1.

Домен	Переменная области значений
S#	SX, SY, ...
P#	PX, PY, ...
NAME	NAMEX, NAMEY, ...
COLOR	COLORX, COLORY, ...
WEIGHT	WEIGHTX, WEIGHTY, ...
QTY	QTYX, QTYY, ...
CHAR	CITYX, CITYY, ...
INTEGER	STATUSX, STATUSY, ...

Таблица 8.1. Переменные области значений, применяемые в примерах данного раздела

Ниже приведено несколько примеров выражений исчисления доменов.

```
SX

SX WHERE S { S# SX }

SX WHERE S { S# SX, CITY 'London' }

{ SX, CITYX } WHERE S { S# SX, CITY CITYX }
             AND   SP { S# SX, P# P#('P2') }

{ SX, PX } WHERE S { S# SX, CITY
          CITYX } AND P { P# PX,
          CITY CITYY } AND   CITYX
          ≠ CITYY
```

Неформально первое выражение обозначает множество всех номеров поставщиков, второе — множество всех номеров поставщиков в переменной отношения S, третье — подмножество номеров поставщиков из Лондона. Следующее выражение — это представленный в терминах исчисления доменов запрос "Определить номера поставщиков и названия городов, в которых находятся поставщики детали с номером P2" (вспомните, что для формулировки этого запроса в терминах исчисления кортежей требовался квантор существования). И последнее выражение — это представленный в терминах исчисления доменов запрос "Найти все такие пары номеров поставщиков и номеров деталей, что поставщики находятся в том же городе, где хранится деталь".

Ниже приведено несколько примеров из числа рассмотренных в разделе 8.3, но на этот раз выраженных в терминах исчисления доменов (часть из них несколько изменена).

8.7.1. Определить номера поставщиков из Парижа со статусом больше 20 (упрощенная версия примера 8.3.1)

```
SX WHERE EXISTS STATUSX
       { STATUSX > 20 AND
         S { S# SX, STATUS STATUSX, CITY 'Paris' } )
```

Этот первый пример выглядит не так изящно, как его аналог, выраженный в терминах исчисления кортежей (особого внимания заслуживает то, что в нем все еще требуется явно использовать кванторы). С другой стороны, существуют ситуации, когда верно обратное утверждение, что видно из более сложных примеров, приведенных ниже.

8.7.2. Найти все такие пары номеров поставщиков, в которых два поставщика находятся в одном городе (см. пример 8.3.2)

```
{ SX AS SA, SY AS SB } WHERE EXISTS CITYZ
                          ( S { S# SX, CITY CITYZ }
                          AND S { S# SY, CITY CITYZ
                          } AND  SX < SY )
```

8.7.3. Определить имена поставщиков по крайней мере одной детали красного цвета (см. пример 8.3.4)

```
NAMEX WHERE EXISTS SX EXISTS PX
          ( S { S# SX, SNAME
         NAMEX }    AND SP { S#
         SX, P# PX }
            AND P { P# PX, COLOR COLOR('Red') } 1
```

8.7.4. Определить имена поставщиков, которые поставляют хотя бы один тип деталей, поставляемых поставщиком с номером S2 (см. пример 8.3.5)

```
NAMEX WHERE EXISTS     SX EXISTS PX
          ( S { S#SX, SNAME NAMEX }
            AND SP      { S# SX, P# PX }
           AND SP{ S# S#('S2'), P# PX } )
```

8.7.5. Определить имена поставщиков, которые поставляют детали всех типов (см. пример 8.3.6)

```
NAMEX WHERE EXISTS SX ( S { S# SX, SNAME
           NAMEX } AND FORALL PX ( IF P { P#
           PX }
                        THEN SP { S# SX, P# PX
                        } END IF )
```

8.7.6. Определить имена поставщиков, которые не поставляют деталь с номером P2 (см. пример 8.3.7)

```
NAMEX WHERE EXISTS SX ( S { S# SX, SNAME
           NAMEX } AND NOT SP { S# SX, P#
           P#('P2') } )
```

8.7.7. Определить номера поставщиков, которые поставляют, по меньшей мере, детали всех типов, поставляемых поставщиком с номером S2 (см. пример 8.3.8)

```
SX WHERE FORALL PX ( IF SP { S# S#('S2'), P#
                    PX } THEN SP { S# SX, P#
                    PX } END IF )
```

8.7.8. Получить номера деталей, которые либо весят более 16 фунтов, либо поставляются поставщиком с номером S2, либо соответствуют и тому, и другому условию (см. пример 8.3.9)

```
PX WHERE EXISTS WEIGHTX
              ( P { P# PX, WEIGHT WEIGHTX }
                AND WEIGHTX > WEIGHT ( 16.0 )
              ) OR SP { S# S#('S2'), P# PX }
```

Исчисление доменов, как и исчисление кортежей, формально эквивалентно реляционной алгебре (т.е. оно является реляционно полным). Доказательство этого утверждения можно найти, например, в статье Ульмана (Ullman) [8.13].

8.8. ЯЗЫК ЗАПРОСОВ ПО ОБРАЗЦУ

Одним из наиболее широко известных примеров языка, основанного на исчислении доменов, является язык запросов по образцу (Query-By-Example — QBE) [8.14]. (Фактически QBE одновременно воплощает в себе средства исчисления доменов и исчисления кортежей, но первое в нем доминирует.) Его синтаксис является очень привлекательным и простым для интуитивного восприятия; он основан на идее внесения записей в пустые таблицы. Например, формулировка на языке QBE запроса: "Определить имена поставщиков,

которые поставляют по меньшей мере одну деталь, поставляемую поставщиком S2" может выглядеть примерно следующим образом.

S	S#	SNAME
	_SX	P._NX

SP	S#	P#
	_SX	_PX

SP	S#	P#
	S2	_PX

Пояснение. Пользователь запрашивает систему вывести на экран три пустые таблицы (одну для поставщиков и две для поставок), затем внести в них показанные выше записи. Записи, начинающиеся с символа подчеркивания, представляют собой примеры элементов (т.е. переменные области определения в исчислении доменов); другие записи представляют собой литеральные значения. Пользователь запрашивает систему предоставить ему ("P." — сокращение от present) имена поставщиков (_NX), такие, что если поставщик имеет номер _sx, то поставщик _sx поставляет некоторую деталь _PX, а эта деталь _PX, в свою очередь, поставляется поставщиком S2. Сравнение этой формулировки QBE с эквивалентной ей формулировкой в исчислении кортежей или доменов (см. примеры 8.3.5 и 8.7.4) показывает, что ее отличие от таких формулировок состоит в отсутствии явно заданных кванторов[10]; это служит еще одной причиной того, что язык QBE является простым для интуитивного восприятия. Имеет также смысл сравнить эту версию запроса QBE с формулировкой на языке SQL (оставляем это в качестве упражнения для читателя).

Ниже приведен ряд примеров, позволяющих проиллюстрировать некоторые из основных особенностей языка QBE. В качестве упражнения рекомендуем читателю сравнить и сопоставить эти примеры на языке QBE с их аналогами, основанными на использовании чистого исчисления доменов.

8.8.1. Определить номера поставщиков, находящихся в Париже, которые имеют статус > 20 (пример 8.7.1)

S	S#	SNAME	STATUS	CITY
	P.		> 20	Paris

Обратите внимание, насколько просто можно представить операции сравнения ">" и "=". Следует также отметить, что нет необходимости явно задавать элемент примера, если он больше нигде не упоминается (но и не будет ошибкой явное задание элемента примера, такого как P._sx). Кроме того, заслуживает внимания то, что символьные строковые значения, такие как Paris, можно задавать, не заключая их в кавычки (но не будет также ошибкой применение таких кавычек, а иногда они даже требуются, например, если строка включает пробелы).

Возможно также ввести запись "P." применительно ко всей строке, например, как показано ниже.

S	S#	SNAME	STATUS	CITY
P.			> 20	Paris

[10] Кстати, аналогичное замечание относится и к языку QUEL (см., например, [8.5]).

Этот пример эквивалентен следующему, в котором запись "P." присутствует в каждой позиции столбца в строке.

S	S#	SNAME	STATUS	CITY
	P.	P.	P. > 20	P.Paris

Из данного примера следует еще один вывод: в системе должны быть предусмотрены средства, позволяющие редактировать на экране пустые таблицы, добавляя и удаляя столбцы и строки, а также расширяя и сужая столбцы. Это позволило бы корректировать структуру таблиц таким образом, чтобы они соответствовали требованиям любых операций, который должны быть сформулированы пользователем; в частности, это дало бы возможность удалять столбцы, которые не требуются в рассматриваемой операции. Например, в первой формулировке на языке QBE обсуждаемого примера можно было бы удалить столбец SNAME И получить следующую таблицу.

S	S#	STATUS	CITY
	P.	> 20	Paris

Поэтому в приведенных ниже примерах мы будем часто удалять столбцы, которые не требуются в формируемом запросе.

8.8.2. Определить номера всех поставляемых деталей, удалив ненужные дубликаты

SP	S#	P#	QTY
UNQ.		P.	

В данной таблице UNQ. является сокращением от unique — уникальный (эта запись соответствует ключевому слову DISTINCT в языке SQL).

8.8.3. Получить номера и данные о статусе поставщиков, находящихся в Париже, вначале выполнив сортировку в порядке убывания статуса, а затем — в порядке возрастания номеров

S	S#	STATUS	CITY
	P.AO(2).	P.DO(1).	Paris

Здесь запись "AO." обозначает сортировку в порядке возрастания, а "DO." — в порядке убывания. Числа в круглых скобках указывают последовательность сортировки столбцов от старшего к младшему; в данном примере STATUS является старшим столбцом, а s# — младшим.

8.8.4. Получить номера и данные о статусе поставщиков, которые либо находятся в Париже, либо имеют статус > 20, либо соответствуют обоим условиям (модифицированная версия примера 8.8.1)

Условия, заданные в одной строке, рассматриваются как соединенные друг с другом логическим оператором "И" (см., допустим, пример 8.8.1). Для того чтобы соединить два условия логическим оператором "ИЛИ", их необходимо задать в разных строках, как показано ниже.

S	S#	STATUS	CITY
	P.		Paris
	P.	>	20

Еще один подход к формированию этого запроса состоит в использовании так называемого *поля условия* (condition box), как показано ниже.

S	S#	STATUS	CITY	CONDITIONS
	P.	_ST	Paris	_SC = Paris OR _ST > 20

В общем поле условия позволяет задавать такие условия, которые слишком сложно выразить в одном столбце пустой таблицы (например, операции сравнения, в которых используются два разных столбца, или операции сравнения с применением агрегирующей функции).

8.8.5. Определить детали, вес которых находится в пределах от 16 до 19 включительно

P	P#	WEIGHT	WEIGHT
	P.	>= 16.0	<= 19.0

8.8.6. Для всех деталей определить номер детали и вес детали в граммах (пример 8.6.2)

P	P#	WEIGHT	GMWT
	P.	_PW	P. _PW * 454

8.8.7. Определить номера поставщиков, которые поставляют деталь P2 (пример 7.5.1)

S	S#	SNAME
	_SX	P.

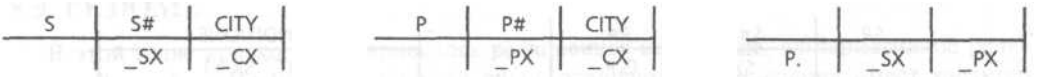
SP	S#	P#
	_SX	P2

В данном случае к строке таблицы SP неявно применен квантор существования. Этот запрос можно выразить иначе, как показано ниже.

Определить имена поставщиков SX, таких что существует поставка, характеризующаяся тем, что поставщик SX поставляет деталь P2.

Таким образом, язык QBE неявно поддерживает квантор существования EXISTS (следует также отметить, что эта неявная переменная области значений определена на отношении, а не на домене, и именно поэтому выше было указано, что язык QBE включает некоторые средства исчисления кортежей). Но данный язык не поддерживает[11] определение NOT EXISTS. Вследствие этого некоторые запросы (например, "Определить имена поставщиков, которые поставляют детали всех типов" — см. пример 8.7.5) нельзя представить на языке QBE, и этот язык не является реляционно полным.

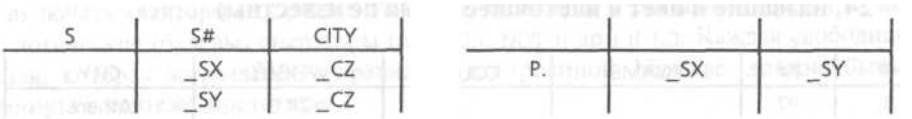
8.8.8. Определить все пары номеров поставщиков и номеров деталей, такие что поставщик находится в том же городе, где хранится рассматриваемая деталь (модифицированная версия примера 8.6.3)



Для этого запроса требуются три пустые таблицы: по одной для отношений S и P (показаны только необходимые столбцы) и одна для результата. Обратите внимание на то, как заданы элементы примера, позволяющие связать между собой эти три таблицы. В целом данный запрос можно выразить иначе, как показано ниже.

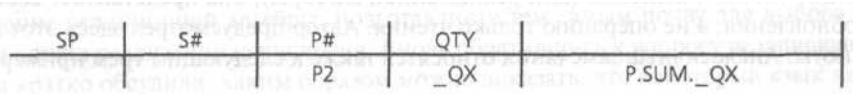
Определить такие пары номеров поставщиков и номеров деталей, скажем, SX и PX, что и SX, и PX находятся в одном и том же городе cx.

8.8.9. Определить все пары номеров поставщиков, таких что оба поставщика в каждой паре находятся в одном городе (пример 8.6.5)



В случае необходимости для определения дополнительного условия _SX < _SY может использоваться поле условия.

8.8.10. Определить общее количество поставляемых деталей P2



В языке QBE поддерживаются все обычные операции агрегирования.

---

[11] По меньшей мере, не поддерживает должным образом. Данное определение поддерживается в нем лишь частично. На первых порах фактически в языке QBE была предусмотрена "полная" поддержка определения NOT EXISTS, но эта поддержка всегда была под сомнением. Основная проблема состояла в том, что не было способа указать порядок, в котором должны были применяться различные неявные кванторы, но, к сожалению, этот порядок имеет значение, если присутствуют какие-либо операторы отрицания NOT. В результате некоторые выражения QBE были двусмысленными. Подробное обсуждение данного вопроса можно найти в [8.3]. См. также упр. 8.2.

8.8.11. Для каждой поставляемой детали определить номер детали и общий объем поставки (пример 8.6.8)

SP	S#	P#	QTY	
		G. P.	_QY	P.SUM._QY

Запись "G. " обеспечивает группирование (она соответствует конструкции GROUP BY в языке SQL).

8.8.12. Определить номера всех деталей, поставляемых больше чем одним поставщиком

SP	S#	P#
	_SX	G. P.

CONDITIONS
CNT._SX >1

8.8.13. Определить номера деталей, которые либо весят больше 16 фунтов, либо поставляются поставщиком S2, либо соответствуют и тому, и другому условию (пример 8.7.8).

P	P#	WEIGHT
	_PX	> 16.0

SP	S#	P#
	S2	_PY

P.	_PX
P.	_PY

8.8.14. Вставить в таблицу P данные о детали с номером P7 (город Афины, вес 24, название и цвет в настоящее время не известны)

P	P#	PNAME	COLOR	WEIGHT	CITY
I.	P7			24.0	Athens

Обратите внимание на то, что запись "I. " применяется ко всей строке, поэтому находится под именем таблицы.

Примечание. Безусловно, операция вставки новых кортежей — это вообще не операция реляционного исчисления (или реляционной алгебры); она представляет собой операцию обновления, а не операцию только чтения. Автор предусмотрел здесь этот пример для полноты. Аналогичные замечания относятся также к следующим трем примерам.

8.8.15. Удалить данные обо всех поставках, в которых количество поставляемых деталей было больше 300

SP	S#	P#	QTY
D.			> 300

Запись "D. " находится под именем таблицы.

8.8.16. Изменить цвет детали P2 на желтый, увеличить ее вес на пять и указать в соответствующей колонке город Осло

P	P#	PNAME	COLOR	WEIGHT	WEIGHT	CITY
	P2		U.Yellow	_WT	U._WT + 5	U.Oslo

8.8.17. Для всех поставщиков, находящихся в Лондоне, изменить объем поставки на пять

SP	S#	QTY
	_SX	U.5

S	S#	CITY
	_SX	London

8.9. РЕЗЮМЕ

В этой главе кратко рассматривалось **реляционное исчисление,** альтернативное реляционной алгебре. Внешне два подхода очень отличаются: исчисление имеет характер **описания,** а алгебра — характер **предписания,** но на более низком уровне они представляют собой одно и то же, поскольку любые выражения исчисления могут быть преобразованы в семантически эквивалентные выражения алгебры и наоборот.

Реляционное исчисление существует в двух версиях: исчисление **кортежей** и исчисление доменов. Основное различие между ними состоит в том, что переменные исчисления кортежей определяются на отношениях, а переменные исчисления доменов определяются на доменах.

Выражение исчисления кортежей состоит из **кортежа-прототипа** и необязательной конструкции WHERE, содержащей логическое выражение или **правильно построенную формулу** (Well-Formed Formula — WFF). Подобная правильно построенная формула может включать **кванторы** (EXISTS и FORALL), **свободные** и **связанные ссылки на переменные,** логические (булевы) операторы (AND, OR, NOT и др.) и т.д. Каждая свободная переменная, которая встречается в правильно построенной формуле, должна быть также упомянута в кортеже-прототипе.

Примечание. В настоящей главе явно этот вопрос не затрагивался, но выражения реляционного исчисления предназначены по существу для тех же целей, что и выражения реляционной алгебры (см. раздел 7.6 главы 7).

На примере было показано, как можно использовать **алгоритм редукции** Кодда для преобразования произвольного выражения реляционного исчисления в эквивалентное выражение реляционной алгебры, подготавливая тем самым почву для выбора возможной стратегии реализации исчисления. Вновь обратившись к вопросу **реляционной полноты,** мы кратко обсудили, каким образом можно доказать, что некоторый язык является в этом смысле полным.

Кроме того, здесь обсуждалось, как можно расширить исчисление кортежей в целях поддержки определенных **вычислительных возможностей** (аналогичные возможности в реляционной алгебре обеспечиваются операциями EXTEND и SUMMARIZE). Затем читателям был представлен обзор соответствующих средств языка SQL. Язык SQL является своеобразной смесью реляционной алгебры и исчисления (кортежей). Например, в нем есть прямая поддержка таких операций реляционной алгебры, как соединение и объединение, но одновременно с этим используются переменные области значений и квантор существования из реляционного исчисления.

Запрос SQL представляет собой **табличное выражение.** Обычно такая конструкция содержит единственное **выражение выборки,** однако поддерживаются и различные типы явных выражений операций **соединения,** причем выражения соединения и выборки могут комбинироваться произвольным образом с помощью операторов **UNION, INTERSECT** и **EXCEPT.** Также упоминалось о возможности использования конструкция **ORDER BY** для определения упорядоченности строк в таблице, являющейся результатом вычисления данного табличного выражения (любого вида).

В частности, были описаны следующие компоненты **выражений выборки.**

- Базовая **конструкция SELECT,** в том числе использование ключевого слова **DISTINCT,** скалярных выражений, введение имен результирующих столбцов и использование сокращения SELECT *.
- **Конструкция FROM,** включая использование **переменных области значений.**
- **Конструкция WHERE,** включая использование оператора **EXISTS.**
- **Конструкция GROUP BY** и **HAVING,** включая использование **агрегирующих функций** COUNT, SUM, AVG И Т.Д.
- Использование **подзапросов** (например) в конструкциях[12] SELECT, FROM И WHERE.

Кроме того, здесь был описан **концептуальный алгоритм вычисления** выражений выборки языка SQL (основа для формального определения этих выражений). Кратко можно отметить, что этот алгоритм предусматривает формирование декартова произведения таблиц, указанных в конструкции FROM, применение операции сокращения к этому произведению в соответствии с логическим выражением, указанным в конструкции WHERE, и наконец, применение операции проекции к результатам операции сокращения по столбцам, указанным в конструкции SELECT. Но следует сразу же сделать оговорку, что это краткое описание является далеко не полным; более подробные сведения приведены в [4.20].

Затем было представлено краткое введение в исчисление **доменов** и указано, без попытки доказать это утверждение, что это исчисление также является реляционно полным. Таким образом, исчисление кортежей, исчисление доменов и реляционная алгебра эквивалентны друг другу. Наконец, кратко описаны средства языка запросов по образцу, который является, по-видимому, наиболее широко известной коммерческой реализацией идеи исчисления доменов.

УПРАЖНЕНИЯ

8.1. Пусть p (x) и q — произвольные правильно построенные формулы, в которых переменная x, соответственно, используется и не используется в качестве свободной переменной. Какие из следующих формулировок верны? (Здесь символ "=>" означает "следует", а символ "≡" означает "эквивалентно". Обратите внимание, что если A => B и B => A, то A ≡ в.)

---

[12] Но следует отметить, что довольно неформально подзапросы в конструкции FROM часто рассматриваются как табличные выражения, подзапросы в конструкции SELECT — как скалярные выражения, а подзапросы в конструкции WHERE — как табличные выражения или скалярные выражения, в зависимости от контекста (!).

а) `EXISTS x ( q ) ≡ q`

б) `FORALL x ( q ) ≡ q`

в) `EXISTS x ( p (x) AND q ) ≡ EXISTS x ( p (x) ) AND q`

г) `FORALL x ( p (x) AND q ) ≡ FORALL x ( p (x) ) AND q`

д) `FORALL x ( p (x) ) => EXISTS x ( p (x) )`

е) `EXISTS x ( TRUE ) s TRUE`

ж) `FORALL x ( FALSE ) s FALSE`

8.2. Пусть p (x, y) — это произвольная правильно построенная формула со свободны ми переменными x и y. Какие из следующих формулировок верны?

а) `EXISTS x EXISTS y ( p(x,y) ) ≡ EXISTS y EXISTS x ( p(x,y) )`

б) `FORALL x FORALL y ( p(x,y) ) ≡ FORALL y FORALL x ( p(x,y) )`

в) `FORALL x ( p(x,y) ) ≡ NOT EXISTS x ( NOT p(x,y) )`

г) `EXISTS x ( p(x,y) ) ≡ NOT FORALL x ( NOT p(x,y) )`

д) `EXISTS x FORALL y ( p(x,y) ) ≡ FORALL y EXISTS x ( p(x,y) )`

е) `EXISTS y FORALL x ( p(x,y) ) => FORALL x EXISTS y ( p(x,y) )`

8.3. Пусть p(x) и q(y) — произвольные правильно построенные формулы, соответст венно, со свободными переменными x и y. Какие из следующих формулировок верны?

а) `EXISTS x ( p(x) ) AND EXISTS y ( q(y) ) ≡`
`EXISTS x EXISTS y ( p(x) AND q(y) )`

б) `EXISTS x ( IF p(x) THEN q(x) END IF ) e`
`IF FORALL x ( p(x) ) THEN EXISTS x ( q(x) ) END IF`

8.4. Еще раз обратимся к запросу "Определить номера поставщиков, по крайней мере, всех типов деталей, поставляемых поставщиком с номером S2". Для этого запроса возможна следующая формулировка в терминах исчисления кортежей.

```
SX.S# WHERE FORALL SPY ( IF SPY.S# = S# ( ' S2') THEN
                           EXISTS SPZ ( SPZ.S# = SX.S# AND
                                        SPZ.P# = SPY.P# )
                           END IF )
```

(Здесь SPZ — это еще одна переменная области значений, которая определена на отношении поставок.) Что будет возвращено при выполнении этого запроса, если поставщик с номером S2 в данный момент не поставляет никаких деталей? Что будет, если в приведенном выражении переменную SX всюду заменить перемен ной SPX?

8.5. Ниже приведен пример запроса к базе данных поставщиков, деталей и проектов (используются обычные соглашения по именованию переменных области опреде ления).

```
{ PX.PNAME, PX.CITY } WHERE FORALL SX FORALL JX EXISTS SPJX
                                ( SX.CITY = 'London' AND
                                  JX.CITY = 'Paris' AND
                                  SPJX.S# = SX.S# AND
                                  SPJX.P# = PX.P# AND
                                  SPJX.J# = JX.J# AND
                                  SPJX.QTY < QTY { 500 )
                                  )
```

а) Сформулируйте этот запрос в словесной форме.

б) Возьмите на себя роль СУБД и выполните этот запрос по предложенному Коддом алгоритму редукции. Можете ли вы указать какие-либо улучшения, которые целесообразно внести в данный алгоритм?

8.6. Выразите в терминах исчисления кортежей запрос "Определить три самые тяжелые детали".

8.7. Рассмотрим отношение спецификации материалов переменной отношения PART_STRUCTURE, представленной в упр. 4.6 главы 4. Обратимся к широко известному запросу разузлования деталей "Получить номера деталей, которые на любых уровнях вхождения являются компонентами некоторой заданной детали (скажем, детали с номером P1)". Результат этого запроса, например отношение PART_BILL (которое, безусловно, является отношением, производным от исходного отношения PART_STRUCTURE), нельзя сформулировать в виде единственного выражения начального реляционного исчисления (или реляционной алгебры). Иначе говоря, производное отношение PART_BILL не может быть получено с помощью единственного выражения начального реляционного исчисления (или реляционной алгебры). Объясните, почему.

8.8. Предположим, что переменную отношения поставщиков S заменили набором переменных отношения LS, PS, AS и т.д., по одной переменной отношения для каждого города (например, переменная отношения LS будет содержать кортежи только для поставщиков из Лондона). Предположим также, что не известно, какие именно существуют города, в которых находятся поставщики, и поэтому не известно, сколько имеется таких переменных отношения. Рассмотрим запрос "Существует ли в базе данных поставщик с номером S1?". Можно ли такой запрос выразить в терминах исчисления (или алгебры)? Обоснуйте свой ответ.

8.9. Покажите, что язык SQL является реляционно полным.

8.10. Существуют ли в языке SQL эквиваленты реляционных операторов EXTEND и SUMMARIZE?

8.11. Существуют ли в языке SQL эквиваленты операторов реляционных сравнений?

8.12. Приведите как можно больше различных формулировок на языке SQL для запроса "Выбрать имена поставщиков детали с номером P2".

Упражнения по запросам

В основу всех остальных упражнений была положена база данных поставщиков, деталей и проектов. В каждом случае вам будет предложено записать выражение, позволяющее выполнить указанный запрос. (В качестве интересного варианта выполнения упражнения попытайтесь вначале посмотреть ответ, приведенный в приложении Д, и определить, что данное выражение означает на естественном языке.)

8.13. Дайте ответы к упр. 7.13—7.50 в терминах исчисления кортежей.

8.14. Дайте ответы к упр. 7.13-7.50, используя средства языка SQL.

8.15. Дайте ответы к упр. 7.13—7.50 в терминах исчисления доменов.

8.16. Дайте ответы к упр. 7.13-7.50, используя средства языка QBE.

СПИСОК ЛИТЕРАТУРЫ

8.1. Codd E.F. A Data Base Sublanguage Founded on the Relational Calculus // Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control. — San Diego, Calif. —November 1971.

8.2. Date C.J. A Note on the Relation Calculus // ACM SIGMOD Record 18. — 1989. — № 4. Переиздано: An Anomaly in Codd's Reduction Algorithm // C.J. Date and Hugh Darwen. Relational Database Writings 1989—1991. Reading, Mass.: Addison-Wesley, 1992.

8.3. Date C.J. Why Quantifier Order Is Important // C.J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

8.4. Date C.J. Relational Calculus as an Aid to Effective Query Formulation // C.J. Date and Hugh Darwen. Relational Database Writings 1989—1991.— Reading, Mass.: Addison-Wesley, 1992.

Почти все представленные на рынке современные реляционные продукты поддерживают язык SQL, а не реляционное исчисление (или реляционную алгебру). В этой статье, тем не менее, обосновывается (и иллюстрируется) применение реляционного исчисления как промежуточного шага в построении запросов SQL.

8.5. Held G.D., Stonebraker M.R., and Wong E. INGRES — A Relational Data Base System // Proc. NCC 44. — Anaheim, Calif. Montvale, N.J.: AFIPS Press, May 1975.

С середины и до конца 1970-х годов разрабатывались два главных реляционных прототипа: система System R в фирме IBM и система Ingres (первоначально ее название записывалось прописными буквами, INGRES) в Калифорнийском университете (город Беркли). Оба этих проекта оказали очень большое влияние на дальнейшие исследования и впоследствии привели к созданию таких коммерческих систем, как СУБД DB2 (в случае системы System R) и коммерческий продукт Ingres (в случае системы Ingres).

Примечание. Прототип Ingres иногда называют "University Ingres" [8.11], чтобы отличать его от коммерческой версии этой системы. Учебный обзор коммерческой версии можно найти в [1.5].

Система Ingres вначале не являлась системой SQL, поскольку исходно поддерживала язык QUEL (Query Language), который во многих отношениях технически превосходил SQL. В самом деле, язык QUEL по-прежнему составляет основу достаточного числа современных исследований баз данных, а примеры на языке QUEL до сих пор появляются в исследовательской литературе. Данная статья, в которой впервые был описан прототип Ingres, включает предварительное определение языка QUEL. См. также [8.10]—[8.12].

8.6. Kuhns J.L. Answering Questions by Computer: A Logical Study // Report RM-5428-PR, Rand Corp. — Santa Monica, Calif., 1967.

8.7. Lacrorix M., Pirotte A. Domain-Oriented Relational Languages // Proc. 3rd Int. Conf. on Very Large Data Bases. — Tokyo, Japan., October 1977.

8.8. Merrett T.H. The Extended Relational Algebra, A Basis for Query Languages // Shneiderman B. (ed.). Databases: Improving Usability and Responsiveness. New York, N.Y.: Academic Press., 1978.

Расширенная реляционная алгебра, указанная в заголовке этой статьи, включает некоторые кванторы — не просто кванторы EXISTS и FORALL, которые описаны в данной главе, но и более общие кванторы "количество" и "соотношение". С их помощью можно выразить такие условия, как "по крайней мере три", "не больше половины", "нечетное число" и т.д.

8.9. Negri M, Pelagatti G., Sbattella L. Formal Semantics of SQL Queries // ACM TODS. - September 1991. - 16, № 3.

Цитата из резюме: "Семантика запросов SQL формально определена с помощью множества правил, определяющих преобразование на основе синтаксиса запроса SQL в формальную модель, называемую *расширенным исчислением трехзначных предикатов* (Extended Three Valued Predicate Calculus — E3VPC), которая главным образом основана на хорошо известных математических понятиях. Приведены также правила преобразования общего выражения E3VPC в каноническую форму; [дополнительно] полностью решены проблемы, подобные анализу эквивалентности запросов SQL". Однако отметим, что в статье рассматривается только тот диалект языка SQL, который определен в самой первой версии стандарта (в 1986 году).

Примечание. Определения терминов *трехзначный предикат* **и** *каноническая форма,* соответственно, приведены в главах 19 и 18.

8.10. Stonebraker M. (ed.) The INGRES Papers: The Anatomy of a Relational Database Management System // Reading, Mass.: Addison-Wesley, 1986.

Сборник статей по проекту "University Ingres", отредактированный одним из первых создателей этого проекта, который написал также аннотацию к этому сборнику.

8.11. Stonebraker M., Wong E., Kreps P., Held G. The Design Implementation of INGRES // ACM TODS. — September, 1976. — 1, № 3. (Переиздано в [8.10].)

Подробное описание прототипа "University Ingres".

8.12. Stonebraker M. Retrospection on a Data Base System // ACM TODS. — 1980. — 5, № 2. (Переиздано в [8.10].)

Доклад по истории разработки проекта прототипа Ingres (до января 1979 года). Акцент сделан скорее на ошибках и полученных уроках, чем на достигнутых успехах.

8.13. Ullman J. D. Principles of Database and Knowledge-Base Systems: Volume I. Rockville, Md.: Computer Science Press, 1988.

Книга Ульмана, в отличие от настоящей книги, содержит более формальное изложение реляционного исчисления и связанных с ним вопросов. В частности, в ней обсуждается понятие **безопасности** выражений исчисления. Эта тема приобретает большое значение, если принята немного измененная версия исчисления, в которой переменные области значений не определяются отдельными операторами, а привязываются к области значений с помощью явных условий в конструкции WHERE. В такой версии исчисления запрос "Получить имена всех поставщиков из Лондона", например, может выглядеть следующим образом.

```
X WHERE X € S AND X.CITY = 'London'
```

Одна из (многих) проблем, возникающих при использовании этой версии исчисления, заключается в том, что в ней на первый взгляд кажутся вполне допустимыми запросы, подобные следующему.

```
X WHERE NOT ( X € S )
```

Такие выражения называются *небезопасными,* поскольку они не возвращают конечный результат (множество всего, что не входит в отношение S, бесконечно). Поэтому необходимо ввести некоторые правила, позволяющие гарантировать, что все допустимые выражения окажутся безопасными. Такие правила (для исчисления как кортежей, так и доменов) описаны в книге Ульмана. Следует отметить, что оригинальное исчисление Кодда неявно включало такие правила.

8.14. Zloof M. M. Query By Example // Proc. NCC 44. — Anaheim, Calif., 1975. — Montvale, N.J.: AFIPS Press, 1977.

Злуф (автор этой работы) первым придумал и разработал язык QBE. Данная статья была первой из многих статей, написанных Злуфом по этой теме.

ГЛАВА 9

Целостность данных

9.1. ВВЕДЕНИЕ

С годами та часть реляционной модели, которая касается целостности данных, подвергается наиболее существенным изменениям (возможно, здесь следовало сказать не "подвергается изменениям", а развивается). На первых порах основное внимание было обращено на исследование именно первичных и внешних ключей (сокращенно будем называть их просто *ключами)*. Но со временем специалисты по базам данных стали понимать важность (а фактически чрезвычайную важность) ограничений целостности как таковых, и поэтому количество работ в этой области значительно возросло; наряду с этим многие исследователи обнаружили определенные сложности, возникающие при использовании только одних ключей. Структура этой главы отражает данное изменение направления исследований, поскольку в ней вначале дано общее описание ограничений целостности в достаточно большом объеме, а затем изложение переходит к рассмотрению ключей (которые по-прежнему имеют важное практическое значение).

Прежде всего отметим, что, неформально выражаясь, **ограничение целостности** — это логическое выражение, связанное с некоторой базой данных, результатом вычисления которого всегда должно быть значение TRUE. Подобное ограничение может рассматриваться как формальное выражение некоторого *бизнес-правила* [9.15], хотя иногда сами бизнес-правила (которые в данной главе рассматриваются как представленные всегда на естественном языке) иногда также называют *ограничениями целостности.* Но так или иначе, ниже приведено несколько примеров бизнес-правил, которые основаны на материале базы данных поставщиков и деталей.

1. Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно.
2. Каждый поставщик из Лондона имеет статус 20.
3. Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна иметь синий цвет.
4. Разные поставщики не могут иметь одинаковые номера поставщиков.
5. Каждая поставка выполняется существующим поставщиком.
6. Ни один поставщик со статусом меньше 20 не поставляет любые детали в количе стве больше 500.

Данные примеры будут широко использоваться в настоящей главе.

Очевидно, что ограничения должны быть формально объявлены для СУБД, после чего СУБД должна предписывать их выполнение. Объявление ограничений сводится просто к использованию соответствующих средств языка базы данных, а соблюдение ограничений осуществляется с помощью контроля со стороны СУБД над операциями обновления, которые могут нарушить эти ограничения, и запрещения тех операций, которые их действительно нарушают. Ниже приведено формальное объявление ограничения, показанного в первом из рассматриваемых примеров, на языке Tutorial D.

```
CONSTRAINT SC1
    IS_EMPTY ( S WHERE STATUS < 1 OR STATUS > 100 ) ;
```

Для соблюдения этого ограничения СУБД должна контролировать все операции, в которых осуществляется попытка вставить новую запись с данными о поставщике или изменить статус существующего поставщика [9.5].

Безусловно, при первоначальном объявлении ограничения система должна проверить, удовлетворяет ли ему в настоящий момент база данных. Если это условие не соблюдается, ограничение должно быть отвергнуто; в противном случае оно принимается (т.е. записывается в каталог системы) и начиная с этого момента соблюдается. Кстати, следует отметить, что в данном примере ограничению присвоено имя — SCI ("suppliers constraint one" — ограничение для поставщиков номер один). При условии, что это ограничение принято СУБД, оно будет зарегистрировано в каталоге под этим именем, после чего указанное имя будет появляться в диагностических сообщениях, вырабатываемых системой в ответ на попытки нарушить данное ограничение.

Ниже показаны еще две возможные формулировки ограничения из примера 1, в которых теперь используется версия языка Tutorial D на основе реляционного исчисления (здесь SX — переменная области значений, которая принимает свои значения среди номеров поставщиков).

```
CONSTRAINT SC1
    NOT EXISTS SX ( SX.STATUS < 1 OR SX.STATUS > 100 ) ;

CONSTRAINT SC1
    FORALL SX ( SX.STATUS > 1 AND SX.STATUS < 100 ) ;
```

Безусловно, все эти три формулировки являются эквивалентными. Но в данной главе в качестве основы для значительной части изложения используется реляционное исчисление, а не реляционная алгебра, по причинам, которые станут очевидными для читателя по мере дальнейшего изложения материала. Предлагаем читателю в качестве упражнения подготовить алгебраические версии приведенных примеров, основанных на исчислении.

Безусловно, необходимо также предусмотреть способ уничтожения существующих ограничений, если они больше не требуются. Для этого применяется следующий оператор.

```
DROP CONSTRAINT <constraint name> ;
```

9.2. ПОДРОБНЫЕ СВЕДЕНИЯ ОБ ОГРАНИЧЕНИЯХ ЦЕЛОСТНОСТИ

В общем, ограничения целостности представляют собой ограничения, налагаемые на значения, которые разрешено принимать некоторой переменной, или комбинации переменных[1]. Поэтому тот факт, что конкретная переменная относится к некоторому определенному типу, представляет собой априорное ограничение, налагаемое на рассматриваемую переменную (это ограничение состоит в том, что значения, которые может принимать данная переменная, должны, безусловно, быть значениями этого типа). Например, переменная отношения S (поставщики) ограничивается тем, что должна содержать значения, являющиеся отношениями, в которых каждое значение s# представляет собой номер поставщика (значение типа s#), каждое значение SNAME является именем (значением типа NAME) и т.д.

Но эти простые априорные ограничения, безусловно, не остаются единственными возможными; и действительно, в этом смысле ни один из шести примеров, приведенных в разделе 9.1, не представлял собой априорное ограничение. Еще раз рассмотрим пример 1.

> *Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно.*
>
> Ниже приведена немного более точная формулировка того же ограничения.
>
> *Если s — поставщик, то s имеет значение статуса в пределах от 1 до 100 включительно.*
>
> А здесь показан еще более точный (или более формальный) способ формулировки этого ограничения[2].

---

[1] Как следует из этого замечания, ограничения целостности применяются (по крайней мере, в принципе) к переменным всех типов. Но по очевидным причинам основное внимание в данной книге уделяется именно переменным отношения.

[2] Обратите внимание на то, что в этих формальных примерах не используется синтаксис языка Tutorial D (версии этих примеров на языке Tutorial D приведены ниже). Кроме того, этот синтаксис не полностью соответствует тому, который был определен для реляционного исчисления в главе 8, хотя и близок к нему (особенно к версии, касающейся исчисления доменов).

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR (
IF { S# s#, SNAME sn, STATUS St, CITY sc } ∈ S  THEN St
≥ 1 AND st ≤ 100 )
```

Это формальное выражение можно прочитать следующим образом (на довольно ломаном естественном языке).

Для всех номеров поставщиков si, всех имен sn, всех целых чисел st и всех символьных строк sc, если в переменной отношения поставщиков появляется кортеж: с атрибутами Si si, SNAME sn, STATUS st и CITY sc, то значение st больше или равно 1 и меньше или равно 100.

Возможно, теперь читатель согласится с тем, что действительно нужно было привести эту альтернативную версию примера 1 на естественном языке, которая представлена выше. Дело в том, что теперь становится очевидным следующий факт — эта альтернативная версия, соответствующее формальное выражение и его аналог на ломаном естественном языке, имеют некоторую общую "форму" (как таковую), которая выглядит примерно следующим образом.

Если (IF) в некоторой ***переменной*** *отношения присутствует некоторый кортеж, то* **('THEN,)** *этот кортеж удовлетворяет некоторому условию.*

Эта "форма" является примером **логической импликации** (которую иногда называют *материальной импликацией),* или просто импликации. Эта конструкция уже встречалась в главе 8; она имеет следующую общую форму.

IF p THEN q

Здесь p и q — логические выражения, соответственно, называемые **посылкой** и **следствием.** Общее выражение (т.е. импликация) является ложным, если p — истинно, а q — ложно, в противном случае оно является истинным; иными словами, само выражение IF p THEN q является логическим и оно логически эквивалентно выражению (NOT p) OR q.

Кстати, следует отметить, что показанная выше форма по умолчанию включает необходимый квантор FORALL, поскольку выражение "Если (IF) ... присутствует некоторый кортеж" по сути означает "Для всех (FORALL) присутствующих кортежей, ...".

Теперь перейдем к аналогичному анализу примеров 2—6 (но **при** этом не используя формулировок на естественном языке).

Примечание. Приведенные ниже формулировки не являются единственно возможными, а также не всегда бывают самыми простыми из всех возможных, но они, по крайней мере, правильны. Следует также отметить, что в каждом примере иллюстрируется по меньшей мере одна новая мысль.

2. *Каждый поставщик из Лондона имеет статус 20.*

```
FORALL s# ∈ S#, sn ∈ NAME, st ∈ INTEGER, sc
  ∈ CHAR ( IF { S# s#, SNAME sn, STATUS st,
  CITY sc } ∈ S THEN ( IF sc = 'London' THEN
  st = 20 ) )
```

В этом примере следствие импликации само является импликацией.

3. *Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна иметь синий цвет.*

```
IF
EXISTS p# e P#, pn € NAME, p1 € COLOR, pw € WEIGHT, pc € CHAR
   ( { P# p#, PNAME pn, COLOR pi, WEIGHT pw, CITY pc } € P
) THEN
EXISTS p# € P#, pn € NAME, p1 € COLOR, pw € WEIGHT, pc € CHAR
   ( { P# p#, PNAME pn, COLOR p1, WEIGHT pw, CITY pc }
    € P AND p1 = COLOR ('Blue') )
```

Обратите внимание на то, что нельзя просто применить формулировку "по меньшей мере одна деталь имеет синий цвет", поскольку необходимо учесть тот случай, когда вообще нет деталей.

Примечание. Хотя это может показаться не совсем очевидным, но данный пример действительно соответствует той же общей форме, как и два предыдущих. Ниже приведена альтернативная формулировка, которая позволяет более наглядно подчеркнуть эту мысль.

```
FORALL p# € P#, pn € NAME, p1 € COLOR, pw € WEIGHT, pc €
  CHAR ( IF { P# p#, PNAME pn, COLOR p1, WEIGHT pw, CITY pc
  } € P THEN EXISTS q# € P#, qn € NAME, ql € COLOR,
                        qw € WEIGHT, qc € CHAR
          ( { P# q#, PNAME qn, COLOR ql,
                    WEIGHT qw, CITY qc } G P
           AND ql = COLOR ('Blue') ) )
```

4. *Разные поставщики не могут иметь одинаковые номера поставщиков,*

```
FORALL x# G S#, xn G NAME, xt G INTEGER, xc G CHAR,
  y# G S#, yn G NAME, yt G INTEGER, yc G CHAR ( IF
  { S# x#, SNAME xn, STATUS xt, CITY xc } G S AND
       { S# y#, SNAME yn, STATUS yt, CITY yc } G S THEN (
    IF x# = y# THEN xn = yn AND xt = yt AND xc = yc ) )
```

Данное выражение представляет собой просто формальное изложение того факта, что {S#} является потенциальным ключом (или, во всяком случае, суперключом) для поставщиков; таким образом, ограничения ключа представляют собой лишь частный случай ограничений как таковых. Синтаксическая конструкция KEY {S#} языка Tutorial D может рассматриваться как сокращение для приведенного выше более сложного выражения. (Кстати, здесь заслуживают внимания фигурные скобки. Они подчеркивают, что ключи всегда представляют собой множества атрибутов, даже если рассматриваемое множество содержит только один атрибут, и поэтому атрибуты ключей в этой книге всегда показаны в фигурных скобках, по меньшей мере, в формальных контекстах.)

Примечание. И потенциальные ключи, и суперключи подробно рассматриваются в разделе 9.10.

Кстати, отметим, что этот пример имеет следующую общую форму.

Если (IF) *некоторые кортежи появляются в некоторой переменной отношения, то* (THEN) *эти кортежи удовлетворяют некоторому условию.*

Сравните между собой примеры 2 и 3, которые оба принимают такую же форму, как и пример 1 (вскоре станет очевидно, что это относится и к примеру 5). В отличие от этого, пример 6 принимает следующую общую форму.

Если (IF) *некоторые кортежи появляются* в *некоторых переменных отношения, то* (THEN) *эти кортежи удовлетворяют некоторому условию.*

Эта последняя форма характеризуется тем, что она в общем относится ко всем ограничениям целостности (первые две можно рассматривать как частные случаи этого самого общего случая).

5. *Каждая поставка выполняется существующим поставщиком.*

```
FORALL  s#  ∈  S#,  p#  ∈  P#,  q ∈e  QTY
  (  IF  {  S#  s#,  P# p#,  QTY q  }  ∈
  SP
    THEN EXISTS  sn ∈  NAME,   st  ∈  INTEGER,   sc  ∈  CHAR
              (   {  S#  S#,   SNAME  sn,   STATUS  st,  CITY  SC  }  ∈  S  )  )
```

Это выражение представляет собой формальное утверждение того факта, что {S#} является внешним ключом для поставок, который соответствует потенциальному ключу {S#} для поставщиков. Поэтому ограничения внешнего ключа также являются лишь частным случаем ограничений как таковых (дополнительная информация по этой теме также приведена в разделе 9.10). Следует отметить, что в этом примере участвуют две отдельные переменные отношения, SP и S, а во всех примерах 1—4 участвует только одна переменная отношения[3].

6. *Ни один поставщик со статусом меньше 20 не поставляет любые детали в количест ве больше 500.*

```
FORALL s# e   S#, sn ∈ NAME, st ∈ INTEGER, sc ∈ CHAR,
       p# ∈P#, q ∈ QTY
  ( IF { S#   s#, SNAME sn, STATUS st, CITY sc } ∈ S AND
       { S#s#, P# p#, QTY q } ∈ SP
    THEN st> 20 OR q < QTY ( 500 ) )
```

В данном примере также рассматриваются две разные переменные отношения, но это ограничение не является ограничением внешнего ключа.

Примеры на языке Tutorial D

В завершение данного раздела рассмотрим версии примеров 2—6 на языке Tutorial D (на основе исчисления). При этом применяются обычные соглашения, касающиеся имен переменных области значений.

---

[3] В предыдущем издании данной книги для обозначения ограничения, в котором участвует одна и только одна переменная отношения, использовался термин "ограничение переменной отношения", а для ограничения, в котором участвовало больше переменных отношений, — термин "ограничение базы данных". Но, как будет показано в разделе 9.9, важность такого разграничения в большей степени относится к сфере практики, чем логики, поэтому в дальнейшем изложении эта тема фактически не рассматривается.

1. *Каждый поставщик из Лондона имеет статус 20.*

```
CONSTRAINT SC2
    FORALL SX ( IF SX.CITY = 'London'
           THEN SX.STATUS =20 END IF ) ;
```

Обратите внимание на то, что в языке Tutorial D логические импликации (выражения IF/THEN) включают обозначение конца выражения "END IF".

2. *Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна иметь синий цвет.*

```
CONSTRAINT PC3
    IF EXISTS PX ( TRUE )
    THEN EXISTS PX ( PX.COLOR = COLOR ('Blue') )
    END IF ;
```

3. *Разные поставщики не могут иметь одинаковые номера поставщиков.*

```
CONSTRAINT SC4
    FORALL SX FORALL SY ( IF SX.S#      =    SY.S#
                          THEN SX.SNAME =      SY.SNAME
                          AND SX.STATUS =      SY.STATUS
                          AND SX.CITY   =
                                         SY.CITY
                          END IF ) ;
```

4. *Каждая поставка выполняется существующим поставщиком.*

```
CONSTRAINT SSP5
    FORALL SPX EXISTS SX ( SX.S# = SPX.S# ) ;
```

5. *Ни один поставщик со статусом меньше 20 не поставляет любые детали в количест ве больше 500.*

```
CONSTRAINT SSP6
    FORALL SX FORALL SPX
         ( IF SX.S# = SPX.S#
           THEN SX.STATUS  ≥  20 OR SPX.QTY ≤ 500 END IF ) ;
```

9.3. ПРЕДИКАТЫ И ВЫСКАЗЫВАНИЯ

Еще раз рассмотрим формальную версию примера 1 ("Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно").

```
FORALL s# € S#, sn 6 NAME, st € INTEGER, sc € CHAR
( IF { S# s#, SNAME sn, STATUS st, CITY sc } € S

    THEN st > 1 AND st < 100 )
```

Эта формальная версия представляет собой логическое выражение. Но следует отметить, что в нем[4] участвует переменная, а именно переменная отношения поставщиков S. Поэтому мы не можем определить, каковым является значение этого выражения (т.е. не можем определить, какое логическое значение оно принимает), до тех пор, пока не подставим конкретное значение вместо этой переменной (вообще говоря, в действительности

[4] В этом выражении участвуют также несколько переменных области значений, но, как было показано в главе 8, переменные области значений не являются переменными в том смысле, какой им придается в языке программирования, а в данной главе термин "переменная" рассматривается именно в том смысле, который ему придается в языке программирования (если явно не указано иное).

различные подстановки должны приводить к получению разных истинностных значений). Иными словами, данное выражение является **предикатом,** а переменная S служит **формальным параметром** этого предиката. Поэтому, когда возникает необходимость "конкретизировать" данный предикат (иначе говоря, когда нам требуется проверить ограничение), мы предоставляем в качестве **фактического параметра** такое отношение, которое является текущим значением переменной отношения S (а переменная отношения S — единственный формальный параметр), и только после этого появляется возможность вычислить это выражение.

Теперь, после конкретизации предиката (которая, по сути, сводится к замене его единственного формального параметра некоторым фактическим параметром), мы приходим к некоторому выражению с истинностным значением, которое вообще не включает переменных, а содержит лишь значения. Аналогичные замечания касаются и ограничений, которые распространяются на два, три, четыре или на любое другое количество переменных отношения; так или иначе, когда возникает необходимость вычислить выражение (т.е. когда требуется проверить ограничение), достаточно заменить каждый формальный параметр отношением, которое является текущим значением соответствующей переменной отношения, после чего мы приходим к выражению с истинностным значением, которое вообще не имеет переменных или, иными словами, является высказыванием. Вне всякого сомнения, любое *высказывание* может быть либо истинным, либо ложным (его можно рассматривать как *вырожденный* предикат, т.е. предикат, для которого множество формальных параметров является пустым, как было описано в предыдущей главе). Ниже приведено несколько простых примеров высказываний.

- Солнце — это звезда.
- Луна — это звезда.
- Солнце находится от Земли дальше, чем Луна.
- Джордж Буш победил на президентских выборах в США в 2000 году.

Выяснение того, какие из этих высказываний являются истинными, а какие ложными, оставляем в качестве упражнения для читателя. Но следует отметить, что не все высказывания являются истинными; широко распространенная ошибка состоит в том, что все высказывания рассматриваются исключительно как истинные.

На основании материала, приведенного в этом разделе, можно сделать следующие выводы: ограничение, определенное формально, становится предикатом, но при проверке этого ограничения вместо формальных параметров предиката подставляются фактические параметры, в результате чего предикат сводится к высказыванию, и к такому высказыванию затем предъявляется требование, чтобы его значение было равно TRUE.

9.4. ПРЕДИКАТЫ ПЕРЕМЕННОЙ ОТНОШЕНИЯ И ПРЕДИКАТЫ БАЗЫ ДАННЫХ

Безусловно, что в общем любая конкретная переменная отношения может становиться объектом действия многих ограничений. Предположим, что R — переменная отношения. В таком случае **предикатом переменной отношения** R является результат применения логической операции "И", или операции конъюнкции ко всем ограничениям, которые распространяются на переменную отношения R (иными словами, в которых она упоминается). Следует учитывать, что здесь возникает определенная опасность противоречивого

толкования: как уже было сказано, каждое отдельное ограничение само является предикатом в полном смысле этого понятия, но **предикат** переменной отношения для R — это конъюнкция *всех* отдельных предикатов, которые относятся к R. Например, если для упрощения предположим, что только шесть ограничений, описанных в разделе 9.1, относятся к базе данных поставщиков и деталей (не считая априорных ограничений), то предикат переменной отношения для поставщиков является конъюнкцией ограничений с номерами 1, 2, 4, 5 и 6, а предикат переменной отношения для поставок — конъюнкцией предикатов с номерами 5 и 6. Обратите внимание, что эти два предиката переменной отношения в некотором смысле "перекрываются", поскольку они имеют некоторые общие составляющие ограничения[5].

Теперь допустим, что R — переменная отношения, а RP — предикат переменной отношения для R. Итак, безусловно, ни в коем случае нельзя допускать, чтобы переменная R приобретала такое значение, что его подстановка в RP вместо R (а также любая другая необходимая подстановка фактических параметров вместо формальных параметров, которая должна быть выполнена в RP), становилась причиной того, чтобы предикат RP принимал значение FALSE. Итак, теперь мы можем ввести **золотое правило** (первую версию), которое имеет большое значение при анализе ограничзний целостности.

Ни одна операция обновления не должна приводить к присваиванию любой переменной отношения такого значения, которое вызывает то, что предикат этой переменной отношения получает значение FALSE.

Теперь предположим, что D является базой данных[6] и что D включает переменные отношения Rl, R2, ..., Rn (и только эти переменные отношения). Допустим, что предикатами переменной отношения для этих переменных отношения, соответственно, являются RP1, RP2, ..., RPn. Тогда **предикат базы данных** для D, скажем, DP, представляет собой конъюнкцию всех таких предикатов переменной отношения.

```
DP ≡ RP1 AND RP2 AND ... AND RPn
```

Ниже приведена расширенная (более общая и фактически окончательная) версия золотого правила.

Ни одна операция обновления не должна приводить к присваиванию любой базе данных такого значения, которое вызывает то, что предикат этой базы данных получает значение FALSE.

---

[5] В предыдущем издании данной книги было приведено определение, что предикат переменной отношения для переменной отношения R представляет собой конъюнкцию всех ограничений переменной отношения, которые относятся к R (где, как указано в разделе 9.2, ограничением переменной отношения называется ограничение, в котором упоминается только одна переменная отношения). Но теперь, в соответствии с [3.3], мы принимаем определение, что предикат переменной отношения R является конъюнкцией всех ограничений, а не только ограничений переменной отношения, которые относятся к R. Автор приносит свои извинения всем, кто считает такое изменение в терминологии причиной дополнительной путаницы.

[6] D, безусловно, также является переменной (см. аннотацию к [3.3]) и поэтому служит объектом действия ограничений целостности.

Безусловно, предикат базы данных принимает значение FALSE тогда и только тогда, когда по меньшей мере один из ее составляющих предикатов переменной отношения принимает это значение. А сам предикат переменной отношения принимает значение FALSE тогда и только тогда, когда по меньшей мере одна из составляющих ограничений принимает это значение.

Примечание. Как уже было сказано, два разных предиката переменной отношения, RPi и RPj (i ≡ j), могут иметь некоторые общие составляющие ограничения. Из этого следует, что одно и то же ограничение может появиться в предикате базы данных DP несколько раз. С логической точки зрения такое положение дел не влечет за собой каких-либо затруднений, поскольку если c — ограничение, то выражение c AND c логически эквивалентно просто c. Итак, хотя, безусловно, в такой ситуации желательно, чтобы система вычисляла ограничение c один, а не два раза, эта проблема относится к реализации, а не к модели.

9.5. ПРОВЕРКА ОГРАНИЧЕНИЙ

В данном разделе рассматриваются две темы. Одна из них относится к реализации, а другая — к модели, и обе эти темы касаются вопроса о том, как фактически должны проверяться объявленные ограничения. Вначале рассмотрим проблему реализации. Еще раз вернемся к примеру 1, в котором, как известно, фактически утверждается, что если некоторый кортеж присутствует в переменной отношения S, то этот кортеж должен удовлетворять определенному условию (в данном случае условию "статус должен находиться в пределах от 1 до 100"). В частности, следует отметить, что в этом ограничении речь идет о кортежах в переменной отношения. Поэтому очевидно, что при осуществлении попытки вставить новый кортеж с данными о поставщике со статусом (скажем) 200, должна происходить описанная ниже последовательность событий.

1. Вставка нового кортежа.
2. Проверка ограничения.
3. Отмена обновления (поскольку проверка окончилась неудачей).

Но это же нелепо! Безусловно, что необходимо обнаружить эту ошибку еще до того, как будет выполнена вставка. Поэтому реализация, очевидно, должна отвечать такому требованию, что в ней формальное выражение ограничения должно использоваться для формирования подходящей процедуры (процедур) проверки, которая должна применяться к кортежам, предлагаемым для вставки, еще до того, как фактически будет выполнена эта вставка.

В принципе, данный процесс формирования процедуры проверки является довольно простым. Предположим, что предикат базы данных включает ограничение в следующей форме.

```
IF { S# s#, SNAME sn, STATUS st, CITY sc } € S THEN ...
```

Это означает, что если посылка в некоторой импликации в составе всего предиката находится в форме, соответствующей утверждению "Некоторый кортеж присутствует в S", то следствие в этой импликации, по сути, представляет собой ограничение на кортежи, предназначенные для вставки в переменную отношения S.

Примечание. Кстати, следует отметить, что если база данных разработана в соответствии с принципом ортогонального проектирования (см. главу 13), и при условии, что СУБД имеет информацию обо всех применимых ограничениях, то любой конкретный кортеж приходится проверять на соответствие не больше чем одному предикату переменной отношения, поскольку он будет представлять собой приемлемого кандидата для вставки с помощью операции INSERT, самое большее, в одну переменную отношения.

Теперь перейдем к проблеме модели (которая, безусловно, является более фундаментальной). Снова рассмотрим приведенное выше золотое правило.

Ни одна операция обновления не должна приводить к присваиванию любой базе данных такого значения, которое вызывает то, что предикат этой базы данных получает значение FALSE.

Хотя эта мысль в разделе 9.4 явно не подчеркивалась, необходимо учитывать, что из этого правила следует требование о немедленном выполнении проверок всех ограничений. С чем это связано? Дело в том, что это правило сформулировано в терминах операций обновления, а не в терминах транзакций (см. следующий абзац). Поэтому, по сути, **золотое правило** требует соблюдения ограничений целостности на этапе перехода от выполнения одного оператора к выполнению другого[7] и в нем не подразумевается понятие *отложенной* проверки ограничений целостности или проверки во время выполнения операции фиксации COMMIT.

Итак, читатель мог уже обнаружить, что выраженная здесь позиция, согласно которой все проверки должны выполняться немедленно, является весьма нетрадиционной; в литературе (включая предыдущие издания этой книги) чаще всего утверждается или просто предполагается, что *единицей контроля целостности* является транзакция и что, по меньшей мере, часть проверок следует откладывать до конца транзакции (т.е. ко времени выполнения операции COMMIT). Но существуют серьезные причины, по которым транзакции являются неприменимыми в качестве *единицы контроля целостности,* и такой единицей вместо них должны служить операторы. К сожалению, эти причины невозможно доходчиво объяснить, не дав вначале более подробного описания всего, что вообще связано с понятием транзакции. Поэтому отложим подробное обсуждение этой темы до главы 16; до этой главы мы просто предполагаем, что требование немедленной проверки является логически оправданным, не пытаясь дополнительно обосновать нашу позицию. (Но один важный довод в пользу позиции автора можно найти в аннотации к [9.16] в конце данной главы.)

9.6. СОПОСТАВЛЕНИЕ ВНУТРЕННИХ И ВНЕШНИХ ПРЕДИКАТОВ

Как было показано выше, каждая переменная отношения имеет предикат переменной отношения, а вся база данных имеет предикат базы данных. Безусловно, все рассматриваемые предикаты являются такими, о которых "в системе имеется информация".

---

[7] По сути, в этом изложении требуется немного более высокая точность, но для внесения таких уточнений будет необходимо в определенной степени коснуться темы конкретного языка базы данных, используемого в рассматриваемых операциях. Но для описания данной темы достаточно отметить, что ограничения целостности должны удовлетворяться к концу выполнения любого оператора, который не содержит в себе синтаксически вложенных других операторов. Или, неформально выражаясь, "Ограничения должны удовлетворяться на этапах обработки базой данных точек с запятыми, отделяющих друг от друга операторы ".

Это означает, что подобные предикаты определены формально (т.е. они фактически входят в состав определения базы данных), кроме того, их соблюдение контролируется системой. По этим причинам иногда удобно называть рассматриваемые предикаты **внутренними** предикатами, поскольку переменные отношения и базы данных в принципе имеют также **внешние** предикаты, описание которых приведено ниже[8].

Первое и наиболее важное замечание состоит в том, что внутренние предикаты — это формальные конструкции, тогда как внешние предикаты — просто неформальные конструкции. Внутренние предикаты (это — не строгая формулировка) являются описанием того, что означают данные для системы; в отличие от этого, внешние предикаты описывают, что означают данные для пользователя. Безусловно, пользователи должны знать не только о внешних предикатах, но и о внутренних, однако еще раз повторим, что система должна иметь информацию о внутренних предикатах (и действительно быть способной учитывать лишь внутренние предикаты). Можно фактически неформально утверждать, что любой внутренний предикат представляет собой аппроксимацию в системе соответствующего внешнего предиката.

В дальнейшем изложении, если не указано иное, речь будет идти исключительно о переменных отношения. Итак, как уже было сказано, внешним предикатом данной конкретной переменной отношения по сути является то, что эта переменная отношения означает для пользователя. Например, в случае переменной отношения поставщиков S внешний предикат может формулироваться таким образом, как показано ниже.

Поставщик с указанным номером поставщика (S#) работает по контракту, имеет указанное имя (SNAME) и указанный статус (STATUS), а также находится в указанном городе (CITY). Кроме того, значение его статуса должно находиться в пределах от 1 до 100 включительно и должно быть равно 20, если городом является Лондон. К тому же никакие два различных поставщика не имеют один и тот же номер поставщика.

Но для удобства дальнейшего обсуждения заменим этот предикат более простым, приведенным ниже.

Поставщик S#работает по контракту, имеет имя SNAME, имеет статус STATUS и находится в городе CITY.

(Вообще говоря, все внешние предикаты являются лишь неформальными, поэтому мы имеем право формулировать их сколь угодно просто или сложно, безусловно, в разумных пределах.)

Теперь отметим, что приведенное выше утверждение действительно является предикатом, поскольку оно имеет четыре формальных параметра (S#, SNAME, STATUS и CITY), соответствующие четырем атрибутам переменной отношения[9], и после подстановки вместо

---

[8] Внешние предикаты уже рассматривались в главах 3 и 6, но тогда они именовались просто предикатами. До сих пор фактически по умолчанию термин "предикат" использовался во всей данной книге как обозначение именно внешнего предиката. Единственным важным исключением было обсуждение операции ограничения, приведенное в главе 7, где условие ограничения было названо предикатом; так оно и есть, но это — не внешний предикат.

[9] В этом изложении термин "формальный параметр" используется немного в ином смысле, чем в разделах 9.3 и 9.4. В этих разделах формальные параметры (и соответствующие фактические параметры) обозначали целые отношения, а в данном изложении они обозначают отдельные атрибуты.

этих формальных параметров фактических параметров соответствующих типов образуется высказывание (т.е. некий логический объект, безусловно принимающий либо истинное, либо ложное значение). Поэтому каждый кортеж, присутствующий в переменной отношения S в любой указанный момент времени, может рассматриваться как обозначающий такое высказывание, полученное путем конкретизации данного предиката. Кроме того (что очень важно!), в данный момент времени рассматриваются как истинные лишь данные конкретные высказывания (т.е. те высказывания, которые теперь представлены кортежами в переменной отношения S). Например, рассмотрим следующий кортеж.

```
{ S# S#('S1'), SNAME NAME('Smith'), STATUS 20, CITY 'London' }
```

Если этот кортеж присутствует в переменной отношения S в некоторый момент времени, то следует считать *истинным фактом,* что в данный момент существует поставщик, работающий по контракту, который имеет номер поставщика s1, имя Smith, статус 20 и находится в Лондоне. Вообще говоря, справедлива следующая формула:

```
IF ( s S ) = TRUE THEN XPS ( s ) = TRUE
```

В этой формуле применяются описанные ниже обозначения. ■

s — это кортеж в следующей форме:

```
{ S# s#, SNAME sn, STATUS st, CITY sc }
```

Здесь s# — значение типа S#, sn — значение типа NAME, st — значение типа INTEGER и sc — значение типа CITY.

- XPS — внешний предикат для поставщиков.
- XPS(S) — это высказывание, полученное путем конкретизации предиката XPS значениями формальных параметров S# = s#, SNAME = sn, STATUS = st и CITY = sc.

Но как было отмечено в главе 6, применительно к внешним предикатам можно принять более широкие допущения, чем ко внутренним. Говоря точнее, по отношению к ним принимается предположение о замкнутости мира, в котором утверждается, что если некоторый кортеж, допустимый по всех прочим признакам, не присутствует в переменной отношения в некоторый момент времени, то на основании принятого соглашения соответствующее высказывание в данный момент рассматривается как ложное. Например, предположим, что в некоторый конкретный момент времени в переменной отношения S не присутствует следующий кортеж.

```
{ S# S#('S6'), SNAME NAME('Lopez'), STATUS 30, CITY 'Madrid' }
```

В этом случае данный факт следует рассматривать таким образом, что в данный момент не существует работающего по контракту поставщика с номером поставщика S6, который имеет имя Lopez, статус 30 и находится в Мадриде. Вообще говоря, справедлива следующая формула:

```
IF ( s ∈ S ) = FALSE THEN XPS ( s ) = FALSE
```

В более краткой форме она выглядит таким образом:

```
IF NOT ( s ∈ S ) THEN NOT XPS ( s )
```

На основании изложенного выше можно вывести следующую формулу:

```
s e s ≡ XPS ( s )
```

Иными словами, любой конкретный кортеж присутствует в конкретной переменной отношения в конкретный момент времени тогда и только тогда, когда в данный момент времени присутствие данного кортежа приводит к тому, что внешний предикат этой переменной отношения принимает значение TRUE. Из этого следует, что данная переменная отношения содержит те и только те кортежи, которые соответствуют истинным конкретизациям внешнего предиката данной переменной отношения в рассматриваемый момент времени.

9.7. СРАВНЕНИЕ ПОНЯТИЙ ПРАВИЛЬНОСТИ И НЕПРОТИВОРЕЧИВОСТИ

По определению, внешние предикаты и высказывания, полученные путем конкретизации таких предикатов, не известны (и фактически не могут быть известными) в системе. Например, система не может иметь информации о том, что некий "поставщик" где-то "находится", или о том, что означает утверждение, будто "поставщик" имеет "некоторый статус" (и т.д.). Все эти вопросы относятся к области интерпретации фактических данных, поскольку данные имеют смысл только для пользователя, но не для системы. Рассмотрим более конкретный пример. Допустим, что в одном и том же кортеже оказались данные о номере поставщика S1 и названии города Лондона. Тогда пользователь может рассматривать этот факт так, как будто он означает, что поставщик S1 находится в Лондоне[10], но (повторяем) нет никакого способа, с помощью которого можно было бы вынудить систему прийти к аналогичным выводам.

Более того, даже если бы системе можно было сообщить, что подразумевается под утверждением, будто *поставщик где-то находится,* все равно система не могла бы априори иметь информацию о том, являются ли истинными данные, сообщенные пользователем! Когда пользователь вносит в систему сведения о том, что поставщик S1 находится в Лондоне (обычно путем выполнения оператора INSERT), система не может применить какой-либо способ, чтобы определить, действительно ли эти сведения являются истинными. Все, что может сделать система, — проверить, не приведет ли ввод этой информации к нарушению каких-либо ограничений (т.е. не вызовет ли это такую конкретизацию любого внутреннего предиката, при которой будет получено значение FALSE). При условии, что этого не происходит, система должна принять эти сведения и с этих пор рассматривать их как истинные (по меньшей мере, пока пользователь не сообщит системе, что данные сведения больше не являются истинными, обычно путем выполнения оператора DELETE).

Кстати, следует отметить, что изложенное выше наглядно показывает, почему предположение о замкнутости мира не относится к внутренним предикатам. А именно, кортеж может удовлетворять внутреннему предикату для данной переменной отношения и

---

[10] Этот факт может также означать, что поставщик S1 в данный момент проживает в Лондоне, или поставщик S1 имеет офис в Лондоне, или поставщик S1 не имеет офиса в Лондоне, а также может иметь бесконечное количество других возможных толкований (соответствующих бесконечному количеству возможных внешних предикатов).

вместе с тем не присутствовать в этой переменной отношения на законных основаниях, поскольку он не соответствует истинному высказыванию в реальном мире.

Подведем итог данному обсуждению, высказав такое неформальное утверждение, что внешний предикат для определенной переменной отношения представляет собой **намеченную интерпретацию** для этой переменной отношения. Как таковая, намеченная интерпретация имеет смысл только для пользователя, но не для системы. Поэтому можно привести еще одно неформальное утверждение о том, что внешний предикат для указанной переменной отношения является **критерием приемлемости обновлений** для рассматриваемой переменной отношения; это означает, что внешний предикат определяет (по меньшей мере, в принципе), можно ли разрешить успешно выполнить затребованные операции INSERT, DELETE или UPDATE на этой переменной отношения. Поэтому в идеальной ситуации система должна иметь информацию о внешнем предикате для каждой переменной отношения, чтобы обладать способностью успешно действовать при всех возможных попытках обновить эту переменную отношения. Но, как было показано выше, эта цель является недостижимой; система не может иметь информацию о внешнем предикате ни для одной переменной отношения. Но система имеет информацию о достаточно качественной аппроксимации этого внешнего предиката, поскольку ей известен соответствующий внутренний предикат, и она следит за его соблюдением. Поэтому прагматическим "критерием приемлемости обновлений" служит внутренний предикат, а не внешний (а внешний может быть таковым лишь в идеальной ситуации). Еще одна, более строгая формулировка этого утверждения приведена ниже.

Система может контролировать только непротиворечивость, но не истинность хранимых в ней данных.

Это означает, что система не может гарантировать наличие в базе данных только истинных высказываний; все, что она может сделать, — это гарантировать отсутствие каких-либо данных, вызывающих нарушение ограничений целостности (т.е. гарантировать то, что она не содержит каких-либо данных, не совместимых с этими ограничениями). Но, к сожалению, истинность и непротиворечивость — не одно и то же! Фактически можно отметить следующее:

- если база данных содержит только истинные высказывания, то она непротиворе чива, но из этого не следует обратное утверждение;
- если база данных не является непротиворечивой, то она содержит по меньшей ме ре одно ложное высказывание, но из этого не следует обратное утверждение.

Приведенные выше формулировки можно более кратко изложить следующим образом: из того, что данные являются правильными, следует, что они непротиворечивы (но не обратное), а из того, что данные не являются непротиворечивыми, следует, что они неправильны (но не обратное). Здесь под словом "правильные" подразумевается, что в базе данных содержатся правильные данные тогда и только тогда, когда она полностью отражает истинное состояние дел в реальном мире.

9.8. ОГРАНИЧЕНИЯ ЦЕЛОСТНОСТИ И ПРЕДСТАВЛЕНИЯ

Важно отметить, что почти все рассуждения, приведенные выше в этой главе, касались в общем всех, а не только базовых переменных отношения. В частности, они касаются и представлений (которые являются виртуальными переменными отношения).

Поэтому представления также служат объектом действия ограничений и имеют предикаты переменной отношения (как внутренние, так и внешние). Например, предположим, что определено представление путем применения к переменной отношения поставщиков операции проекции по атрибутам S#, SNAME и STATUS (что фактически приводит к удалению атрибута CITY). В таком случае внешний предикат для этого представления определен примерно так, как показано ниже.

Существует некоторый город CITY, такой что работающий по контракту поставщик S# имеет имя SNAME, статус STATUS и находится в городе CITY.

Следует отметить, что сам этот предикат, как и требуется, имеет три формальных параметра, соответствующих трем атрибутам представления, а не четыре (атрибут CITY больше не является параметром, а выполняет функции *связанной переменной* в силу того, что к нему применен квантор существования в виде утверждения "существует некоторый город"). Еще один способ выразить ту же мысль (возможно, более наглядный) состоит в следующем. Можно отметить, что рассматриваемый предикат логически эквивалентен приведенному ниже.

Поставщик s#работает по контракту, имеет имя SNAME, имеет статус STATUS и находится в некотором городе.

Вполне очевидно, что данная версия предиката имеет только три формальных параметра. А что можно в этом случае утверждать в отношении внутреннего предиката? Еще раз рассмотрим шесть примеров, применяемых в данной главе.

1. Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно.
2. Каждый поставщик из Лондона имеет статус 20.
3. Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна
   быть синего цвета.
4. Разные поставщики не могут иметь одинаковые номера поставщиков.
5. Каждая поставка выполняется существующим поставщиком.
6. Ни один поставщик со статусом меньше 20 не поставляет любые детали в количе стве больше 500.

Предположим, что рассматриваемое представление (проекция переменной отношения поставщиков по атрибутам s#, SNAME и STATUS) называется SST. В таком случае, если речь идет о представлении SST, то пример 3, безусловно, к нему не относится, поскольку в нем рассматриваются детали, а не поставщики. А что касается других примеров, то каждый из них в определенной степени связан с представлением SST, но в несколько модифицированной форме. В частности, ниже показана модифицированная форма примера 5.

```
FORALL s# ∈ S#, p# ∈ P#, q ∈
  QTY  ( IF { S# s#, P# p#, QTY q
  } ∈ SP
    THEN EXISTS sn ∈ NAME, St ∈ INTEGER
           ( { S# s#, SNAME sn, STATUS st } ∈ SST ) )
```

Изменения наблюдаются в третьей и четвертой строках. В них все упоминания об атрибуте CITY удалены, а ссылка на s заменена ссылкой на SST. Обратите внимание на то, что это ограничение для SST может рассматриваться как производное от соответствующего ограничения для S точно так же, как сама переменная отношения SST происходит от переменной отношения S (и в конечном итоге сам внешний предикат для SST является производным от внешнего предиката для S)[11].

Аналогичные замечания относятся непосредственно к примерам 1, 2, 4 и 6. Но, как показано ниже, пример 2 немного сложнее, поскольку он требует введения конструкции EXISTS, соответствующей атрибуту, который был удален в результате выполнения операции проекции.

```
FORALL s# € S#, sn € NAME, st € INTEGER
  ( IF { S# s#, SNAME sn, STATUS st } €
    SST THEN EXISTS SC € CHAR
      ( { S# s#, SNAME sn, STATUS st, CITY sc } €
        S AND ( IF sc = 'London' THEN st = 20 ) )
```

Однако и в данном случае это ограничение может рассматриваться как производное от соответствующего ограничения для S.

9.9. СХЕМА КЛАССИФИКАЦИИ ОГРАНИЧЕНИЙ

В данном разделе будет кратко намечена схема классификации для ограничений (по сути, это та же схема, которая была принята в [3.3]). Кратко отметим, что здесь предусмотрено распределение ограничений по четырем основным категориям: ограничения базы данных, ограничения переменной отношения, ограничения атрибута и ограничения типа. Краткие определения этих ограничений приведены ниже.

- *Ограничением базы данных* называется ограничение на значения, которые разреше но принимать указанной базе данных.
- *Ограничением переменной отношения* называется ограничение на значения, которые разрешено принимать указанной переменной отношения.
- *Ограничением атрибута* называется ограничение на значения, которые разрешено принимать указанному атрибуту.
- *Ограничение типа* представляет собой не что иное, как определение множества значений, из которых состоит данный тип.

Но задачу описания этих ограничений проще всего выполнить, рассматривая их в обратном порядке.

Ограничения типа

До настоящего времени в этой главе ограничения типа вообще не упоминались. Но эти ограничения достаточно подробно рассматривались в главе 5, поэтому приведенное ниже описание предназначено лишь для того, чтобы напомнить читателю, о чем было

[11] Обратите также внимание на то, что это производное ограничение фактически является ограничением внешнего ключа от базовой переменной отношения к представлению! (См. раздел 9.10.)

сказано в той главе. Прежде всего, еще раз отметим, что *ограничение типа* является не чем иным, как спецификацией значений, из которых состоит рассматриваемый тип. Ниже приведен один пример (повторение примера из главы 5).

```
TYPE WEIGHT POSSREP { D DECIMAL (5,1)
                      CONSTRAINT D > 0.0 AND D < 5000.0 } ;
```

Он имеет следующий смысл: допустимыми значениями типа WEIGHT ЯВЛЯЮТСЯ такие и только такие значения, которые можно представить десятичными числами с точностью пять цифр и с одной цифрой после десятичной точки, где рассматриваемое десятичное число больше нуля и меньше 5000.

Теперь должно быть очевидно, что единственный способ, благодаря которому любое выражение может принять значение типа WEIGHT, СОСТОИТ В использовании некоторого вызова селектора WEIGHT. Поэтому единственная возможность нарушения в любом таком выражении ограничения типа WEIGHT состоит в том, что это ограничение будет нарушено в вызове селектора. Из этого следует, что ограничения типа всегда могут рассматриваться (по меньшей мере, концептуально) как проверяемые во время некоторого вызова селектора. Например, рассмотрим следующий вызов селектора для типаWEIGHT.

```
WEIGHT ( 7500.0 )
```

Это выражение активизирует исключение во время прогона программы (которое словесно выражается как "нарушение ограничения типа WEIGHT — значение выходит за допустимые пределы").

На основании сказанного можно утверждать, что ограничения типа всегда проверяются немедленно, а значит, в частности, считать, что ни одна переменная отношения ни при каких условиях не получит значения для любого атрибута любого кортежа, которое не имело бы соответствующего типа (безусловно, речь идет о системе, которая поддерживает ограничения типа должным образом!).

Поскольку ограничения типа фактически представляют собой просто спецификацию значений, из которых состоит рассматриваемый тип, в языке Tutorial D предусмотрена возможность увязывать такие ограничения с определением соответствующего типа и обозначать их с помощью соответствующего имени типа. Из этого следует, что ограничение типа может быть удалено только путем удаления самого определения типа.

Ограничения атрибута

Ограничения атрибута, по сути, состоят в том, что в разделе 9.2 было названо *априорными ограничениями;* иными словами, ограничением атрибута фактически является просто объявление, которое гласит, что указанный атрибут указанной переменной отношения имеет указанный тип. Например, снова рассмотрим определение переменной отношения поставщиков, которое приведено ниже.

```
VAR S BASE RELATION
    { S#     S#,   SNAME
      NAME, STATUS
      INTEGER, CITY
      CHAR } ... ;
```

В этой переменной отношения на значения атрибутов s#, SNAME, STATUS и CITY наложено ограничение, согласно которому они должны, соответственно, иметь типы S#,

NAME, INTEGER и CHAR. Иными словами, ограничения атрибута входят в состав определения рассматриваемого атрибута и сами могут быть обозначены с помощью соответствующего имени атрибута. Из этого следует, что ограничение атрибута может быть уничтожено только путем уничтожения самого атрибута (а это на практике обычно означает уничтожение содержащей этот атрибут переменной отношения).

Примечание. В принципе, любая попытка ввести в базу данных (с помощью операции INSERT или UPDATE) значение атрибута, не имеющее допустимого типа, должна быть просто отвергнута. Но на практике подобная ситуация не должна даже возникать, если система действительно контролирует соблюдение ограничений типа, как описано в предыдущем подразделе.

Ограничения переменной отношения и базы данных

До сих пор в настоящей главе в основном рассматривались только ограничения переменной отношения и базы данных; различие между ними состоит в том, что ограничение переменной отношения распространяется на одну и только одну переменную отношения, а ограничение базы данных распространяется на две или больше переменных отношения. Но как было указано в разделе 9.2, с теоретической точки зрения это различие не является слишком важным (хотя и может иметь смысл стремление его учитывать с точки зрения практики).

Но мы еще до сих пор не касались одной темы, которая заключается в том, что некоторые ограничения переменной отношения или базы данных могут представлять собой ограничения перехода. **Ограничением перехода** называется ограничение, регламентирующее допустимые переходы для определенной переменной (в частности, для конкретной переменной отношения или базы данных), которые она может выполнять, переходя от одного значения к другому[12]; например, семейное положение любого лица может измениться со значения "никогда не состоял в браке" на значение "состоит в браке", но возврат к прежнему состоянию невозможен. При условии, что предусмотрен способ включить в одно выражение одновременно и то значение, которое рассматриваемая переменная имела до любой произвольной операции обновления, и значение той же переменной после того же обновления, то существует возможность сформулировать любое желаемое ограничение перехода. Ниже приведен один пример формулировки подобного ограничения ("статус поставщика ни в коем случае не должен уменьшаться").

```
CONSTRAINT TRC1 FORALL SX'
    FORALL SX { SX'.S# ≠ SX.S# OR SX'.STATUS < SX.STATUS ) ;
```

Пояснение. Примем такое соглашение, что имя переменной области значений со штрихом, такое как SX’ в данном примере, обозначает ссылку на соответствующую переменную отношения, которой она была до выполнения рассматриваемой операции обновления. Таким образом, ограничение, показанное в этом примере, можно описать следующим образом: "Если SX' — кортеж поставщика перед выполнением обновления, то не должен существовать кортеж поставщика SX после обновления с таким же номером поставщика, как SX’, и со значением статуса меньше чем SX'".

---

[12] Ограничения, не относящиеся к типу ограничений перехода, иногда называют ограничениями состояния.

Следует отметить, что приведенное выше ограничение TRC1 является офаничением перехода переменной отношения (оно касается только одной переменной отношения — S). Ниже для сравнения приведено ограничение перехода базы данных ("Общее количество деталей любого конкретного типа, имеющихся у всех поставщиков, ни в коем случае не должно уменьшаться").

```
CONSTRAINT
    TRC2
    FORALL PX
       SUM ( SPX' WHERE SPX'.P# = PX.P#, QTY
       ) SUM { SPX WHERE SPX .P# = PX.P#, QTY
       ) ;
```

Понятие ограничений перехода не распространяется на ограничения типа или атрибута.

9.10. КЛЮЧИ

Как было отмечено в разделе 9.1, в реляционной модели всегда придавалось большое значение понятию ключей, хотя, как было показано выше, ключи фактически представляют собой лишь частный случай более общего феномена (пусть даже и важный с точки зрения практики). В этом разделе рассматриваются именно ключи.

Потенциальные ключи

Допустим, что R — переменная отношения. По определению, множество всех атрибутов R обладает свойством уникальности; это означает, что в любой конкретный момент времени никакие два кортежа в значении R не являются дубликатами друг друга. На практике часто встречается ситуация, в которой определенное собственное подмножество множества всех атрибутов R также обладает свойством уникальности, например, в случае переменной отношения поставщиков S таким свойством обладает подмножество, содержащее только атрибут s#. На основании этих соображений может быть сформулировано приведенное ниже неформальное определение, продиктованное интуицией[3].

- Допустим, что к — множество атрибутов переменной отношения R. В таком случае к является потенциальным ключом для R тогда и только тогда, когда оно обладает одновременно двумя перечисленными ниже свойствами.
  а) *Уникальность.* Ни одно допустимое значение R никогда не содержит два разных кортежа с одним и тем же значением к.
  б) *Несократимость.* Никакое строгое подмножество к не обладает свойством уникальности.

Каждая переменная отношения имеет по меньшей мере один потенциальный ключ. Свойство уникальности подобных ключей не требует пояснения. А что касается свойства несократимости, то его смысл состоит в том, что если бы был определен *потенциальный ключ,* не являющийся несократимым, то система не имела бы информации об истинном состоянии дел и поэтому не могла бы следить за выполнением соответствующего ограничения целостности должным образом. Например, предположим, что в качестве потенциального

[13] Следует отметить, что это определение касается именно переменных отношения; аналогичное понятие может быть также определено для значений отношения [3.3], но переменные отношения — наиболее важный случай. Заслуживает также внимания то, что и в этом случае мы исходим из понятия равенства кортежей (точнее, это понятие применяется в определении свойства уникальности).

ключа для поставщиков определена комбинация атрибутов {S#,CITY} вместо одного только атрибута {S#}. В таком случае система не могла бы следить за соблюдением ограничения, согласно которому номера поставщиков являются "глобально" уникальными, а обладала бы способностью контролировать только более слабое ограничение, состоящее в том, что номера поставщиков являются уникальными "локально", в пределах одного города. Кроме всего прочего, по этой причине в приведенном выше определении требуется, чтобы потенциальные ключи не содержали никаких атрибутов, которые не обеспечивают решения задачи уникальной идентификации[14].

Следует отметить, что понятие несократимости, определение которого приведено выше, в литературе (включая первые издания этой книги) часто обозначается термином *минимальность.* Но в действительности термин *минимальность* является не вполне подходящим, поскольку из утверждения, что потенциальный ключ К1 является *минимальным,* не следует, что не может быть найден другой потенциальный ключ, К2, с меньшим количеством компонентов; вполне возможно, что, например, К1 имеет четыре компонента, а К2 — только два. Поэтому автор придерживается термина *несократимость.*

В языке Tutorial D для определения потенциального ключа рассматриваемой переменной отношения в объявлении переменной отношения используется следующий синтаксис.

KEY { *<attribute name commalist>* }

Ниже приведено еще несколько примеров.

```
VAR S BASE RELATION
  { S#     S#, SNAME
  NAME, STATUS
  INTEGER, CITY
  CHAR } KEY { S# }
  ;
```

Примечание. В предыдущих главах это определение было приведено с конструкцией PRIMARY KEY, а не с неуточненной конструкцией KEY. Дополнительные описания и пояснения даны в подразделе "Первичные и альтернативные ключи" данного раздела.

```
VAR SP BASE
     RELATION { S#
     S#, P#  P#, QTY
     QTY }
KEY { S#, P# } ... ;
```

В данном примере показана переменная отношения с составным потенциальным ключом (т.е. ключом, состоящим из двух или большего количества атрибутов). *Простым потенциальным ключом* называется ключ, который не является составным.

```
VAR ELEMENT BASE RELATION { NAME    NAME,
                    SYMBOL  CHAR, ATOMIC#
                    INTEGER } KEY { NAME }
                    KEY { SYMBOL } KEY {
                    ATOMIC# } ;
```

[14] Еще одна существенная причина, по которой потенциальные ключи должны быть несократимыми, состоит в следующем: любой внешний ключ, который ссылается на "сократимый" потенциальный ключ (если бы это было возможно), также был бы "сократимым", и поэтому содержащая его переменная отношения почти наверняка нарушала бы принципы дальнейшей нормализации, как будет описано в главе 12.

■ В этом примере показана переменная отношения с несколькими различными потенциальными ключами, причем все они являются простыми.

```
VAR MARRIAGE BASE RELATION { HUSBAND                    NAME,
                             WIFE                       NAME,
                             DATE /* Дата регистрации брака */ DATE
                             }
/* Предполагается, что во всех рассматриваемых браках отсутствуют
такие */
/* ситуации, как многомужие, многоженство и повторная регистрация
брака */
/* одних и тех же супругов после их развода ...*/
                  KEY { HUSBAND, DATE }
                  KEY { DATE, WIFE } KEY
                  { WIFE, HUSBAND } ;
```

В данном примере показана переменная отношения с несколькими различными потенциальными ключами, причем все они являются составными. Обратите внимание также на то, что некоторые из этих ключей перекрываются.

Безусловно, как было указано в разделе 9.2, определение потенциального ключа по сути является просто сокращением для некоторого ограничения переменной отношения. Такое сокращение является полезным по многим причинам. Одна из них просто состоит в том, что потенциальные ключи имеют важное значение с точки зрения практики. В частности, в реляционной модели они предоставляют основной **механизм адресации на уровне кортежей.** Это означает, что единственный гарантируемый системой способ точного определения некоторого конкретного кортежа состоит в использовании определенного значения потенциального ключа. Например, гарантируется, что следующее выражение обеспечивает получение не больше одного кортежа (точнее, оно обеспечивает получение значения отношения, содержащего не больше одного кортежа).

```
S WHERE S# = S# ('S3')
```

В отличие от этого, приведенное ниже выражение в общем обеспечивает получение непредсказуемого количества кортежей (вернее, отношения, содержащего непредсказуемое количество кортежей).

```
S WHERE CITY = 'Paris',
```

Из этого следует, что *понятие потенциальных ключей является настолько же важным для успешной эксплуатации реляционных систем, как и понятие адресов оперативной памяти для успешной эксплуатации операционной системы самого компьютера.* Из этого следуют приведенные ниже выводы.

1. С позволения сказать, такие "переменные отношения", которые не имеют ни од ного потенциального ключа (т.е. "переменные отношения", которые допускают наличие дубликатов кортежей), неизбежно проявляют время от времени странную и аномальную реакцию на выполняемые с ними операции.
2. Система, в которой не реализовано понятие потенциальных ключей, неизбежно проявляет время от времени такую реакцию, которую нельзя назвать "полностью соответствующей реляционной модели", даже если представленные в ней пере менные отношения действительно являются в полном смысле слова настоящими переменными отношения и не содержат дубликатов кортежей.

Реакции на выполняемые операции, которые были названы выше "странными и аномальными" и "не полностью соответствующими реляционной модели", касаются таких

вопросов, как обновление представлений и оптимизация (эти темы рассматриваются, соответственно, в главах 10 и 18).

В завершение этого раздела приведем несколько замечаний.

- Потенциальные ключи должны иметь не только базовые переменные отношения! Это требование распространяется на все переменные отношения, включая, в частности, представления. Но что касается, например, представлений, вопрос о том, могут ли или должны быть объявлены такие ключи, отчасти зависит от того, предусмотрена ли в системе возможность использоватьссылки **на потенциальный ключ** [3.3].
- Надмножество потенциального ключа называется **суперключом** (например, суперключом для переменной отношения S является множество атрибутов {S#,CITY}). Суперключ обладает свойством уникальности, но не обязательно обладает свойством несократимости. Безусловно, потенциальный ключ — это частный случай суперключа.
- Если SK — суперключ для переменной отношения R, а A — атрибут R, то в R обязательно существует **функциональная зависимость** SK → A. В действительности, мы можем определить суперключ как подмножество SK атрибутов R, такое что функциональная зависимость SK → A имеет место для всех атрибутов A в R.

  Примечание. Важное понятие функциональной зависимости подробно рассматривается в главе 11.
- Наконец, следует отметить, что логическое понятие потенциального ключа не следует путать с физическим понятием *уникального индекса* (даже несмотря на то, что последний часто используется для реализации первого). Иными словами, нет никаких оснований требовать, чтобы на некотором потенциальном ключе был определен индекс (или, скажем, какой-либо иной специальный физический путь доступа). На практике, по-видимому, должен быть действительно предусмотрен некоторый путь доступа, но рассмотрение вопроса о том, существует ли он или нет, выходит за рамки реляционной модели как таковой.

Первичные и альтернативные ключи

Как уже было сказано выше, в любой переменной отношения возможно наличие двух или большего количества потенциальных ключей. В таком случае в реляционной модели традиционно предъявлялось такое требование (по меньшей мере, к базовым переменным отношения), чтобы точно один из этих ключей был выбран в качестве **первичного** ключа; с тех пор остальные ключи назывались **альтернативными.** Например, в случае определения таблицы элементов Менделеева, ELEMENT, МОЖНО выбрать {SYMBOL} в качестве первичного ключа, после чего применять {NAME} и {ATOM1C#} как альтернативные ключи. При наличии даже одного потенциального ключа в реляционной модели по традиции предъявлялось требование, чтобы этот потенциальный ключ рассматривался как первичный ключ для данной базовой переменной отношения. Поэтому считалось, что каждая базовая переменная отношения всегда имеет первичный ключ.

Во многих случаях (возможно, даже в большинстве случаев) действительно может оказаться, что определение одного из потенциальных ключей в качестве первичного (если между ними есть выбор) может стать целесообразным, но, безусловно, такое решение не является бесспорным во всех без исключения случаях. Аргументированные доводы в

поддержку такой позиции приведены в [9.14], а в этом разделе достаточно привести лишь один из них, который состоит в том, что выбор первичного ключа не диктуется логикой, а вместо этого, по сути, является произвольным. (Процитируем слова Кодда [9.9]: "Обычно основанием [для выбора первичного ключа] является упрощение работы, но этот аспект выходит за рамки реляционной модели".) В примерах, приведенных в этой книге, первичный ключ иногда указан, а иногда — нет. Но в них всегда определен по меньшей мере один потенциальный ключ.

Внешние ключи

Неформально выражаясь, *внешний ключ* представляет собой множество атрибутов некоторой переменной отношения R2, значения которых должны совпадать со значениями некоторого потенциального ключа некоторой переменной отношения R1. Например, рассмотрим множество атрибутов {S#} переменной отношения SP. Должно быть ясно, что заданное значение {S#} может присутствовать в переменной отношения SP, только если такое же значение присутствует и в переменной отношения S в качестве значения единственного потенциального ключа {S#} (поскольку поставка не может быть выполнена несуществующим поставщиком). Аналогичным образом, любое конкретное значение для множества атрибутов {P#} может присутствовать в переменной отношения SP, только если такое же значение присутствует в переменной отношения P в качестве значения единственного потенциального ключа {P#} (поскольку не может быть также выполнена поставка несуществующей детали). На основании этих примеров может быть сформулировано приведенное ниже определение[15].

- Допустим, что R2 — переменная отношения. В таком случае **внешним ключом** в R2 является множество атрибутов R2, скажем, FK, такое что выполняются следующие требования:
  - а) существует переменная отношения R1 (R1 и R2 не обязательно должны быть разными) с потенциальным ключом ск;
  - б) существует возможность переименования некоторого подмножества атрибутов FK, такое что FK преобразуется (скажем) в FK ■, а FK' и ск относятся к одному и тому же типу (кортежу);
  - в) в любое время каждое значение FK в текущем значении R2 приводит к получе нию значения для FK', которое идентично значению СК в некотором кортеже в текущем значении R1.

Из этого следуют приведенные далее выводы.

1. На практике необходимость действительно выполнять *переименование* возникает редко; это означает, что подмножество атрибутов FK, требующих переименования, обычно бывает пустым (пример такой ситуации, в которой это условие не соблю дается, показан в п. 7). Поэтому для упрощения в дальнейшем предполагается, что подмножества FK и FK' идентичны, если явно не указано иное.
2. Следует отметить, что каждое значение FK должно присутствовать в качестве зна чения СК, но обратное требование не предъявляется; это означает, что R1 может

[15] Обратите внимание на то, что это определение также базируется на понятии равенства кортежей.

содержать значение ск, которое в настоящее время не присутствует в R2 в качестве значения FK. Например, а случае поставщиков и деталей (примеры значений в этой базе данных приведены на рис. 3.8 на стр. 119) номер поставщика S5 присутствует в переменной отношения S, но не в переменной отношения SP, поскольку поставщик S5 в настоящее время не поставляет каких-либо деталей.

3. Внешний ключ FK является **простым** или **составным** в зависимости **от** того, являет ся ли простым или составным потенциальный ключ СК.

4. Любое значение FK представляет собой **ссылку** на кортеж, содержащий соответст вующее значение СК (он называется **кортежем, упомянутым в ссылке).** Ограничение, согласно которому значения FK должны соответствовать значениям СК, называет ся **ограничением ссылочной целостности.** Переменная отношения R2 называется **ссылающейся** переменной отношения, а переменная отношения R1 — переменной отношения, **указанной в** ссылке. Задача обеспечения того, чтобы база данных не включала каких-либо недопустимых значений внешнего ключа, называется зада чей поддержки **ссылочной целостности** (см. п. **12).**

5. Рассмотрим еще раз базу данных поставщиков и деталей. Ограничения ссылочной целостности в этой базе данных можно представить с помощью следующей **ссылочной диаграммы.**

S ← SP → P

Каждая стрелка означает, что в той переменной отношения, из которой исходит стрелка, имеется внешний ключ, ссылающийся на некоторый потенциальный ключ, заданный в переменной отношения, на которую указывает стрелка.

Примечание. Для наглядности иногда целесообразно обозначать каждую стрелку на ссылочной диаграмме именем (именами) атрибута (атрибутов), из которого состоит соответствующий внешний ключ[16], например, как показано ниже.

```
   S#      P#
S ←  SP →P
```

Но в данной книге подобные метки показаны только в тех случаях, если их отсутствие может привести к путанице или к неоднозначности.

6. Любая конкретная переменная отношения может быть одновременно и указанной в ссылке, и ссылающейся, как в случае переменной отношения R2, показанной ниже.

R3 → R2 → R1

Вообще говоря, допустим, что существуют переменные отношения Rn, R (n-1) , ..., R2, R1, такие что имеется ограничение ссылочной целостности, связывающее Rn с R(n-l), ограничение ссылочной целостности, связывающее R(n-l) с R(n-2) , ..., и ограничение ссылочной целостности, связывающее R2 с R1 следующим образом.

Rn→ R(n-l) → R(n-2) → ... → R2 → R1

[16] Иной способ (который, возможно, является предпочтительным) предусматривает присваивание имен внешним ключам, а затем использование этих имен для обозначения стрелок.

В таком случае цепочка стрелок, проходящая от Rn до R1, представляет собой ссылочный путь от Rn до R1.

7. Следует отметить, что переменные отношения R1 и R2, указанные в определении внешнего ключа, не обязательно должны быть разными. Это означает, что любая переменная отношения может иметь внешний ключ, значения которого должны совпадать со значениями некоторого потенциального ключа в той же переменной отношения. В качестве примера рассмотрим следующее определение переменной отношения (его синтаксис будет описан чуть позже, но в любом случае он должен быть достаточно очевидным).

```
VAR EMP BASE RELATION
          { EMP# EMP#, ..., MGR EMP# EMP#, ... }
  KEY { EMP# } FOREIGN KEY { RENAME MGR EMP# AS EMP# }
  REFERENCES EMP ;
```

Здесь атрибут MGR_EMP# обозначает табельный номер руководителя того служащего, который обозначен атрибутом EMP#; например, кортеж EMP для служащего Е4 может включать значение MGR_EMP# Е3, равное Е3, которое представляет собой ссылку на кортеж EMP для служащего Е3. (Как было обещано в п. 1, здесь показан пример, в котором требуется некоторое явное переименование атрибута.) Переменные отношения, подобные EMP, иногда называют *ссылающимися сами на себя* (или *самоссылающимися).*

Упражнение. Подготовьте данные, которые могли бы использоваться в качестве примера значения переменной отношения EMP.

8. Самоссылающиеся переменные отношения фактически представляют частный случай более общей ситуации, а именно, они показывают, что могут существовать ссылочные циклы. Переменные отношения Rn, R(n-l), R(n-2), ..., R2, Rl образуют такой цикл, если Rn содержит внешний ключ, ссылающийся на R(n-l), а R(n-l) содержит внешний ключ, ссылающийся на R(n-2) , ..., и т.д., нако нец, R1 содержит внешний ключ, снова ссылающийся на Rn. Кратко можно сформулировать это определение таким образом, что ссылочный цикл существует, если имеется ссылочный путь от некоторой переменной отношения Rn к ней са мой, как показано ниже.

$$Rn \rightarrow R(n\text{-}1) \rightarrow R(n\text{-}2) \rightarrow \ldots \rightarrow R2 \rightarrow R1 \rightarrow Rn$$

9. Иногда встречается такое небеспочвенное утверждение, что согласованность внешних и потенциальных ключей представляет собой своего рода "клей", кото рый скрепляет воедино всю базу данных. Еще один способ формулировки той же идеи состоит в том, что подобные согласования представляют собой определенные связи. Но необходимо учитывать важное замечание, что не все такие связи опреде лены лишь ключами, которые применяются указанным способом. Например, мо жет существовать некоторая связь между поставщиками и деталями ("нахождение в одном городе"), для представления которой применяются атрибуты CITY пере менных отношения S и P; некоторый поставщик и некоторая деталь считаются *на ходящимися в одном городе,* если данный поставщик находится в том же городе, где хранится данная деталь. Но эта связь не определена с помощью ключей.

10. По традиции, понятие внешнего ключа было сформулировано только для базовых переменных отношения, и этот факт сам по себе вызывает некоторые вопросы (см. обсуждение принципа взаимозаменяемости в разделе 10.2 главы 10). Сам ав тор не налагает здесь указанное ограничение, но для упрощения сводит обсужде ние только к базовым переменным отношения (причем эти указания во многом относятся непосредственно и к другим переменным отношения).
11. В определении реляционной модели первоначально предъявлялось требование, чтобы внешние ключи ссылались именно лишь на первичные ключи, а не просто на любые потенциальные ключи (например, еще раз рекомендуем ознакомиться с [9.9]). Автор в целом отвергает это ограничение как ненужное и вообще нежела тельное, хотя на практике оно может часто служить хорошей рекомендацией [9.14].
12. Наряду с понятием внешнего ключа в реляционной модели определено приведен ное ниже правило (правило ссылочной целостности).

    - **Ссылочная целостность.** База данных не должна содержать каких-либо несогла сованных значений внешнего ключа[17].

    В этом определении термин "несогласованное значение внешнего ключа" обозна чает значение внешнего ключа в некоторой ссылающейся переменной отношения, для которого не существует согласованного значения соответствующего потенци ального ключа в соответствующей переменной отношения, указанной в ссылке. Иными словами, это ограничение можно сформулировать просто как следующее требование: "Если значение в ссылается на A, то A должно существовать".

Ниже показан синтаксис определения внешнего ключа.

FOREIGN KEY { *<item commalist>* } REFERENCES *<relvar name>*

Эта конструкция присутствует в определении ссылающейся переменной отношения; здесь *<relvar name>* обозначает переменную отношения, указанную в ссылке, а каж дый элемент <it*em*> представляет собой либо имя атрибута *<attribute name>* в ссы лающейся переменной отношения, либо выражение в следующей форме.

RENAME *<attribute name>* AS <att*ribute name*>

(Пример применения варианта с использованием ключевого слова RENAME приведен в определении самоссылающейся переменной отношения EMP в п. 7.) Примеры объявлений внешнего ключа были приведены выше во многих разделах данной книги (в частности, см. рис. 3.9 в главе 3). Безусловно, как было указано в разделе 9.2, определение внешнего ключа фактически представляет собой просто сокращение для некоторого ограничения базы данных (или для некоторого ограничения переменной отношения, в

---

[17] Обратите внимание на то, что это правило ссылочной целостности может рассматриваться как "метаограничение" (сверхограничение), поскольку в нем подразумевается, что любая отдельно взятая база данных должна быть объектом действия некоторых конкретных ограничений, свойственных рассматриваемой базе данных и совместно гарантирующих, чтобы в этой определенной базе данных не нарушалась ссылочная целостность. Кстати, следует отметить, что реляционная модель обычно рассматривается как включающая еще одно подобное "метаограничение" — правило целостности сущностей. Отложим обсуждение этого правила до главы 19.

случае самоссылающейся переменной отношения), за исключением той ситуации, когда определение внешнего ключа распространяется на некоторые "ссылочные действия"; тогда оно представляет собой нечто большее, чем просто ограничение ссылочной целостности как таковое. Эта тема рассматривается в следующем подразделе, "Ссылочные действия".

Ссылочные действия

Рассмотрим следующий оператор на языке Tutorial D.

```
DELETE S WHERE S# = S# ('S1') ;
```

Предположим, что операция DELETE выполняет именно то, что в ней указано, т.е. удаляет кортеж поставщика si, не больше и не меньше. Предположим также, что соблюдаются еще два условия: во-первых, база данных включает сведения о поставках, выполненных поставщиком S1, во-вторых, приложение не удаляет эти сведения о поставках. Но после того как система проверяет ограничение ссылочной целостности, которое связывает данные о поставке с данными о поставщиках, она обнаруживает нарушение и активизирует исключение.

Однако возможен альтернативный подход, который может оказаться предпочтительным в некоторых ситуациях. Он заключается в том, что система выполняет соответствующее компенсирующее действие, которое позволяет гарантировать, что общий результат всегда будет удовлетворять ограничению. В данном примере очевидно, что компенсирующее действие со стороны системы состоит в "автоматическом" удалении системой данных о поставках, выполненных поставщиком S1. Такого эффекта можно достичь, дополнив определение внешнего ключа, как показано ниже.

```
VAR SP BASE RELATION { ... } ...
    FOREIGN KEY { S# } REFERENCES S
                      ON DELETE CASCADE ;
```

Спецификация ON DELETE CASCADE определяет правило удаления для данного конкретного внешнего ключа, а спецификация CASCADE является *ссылочным действием* для этого правила удаления. Смысл данных спецификаций состоит в том, что выполнение операции DELETE на переменной отношения поставщиков "каскадно" приводит к удалению также соответствующих кортежей (если они имеются) в переменной отношения поставок.

Еще одним широко применяемым вариантом ссылочного действия является RESTRICT (оно не имеет ничего общего с операцией сокращения реляционной алгебры). В данном случае использование ключевого слова RESTRICT означает, что операции DELETE должны "ограничиваться" теми ситуациями, в которых отсутствуют согласованные поставки (в противном случае они будут отвергнуты). Если же в определении конкретного внешнего ключа ссылочное действие не указано, это эквивалентно применению ключевого слова NO ACTION, которое означает именно то, что в нем сказано: операция DELETE выполняется точно так же, как в ней указано, не больше и не меньше. (Если в рассматриваемой ситуации задано ключевое слово NO ACTION, а с удаляемыми данными о поставщике связаны согласованные данные о поставках, то в дальнейшем возникает нарушение ограничения ссылочной целостности, поэтому конечный результат

становится аналогичным такому, как при использовании ключевого слова RESTRICT.) Из этого следуют приведенные ниже выводы.

1. Операция DELETE не является единственной операцией, для которой имеет смысл определять ссылочные действия. Например, что произойдет при попытке обновить номер такого поставщика, для которого существует по меньшей мере одна согласованная поставка? Очевидно, что требуется не только правило удаления, но и правило обновления. В общем существуют такие же варианты ссылочных действий для операции UPDATE, как и для операции DELETE, которые описаны ниже.

   - CASCADE. Действие операции UPDATE распространяется каскадно для обновления внешнего ключа также и в этих согласованных поставках.
   - RESTRICT. Действие операции UPDATE ограничивается тем случаем, когда отсутствуют такие согласованные поставки (в противном случае эта операция отвергается).
   - NO ACTION. Операция UPDATE выполняется в точном соответствии с указанием (но в дальнейшем может произойти нарушение ссылочной целостности).

2. Безусловно, CASCADE, RESTRICT и NO ACTION не являются единственными возможными ссылочными действиями; они просто относятся к таким типам, которые часто требуются на практике. А в принципе может существовать произвольное количество допустимых ответов (например) на попытку удалить данные о некотором поставщике. Ниже приведено несколько примеров.

   - Такая попытка может быть по определенной причине сразу же отвергнута.
   - Информация может быть записана в некоторую архивную базу данных.
   - Поставки, относящиеся к рассматриваемому поставщику, могут быть переданы некоторому другому поставщику.

   Практически не осуществима задача предусмотреть декларативный синтаксис для всех ответов, которые можно себе представить. Поэтому в общем должна быть предусмотрена возможность определять ссылочное действие, состоящее из произвольной процедуры, определяемой пользователем (см. следующий раздел). Более того, выполнение этой процедуры должно рассматриваться как часть выполнения оператора, вызвавшего соответствующую проверку целостности; эта проверка целостности должна осуществляться повторно, после выполнения указанной процедуры (поскольку очевидно, что эта процедура не должна оставлять базу данных в таком состоянии, при котором в ней нарушено указанное ограничение).

3. Предположим, что R2 и R1, соответственно, — ссылающаяся переменная отношения и связанная с ней переменная отношения, указанная в ссылке, как показано ниже.

   R2 → R1

   Допустим, что в применяемом правиле удаления задано действие CASCADE. В таком случае выполнение операции DELETE над указанным кортежем R1 влечет за собой (в общем) выполнение операции DELETE над некоторыми кортежами переменной отношения R2. Если же теперь предположить, что на переменную отношения R2, в

свою очередь, ссылается некоторая другая переменная отношения R3, как показано ниже, то, по сути, выполнение подразумеваемой операции DELETE на кортежах R2 равносильно попытке непосредственного удаления этих кортежей.

R3 → R2 → R1

Это означает, что результаты выполнения последней операции зависят от того, какое правило удаления определено для ограничения ссылочной целостности, направленного от R3 к R2. Если эта подразумеваемая операция DELETE завершается неудачей (согласно правилу удаления, которое связывает R3 с R2, или по какой-либо иной причине), то окончится неудачей вся операция и база данных остается неизменной. Подобная процедура распространяется рекурсивно на любое количество уровней.

Аналогичные замечания относятся также к правилу каскадного обновления, с учетом соответствующих поправок, если внешний ключ переменной отношения R2 имеет какие-либо общие атрибуты с потенциальным ключом той переменной отношения, на которую ссылается внешний ключ переменной отношения R3.

4. Из сказанного выше следует, что с логической точки зрения операции обновления базы данных всегда являются атомарными, или неразрывными (выполняются по принципу "все или ничего"), даже если они фактически связаны с осуществлением нескольких операций обновления нескольких переменных отношения, например, в связи с тем, что участвуют в каскадном ссылочном действии.

9.11. ТРИГГЕРЫ (НЕБОЛЬШОЕ ОТСТУПЛЕНИЕ)

Из всего сказанного выше в данной главе должно быть очевидно, что для нас особый интерес представляет декларативная поддержка целостности. И хотя ситуация за последние годы улучшилась, остается фактом, что лишь немногие продукты (если они вообще есть) обеспечивают такую поддержку со времени своего первоначального появления на рынке. Вследствие этого ограничения целостности чаще всего реализуются процедурное использованием триггерных процедур. Последние представляют собой заранее откомпилированные процедуры, которые хранятся вместе с базой данных (возможно, в самой базе данных) и вызываются автоматически при возникновении некоторых указанных событий. В частности, пример 1 ("значения статуса должны находиться в пределах от 1 до 100 включительно") может быть реализован с помощью триггерной процедуры, которая вызывается каждый раз при вставке кортежа в переменную отношения S, проверяет этот вновь вставляемый кортеж и снова его удаляет, если значение статуса не входит в указанные пределы. В этом разделе приведено краткое описание триггерных процедур в связи с тем, что они имеют значительную практическую важность. Но необходимо сразу же привести следующие замечания.

1. Именно потому, что они являются процедурами, триггерные процедуры нельзя считать рекомендуемым способом реализации ограничений целостности. Пользователям сложнее понять, как действуют процедуры, а для системы процедуры создают дополнительные трудности при оптимизации. Следует также отметить, что декларативные ограничения проверяются при всех соответствующих обнов-

лениях[18], а триггерные процедуры выполняются только при возникновении указанного события (допустим, при вставке кортежа в переменную отношения S).

2. Область применения триггерных процедур не ограничивается задачами поддержки целостности, которые являются темой настоящей главы. Вместо этого, если учесть замечания, сделанные в п. 1, фактически они могут выполнять другие полезные задачи и именно поэтому имеют право на существование. Некоторые примеры таких "других полезных задач" приведены ниже.

   а) Передача пользователю предупреждения о том, что возникло некоторое ис ключение (например, выдача предупреждающего сообщения, если наличное количество некоторых деталей на складе становится ниже критического уровня).

   б) Отладка (т.е. отслеживание ссылок на указанные переменные и/или контроль над изменениями состояния этих переменных).

   в) Аудит (например, регистрация информации о том, кто и когда внес те или иные изменения в определенные переменные отношения).

   г) Измерение производительности (например, регистрация времени наступления или трассировка указанных событий в базе данных).

   д) Проведение компенсирующих действий (например, каскадная организация удаления кортежа поставщика для удаления также соответствующих кортежей поставок)[19].

   Поэтому данный раздел, как и указано в его названии, носит характер отступления от основной темы.

Рассмотрим следующий пример. (Этот пример основан на языке SQL, а не на языке Tutorial D, поскольку в [3.3] не предписана, и не могла быть предписана какая-либо поддержка триггерных процедур; фактически он основан на коммерческом программном продукте, а не на стандарте SQL, поскольку стандарт SQL не поддерживает конкретное средство, показанное в данном примере.) Предположим, что в базе данных имеется представление LONDON_SUPPLIER, которое определено следующим образом.

```
CREATE VIEW LONDON SUPPLIER
    AS SELECT S#, SNAME,
       STATUS FROM   S WHERE
       CITY = 'London' ;
```

При обычных обстоятельствах, если пользователь попытается вставить строку в это представление, то среда поддержки языка SQL действительно вставит строку в соответствующую базовую таблицу S с таким значением CITY, которое задано по умолчанию для

[в] Обратите внимание на то, что в спецификациях декларативных ограничений не предусмотрены явные указания для СУБД, когда должны выполняться проверки целостности. А это и не требуется, во-первых, потому, что такие явные указания вынуждали бы пользователя, объявляющего ограничения, вы полнять лишнюю работу, во-вторых, потому, что пользователь мог бы их задать неправильно. Вместо этого желательно, чтобы сама система решала, когда должны проводиться эти проверки (см. аннотацию к [9.5]).

[д] И действительно, каскадное удаление — это типичный пример триггерной процедуры. Но заслу живает внимания то, что такое удаление задано декларативно! Автор отнюдь не утверждает, что ссылоч ные действия — это неудачная идея только потому, что они реализуются "триггерно" (т.е. как реакция на некоторое событие).

столбца CITY (см. главу 10). При условии, что по умолчанию задан город, отличный от Лондона общий эффект этого действия окажется таким, что новая строка не появится в этом представлении! Поэтому создадим триггерную процедуру следующим образом.

```
CREATE TRIGGER LONDON_SUPPLIER_INSERT
       INSTEAD OF INSERT ON LONDON_SUPPLIER
       REFERENCING NEW ROW AS R
       FOR EACH ROW
        INSERT INTO S ( S#, SNAME, STATUS, CITY )
              VALUES ( R.S#, R.SNAME, R.STATUS, 'London' ) ;
```

Теперь вставка строки в это представление повлечет за собой то, что строка будет вставлена в соответствующую базовую таблицу со значением CITY, равным London, а не со значением, применяемом по умолчанию (и новая строка теперь будет появляться в представлении, в полном соответствии с поставленной задачей).

Из этого примера следуют определенные выводы, приведенные ниже. *Примечание.* Эти выводы не относятся только к языку SQL, несмотря на тот факт, что приведенный пример основан на SQL (конкретные сведения о средствах SQL даны в следующем разделе).

1. В общем, в операторе создания триггера CREATE TRIGGER, кроме всего прочего, определены событие, условие и действие следующим образом:

   - **событием** является операция в базе данных (в этом примеое "INSERT ON LONDON_SUPPLIER");
   - **условие—** это логическое выражение, которое должно принимать значение TRUE для того, чтобы было выполнено действие (если условие не указано явно, как в данном примере, то оно по умолчанию равно просто TRUE);
   - **действие** — это и есть сама триггерная процедура (в данном примере "INSERT INTO S ...").

   Событие и условие вместе иногда называют *триггерным событием,* а сочетание всех трех компонентов (событие, условие и действие) обычно называют просто **триггером.** По очевидным причинам триггеры именуются также правилами "событие-условие-действие" (Event-Condition-Action — ECA) или сокращенно правилами ECA.

2. К возможным событиям относится выполнение операций INSERT, DELETE, UPDATE (возможно, над определенными атрибутами), достижение конца транзак ций (COMMIT), наступление указанного времени суток, истечение определенного интервала времени, нарушение указанного ограничения и т.д.
3. В общем, действие может выполняться до (BEFORE), после (AFTER) или вместо (INSTEAD OF) действия, обусловленного указанным событием, при условии, что эти варианты имеют смысл.
4. В общем, действие может выполняться для каждой строки (FOR EACH ROW) или для каждого оператора (FOR EACH STATEMENT), при условии, что эти варианты имеют смысл.

5. В общем, при выполнении действия, определяемого триггером, должен быть предусмотрен способ, позволяющий ссылаться на данные в том виде, какой они имеют до и после возникновения указанного события, при условии, что применение такого средства действительно имеет смысл.
6. Базу данных, которая имеет связанные с ней триггеры, иногда называют *активной базой данных.*

В завершение этого раздела следует отметить, что триггеры, безусловно, имеют важную область применения, но ими следует пользоваться с осторожностью и, вероятно, не прибегать к ним вообще, если есть альтернативный способ решения насущной задачи. Ниже перечислены некоторые причины, по которым применение триггеров на практике может вызвать появление определенных проблем.

- Если одно и то же событие вызывает *запуск* (как принято называть это событие на профессиональном жаргоне) нескольких разных триггеров, то последовательность их активизации может оказаться, с одной стороны, важной, а с другой — неопределенной.
- Могут возникать цепочки запуска триггеров, при которой запуск триггера T1 вызывает активизацию триггера T2, который вызывает активизацию триггера T3 и т.д.
- Запуск триггера т может даже снова вызвать рекурсивный запуск самого этого триггера.
- В результате наличия триггеров даже "простые" операции INSERT, DELETE или UPDATE могут приводить к такому эффекту, который принципиально отличается от ожидаемого пользователем (особенно если задано ключевое слово INSTEAD OF, как в приведенном выше примере).

Если учесть все описанные выше замечания, то должно быть очевидно, что суммарные результаты действия некоторой определенной совокупности триггеров могут оказаться весьма сложными для понимания. Декларативные решения, если они возможны, всегда являются более предпочтительными по сравнению с процедурными.

9.12. СРЕДСТВА SQL

Начнем с описания поддержки в языке SQL (или, скорее, по большей части, с констатации отсутствия такой поддержки) схемы классификации ограничений, описанной в разделе 9.9.

- В языке SQL вообще не поддерживаются *ограничения типа,* за исключением тех примитивных ограничений, которые являются прямым следствием применения определенного физического представления. Например, как было показано в главе 5, допустимо утверждать, что значения типа WEIGHT должны быть представлены в виде чисел DECIMAL (5,1), но нельзя указать, что эти числа должны быть больше нуля и меньше 5000.
- В языке SQL (безусловно) поддерживаются *ограничения атрибута.*
- В языке SQL не поддерживаются *ограничения переменной отношения* как таковые. В нем обеспечивается поддержка ограничений базовой таблицы, но такие ограничения распространяются именно только на базовые таблицы, а не на все таблицы

в целом (в частности, они не охватывают представления), и они не сводятся к упоминанию просто одной такой таблицы, но фактически могут иметь произвольную СЛОЖНОСТЬ.

- В языке SQL не поддерживаются *ограничения базы данных* как таковые. В нем поддерживаются общие ограничения, которые в спецификации этого языка называются *утверждениями* (assertion), но не обязательно нужно использовать лишь такие ограничения, если требуется указать в ограничении больше одной таблицы. (В действительности, ограничения базовой таблицы и общие ограничения языка SQL являются логически взаимозаменяемыми, за исключением той странности, которая отмечена в самом конце следующего подраздела, "Ограничения базовой таблицы".)

В языке SQL отсутствует также непосредственная поддержка ограничений перехода, но такие ограничения могут быть реализованы процедурно с помощью триггеров. Кроме того, в этом языке не представлено явно понятие предиката переменной отношения (или таблицы), а эта особенность языка SQL является очень важной, как показано в следующей главе.

Ограничения базовой таблицы

Ограничения базовой таблицы в языке SQL задаются в операторе CREATE TABLE или ALTER TABLE. Каждое такое ограничение представляет собой ограничение потенциального ключа, ограничение внешнего ключа или ограничение проверки. Рассмотрим каждое из них по очереди.

Примечание. Перед определением любого из этих ограничений может находиться необязательная спецификация CONSTRAINT *<constraint name>,* которая позволяет присвоить ограничению имя. Автор не рассматривает эту опцию для сокращения изложения (но необходимо отметить, что на практике, по-видимому, следует присваивать имена всем ограничениям). Здесь также не рассматриваются некоторые сокращения, например, возможность задавать потенциальный ключ (как "встроенный" в составе определения столбца), по той же причине.

Потенциальные ключи

Любое определение потенциального ключа SQL принимает одну из следующих двух форм.

```
PRIMARY KEY ( <column name commalist>
) UNIQUE ( <column name commalist> )
```

В обоих случаях выражение *<column name commalist>* с разделенным запятыми списком имен столбцов не должно быть пустым[20]. Любая конкретная базовая таблица может иметь не больше одной спецификации с определением первичного ключа PRIMARY KEY, но любое количество спецификаций потенциального ключа UNIQUE. В случае PRIMARY KEY каждый указанный столбец дополнительно рассматривается как соответствующий требованию NOT NULL, т.е. не содержащий пустых значений, даже если ключевое слово NOT NULL не задано явно (см. приведенное ниже описание ограничений проверки).

[20] См. упражнение 9.10.

Внешние ключи

Определение внешнего ключа языка **SQL** принимает следующую форму.

```
FOREIGN KEY ( <column name commalist> )
  REFERENCES <base table name> [ ( <column name
commalist> ) ] [ ON DELETE <referential action> ] [ ON
UPDATE <referential action> ]
```

Здесь спецификация ссылочного действия <*referential action*> может принимать значение NO ACTION (по умолчанию), RESTRICT, CASCADE, SET DEFAULT или SET NULL[21]. Отложим обсуждение конструкций SET DEFAULT и SET NULL до главы 19; другие опции описаны в разделе 9.10. Вторая спецификация <*column name commalist*> требуется, если внешний ключ ссылается на потенциальный ключ, который не является первичным ключом.

Примечание. Согласование внешнего и потенциального ключей осуществляется не на основе имен столбцов, а с учетом позиции столбца (слева направо) в разделенном запятыми списке.

Ограничения проверки

Любое ограничение проверки в языке **SQL** принимает следующую форму.

CHECK (<*bool exp*>)

Допустим, что CC — ограничение проверки для базовой таблицы т. В таком случае считается, что в таблице т ограничение CC нарушается тогда и только тогда, когда эта таблица в настоящее время содержит по меньшей мере одну строку (см. последний абзац данного подраздела) и проверка текущего значения т влечет за собой то, что логическое выражение <*bool exp*> для CC принимает значение FALSE.

Примечание. В общем, следует отметить, что выражения <*bool* exp> языка SQL могут быть сколь угодно сложными; даже в данном рассматриваемом контексте они явно не ограничиваются ссылками только на базовую таблицу т, но могут вместо этого ссылаться на любую доступную часть базы данных.

Ниже приведен пример оператора CREATE TABLE, в котором иллюстрируется **применение** ограничений базовой таблицы всех трех видов.

```
CREATE TABLE SP
     ( S# S# NOT NULL, P# P# NOT NULL, QTY QTY NOT
       NULL,
       PRIMARY KEY ( S#, P# ),
       FOREIGN KEY ( S# ) REFERENCES S
                        ON DELETE CASCADE
                         ON UPDATE
       CASCADE, FOREIGN KEY ( P# )
       REFERENCES P
                        ON DELETE CASCADE
                         ON UPDATE CASCADE, CHECK ( QTY
       > QTY ( 0 ) AND QTY < QTY { 5000 ) ) ) ;
```

[21] Кстати, следует отметить, что для поддержки некоторых действий типа <ref erential action> (в частности CASCADE) требуется, чтобы система (хотя бы неявно) поддерживала некоторые виды операций множественного реляционного присваивания! Причем это требование существует несмотря на то, что подобные операции не поддерживаются в языке SQL как таковые.

Здесь предполагается, что атрибуты s# и P# были явно определены как предназначенные для использования, соответственно, в качестве первичных ключей для таблиц S и P. Кроме того, в этом операторе используется сокращение, согласно которому ограничение проверки в указанной ниже форме может быть заменено простой спецификацией NOT NULL в определении рассматриваемого столбца <*column name*>.

```
CHECK ( <column name> IS NOT NULL )
```

Поэтому в данном примере три ограничения проверки, которые могли оказаться довольно сложными, заменены тремя спецификациями NOT NULL.

В заключение этого подраздела еще раз повторим замечание, что ограничение базовой таблицы SQL всегда считается удовлетворенным, если рассматриваемая базовая таблица оказалась пустой, даже если это ограничение имеет форму (скажем) "1 = 2" (или даже, если на то пошло, оно по сути имеет такую форму, что "эта таблица не должна быть пустой"!).

Утверждения

Теперь перейдем к описанию общих ограничений SQL, или утверждений. Такие ограничения определяются с помощью оператора CREATE ASSERTION, который имеет показанный ниже синтаксис.

```
CREATE ASSERTION <constraint
       name> CHECK ( <bool exp> )
       ;
```

А ниже показан синтаксис оператора DROP ASSERTION.

```
DROP ASSERTION <constraint name> ;
```

Обратите внимание на то, что в отличие от большинства других форм оператора DROP языка SQL (например, DROP TYPE, DROP TABLE, DROP VIEW), в операторе DROP ASSERTION не предусмотрены две противоположные опции RESTRICT и CASCADE.

Ниже приведено шесть примеров из раздела 9.1, выраженных в форме утверждений языка SQL. Рекомендуем читателю в качестве упражнения попытаться вместо этого сформулировать данные примеры в виде ограничений базовой таблицы.

1. *Значение статуса каждого поставщика должно находиться в пределах от 1 до 100 включительно.*

```
CREATE ASSERTION SC1 CHECK
     ( NOT EXISTS ( SELECT * FROM S
                    WHERE  S.STATUS < 0    OR
                      S.STATUS > 100 ) ) ;
```

2. *Каждый поставщик из Лондона имеет статус 20.*

```
CREATE ASSERTION SC2 CHECK
     ( NOT EXISTS ( SELECT * FROM S
                    WHERE  S.CITY = 'London'
                    AND    S.STATUS ≠ 20 ) )
                    ;
```

3. *Если вообще имеются какие-либо детали, то по меньшей мере одна из них должна быть синего цвета.*

```
CREATE ASSERTION PC3 CHECK      ( NOT
     EXISTS ( SELECT * FROM P )
       OR  EXISTS ( SELECT * FROM P
                    WHERE  P.COLOR = COLOR ('Blue') ) ) ;
```

4. Разные поставщики не могут иметь одинаковые номера поставщиков.

```
CREATE ASSERTION SC4 CHECK
     ( UNIQUE ( SELECT S.S# FROM S ) ) ;
```

В этом операторе UNIQUE представляет собой операцию SQL, которая принимает в качестве фактического параметра таблицу и возвращает значение TRUE, если эта таблица не содержит дубликатов строк, а в противном случае возвращает значение FALSE.

5. *Каждая поставка выполняется существующим поставщиком.*

```
CREATE ASSERTION SSP5 CHECK
     . ( NOT EXISTS
         ( SELECT * FROM SP
           WHERE  NOT EXISTS
                ( SELECT * FROM S
                  WHERE  S.S# = SP.S# ) ) ) ;
```

6. *Ни один поставщик со статусом меньше 20 не поставляет любые детали в количестве больше 500.*

```
CREATE ASSERTION SSP6 CHECK
     ( NOT EXISTS ( SELECT * FROM S, SP
                    WHERE  S.STATUS < 20  AND    S.S#
                    = SP.S# AND    SP.QTY > QTY ( 500
                    ) ) ) ;.
```

Кратко рассмотрим еще один пример. Для этого обратимся к приведенному ниже определению представления из предыдущего раздела.

```
CREATE VIEW LONDON SUPPLIER
    AS SELECT S#, SNAME,
       STATUS FROM   S WHERE
       CITY = 'London' ;
```

Как уже было сказано, в это определение представления нельзя включать спецификацию в такой форме.

```
UNIQUE  (  S#  ) '
```

Но, как ни странно, в этом определении можно задать общее ограничение в следующей форме.

```
CREATE ASSERTION LSK CHECK
     ( UNIQUE ( SELECT S# FROM LONDON_SUPPLIER ) ) ;
```

Отложенная проверка

Ограничения SQL отличаются также от ограничений, описанных в предыдущих разделах данной главы, в той части, которая касается выполнения проверки. В рассматриваемой выше схеме все ограничения проверяются немедленно. В отличие от этого, в

языке SQL ограничения[22] могут быть определены как допускающие отложенную проверку (DEFERRABLE) или не допускающие такую проверку (NOT DEFERRABLE); если заданное ограничение объявлено как DEFERRABLE, оно может быть дополнительно определено как отложенное первоначально (INITIALLY DEFERRED) или немедленно выполняемое с самого начала (INITIALLY IMMEDIATE); эти ключевые слова определяют состояние ограничения в начале каждой транзакции. Ограничения NOT DEFERRABLE всегда проверяются немедленно, а проверку ограничений DEFERRABLE можно динамически включать и выключать с помощью следующего оператора.

SET CONSTRAINTS <*constraint name commalist*> <*option*> ;

Здесь опция <*option*> принимает значение IMMEDIATE или DEFERRED. Пример применения такой опции приведен ниже.

```
SET CONSTRAINTS SSP5, SSP6 DEFERRED ;
```

Ограничения DEFERRABLE проверяются, только если они находятся в состоянии IMMEDIATE. Перевод ограничения DEFERRABLE в состояние IMMEDIATE вызывает немедленную проверку этого ограничения; если данная проверка оканчивается неудачей, то и выполнение оператора SET IMMEDIATE оканчивается неудачей. Выполнение оператора COMMIT влечет за собой вызов операторов SET IMMEDIATE для всех ограничений DEFERRABLE; если после этого хотя бы одна из проверок целостности оканчивается неудачей, происходит откат транзакции.

Триггеры

Оператор CREATE TRIGGER языка SQL выглядит следующим образом.

```
CREATE TRIGGER < trigger name>
       <before or after> <event> ON <base table
     name> [ REFERENCING <naming commalist> } [
     FOR EACH <row or statement> ] [ WHEN (
     <bool exp> ) ] <action> ;
```

Пояснения к этому оператору приведены ниже.

1. Спецификация с указанием времени проверки до или после активизации триггера, <*before or after*>, принимает значение BEFORE или AFTER (в стандарте SQL не поддерживается ключевое слово INSTEAD OF, но в некоторых программных продуктах такая поддержка предусмотрена).
2. Событие <*event*> может принимать значение INSERT, DELETE или UPDATE. Значение UPDATE может дополнительно уточняться с помощью спецификации OF <*column name commalist*>.
3. Каждое определение именования <*naming*> может принимать одну из следующих форм.

---

[22] Но некоторые ограничения должны быть обязательно указаны как относящиеся к типу NOT DEFERRABLE. Например, если FK — внешний ключ, то ограничение потенциального ключа для соответствующего потенциального ключа должно быть задано как NOT DEFERRABLE.

```
OLD     ROW     AS
<name>  NEW   ROW
AS    <name>    OLD
TABLE   AS   <name>
NEW    TABLE    AS
<name>
```

4. Спецификация с определением строки или оператора <*row or statement*> принимает значение ROW или STATEMENT (STATEMENT применяется по умолча нию). Ключевое слово ROW означает, что триггер активизируется для каждой от дельной строки, на которую распространяется действие триггерного оператора, а STATEMENT означает, что триггер активизируется только один раз для данного оператора, рассматриваемого как единое целое.
5. Если определена конструкция WHEN, она означает, что действие <*action*> должно выполняться, только если логическое выражение <*bool exp*> принимает значе ние TRUE.
6. Спецификация <*action*> задает отдельный оператор SQL (но этот отдельный оператор может быть достаточно сложным, т.е. составным, а это неформально оз начает, что такой оператор может состоять из последовательности операторов, обозначенных разграничителями BEGIN и END).

   Наконец, ниже показан синтаксис оператора DROP TRIGGER.

```
DROP TRIGGER <trigger name> ;
```

Как и в операторе DROP ASSERTION, в операторе DROP TRIGGER не предусмотрено использование пары противоположных опций RESTRICT И CASCADE.

9.13. РЕЗЮМЕ

В настоящей главе рассматривается важное понятие целостности. Задача обеспечения целостности представляет собой задачу обеспечения правильности данных в базе данных (по меньшей мере, обеспечения правильности в максимально возможной степени; к сожалению, лучшее, чего мы можем достичь, состоит в обеспечении совместимости данных с установленными ограничениями). Безусловно, для нас наибольший интерес представляют **декларативные** решения этой задачи.

В начале этой главы было показано, что ограничения целостности приведенную ниже общую форму.

Если (IF) некоторые кортежи присутствуют в некоторых переменных отношения, то (THEN) эти кортежи удовлетворяют некоторому условию.

(Ограничения типа характеризуются определенными отличиями, как показано в приведенном ниже описании.) В этой главе рассматривается синтаксис определения таких ограничений, основанный на той версии языка Tutorial D, в которой применяется исчисление предикатов, и указано, что в этом синтаксисе не предусмотрено каких-либо способов, с помощью которых пользователь мог бы сообщить СУБД, когда должна быть выполнена проверка таких ограничений, поскольку желательно, чтобы время этой проверки определяла сама СУБД. Кроме того, в данной главе утверждается (но пока еще без обоснования такой позиции), что вся проверка ограничений должна выполняться **немедленно.**

Затем было показано, что любое ограничение в том виде, в каком оно сформулировано, представляет собой **предикат,** а при его проверке (т.е. при подстановке текущих значений отношений вместо переменных отношения, указанных в этом предикате), оно становится **высказыванием. Предикатом переменной отношения** для определенной переменной отношения является логическое выражение, состоящее из всех предикатов, которые применяются к данной переменной отношения, соединенных операторами "И", а **предикатом базы данных** для определенной базы данных является логическое выражение, состоящее из всех предикатов, которые применяются к этой базе данных, соединенных операторами "И". Кроме того, в этой главе сформулировано **золотое правило,** приведенное ниже.

Ни одна операция обновления не должна приводить к присваиванию любой базе данных такого значения, которое вызывает то, что предикат этой базы данных получает значение FALSE.

Затем в этой главе доказано различие между **внутренними** и **внешними** предикатами. Внутренние предикаты заданы формально. Об этих предикатах имеются сведения в системе, а их проверка выполняется СУБД (предикаты переменной отношения и предикаты базы данных, о которых шла речь в предыдущем абзаце, являются внутренними предикатами). В отличие от внутренних предикатов, внешние предикаты задаются только неформально. Сведения о них известны только пользователю, но не системе. **Предположение о замкнутости мира** относится к внешним предикатам, а не ко внутренним. и Кстати, как уже может быть известно читателю, то, что обычно называется *обеспечением целостности,* в контексте базы данных фактически определяет **семантику,** поскольку **смысл** данных определяется именно ограничениями целостности (в частности предикатами переменной отношения и базы данных). И в этом состоит одна из причин, по которой обеспечение целостности является такой исключительно важной задачей, как было указано во введении к этой главе.

Кроме того, было указано (и это вполне укладывается в рамки здравого смысла), что требования поддержки целостности распространяются на все переменные отношения (в частности, они распространяются на представления), несмотря на то, что, безусловно, ограничения, применяемые к конкретному представлению, могут быть выведены из тех ограничений, которые применяются к переменным отношения, послужившим основой для создания рассматриваемого представления.

В дальнейшем изложении было показано, что ограничения целостности подразделяются на четыре описанные ниже категории.

- Ограничения **типа** определяют допустимые значения для конкретного типа (или домена) и проверяются во время вызова соответствующего селектора.
- Ограничения **атрибута** задают допустимые значения для конкретного атрибута, и если предусмотрена проверка ограничений типа, то вероятность нарушения ограничений атрибута исключена.
- Ограничения **переменной отношения** задают допустимые значения для конкретной переменной отношения и проверяются при обновлении рассматриваемой переменной отношения.
- Ограничения базы **данных** задают допустимые значения для конкретной базы данных и проверяются при обновлении рассматриваемой базы данных.

Но было указано, что различия между ограничениями переменной отношения и базы данных в большей степени относятся к сфере практики, чем логики. Кроме того, были кратко описаны ограничения **перехода.**

Затем были описаны такие важные с точки зрения практики частные случаи ограничений, как **потенциальные, первичные, альтернативные** и **внешние** ключи. Потенциальные ключи обладают свойствами **уникальности** и **несократимости,** причем каждая переменная отношения (без каких-либо исключений!) должна иметь по меньшей мере один такой ключ. Ограничения, согласно которым значения определенного внешнего ключа должны совпадать со значениями соответствующего потенциального ключа, называются **ограничениями ссылочной целостности;** в этой главе описано несколько областей применения идеи ссылочной целостности, включая, в частности, понятие **ссылочных действий** (причем наиболее важной из этих областей применения является каскадное распространение действий с помощью ключевого слова CASCADE). По материалам обсуждения этой последней темы было сделано краткое отступление, касающееся темы **триггеров.**

В завершение данной главы были описаны соответствующие средства языка SQL. Ограничения типа в языке SQL развиты очень слабо; по сути, они сводятся к определению того, что рассматриваемый тип должен иметь определенное физическое представление. Ограничения базовой таблицы SQL (которые обеспечивают в качестве частного случая поддержку для ключей) и общие ограничения ("утверждения"), представляют собой аналоги ограничений переменной отношения и базы данных (кроме ограничений перехода), но их классификация определена гораздо менее четко по сравнению с языком Tutorial D (фактически они являются почти взаимозаменяемыми и не совсем понятно, почему в языке SQL предусмотрены и ограничения базовой таблицы, и общие ограничения). Кроме того, язык SQL поддерживает **отложенную проверку ограничений.** Наконец, в этой главе кратко показано, как в языке SQL осуществляется поддержка триггеров.

УПРАЖНЕНИЯ

9.1. Какие операции могут вызвать нарушение ограничений, которые определены в примерах 1—6 раздела 9.1?

9.2. Приведите формулировки примеров 1—6 из раздела 9.1 на той версии языка Tutorial D, в которой применяется алгебра. Какие формулировки, по вашему мне нию, являются более удобными — на основе исчисления предикатов или алгебры? Почему?

9.3. Составьте ограничения целостности для приведенных ниже *бизнес-правил* для базы данных поставщиков, деталей и проектов с использованием синтаксиса Tutorial D на основе исчисления предикатов, который описан в разделе 9.2.

а) В базе данных могут находиться данные только о следующих городах: Лондон, Париж, Рим, Афины, Осло, Стокгольм, Мадрид и Амстердам (London, Paris, Rome, Athens, Oslo, Stockholm, Madrid, Amsterdam).

б) Единственно допустимыми номерами поставщиков являются такие номера, которые могут быть представлены в виде символьной строки с длиной меньше двух символов, из которых первым является "S", а остальные обозначают деся тичное целое число в пределах от 0 до 9999.

в) Все детали красного цвета должны иметь вес меньше 50 фунтов.

г) Никакие два проекта не могут находиться в одном и том же городе.

д) В Афинах не могут находиться одновременно больше одного поставщика.

е) Ни одна поставка не может иметь объем, превышающий больше чем в два раза средний объем всех таких поставок.

ж) Поставщик с самым высоким статусом не должен находиться в том же городе, где находится поставщик с самым низким статусом.

з) Каждый проект должен находиться в том городе, где находится по меньшей мере один поставщик.

и) Каждый проект должен находиться в том городе, где находится по меньшей мере один поставщик деталей для этого проекта.

к) Должна существовать по меньшей мере одна деталь красного цвета.

л) Средний статус поставщика должен быть больше 19.

м) Каждый поставщик из Лондона должен поставлять деталь с номером P2.

н) По меньшей мере одна деталь красного цвета должна иметь вес меньше 50 фунтов.

о) Поставщики из Лондона должны поставлять больше деталей разных видов, чем поставщики из Парижа.

п) В общем поставщики из Лондона должны поставлять больше деталей, чем поставщики из Парижа.

р) Ни один объем поставки не может быть сокращен (в одной операции обновления) меньше чем наполовину его текущего значения.

с) Поставщики из Афин могут переезжать только в Лондон или Париж, а поставщики из Лондона могут переезжать только в Париж.

9.4. Применительно к каждому из полученных вами ответов на упр. 9.3 выполните следующее:

а) укажите, является ли сформированное ограничение ограничением переменной отношения или ограничением базы данных;

б) приведите примеры операций, которые могут вызвать нарушение ограничения.

9.5. С использованием примеров данных о поставщиках, деталях и проектах, приведенных на рис. 4.5 (см. стр. 154), определите, каким будет результат каждой из следующих операций:

а) обновление с помощью операции UPDATE данных о проекте J7 — присваивание атрибуту CITY значения New York;

б) обновление с помощью операции UPDATE данных о детали P5 — присваивание атрибуту P# значения P4;

в) обновление с помощью операции UPDATE данных о поставщике S5 — присваивание атрибуту S# значения S8, если в качестве соответствующего ссылочного действия определено RESTRICT;

г) удаление с помощью операции DELETE данных о поставщике S3, если в качестве соответствующего ссылочного действия определено CASCADE;

д) удаление с помощью операции DELETE данных о детали P2, если в качестве со ответствующего ссылочного действия определено RESTRICT;

е) удаление с помощью операции DELETE данных о проекте J4, если в качестве соответствующего ссылочного действия определено CASCADE;

ж) обновление с помощью операции UPDATE данных о поставке s1-P1-J1—присваивание атрибуту s# значения S2;

з) обновление с помощью операции UPDATE данных о поставке S5-P5-J5 — присваивание атрибуту J# значения J7;

и) обновление с помощью операции UPDATE данных о поставке S5-P5-J5 — присваивание атрибуту J# значения J8;

к) вставка с помощью операции INSERT данных о поставке S5-P6-J7; .

л) вставка с помощью операции INSERT данных о поставке S4-P7- J6;

м) вставка с помощью операции INSERT данных о поставке S1-P2 - j j j (где j j j представляет собой заданный по умолчанию номер проекта).

9.6. В данной главе рассматривались правила удаления и обновления внешнего ключа, но ни разу не упоминалось какое-либо "правило вставки" внешнего ключа. Объясните, почему не существует такого правила.

9.7. В базе данных повышения квалификации содержится информация о внутрифирменной системе повышения квалификации служащих компании. Для каждого курса обучения в базе данных имеются сведения обо всех подготовительных курсах, необходимых для освоения этого курса, и обо всех потоках, предусмотренных для прохождения этого курса; для каждого потока в базе данных имеются сведения обо всех преподавателях и обо всех служащих, зачисленных на этот поток. Кроме того, в базе данных хранится информация о служащих. Ниже приведены краткие определения соответствующих переменных отношения.

```
COURSE     { COURSE#, TITLE }          /* Курс          */
PREREQ     { SUP COURSE#, SUB COURSE# } /* Подготовительный
курс */
OFFERING   { COURSE#, OFF#, OFFDATE, LOCATION } /* Поток     */
TEACHER    { COURSE#, OFF#, EMP# }       /* Преподаватель   */
ENROLLMENT{ COURSE#, OFF#, EMP#, GRADE }  /* Зачисление  */
EMPLOYEE   { EMP#, ENAME, JOB }          /* Служащий    */
```

Смысл переменной отношения PREREQ состоит в том, что в ней показано, какой вспомогательный курс (SUB_COURSE#) применяется для непосредственной подготовки к усвоению основного курса (SUP_COURSE#). Назначение остальных переменных отношения должно быть понятно без каких-либо дополнительных пояснений. Начертите для этой базы данных соответствующую ссылочную диаграмму. Приведите также соответствующие определения базы данных (т.е. запишите соответствующий набор определений типов и переменных отношения).

9.8. Две следующие переменные отношения представляют базу данных, содержащую информацию об отделах и служащих.

```
DEPT { DEPT#, . . . , MGR EMP#,
... } EMP { EMP#, ..., DEPT#,
... }
```

В каждом отделе есть руководитель (MGR_EMP#), и каждый из служащих работает в одном из отделов (DEPT#). Начертите ссылочную диаграмму и составьте необходимые определения данных для этой базы данных.

9.9. Две следующие переменные отношения представляют базу данных, содержащую информацию о служащих и программистах.

```
EMP  { EMP#, ..., JOB, ...
} PGMR { EMP#, ..., LANG,
... }
```

Каждый программист является служащим, но обратное утверждение неверно. И в этом случае начертите ссылочную диаграмму и составьте необходимые определения данных для этой базы данных.

9.10. Потенциальные ключи по определению являются множествами атрибутов. Что произойдет, если рассматриваемое множество окажется пустым (т.е. не содержащим атрибутов)? Можете ли вы представить себе какую-либо область применения такого "пустого" (или "нуль-арного") потенциального ключа?

9.11. Допустим, что R — переменная отношения степени n. Каково максимальное количество потенциальных ключей, которыми может обладать R?

9.12. Допустим, что A и B— две переменные отношения. Определите потенциальный ключ (ключи) для каждого из следующих операторов.

а) `A WHERE . . .`

б) `A { . . . }`

в) `A TIMES B`

г) `A UNION B`

д) `A INTERSECT B`

е) `A MINUS B`

ж) `A JOIN B`

з) `EXTEND A ADD exp AS Z`

и) `SUMMARIZE A PER B ADD exp AS Z`

к) `A SEMIJOIN B`

л) `A SEMIMINUS B`

В каждом случае предполагается, что A и B соответствуют требованиям к рассматриваемой операции (например, в случае UNION они имеют один и тот же тип).

9.13. Повторно выполните упр. 9.10, заменив слово "потенциальный" словом "внешний" (дважды).

9.14. Приведите решение упр. 9.3 на языке SQL.

9.15. Составьте определения базы данных по условиям упр. 9.7-9.9 на языке SQL.

9.16. В данной главе было показано, что каждая переменная отношения (и фактически каждое отношение) соответствует некоторому предикату. Является ли истинным обратное утверждение?

9.17. В одной из сносок в разделе 9.7 указано, что если значения s1 и London присутствуют вместе в некотором кортеже, то это может означать (в числе многих других возможных интерпретаций), что поставщик S1 не имеет офиса в городе London. Фактически данная конкретная интерпретация чрезвычайно маловероятна. Объясните, почему.

Подсказка. Вспомните предположение о замкнутости мира.

СПИСОК ЛИТЕРАТУРЫ

9.1. Aiken A., Hellerstein J.M., and Widom J. Static Analysis Techniques for Predicting the Behavior of Active Database Rules//ACM TODS. — March 1995. — 20, № 1.

В этой статье продолжена работа, начатая в [9.2], [9.5], в том числе, над *системами экспертных баз данных* (здесь они именуются *системами активных баз данных).* В частности, в статье описана система правил, применяемая в прототипе Starburst компании IBM (см. [18.21], [18.48], [26.19], [26.23] , [26.29], [26.30] и [9.25]).

9.2. Baralis E. and Widom J. An Algebraic Approach to Static Analysis of Active Database Rules // ACM TODS. — September 2000. — 25, № 3. Ранняя версия этой статьи: Baralis E. and Widom J. An Algebraic Approach to Rule Analysis in Expert Database Systems // Proc. 20th Int. Conf. on Very Large Data Bases. — Santiago, Chile. — September 1994.

В этой статье под словом "правила", вынесенным в заголовок, по сути подразумеваются триггеры. Одна из проблем, связанных с использованием таких правил, состоит в том, что (как отмечено в разделе 9.11) их поведение является исключительно трудным как для понимания, так и для прогнозирования. В статье предложены методы определения (еще до этапа вызова правил на выполнение) того, обладает ли данное множество правил свойствами завершенности и конфлюэнтности. Свойство *завершенности* означает, что обработка правила гарантированно не будет продолжаться бесконечно. Свойство *конфлюэнтности* означает, что окончательное состояние базы данных не зависит от порядка, в котором выполнялись эти правила.

9.3. Bernstein P.A., Blaustein B.T., Clarke E.M. Fast Maintenance of Semantic Integrity Assertions Using Redundant Aggregate Data // Proc. 6th Intern. Conf. on Very Large Data Bases. — Montreal, Canada. — October 1980.

Здесь представлен эффективный метод приведения в действие ограничений целостности особого рода, например, таких: "Любое значение в множестве A должно быть меньше любого значения в множестве B". Метод предписания этих ограничений основан, в частности, на том наблюдении, что данное ограничение, по сути, логически эквивалентно ограничению: "Максимальное значение в множестве A должно быть меньше минимального значения в множестве B". Распознавая ограничения подобного рода и автоматически сопровождая соответствующие максимальные и минимальные значения в скрытых переменных, система может сократить количество сравнений, связанных с соблюдением ограничения на данное обновление от величины, порядок которой определяется кардинальностью множеств A или в (в зависимости от того, к какому множеству применяется обновление)

примерно до единицы, безусловно, за счет расходов на сопровождение скрытых переменных.

9.4. Buneman O.P., Clemons E.K. Efficiently Monitoring Relational Databases // ACM TODS. — September 1979. — 4, № 3.

В статье рассматриваются задачи эффективной реализации триггерных процедур (которые здесь именуются *предупреждающими),* в частности, анализируется возможность определения того, удовлетворяется ли триггерное условие, без обязательного вычисления этого условия. В ней описан метод (алгоритм предотвращения — avoidance algorithm), позволяющий обнаружить обновления, которые, по всей вероятности, не будут удовлетворять данному триггерному условию. Кроме того, представлен способ сокращения издержек обработки в тех случаях, когда выполнение алгоритма предотвращения оканчивается неудачей. Этот способ предусматривает проведение оценки триггерного условия для некоторого небольшого подмножества (топологического фильтра) полного множества охватываемых кортежей.

9.5. Ceri S., Widom J. Deriving Production Rules for Constraint Maintenance // Proc. 16th Intern. Conf. on Very Large Data Bases. — Brisbane, Australia. — August 1990.

В статье рассматривается язык на основе SQL, предназначенный для определения ограничений, и приведен алгоритм, с помощью которого система может обнаружить все операции, способные нарушить указанное ограничение. (Краткое предварительное описание этого алгоритма было приведено в [9.12].) Данная статья касается также вопросов оптимизации и правильности.

9.6. Ceri S., Cochrane R.J., Widom J. Practical Applications of Triggers and Constraints: Successes and Lingering Issues // Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt. — September 2000.

Приведем цитату из резюме: "Значительная часть ... триггерных приложений фактически представляет собой не что иное, как средства поддержки ограничений целостности различных классов". По ходу изложения в данной статье показано, что многие триггеры, включая, в частности, триггеры, названные в статье "средствами поддержки ограничений", действительно могут быть сгенерированы автоматически на основании декларативных спецификаций.

9.7. Ceri S., Fraternali P., Paraboschi S., and Tanca L. Automatic Generation of Production Rules for Integrity Maintenance //ACM TODS. — September 1994. — 19, № 3.

В статье, основанной на работе [9.5], представлены возможности автоматического исправления искажений, внесенных в результате нарушения ограничения. Ограничения транслируются в порождающие правила с помощью следующих компонентов.

1. Список операций, которые могут нарушить ограничение.
2. Логическое выражение, в результате вычисления которого будет получено зна чение TRUE, если ограничение нарушено (по сути, оно представляет собой про сто отрицание первоначального ограничения).
3. Восстановительная процедура на языке SQL.

В статье приведен также широкий обзор работ, связанных с этой темой.

9.8. Cochrane R., Pirahesh H., and Mattos N. Integrating Triggers and Declarative Constraints in SQL Database System // Proc. 22 Int. Conf. on Very Large Data Bases. — Mumbai (Bombay), India. — September 1996.

Приведем цитату из этой статьи: "Семантика взаимодействия триггеров и декларативных ограничений должна быть тщательно определена, чтобы можно было избежать нарушений совместимости базы данных с ограничениями при выполнении операций и предоставить пользователям всестороннюю модель, позволяющую понять такие взаимодействия. Такая модель определена [в данной статье]". Рассматриваемая модель стала основой для разработки соответствующих спецификаций в стандарте SQL: 1999.

9.9. Codd E.F. Domains, Keys, and Referential Integrity in Relational Databases // InfoDB3. - 1988.-3,№1.

Обсуждение понятий домена, первичного ключа и внешнего ключа. Безусловно, данная статья Кодда представляет значительный интерес, поскольку все эти три понятия были предложены Коддом. Тем не менее, по мнению автора данной книги, в этой статье слишком много вопросов осталось нерешенными или необъясненными. Кстати, в ней приведен следующий довод в пользу правила, согласно которому один из потенциальных ключей обязательно должен быть выбран в качестве первичного: "Отказ от использования этого правила равносилен стремлению использовать компьютер со схемой адресации ..., в которой основание системы счисления изменяется всякий раз, когда происходит событие определенного вида (например, встречается адрес, который представляет собой простое число)". Но если мы согласимся с этим доводом, то почему бы не согласиться с его логическим заключением и не использовать идентичную схему адресации для всех данных? Разве мы не допускаем такое же "упущение", когда адресуем данные о поставщике с помощью номера поставщика и данные о деталях с помощью номеров деталей, не говоря уже о поставках, когда применяются составные "адреса". (Фактически, по поводу идеи глобально единообразной схемы адресации можно сказать еще очень многое. См. обсуждение темы свернутых ключей в аннотации к [14.21] в главе 14.)

9.10. Date C.J. Referential Integrity // Proc. 7th Intern. Conf. on Very Large Data Bases. — Cannes, France. — September 1981. Позднее была издана пересмотренная версия этой статьи (Date C.J. Relational Database: Selected Writings. — Reading, Mass.: Addison-Wesley, 1986).

В статье впервые дано определение понятия *ссылочных действий* (преимущественно RESTRICT и CASCADE), которые обсуждались в разделе 9.10 этой главы. Основное различие между первой версией статьи *(VLDB,* 1981) и пересмотренной версией состоит в том, что в первой версии, которая была написана под влиянием [14.7], допускалось, чтобы внешний ключ ссылался на любое количество переменных отношения, тогда как в пересмотренной версии (по причинам, подробно описанным в [9.10]) автор уже не придерживается такой чрезмерно общей позиции.

9.11. Date C.J. Referential Integrity and Foreign Keys (в двух частях) // Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

В первой части этой статьи обсуждается история понятия ссылочной целостности, а также предлагается ряд предпочитаемых автором основных определений (с пояснениями). Во второй части приведены дополнительные доводы, позволяющие понять, почему были выбраны именно эти определения, и даны некоторые практические рекомендации. В частности, обсуждаются проблемы, которые могут возникать при использовании перекрывающихся внешних ключей, составных внешних ключей, частично представленных пустыми значениями, и *смежных ссылочных путей* (т.е. различимых ссылочных путей, начальная и конечная точки которых совпадают).

Примечание. Определенные положения этой статьи могут быть поставлены под сомнение (но не очень существенно) в соответствии с доводами, изложенными в [9.14].

9.12. Date C.J. A Contribution to the Study of Database Integrity // Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

Приведем цитату из резюме: "В статье предпринята попытка в определенной степени структурировать задачу [обеспечения целостности]. Во-первых, в ней предложена схема классификации ограничений целостности, во-вторых, эта схема используется для разъяснения основополагающих понятий целостности данных, в-третьих, кратко описан подход к созданию конкретного языка формулирования ограничений целостности, и, в-четвертых, дано определение конкретных областей для дальнейшего исследования". Настоящая глава частично основана на материалах этой ранней статьи, но саму предлагаемую схему классификации следует рассматривать как устаревшую и замененную пересмотренной версией, которая описана в разделе 9.9 настоящей главы.

9.13. Date C.J. Integrity // Глава 11 книги Date C.J. and Colin J.W. A Guide to DB2 (4th edition). — Reading, Mass.: Addison-Wesley, 1993.

В программном продукте DB2 компании IBM действительно была предусмотрена декларативная поддержка первичных и внешних ключей (фактически этот программный продукт был одним из первых, в которых имелась такая поддержка, возможно, даже самым первым). Но, как показано в [9.13], предусмотренная в DB2 поддержка первичных и внешних ключей страдает от определенных ограничений реализации, основным назначением которых является *обеспечение гарантий предсказуемого поведения.* Рассмотрим следующий простой пример. Предположим, что переменная отношения R в настоящее время содержит только два кортежа со значениями первичного ключа, соответственно, 1 и 2, и рассмотрим такой запрос на обновление: "Удвоить значение каждого первичного ключа в R". Правильным результатом выполнения этого запроса является то, что кортежи должны теперь иметь значения первичного ключа, соответственно, 2 и 4. Если система вначале обновит значение "2" (заменив его на "4"), а затем обновит "1" (заменив его на "2"), то данный запрос будет выполнен успешно. А если, с другой стороны, система обновит (или, скорее, попытается обновить) вначале "1" (заменив его на "2"), то будет обнаружено нарушение ограничения уникальности и запрос окончится неудачей (база данных останется в неизменном состоянии). Иными словами, *результат данного запроса является непредсказуемым.* В целях предотвращения такого

непредсказуемого поведения в DB2 просто запрещаются ситуации, в которых могли бы возникать подобные варианты. Но, к сожалению, некоторые из создаваемых в результате ограничений становятся весьма строгими [9.20].

Следует отметить, что (как показывает приведенные выше пример) в системе DB2 обычно выполняется "оперативная проверка". Это означает, что в этой системе
проверки целостности применяются к каждому отдельному кортежу во время обновления данного кортежа. Но такая оперативная проверка является логически неправильной (см. подраздел "Обновление переменных отношения" в разделе 6.5 главы 6); дело в том, что подобный способ проверки принят в основном в целях повышения производительности.

9.14. Date C.J. The Primacy of Primary Keys: An Investigation // Relational Database Writings 1991-1994. — Reading, Mass.: Addison-Wesley, 1995.

В статье представлены доводы в пользу такой позиции, что иногда не совсем оправдан такой подход, когда один потенциальный ключ становится "первым среди равных", т.е. первичным.

9.15. Date C.J. WHAT Not HOW: The Business Rules Approach to Application Develop ment. — Reading, Mass.: Addison-Wesley, 2000.

Очень неформальное (и не требующее больших усилий в изучении) введение в понятие *бизнес-правил.* См. также [9.21] и [9.22].

9.16. Date C.J. Constraints and Predicates: A Brief Tutorial (в трех частях) // http: //www.dbdebunk. com, (May 2001).

Настоящая глава написана в основном по материалам этого учебника. То же самое можно сказать по поводу приведенной ниже отредактированной (и сокращенной) версии заключительного раздела учебника.

"Как уже было отмечено, база данных представляет собой коллекцию истинных высказываний. Фактически база данных вместе с операциями, которые могут применяться к высказываниям, хранящимся в этой базе данных, представляет собой **логическую** систему. А здесь под термином *логическая система* подразумевается формальная система (которую можно, например, сравнить с эвклидовой геометрией). В ней имеются аксиомы ("высказывания, рассматриваемые как истинные") и правила вывода, с помощью которых мы можем доказывать теоремы ("производные истинные высказывания") на основании этих аксиом. И действительно, идея Кодда (возникшая в период разработки реляционной модели в 1969 году), согласно которой база данных, несмотря на то, что она носит такое название, фактически является не просто коллекцией данных, а скорее коллекцией фактов, или тех выражений смысла, которые в логике называются *истинными высказываниями,* была исключительно важным выводом. Эти высказывания (после того, как они сформулированы, т.е. представлены в базовых переменных отношения) становятся аксиомами рассматриваемой логической системы. А правилами логического вывода по сути являются правила, с помощью которых из существующих высказываний образуются новые; иными словами, они являются правилами, позволяющими нам узнать, как должны использоваться операции реляционной алгебры. Таким образом, при вычислении в системе некоторого реляционного

выражения (в частности, при формировании в ней ответа на некоторый запрос) фактически происходит формирование новых истинных высказываний на основании существующих. В действительности, система при этом доказывает теорему!

Как только мы признаем справедливость изложенного выше представления, то сумеем понять, что для плодотворного решения *проблем базы данных* может использоваться весь аппарат формальной логики. Иными словами, вопросы, подобные перечисленным ниже (не считая вопроса о том, как должны выглядеть ограничения целостности), становятся фактически логическими задачами, которые могут рассматриваться в логической трактовке и поддаются решению в рамках логического подхода.

- Как база данных должна выглядеть с точки зрения пользователя?
- Каким должен быть язык запросов?
- Каким образом результаты должны быть представлены пользователю?
- Как следует наилучшим образом реализовывать запросы (или, если речь идет о более широкой постановке задачи, вычислять выражения базы данных)?
- Какие задачи необходимо решать в первую очередь при проектировании базы данных?

Безусловно, следует также отметить, что реляционная модель непосредственно поддерживает изложенный выше подход к восприятию общей концепции базы данных. Именно по этой причине, по мнению автора, реляционная модель является абсолютно надежной, обоснованной и имеющей неограниченные перспективы дальнейшего развития.

Наконец, зная о том, что база данных вместе с реляционными операциями действительно представляет собой логическую систему, мы можем понять, в чем состоит **столь значительная важность** ограничений целостности. Если в базе данных нарушены какие-то ограничения целостности, то логическая система, о которой мы здесь упоминаем, становится несовместимой с этими ограничениями. А на ответы, полученные от системы, содержащей противоречия, абсолютно невозможно положиться! Предположим, что рассматриваемая система находится в таком состоянии, что в ней одновременно являются истинными выражения p и NOT p (в этом и заключается несовместимость с ограничениями), где p — некоторое высказывание. Теперь допустим, что q — другое, произвольное высказывание. Из этого следуют приведенные ниже выводы.

- На основании того, что выражение p является истинным, можно заключить, что выражение p OR q является истинным.
- На основании того, что выражения p OR q и NOT p являются истинными, можно заключить, что выражение q является истинным.

Но высказывание q было взято произвольным образом! Это означает, что в системе, содержащей противоречия, можно доказать истинность любого высказывания".

9.17. Hammer M.M., Sarin S.K. Efficient Monitoring of Database Assertions // Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data. — Austin, Texas. — May/June 1978.

В статье приведен набросок алгоритма, предназначенного для выработки процедур проверки целостности, которые являются более эффективными по сравнению с тем очевидным методом простой проверки ограничений после выполнения обновлений, который можно назвать методом с применением "грубой силы". Предлагаемые процедуры проверки встраиваются в объектный код приложения во время компиляции. В некоторых случаях рассматриваемый алгоритм позволяет обнаружить, что проверки на этапе прогона приложения вообще не требуются. Но даже если такие проверки являются необходимыми, часто имеется возможность значительно сократить количество операций доступа к базе данных с помощью различных способов.

9.18. Horowitz B. M. A Run-Time Execution Model for Referential Integrity Maintenance // Proc. 8th IEEE Int. Conf. on Data Engineering. — Phoenix, Ariz. — February 1992.

Широко известно, что определенные комбинации перечисленных ниже информационных компонентов при совместном их применении могут приводить к некоторым конфликтным ситуациям и, в принципе, могут вызывать непредсказуемое поведение со стороны приложения (дополнительные сведения можно найти, например, в [9.11]).

1. Ссылочные структуры (т.е. коллекции переменных отношения, между которы ми установлены связи с помощью ограничений ссылочной целостности).
2. Правила удаления и обновления внешнего ключа.
3. Фактические значения данных в базе данных.

Существует три основных подхода к решению этой проблемы:

а) предоставить пользователю возможность самому справляться с этими пробле мами;
б) предусмотреть в системе средства обнаружения и исключения попыток опреде лить структуры, применение которых может в принципе привести к конфлик там во время выполнения;
в) предусмотреть в системе средства обнаружения и исключения конфликтов, фактически возникающих во время выполнения.

Вариант а) является, безусловно, неприемлемым, а вариант б) требует исключительно тщательной проработки ([9.13], [9.20]), поэтому автор данной статьи Горовиц (Horowitz) предлагает использовать вариант в). В статье предложен набор правил для определения таких действий, выполняемых во время прогона приложения, и доказана их правильность. Но следует отметить, что в этой статье не рассматривается вопрос о том, как повлияет на производительность такая организация проверки во время прогона приложения.

Горовиц активно участвовал в работе комитета, ответственного за определение стандарта SQL: 1992, и в тех частях указанного стандарта SQL, которые касаются

вопросов ссылочной целостности, фактически подразумевается, что должны поддерживаться предложения данной статьи.

9.19. Markowitz V. M. Referential Integrity Revisited: An Object-Oriented Perspective // Proc. 16th Int. Conf. on Very Large Data Bases. — Brisbane, Australia. — August 1990.

В заголовке этой статьи есть слова "перспективы объектно-ориентированного подхода" ("Object-Oriented Perspective"), полностью соответствующие утверждению, с которого она начинается: "Ограничения ссылочной целостности лежат в основе реляционного представления объектно-ориентированных структур". Но фактически эта статья вообще не посвящена описанию объектов в объектно-ориентированном смысле. Скорее, она является описанием алгоритма, позволяющего составить такое определение реляционной базы данных, начиная с диаграммы "сущность—связь" (см. главу 14), в котором гарантированно не возникают некоторые проблемные ситуации, указанные в [9.11] (например, перекрывающиеся ключи).

Кроме того, в данной статье с точки зрения ссылочной целостности рассматриваются три коммерческих продукта (DB2, Sybase и Ingres, по состоянию примерно на 1990 год). В ней показано, что в DB2, где предусмотрена декларативная поддержка, ограничения являются слишком жесткими. Что касается Sybase и Ingres, в которых предусмотрена процедурная поддержка (соответственно, с помощью *триггеров* и *правил),* то они являются менее ограничительными, чем DB2, но громоздкими и сложными в эксплуатации (хотя и отмечено, что поддержка ограничений в Ingres является "технически более совершенной" по сравнению с Sybase).

9.20. Markowitz V. M. Safe Referential Integrity Structures in Relational Databases // Proc. 17th Int. Conf. on Very Large Data Bases. — Barselona, Spain. — September 1991.

Предложены два формальных *условия безопасности,* гарантирующие, что не могут возникнуть некоторые из проблемных ситуаций, описанных, например, в [9.11] и [9.18]. В этой статье также рассматривается вопрос о том, что требуется для обеспечения этих условий в DB2, Sybase и Ingres (это опять-таки касается состояния дел примерно к 1990 году). Применительно к DB2 показано, что некоторые ограничения реализации, налагаемые в целях обеспечения безопасности [9.13], не являются логически необходимыми, и вместе с тем, что другие ограничения являются недостаточными (т.е. DB2 все равно допускает возникновение некоторых небезопасных ситуаций). А что касается Sybase и Ingres, то в статье утверждается, что процедурная поддержка, предусмотренная в этих программных продуктах, не обеспечивает обнаружения небезопасных (или даже неправильных!) спецификаций в ограничениях ссылочной целостности.

9.21. Ross R. G. The Business Rule Book: Classifying, Defining, and Modeling Rules (Version 3.0). — Boston, Mass.: Database Research Group, 1994.

См. аннотацию к [9.22].

9.22. Ross R.G. Business Rule Concepts. — Houston, Tex.: Business Rule Solutions Inc., 1998.

За последние несколько лет в сообществе пользователей коммерческих СУБД намного вырос интерес к поддержке *бизнес-правил.* Некоторые ключевые фигуры в

промышленности заняли такую позицию, согласно которой бизнес-правила могли бы служить лучшей основой для проектирования и создания баз данных, а также приложений для баз данных (это означает, что они считают бизнес-правила лучшими по сравнению с такими более сложившимися методами, как применение модели "сущность—связь", объектное моделирование, семантическое моделирование и др.). И автор с этим согласен, поскольку бизнес-правила по сути представляют собой не что иное, как более дружественный (т.е. менее сухой и менее формальный) способ рассуждения о предикатах, высказываниях и всех прочих аспектах целостности, которые рассматривались в настоящей главе. Одним из самых выдающихся защитников подхода, основанного на использовании бизнес-правил, является Росс (Ross), и его книги можно смело рекомендовать самым эрудированным практикам. В [9.21] дано всестороннее описание данной темы, а в [9.22] приведено краткое учебное руководство.

Примечание. Ко времени написания настоящей книги была опубликована еще одна книга Росса (Principles of the Business Rule Approach. Addison-Wesley, 2003).

9.23. Stonebraker M.R., Wong E. Access Control in a Relational Data Base Management System by Query Modification // Proc. ACM National Conf. — San Diego, Calif. — November 1974.

В прототипе СУБД University Ingres [8.11] был впервые применен интересный подход к определению ограничений целостности (и ограничений защиты — см. главу 17), основанный на одном из вариантов запроса. В этой СУБД для определения ограничений целостности применялся оператор DEFINE INTEGRITY, который имеет следующий синтаксис.

DEFINE INTEGRITY ON <*relvar name*> IS <*bool exp*>

Например, один из таких операторов может быть представлен следующим образом.

```
DEFINE INTEGRITY ON S IS S.STATUS > 0
```

Предположим, что пользователь и предпринимает попытку выполнить следующий оператор REPLACE, который представлен на языке QUEL.

```
REPLACE S ( STATUS = S.STATUS - 10 ) WHERE S.CITY = "London"
```

В таком случае СУБД Ingres автоматически преобразует оператор REPLACE в следующую форму.

```
REPLACE S ( STATUS = S.STATUS -
10 ) WHERE   S.CITY = "London"
AND      ( S.STATUS - 10 ) > 0
```

Очевидно, что вероятность нарушения ограничения целостности при использовании этого модифицированного оператора практически отсутствует.

Один из недостатков указанного подхода состоит в том, что с помощью такого простого способа нельзя обеспечить соблюдение всех ограничений; фактически в языке QUEL поддерживались только такие ограничения, в которых простым условием ограничения было логическое выражение. Однако в то время даже столь

ограниченная поддержка намного превосходила возможности большинства других систем.

9.24. Walker A., Salveter S. C. Automatic Modification of Transactions to Preserve Data Base Integrity Without Undoing Updates: Technical Report 81/026.— State University of New York, Stony Brook, N.Y.: Technical Report 81/026. — June 1981.

В этой работе описан метод автоматического преобразования любого *шаблона транзакции* (т.е. исходного кода транзакции) в соответствующий безопасный шаблон. Он является безопасным в том смысле, что ни одна транзакция, соответствующая этому модифицируемому шаблону, по всей вероятности, не может нарушить какие-либо объявленные ограничения целостности. В основе этого метода лежит процедура добавления к первоначальному шаблону запросов и проверок для обеспечения того, чтобы никакие ограничения не были нарушены (еще до выполнения любых операций обновления). Если на этапе прогона приложения любая из заранее предусмотренных проверок оканчивается неудачей, то запрос на выполнение транзакции отклоняется и вырабатывается сообщение об ошибке.

9.25. Widom J. and Ceri S. (eds.). Active Database Systems: Triggers and Rules for Advanced Database Processing. — San Francisco, Calif: Morgan Kaufmann, 1996.

Полезный сборник исследований и руководств по *активным системам баз данных* (так принято называть системы баз данных, которые автоматически выполняют указанные действия в ответ на указанные события, другими словами, системы базы данных с триггерами). В него включены описания нескольких прототипов систем, в том числе СУБД Starburst, разработанной в лаборатории IBM Research (см. [18.21], [18.48], [26.19], [26.23], [26.29] и [26.30]), и СУБД Postgres, которая разработана в Калифорнийском университете, г. Беркли (см. [26.36], [26.40], [26.42] и [26.43]). Кроме того, в этой книге приведены относящиеся к рассматриваемой теме сведения о спецификации SQL: 1992, ранней версии спецификации SQL: 1999 и о некоторых коммерческих продуктах (в том числе о СУБД Oracle, Informix и Ingres). В книге представлена обширная библиография.

ГЛАВА 10

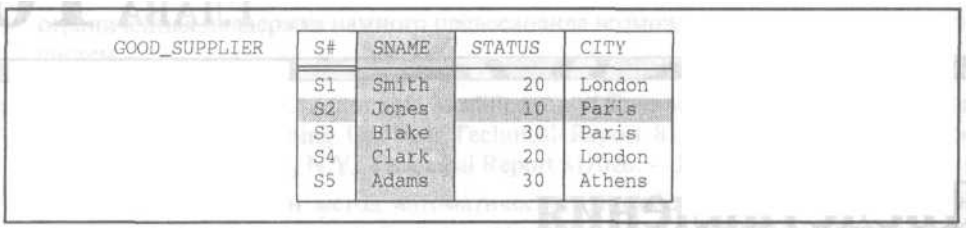
Представления

10.1. ВВЕДЕНИЕ

Как отмечалось в главе 3, *представление* в реляционной модели, по сути, является именованным выражением реляционной алгебры (либо конструкцией, эквивалентной выражению реляционной алгебры). Ниже приведен пример на языке Tutorial D.

```
VAR GOOD_SUPPLIER VIEW
    ( S WHERE STATUS > 15 ) { S#, STATUS, CITY } ;
```

При выполнении данного оператора выражение реляционной алгебры (которое называется выражением, определяющим представление) не вычисляется, а просто запоминается системой посредством его записи в каталог базы данных под указанным именем, GOOD_SUPPLIER. Тем не менее, со стороны пользователя это выглядит так, как будто в базе данных существует реальная переменная отношения GOOD_SUPPLIER с собственными кортежами и атрибутами, показанными в незатененной части рис. 10.1 (имеется в виду, конечно, постоянно используемый в этой книге пример значений данных). Другими словами, имя GOOD_SUPPLIERS обозначает производную (и виртуальную) переменную отношения. Ее значением в любой момент является отношение, которое было бы получено в качестве результата, если бы выражение, определяющее представление, было действительно вычислено в данный момент.



GOOD_SUPPLIER

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

Рис. 10.1. Представление GOOD_SUPPLIER как заданный фрагмент базовой переменной отношения S (незатененные части таблицы)

В главе 3 было также показано, что любое представление, такое как GOOD_SUPPLIER, в некотором смысле можно считать просто *окном* для просмотра данных. Любые изменения, вносимые в исходную базовую таблицу, будут автоматически и мгновенно отображаться в представлении (как в окне), конечно, если они попадут в область, охваченную данным представлением. И наоборот, все изменения, вносимые в представление, будут автоматически переноситься в данные исходной базовой таблицы, в результате чего станут видимыми[1] и в самом представлении.

В реальной ситуации многие пользователи могут даже не предполагать, что GOOD_SUPPLIER — это представление. Одни пользователи могут знать об этом, а также о том, что существует реальная переменная отношения S, тогда как другие будут искренне верить, что GOOD_SUPPLIERS — настоящая переменная отношения. Но, в любом случае, важнее всего то, что с представлением можно работать так, как будто оно действительно *является* настоящей переменной отношения. Приведем пример запроса к представлению GOOD_SUPPLIER.

```
GOOD_SUPPLIER WHERE CITY ≠ 'London' ;
```

Ниже показан результат выполнения этого запроса при использовании данных, приведенных на рис. 10.1.

S#	STATUS	CITY
S3	30	Paris
S5	30	Athens

Этот запрос, безусловно, подобен обычному запросу к обычной *реальной* переменной отношения. Как отмечалось в главе 3, система обрабатывает запросы такого типа путем их преобразования в запросы к базовой переменной отношения (или к нескольким базовым переменным отношения). Подобное преобразование осуществляется посредством замены в определении запроса всех вхождений *имени* представления тем выражением, которое его определяет. В нашем примере после выполнения **процедуры подстановки** будет получен следующий запрос.

```
( ( S WHERE STATUS > 15 ) { "S#, STATUS, CITY } )
                          WHERE CITY Φ 'London' ;
```

[1] Фактически могут возникать ситуации, при которых эти изменения окажутся невидимыми в SQL! Эта тема рассматривается при обсуждении конструкции WITH CHECK OPTION в разделе 10.6.

Вполне очевидно, что данное выражение можно легко преобразовать в более простую форму.

```
( S WHERE STATUS > 15 AND CITY ≠ 'London' )
                                    { S#, STATUS, CITY } ;
```

Результат вычисления последнего выражения совпадает с приведенным выше результатом запроса к представлению.

В этой связи необходимо отметить, что процесс подстановки (т.е. процесс замены имени представления определяющим его выражением) выполняется *корректно вследствие реляционного свойства замкнутости.* Свойство замкнутости (кроме всего прочего) означает, что в любом месте выражения, где разрешено использовать имя переменной отношения R, вместо него можно применять некоторое реляционное выражение произвольной сложности (при том условии, что в результате вычисления этого выражения будет получено отношение, имеющее такой же тип, как и R). Другими словами, возможность использовать представления появляется именно благодаря тому, что в реляционной алгебре множество всех отношений является замкнутым, и это еще один пример, подтверждающий фундаментальное значение реляционного свойства замкнутости.

Операции обновления на представлениях выполняются аналогичным образом. Например, рассмотрим следующую операцию.

```
UPDATE GOOD SUPPLIER WHERE CITY =
     'Paris' { STATUS := STATUS + 10
     } ;
```

При обработке она будет приведена к следующему виду.

```
UPDATE S WHERE STATUS > 15 AND CITY =
     'Paris' { STATUS := STATUS +10 }
     ;
```

При использовании операций INSERT и DELETE применяется тот же подход..

Дополнительные примеры

Ниже приведены некоторые дополнительные примеры, на основании которых можно сделать важные выводы.

1.
```
VAR  REDPART VIEW
       ( P WHERE COLOR = COLOR ('Red') ) { ALL BUT COLOR }
                                        RENAME WEIGHT AS WT ;
```
   Представление REDPART — это проекция сокращения (в которой предусмотрено также переименование атрибутов), применяемая к переменной отношения с данными о деталях. В нем присутствуют атрибуты P#, PNAME, WT и CITY, и помещаются только те кортежи, которые описывают детали красного цвета.

2.
```
VAR  PQ  VIEW
     SUMMARIZE SP PER P { P# } ADD SUM ( QTY ) AS TOTQTY ;
```
   Представление PQ можно рассматривать как *статистический итог* или результат *обобщения данных* исходной переменной отношения.

3.
```
VAR  CITY_PAIR VIEW
       ( ( S RENAME CITY AS SCITY ) JOIN SP JOIN
         ( P RENAME CITY AS PCITY ) ) { SCITY, PCITY } ;
```

В представлении CITY_PAIR выполняется соединение таблиц с данными о поставщиках, деталях и поставках по номерам поставщиков и номерам деталей, а затем проекция результатов по столбцам SNAME и PNAME. Говоря неформально, в представлении CITY_PAIR пара имен городов (x,y) появляется в результате тогда и только тогда, когда поставщик, находящийся в городе x, поставляет детали, которые хранятся в городе y. Например, предположим, что поставщик si поставляет детали P1; поставщик S1 находится в Лондоне и детали P1 хранятся в Лондоне; в этом случае в представлении появляется пара (London, London).

```
4. VAR HEAVY_REDPART VIEW
       REDPART WHERE WT > WEIGHT ( 12.0 ) ;
```

В этом примере показано, как одно представление определяется через другое.

Определение и удаление представлений

Синтаксис оператора определения представления на языке Tutorial D выглядит следующим образом.

```
VAR <relvar name> VIEW <relation exp> <candidate key def list> ;
```

В приведенном выражении параметр с указанием списка потенциальных ключей <*candidate key def list*> может быть пустым (это равносильно тому, что может быть опущена соответствующая спецификация), поскольку система должна обладать способностью определить потенциальные ключи представления самостоятельно [3.3]. Например, в случае представления GOOD_SUPPLIER системе должно быть известно, что единственным потенциальным ключом, о котором может идти речь, является ключ {S#}, унаследованный от исходной базовой переменной отношения S.

Как уже отмечалось (используя терминологию ANSI/SPARC, обсуждавшуюся в главе 2), определения представлений объединяют в себе функции *внешней схемы* и функции *отображения уровня "внешний—концептуальный",* поскольку они описывают и сам внешний объект (т.е. представление), и способ его отображения на концептуальный уровень (т.е. на одну или несколько исходных базовых переменных отношения).

Примечание. Некоторые определения представлений задают не отображение уровня "внешний—концептуальный", а отображение уровня *"внешний—внешний"*. Представление HEAVY_REDPART из предыдущего раздела относится именно к такой категории представлений.

Синтаксис оператора удаления представления имеет следующий вид.

```
DROP VAR <relvar name>  ;
```

Здесь параметр <rel var *name*> определяет имя удаляемого представления. В главе 6 было выдвинуто предположение, что попытка удалить базовую переменную отношения должна завершиться неудачей, если существует хотя бы одно определение представления, ссылающееся на удаляемую переменную отношения. Аналогично, следует полагать, что удаление представления, на которое имеется ссылка в определении какого-либо другого представления, также приведет к неудачному завершению. Другой вариант состоит в том, что можно дополнить синтаксис оператора определения представления (по аналогии со ссылочными ограничениями), включив в него, кроме подразумеваемой опции RESTRICT, и опцию CASCADE. Опция RESTRICT, применяемая по умолчанию, означает,

что попытка удаления любой переменной отношения, на которую имеется ссылка в определении данного представления, будет отвергнута. А при использовании опции CASCADE такое удаление происходит успешно, но одновременно происходит также каскадное удаление всех прочих представлений, которые ссылаются на удаляемое представление.

Примечание. В стандартной версии языка SQL такая опция поддерживается, но ее определение должно быть помещено в оператор удаления DROP, а не в оператор определения представления. Кроме того, в операторе DROP не предусмотрено использование того или другого значения опции по умолчанию; в нем необходимо явно указывать одно из значений опции (см. раздел 10.6).

10.2. ОБЛАСТЬ ПРИМЕНЕНИЯ ПРЕДСТАВЛЕНИЙ

Поддержка представлений желательна по многим причинам. Укажем некоторые из них.

- *Пользователям предоставляется возможность использовать средства сокращенной записи операторов — своего рода "макросы ".*

  Рассмотрим запрос "Определить все города, в которых хранятся детали, поставляемые некоторым поставщиком, находящимся в Лондоне". Требуемый запрос можно легко сформулировать с помощью представления CITY_PAIR (пары городов), определенного в подразделе "Дополнительные примеры" предыдущего раздела.

```
( CITY_PAIR WHERE SCITY = 'London' ) { PCITY }
```

  Сформулировать данный запрос, не пользуясь этим представлением, будет намного сложнее.

```
( ( ( S RENAME CITY AS SCITY ) JOIN SP JOIN
    ( P RENAME CITY AS PCITY ) )
 WHERE SCITY = 'London' ) { PCITY
 }
```

  Хотя пользователь и мог бы применить последнюю формулировку, обращаясь непосредственно к базовой переменной отношения (конечно, только в том случае, когда установленные ограничения защиты позволят ему это сделать), первая формулировка запроса, безусловно, проще. Разумеется, что первая формулировка запроса на самом деле является просто сокращением второй. Перед выполнением запроса системный механизм обработки представлений развертывает первое выражение и получает запрос в виде второго выражения.

  В этом просматривается явная аналогия с *макросами* в языках программирования. В принципе, пользователь *может* непосредственно вводить в программный код развернутые выражения, но (по очевидным причинам) намного удобнее использовать сокращенную запись в форме макросов, возложив процедуру их развертывания на макропроцессор языка программирования. Аналогичные замечания применимы и в отношении представлений. Таким образом, в СУБД представления играют роль, аналогичную роли макросов в системах программирования, а хорошо известные преимущества и выгода от использования макросов имеют место (с *соответствующими оговорками)* и в случае представлений. В частности, следует отметить, что использование представлений, как и использование макросов, не приводит на этапе выполнения к снижению производительности приложений —

некоторые дополнительные затраты имеют место только на этапе развертывания представлений (как и в случае макросов).

- *Представления позволяют разным пользователям различным образом видеть одни и те же данные в одно и то же время.*

  Другими словами, представления позволяют различным пользователям сосредоточить свое внимание и, возможно, логически реструктуризировать только ту часть базы данных, которая их интересует, игнорируя все остальные хранимые данные. Это соображение особенно важно для интегрированных баз данных, с которыми одновременно и независимо друг от друга работает множество категорий пользователей, имеющих самые различные требования.

- *Обеспечивается автоматическая защита скрытых данных.*

  Под *скрытыми данными* здесь подразумеваются данные базовых таблиц, которые не видны в определенном представлении (например, в случае представления GOOD_SUPPLIER это имена поставщиков). Такие данные надежно защищены от нежелательного доступа через конкретное представление (по крайней мере, с помощью операции выборки). Таким образом, ограничив доступ пользователей к базе данных некоторым набором представлений, можно реализовать простой и эффективный механизм защиты. Мы возвратимся к вопросу использования представлений в целях защиты в главе 17.

- *Представления могут обеспечивать логическую независимость от данных.*

  Это одно из важнейших потенциальных преимуществ представлений, поэтому мы рассмотрим его отдельно в следующем разделе.

Логическая независимость от данных

Напомним, что логическая независимость от данных может быть определена как отсутствие влияния *изменений в логической структуре базы данных на работу пользователей и пользовательских программ* (где под *логической структурой* подразумевается концептуальный или *общий логический* уровень; см. главу 2). Несомненно, что представления являются именно тем средством, с помощью которого в реляционной системе может быть достигнута логическая независимость от данных. Логическая независимость от данных имеет два важных следствия — **рост** и **реструктуризация** базы данных становятся беспрепятственными.

Примечание. Аспект *роста* здесь обсуждается только для полноты изложения. Он достаточно важен, но его связь с представлениями весьма относительна.

- *Рост базы данных.*

  По мере роста базы данных, аккумулирующей новые виды информации, должно соответственно возрастать и количество используемых в ней определений структур данных. Возможны две разновидности роста.

  а) Расширение существующей базовой переменной отношения в целях включения нового атрибута. В таком случае новый атрибут предназначен для вновь добавляемых данных, относящихся к существующему типу объектов. В качестве примера приведем добавление атрибута DISCOUNT (скидка) в базовую переменную отношения поставщиков.

б) Создание новой базовой переменной отношения для добавления в базу данных информации об объектах нового типа. Примером может служить внесение в базу данных поставщиков и деталей сведений о проектах.

Ни одна из указанных разновидностей роста не должна оказывать какого-либо влияния на работу существующих пользователей или пользовательских программ, по крайней мере, в принципе (однако см. в этой связи пункт 6 упражнения 8.6.1, приведенного в главе 8, в котором рассматривается одна из проблем, возникающих при использовании конструкции "SELECT *" в языке SQL).

■ *Реструктуризация базы данных.*

Иногда возникает необходимость провести реструктуризацию базы данных таким образом, чтобы ее общее информационное наполнение оставалось тем же, а изменилось только *логическое расположение* данных. Другими словами, иногда требуется та или иная перегруппировка атрибутов базовых переменных отношения. Рассмотрим лишь один простой пример реструктуризации. Предположим, что по какой-то причине (в данном случае не важно, в чем она состоит) необходимо заменить переменную отношения S следующими двумя переменными отношения.

```
VAR SNC BASE RELATION { S# S#, SNAME NAME, CITY CHAR }
    KEY { S# } ;
```

ill.

```
VAR ST BASE RELATION { S# S#, STATUS INTEGER
    } KEY { S# } ;
```

Здесь существенно то, что *прежняя переменная отношения* S *становится соединением двух новых переменных отношения* SNC *и* ST (а обе новые переменные отношения, SNC и ST, являются проекциями старой переменной отношения S). Следовательно, можно создать представление, которое будет предусматривать выполнение указанного соединения, и присвоить ему имяэ.

```
VAR  S  VIEW
    SNC  JOIN  ST  ;
```

Теперь любое выражение, в котором раньше использовалась переменная отношения S, будет ссылаться на представление S. Следовательно, *если предположить, что система корректно поддерживает операции манипулирования данными в представлениях,* то пользователи и пользовательские программы действительно окажутся логически независимыми[2] от данной конкретной реструктуризации базы данных.

---

[2] Это возможно лишь в принципе! К сожалению, современные продукты SQL (и сам стандарт языка SQL) в целом не поддерживают операции манипулирования данными в представлениях должным образом и, следовательно, не обеспечивают в полной мере логической независимости от изменений, подобных показанным в этом примере. Говоря конкретнее, большинство программных продуктов (но не все) в настоящее время корректно поддерживают только операции выборки данных через представления, но, насколько известно автору, ни один из продуктов SQL не поддерживает правильно операции обновления с помощью представлений (к тому же, безусловно, это не предусмотрено и стандартом), поэтому современные продукты SQL не обеспечивают полную логическую независимость от данных применительно к операциям обновления. *Примечание.* Есть один программный продукт, который правильно поддерживает операции обновления с помощью представлений (хотя он не является программным продуктом SQL), который описан в [20.1].

Следует отметить, что замена исходной переменной отношения S, содержащей данные о поставщиках, двумя проекциями этой переменной отношения, SNC и ST, в общем случае не такая уж тривиальная задача. В частности, следует учитывать, что некоторых дополнительных действий потребует переменная отношения SP, содержащая сведения о поставках, поскольку в ней используется внешний ключ, ссылающийся на исходную переменную отношения с данными о поставщиках S (см. упр. 10.14).

Но возвратимся к главной теме обсуждения. Из примера с переменными отношения SNC и ST, конечно, не следует, что логическая независимость от данных может быть достигнута при *любой возможной* реструктуризации. Ключевой вопрос здесь состоит в том, возможно ли однозначное отображение между версией базы данных после реструктуризации и ее исходной версией (т.е. обратима ли выполненная реструктуризация базы данных). Другими словами, вопрос заключается в том, являются ли эти две версии базы данных **информационно эквивалентными.** Если нет, то совершенно очевидно, что логическая независимость от данных не будет достигнута.

Два важных принципа

В результате проведенного выше обсуждения логической независимости от данных возникает еще один вопрос. Дело в том, что фактические представления имеют два совершенно разных назначения.

- Пользователь, который *сам определяет* некоторое представление V, безусловно, знаком с соответствующим выражением X, определяющим это представление. Он может использовать имя V везде, где применимо выражение X, рассматривая его при этом просто как сокращенную запись данного выражения.
- Пользователь, которому известно лишь то, что представление v существует и его можно применять, чаще всего незнаком с выражением X, определяющим это представление. С его точки зрения представление V должно выглядеть и действовать точно так же, как обычная базовая переменная отношения.

Из сказанного можно сделать вывод, что вопрос о том, какой является данная переменная отношения, базовой или производной (т.е. представлением), в известной степени зависит от точки зрения пользователя! Рассмотрим случай переменных отношения S, SNC и ST, использовавшихся в обсуждении вопросов *реструктуризации* в предыдущем разделе. Очевидно, что существуют следующие два возможных подхода к анализу взаимосвязи между этими переменными отношения:

- определить S как базовую переменную отношения, а SNC и ST — как представления, являющиеся проекциями этой базовой переменной отношения; или
- определить SNC и ST как базовые переменные отношения, а S — как представление, являющееся соединением этих двух базовых переменных отношения[3].

Из этого следует, что не должно быть никаких произвольных и ненужных различий между базовыми и производными переменными отношения. Мы называем это утверждение **принципом взаимозаменяемости** базовых и производных переменных отношения. Заметим

[3] См. обсуждение *декомпозиции без потерь* в разделе 12.2 главы 12.

в частности, что из этого принципа следует, что *должна* существовать возможность обновления представлений. Иначе говоря, возможности обновления базы данных не должны зависеть от произвольного по существу решения, какие переменные отношения считаются базовыми, а каким — производными. Обсуждение этого вопроса будет продолжено в разделе 10.4.

А пока условимся называть множество всех базовых переменных отношения *реальной* базой данных. Но типичный пользователь взаимодействует (в общем случае) не с реальной базой данных *как таковой, а.* с тем, что можно назвать *представительной* базой данных, состоящей (опять-таки, в общем случае) из некоторой смеси базовых переменных отношения и представлений. Теперь предположим, что ни одна из переменных отношения в такой представительной базе данных не может быть производной от остальных (в противном случае такая переменная отношения могла бы быть удалена без потери данных). Поэтому *с точки зрения пользователя* все эти переменные отношения являются базовыми переменными отношения по определению, поскольку они, безусловно, не зависят одна от другой (т.е. все они автономны в соответствии с терминологией главы 3). То же самое относится и к самой базе данных, т.е. выбор, какая база данных является *реальной,* может быть сделан произвольно, поскольку все возможные варианты информационно равносильны. Последний вывод мы будем называть **принципом относительности базы данных.**

10.3. ВЫБОРКА ДАННЫХ ИЗ ПРЕДСТАВЛЕНИЙ

В предыдущих разделах кратко описывалась процедура преобразования операций выборки из представлений в эквивалентные операции выборки из одной или нескольких базовых переменных отношения. В данном разделе приводится более формальное описание этого преобразования.

Прежде всего, следует отметить (как указано в конце раздела 6.4 главы 6), что любое заданное реляционное выражение можно рассматривать как **функцию** на множестве отношений. Иными словами, текущие значения различных переменных отношения, упоминаемых в выражении, представляют собой фактические параметры данного вызова этой функции, а результатом ее вычисления является другая переменная отношения. Пусть D — это база данных (которая будет представлена в данном случае как множество переменных отношения), а V— это представление, определенное на множестве D, т.е. представление, определение которого является функцией x на множестве D, как показано ниже.

V = X (D)

А теперь предположим, что RO — операция выборки из представления V. Тогда очевидно, что RO также является функцией на множестве отношений, а результат выборки будет иметь следующий вид.

RO (V) = RO (X (D))

Таким образом, результат операции выборки по определению совпадает с результатом вычисления функции X на множестве D, т.е. с результатом **материализации** копии отношения, являющегося текущим значением представления V, с последующим применением операции RO к этой материализованной копии. Но на практике обычно эффективнее вместо этой операции использовать описанную выше процедуру подстановки (см. раздел 10.1).

Эта процедура равносильна формированию функции с (...), являющейся *композицией* RO (x(...)) функций x и RO (именно в этом порядке), и вычислению результата применения функции с непосредственно к множеству D. Но как бы то ни было, все-таки удобнее, по крайней мере, концептуально, дать определение семантики операции выборки из представлений в терминах материализации, а не подстановки. Другими словами, подстановка допустима лишь постольку, поскольку она гарантирует получение того же результата, который мог быть получен при использовании материализации (и это, безусловно, гарантируется).

Все изложенное в предыдущем разделе должно быть в основном уже знакомо читателю, по крайней мере, в принципе, так как эти темы уже рассматривались в данной книге. Тем не менее, автор счел необходимым еще раз привести эти сведения по следующим причинам.

- Во-первых, они создают основу для аналогичного (но более глубокого) обсуждения операций обновления в следующем разделе.
- Во-вторых, становится очевидным, что материализация представляет собой безусловно допустимый способ реализации представлений, по крайней мере, в случае операций выборки (хотя, возможно, довольно неэффективный). Однако этот способ, конечно же, не может быть использован при выполнении операции обновления, поскольку смысл обновления представления заключается в применении указанной операции обновления именно к лежащим в основе представления базовым переменным отношения, а не просто к некоторой материализованной *копии* их данных (об этом также более подробно будет сказано в следующем разделе).
- В-третьих, хотя, в принципе, процедура замены представления его определением вполне понятна и теоретически полностью обоснована, весьма огорчает тот факт, что в некоторых программных продуктах SQL (ко времени написания данной книги) этот процесс практически *не* действует успешно! Иначе говоря, в подобных продуктах выборка данных из представлений может совершенно неожиданно завершаться неудачей. Процедура подстановки определения не реализована в версиях стандарта SQL, предшествующих версии SQL: 1992. А причиной, по которой в приложениях и предыдущих версиях стандарта SQL операции выборки из представлений не действуют должным образом, является неполная поддержка ими реляционного свойства замкнутости (см. упр. 10.15, часть *а)*.

10.4. ОБНОВЛЕНИЕ ДАННЫХ В ПРЕДСТАВЛЕНИЯХ

Представления — это переменные отношения и поэтому (как и все переменные) должны быть обновляемыми по определению. Но задача обновления представлений всегда рассматривалась как очень сложная. Проблема обновления данных в представлениях может быть сформулирована следующим образом. Пусть дана некоторая операция обновления данных в заданном представлении. Какие обновления и в какие исходные базовые переменные отношения нужно внести, чтобы реализовать исходное обновление представления? Формальное описание проблемы выглядит так. Пусть D — это база данных, а V — представление, определенное на D (т.е. представление, определение которого является функцией X на множестве D), следующим образом (как показано в разделе 10.3).

V = X (D)

Теперь предположим, что uo — это операция обновления в представлении V. Так как uo можно считать операцией, результат выполнения которой состоит в изменении ее фактического параметра, то допустимо представить ее в следующем виде.

UO (V) = UO (X (D))

Тогда проблема выполнения обновления в представлении сводится к поиску такой операции обновления uo' на множестве D, для которой истинно следующее выражение.

UO (X (D)) = X (UO' (D))

Операция подобного вида требуется по той причине, что реально существует только множество D (представления по определению виртуальны) и выполнять операции обновления непосредственно в представлениях *как таковых* невозможно.

Следует подчеркнуть, что на протяжении последних лет проблема обновления представлений была предметом ряда важных исследований, в результате проведения которых было разработано множество различных подходов к решению этой проблемы. Подробные сведения об этих исследованиях можно найти, например, в [10.4], [10.7]—[10.10], [10.12]. К ним относятся, в частности, предложения Кодда (Codd) по реляционной модели RM/V2 [6.2]. В этой главе изложен относительно новый подход [10.6], [10.11], менее *произвольный* (характеризующийся наличием лучшего теоретического обоснования) по сравнению с применявшимися ранее подходами. Но самым важным его преимуществом является совместимость с наилучшими средствами предыдущих подходов. Кроме того, новый подход позволяет считать обновляемым гораздо более широкий класс представлений по сравнению с прежними подходами. При этом подходе фактически *все* представления считаются потенциально обновляемыми, при том условии, что не нарушаются установленные ограничения целостности.

Еще раз о золотом правиле

Напомним первую (более простую) версию *золотого правила,* представленную в предыдущей главе.

Никакая операция обновления не должна ни при каких условиях присваивать переменной отношения такое значение, при котором предикат переменной отношения принимает значение FALSE.

Другая (немного менее формальная) версия этого золотого правила приведена ниже.

Ни одна переменная отношения ни при каких условиях не должна противоречить своему собственному предикату.

Примечание. Во всей данной главе термин *предикат переменной отношения* используется для обозначения именно соответствующего внутреннего предиката, а неуточненный термин *предикат* используется для обозначения только такого предиката переменной отношения. Фактически такое же соглашение применяется во всей данной книге, если явно не указано иное.

При определении этого правила подчеркивалось, что оно применимо ко *всем* переменным отношения, как к базовым, так и к производным. В частности, было указано, что производные переменные отношения также имеют предикаты, как и должно быть

согласно *принципу взаимозаменяемости.* Таким образом, эти предикаты должны быть известны системе, что позволит ей правильно выполнять обновления представлений. Что же собой представляет предикат представления? Очевидно, что нам прежде всего необходим набор **правил вывода предиката,** такой что если известен предикат (предикаты) на входе (входах) любой реляционной операции, то с его помощью можно определить предикат на выходе этой операции. Если будет получен такой набор правил, то мы сможем вывести предикат представления из предиката базовой переменной отношения (или переменных отношения), в терминах которой прямо или косвенно было определено это представление. (Безусловно, предикаты для всех базовых переменных отношения можно считать известными: они представляют собой логическую конъюнкцию всех ограничений, которые были объявлены для данной базовой переменной отношения.)

На самом деле, определить требуемый набор правил очень легко — они следуют непосредственно из определений реляционных операторов. Например, если A и B — две произвольные переменные отношения некоторого типа, имеющие предикаты, соответственно, PA и PB, а представление C определено как A INTERSECT B, то очевидно, что предикат PC этого представления будет определяться выражением (PA) AND (PB). Доказательство этого утверждения приведено ниже.

- Кортеж t появится в представлении c лишь при том условии, что он присутствует одновременно и в представлении A, и в представлении в.
- Если кортеж t присутствует в представлении A, то выражение PA (t) должно быть истинным (здесь выражение "PA(t)" применяется как обозначение высказывания, которое становится результатом конкретизации предиката PA с использованием значений атрибутов t в качестве формальных параметров).
- Аналогичным образом, если кортеж t присутствует в представлении B, то должно быть истинным выражение PB (t).
- Следовательно, должно быть также истинным выражение PA (t) AND PB (t); это означает, что предикат PC представляет собой результат конкатенации предикатов PA и PB (что и требовалось доказать).

О других реляционных операторах речь пойдет ниже в этом же разделе.

Таким образом, производные переменные отношения автоматически *наследуют* определенные ограничения от тех переменных отношения, на основе которых они определены. Однако вполне возможно, что некоторая производная переменная отношения станет объектом определенных дополнительных ограничений, налагаемых помимо и сверх унаследованных ограничений [3.3]. Поэтому для производных переменных отношения желательно иметь возможность явно устанавливать требуемые ограничения (например, вводить ограничения потенциального ключа для представления). Язык Tutorial D действительно поддерживает такую возможность. Однако для упрощения изложения в дальнейшем мы будем чаще всего игнорировать эту возможность.

Принципы создания механизма обновления представлений

Существует еще несколько важных принципов, которые должны соблюдаться при любом систематическом подходе к проблеме обновления представлений. (Безусловно, **золотое правило** — важнейшее из них, но не единственное.) Перечислим эти принципы.

1. Проблема обновляемое™ представлений является семантической, а не синтаксической. Другими словами, ее решение не зависит от выбранной формы записи определения представления. Например, два приведенных ниже представления семантически эквивалентны.

```
VAR V VIEW
    S WHERE STATUS > 25 OR CITY = 'Paris' ;

VAR V VIEW
   ( S WHERE STATUS > 25 ) UNION ( S WHERE CITY = 'Paris' ) ;
```

   Очевидно, эти оба представления должны быть одновременно либо обновляемыми, либо не обновляемыми (в действительности они, безусловно, должны быть обновляемыми). Однако вопреки этому утверждению и в стандарте SQL, и в большинстве существующих на данный момент продуктов SQL, принят совершенно необоснованный подход, что первое из приведенных представлений должно быть обновляемым, а второе — нет (подробности приведены в разделе 10.6).

2. Как следует из приведенных выше рассуждений, механизм обновления представлений должен действовать правильно и в тех частных случаях, когда *представление* на самом деле является базовой переменной отношения, поскольку любая базовая переменная отношения в семантически неотличима от представления V, определенного на основании операции в UNION в, или в INTERSECT в, или в WHERE TRUE, или любого другого выражения, тождественно эквивалентного переменной отношения в. Таким образом, правила обновления, установленные, например, для представления, которое определено с помощью операции объединения v = в UNION в, должны давать тот же результат, как и в случае, когда рассматриваемая операция обновления применяется непосредственно к базовой переменной отношения в. Иначе говоря, хотя тема данного раздела и звучит как *обновление представлений,* на самом деле в общем случае здесь рассматривается обновление *переменных отношения.* Поэтому здесь рассматривается теория обновления, применимая ко всем переменным отношения, а не только к представлениям.

3. В применяемых правилах обновления должна соблюдаться симметричность во всех случаях, когда она имеет смысл. Например, правило удаления кортежа из представления, определенного как пересечение V = A INTERSECT в, не должно допускать произвольного удаления кортежа из переменной отношения А, но не из в, даже в том случае, если и такое одностороннее удаление кортежа наверняка повлечет за собой в дальнейшем удаление этого кортежа из рассматриваемого представления. В подобном случае должно быль предусмотрено удаление кортежа из обеих переменных отношения А и В. (Другими словами, не должно быть какой-либо *неоднозначности.* Всегда должен существовать единственный способ реализации данного обновления, который эффективно осуществляется во всех случаях. В частности, не должно быть никаких логических различий между представлениями, определенными какА INTERSECT Ви В INTERSECT А.)

4. В правилах обновления должны учитываться любые соответствующие активизируемые действия, включая, в частности, конкретные действия, касающиеся поддержки ссылочной целостности, такие как каскадное удаление.

5. Для упрощения синтаксиса желательно рассматривать операцию UPDATE как сокращенную запись последовательности операций удаления и вставки (DELETE—INSERT); именно так мы и будем ее рассматривать. Это сокращение вполне приемлемо при *соблюдении* следующих условий.
   - Не должно осуществляться никаких проверок предикатов переменных отношения в ходе выполнения любого требуемого обновления. Под этим подразумевается, что расширенная запись операции UPDATE выглядит как DELETE-INSERT—*проверка,* а не как DELETE—*проверка*—INSERT—*проверка.* Причина этого состоит в том, что операция DELETE может на время нарушить истинность предиката переменной отношения, притом что законченная операция UPDATE ее не нарушает. Например, предположим, что переменная отношения R содержит ровно 10 кортежей, и рассмотрим воздействие операции UPDATE на некоторый кортеж переменной отношения R, при том условии, что согласно предикату этой переменной отношения R, в ней не может содержаться меньше 10 кортежей. Очевидно, что проверка результатов выполнения операции DELETE завершится неудачей из-за нарушения условий предиката.
   - Триггерные процедуры также никогда не должны активизироваться в ходе выполнения любой операции обновления. (Фактически триггерные процедуры активизируются после завершения операции, непосредственно перед проверкой предиката отношения.)
   - Сокращенная запись требует некоторых незначительных уточнений, если она используется для представлений, определяемых на основе операции проекции (подробности приведены ниже в этом разделе).
6. Все обновления данных в представлениях должны быть реализованы как обновления того же типа в исходных переменных отношения. Иначе говоря, операции вставки (INSERT) отображаются в операции вставки, а операции удаления (DELETE) отображаются в операции удаления (как было сказано выше, операции обновления UPDATE можно не рассматривать). Предположим обратное, т.е. что существует некоторый тип представлений (скажем, представления, определяемые на основе операции объединения), для которых операции INSERT отображаются в операции DELETE в исходных переменных отношения. Но тогда операции INSERT *для базовых переменных отношения* также должны в каких-то случаях отображаться в операции DELETE! Этот вывод непосредственно следует из утверждения (как мы уже убедились в ходе рассуждений в п. 2), что любую базовую переменную отношения B можно представить в виде семантически идентичного представления V = в UNION в. Аналогичное рассуждение можно применить и для представлений остальных типов (представлений на основе операций сокращения, проекции, пересечения и т.д.). Однако утверждение, что в базовых переменных отношения операции вставки данных могут в действительности представляться операциями удаления, — очевидный абсурд. Таким образом, утверждение, что операции INSERT и DELETE в представлениях отображаются в соответствующие операции INSERT и DELETE в базовых переменных отношения, является истинным.

7. В общем случае правила обновления при их применении к заданному представле нию V определяют операции, которые должны быть применены к переменным от ношения, на которых определено данное представление V. Эти правила должны правильно срабатывать даже в том случае, если те переменные отношения, на ос нове которых определено данное представление, в свою очередь, также являются представлениями. Другим словами, правила должны допускать возможность *ре курсивного применения*. Конечно, если попытка обновить исходную переменную отношения по каким-то причинам завершится неудачно, то и исходная операция обновления не будет выполнена. Другими словами, обновления в представлениях выполняются по принципу "все или ничего", точно так же, как и в случае базовых переменных отношения.
8. Правила не должны основываться на предположении, что база данных хорошо спроектирована, т.е. полностью нормализована (подробности приведены в гла вах 12 и 13). Тем не менее, выполнение установленных правил может привести к несколько неожиданным результатам, если база данных спроектирована действи тельно *неудачно.* Это еще раз свидетельствует в пользу необходимости создания добротных проектов базы данных. В следующем разделе будет приведен подобный пример с несколько неожиданными результатами.
9. Не должно существовать никаких *априорно признаваемых весомыми* причин того, чтобы в любом заданном представлении одни операции обновления разрешались, а другие — нет (например, когда разрешена операция DELETE, НО запрещена опе рация INSERT).
10. Операции INSERT И DELETE ДОЛЖНЫ быть, насколько это возможно, взаимообратными.

Необходимо сделать одно важное напоминание. Как было показано в главе 6, реляционные операции, в частности реляционные обновления, всегда выполняются на уровне множеств, а множество, состоящее из одного кортежа, просто является одним из частных случаев. Более того, иногда *необходимо* выполнять обновление сразу многих кортежей (например, если операцию обновления нельзя воспроизвести с помощью серии обновлений отдельных кортежей). Это утверждение в общем случае верно как для базовых переменных отношения, так и для представлений. Для простоты изложения большую часть обсуждаемых правил обновления представлений мы будем формулировать в терминах операций обновления отдельных кортежей; тем не менее, читатель не должен забывать о том, что подобный подход — это всего лишь дань простоте изложения, в некоторых случаях даже чрезмерная.

Далее мы по очереди рассмотрим отдельные операции реляционной алгебры (объединение, пересечение и разность), а затем — все остальные операции. В первых трех случаях, в частности, подразумевается, что речь идет о представлении, определяющее выражение которого имеет, соответственно, одну из следующих трех форм: A UNION в, A INTERSECT в и A MINUS в, где A и в, в свою очередь, являются некоторыми реляционными выражениями (т.е. не обязательно представляют собой базовые переменные отношения). Переменные отношения A и в должны иметь один и тот же реляционный тип. Соответствующими предикатами рассматриваемых переменных отношения являются РА И РВ.

Примечание. В некоторых правилах и примерах, приведенных ниже в этой главе, упоминается возможность возникновения побочных эффектов, а побочные эффекты, как известно, обычно нежелательны. Тем не менее, побочные эффекты могут стать неизбежными, если переменные отношения A и в представляют собой пересекающиеся подмножества одной и той же базовой переменной отношения, как часто бывает в случае применения к представлениям операций объединения, пересечения и разности. Мало того, рассматриваемые побочные эффекты (на этот раз) становятся отнюдь не нежелательными, а, напротив, желательными.

Операция объединения

Приведем правило вставки для представления видаA UNION B.

- **Правило INSERT.** Вновь добавляемый кортеж должен удовлетворять либо предикату PA, либо предикату PB, либо обоим предикатам одновременно. Если новый кортеж удовлетворяет предикату PA, то этот кортеж должен быть вставлен в переменную отношения A (заметим, что операция вставки может иметь побочный эффект, в результате которого новый кортеж будет вставлен и в переменную отношения в). Если новый кортеж удовлетворяет предикату PB, то он вставляется и в переменную отношения в, но лишь в том случае, если он еще не вставлен в эту переменную отношения в результате побочного эффекта от вставки кортежа в переменную отношения A.

Пояснение. Новый кортеж должен удовлетворять по крайней мере одному из предикатов, PA или PB, так как в противном случае данный кортеж не будет включен в объединение A UNION в, поскольку он не будет удовлетворять предикату производной переменной отношения для объединения A UNION B, а именно — (PA) OR (PB). (Кроме того, мы предполагаем, хотя такое предположение и не является строго обязательным, что новый кортеж не должен в данный момент присутствовать ни в переменной отношения A, ни в переменной отношения в, так как в противном случае это означало бы попытку вставить уже существующий кортеж.) Если предположить, что перечисленные условия удовлетворяются, то новый кортеж будет вставлен в переменную отношения A или переменную отношения в в зависимости от того, к какой из них этот кортеж логически принадлежит (возможно также, что он логически принадлежит к обеим).

Примечание. Конкретный процедурный способ, используемый в формулировке приведенного выше правила (согласно которому вначале предлагается вставить кортеж в переменную отношения A, а затем — в переменную отношения в), следует воспринимать только как педагогический прием. Не нужно полагать, что СУБД должна будет выполнять операцию вставки именно в той последовательности, которая указана в определении правила. В действительности, согласно принципу симметрии (принцип 3, приведенный в предыдущем подразделе), при вставке кортежа ни одна из переменных отношения, A или в, не имеет преимуществ перед другой. Аналогичные поправки относятся ко всем правилам, рассматриваемым в этом разделе.

Примеры. Пусть представление uv определено следующим образом.

```
VAR UV VIEW
  ( S WHERE STATUS > 25 ) UNION ( S WHERE CITY = 'Paris' ) ;
```

Рис. 10.2. Представление UV (пример значений)

На рис. 10.2 приведены возможные значения данных в этом представлении, соответствующие применяемому в данной книге примеру базы данных.

UV

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris
S5	Adams	30	Athens

- Пусть кортеж, который необходимо вставить в представление UV, имеет вид[4] (S6, Smith, 50, Rome). Этот кортеж удовлетворяет предикату s WHERE STATUS > 25, но не предикату S WHERE CITY = 'Paris'. Следовательно, новый кортеж вставляется в переменную отношения, удовлетворяющую предикату s WHERE STATUS > 25. В соответствии с правилами вставки кортежей в представление на основе операции сокращения (которые являются вполне очевидными, как пока зано ниже в этом разделе), новый кортеж будет вставлен в базовую переменную отношения с данными о поставщиках, и по этой причине он также появится в представлении, что и требовалось получить.
- Теперь предположим, что в представление uv нужно вставить кортеж вида (S7, Jones, 50, Paris). Этот кортеж удовлетворяет одновременно двум преди катам — S WHERE STATUS > 25 И S WHERE CITY = 'Paris'. ЛОГИЧНО предположить, что он будет вставлен в обе переменные отношения, удовлетворяющие каждому из этих предикатов. Тем не менее, следует заметить, что вставка кортежа в одну из переменных отношения будет иметь побочный эффект, вследствие кото рого кортеж окажется автоматически вставленным и в другую переменную отно шения. Таким образом, вторую операцию вставки INSERT ЯВНО ВЫПОЛНЯТЬ не тре буется.

Теперь рассмотрим две различные *базовые* переменные отношения, SA и SB. Переменная отношения SA содержит информацию о поставщиках, для которых значение атрибута STATUS превышает 25, а в переменной отношения SB содержатся сведения о поставщиках из Парижа (рис. 10.3). Предположим, что представление uv определено как объединение SA UNION SB, и вновь рассмотрим операцию вставки INSERT двух уже упомянутых выше кортежей. Вставка кортежа (S6, Smith, 50,Rome) в представление UV приведет к вставке кортежа в базовую переменную отношения SA, что, по-видимому, и требуется. Однако вставка кортежа (S7, Jones, 50, Paris) в представление UV приведет к вставке кортежа в *обе* переменные отношения SA и SB! Полученный результат логически корректен, хотя интуитивно и не совсем понятен (именно этот результат в предыдущем подразделе был назван "несколько неожиданным"). *С точки зрения автора подобные "неожиданности"могут иметь место исключительно как следствие плохо спроектированной структуры базы данных.* В частности, на его взгляд, если проект базы данных позволяет

[4] Подобное упрощенное обозначение для кортежей переменной-отношения используется во всем этом разделе из соображений наглядности.

одним и тем же кортежам появляться (т.е. удовлетворять их предикатам) в разных базовых переменных отношения, то это, определенно, плохой проект. Такая позиция (возможно, спорная) подробно рассматривается в разделе 13.6 главы 13.

SA

S#	SNAME	STATUS	CITY
S3	Blake	30	Paris
S5	Adams	30	Athens

SB

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris
S3	Blake	30	Paris

Рис. 10.3. Базовые переменные отношения SA и SB (пример значений атрибутов)

Теперь рассмотрим правила удаления кортежей из представлений типа UNION в.

- **Правило DELETE.** Если удаляемый кортеж принадлежит к переменной отношения A, то он удаляется из нее (заметьте, что данная операция удаления DELETE может иметь побочный эффект, вследствие которого этот же кортеж будет удален и из переменной отношения в). Если после удаления кортежа из переменной отношения A этот кортеж все еще остается в переменной отношения в, то он будет удален и из переменной отношения в.

Оставляем читателю в качестве упражнения подготовку примеров, иллюстрирующих данное правило. Следует заметить, что удаление кортежа из переменной отношения A или в может привести к каскадному удалению или к выполнению других активизируемых действий.

И наконец, рассмотрим правило выполнения операции обновления.

- **Правило UPDATE.** Обновляемый кортеж должен быть таким, чтобы его обновленная версия удовлетворяла либо предикату PA, либо предикату PB, либо обоим этим предикатам одновременно. Если кортеж принадлежит переменной отношения A, то его удаляют из нее без запуска каких-либо активизируемых действий (каскадного удаления и т.п.), которые могут быть связаны с выполнением обычной операции удаления. Кроме того, не выполняется проверка соблюдения условий предиката переменной отношения A. Следует заметить, что эта операция удаления DELETE может иметь побочный эффект, вследствие которого выбранный кортеж будет удален и из переменной отношения в. Если после удаления из переменной отношения A обновляемый кортеж все еще сохраняется в переменной отношения в, то он удаляется из этой переменной отношения (опять-таки, без запуска каких-либо активизируемых действий и проверки ее предиката). Далее, если обновленная версия кортежа удовлетворяет предикату PA, кортеж вставляется в переменную отношения A (эта операция может иметь побочный эффект, вследствие которого новая версия данного кортежа может появиться и в переменной отношения в). И наконец, если обновленная версия кортежа удовлетворяет предикату PB, то кортеж вставляется в переменную отношения в, но только в том случае, если он не был вставлен в данную переменную отношения в результате побочного эффекта от вставки обновленного кортежа в переменную отношения A.

Приведенное выше правило обновления, по сути, состоит из правила удаления, за которым следует правило вставки, за исключением того, что, как указывалось ранее, после удаления старой версии кортежа с помощью операции DELETE не выполняются активизируемые действия и не происходит проверка предикатов (концептуально все активизируемые действия, связанные с операцией обновления, выполняются после завершения всех операций удаления и вставки, непосредственно перед проверкой предикатов).

Необходимо отметить одно важное следствие такого рода трактовки операции обновления UPDATE, которое состоит в том, что после обновления представления измененный кортеж может, выражаясь неформально, *мигрировать* из одной переменной отношения в другую. В базе данных, показанной на рис. 10.3, обновление кортежа (S5, Adams ,30, Athens) в представлении UV с преобразованием в (S5, Adams, 15, Paris) приведет к удалению старого кортежа из переменной отношения SA и его вставке в переменную отношения SB.

Операция пересечения

Приведем правила обновления представлений вида A INTERSECT B. Причем в данном случае ограничимся просто формулированием правил без каких-либо дополнительных пояснений (они аналогичны пояснениям для правил обновления объединений). Единственное, о чем следует сказать, — это то, что для представлений вида A INTERSECT в предикат принимает вид (PA) AND (PB). Подготовку примеров применения приведенных ниже правил оставляем читателю в качестве упражнения.

- **Правило INSERT.** Новый кортеж должен удовлетворять предикатам PA и PB одновременно. Если новый кортеж на текущий момент отсутствует в переменной отношения A, то он будет вставлен в эту переменную отношения (заметьте, что операция вставки может иметь побочный эффект, вследствие которого новый кортеж появится и в переменной отношения в). Если новый кортеж все еще отсутствует в переменной отношения в, то он будет вставлен в эту переменную отношения.
- **Правило DELETE.** Удаляемый из представления кортеж удаляется из переменной отношения A (заметьте, что эта операция удаления может иметь побочный эф фект, в результате которого удаляемый кортеж исчезнет и из переменной отноше ния в). Если удаляемый кортеж (все еще) присутствует в переменной отношения в, то он будет удален и из этой переменной отношения.
- **Правило UPDATE.** Обновляемый кортеж должен быть таким, чтобы его обновлен ная версия удовлетворяла одновременно обоим предикатам, PA и PB. Кортеж уда ляется из переменной отношения A без запуска активизируемых действий и про верки предиката этой переменной отношения (заметим, что данная операция может иметь побочный эффект, вследствие которого кортеж будет удален и из перемен ной отношения в). Если до этого момента обновляемый кортеж все еще не был удален из переменной отношения в, то он будет удален из нее, опять-таки, без за пуска активизируемых действий и проверки предиката. Далее, если обновленная версия кортежа в данный момент в переменной отношения A отсутствует, то изме ненный кортеж будет вставлен в переменную отношения A (заметьте, что операция вставки может также иметь побочный эффект, вследствие которого новый кортеж

автоматически появится и в переменной отношения в). Если измененный кортеж все еще отсутствует в переменной отношения в, то он будет вставлен и в эту переменную отношения.

Операция разности

Ниже приведены правила обновления представлений вида A MINUS в (для представлений данного вида предикат переменной отношения можно записать как (PA) AND NOT (PB)).

- **Правило INSERT.** Новый кортеж должен удовлетворять предикату PA и не должен удовлетворять предикату PB. Новый кортеж вставляется в переменную отношения A.
- **Правило DELETE.** Удаляемый из представления кортеж удаляется из переменной отношения A.
- **Правило UPDATE.** Обновляемый кортеж должен быть таким, чтобы его обновленная версия удовлетворяла предикату PA и не удовлетворяла предикату PB. Кортеж удаляется из переменной отношения A без выполнения активизируемых действий и проверки предиката. Затем обновленная версия кортежа вставляется в переменную отношения A.

Операция сокращения

Допустим, что выражение для определения представления V можно задать в следующем виде: A WHERE p, а предикатом переменной отношения A является PA. Тогда предикат представления v примет следующий вид.

(PA) AND (p)

Например, предикат для операции сокращения S WHERE CITY = 'London' будет иметь вид (PS) AND (CITY = 'London'), где PS является предикатом переменной отношения с данными о поставщиках. Приведем правила обновления для представлений ВИДа A WHERE p.

- **Операция INSERT.** Новый кортеж должен удовлетворять предикату PA и условию p. Новый кортеж вставляется в переменную отношения A.
- **Операция DELETE.** Удаляемый кортеж удаляется из переменной отношения A.
- **Операция UPDATE.** Обновляемый кортеж должен быть таким, чтобы его обновленная версия удовлетворяла и предикату PA, и условию p. Кортеж удаляется из переменной отношения A без запуска каких-либо активизируемых действий и проверки предиката. Затем в переменную отношения A вставляется обновленная версия кортежа.

Примеры. Пусть представление LS определено следующим образом.

```
VAR LS VIEW
    S WHERE CITY = 'London' ;
```

На рис. 10.4 приведен пример значений для этого представления.

LS

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S4	Clark	20	London

Рис. 10.4. Представление LS (пример значений)

- Попытка вставить в представление LS кортеж вида (S6, Green, 20, London) будет завершена успешно. Новый кортеж будет вставлен в переменную отношения s и, следовательно, появится также в представлении LS.
- Попытка вставить в представление LS кортеж вида (S1,Green, 20, London) окончится неудачей, поскольку этот кортеж не удовлетворяет предикату переменной отношения S (и, следовательно, представления LS). Причина ошибки в том, что этот кортеж нарушает требование уникальности потенциального ключа {S#}.
- Попытка вставить в представление LS кортеж вида (Sб,Green, 20, Athens) завершится неудачно, поскольку этот кортеж нарушает условие CITY = 'London'.
- Попытка удалить из представления LS кортеж (S1, Smith, 20, London) завершится успешно. Кортеж будет удален из переменной отношения S и, следовательно, из представления LS.
- Попытка обновить в представлении LS кортеж (S1, Smith, 20, London) с преобразованием в (S6, Green, 20, London) завершится успешно. Попытки обновить тот же кортеж (S1, Smith, 20, London) с преобразованием в (S1, Smith, 20, London) или в (S1, Smith, 20, Athens) будут неудачными (в каждом конкретном случае объясните, почему).

Операция проекции

Этот раздел также начинается с рассмотрения соответствующего предиката. Пусть атрибуты переменной отношения A (с предикатом PA) разделены на две непересекающиеся группы, допустим, X и Y . Полагая, что X и Y представляют собой *отдельные составные* атрибуты, рассмотрим проекцию A{X} переменной отношения A по атрибуту x. Обозначим один из кортежей этой проекции через (x). Тогда должно быть очевидно, что предикатом для этой проекции, по сути, будет следующий предикат: "Для всех таких x существует такое значение y из домена значений атрибута Y, что кортеж (x, y) удовлетворяет предикату PA". Например, рассмотрим проекцию переменной отношения S по атрибутам S#, SNAME и CITY. Для каждого кортежа (s, n, c), который входит в данную проекцию, существует значение статуса t, такое что кортеж (s,n,t,c) удовлетворяет предикату переменной отношения S.

Ниже приведены правила обновления проекцииA{x}.

- **Правило INSERT.** Пусть (x) — кортеж, который должен быть вставлен, и пусть y — значение, выбираемое по умолчанию из области значений атрибута Y (было бы ошибкой, если бы такое применяемое по умолчанию значение не существовало, т.е. для атрибута Y применялось бы правило "значения по умолчанию запрещены"). Тогда кортеж (x, y), который должен удовлетворять предикату PA, будет вставлен в переменную отношения A.

Примечание. Для потенциальных ключей обычно (но не всегда) значения по умолчанию не предусмотрены (глава 19), поэтому представление, которое содержит не все потенциальные ключи исходной переменной отношения, обычно не допускает выполнения операций вставки.

- **Правило DELETE.** Из переменной отношения A будут удалены все кортежи, у которых значение атрибута x совпадает со значением атрибута кортежа, удаляемого из представления A {X}.

  Примечание. На практике желательно, чтобы множество атрибутов X включало хотя бы один потенциальный ключ переменной отношения A, для того чтобы кортежу, удаляемому из проекции A{X}, соответствовал единственный кортеж в переменной отношения A. Тем не менее, нет никаких резонных оснований превращать это пожелание в жесткое требование. Аналогичная оговорка относится и к операции UPDATE (см. ниже).

- **Правило UPDATE.** Пусть (x) — это кортеж, который необходимо обновить, а (x') — это обновленная версия кортежа (x). Допустим, что a — это кортеж переменной отношения A, имеющий те же значения атрибутов x, что и кортеж X, а значения атрибутов множества Y в кортеже a равны y. Тогда вначале из переменной отношения A будут удалены все кортежи a, отвечающие указанным выше требованиям, причем без выполнения активизируемых процедур и проверки предиката отношения. После этого для каждого определенного выше значения y в переменную отношения A будет вставлен кортеж (x' , y), если он удовлетворяет предикату PA.

  Примечание. Именно в этом определении присутствуют те "небольшие усовершенствования", касающиеся операции проекции, которые упоминались в правиле 5 обновления в подразделе "Принципы создания механизма обновления представлений" (который находится в начале текущего раздела). Обратите внимание, что на последнем этапе выполнения правила для операции UPDATE (этапе вставки) в каждом вставляемом кортеже восстанавливаются предыдущие значения атрибутов Y *вместо* обычной замены значениями, используемыми по умолчанию, как это имеет место при выполнении самостоятельных операций INSERT.

Примеры. Пусть представление SC определено с помощью следующего выражения.

SC { S#, CITY }

На рис. 10.5 показан пример содержимого этого представления.

SC

S#	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

Рис. 10.5. Представление SC (значения, взятые для примера)

- Попытка вставить в представление SC кортеж (S6, Athens) будет успешной. В результате этой операции в переменную отношения s будет вставлен кортеж *(S6, n, t,* Athens), где л и *t* являются значениями, которые используются по умолчанию для атрибутов SNAME и STATUS, соответственно.
- Попытка вставить в представление SC кортеж (S1, Athens) окончится неудачей, так как этот кортеж не удовлетворяет предикату переменной отношения S (а значит, и представления SC). В частности, вставка этого кортежа нарушает требование уникальности потенциального ключа {S#}.
- Попытка удалить из представления SC кортеж (s1,London) будет успешной. Кортеж поставщика с номером S1 будет удален из переменной отношения S.
- Попытка обновить кортеж (S1, London) представления SC с преобразованием в (S1, Athens) будет успешной. Это приведет к обновлению в переменной отношения S кортежа (S1, Smith, 20, London) с преобразованием в кортеж (S1, Smith, 20, Athens), но *не* в кортеж (S1,*n, t,* Athens), где *n* и t — соответствующие значения по умолчанию. Обязательно обратите на это внимание.
- Попытка обновить в представлении SC тот же кортеж (SI, London) с преобразованием в (S2, London) окончится неудачей (объясните, почему именно).

Оставляем в качестве упражнения для читателей анализ случая, когда проекция не включает потенциальных ключей исходной переменной отношения (таковой является, например, проекция переменной отношения s по атрибутам STATUS и CITY).

Операция расширения

Пусть представление V создано с помощью следующего определяющего выражения.

EXTEND A ADD *exp* AS *X*

Как обычно, предполагается, что предикатом переменной отношения А является PA. Тогда предикат PE представления V будет иметь такой вид, как показано ниже.

«PA (a) AND e. *X* = *exp* (a)

Здесь e — это кортеж представления V, а a — кортеж, который остается после удаления компонента *X* кортежа e (т.е. a — это проекция кортежа e по всем атрибутам переменной отношения A). На немного формализованном обычном языке эту операцию можно описать следующим образом.

Каждый кортеж e *в переменной отношения, полученной в результате операции расширения, имеет следующие свойства: во-первых, кортеж* a, *создаваемый из кортежа* e *посредством операции проекции, исключающей компонент X, удовлетворяет предикату PA; во-вторых, значение компонента* X *соответствует результату применения выражения* exp *к кортежу* a.

Приведем правила обновления для представлений, определяемых с **помощью операции** расширения.

- **Правило INSERT.** Пусть e — это кортеж, который нужно вставить в переменную отношения. Он должен удовлетворять предикату PE. В переменную отношения A будет вставлен кортеж a, созданный из кортежа e с помощью операции проекции, исключающей компонент *x*.
- **Правило DELETE.** Пусть e — это кортеж, который нужно удалить. Из переменной отношения A будет удален кортеж a, созданный из кортежа e с помощью операции проекции, исключающей компонент *x*.
- **Правило UPDATE.** Пусть e — это кортеж, который нужно обновить, а e ' — обновленная версия кортежа e, причем кортеж e ' должен удовлетворять предикату PE. Сначала из переменной отношения A без выполнения активизируемых действий и проверки предиката этой переменной отношения будет удален кортеж a, который создан из кортежа e с помощью операции проекции, исключающей компонент *X*. Затем в переменную отношения A будет вставлен кортеж a ', который создан из кортежа e ' с помощью операции проекции, исключающей компонента.

Примеры. Пусть представление VPX определено с помощью следующего выражения.

```
EXTEND P ADD ( WEIGHT * 454 ) AS GMWT
```

На рис. 10.6 приведен пример возможных значений этого представления.

VPX

P#	PNAME	COLOR	WEIGHT	CITY	GMWT
P1	Nut	Red	12.0	London	5448.0
P2	Bolt	Green	17.0	Paris	7718.0
P3	Screw	Blue	17.0	Oslo	7718.0
P4	Screw	Red	14.0	London	6356.0
P5	Cam	Blue	12.0	Paris	5448.0
P6	Cog	Red	19.0	London	8626.0

Рис. 10.6. Представление VPX (эти значения приведены в качестве примера)

- Попытка вставить кортеж (P7, Cog, Red, 12, Paris, 5448) будет завершена успешно и приведет к вставке кортежа (P7, Cog, Red, 12, Paris) в переменную отношения P.
- Попытка вставить кортеж (P7,Cog,Red, 12, Paris, 5449) окончится неудачей (объясните, почему).
- Попытка вставить кортеж (P1, Cog, Red, 12, Paris, 5448) окончится неудачей (объясните, почему).
- Попытка удалить кортеж с ключом P1 будет успешной и приведет к удалению кортежа с ключом P1 из переменной отношения P.
- Попытка обновить кортеж с ключом P1 путем преобразования его в (Pl,Nut, Red, 10, Paris, 4540) будет успешной и приведет к замене кортежа (P1,Nut, Red, 12, London) в переменной отношения P кортежем (Pl,Nut, Red, 10, Paris).
- Закончится неудачей попытка обновить тот же кортеж посредством замены номера детали номером P2 (без изменения остальных атрибутов) или попытка его приведения к виду, в котором значение атрибута GMWT не равно значению атрибута WEIGHT, умноженному на 454 (в каждом случае укажите причину неудачи).

Операция соединения

В большинстве рассматривавшихся ранее трактовок проблемы обновления представлений (включая трактовки, изложенные в нескольких предыдущих изданиях этой книги и в других книгах автора) утверждалось, что способность (или неспособность) каждого конкретного соединения к обновлению зависит (по крайней мере, частично) от того, принадлежит ли соединение к типу "один к одному", "один ко многим" или "многие ко многим". Но в отличие от всех предыдущих трактовок проблемы обновления подобных представлений, в этой книге автор утверждает, что результат операции соединения *всегда* является обновляемым. Более того, для всех трех перечисленных типов соединений правила идентичны и, в целом, вполне очевидны. Справедливость этого утверждения, на первый взгляд кажущегося удивительным, подкрепляется новым видением проблемы, ставшим возможным благодаря принятию некоторого **золотого правила,** что мы и постараемся сейчас пояснить.

В общем случае назначение функции поддержки представлений всегда состояло в стремлении стереть, насколько возможно, разницу между представлениями и базовыми переменными отношения. Тем не менее:

- обычно предполагалось (неявно), что отдельный кортеж базовой переменной отношения всегда можно обновить независимо от всех остальных кортежей этой базовой переменной отношения;
- в то же время утверждалось (явно), что обновить отдельный кортеж представления независимо от всех остальных кортежей этого представления не всегда возможно.

Например, в [12.2] Кодд показал, что из некоторого соединения невозможно удалить только один кортеж, так как это приведет к получению отношения, которое "больше не является соединением каких-либо двух отношений" (это, в свою очередь, означает, что результат, возможно, не будет удовлетворять предикату переменной отношения для представления). Поэтому по традиции сформировался такой подход к подобным операциям обновления представлений, что от них стали вообще отказываться ввиду невозможности сделать эти операции полностью идентичными обновлениям базовых переменных отношения.

Наш подход значительно отличается от изложенного выше. А именно, мы признаем тот факт, что даже в базовых переменных отношения не всегда возможно обновить отдельный кортеж независимо от остальных кортежей. Поэтому мы принимаем как допустимые те операции обновления представлений, которые по сложившейся традиции до сих пор не рассматривались, и даем интерпретацию этих операций в виде четкого и логически корректного способа обновления исходных переменных отношения. Более того, мы считаем эти операции допустимыми, полностью признавая тот факт, что обновление исходных переменных отношения может вызывать побочные эффекты, которые отразятся на представлении. *Однако такие побочные эффекты неизбежны и необходимы, так как в противном случае возникает опасность, что представление перестанет удовлетворять своему предикату.*

Закончив эту вступительную часть, перейдем к конкретному обсуждению проблемы. Прежде всего, определим необходимые термины. После этого приведем *правила обновления представлений* на основе соединений. Затем последовательно рассмотрим применение этих правил для каждого из трех типов соединений ("один к одному", "один ко многим", "многие ко многим").

Рассмотрим соединение J = A JOIN B. Здесь (как и в разделе 7.4 главы 7) переменные отношения A, в и J, соответственно, имеют заголовки {x, Y}, {Y, Z} и {X, Y, Z}. Пусть PA и PB — это предикаты переменных отношения A и B. Тогда предикат PJ представления J будет иметь следующий вид.

PA (a) AND PB (Ь)

Здесь для каждого заданного кортежа соединения *j* кортеж а является "относящейся к А частью" кортежа j (т.е. а — это кортеж, порождаемый из кортежа *j* посредством операции проекции, исключающей компонент Z), а Ъ является "относящейся к в частью" кортежа *j* (т.е. кортежем, порождаемым из кортежа *j* посредством операции проекции, исключающей компонент *x).* Другими словами, каждый кортеж в соединении является таковым, что часть его, относящаяся к А, удовлетворяет предикату PA, а часть, относящаяся к в, удовлетворяет предикату PB. Например, предикат для соединения переменных отношения s и SP по атрибуту s# можно сформулировать следующим образом.

Каждый кортеж (s,n, t,c,p,q) в соединении является таковым, что кортеж (s, n, t , c) удовлетворяет предикату переменной отношения S, а кортеж (s,p, q) удовлетворяет предикату переменной отношения SP.

Приведем правила обновления представлений вида J = A JOIN в.

- **Правило INSERT.** Новый кортеж j должен удовлетворять предикату PJ. Если относящаяся к А часть[5] кортежа *j* не входит в переменную отношения А, то она вставляется в А. Если относящаяся к в часть кортежа *j* не присутствует в переменной отношения в, то она вставляется в в.
- **Правило DELETE.** Относящаяся к А часть удаляемого кортежа удаляется из пере менной отношения А, а относящаяся к в часть удаляемого кортежа удаляется из переменной отношения в.
- **Правило UPDATE.** Обновляемый кортеж должен быть таким, чтобы его обновлен ная версия удовлетворяла предикату P J. Относящаяся к А часть этого кортежа уда ляется из переменной отношения А без выполнения каких-либо активизируемых действий и проверки предиката, а относящаяся к **в** часть кортежа удаляется из пе ременной отношения в, опять же, без выполнения каких-либо активизируемых действий и проверки предиката. Если после этого относящаяся к А часть обнов ленного кортежа все еще отсутствует в переменной отношения А, то относящаяся к А часть вставляется в А. Если относящаяся к **в** часть обновленного кортежа от сутствует в переменной отношения в, то она вставляется в в.

.Теперь проверим возможность применения сформулированных правил ко всем трем существующим типам соединений.

---

[5] Отметим, что операция INSERT может иметь побочный эффект, вследствие которого относящаяся к в часть кортежа *j* будет вставлена в переменную-отношение в, как в случае с представлениями, основанными на объединении, пересечении и разности (см. выше). Аналогичное замечание касается правил для представлений, основанных на операциях DELETE и UPDATE. Для краткости мы не будем подробно рассматривать в каждом случае эту возможность.

Случай 1: соединения типа "один к одному"

Прежде всего, отметим, что в данном случае термин "один к одному" можно заменить более точным термином "(нуль или один) к (нулю или одному)". Другими словами, имеет место ограничение целостности, гарантирующее, что для каждого кортежа переменной отношения A будет существовать не больше одного соответствующего кортежа переменной отношения в, и наоборот. А это означает, что множество атрибутов Y, по которому выполняется соединение, должно быть суперключом для обеих переменных отношения, А И В.

Примеры

- В качестве первого примера читателю предлагается рассмотреть результат приме нения приведенных выше правил к соединению переменной отношения с данны ми о поставщиках S с самой собой (только) по атрибуту номера поставщикаs#.
- Во втором примере предположим, что в базе данных поставщиков и деталей со держится еще одна переменная отношения, SR, с атрибутами S# и REST, где атри бут s# идентифицирует поставщика, а атрибут REST содержит данные о его люби мом ресторане. Предположим, что в переменной отношения SR представлены не все поставщики, сведения о которых имеются в переменной отношения S. Читате лю предлагается рассмотреть результат применения правил обновления соедине ния к представлению, определенному как соединение s JOIN SR. Что изменится, *если* поставщик будет представлен в переменной отношения SR, но не будет пред ставлен в переменной отношения S?

Случай 2: соединения типа "один ко многим"

Здесь термин "один ко многим" можно заменить более точным термином "(нуль или один) к (нулю или многим)". Другими словами, имеет место ограничение целостности, гарантирующее, что для каждого кортежа из переменной отношения в существует не больше одного соответствующего кортежа в переменной отношения А. Обычно это означает, что множество атрибутов Y, по которому выполняется операция соединения, должно содержать подмножество (скажем, к), такое что к является потенциальным ключом для переменной отношения А и соответствующим внешним ключом для переменной отношения в.

Примечание. Если выполняются сформулированные выше требования, то выражение "нуль или один" можно заменить выражением "точно один".

Примеры. Пусть представление SSP определено следующим выражением.

S JOIN SP

(Безусловно, это — соединение типа "внешний ключ с соответствующим потенциальным ключом".) Пример содержимого этого представления показан на рис. 10.7.

- Попытка вставить в представление SSP кортеж (S4,Clark,20,London,Рб,100) будет завершена успешно и приведет к вставке кортежа (S4, Рб, 100) в перемен ную отношения SP (в результате чего новый кортеж появится и в представлении).
- Попытка вставить в представление SSP кортеж (S5, Adams ,30, Athens, Рб, 100) будет завершена успешно и приведет к вставке кортежа (S5, Рб, 100) в перемен ную отношения SP (следовательно, новый кортеж добавится и к представлению).

- Попытка вставить в представление SSP кортеж (S6, Green, 20, London, P6, 100) будет завершена успешно и приведет к вставке кортежа (S6, Green, 20, London) в переменную отношения S и к вставке кортежа (S6, P6,100) в переменную отношения SP (в результате чего новый кортеж появится и в представлении). *Примечание.* Примем на время такое предположение, что возможно существование кортежей SP без соответствующего кортежа s. Кроме того, допустим, что переменная отношения SP уже включает некоторые кортежи с номером поставщика S6, но в их число не входят кортежи с номером поставщика S6 и номером детали P1. При таких условиях операция INSERT, которая применялась в описанном выше примере, будет иметь побочный эффект, связанный со вставкой в представление некоторых дополнительных кортежей, а именно кортежей, полученных в результате соединения кортежа (S6, Green, 20, London) с каждым из тех ранее существовавших кортежей SP, которые относятся к поставщику S6.

SSP

S#	SNAME	STATUS	CITY	P#	QTY
S1	Smith	20	London	P1	300
S1	Smith	20	London	P2	200
S1	Smith	20	London	P3	400
S1	Smith	20	London	P4	200
S1	Smith	20	London	P5	100
S1	Smith	20	London	P6	100
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
S3	Blake	30	Paris	P2	200
S4	Clark	20	London	P2	200
S4	Clark	20	London	P4	300
S4	Clark	20	London	P5	400

Рис. 10.7. Представление SSP (значения для примера)

- Попытка вставить в представление SSP кортеж (S4,clark, 20, Athens, Рб, 100) окончится неудачей (объясните, почему).
- Попытка вставить в представление SSP кортеж (SI, Smith, 20, London, PI, 400) окончится неудачей (объясните, почему).
- Попытка удалить из представления SSP кортеж (S3,Blake, 30, Paris, P2, 200) завершится успешно и приведет к удалению кортежа (S3, Blake, 30, Paris) из переменной отношения S и кортежа (S3, P2, 2 00) из переменной отношения SP.
- Попытка удалить из представления SSP кортеж (S1, Smith, 20, London, P1, 300) завершится "успешно" (с определенными оговорками; см. приведенное ниже примечание) и приведет к удалению кортежа (S1, Smith, 20, London) из переменной отношения S и кортежа (S1,P1,300) из переменной отношения SP.

  Примечание. В действительности, общий результат приведенной операции удаления будет зависеть от установленного правила удаления внешнего ключа, связывающего отношения с данными о поставках и поставщиках. Если указана опция NO ACTION или RESTRICT, то данная операция удаления окончится неудачей. Если указана опция CASCADE, то операция удаления будет иметь побочный

эффект, который выразится в дополнительном удалении из переменной отношения SP всех остальных кортежей (а значит, и кортежей представления SSP), содержащих данные о поставщике с номером S1.

- Попытка обновить в представлении SSP кортеж (S1, Smith, 20,London, P1, 300) с преобразованием в (S1, Smith, 20, London, P1, 400) будет успешно завершена и приведет к обновлению кортежа (S1, P1, 300) в переменной отношения SP с преобразованием в (S1,P1,400).
- Попытка обновить в представлении SSP кортеж (SI, Smith, 20, London, P1, 300) с преобразованием в (S1, Smith, 20, Athens, PI, 400) будет успешно завершена
  и приведет к обновлению кортежа (S1, Smith, 20, London) в переменной отношения S с преобразованием в (S1,Smith,20,Athens) и кортежа (S1,P1,300) в переменной отношения SP с преобразованием в (S1, P1, 400).
- Попытка обновить в представлении SSP кортеж (S1, Smith, 20, London, P1, 300) с преобразованием в (S6, Smith, 20, London, P1, 300) будет завершена "успешно" (с определенной оговоркой; см. приведенное ниже примечание) и приведет к обновлению кортежа (S1, Smith, 20, London) в переменной отношения S с преобразованием в (S6, Smith, 20, London) и кортежа (S1,P1,300) в переменной отношения SPC преобразованием в (S6, P1, 300).

  Примечание. В действительности, общий эффект приведенной выше операции обновления будет зависеть от установленного правила обновления внешнего ключа, связывающего отношения с данными о поставках и поставщиках. Подробный анализ этой ситуации оставляем читателю в качестве упражнения.

Случай 3: соединения типа "многие ко многим"

Здесь термин "многие ко многим" следовало бы заменить более точным термином "(нуль или многие) к (нулю или многим)". Другими словами, не существует стандартного ограничения целостности, которое могло бы дать гарантии, аналогичные тем, которые мы имели в случае соединений, рассматриваемых в первом или втором случае.

Примеры. Предположим, что обсуждаемое представление определено с помощью следующего выражения.

```
S JOIN P
```

Это соединение переменных отношения S и P по атрибуту CITY имеет тип "многие ко многим". Пример содержимого данного представления показан на рис. 10.8.

- Вставка в представление кортежа (S7, Bruce, 15, Oslo, P8,Wheel, White, 25) будет успешно завершена и приведет к вставке кортежа (S7, Bruce, 15, Oslo) в переменную отношения S и кортежа (P8, Wheel, White, 25, Oslo) в переменную отношения P (в результате указанный кортеж появится в представлении).
- Вставка в представление кортежа (SI, Smith, 20, London, P7, Washer, Red, 5) будет успешно завершена и приведет к вставке кортежа (P7, Washer, Red, 5, London) в переменную отношения P (в результате чего в представление попадут два кортежа — кортеж (S1, Smith, 20, London, P7,Washer, Red, 5), заданный в операции вставки, и кортеж (S4, Clark, 20, London, P7,Washer,Red, 5)).

S#	SNAME	STATUS	CITY	P#	PNAME	COLOR	WEIGHT
S1	Smith	20	London	P1	Nut	Red	12.0
S1	Smith	20	London	P4	Screw	Red	14.0
S1	Smith	20	London	P6	Cog	Red	19.0
S2	Jones	10	Paris	P2	Bolt	Green	17.0
S2	Jones	10	Paris	P5	Cam	Blue	12.0
S3	Blake	30	Paris	P2	Bolt	Green	17.0
S3	Blake	30	Paris	P5	Cam	Blue	12.0
S4	Clark	20	London	P1	Nut	Red	12.0
S4	Clark	20	London	P4	Screw	Red	14.0
S4	Clark	20	London	P6	Cog	Red	19.0

Рис. 10.8. Результат соединения переменных отношения S и P по атрибуту CITY

- Вставка в представление кортежа (S6, Green, *20,* London, P7, Washer, Red, 5) будет успешно завершена и приведет к вставке кортежа (S6, Green, 20, London) в переменную отношения S и кортежа (P7,Washer,Red, 5, London) в переменную отношения p (в результате в представление будет добавлено *шесть* новых кортежей).
- Удаление из представления кортежа (SI, Smith, 20, London, PI,Nut, Red, 12) будет успешно завершено и приведет к удалению кортежа (SI, Smith, 20, London) из переменной отношения S и кортежа (PI,Nut,Red, 12,London) из переменной отношения P (в результате из представления будут удалены *четыре* кортежа).

Предлагаем читателю подготовить дополнительные примеры самостоятельно в качестве упражнения.

Прочие операции

В этом разделе вопросы использования в представлениях остальных операций реляционной алгебры рассматриваются лишь кратко. Прежде всего, отметим, что операции тета-соединения, полусоединения, полуразности и деления не относятся к типу элементарных операций, поэтому правила для них могут быть сформированы на основании правил для тех операций, в терминах которых они определены. А сведения о других операциях приведены ниже.

- **Переименование.** Тривиальный случай.
- **Декартово произведение.** Как уже отмечалось в разделе 7.4 главы 7, декартово произведение является частным случаем естественного соединения (операция A JOIN B вырождается в операцию A TIMES B, если отношения A и **в** не имеют общих атрибутов). Поэтому правила обновления для операции декартова произведения (A TIMES в) являются частным случаем правил обновления для операции соединения (конечно, как и правила обновления для операции пересечения A INTERSECT B).
- **Формирование** итогов. Операция формирования итогов (SUMMARIZE) тоже не является элементарной и определяется в терминах операции расширения. Поэтому правила обновления для операции формирования итогов являются производными от правил обновления для операции расширения.

  Примечание. Общепринятое мнение, что на практике оканчивается неудачей основная часть попыток выполнить обновление представлений, которые определены с

использованием операции формирования итогов, недалеко от истины. Однако причина заключается не в том, что такие представления *по самой своей сути* не являются обновляемыми. Попытки их обновления завершаются неудачно лишь из-за противоречий с некоторыми установленными ограничениями целостности. Например, пусть для определения представления используется следующее выражение.

```
SUMMARIZE SP BY { S# } ADD SUM ( QTY ) AS TOTQTY
```

Тогда попытка удаления кортежа, скажем, для поставщика с номером S1, будет вполне успешной. Однако попытка обновить кортеж, скажем, (S4, 900), с преобразованием в (S4, 800) окончится неудачей, поскольку эта попытка нарушает ограничение, согласно которому значение атрибута TOTQTY должно быть равным сумме всех соответствующих отдельных значений QTY. Попытка вставить кортеж (S5, 0) тоже приведет к ошибке, но уже по другой причине (укажите, по какой именно).

- **Группирование и разгруппирование.** Все замечания, которые были сделаны для операции формирования итогов, справедливы и для этих операций [3.3].
- **Транзитивное замыкание Tclose.** И в данном случае справедливы почти такие же замечания.

10.5. СНИМКИ (НЕБОЛЬШОЕ ОТКЛОНЕНИЕ ОТ ОСНОВНОЙ ТЕМЫ)

Здесь будет уместно, несколько отклонившись от основной темы, обсудить понятие **снимков** (snapshot) [10.1]. Снимки в действительности имеют много общего с представлениями, но не следует путать эти понятия. Как и представления, снимки — это производные переменные отношения, но, в отличие от представлений, снимки реальны, а не виртуальны, т.е. снимки представлены в базе данных не только в виде собственных определений в терминах других переменных отношения, но и (по крайней мере, концептуально) в виде собственной материализованной копии данных, например, как показано ниже.

```
VAR P2SC SNAPSHOT
    ( ( S JOIN SP ) WHERE P# = P# ('P2') ) { S#,
    CITY } REFRESH EVERY DAY ;
```

Определение снимка во многом подобно выполнению запроса, за исключением следующего.

1. Результат выполнения этого запроса хранится в базе данных под указанным именем (в приведенном выше примере это P2SC) как *переменная отношения,* доступ к которой разрешен только для *чтения* (не считая операции периодического обновления; см. пункт 2).
2. Периодически (в нашем примере — раз в сутки, что устанавливается опцией EVERY DAY) содержание снимка обновляется, т.е. текущие данные аннулируются и запрос выполняется повторно, после чего полученный результат запроса запи сывается в качестве нового значения снимка.

Таким образом, снимок P2SC всегда представляет состояние данных, но на тот момент времени, который отстоит от текущего не больше чем на 24 часа (предлагаем читателю определить самостоятельно, каким в таком случае должен быть предикат данного отношения).

Суть самой идеи снимков состоит в том, что для многих приложений (возможно даже для большинства) допустимо или даже необходимо использовать для обработки данные в том состоянии, в котором они находились в определенный момент времени. В частности, к этой категории приложений относятся многие приложения для создания отчетов и ведения бухгалтерского учета. Подобные приложения обычно требуют фиксации состояния данных в установленное время (например, на конец периода отчетности), и концепция снимков позволяет выполнить такую фиксацию, не влияя на работу других транзакций, обновляющих рассматриваемые данные в режиме реального времени (т.е. обновляющих *реальные данные).* Аналогичным образом может потребоваться зафиксировать состояние большого объема данных, которые используются для выполнения сложного запроса или приложения, не требующего модификации исходных данных, опять же, чтобы избежать блокирования обновления данных на время их выполнения или изменения этих данных.

Примечание. Эта идея становится еще привлекательнее в среде распределенных баз данных или приложений поддержки принятия решений (подробности приводятся, соответственно, в главах 21 и 22). Отметим также, что снимки представляют важный частный случай *контролируемой избыточности* (см. главу 1), а процедура *обновления снимка* — это соответствующий процесс *распространения обновления* (снова см. главу 1).

В общем случае определение снимка имеет следующий синтаксис.

```
VAR <relvar name> SNAPSHOT <relation
    exp> <candidate key def list>
    REFRESH EVERY <now and then> ;
```

В этом определении для указания периода обновления снимка используется параметр *<now and then>,* который может принимать, например, следующие значения: MONTH (Месяц), WEEK (Неделя), HOUR (Час), *n* MINUTES *(n* минут), MONDAY (Понедельник), WEEKDAY (День недели) и т.п. Следует особо отметить, что для поддержки постоянной синхронизации снимка с одной или несколькими переменными отношения, на основании которых он был создан, может использоваться спецификация в форме REFRESH [ON] EVERY UPDATE.

Ниже приведен синтаксис оператора DROP, применяемого для удаления определения снимка.

DROP VAR *<relvar name>* ;

Здесь параметр *<relvar name>* задает имя удаляемого снимка.

Примечание. Подразумевается, что операция удаления снимка завершится неудачно, если какая-либо переменная отношения в данный момент ссылается на удаляемый снимок. Альтернативным решением может быть расширение приведенного выше определения снимка за счет включения опций RESTRICT и CASCADE. Здесь мы не будем обсуждать эту возможность.

Примечание, касающееся терминологии. Ко времени подготовки первого издания данной книги снимки были известны не под их современным названием, а именовались (фактически почти исключительно) материализованными представлениями[6] (см. раздел "Список литературы" в главе 22). Однако этот термин является крайне неудачным, и, по

[6] Некоторые авторы (но не все) применяют термин "материализованное представление" исключительно для обозначения снимков, в отношении которых можно гарантировать, что они всегда будут оставаться актуальными (т.е. для создания которых применяется оператор REFRESH ON EVERY UPDATE).

мнению автора, необходимо настойчиво бороться с его употреблением. Снимки — не представления. Весь смысл представлений состоит в том, что они не являются материализованными, по крайней мере, если речь идет о модели. (Вопрос о том, должны ли они быть действительно материализованы незаметно для пользователя или нет, рассматривается на уровне реализации и не имеет никакого отношения к модели.) Иными словами, если речь идет о модели, то термины, из которых состоит выражение *материализованное представление,* противоречат друг другу. Тем не менее, сам термин *материализованное представление* (по вполне очевидным причинам) применяется настолько широко, что под неуточненным термином *представление* чаще всего подразумевается именно *материализованное представление*! Поэтому автор лишился возможности использовать качественный термин, когда нужно упомянуть о представлении в его первоначальном смысле. Безусловно, ему приходится идти на серьезный риск быть неправильно понятым, когда он использует для этой цели неуточненный термин *представление.* Но в этой книге автор все же решил принять это рискованное решение. Итак, укажем со всей определенностью, что в этой книге термин *материализованное представление* вообще не используется (кроме как в цитатах, взятых из других источников), термин *снимок* закрепляется за рассматриваемой концепцией, а неуточненный термин *представление* всегда используется в его первоначальном реляционном смысле.

10.6. ПОДДЕРЖКА ПРЕДСТАВЛЕНИЙ В ЯЗЫКЕ SQL

В этом разделе будут рассмотрены средства поддержки представлений, существующие в языке SQL (ко времени написания данной книги в языке SQL не была предусмотрена поддержка снимков). Прежде всего, рассмотрим синтаксис оператора создания представления **CREATE VIEW,** как показано ниже. (Здесь для краткости не рассматривается целый ряд опций и альтернатив, в частности, возможность определить представление как имеющее некоторый структурированный тип с помощью ключевого слова"OF".)

```
CREATE VIEW <view name> AS <table exp>
     [ WITH [ <qualifier> ] CHECK OPTION ] ;
```

Пояснения

1. Значение параметра *<table exp>* представляет собой определение представления.
2. Конструкция WITH CHECK OPTION, если она указана, означает, что операции вставки (INSERT) и обновления (UPDATE) для данного представлении будут отменены в случае нарушения ограничений целостности, указанных в определении представления. Поэтому необходимо учитывать, что подобные операции будут окачиваться неудачей лишь в *том* случае, когда конструкция WITH CHECK OPTION задана явно, т.е. по умолчанию любые операции вставки и обновления кортежей будут завершаться *успешно.* На основании выводов, сделанных в разделе 10.4, можно заключить, что такой способ организации работы логически не оправдан. Поэтому настоятельно рекомендуется на практике *всегда* указывать опцию WITH CHECK OPTION в определениях любых создаваемых представлений[7] [10.5].

[7] Безусловно, речь идет только об обновляемых представлениях. Как мы убедимся позже, в языке SQL представления часто не являются обновляемыми, и наличие опции WITH CHECK OPTION в таких случаях недопустимо в соответствии с требованиями языка SQL.

Примеры

```
1. CREATE VIEW GOOD_SUPPLIER
      AS SELECT S.S#, S.STATUS, S.CITY
         FROM   S
         WHERE  S.STATUS > 15
      WITH CHECK OPTION ;
```

```
2. CREATE VIEW REDPART
      AS SELECT P.P#, P.NAME, P.WEIGHT AS WT, P.CITY
         FROM   P
         WHERE  P.COLOR =
      'Red' WITH CHECK OPTION
      ;
```

```
3 . CREATE VIEW PQ
      AS SELECT P.P#, ( SELECT SUM (
                        SP.QTY ) FROM   SP
                        WHERE SP.P# = P.P# ) AS
         TOTQTY FROM P ;
```

В языке SQL это представление не рассматривается как обновляемое, поэтому конструкция WITH CHECK OPTION должна быть опущена.

```
4. CREATE VIEW CITY_PAIR
      AS SELECT DISTINCT S.CITY AS SCITY, P.CITY AS
         PCITY FROM S, SP, P WHERE S.S# = SP.S# AND
         SP.P# = P.P# ;
```

В языке SQL это представление также не рассматривается как обновляемое, поэтому конструкция WITH CHECK OPTION должна быть опущена.

```
5. CREATE VIEW HEAVY_REDPART
      AS SELECT RP.P#, RP.PNAME, RP.WT, RP.CITY
         FROM REDPART AS RP
         WHERE RP.WT > 12.0
      WITH CHECK OPTION ;
```

Существующее представление может быть удалено с помощью оператора DROP VIEW, синтаксис которого приведен ниже.

```
DROP VIEW <view name> <behavior> ;
```

В этом операторе (как обычно) параметр *<behavior>,* который указывает способ удаления, может принимать значения RESTRICT И CASCADE. ЕСЛИ задана опция RESTRICT, а удаляемое представление где-то используется (например, в определениях других представлений или в ограничениях целостности), то данная операция DROP окончится неудачей. Если же указана опция CASCADE, то выполнение данной операции DROP завершится успешно и повлечет за собой неявное применение операторов DROP ... CASCADE ко всем компонентам базы данных, в которых в настоящее время используется данное представление.

Выборка данных из представлений

Как было отмечено в разделе 10.3, текущая версия стандарта SQL гарантирует правильное выполнение всех операций выборки данных из любых представлений. К сожалению, этого нельзя сказать о некоторых современных программных продуктах, а также о версиях стандарта, предшествующих SQL: 1992.

Обновление данных в представлениях

В языке SQL поддержка средств обновления представлений остается ограниченной. Причину такого положения дел чрезвычайно трудно обосновать какими-то разумными доводами! Фактически, в этой области стандарт языка SQL остается еще более "непробиваемым", чем обычно[8]. Ниже приведена типичная выдержка из этого стандарта (лишь немного отредактированная для согласования с текущим контекстом).

Выражение запроса <query expression^ *QE1, является обновляемым тогда и только тогда, когда каждое выражение запроса* <query expression> *или каждая спецификация запроса* «juery specification^ *QE2, которая просто содержится в QE1, соответствует указанным ниже требованиям.*

а) QE1 включает QE2 без промежуточных выражений, отличных от запросов на соединение, *<non join query expression>,* которые определяют собой КОНСТРУКЦИИ[9] UNION DISTINCT, EXCEPT ALL ИЛИ EXCEPT DISTINCT.

б) Если QE1 просто содержит выражение, отличное от запроса на соединение (Non Join Query Expression — NJQE), который определяет конструкцию UNION ALL, то имеет место следующее:

- выражение NJQE непосредственно содержит выражение запроса *<query expression>,* LO, и терм запроса <guery *term>,* RO, такие что ни одна таблица L0, в целом лежащая в основе лист-отношения, не является также таблицей R0, в целом лежащей в основе лист-отношения;
- для каждого столбца в выражении NJQE основополагающие столбцы в табли цах, обозначенных, соответственно, как L0 и RO, являются одновременно либо обновляемыми, либо не обновляемыми.

в) QE1 включает QE2 без промежуточных выражений, отличных от запросов на соеди нение, *<non join query expression>,* которые определяют собой конструк цию INTERSECT.

г) QE2 является обновляемым.

Обратите внимание: во-первых, приведенное выше правило представляет собой только одно из многих правил, которые должны рассматриваться в их сочетании при определении того, является ли обновляемым данное представление, во-вторых, все рассматриваемые правила не приведены в одном месте, а разбросаны по многим разным частям документа, и, в третьих, во всех этих правилах упоминается множество разнообразных дополнительных концепций и конструкций, таких как *обновляемый столбец* (updatable column), *таблица, в целом лежащая в основе лист-отношения* (leaf generally underlying table), *выражение, отличное от запросов на соединение* (non join query term), которые, в свою очередь, определены совсем в других частях документа.

---

[8] Цитата из [10.11]: "Стандарт SQL был и продолжает оставаться барьером на пути к разработке (не говоря уже об осуществлении) общих методов обновления представлений".

[9] Мы не упоминали об этом в главе 8, но в стандарте SQL: 1999 дополнительно предусмотрена воз можность явно задавать спецификатор DISTINCT вместо ALL в сочетании с ключевыми словами UNION, INTERSECT и EXCEPT. Аналогичным образом, может быть также явно задан спецификатор ALL вместо DISTINCT в сочетании с ключевым словом SELECT. Но необходимо учитывать, что по умолчанию с ключевыми словами UNION, INTERSECT И EXCEPT применяется DISTINCT, с ключевым СЛОВОМ SELECT по умолчанию применяется ALL.

По этим соображениям автор даже не пытается привести здесь точное определение представлений, которые являются обновляемыми в языке SQL. Но неформально можно отметить, что в языке SQL как обновляемые рассматриваются представления, перечисленные ниже.

1. Представления, которые определены как сокращение и/или проекция одной базо вой таблицы.
2. Представления, определенные как соединение "один ко одному" или "один ко многим" двух базовых таблиц[10] (в случае соединения "один ко многим" обновляе мой является только сторона "многие").
3. Представления, определенные выражениями UNION ALL или INTERSECT с двумя отдельными базовыми таблицами.
4. Некоторые комбинации вариантов, которые относятся к описанным выше случаям 1 —3.

Мало того, даже эти ограниченные варианты трактуются неправильно из-за того, что в языке SQL недостаточно широко используется такое понятие, как *предикат,* и особенно в связи с тем фактом, что в языке SQL разрешаются дубликаты строк. И вся эта картина усложняется еще больше ввиду того, что в языке SQL распознаются четыре отдельных случая, когда данное представление может быть обновляемым, потенциально обновляемым, просто обновляемым или применимым для вставки[11] (где понятие *обновляемый* относится к операторам UPDATE и DELETE, а понятие *применимый для вставки* — к операторам INSERT, и представление не может быть применимым для вставки, если оно не является обновляемым). Но что касается описанного выше случая 1, то необходимо сформулировать соответствующие требования немного точнее. А именно, представление SQL безусловно является обновляемым, если удовлетворяются все перечисленные ниже восемь условий.

1. Табличное выражение, определяющее представление, должно быть простым вы ражением выборки, т.е. оно не должно содержать ни одного из ключевых слов JOIN, UNION, INTERSECT ИЛИ EXCEPT.
2. Предложение SELECT В выражении выборки не должно непосредственно содер жать ключевое слово DISTINCT.
3. Каждый выбираемый элемент в предложении SELECT (после любых требуемых расширений этого списка, заданных шаблоном "звездочка", *) должен быть име нем столбца (возможно, уточненным и при необходимости сопровождаемым фра зой AS), представляющим простую ссылку на столбец исходной таблицы (см. п. 5),

---

[0] В связи с соединениями "один ко одному" отметим еще одну странность. В стандарте SQL содер жится вполне резонное требование, чтобы обновление таких соединений осуществлялось по принципу "все или ничего". Но из этого требования (как и из требования, согласно которому все обновления в целом должны осуществляться по принципу "все или ничего", даже если они предусматривают выпол нение таких действий по поддержке ссылочной целостности, как каскадное удаление) следует, что сис тема, по крайней мере, на уровне реализации, должна поддерживать своего рода множественное реля ционное присваивание, несмотря на тот факт, что в языке SQL не предусмотрена явная поддержка для любого подобного оператора.

[1] В стандарте SQL эти термины определены формально, но не дано никакого намека на то, каково их интуитивное значение и почему они были выбраны. Обратите внимание на нарушение в этом прин ципов 9 и 10, которые были описаны в подразделе "Направления создания механизма обновления пред ставлений" раздела 10.4.

причем такая ссылка на столбец ни в коем случае не должна появляться больше одного раза.

4. Конструкция FROM в выражении выборки должна содержать точно одну ссылку на таблицу.
5. Данная ссылка на таблицу должна задавать либо базовую таблицу, либо представление, соответствующее требованиям пп. 1—8.

   Примечание. Таблица, заданная с помощью ссылки на таблицу, называется *основополагающей таблицей* для рассматриваемого обновляемого представления (см. п. 3).
6. Заданное выражение выборки не должно включать конструкцию WHERE с подзапросом, в котором содержится конструкция FROM, ссылающаяся на ту же таблицу, что и основная конструкция FROM, указанная в п. 4.
7. Выражение выборки не должно включать предложения GROUP BY.
8. Выражение выборки не должно включать предложения HAVING.

10.7. РЕЗЮМЕ

Представление — это, по сути, именованное реляционное выражение. Представление можно рассматривать как **производную виртуальную переменную отношения.** Операции над представлениями обычно реализуются с помощью процедуры **подстановки,** состоящей в замене ссылки на *имя* представления тем выражением, которое это представление *определяет.* Процедура подстановки работает корректно благодаря **реляционному свойству замкнутости.** Для операций **выборки** процесс подстановки корректно выполняется в 100% случаев (по крайней мере, теоретически, но не обязательно на практике для существующих продуктов). Для операций **обновления** процесс подстановки корректно выполняется также в 100% случаев (опять же, теоретически, но не обязательно на практике). Однако для некоторых представлений (например, представлений, определяемых в терминах операции формирования итогов) попытка обновления обычно приводит к ошибке, поскольку нарушаются установленные в системе ограничения целостности. В этой главе также рассматривался обширный набор **принципов,** которым должна удовлетворять схема обновления. Была подробно рассмотрена работа схемы обновления для представлений, определенных в терминах операций **объединения, пересечения, разности, сокращения, проекции, соединения** и **расширения.** Для каждой из этих операций были описаны соответствующие **правила вывода предиката.**

Также была затронута проблема представлений и **логической независимости от данных.** Существует два аспекта такой независимости: аспект **роста** и аспект **реструктуризации.** Среди других преимуществ представлений можно отметить их способность скрывать данные и, следовательно, обеспечивать определенный уровень **защиты,** а также их способность выступать в роли сокращенной записи и тем самым упрощать работу пользователей. Были рассмотрены два важных принципа — **принцип взаимозаменяемости** (из которого, в частности, следует, что представления должны быть обновляемыми) и **принцип относительности базы данных.**

Отклонившись на некоторое время от основной темы, мы кратко обсудили использование **снимков** (иногда называемых также *материализованными представлениями,* хотя от этого термина следует полностью отказаться). И наконец, в главе кратко были описаны те разделы языка SQL, которые имеют отношение к обсуждаемой теме.

УПРАЖНЕНИЯ

10.1. Определите представление с данными о поставщиках, находящихся в Лондоне.

10.2. Дайте определение представления, содержащего номера поставщиков и номера тех деталей, которые не хранятся в том же городе, где находится поставщик.

10.3. Определите переменную отношения SP, соответствующую базе данных о поставщиках и деталях, как представление переменной отношения SPJ, соответствующей базе данных о поставщиках, деталях и проектах.

10.4. На основе базы данных о поставщиках, деталях и проектах определите представление, включающее данные обо всех проектах (только с такими атрибутами, как номер проекта и название города), детали для которых поставляет поставщик с номером si и в которых используется деталь с номером P1.

10.5. Пусть имеется представление, определенное следующим образом.

```
VAR HEAVYWEIGHT VIEW
   ( ( P RENAME ( WEIGHT AS WT, COLOR AS COL ) )
      WHERE WT > WEIGHT ( 14.0 ) ) { P#, WT, COL } ;
```

Покажите преобразованную форму для каждого из перечисленных ниже выражений и операторов после выполнения процедуры подстановки.

```
а) HEAVYWEIGHT WHERE COL = COLOR ('Green')

б) ( EXTEND HEAVYWEIGHT ADD ( WT + WEIGHT ( 5.3 ) ) AS WTP )
                    { P#, WTP }

В) INSERT HEAVYWEIGHT
   RELATION { TUPLE { P#  P# ('P99'),
                    WT  WEIGHT ( 12.0 ) ,
                     COL COLOR ('Purple') } } ;

Г) DELETE HEAVYWEIGHT WHERE WT < WEIGHT ( 10.0 ) ;

Д) UPDATE HEAVYWEIGHT WHERE WT = WEIGHT ( 18.0
         ) { COL := 'White[1]' } ;
```

10.6. Предположим, что приведенное в упр. 10.5 определение представления HEAVYWEIGHT изменено следующим образом.

```
VAR HEAVYWEIGHT VIEW ( ( ( EXTEND P
ADD ( WEIGHT * 454 ) AS WT )
     RENAME COLOR AS COL ) WHERE WT > WEIGHT ( 6356.0 ) )
                                          { P#, WT, COL } ;
```

(Атрибут WT теперь содержит вес в граммах, а не в фунтах.) Выполните приведенное в упр. 10.5 задание для этого варианта.

10.7. Используя реляционное исчисление, предоставьте аналоги алгебраических определений представлений, приведенных в разделе 10.1.

10.8. Конструкция ORDER BY в определении представления не имеет смысла (несмотря на тот факт, что она разрешена для использования по меньшей мере в одном известном программном продукте!). Объясните, почему она не нужна.

10.9. В главе 9 указывалось, что для представлений иногда желательно иметь возможность объявления потенциальных ключей (или первичного ключа). Объясните, в связи с чем может потребоваться такая возможность.

10.10. Какие расширения системного каталога необходимы для поддержки механизма представлений? Что можно дополнительно сказать по поводу снимков?

10.11. Предположим, что некоторая базовая переменная отношения R была заменена двумя сокращениями, A и B, причем такими, что объединение A UNION B всегда
эквивалентно исходному отношению R, а пересечение A INTERSECT в всегда пусто. Достижима ли в этом случае логическая независимость от данных?

10.12. Если отношения A и B относятся к одному и тому же типу, то пересечение A INTERSECT в эквивалентно соединению A JOIN в (это соединение типа "один к одному", но не в строгом смысле, поскольку в переменной отношения A могут существовать кортежи, для которых нет соответствующих кортежей в переменной отношения в, и наоборот). Являются ли правила обновления представлений, изложенные в разделе 10.4, совместимыми с подобными эквивалентностями, определенными посредством операций пересечения и соединения?

10.13. Кроме того, пересечение A INTERSECT в эквивалентно разности A MINUS (A MINUS в) и разности в MINUS (в MINUS A). Являются ли правила обновления представлений, изложенные в разделе 10.4, совместимыми с подобными эквивалентностями, определенными посредством операций пересечения и разности?

10.14. Один из принципов, изложенных в разделе 10.4, состоит в том, что операции INSERT и DELETE должны быть противоположными одна другой в максимально возможной степени. Соответствуют ли этому принципу изложенные в этом разделе правила обновления представлений, определенных посредством операций объединения, разности и пересечения?

10.15. В разделе 10.2 (при рассмотрении проблемы логической независимости отданных) обсуждалась возможность реструктуризации базы данных о поставщиках и деталях посредством замены базовой переменной отношения s двумя проекциями этой переменной отношения с именами SNC и ST. Там же отмечалось, что подобная реструктуризация не всегда тривиальна. Какой вывод следует из последнего замечания?

10.16. Исследуйте любой доступный вам продукт SQL и дайте ответы на следующие вопросы.

а) Можно ли привести примеры каких-либо операций выборки данных из представлений, которые приведут к ошибке?

б) Каковы правила обновления представлений для этого продукта? (Возможно, они окажутся во многом отличными от правил, соответствующих стандарту SQL: 1999, которые были изложены в разделе 10.6.)

10.17. Рассмотрим базу данных о поставщиках и деталях, но для простоты будем игнорировать наличие переменной отношения с данными о деталях. Ниже сокращенно описаны два возможных варианта проекта базы данных о поставщиках и поставках.

```
a) S { S#, SNAME, STATUS, CITY }
   SP { S#, P#, QTY }
```

```
б) SSP { S#, SNAME, STATUS, CITY, P#,
   QTY }' XSS { S#, SNAME, STATUS, CITY
   }
```

Проект *а* можно считать обычным. В проекте *б,* напротив, переменная отношения SSP содержит кортеж для каждой поставки, включающий соответствующий номер детали, количество деталей и полную информацию о поставщике, тогда как переменная отношения XSS содержит информацию о тех поставщиках, которые вовсе не поставляют деталей. (Обратите внимание, что оба проекта информационно равносильны и, следовательно, эти проекты служат иллюстрацией *принципа взаимозаменяемости.)* Напишите определения представлений для выражения проекта *б* в виде представлений в составе проекта *а* и наоборот. Также предоставьте все необходимые ограничения базы данных для каждого из вариантов проекта (см. главу 9, если потребуется освежить в памяти материал об ограничениях базы данных). Имеет ли тот или иной из проектов какие-либо явные преимущества по сравнению с другим? Если да, то какие?

10.18. Приведите решения упр. 10.1 —10.4 на языке SQL.

10.19. Алгоритм обновления представлений, сформированных на основе соединения, который приведен в разделе 10.4, часто подвергается критике, особенно на том ос новании, что (например) под операцией удаления кортежа из соединения таблиц поставщиков и отгрузок должна, безусловно, подразумеваться операция удаления из переменной отношения SP только соответствующих данных об отгрузке (т.е. при этом из переменной отношения S не должны удаляться данные о поставщике). При ведите свои соображения на эту тему.

10.20. В качестве заключительного (весьма важного!) упражнения в этой части книги еще раз прочитайте определение реляционной модели, приведенное в конце разде ла 3.2 главы 3, и убедитесь, что вы его полностью понимаете.

СПИСОК ЛИТЕРАТУРЫ

10.1. Adiba M. Derived Relations: A Unified Mechanism for Views, Snapshots, and Distributed Data // Proc. 1981 Int. Conf. on Very Large Data Bases.— Cannes, France.— September 1981. См. также более раннюю версию: Adiba M.E., Lindsay B.G. Database Snapshots// IBM Research Report RJ2772. — March 1980.

В работе впервые представлена концепция снимков, а также предложены соответствующая семантика и принципы реализации. Относительно реализации следует заметить, что внутри системы можно использовать различные типы *дифференциальных обновлений* или *инкрементное сопровождение,* т.е. системе не всегда требуется при обновлении снимка повторно выполнять исходный запрос в полном объеме.

10.2. Buff H.W. Why Codd[f]s Rule №6 Must Be Reformulated // ACM SIGMOD.- December 1988. — 17, №4.

В 1985 году Кодд (Codd) опубликовал набор из двенадцати правил, предназначенных, по его словам, для использования "в составе испытаний для определения того, действительно ли является таковым продукт, объявленный как полностью реляционный" [10.3]. Правило Кодда № 6 требует, чтобы "все представления, которые теоретически являются обновляемыми", были обновляемыми и в данной конкретной системе. В своей короткой статье Бафф (Buff) критикует это правило и утверждает,

что общая проблема обновляемое™ представлений неразрешима, т.е. не существует общего алгоритма определения обновляемости (или необновляемости) для произвольного представления. По мнению Макговерена (McGoveran) [10.11], эта статья "стала решающим и наиболее серьезным препятствием на пути к развертыванию исследований по проблеме обновления представлений". Но любая практическая реляционная реализация теоретической модели становится предметом разнообразных конечных ограничений (например, касающихся максимальной длины выражения), а из этого следует, что результаты Баффа не распространяются на подобную конкретную систему. Еще раз процитируем Макговерена: "Бафф не учитывает, что реляционная модель может применяться на физических компьютерах на основе лишь тех ограниченных реализаций реляционной алгебры, которые требуются для ее преобразования в соответствующую сокращенную форму; вместо этого в его статье рассматриваются исключительно чистая математика, которая служит для обоснования абстрактных, теоретических алгоритмов" [10.11].

10.3. E. F. Codd: "Is Your DBMS Really Relational?" и "Does Your DBMS Run by the Rules?" Computerworld (October Hand 21, 1985).

10.4. Chamberlin D.D., Gray J.N., Traiger I.L. Views, Authorization, and Locking in a Relational Data Base System // Proc. NCC 44. — Anaheim, Calif. Montvale, N.J.: AFIPS Press.-May 1975.

Содержит краткое логическое обоснование подхода, выбранного для организации обновления представлений в системе System R (и, следовательно, в системах SQL/DS и DB2, в стандарте SQL и т.п.). См. также [10.12], где можно найти аналогичное обоснование для системы University Ingres.

10.5. Darwen H. Without Check Option // Date C. J., Darwen H. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

10.6. Date C.J., McGoveran D. "Updating Union, Intersection, and Difference Views" и "Updating Joins and Other Views" // Date C J. Relational Database Writings **1991 — 1994.** — Reading, Mass.: Addison-Wesley, 1995.

Эти две статьи представляют собой неформальное описание схемы обновления представлений, которая частично рассматривалась в разделе 10.4. Один из их авторов (Макговерен) подготовил формальное описание схемы и на основе этого описания подал заявку на получение патента США [10.11].

10.7. Dayal U., Bernstein P.A. On the Correct Translation of Update Operations on Relational Views//ACM TODS. — September 1982. — 7, № 3.

Это одно из первых формальных описаний проблемы обновления представлений (только для представлений, заданных с помощью операции сокращения, проекции и соединения). Предикаты переменных отношения в нем не рассматриваются.

10.8. Furtado A.L., Casanova M.A. Updating Relational Views// [18.1].

Здесь описаны два достаточно общих подхода к решению проблемы обновления представлений. Один из них, подробно рассмотренный в данной главе, представляет собой попытку разработки общего механизма, работающего независимо от конкретной рассматриваемой базы данных. В этом подходе используются исключительно определения исследуемых представлений (т.е. семантика, имеющаяся в распоряжении системы). Другой подход, менее амбициозный, требует, чтобы администратор

базы данных явно указывал для каждого представления, какие обновления для него допустимы и какова семантика соответствующих операций. Это выполняется (фактически) посредством создания процедурного кода, реализующего обновления представлений в контексте исходных базовых переменных отношения. Статья включает обзор результатов, достигнутых в рамках каждого из подходов (по состоянию на 1985 год). Кроме того, представлен обширный список литературы. **10.9.** Goodman N. View Update Is Practical // InfoDB 5, № 2 (Summer 1990).

Весьма неформальный обзор проблемы обновляемости представлений. Вот цитата из введения (несколько перефразированная): "Дайал (Dayal) и Бернштейн (Bernstein) [10.7] доказали, что, по существу, все перспективные типы представлений не являются обновляемыми. Бафф (Buff) [10.2] доказал, что не существует алгоритма определения обновляемости произвольного представления. Поэтому может показаться, что нам практически больше не на что надеяться. Но трудно найти утверждение, более далекое от истины. В действительности обновление представлений возможно и реально". И далее в статье приведен ряд методов обновления представлений. Однако в ней не рассматривается исключительно важное понятие *предиката переменной отношения.*

10.10. Keller A.M. Algorithms for Translating View Updates to Database Updates for Views Involving Selections, Projections, and Joins // Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems. — Portland, Ore. — March 1985.

Предложен набор из пяти критериев, которым должен удовлетворять алгоритм обновления представлений: отсутствие побочных эффектов, только одношаговые изменения, отсутствие ненужных изменений, отсутствие возможности более простых замен и отказ от применения пар операций DELETE—INSERT вместо операции UPDATE. В этой работе также представлены алгоритмы, которые удовлетворяют изложенным критериям. Кроме всего прочего, приведенные в книге алгоритмы позволяют реализовать обновления одного типа с помощью операций другого типа. Например, операция DELETE ДЛЯ представления может быть реализована посредством операции UPDATE в исходной базовой переменной отношения (например, поставщик может быть удален из представления, описывающего лондонских поставщиков, посредством замены значения London атрибута города CITY значением Paris). В качестве другого примера, не упомянутого в работе Келлера (Keller), можно указать операцию DELETE В представлении v (где V определено посредством операции разности, v = A MINUS в), которая может быть реализована как вставка кортежа INSERT в переменную отношения в, а не как удаление кортежа DELETE из переменной отношения А. Заметьте, что в данной главе мы исключили подобные способы реализации операций обновления на основании описанного в ней принципа 6.

10.11. McGoveran D.O. Accessing and Updating Views and Relations in a Relational Database // U.S. Patent Application 10/114 609. — April 2002.

10.12. Stonebraker M.R. Implementation of Views and Integrity Constraints by Query Modification // Proc. ACM SIGMOD Intern. Conf. on Management of Data. — San Jose, Calif. — May 1975.

См. комментарии к [10.4].

ЧАСТЬ III

ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ

В этой части книги речь пойдет о проектировании базы данных, точнее — о проектировании *реляционной* базы данных. В общем эта задача формулируется следующим образом: выбрать подходящую логическую структуру для заданного массива данных, которые требуется поместить в базу данных. Иначе говоря, нужно решить вопрос, какие необходимы базовые переменные отношения и какой набор атрибутов они должны включать. Совершенно очевидно, что решение этой задачи имеет большое практическое значение.

Прежде чем приступить к подробному изложению данной темы, сделаем несколько предварительных замечаний.

- Следует заметить, что речь здесь пойдет о *логическом* (или *концептуальном)* проектировании, а не о разработке физического проекта. Это вовсе не значит, что физическое проектирование не имеет большого значения. Наоборот, создание физического проекта играет очень важную роль. Тем не менее, необходимо сделать следующие оговорки.

  а) Физическое проектирование может рассматриваться как отдельный последующий этап. Иначе говоря, для "правильного" проектирования базы данных прежде всего необходимо создать ее приемлемый логический (т.е. реляционный) проект, и лишь затем в качестве отдельного этапа разработки выполнить отображение этого логического проекта на определенные физические структуры, поддерживаемые конкретной СУБД. Другими словами, как уже отмечалось в главе 2, физический проект создается на базе логического и никак иначе.

---

[1] Фактически, в идеале система должна быть способна формировать физический проект автоматически, вообще не требуя вмешательства человека в этот процесс. Хотя на первый взгляд эта цель может показаться утопической, в приложении А описан подход к реализации, позволяющий перенести ее в область осуществимости.

> Физическое проектирование по определению является зависимым от специфики конкретной целевой СУБД. Поэтому данная тема выходит за рамки учебника общего плана, к которым следует отнести настоящую книгу. Логический проект, наоборот, совершенно независим от СУБД, и для его реализации могут использоваться некоторые строгие теоретические принципы. Безусловно, именно эти принципы и должны описываться в книге подобного рода.

К сожалению, мы живем в несовершенном мире и на практике часто случается так, что решения, принятые в процессе физической реализации проекта, могут оказывать существенное обратное влияние на его логический уровень. (Если говорить точнее, как уже несколько раз отмечалось в этой книге, это имеет место в основном из-за того, что в современных СУБД обычно поддерживаются только относительно простые средства отображения между логическим и физическим уровнями.) Иначе говоря, может потребоваться выполнить нескольких итераций цикла "логическое проектирование—физическое проектирование" и пойти на определенные компромиссы. Тем не менее, изложение материала в этой части ведется исходя из того, что предварительно необходимо создать логический проект без учета особенностей его будущей физической реализации (например, без учета требований к определенному уровню производительности). Таким образом, материал этой части книги в основном нацелен на решение задачи предварительного создания логического проекта базы данных.

- Хотя, как отмечалось ранее, речь пойдет, главным образом, о проекте *реляционной* системы, нет ни малейшего сомнения в том, что представленные здесь идеи в рав ной степени относятся и к нереляционным базам данных. Иначе говоря, мы счи таем, что правильный способ создания проекта нереляционной системы состоит в предварительной разработке ее корректного реляционного проекта с последую щим отображением (в виде отдельного этапа) на любые нереляционные структуры (например, иерархические), которые фактические поддерживаются в целевой СУБД.
- После всего сказанного следует подчеркнуть, что проектирование базы данных все еще во многом продолжает оставаться скорее искусством, чем наукой. Конечно, здесь снова следует повторить, что *существуют* некоторые научные принципы та кого проектирования, которые будут изложены в следующих четырех главах. Однако при проектировании базы данных возникает множество других проблем, которые не всегда можно решить, руководствуясь этими принципами. В результа те теоретики и практики в области создания баз данных разработали множество методологий[2] проектирования. Среди них есть как достаточно точные и строгие, так и не относящиеся к таковым. Однако все они в той или иной степени специа лизированы и предназначены для решения именно той проблемы, которая счита лась неразрешимой на момент создания конкретной методики. Иными словами, они предназначались для поиска *такого* варианта логического проекта, который был бы, бесспорно, лучшим в данной ситуации. Поскольку все эти методологии *остаются* в большей или меньшей степени *специализированными,* практически не

[2] Термин "методология" первоначально означал *изучение методов,* но со временем стал использоваться и для обозначения "системы методов и правил, которые применяются для исследований или выполнения работы в некоторой области науки или искусства" (см. *Chambers Twentieth Century Dictionary).*

существует объективных критериев, позволяющих выбрать среди них наиболее предпочтительную. Все же несмотря на это, в главе 14 будет описан один широко известный подход, менее специализированный, чем другие. Там же вы кратко ознакомитесь и с другими подходами, получившими коммерческую поддержку.

- Следует отметить некоторые допущения, используемые в дальнейшем изложении.

  а) Проектирование базы данных заключается не только в создании правильной структуры данных. Еще одной и, вероятно, более важной задачей является обеспечение целостности данных. Это замечание будет неоднократно повторено и усилено в тексте последующих глав.

  б) Далее в большинстве случаев проектирование рассматривается *независимо от приложения.* Иначе говоря, интерес представляют сами данные, а не то, как они будут *использоваться.* Независимость от приложения в этом смысле желательна по той простой причине, что в момент проектирования базы данных обычно еще неизвестны все возможные способы использования ее данных. Таким образом, необходимо, чтобы созданный проект был *стабильным,* т.е. оставался работоспособным даже при возникновении в приложениях новых (т.е. неизвестных на момент создания исходного макета) требований к данным. Следуя этим допущениям (и используя терминологию из главы 2), можно сказать, что в этой части книги обсуждается создание *концептуальной схемы,* т.е. абстрактного логического проекта, не зависящего от аппаратного обеспечения, операционной системы, целевой СУБД, языка программирования, требований пользователей и т.д. В частности, как указывалось ранее, здесь нас *не* интересуют компромиссные решения, принимаемые, например, в целях достижения требуемой производительности.

- Как отмечалось выше, задача проектирования базы данных заключается в том, чтобы решить, какие базовые переменные отношения и с каким набором атрибутов следует использовать. Фактически для этого также необходимо выяснить, какие следует использовать *домены,* или *типы.* Ко времени написания данной книги этой теме было посвящено совсем немного публикаций, поэтому освещение данного вопроса будет очень кратким (исключениями являются лишь [14.12], [14.44]).

Данная часть имеет следующую структуру. В главе 11 даны некоторые теоретические основы, а в главах 12 и 13 описаны основные идеи *дальнейшей нормализации,* построенные на этих теоретических принципах и позволяющие придать смысл неформальным утверждениям о преимуществах того или иного проекта. Затем в главе 14 рассматривается *семантическое моделирование,* в частности, описываются концепции построения модели "сущность—связь" (или ER-модели) и демонстрируется, как их можно использовать на практике для нисходящего проектирования систем (начиная с сущностей реального мира и заканчивая формальным реляционным проектом базы данных).

ГЛАВА 11

Функциональные зависимости

11.1. ВВЕДЕНИЕ

В этой главе речь пойдет о концепции **функциональной зависимости,** которая была названа Хью Дарвеном (Hugh Darwen) в личной беседе с автором если "не совсем фундаментальной, то очень близкой к таковой". Эта концепция лежит в основе многих обсуждаемых в последующих главах тем, включая, в частности, теорию проектирования базы данных, описанную в главе 12. Но следует сразу же отметить, что ее значимость не ограничивается только указанной областью применения; поэтому фактически данную главу вполне можно было включить в часть **II** настоящей книги вместо части **III.**

По сути, функциональная зависимость (далее для ее обозначения часто будет использоваться аббревиатура ФЗ) является *связью типа "многие к одному"* между множествами атрибутов внутри данной переменной отношения. Например, для переменной отношения поставок SP существует функциональная зависимость между множествами атрибутов {S#,P#} и {QTY}. Это означает, что для любого допустимого значения этой переменной отношения справедливы следующие правила.

- Для любой заданной пары значений атрибутов s# и P# существует только одно соответствующее им значение[1] атрибута QTY.
- Но одно и то же соответствующее им значение атрибута QTY (в общем случае) могут иметь многие разные пары значений атрибутов S# и P#.

Обратите внимание, что в показанном на рис. 3.8 (см. стр. 128) примере значения переменной отношения SP удовлетворяют этим правилам. Следует также отметить — нам снова приходится сталкиваться с концепцией, определение которой основано на понятии равенства кортежей.

В разделе 11.2 этой главы концепция функциональной зависимости определена более точно, с учетом того, что существуют и такие функциональные зависимости, которые выполняются применительно к данной переменной отношения лишь в некоторых частных случаях, и такие, которые выполняются применительно к ней *всегда*. Как было указано выше, функциональные зависимости представляют собой основу для применения научного подхода к решению целого ряда практических задач, поскольку обладают богатым набором интересных свойств, позволяющих формально и строго решить многие проблемы. Ниже, в разделах 11.3—11.6, будут подробно описаны некоторые из этих свойств и даны объяснения по поводу их практического применения. В конце главы, в разделе 11.7, представлено краткое резюме.

Примечание. Эта глава книги — самая формальная, и при первом чтении многие ее разделы можно пропустить. Фактически основная часть материала, необходимого для изучения следующих трех глав, содержится в разделах 11.2 и 11.3. Поэтому все остальные разделы можно лишь просмотреть, а затем вновь вернуться к ним, ознакомившись с остальными тремя главами этой части.

Небольшое примечание в отношении используемой терминологии. В англоязычной литературе почти как равноценные употребляются два термина: functional **dependence** и functional **dependency.** Согласно нормам использования английского языка, термин *dependence* следовало бы применять для обозначения самой функциональной зависимости, а термин *dependency* — для обозначения зависимых объектов. Однако термин *функциональная зависимость* часто приходится применять во множественном числе, а в таких случаях термин *dependencies* кажется более удобным для произношения, чем *dependences*. Именно это и определяет смешанное использование в литературе обоих терминов.

11.2. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Для демонстрации основных идей данного раздела используется несколько измененная версия переменной отношения с данными о поставках, которая в дополнение к обычным атрибутам s#, P# и QTY будет содержать также атрибут CITY, представляющий

---

[1]Следует отметить, что это довольно формальное утверждение является истинным именно потому, что на рассматриваемые данные распространяются некоторые "практические ограничения" (см. главу 9). В данном случае таковым является правило, что в любой отдельно взятый момент времени любой отдельно взятый поставщик не может выполнить больше одной поставки любой отдельно взятой детали. Другими словами, образно говоря, функциональные зависимости относятся к сфере семантики (они определяют, что означают данные) и поддерживаются сознательно, а не возникают в результате счастливой случайности, благодаря которой в определенное время в базе данных вдруг появляются некоторые значения, связанные между собой определенной закономерностью.

Рис. 11.1. Пример значения переменной отношения SCP

город соответствующего поставщика. Во избежание путаницы, далее эту измененную переменную отношения мы будем называть SCP. Она представлена на рис. 11.1 в виде таблицы.

SCP

S#	CITY	P#	QTY
S1	London	P1	100
S1	London	P2	100
S2	Paris	P1	200
S2	Paris	P2	200
S3	Paris	P2	300
S4	London	P2	400
S4	London	P4	400
S4	London	P5	400

В этой области, как и во многих других, важно очень четко различать, во-первых, *значение* переменной отношения в определенный момент (вариант *а),* во-вторых, *множество всех возможных значений,* которые переменная отношения может принимать в различные моменты (вариант *б).* Сначала дадим определение концепции функциональной зависимости для варианта *а,* а затем — для варианта *б.*

и Функциональная зависимость, вариант *а.* Пусть r является отношением, а X и Y — произвольными подмножествами множества атрибутов отношения r. Тогда Y **функционально зависимо** от x, что в символическом виде записывается как

X → Y

(читается либо как "X **функционально определяет** Y", либо как "X стрелка Y") тогда и только тогда, когда каждое значение множества x отношения r связано точно с одним значением множества Y отношения r. Иначе говоря, если два кортежа отношения r совпадают по значению x, они совпадают и по значению Y.

Например, отношение SCP (см. рис. 11.1) удовлетворяет требованиям приведенной ниже функциональной зависимости, поскольку все кортежи отношения SCP с одинаковыми значениями атрибута S# имеют одно и то же значение атрибута CITY.

{ S# } → { CITY }

На самом деле, это отношение удовлетворяет требованиям сразу нескольких функциональных зависимостей.

```
{S#,P#}→{QTY}
{S#,P#}→{CITY}
{S#,P#}→{CITY,QTY}
{S#,P#}→{S#}
{S#,P#}→{S#,P#,CITY,QTY}
  {S#} →{QTY}
{QTY} → {S#}
```

Упражнение. Проверьте правильность этого утверждения.

Левую **и** правую части символической записи функциональной зависимости иногда называют, соответственно, **детерминантом** и **зависимой частью.** Как говорится в определении, детерминант и зависимая часть являются *множествами* атрибутов. Когда множество содержит только один атрибут, оно называется *одноэлементным множеством.* В таком случае скобки исключаются и символическая запись принимает следующий вид.

S# → CITY

Как уже было указано, эти функциональные зависимости относятся к варианту *а,* т.е. к отдельным *значениям* переменных отношения. Однако при рассмотрении самих *переменных отношения,* в частности *базовых* переменных отношения, интерес представляют не столько функциональные зависимости в их существующих на некоторый момент конкретных значениях, сколько функциональные зависимости, выполняющиеся для *всех возможных значений* данной переменной отношения. Например, в случае переменной отношения SCP функциональная зависимость

S# → CITY

выполняется для всех возможных значений переменной отношения SCP, поскольку в любой момент одному поставщику соответствует в точности один город. По этой причине любые два кортежа переменной отношения SCP в один и тот же момент и с одним и тем же номером поставщика должны соответствовать одному и тому же городу. Практически утверждение, что данная функциональная зависимость выполняется "всегда" (т.е. для всех возможных значений SCP), является *ограничением целостности* для переменной отношения SCP, поскольку при этом накладываются определенные ограничения на *все* ее допустимые значения. Ниже приведена формулировка этого ограничения, которая выполнена с использованием основанного на исчислении предикатов синтаксиса языка Tutorial D, который был представлен в главе 9.

```
CONSTRAINT S# CITY FD
    FORALL SCPX FORALL SCPY
       ( IF SCPX.S# = SCPY.S#
         THEN SCPX.CITY = SCPY.CITY END IF ) ;
```

Здесь SCPX и SCPY — переменные области значений, принимающие свои значения в области определения переменной отношения SCP. Выражение s# → CITY может расцениваться как сокращенный способ представления этой более длинной формулировки.

Упражнение. Приведите алгебраическую версию этого определения.

Ниже приведено определение концепции функциональной зависимости для варианта *б* (дополнения к определению, которое соответствует варианту *а,* отмечены **полужирным шрифтом).**

- Функциональная зависимость, вариант *б.* Пусть R является **переменной** отношения, а X и Y — произвольными подмножествами множества атрибутов переменной! отношения R. Тогда Y функционально зависимо от X, что в символическом виде| записывается как

  X → Y

  (и читается либо как "X функционально определяет Y", либо как "X стрелка Y") тогда и только тогда, когда **для любого допустимого значения переменной отношения R** каждое значение множества X отношения R связано точно с одним значением

множества Y отношения R. Иначе говоря, **для любого допустимого значения переменной отношения R,** если два кортежа переменной отношения R совпадают по значению X, они также совпадают и по значению Y.

Впредь термин *функциональная зависимость* будет использоваться в последнем толковании, *более ограничительном и безотносительном ко времени* (за исключением особо отмеченных случаев).

Ниже перечислено несколько функциональных зависимостей, выполняющихся (безотносительных ко времени) для переменной отношения SCP.

```
{ S#, P# } → QTY
{ S#, P# } → CITY
{ S#, P# } → { CITY, QTY }
{ S#, P# } → S#
{ S#, P# } → { S#, P#, CITY, QTY }
{ S# }     → CITY
```

Обратите внимание, в частности, на функциональные зависимости, которые выполняются для отношения, представленного на рис. 11.1, но *не* "всегда" выполняются для переменной отношения SCP.

```
S#  → QTY
QTY → S#
```

Иначе говоря, такое утверждение, как (например) "количество деталей в каждой поставке данного поставщика одинаково", действительно оказалось истинным для конкретных значений, присутствующих в отношении на рис. 11.1, но ложным для всех возможных допустимых значений переменной отношения SCP.

Следует отметить, что если x является потенциальным ключом переменной отношения R, то все атрибуты Y переменной отношения R должны обязательно быть функционально зависимыми от X (этот факт упоминается в разделе 9.10 главе 9 и непосредственно следует из определения потенциального ключа). Аналогично, в переменной отношения деталей p необходимо, чтобы всегда выполнялась следующая зависимость.

```
P# → { P#, PNAME, COLOR, WEIGHT, CITY }
```

Действительно, если переменная отношения R удовлетворяет функциональной зависимости A → в и A *не* является потенциальным ключом[2], то R обязательно будет характеризоваться некоторой **избыточностью.** Например, если обратиться к переменной отношения SCP, то наличие в ней функциональной зависимости S# → CITY приведет к тому, что сведения о месте расположения поставщика в определенном городе повторятся много раз (это хорошо видно на рис. 11.1). Подробнее данный вопрос обсуждается в следующей главе.

Теперь, даже если ограничиться рассмотрением функциональных зависимостей, которые имеют место в любой момент, полный набор функциональных зависимостей, выполняющихся для всех допустимых значений заданной переменной отношения, может быть все еще очень большим, что можно видеть на примере переменной отношения SCP.

---

[2] При тех условиях, что эта функциональная зависимость не является *тривиальной* (раздел 10.3), причем A не является *суперключом* (раздел 10.5), а R содержит по крайней мере два кортежа (!).

(Упражнение. Попробуйте записать полное множество функциональных зависимостей, удовлетворяемых переменной отношения SCP.) Большая часть оставшегося в этой главе материала посвящена поискам методов сокращения обширного множества функциональных зависимостей до некоторых допустимых размеров.

Почему эта цель столь важна? Как уже отмечалось, одна из причин состоит в том, что функциональные зависимости являются ограничениями целостности, поэтому желательно, чтобы СУБД обеспечивала их соблюдение. Следовательно, для каждого заданного множества функциональных зависимостей s желательно найти такое множество т, которое (в идеальной ситуации) было бы *существенно* меньше множества S и при этом каждая функциональная зависимость в множестве S могла бы быть заменена функциональной зависимостью из множества т. Если бы такое множество т было найдено, то СУБД достаточно было бы контролировать выполнение функциональных зависимостей из множества т, что автоматически обеспечивало бы соблюдение всех функциональных зависимостей из множества s. Именно поэтому задача поиска подходящего множества т представляет большой практический интерес.

11.3. ТРИВИАЛЬНЫЕ И НЕТРИВИАЛЬНЫЕ ЗАВИСИМОСТИ

Примечание. Далее в этой главе выражение *функциональная зависимость* будет иногда для краткости заменяться словом *зависимость,* а *функционально зависит от* — словами *функционально определяется как* и т.п.

Очевидным способом сокращения существующего набора функциональных зависимостей является исключение из него *тривиальных* зависимостей. Зависимость называется **тривиальной,** если она не может не выполняться. В качестве примера приведем следующую тривиальную функциональную зависимость, существующую в переменной отношения SCP, которая обсуждалась в предыдущем разделе.

{ S#, P# } → S#

Действительно, функциональная зависимость является *тривиальной* тогда и только тогда, когда правая часть ее символической записи является подмножеством (не обязательно строгим подмножеством) левой части.

Как подразумевается в самом их названии, с практической точки зрения подобные зависимости не представляют значительного интереса, в отличие от **нетривиальных** зависимостей, которые действительно являются ограничениями целостности в полном смысле этого понятия. Однако с точки зрения формальной теории зависимостей необходимо учитывать все зависимости, как тривиальные, так и нетривиальные.

11.4. ЗАМЫКАНИЕ МНОЖЕСТВА ЗАВИСИМОСТЕЙ

Как упоминалось выше, из одних функциональных зависимостей могут следовать другие функциональные зависимости. Например, рассмотрим приведенную ниже зависимость.

{ S#, P# } → { CITY, QTY }

Из нее следуют приведенные ниже функциональные зависимости.

{ S#, P# } →
CITY { S#, P# }
→ QTY

В качестве более сложного примера можно привести переменную отношения R с атрибутами A, в и C, для которых выполняются функциональные зависимости A → в и B → C. Нетрудно заметить, что в этом случае также выполняется функциональная зависимость A B и C с, которая называется **транзитивной** функциональной зависимостью, т.е. с зависит от A *транзитивно,* или *проходя через* в.

Множество всех функциональных зависимостей, которые следуют из заданного множества функциональных зависимостей s, называется **замыканием** множества s и обозначается символом S+ (необходимо учитывать то, что оно не имеет ничего общего с понятием замыкания, которое рассматривается в реляционной алгебре). Из приведенного определения следует, что для решения сформулированной задачи необходимо найти алгоритм вычисления S_+ на основе S. Первая попытка решить эту проблему была предпринята в статье Армстронга (Armstrong) [11.2], в которой автор предложил набор **правил вывода** новых функциональных зависимостей на основе заданных (эти правила также часто называют **аксиомами Армстронга).** Эти правила вывода могут формулироваться разными способами, из которых самым простым является следующий. Пусть A, **в** и C — произвольные подмножества множества атрибутов заданной переменной отношения R. Условимся также, что символическая запись AB означает объединение множеств A И B. Тогда правила вывода определяются следующим образом.

1. Правило **рефлексивности.** Если множество **в** является подмножеством множества A, **ТО A → B.**
2. Правило **дополнения.** Если A → B, то AC → BC.
3. Правило **транзитивности.** Если A → B и B→C, то A → C.

Каждое из этих трех правил может быть непосредственно доказано на основе определения функциональной зависимости (безусловно, первое из них — это просто само определение **тривиальной** зависимости). Более того, эти правила являются **полными** в том смысле, что для заданного множества функциональных зависимостей s минимальный набор функциональных зависимостей, которые подразумевают все зависимости из множества S, может быть выведен из ФЗ множества s на основе только этих правил. Они являются также **непротиворечивыми,** поскольку с их помощью не могут быть выведены никакие дополнительные функциональные зависимости (т.е. зависимости, которые не обусловлены функциональными зависимостями множества S). Иначе говоря, эти правила могут использоваться для получения замыкания S_+.

В целях упрощения задачи практического вычисления замыкания S_+, из трех приведенных выше правил можно вывести несколько дополнительных правил. (В дальнейшем предполагается, что D — это еще одно произвольное подмножество множества атрибутов R.)

4. Правило **самоопределения.** A → A.
5. Правило **декомпозиции.** Если A → BC, то A → Ви A → C.
6. Правило **объединения.** Если A → в и A → C, то A → BC.
7. Правило **композиции.** Если A → B и C → D, то AC → BD.

Кроме того, Дарвен (Darwen) в своей работе [11.7] доказал следующее правило, которое он назвал *общей теоремой объединения.*

8. Если A→ B и C → D, то A ∪ (C - B) → BD (здесь символ "∪" обозначает операцию объединения множеств, а символ "-" — операцию разности множеств).

Название *общая теорема объединения* указывает на то, что некоторые из перечисленных выше правил могут быть выведены как частные случаи этой теоремы [11.7].

Пример. Пусть дана некоторая переменная отношения R с атрибутами A, B, C, D, E, F и следующими функциональными зависимостями.

A → BC
B → E
CD →
EF

Обратите внимание, что способ записи был немного дополнен (без ущерба для смысла); например, символы BC означают множество, состоящее из атрибутов B и C, хотя раньше они означали *объединение* B и C, где B и C были *множествами* атрибутов.

Примечание. Если необходимо, то этому примеру можно придать более конкретный смысл, а именно: A— личный номер сотрудника, в — номер отдела, с — личный номер руководителя (начальника) данного сотрудника, D— номер проекта, возглавляемого данным руководителем (уникальный для каждого отдельно взятого руководителя), E — название отдела, F — количество времени, уделяемое данным руководителем указанному проекту.

Теперь можно показать, что для переменной отношения R выполняется также функциональная зависимость AD → F, которая вследствие этого принадлежит к замыканию заданного множества функциональных зависимостей.

1. A → BC (Дано).
2. A→ C (Следует из п. 1 согласно правилу декомпозиции).
3. AD → CD (Следует из п. 2 согласно правилу дополнения).
4. CD → EF (Дано).
5. AD → EF (Следует из пп. 3 и 4 согласно правилу транзитивности).
6. AD → F (Следует из п. 5 согласно правилу декомпозиции).

11.5. ЗАМЫКАНИЕ МНОЖЕСТВА АТРИБУТОВ

В принципе, замыкание S_+ для заданного множества функциональных зависимостей S можно вычислить с помощью следующего алгоритма: "Повторно применять правила из предыдущего раздела до тех пор, пока остается возможным создание новых функциональных зависимостей". Но на практике редко требуется вычислить замыкание *как таковое,* а потому и только что упомянутый алгоритм вряд ли будет достаточно эффективным. Однако из этого раздела вы узнаете, как можно вычислить некоторое подмножество замыкания, а именно то подмножество, которое состоит из всех функциональных зависимостей с некоторым (указанным) множеством z атрибутов, расположенных в левой части выражения зависимости. Точнее говоря, мы покажем, что для заданной переменной отношения R, заданного множества атрибутов этой переменной отношения z и заданного множества функциональных зависимостей S, выполняющихся для переменной

отношения R, можно найти множество[3] всех атрибутов переменной отношения R, которые функционально зависимы от Z, т.е. так называемое замыкание Z_+ множества Z в пределах S. Простой алгоритм вычисления этого замыкания показан на рис. 11.2.

Упражнение. Докажите правильность этого алгоритма.

```
CLOSURE[Z,S] := Z ;
do <бесконечно> ;
   for каждой функциональной зависимости X → Y в S
       do ;
          if X ⊆ CLOSURE[Z,S]              /* CLOSURE - замыкание */
          then CLOSURE[Z,S] := CLOSURE[Z,S] ∪ Y ;
       end
   if CLOSURE[Z,S] не изменилось после этой итерации
   then выйти из цикла ;                   /* Вычисления закончены */
end ;
```

Рис. 11.2. Вычисление замыкания Z+ множества Z в пределах S

Пример. Предположим, что дана переменная отношения R с атрибутами A, B, C, D, E и F и следующими функциональными зависимостями.

A → BC
E → CF
B → E
CD → EF

Вычислим замыкание $\{A, B\}_+$ множества атрибутов {A, B} исходя из заданного множества функциональных зависимостей.

1. Присвоим замыканию CLOSURE [Z, S] начальное значение — множество {A, B}.
2. Выполним внутренний цикл четыре раза — по одному разу для каждой заданной функциональной зависимости. После первой итерации (для зависимости A → BC) будет обнаружено, что левая часть действительно является подмножеством замыкания CLOSURE [Z, S]. Таким образом, к результату можно добавить атрибуты B и C. Теперь замыкание CLOSURE [Z, S] представляет собой множество {A, B, C}.
3. После второй итерации (для зависимости E → CF) обнаруживается, что левая часть *не* является подмножеством полученного до этого момента результата, который, таким образом, остается неизменным.
4. После третьей итерации (для зависимости B → E) к замыканию CLOSURE [Z, S] будет добавлено множество E, поэтому замыкание теперь будет иметь вид {A, B, C, E}.
5. После четвертой итерации (для зависимости CD → EF) замыкание CLOSURE [Z, S] останется неизменным.

---

[3] Обратите внимание, что тем самым определены два типа замыканий: замыкание множества функциональных зависимостей и замыкание множества атрибутов, которые определены в множестве функциональных зависимостей. Следует также отметить, что для тех и других применяется одна и та же система обозначений в виде знака "плюс" в позиции верхнего индекса. Автор надеется, что такое двойное применением одинаковой системы обозначений не приведет к путанице.

6. Далее внутренний цикл выполняется еще четыре раза. После первой итерации результат останется прежним, после второй он будет расширен до $\{A,B,C,E,F\}$, а после третьей и четвертой — снова не изменится.
7. Наконец, после еще одного четырехкратного прохождения цикла замыкание CLOSURE [Z,S] останется неизменным и весь процесс завершится с результатом $\{A,B\}_+ = \{A,B,C,E,F\}$.

Следует отметить, что если Z (как было указано выше) — множество атрибутов переменной отношения R, а S — множество функциональных зависимостей, которые соблюдаются в R, то множество функциональных зависимостей, которые соблюдаются в R и содержат z в левой части, представляет собой множество, состоящее из всех функциональных зависимостей в форме Z → Z', где Z' — некоторое подмножество замыкания Z_+ множества z, принадлежащего к S. В таком случае замыкание S_+ первоначального множества функциональных зависимостей S представляет собой объединение всех подобных множеств функциональных зависимостей, взятых по всем множествам атрибутов Z.

Из сказанного выше можно сделать очень важное заключение: для заданного множества функциональных зависимостей S легко можно указать, будет ли заданная функциональная зависимость x → Y следовать из S, поскольку это возможно тогда и только тогда, когда множество Y является подмножеством замыкания x_+ множества X для заданного множества s. Иначе говоря, таким образом может быть создан простой способ определения того, будет ли данная функциональная зависимость x → Y включена в замыкание S_+ множества S, фактически не связанный с необходимостью вычисления самого этого замыкания S_+.

Еще одно важное заключение основано на следующем факте. Прежде всего, вспомним определение понятия **суперключа** из главы 9. Суперключ переменной отношения R — это множество атрибутов переменной отношения R, которое в виде подмножества (но не обязательно строгого подмножества) содержит по крайней мере один *потенциальный* ключ. Из этого определения непосредственно следует, что суперключи для данной переменной отношения R — это такие подмножества к множества атрибутов переменной отношения R, что функциональная зависимость

$$K \rightarrow A$$

соблюдается для каждого атрибута A переменной отношения R. Другими словами, множество к является суперключом тогда и только тогда, когда замыкание $к_+$ для множества к в пределах заданного множества функциональных зависимостей является множеством абсолютно всех атрибутов переменной отношения R. (Кроме того, множество к является *потенциальным* ключом тогда и только тогда, когда оно является неприводимым суперключом).

11.6. НЕПРИВОДИМЫЕ МНОЖЕСТВА ЗАВИСИМОСТЕЙ

Пусть S1 и S2 — два множества функциональных зависимостей. Если любая функциональная зависимость, которая следует из множества зависимостей si, следует также из множества зависимостей S2 (т.е. если замыкание $S1_+$ является подмножеством замыкания $S2_+$,

то множество S2 называется **покрытием**[4] для множества si. Это означает, что если СУБД обеспечит соблюдение ограничений, представленных зависимостями множества S2, то автоматически будут соблюдены и все ограничения, устанавливаемые зависимостями множества S1.

Далее, если множество S2 является покрытием для множества S1, а множество S1 одновременно является покрытием для множества S2 (т.е. если $S1_+=S2_+$), то множества S1 и S2 эквивалентны. Ясно, что если множества S1 и S2 эквивалентны, то соблюдение СУБД ограничений, представленных зависимостями множества S2, автоматически обеспечит соблюдение ограничений, представленных зависимостями множества S1, и наоборот.

Множество функциональных зависимостей называется **неприводимым**[5] тогда и только тогда, когда оно обладает всеми тремя перечисленными ниже свойствами.

1. Правая (зависимая) часть каждой функциональной зависимости из множества S содержит только один атрибут (т.е. является одноэлементным множеством).
2. Левая часть (детерминант) каждой функциональной зависимости из множества S, в свою очередь, является неприводимой, т.е. ни один атрибут из детерминанта не может быть опущен без изменения замыкания S_+ (без преобразования множества S в какое-то другое множество, не эквивалентное множеству S). В этом случае функциональная зависимость называется **неприводимой слева.**
3. Ни одна функциональная зависимость из множества S не может быть удалена из множества s без изменения его замыкания S_+ (т.е. без преобразования множества s в некоторое иное множество, не эквивалентное множеству S).

В отношении пп. 2 и 3 следует отметить, что не обязательно иметь полную информацию о *составе* замыкания S_+ для получения ответа на вопрос, изменится ли это замыкание при удалении из исходного множества какой-либо функциональной зависимости. В качестве примера рассмотрим уже знакомую переменную отношения деталей P. В ней соблюдаются, кроме всех прочих, функциональные зависимости, перечисленные ниже.

```
P# → PNAME
P# → COLOR
P# → WEIGHT
P# → CITY
```

Нетрудно заметить, что это множество функциональных зависимостей является неприводимым: правая часть каждой зависимости содержит только один атрибут, а левая часть также, безусловно, является неприводимой. Кроме того, ни одна из перечисленных функциональных зависимостей не может быть исключена без изменения замыкания множества (т.е. без *утраты некоторой информации).* В противоположность этому, приведенные ниже множества функциональных зависимостей не являются неприводшыми.

1. P# → { PNAME, COLOR }
   P# → WEIGHT
   P# → CITY

---

[4] Некоторые авторы используют термин "покрытие" для обозначения *эквивалентного* множества (это понятие также будет скоро определено в данной книге).

[5] В литературе подобное множество часто называется *минимальным.*

(Правая часть первой ФЗ не является одноэлементным множеством.)

2. { P#, PNAME } → COLOR
   P# → PNAME
   P# → WEIGHT
   P# → CITY
   (Первую ФЗ можно упростить, исключив атрибут PNAME в левой части без изменения замыкания, т.е. она не является неприводимой слева.)
3. p# → p#
   P# → PNAME
   P# → COLOR
   P# → WEIGHT
   P# → CITY
   (Здесь первую ФЗ можно исключить без изменения замыкания.)

Теперь можно сформулировать утверждение, что для любого множества функциональных зависимостей существует по крайней мере одно эквивалентное множество, которое является неприводимым. Это можно доказать достаточно легко. Пусть дано исходное множество зависимостей s. Тогда в силу того, что существует правило декомпозиции, можно без утраты общности предположить, что каждая функциональная зависимость в этом множестве s имеет одноэлементную правую часть. Далее для каждой зависимости f из этого множества S следует проверить каждый атрибут A в левой части зависимости f. Если удаление атрибута A из левой части зависимости f не приводит к изменению замыкания S_+, то этот атрибут следует удалить. Затем для каждой зависимости f, оставшейся в множестве s, необходимо проверить, приводит ли ее удаление из множества S к изменению замыкания S_+; в случае отрицательного ответа следует удалить зависимость f из множества S. Получившееся в результате таких действий множество S является неприводимым и эквивалентным исходному множеству S.

Пример. Пусть дана переменная отношения R с атрибутами A, в, с, D и следующими функциональными зависимостями.

A → BC
B → C
A → B
AB → C
AC → D

Теперь попробуем найти неприводимое множество функциональных зависимостей, эквивалентное данному множеству.

1. Прежде всего, следует переписать заданные ФЗ таким образом, чтобы каждая из них имела одноэлементную правую часть.

   A → B
   A → C
   B → C
   A → B
   AB → C
   AC → D

Нетрудно заметить, что зависимость A→ **в** встречается дважды, так что одну из них можно удалить.

2. Затем в левой части зависимости AC → D может быть исключен атрибут с, по скольку дана зависимость A → C, из которой по правилу дополнения можно по лучить зависимость A →AC. Кроме того, дана зависимость AC → D, из которой по правилу транзитивности можно получить зависимость A → D. Таким образом, атрибут **с** в левой части исходной зависимости AC → D является избыточным.
3. Также заметим, что может быть исключена зависимость AB → C, поскольку дана зависимость A→ C, из которой по правилу дополнения можно получить зависи мость AB → CB, а затем по правилу декомпозиции—зависимость AB → C.
4. Наконец, зависимость A→ с следует из зависимостей A → B и B → C, так что она также может быть отброшена. В результате получено следующее неприводи мое множество зависимостей.

A → B
B → C
A → D

Множество функциональных зависимостей I, которое неприводимо и эквивалентно другому множеству функциональных зависимостей s, называется **неприводимым эквивалентом** множества S. Таким образом, в системе вместо исходного множества функциональных зависимостей S с тем же успехом может использоваться его неприводимый эквивалент I (здесь следует повторить, что для вычисления неприводимого эквивалента I не обязательно вычислять замыкание s_+). Однако необходимо отметить, что для любого заданного множества функциональных зависимостей не всегда существует *уникальный* неприводимый эквивалент (см. упр. 11.12).

11.7. РЕЗЮМЕ

Функциональная зависимость— это связь типа "многие **к** однбму" между двумя множествами атрибутов заданной переменной отношения (она представляет собой наиболее широко распространенный и важный вид ограничения целостности). Для заданной переменной отношения R зависимость A → B (где A и **в** являются подмножествами множества атрибутов переменной отношения R) выполняется для переменной отношения R тогда и только тогда, когда любые два кортежа переменной отношения R с одинаковыми значениями атрибутов множества A имеют одинаковые значения атрибутов множества в. Каждая переменная отношения обязательно удовлетворяет некоторым **тривиальным** функциональным зависимостям; причем функциональная зависимость тривиальна тогда и только тогда, когда ее правая **(зависимая)** часть является подмножеством ее левой части **(детерминанта).**

Из одних функциональных зависимостей следуют другие зависимости. Для данного множества зависимостей S **замыканием** S_+ называется множество всех функциональных зависимостей, которые следуют из зависимостей множества S. Множество S обязательно является подмножеством собственного замыкания S_+. **Правила логического вывода Армстронга** обеспечивают **непротиворечивую** и **полную** основу для вычисления замыкания S_+

для заданного множества s (но в действительности обычно проведение этих вычислений не требуется). На основании правил Армстронга можно легко получить несколько дополнительных удобных правил.

Для данного подмножества z множества атрибутов переменной отношения R и множества функциональных зависимостей S, которые выполняются в переменной отношения R, **замыканием** z_+ подмножества Z для множества S называется такое множество всех атрибутов A переменной отношения R, что функциональная зависимость $Z \rightarrow A$ является членом замыкания S_+. Если замыкание z_+ состоит из всех атрибутов переменной отношения R, то подмножество Z называют **суперключом** переменной отношения R (а неприводимый суперключ, в свою очередь, называется **потенциальным ключом). В** этой главе было дано описание простого алгоритма для получения замыкания z_+ на основе z и s и, следовательно, для определения того, является ли данная зависимость $x \rightarrow Y$ членом замыкания s_+ (функциональная зависимость $X \rightarrow$ У является членом замыкания S_+ тогда и только тогда, когда множество Y является подмножеством замыкания x_+).

Два множества функциональных зависимостей s1 и S2 **эквивалентны** тогда и только тогда, когда они **являются покрытиями** друг для друга, т.е. $S1_+=S2_+$. Каждое множество функциональных зависимостей эквивалентно по крайней мере одному **неприводимому** множеству. Множество функциональных зависимостей является неприводимым, если, во-первых, каждая функциональная зависимость этого множества имеет одноэлементную правую часть; во-вторых, если ни одна функциональная зависимость множества не может быть удалена без изменения замыкания этого множества; в-третьих, если ни один атрибут не может быть удален из левой части любой функциональной зависимости данного множества без изменения замыкания множества. Если I является неприводимым множеством, которое эквивалентно множеству S, то проверка выполнения функциональных зависимостей из множества I автоматически обеспечит выполнение всех функциональных зависимостей из множества S.

В заключение следует отметить, что многие высказанные выше соображения можно расширить в отношении ограничений целостности вообще, а не только функциональных зависимостей. Например, в общем случае верны следующие допущения и выводы.

- Некоторые ограничения целостности являются тривиальными.
- Из одних ограничений целостности следуют другие ограничения.
- Множество всех ограничений, которые следуют из заданного множества ограничений, может рассматриваться как замыкание этого заданного множества.
- Выяснение вопроса, будет ли некоторое ограничение находиться в некотором за мыкании (т.е. будет ли заданное ограничение следовать из некоторых заданных ограничений), является очень интересной практической задачей.
- Не менее интересной практической задачей является поиск неприводимого эквивалента для некоторого заданного множества установленных ограничений.

Благодаря наличию непротиворечивого и полного множества правил вывода различных функциональных зависимостей, работать с ними удобнее, чем с ограничениями целостности как таковыми. В списках рекомендуемой литературы в конце этой главы и главы 13 даны ссылки на работы, в которых описываются несколько других типов ограничений (MVD, JD и IND); для них также существуют подобные наборы правил вывода. Однако в

данной книге другие существующие типы ограничений не рассматриваются, столь же подробно и полно, как функциональные зависимости.

УПРАЖНЕНИЯ

11.1. Сформулируйте ответы на следующие вопросы, касающиеся функциональных зависимостей.

а) Пусть R является переменной отношения степени n. Каково максимальное количество функциональных зависимостей (как тривиальных, так и нетривиальных), которым может удовлетворять переменная отношения R?

б) Предположим, что A и B — множества атрибутов в функциональной зависимости A → B. Что произойдет, если любое из этих множеств окажется пустым?

11.2. Что конкретно означает утверждение, что правила Армстронга являются непротиворечивыми и полными?

11.3. **Докажите** правила *рефлексивности, дополнения* **и** *транзитивности,* используя только основное определение функциональной зависимости.

11.4. Докажите, что из трех указанных выше правил следуют правила *самоопределения, декомпозиции, объединения* и *композиции.*

11.5. Докажите *общую теорему объединения,* предложенную Дарвеном. Какие из перечисленных выше правил для этого потребуются? Какие правила можно вывести в виде частных случаев применения этой теоремы?

11.6. Сформулируйте определения следующих понятий:

а) замыкание множества функциональных зависимостей;

б) замыкание множества атрибутов для заданного множества функциональных зависимостей.

11.7. Перечислите множество всех функциональных зависимостей, которым должна соответствовать переменная отношения поставок SP.

11.8. Ниже приведено множество функциональных зависимостей, имеющих место для переменной отношения R{A,B,C,D,E,F,G}.

A → B
BC →
DE AEF
→ G

Вычислите замыкание $\{A,C\}_+$ функциональных зависимостей для данного множества. Следует ли из этого множества функциональная зависимость ACF → DG?

11.9. Сформулируйте определение понятия эквивалентности двух множеств функциональных зависимостей S1 и S2.

11.10. Сформулируйте определение понятия неприводимости множества функциональных зависимостей.

11.11. Определите, эквивалентны ли два приведенных ниже множества функциональны) зависимостей, установленных для переменной отношения R{A, B, C, D, E}.

1. A → B AB → C D → AC D → E
2. A → BC D → AE

11.12. Найдите неприводимое покрытие приведенного ниже множества функциональ ных зависимостей, заданных для переменной отношения R{ A, в, с, D, E, F}.

AB → C
C → A
BC → D
ACD→ B
BE → C
CE → FA
CF → BD
D → EF

11.13. В переменной отношения TIMETABLE определены перечисленные ниже атрибуты.

D День недели (1—5)
P Период времени в течение дня (1—6)
с Номер аудитории
T Имя преподавателя
L Название учебного предмета

Кортеж (d,p,c, t, 1) является элементом этой переменной отношения тогда и только тогда, когда лекция 1 проводится преподавателем t в аудитории с в момент времени (d,p) (при этом применяется упрощенная система обозначения кортежей, представленная в разделе 10.4). Предположим, что продолжительность всех лекций равна одному периоду времени и, кроме того, каждая лекция имеет название, уникальное по отношению ко всем лекциям за эту неделю. Какие функциональные зависимости соблюдаются для этой переменной отношения? Какие потенциальные ключи существуют для этой переменной отношения?

11.14. Пусть задана переменная отношения NADDR с атрибутами NAME (Уникальное имя), STREET (Улица), CITY (Город), STATE (Штат) и ZIP (Почтовый индекс). Предпо ложим, во-первых, что каждому почтовому индексу соответствует только один го род и штат, во-вторых, что каждой улице, городу и штату соответствует только один почтовый индекс. Найдите неприводимое множество функциональных зави симостей для этой переменной отношения. Какие потенциальные ключи сущест вуют для этой переменной отношения?

11.15. Соблюдаются ли на практике предположения, принятые в предыдущем упраж нении?

11.16. Пусть дана переменная отношения R с атрибутами A, в, с, D, E, F, G, н, I и J, для ко торой выполняется приведенное ниже множество функциональных зависимостей.

ABD →
E AB →
G B →
F

c → J
CJ → I
G → H

Является ли это множество неприводимым? Какие потенциальные ключи существуют для данной переменной отношения?

СПИСОК ЛИТЕРАТУРЫ

Как было отмечено в разделе 11.1, данная глава является наиболее формальной во всей книге; в связи с этим автор счел приемлемым включить в этот список литературы ссылки **на [11.1], [11.3] и [11.10],** поскольку каждая из этих работ содержит формальное описание различных аспектов теории баз данных (а не только функциональных зависимостей как таковых).

11.1. Abiteboul S., Hull R., Vianu V. Foundations of Databases // Reading, Mass.: Addison-Wesley, 1995.

11.2. Armstrong W. W. Dependency Structures of Data Base Relationships // Proc. IFIP Congress. — Stockholm, Sweden, 1974.

Здесь впервые изложена формальная теория функциональных зависимостей (т.е. в этой работе впервые были опубликованы аксиомы Армстронга), а также приведена точная характеристика потенциальных ключей.

11.3. Atzeni P., De Antonellis V. Relational Database Theory // Redwood City, Calif.: Benjamin/Cummings, 1993.

11.4. Casanova M.A., Fagin R., Papadimitriou C.H. Inclusion Dependencies and Their Interaction with Functional Dependencies // Proc. 1st ACM SIGACT-SIGMOD Symposium on Principles of Database Systems. — Los Angeles, Calif. — March 1982.

Зависимости включения (INclusion Dependency— IND) являются обобщением концепции ограничений целостности. Например, следующая зависимость включения

SP.S# → S.S#

(для записи которой здесь используется система обозначений, отличная от используемой в данной статье) позволяет указать, что множество значений атрибута SP.S# должно быть подмножеством (не обязательно собственным подмножеством) множества значений атрибута S. S#. Это, конечно, частный случай ограничения ссылочной целостности, в общем же случае для зависимостей включения не существует никаких требований, согласно которым левая часть должна быть внешним ключом, а правая — потенциальным.

Примечание. Зависимости включения имеют некоторое сходство с функциональными зависимостями, поскольку оба эти типа зависимостей представляют связь "многие к одному". Однако зависимости включения обычно охватывают переменные отношения, тогда как функциональные зависимости их не охватывают.

В статье предложено непротиворечивое и полное множество правил вывода для зависимостей включения, которые могут быть (в несколько нестрогой форме) представлены следующим образом.

1. A → A.
2. ЕСЛИ AB → CD, ТО A → C И B → D.
3. Если A → B и B → C, то A → C.

11.5. Casey R.G., Delobel C. Decomposition of a Data Base and the Theory of Boolean Switching Functions// IBM J. R&D. — September 1973. — 17, № 5.

В работе показано, что для данной переменной отношения множество функциональных зависимостей (которые в этой статье называются *функциональными отношениями)* может быть представлено как *булева переключательная функция.* **Кроме** того, отмечено, что эта функция в некотором смысле уникальна, поскольку исходные функциональные зависимости могут быть заданы многими с виду разными (но по сути эквивалентными) способами, каждый из которых в общем приводит к возникновению с виду разных булевых функций, но все такие функции могут быть сведены к одной той же канонической форме с помощью законов булевой алгебры (см. главу 18). Показано, что проблема *декомпозиции* исходной переменной отношения (т.е. декомпозиция без потерь; подробности приводятся в главе 12) логически эквивалентна хорошо известной в булевой алгебре проблеме поиска *покрывающего множества простых импликант* для булевой функции, соответствующей этой переменной отношения вместе с ее функциональными зависимостями. Следовательно, исходная задача может быть переформулирована в эквивалентную задачу булевой алгебры, а для ее решения могут применяться хорошо известные методы.

Это одна из первых работ, в которой проводятся аналогии между теорией зависимостей и другими научными дисциплинами. Этой теме посвящена также публикация [11.8] и несколько работ, упомянутых в списках литературы в главе 13.

11.6. Codd E.F. Further Normalization of the Data Base Relational Model // Rustin R.J. (ed.), Data Base Systems, Courant Computer Science Symposia Series 6. — Englewood Cliffs, N.J.: Prentice-Hall, 1972.

В этой работе впервые представлена концепция функциональной зависимости (не считая более раннего упоминания в одном из внутренних документов фирмы IBM, который также был подготовлен Коддом). Термин *дальнейшая нормализация* (Further Normalization), приведенный в заголовке, относится к некоторому конкретному подходу к проектированию базы данных, рассматриваемому в главе 12 (назначение этой статьи заключалось в демонстрации возможности использования идеи функциональной зависимости для решения проблем проектирования базы данных). Действительно, функциональные зависимости представляют собой одну из первых научно обоснованных попыток решения этой проблемы. Но как отмечено в разделе 11.1, с тех пор идея ФЗ доказала свою полезность и для использования в более широкой области (см., например, [11.7]).

11.7. Darwen H. The Role of Functional Dependence in Query Decomposition // Date C.J., Darwen H. Relational Database Writings 1989—1991.— Reading, Mass.: Addison-Wesley, 1992.

В этой работе представлен ряд правил логического вывода для функциональных зависимостей, с помощью которых может быть доказана выполнимость функциональных зависимостей для данной произвольной переменной отношения исходя

из выполнимости этих зависимостей в переменной отношения (или нескольких переменных отношения), на основе которой получена данная переменная отношения. Сформированное таким образом множество ФЗ может использоваться для определения потенциальных ключей для некоторой производной переменной отношения. В результате можно получить правила наследования для *потенциальных ключей,* кратко упомянутые в главах 9 и 10 как "наиболее желательные". В статье показано, как эти правила наследования для функциональных зависимостей и потенциальных ключей могут использоваться для значительного повышения производительности СУБД, расширения ее функциональных возможностей и обеспечения удобства пользования.

Примечание. Подобные правила используются в стандарте SQL: 1999, во-первых, для небольшого расширения перечня представлений, которые рассматриваются как обновляемые (см. главу 10), а во-вторых, для расширения в этом стандарте трактовки того, что подразумевается под определением выражения, "однозначного в пределах каждой группы" (например, в конструкции SELECT).

В качестве иллюстрации ко второму из указанных пунктов можно привести следующий запрос.

```
SELECT S.S#, S.CITY, SUM ( SP.QTY ) AS TQ
FROM S, SP
WHERE S.S# = SP.S#
GROUP BY S.S# ;
```

Этот запрос был недопустимым по стандарту SQL: 1992, поскольку атрибут s. CITY упоминается в конструкции SELECT, но не в конструкции GROUP BY, но является допустимым по стандарту SQL: 1999, поскольку машина обработки запросов SQL теперь способна определить, что переменная отношения S удовлетворяет функциональной зависимости s# → CITY.

11.8. Fagin R. Functional Dependencies in a Relational Database and Propositional Logic // **IBM** J.R&D. - November 1977. - 21, № 6.

Показано, что аксиомы Армстронга [11.2] строго эквивалентны системе импликационных утверждений в логике высказываний. Иначе говоря, задается отображение между функциональными зависимостями и утверждениями в логике высказываний, а затем демонстрируется, что данная зависимость f является следствием из заданного множества зависимостей S тогда и только тогда, когда высказывание, соответствующее f, является логическим следствием множества высказываний, соответствующих множеству S.

11.9. Lucchesi C.L., Osborn S.L. Candidate Keys for Relations // J. Cотр. and Sys. Sciences. — 1978. — 17, № 2.

Представлен алгоритм поиска всех потенциальных ключей для заданной переменной отношения, если дано множество функциональных зависимостей, которые соблюдаются в этой переменной отношения.

11.10. Maier D. The Theory of Relational Databases // Rockville, Md.: Computer Science Press, 1983.

ГЛАВА 12

Дальнейшая нормализация: формы 1НФ, 2НФ, 3НФ и НФБК

12.1. ВВЕДЕНИЕ

До сих пор в этой книге в качестве примера рассматривалась база данных поставщиков и деталей с приведенной ниже логической структурой.

```
S  { S#, SNAME,
   STATUS, CITY }
   PRIMARY KEY { S# }

P  { P#, PNAME, COLOR,
   WEIGHT, CITY } PRIMARY KEY
   { P# }

SP { S#, P#, QTY }
   PRIMARY KEY { S#, P# }
   FOREIGN KEY { S# } REFERENCES S
   FOREIGN KEY { P# } REFERENCES P
```

Примечание. В настоящей главе (если не указано иное) предполагается, что переменные отношения всегда имеют конкретный первичный ключ. Примерами таких переменных отношения могут служить приведенные выше определения.

На первый взгляд этот проект базы кажется вполне приемлемым. И действительно, в нем предусмотрены три переменные отношения (s, P и SP), необходимые для представления всей рассматриваемых данных, а в переменные отношения включены подходящие атрибуты. В частности, кажется "вполне очевидным", что атрибут CITY для города поставщика должен быть определен в переменной отношения S, атрибут COLOR для цвета детали — в переменной отношения P, атрибут QTY для количества деталей — в переменной отношения SP и т.д. Но на чем основана такая уверенность? Причины, которыми обусловлен выбор именно такого проекта базы данных, можно понять, изменив его определенным образом. Предположим, например, что атрибут CITY удален из переменной отношения поставщиков S и добавлен в переменную отношения поставок SP. (Однако интуитивно это действие воспринимается как ошибочное, поскольку понятие "город поставщика" очевидным образом связано с поставщиками, а не с поставками.) На рис. 12.1 представлен пример содержимого переменной отношения поставок, измененной подобным образом (исходный вариант приведен на рис. 11.1 в главе 11).

Примечание. Чтобы избежать путаницы, связанной с исходной переменной отношения SP, которой мы оперировали ранее, эта измененная переменная отношения будет далее обозначаться SCP, как и в главе 11.

На рис. 12.1 можно легко заметить существенный недостаток этого проекта — избыточность. А именно, в каждом кортеже переменной отношения SCP для поставщика с номером S1 содержится информация о том, что этот поставщик находится в Лондоне; в каждом кортеже переменной отношения SCP для поставщика с номером S2 приведены данные о том, что этот поставщик находится в Париже, и т.д. Иначе говоря, сведения о городе, в котором находится конкретный поставщик, повторяются в отношении столько раз, сколько поставок выполняет данный поставщик. Эта избыточность, в свою очередь, приводит к некоторым другим проблемам. Например, после обновления данных[1] о местонахождении поставщика с номером S1 в одном из кортежей может быть указан Лондон, а в другом — Амстердам. Таким образом, для создания хорошего проекта следует придерживаться принципа "по одному факту в одном месте" (т.е. избегать избыточности данных). *Предметом нормализации, в сущности, становится всего лишь формализация подобных простых идей,* однако это должна быть формализация, которая действительно будет иметь большое практическое значение при проектировании базы данных.

Конечно, как уже упоминалось в главе 6, сами *отношения* в реляционной модели всегда нормализованы. Можно сказать, что *переменная отношения* также нормализована, поскольку ее допустимыми значениями являются нормализованные отношения. Следовательно, в

---

[1] Далее в этой и последующих главах нужно принять предположение (достаточно правдоподобное!) о том, что контроль предикатов переменных отношения поддерживается не в полном объеме. Это необходимо, поскольку в противном случае описанные выше проблемы просто не могли бы возникнуть (было бы невозможно обновить данные о городе поставщика с номером S1 только в некоторых кортежах). В действительности нормализацию целесообразно понимать следующим образом: она помогает спроектировать базу данных таким образом, чтобы сделать более логически приемлемыми операции обновления отдельных кортежей, что в противном случае (т.е. когда проект базы данных не нормализован) может оказаться затруднительным. Эта цель достигается благодаря тому, что в полностью нормализованном проекте предикаты переменных отношения имеют более простой вид.

контексте реляционной модели переменная отношения также всегда нормализована. Аналогично можно сказать, что переменные отношения (и значения отношения) всегда находятся в **первой нормальной форме,** или **1НФ.** Иначе говоря, понятия "нормализованная переменная отношения" и "переменная отношения в **1НФ"** означают *в точности одно и то же,* хотя следует иметь в виду, что понятие "нормализованная переменная отношения" может также относиться к нормализации более высоких уровней (обычно это выражение служит для обозначения *третьей* нормальной формы, или 3НФ). Последний вариант использования этого термина не совсем точен, но достаточно широко распространен.

SCP

S#	CITY	P#	QTY
S1	London	P1	300
S1	London	P2	200
S1	London	P3	400
S1	London	P4	200
S1	London	P5	100
S1	London	P6	100
S2	Paris	P1	300
S2	Paris	P2	400
S3	Paris	P2	200
S4	London	P2	200
S4	London	P4	300
S4	London	P5	400

Рис. 12.1. Пример значений данных в переменной отношения SCP

Однако некоторая переменная отношения может быть нормализованной в указанном смысле и все еще обладать определенными нежелательными свойствами. Примером может служить переменная отношения SCP, показанная на рис. 12.1. Принципы дальнейшей нормализации позволяют распознать подобные случаи и привести такие переменные отношения к более приемлемой форме. В случае переменной отношения SCP эти принципы позволили бы точно установить ее недостатки и указать на необходимость ее разбиения на две "более приемлемые" переменные отношения: одну с атрибутами S# и CITY, а другую с атрибутами s#, P# и QTY.

Нормальные формы

Процесс дальнейшей нормализации, который ниже будет упоминаться просто как *нормализация,* основывается на концепции **нормальных форм.** Говорят, что переменная отношения находится в определенной нормальной форме, если она удовлетворяет заданному набору условий. Например, переменная отношения находится во второй нормальной форме (или в 2НФ) тогда и только тогда, когда она находится в **1НФ** и удовлетворяет дополнительному условию, приведенному в разделе 12.3.

На рис. 12.2 показано несколько нормальных форм, которые определены к настоящему времени. Первые три (1НФ, **2НФ** и 3НФ) были описаны Коддом (Codd) [11.6]. Как видно из рис. 12.2, все нормализованные переменные отношения находятся в **1НФ,** некоторые переменные отношения в **1НФ** также находятся в 2НФ, и некоторые переменные отношения в **2НФ** также находятся в 3НФ. Мотивом, которым руководствовался Кодд при введении дополнительных определений, было то, что вторая нормальная форма "более желательна" (в смысле, который будет разъяснен ниже), чем первая, а третья, в

свою очередь, "более желательна", чем вторая. Таким образом, в общем случае при проектировании базы данных целесообразно использовать переменные отношения в третьей нормальной форме, а не в первой или второй.

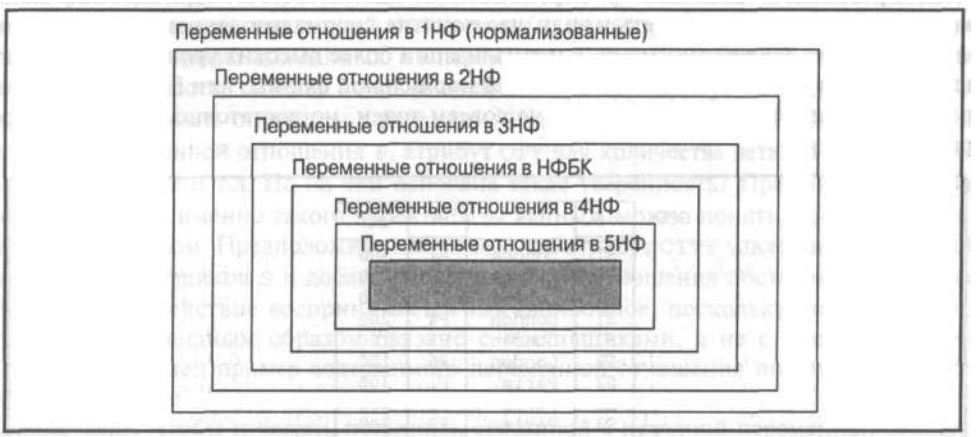


Рис. 12.2. Уровни нормализации

В [11.6] приведено также описание процедуры нормализации, с помощью которой переменная отношения в некоторой нормальной форме, например в 2НФ, может быть преобразована в несколько переменных отношения в другой, более желательной нормальной форме, например в ЗНФ. (В исходном варианте эта процедура определена только до третьей формы, но, как будет показано в следующей главе, она может быть последовательно расширена вплоть до пятой нормальной формы.) Такую процедуру можно охарактеризовать как *последовательное приведение заданного набора переменных отношения к некоторой все более желательной форме.* Следует отметить, что эта процедура **обратима,** т.е. всегда можно использовать ее результат (например, множество переменных отношения, находящихся в ЗНФ) для обратного преобразования (в исходную переменную отношения, находящуюся в 2НФ). Возможность выполнения обратного преобразования является весьма важной характеристикой, поскольку это означает, что в процессе нормализации **сохраняется информация (не происходит ее потеря).**

Возвращаясь к рассмотрению *собственно* нормальных форм, заметим, что, как будет показано в разделе 12.5, оригинальное определение Кодда для ЗНФ [11.6] приводит к некоторой неадекватности. Переработанное и более точное определение, приведенное Бойсом (Boyce) и Коддом в [12.2], является более строгим в том смысле, что любая переменная отношения в ЗНФ по новому определению является таковой и по старому определению, но не всякая переменная отношения в ЗНФ по старому определению может являться таковой по новому определению. Для того чтобы эти определения можно было различать, переменную отношения в ЗНФ по новому определению обычно называют **нормальной формой Бойса-Кодда — НФБК.**

Впоследствии Фейгином (Fagin) в [12.8] была определена новая, **четвертая нормальная форма (4НФ),** которая стала четвертой по счету, поскольку в момент ее создания нормальная форма Бойса-Кодда считалась третьей. Затем в [12.9] Фейгин дал определение еще одной нормальной формы, которую назвал **проекционно-соединительной нормальной**

формой (ПСНФ); ее называют также **петой нормальной формой** или **5НФ.** Как показано на рис. 12.2, некоторые переменные отношения в НФБК находятся также в 4НФ, а некоторые переменные отношения в 4НФ находятся также в 5НФ.

Возникает вопрос, будет ли продолжаться создание новых информационных структур для получения шестой, седьмой нормальной форм и так *до бесконечности.* Мы еще не готовы дать подробный ответ на этот интересный вопрос. Можно лишь уклончиво заметить, что действительно существуют дополнительные нормальные формы, которые не показаны на рис. 12.2, однако 5НФ можно рассматривать как "окончательную" нормальную форму в некотором (и очень важном) смысле. Мы вернемся к этой теме в главе 13.

Структура главы

Назначение данной главы — предоставить описание концепции дальнейшей нормализации до уровня нормальной формы Бойса— Кодда включительно. Оставшиеся формы будут рассмотрены в главе 13. В целом, материал будет излагаться по следующему плану. После этого несколько затянувшегося введения в разделе 12.2 описывается фундаментальная концепция *декомпозиции без потерь* и демонстрируется важное значение понятия *функциональной зависимости* (ФЗ) в этой концепции. (Действительно, понятие функциональной зависимости является основой выделения трех нормальных форм Кодда и нормальной формы Бойса-Кодца.) Далее, в разделе 12.3, подробно рассматриваются три начальные нормальные формы и на примере некоторой переменной отношения демонстрируется, как выполняется нормализация вплоть до достижения 3НФ. В разделе 12.4 будет сделано небольшое отступление в целях обсуждения вопроса *альтернативных декомпозиций,* т.е. проблемы выбора "наилучшей декомпозиции" для конкретной переменной отношения, если, конечно, такой выбор возможен. В разделе 12.5 обсуждается НФБК, а в разделе 12.6 рассматриваются особенности работы с атрибутами, принимающими отношения в качестве значений. Наконец, в разделе 12.7 приводится краткое резюме и дается несколько заключительных замечаний.

Важно отметить, что далее изложение ведется не столь строго, как раньше, и в своих рассуждениях автор в основном полагается на интуитивное понимание. Подобный подход может быть оправдан тем, что такие понятия, как "декомпозиция без потерь", "нормальная форма Бойса—Кодда" и другие, несмотря на их таинственные и загадочные названия, по сути весьма просты и общедоступны. Во многих работах, на которые даны ссылки в настоящей книге, этот материал излагается в более строгой форме. Хороший учебник можно найти в [12.5].

Кроме того, следует сделать еще два предварительных замечания.

1. Как уже отмечалось, общая идея *нормализации* заключается в том, что при проектировании базы данных считается более целесообразным определять переменные отношения в "окончательной" нормальной форме (т.е. в пятой). Однако эту рекомендацию не следует толковать как обязательное правило, поскольку возможны *{довольно часто)* ситуации, когда принципами нормализации приходится пренебрегать (об этом упоминается в упр. 12.7). Действительно, здесь необходимо подчеркнуть, что проектирование базы данных может представлять собой чрезвычайно сложную задачу. Нормализация значительно упрощает этот процесс, но не является панацеей. Хотя разработчику проекта базы данных рекомендуется знать основные принципы нормализации, это не означает, что проект обязательно

должен быть создан на основе исключительно этих принципов. В главе 14 обсуждается несколько других аспектов проектирования базы данных, которые имеют весьма отдаленное отношение к нормализации как таковой или совсем не имеют к ней отношения.

2. Как упоминалось выше, процедура нормализации будет использована в качестве основы при описании различных нормальных форм. Однако это не означает, что на практике создание проекта базы данных будет выполняться с помощью этой процедуры. На самом деле, вероятнее всего, для этого будет использована описанная в главе 14 схема нисходящего проектирования. Идеи нормализации можно использовать на последующих этапах для *проверки* того, что полученный в результате проект не нарушает, вопреки ожидаемому, каких-либо ее принципов. Как бы там ни было, процедура нормализации является удобным способом описания этих принципов. Поэтому для упрощения изложения в данной главе будет принято полезное допущение о том, что проектирование выполняется с помощью процедуры нормализации.

12.2. ДЕКОМПОЗИЦИЯ БЕЗ ПОТЕРЬ И ФУНКЦИОНАЛЬНЫЕ ЗАВИСИМОСТИ

Прежде чем приступить к рассмотрению процедуры нормализации, следует обсудить один существенный аспект этой процедуры, а именно — концепцию **декомпозиции без потерь.** Как уже упоминалось, процедура нормализации предусматривает разбиение, или *декомпозицию,* данной переменной отношения на другие переменные отношения, причем декомпозиция должна быть *обратимой,* т.е. выполняться без потерь информации. Иначе говоря, интерес представляют только те операции, которые выполняются без потерь информации. Вопрос о том, происходит ли утрата информации при декомпозиции, тесно связан с концепцией функциональной зависимости.

В качестве примера рассмотрим уже знакомую переменную отношения поставщиков S с атрибутами s#, STATUS и CITY (для упрощения задачи атрибут SNAME в данном случае игнорируется). На рис. 12.3 показан пример значений данных в этой переменной отношения и указаны два возможных варианта ее декомпозиции: *а* и *б.*

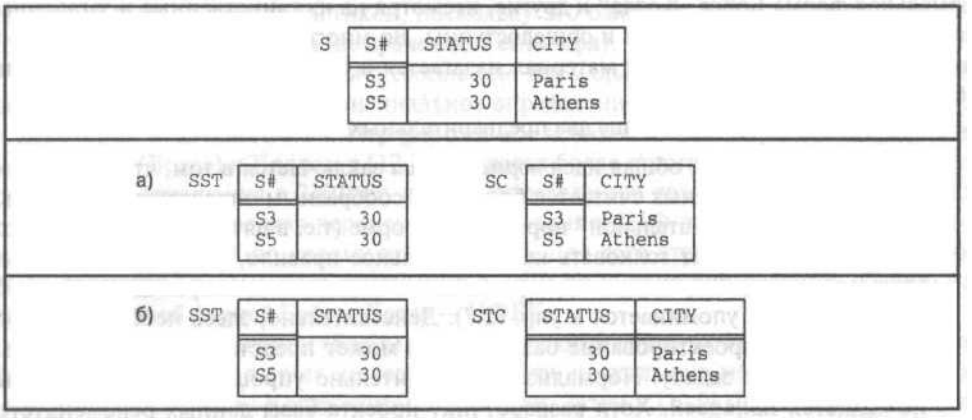


Рис. 12.3. Пример значения переменной отношения S и два возможных варианта ее

Внимательно ознакомившись с предложенными вариантами декомпозиции, можно заметить две особенности.

1. В случае *а* информация не утрачивается, поскольку переменные отношения SST и *SC* все еще содержат данные о том, что поставщик с номером S3 имеет статус 30 и находится в Париже (Paris), а поставщик с номером S5 имеет статус 30 и нахо дится в Афинах (Athens). Иначе говоря, первая декомпозиция действительно яв ляется декомпозицией без потерь.
2. В случае *б,* напротив, некоторая информация утрачивается, поскольку оба по ставщика имеют статус 30, но при этом нельзя сказать, в каком городе находится каждый из них. Иначе говоря, вторая декомпозиция не является декомпозицией без потерь.

Почему же получилось так, что одна декомпозиция была проведена без потери, а другая — с потерей информации? Прежде всего следует отметить, что процесс, который до сих пор назывался декомпозицией, на самом деле является операцией **проекции,** т.е. каждая из показанных на данном рисунке переменных отношения — SST, SC и STC — в действительности является проекцией исходной переменной отношения S. Таким образом, оператор декомпозиции в этой процедуре нормализации фактически является оператором *проекции.*

Примечание. Как и в части II этой книги, высказывание типа "SST является проекцией переменной отношения S" используется вместо более точного высказывания типа "SST является переменной отношения, значение которой в любой момент времени является проекцией значения переменной отношения S, которое она имеет в это же время". Надеемся, что использование указанной сокращенной формулировки не приведет к путанице.

Обратите внимание, что в случае *а* сохранение информации в полном объеме означает, что *при обратном соединении переменных отношения* SST *и* SC *будет получена исходная переменная отношения* S. В случае *б* дело обстоит иначе, поскольку при обратном соединении переменных отношения SST и SC исходная переменная отношения S получена *не* будет, а это значит, что некоторая информация будет утрачена[2]. Иначе говоря, "обратимость" означает, что для получения исходной переменной отношения достаточно применить к ее проекциям операцию соединения, т.е. *исходная переменная отношения* равна соединению ее проекций. Если операцией декомпозиции в процедуре нормализации является операция проекции, то обратной операцией (назовем ее *рекомпозицией)* должна быть операция **соединения.**

Исходя из сказанного выше, можно задать следующий интересный вопрос. Пусть R1 и R2 являются проекциями некоторой переменной отношения R, содержащими все атрибуты переменной отношения R. Какие условия должны быть соблюдены для того, чтобы при обратном соединении проекций R1 и R2 можно было гарантировать получение

---

[2] Точнее, в исходной переменной отношения S вместе со всеми кортежами будут содержаться "фиктивные" кортежи, поскольку при обратной операции никогда не удастся получить переменную отношения, которая была бы *меньше* исходной переменной отношения S. *{Упражнение.* Попробуйте доказать это утверждение.) А поскольку не существует общего метода различения фиктивных и подлинных кортежей, информация в этом случае действительно будет утеряна.

исходной переменной отношения R? Именно для получения ответа на этот вопрос необходимо обратиться к функциональным зависимостям. В рассматриваемом примере переменная отношения S удовлетворяет представленному ниже неприводимому множеству функциональных зависимостей.

S# →
STATUS S#
→ CITY

Учитывая данный факт, согласно которому переменная отношения S удовлетворяет приведенным функциональным зависимостям, нельзя считать простым совпадением то, что эта переменная отношения равна соединению своих проекций по атрибутам {S#, STATUS} и {S#,CITY}. И то, что это не совпадение, а закономерность, подтверждается теоремой Хита (Heath) [12.4].

- **Теорема Хита.** Пусть R{A, в, C} является переменной отношения, где A, в и с — множества атрибутов этой переменной отношения. Если R удовлетворяет функциональной зависимости A→ в, то R равна соединению ее проекций по атрибутам {A,B} И {A,C}.

Если принять, что A— это атрибут s#, в — это атрибут STATUS, а с — это атрибут CITY, то данная теорема подтверждает, как отмечалось выше, что переменная отношения S может быть разбита с помощью операции декомпозиции на проекции по атрибутам (S#, STATUS} и {S#,CITY} без потери информации. В то же время уже известно, что переменная отношения S *не может* быть разбита без потери информации на проекции по атрибутам {S#, STATUS} и {STATUS,CITY}. Теорема Хита не дает объяснения, почему так происходит[3]. Однако интуитивно ясно, что *при такой декомпозиции утрачивается одна из функциональных зависимостей,* т.е. функциональная зависимость S# → STATUS все еще представлена (благодаря проекции по атрибутам {S#, STATUS}), а функциональная зависимость S# → CITY утрачена.

В заключение можно отметить, что декомпозиция переменной отношения R на проекции R1, R2, ..., Rn выполняется **без потерь,** если R равна соединению R1, R2, ..., Rn.

Примечание. Вероятно, с точки зрения практики следовало бы выдвинуть дополнительное требование, чтобы в соединении обязательно были нужны все проекции R1, R2, ..., Rn. Это позволяет гарантировать, что удастся избежать определенной избыточности, которая могла бы возникнуть в ином случае. Например, вряд ли стоит рассматривать декомпозицию переменной отношения S на проекции (скажем) по атрибутам {S#}, {S#, STATUS} и {S#,CITY} как качественную декомпозицию без потерь, хотя S в конечном итоге становится равной соединению этих трех проекций. Для простоты в дальнейшем будем считать, что это дополнительное требование всегда остается в силе (если явно не указано иное).

---

[3] Дело в том, что эта теорема сформулирована в выражениях "если..., то...", а не "тогда и только тогда, когда..." (см. упр. 12.1, приведенное в конце главы). Более строгая формулировка теоремы Хита будет представлена в разделе 13.2 главы 13.

Дополнительные сведения о функциональных зависимостях

В завершение перечислим некоторые дополнительные замечания, касающиеся функциональных зависимостей.

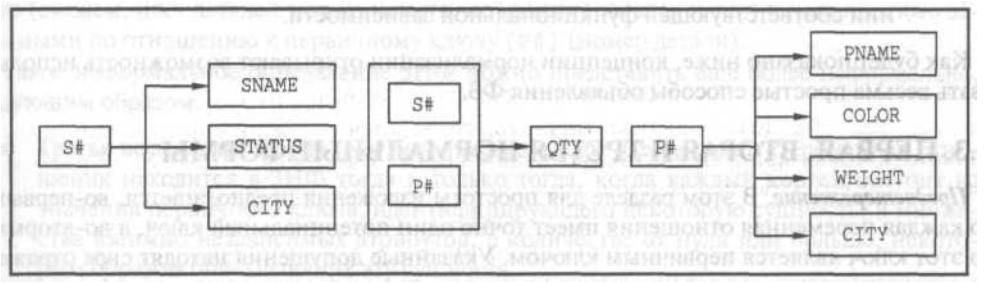
1. **Неприводимость.** Как указано в главе 11, функциональная зависимость называете; **неприводимой слева,** если ее левая часть "не слишком велика". Рассмотрим, например, *переменную отношения SCP, приведенную в разделе 12.1, которая удовлетворяет* следующей функциональной зависимости.

   { S#, P# } → CITY

   Однако атрибут p# в левой части этой функциональной зависимости является избыточным, поэтому она может быть переписана в следующем виде.

   S# → CITY

   (Иначе говоря, атрибут CITY функционально зависит только от s#.) Последняя функциональная зависимость является неприводимой слева, а предыдущая — нет. Наряду с этим, можно сказать, что атрибут CITY является неприводимо зависимым[4] от атрибута S#, но не является неприводимо зависимым от множества атрибутов {s#, P#}. Неприводимые слева ФЗ и неприводимые ФЗ играют важную роль при определении второй и третьей нормальной форм (подробности приведены в разделе 12.3).
2. **Диаграммы** ФЗ. Пусть дана переменная отношения R и пусть к ней применимо некоторое неприводимое множество функциональных зависимостей I (более подробные сведения о неприводимых множествах ФЗ приводятся в главе 11). Удобнее всего можно представить это множество I в виде *диаграммы функциональных зависимостей* (диаграммы ФЗ). Например, на рис. 12.4 показаны вполне очевидные по смыслу диаграммы функциональных зависимостей, соответственно, для переменных отношения s, SP и P. Такие диаграммы будут часто использоваться далее в этой главе.



Рис, 12.4. Диаграмма функциональных зависимостей для переменных отношения S, SP и P

[4] Здесь термины "неприводимая слева ФЗ" и "неприводимо зависимый" используются вместо терминов **"полная ФЗ"** и **"полностью** зависимый", которые часто можно встретить в литературе (и в прежних изданиях этой книги). Хотя последние термины отличаются краткостью, они менее информативны и не очень удобны.

Как можно видеть, на рис. 12.4 *каждая стрелка начинается с потенциального ключа* (в действительности — с первичного ключа) соответствующей переменной отношения. По определению стрелки должны начинаться с каждого потенциального ключа[5], поскольку одному значению такого ключа всегда соответствует еще по крайней мере одно какое-либо значение; такие стрелки нельзя удалять ни при каких условиях. *Если же на диаграмме имеются какие-то другие стрелки, то возникают сложности.* Таким образом, процедуру нормализации можно довольно неформально охарактеризовать как процедуру исключения стрелок, которые не начинаются с потенциальных ключей.

3. ФЗ как семантическое понятие. Как было отмечено в главе 11, функциональные зависимости безусловно представляют собой особый вид ограничений целостности. К тому же они, несомненно, относятся к категории *семантических* понятий (фактически функциональные зависимости входят в состав определения предиката переменной отношения). Выявление функциональных зависимостей представляет собой часть процесса выяснения *смысла* тех или иных данных. Например, тот факт, что переменная отношения S удовлетворяет функциональной зависимости S# → CITY, по сути означает, что каждый поставщик находится точно в одном городе. Иначе эту ситуацию можно охарактеризовать следующим образом.
   - В реальном мире существует некоторое ограничение, представленное в этой базе данных, согласно которому каждый поставщик находится точно в одном городе.
   - Поскольку это ограничение является частью семантического описания пред метной области, оно должно быть каким-то образом представлено в базе дан ных.
   - Один из способов обеспечения этого состоит в том, что указанное ограничение необходимо задать в определении базы данных таким образом, чтобы оно мог ло быть предписано с помощью средств СУБД.
   - Способ описания ограничения в определении базы данных состоит в объявле нии соответствующей функциональной зависимости.

Как будет показано ниже, концепции нормализации открывают возможность использовать весьма простые способы объявления ФЗ.

12.3. ПЕРВАЯ, ВТОРАЯ И ТРЕТЬЯ НОРМАЛЬНЫЕ ФОРМЫ

Предостережение. В этом разделе для простоты изложения предполагается, во-первых, что каждая переменная отношения имеет точно один потенциальный ключ, а во-вторых, что этот ключ является первичным ключом. Указанные допущения находят свое отражение в приведенных в данной главе определениях, которые (как уже отмечалось) являются не очень строгими. Далее, в разделе 12.5, будет рассмотрен случай, когда переменная отношения имеет больше одного потенциального ключа.

---

[5] Точнее, стрелки должны начинаться с *суперключей.* Но если множество функциональных зависимостей I является неприводимым, как указано выше, то все функциональные зависимости (или "стрелки") в I будут неприводимыми слева.

Прежде чем перейти к подробному описанию трех нормальных форм, первоначально предложенных Коддом, следует дать предварительное и весьма неформальное определение третьей нормальной формы для того, чтобы в общих чертах обрисовать основную цель изложения. Затем будет рассмотрена процедура приведения произвольной переменной отношения к эквивалентному набору переменных отношения в ЗНФ. Попутно будут даны более точные определения первых трех нормальных форм. Однако в качестве отступления следует отметить, что формы 1НФ, 2НФ и ЗНФ сами по себе не имеют особо важного значения и должны рассматриваться лишь как промежуточные этапы на пути к созданию формы НФБК (и форм более высокого уровня).

Итак, ниже приведено предварительное определение ЗНФ.

- **Третья нормальная форма** *{очень неформальное определение).* Переменная отношения находится в ЗНФ тогда и только тогда, когда ее неключевые атрибуты (если они вообще существуют) являются одновременно:

  а) взаимно независимыми; и

  б) неприводимо зависим ыми от первичного ключа.

  Ниже приведены (неформальные) определения понятий *неключевые атрибуты* и *взаимно независимые атрибуты.*

- *Неключевой атрибут* — это атрибут, который не входит в состав первичного ключа рассматриваемой переменной отношения.
- Два или больше атрибутов называются *взаимно независимыми,* если ни один из них функционально не зависит от какой-либо комбинации остальных атрибутов. Подобная независимость подразумевает, что каждый такой атрибут может обновляться независимо от остальных атрибутов.

Например, согласно приведенному выше определению, переменная отношения P (переменная отношения деталей) находится в ЗНФ, а именно, атрибуты PNAME (название детали), COLOR (цвет), WEIGHT (вес) и CITY (город) являются независимыми друг от друга (скажем, цвет деталей можно менять, не изменяя их веса и т.д.) и неприводимо зависимыми по отношению к первичному ключу {P#} (номер детали).

Такое неформальное определение ЗНФ можно представить еще более неформально, следующим образом.

- **Третья нормальная форма** *{еще более неформальное определение).* Переменная отношения находится в ЗНФ тогда и только тогда, когда каждый кортеж состоит из значения первичного ключа, идентифицирующего некоторую сущность, и множества взаимно независимых атрибутов, в количестве от нуля или больше, некоторым образом описывающих эту сущность.

К переменной отношения P применимо и это определение, поскольку каждый кортеж переменной отношения P состоит из значения первичного ключа (номер детали), идентифицирующего в реальном мире некоторую деталь, и четырех дополнительных взаимно независимых значений (название, цвет, вес детали и город, в котором она хранится), описывающих отдельные свойства детали.

Теперь можно вернуться к описанию процедуры нормализации и дать определение первой нормальной формы.

- **Первая нормальная форма.** Переменная отношения находится в 1НФ тогда и только тогда, когда в любом допустимом значении этой переменной отношения каждый ее кортеж содержит только одно значение для каждого из атрибутов.

Фактически в этом определении всего лишь утверждается, что все переменные отношения всегда находятся в 1НФ, что, несомненно, верно. Однако переменная отношения, которая находится *только* в **1НФ** (т.е. не находится ни во второй, ни в третьей нормальной форме), обладает структурой, не совсем приемлемой по некоторым причинам. Для иллюстрации этого факта допустим, что информация о поставщиках и поставках содержится не в двух переменных отношения S и SP, а в одной, имеющей следующую структуру.

```
FIRST { S#, STATUS, CITY, P#,
        QTY } PRIMARY KEY { S#,
        P# }
```

Это — расширенная версия переменной отношения SCP, приведенной выше, в разделе 12.1. Ее атрибуты имеют тот же смысл, что и раньше, за исключением следующего дополнительного ограничения, специально введенного в этом примере.

CITY → STATUS

(Иными словами, атрибут STATUS функционально зависим от атрибута CITY, и смысл этого ограничения состоит в том, что статус поставщика определяется его местонахождением, например, все поставщики из Лондона *должны* иметь статус 20.) Кроме того, для упрощения атрибут SNAME снова игнорируется. Первичным ключом переменной отношения FIRST является комбинация атрибутов {S#, P#}, а диаграмма ее функциональных зависимостей будет иметь вид, представленный на рис. 12.5.

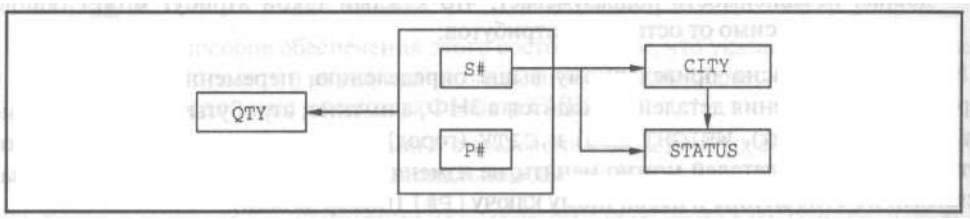


Рис. 12.5. Функциональные зависимости в переменной отношения FIRST

Обратите внимание, что эта диаграмма функциональных зависимостей с виду "сложнее" аналогичной диаграммы для переменной отношения в ЗНФ. Как было сказано в предыдущем разделе, на диаграмме ФЗ для переменной отношения в ЗНФ стрелки всегда выходят *только* из потенциальных ключей, тогда как на диаграмме ФЗ для переменной отношения, которая не находится в ЗНФ (например, на диаграмме ФЗ для переменной отношения FIRST), есть стрелки, начинающиеся с потенциальных ключей, и *дополнительные стрелки,* которые и являются источником затруднений. Фактически в переменной отношения FIRST нарушаются оба условия, указанные в приведенном выше определении ЗНФ: не все неключевые атрибуты взаимно независимы, поскольку атрибут STATUS зависит от атрибута CITY (одна дополнительная стрелка), и не все неключевые атрибуты неприводимо зависимы от первичного ключа, поскольку атрибуты STATUS и CITY, каждый в отдельности, зависимы от атрибута s# (еще две дополнительные стрелки).

Для иллюстрации некоторых трудностей, связанных с устранением этих дополнительных стрелок, на рис. 12.6 приведен пример данных в переменной отношения FIRST. Это тот же набор значений, который обычно используется нами в примерах, но значение 30 статуса поставщика с номером S3 заменено значением 10 в соответствии с новым ограничением, согласно которому значение атрибута CITY определяет значение атрибута STATUS. Возникшая в результате избыточность данных вполне очевидна, поскольку в каждом кортеже для поставщика с номером S1 атрибут CITY имеет значение London и, кроме того, в каждом кортеже со значением London в атрибуте CITY указано значение 20 для атрибута STATUS.

Избыточность в переменной отношения FIRST приводит к разным **аномалиям обновления,** получившим такое название по историческим причинам. Под этим подразумеваются определенные трудности, появляющиеся при выполнении операций обновления INSERT, DELETE и UPDATE. Для начала рассмотрим избыточность данных "поставщик-город", соответствующих функциональной зависимости S# → CITY. Ниже описаны проблемы, которые возникнут при выполнении каждой из указанных операций обновления.

- **Операция INSERT.** Нельзя поместить в переменную отношения FIRST информацию о том, что некоторый поставщик находится в определенном городе, до тех пор пока этот поставщик не выполнит поставку хотя бы одной детали. Действительно, **в** таблице на рис. 12.6 нет сведений о поставщике из Афин с номером S5, поскольку до тех пор, пока этот поставщик не начнет поставку какой-либо детали, для него невозможно будет сформировать значение первичного ключа. (Как и в разделе 10.4 главы 10, в настоящей главе принято достаточно обоснованное предположение, что атрибуты первичного ключа не могут иметь значений, применяемых по умолчанию.)

FIRST

S#	STATUS	CITY	P#	QTY
S1	20	London	P1	300
S1	20	London	P2	200
S1	20	London	P3	400
S1	20	London	P4	200
S1	20	London	P5	100
S1	20	London	P6	100
S2	10	Paris	P1	300
S2	10	Paris	P2	400
S3	10	Paris	P2	200
S4	20	London	P2	200
S4	20	London	P4	300
S4	20	London	P5	400

Рис. 12.6. Пример данных в переменной отношения FIRST

- **Операция DELETE.** Если из переменной отношения FIRST удалить кортеж, который является единственным для некоторого поставщика, будет удалена не только информация о поставке поставщиком некоторой детали, но также информация о том, что этот поставщик находится в определенном городе. Например, если из переменной отношения FIRST удалить кортеж со значением S3 в атрибуте s# и

значением P2 в атрибуте P#, будет утрачена информация о том, что поставщик с номером S3 находится в Париже. (По сути, проблемы удаления и вставки представляют собой две стороны одной медали.)

Примечание. **В** действительности проблема заключается в том, что в переменной отношения FIRST *содержится слишком много информации,* собранной в одном месте, потому и *теряется слишком много информации* при удалении некоторых кортежей. Точнее говоря, переменная отношения FIRST одновременно содержит информацию и о поставках, и о поставщиках. В результате удаление информации о поставке вызывает также удаление информации о поставщике. Для решения этой проблемы необходимо разделить информацию на несколько частей, т.е. собрать сведения о поставках в одной переменной отношения, а о поставщиках — в другой (именно это и будет сделано чуть ниже). Следовательно, процедуре нормализации можно дать еще одно неформальное определение: охарактеризовать ее как процеДУРУ *разбиения* логически несвязанной информации на отдельные переменные отношения.

- **Операция UPDATE.** В общем, название города для каждого поставщика повторяется в переменной отношения FIRST несколько раз, и эта избыточность приводит к возникновению проблем при обновлении. Например, если поставщик с номером S1 переместится из Лондона в Амстердам, то *необходимо* будет отыскать в переменной отношения FIRST все кортежи, в которых значения S1 и London связаны между собой (для внесения соответствующих изменений), *иначе* база данных окажется в противоречивом состоянии (в одних кортежах в качестве города, в котором находится поставщик с номером S1, будет указан Лондон, а в других— Амстердам).

Для решения всех этих проблем, как предлагалось выше, необходимо заменить переменную отношения FIRST двумя следующими переменными отношения.

```
SECOND { S#, STATUS, CITY
} SP     { S#, P#, QTY }
```

Диаграммы функциональных зависимостей для этих двух переменных отношения показаны на рис. 12.7, а примеры значений данных — на рис. 12.8. Обратите внимание, что теперь в них имеется и информация о поставщике с номером S5 (в переменной отношения SECOND, но не в переменной отношения SP). Фактически теперь переменная отношения SP точно совпадает с нашей обычной переменной отношения поставок.

Легко убедиться, что измененная подобным образом структура данных позволяет преодолеть все перечисленные выше проблемы, связанные с выполнением операций обновления.

- **Операция INSERT.** Теперь информацию о том, что поставщик с номером S5 нахо дится в Афинах, можно поместить в базу данных, вставив соответствующий кортеж в переменную отношения SECOND, причем даже в том случае, если он в настоящее время не поставляет никаких деталей.
- **Операция DELETE.** Теперь вполне можно исключить информацию о поставке, в ко торой собраны сведения о поставщике с номером S3 и о детали с номером P2. Доста точно удалить соответствующий кортеж из переменной отношения SP, причем ин формация о том, что поставщик с номером S3 находится в Париже, не теряется.

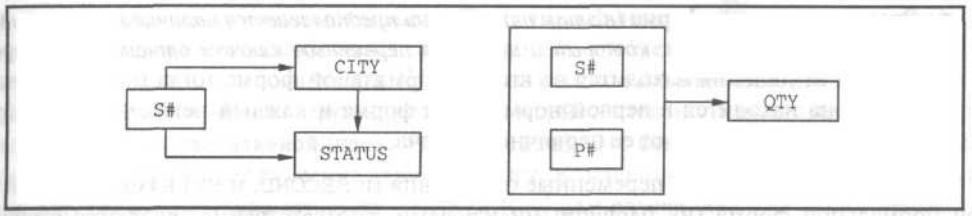


Рис. 12.7. Функциональные зависимости в переменных отношения SECOND и SP

SECOND

S#	STATUS	CITY
S1	20	London
S2	10	Paris
S3	10	Paris
S4	20	London
S5	30	Athens

Аналог этой строки на рис. 12.6 отсутствует

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Рис. 12.8. Примеры значений данных в переменных отношения SECOND и SP

- **Операция UPDATE.** В переработанной структуре название города для каждого поставщика указывается всего один раз, поскольку существует только один кортеж для данного поставщика в переменной отношения SECOND (первичным ключом этой переменной отношения является атрибут {S#}). Иначе говоря, избыточность данных S#-CITY устранена. Благодаря этому теперь можно изменить название города Лондон для поставщика с номером si на Амстердам, не рискуя привести базу данных в несогласованное состояние, поскольку достаточно изменить название города в соответствующем кортеже переменной отношения SECOND.

Сравнивая рис. 12.5 и 12.7, можно заметить, что суть разбиения переменной отношения FIRST на переменные отношения SECOND и SP состояла в исключении зависимостей, которые не являлись неприводимыми. Именно благодаря этому в новом варианте удается избежать упомянутых ранее трудностей. Интуитивно ясно, что в переменной отношения FIRST атрибут CITY описывал не сущность, которая идентифицируется первичным ключом (поставка), а *поставщика,* выполнявшего эту поставку (аналогичное утверждение можно сделать и об атрибуте STATUS). Смешивание этих двух типов информации в одной переменной отношения и было основной причиной возникновения описанных выше проблем.

Теперь можно дать определение второй нормальной формы[6].

[6] Строго говоря, 2НФ может быть определена только по отношению к заданному множеству зависимостей, но в неформальном контексте эта особенность обычно игнорируется. Аналогичные замечания применимы также ко всем нормальным формам (кроме, конечно же, первой нормальной формы).

- Вторая нормальная форма *(в этом определении предполагается наличие только одного потенциального ключа, который и является первичным ключом отношения).* Переменная отношения находится во второй нормальной форме тогда и только тогда, когда она находится в первой нормальной форме и каждый неключевой атрибут неприводимо зависит от ее первичного ключа.

Обе вновь образованные переменные отношения (и SECOND, и SP) находятся во второй нормальной форме (их первичными ключами, соответственно, являются атрибут {S#} и комбинация атрибутов {S#, p#}), тогда как переменная отношения FIRST не находится в этой форме. Всякую переменную отношения, которая находится в первой нормальной форме, но не находится во второй, всегда можно свести к эквивалентному множеству переменных отношения, находящихся в 2НФ. Этот процесс заключается в замене переменной отношения в 1НФ подходящим набором проекций, эквивалентных исходной переменной отношения, в том смысле, что при необходимости ее всегда можно будет восстановить с помощью обратной операции соединения данных проекций. В нашем примере переменные отношения SECOND и SP — это проекции[7] переменной отношения FIRST, а переменная отношения FIRST является соединением переменных отношения SECOND и SP по атрибуту S#.

Таким образом, первый этап процедуры нормализации состоит в создании проекций, которые позволяют исключить функциональные зависимости, не являющиеся неприводимыми. Пусть дана переменная отношения R, имеющая следующий вид.

```
R { A, B, C, D }
  PRIMARY KEY { A, B }
  /* Предполагается наличие функциональной зависимости A → D */
```

Процедура нормализации предусматривает замену этой переменной отношения следующими двумя проекциями, R1 и R2.

```
R1 { A, D }
     PRIMARY KEY { A }

R2 { A, B, C }
     PRIMARY KEY { A, B }
FOREIGN KEY { A } REFERENCES R1
```

Переменная отношения R всегда может быть восстановлена посредством соединения переменных отношения R1 и R2 по внешнему ключу и соответствующему ему первичному ключу этих переменных отношения.

Вернемся к рассматриваемому примеру. Следует отметить, что выбранная структура переменных отношения SECOND и SP все еще может вызвать некоторые проблемы. Структура переменной отношения SP вполне удовлетворительна, поскольку она фактически находится в 3НФ. Поэтому мы больше не будем уделять ей внимание до конца данного раздела. Однако в переменной отношения SECOND неключевые атрибуты все

---

[7] Если не учитывать того факта, что переменная отношения SECOND может содержать дополнительные кортежи (например, кортеж для поставщика с номером S5), которые не будут иметь аналога в переменной отношения FIRST (см. рис. 11.8). Иначе говоря, новая структура может содержать информацию, которую невозможно будет представить в исходной структуре. В этом смысле новую структуру можно рассматривать как более адекватное представление реального мира.

еще не являются взаимно независимыми. Диаграмма функциональных зависимостей для нее по-прежнему имеет вид более сложный, чем это требуется для диаграммы ФЗ переменной отношения, находящейся в 3НФ. В частности, зависимость атрибута STATUS от атрибута s#, хотя и *является* функциональной и действительно неприводимой, одновременно является **транзитивной** (через атрибут CITY). Это означает, что каждое значение атрибута s# определяет значение атрибута CITY, а значение атрибута CITY в свою очередь определяет значение атрибута STATUS. В общем случае, как уже было показано в главе 11, если имеют место две функциональные зависимости, A → в и в → с, то имеет место и транзитивная функциональная зависимость A → C. Однако наличие транзитивных зависимостей может привести к возникновению описанных ниже аномалий обновления. (В данном случае основное внимание будет сосредоточено на избыточности данных "город—статус", соответствующей функциональной зависимости CITY → STATUS.)

- **Операция INSERT.** Нельзя поместить в базу данных сведения о том, что определенный город характеризуется некоторым статусом (например, нельзя указать, что все поставщики из Рима должны иметь статус 50), до тех пор, пока в этом городе не появится конкретный поставщик.
- **Операция DELETE.** При удалении из переменной отношения SECOND кортежа для некоторого города, представленного в ней этим единственным кортежем, будут удалены не только сведения о поставщике из данного города, но и информация о том, какой статус соответствует самому городу. Например, при удалении из переменной отношения SECOND кортежа для поставщика с номером S5 будет утрачена информация о том, что для Афин был установлен статус 30. (И в этом случае операции вставки и удаления фактически являются двумя сторонами одной и той же медали.)

  Примечание. И вновь причиной подобных неприятностей является смешивание информации — переменная отношения SECOND содержит информацию о поставщиках и *вместе с ней* информацию о городах. Для выхода из этой ситуации следует поступить так же, как и раньше, т.е. разделить смешанную информацию и перенести одну ее часть в переменную отношения со сведениями о поставщиках, а другую — в переменную отношения со сведениями о городах.
- **Операция UPDATE.** В переменной отношения SECOND значение статуса для каждого города повторяется несколько раз (поэтому она все еще обладает некоторой избыточностью). Следовательно, если нужно будет изменить для Лондона значение статуса 20 на 30, то потребуется отыскать в переменной отношения SECOND все кортежи, в которых связаны между собой значения London и 20 (для внесения соответствующих изменений). В противном случае база данных окажется в противоречивом состоянии (в одних кортежах статус для Лондона будет равен 20, а в других— 30).

И вновь для решения этой проблемы следует заменить исходную переменную отношения (в данном случае SECOND) следующими двумя проекциями.

```
SC { S#, CITY }
CS { CITY, STATUS }
```

Диаграммы функциональных зависимостей для этих переменных отношения показаны на рис. 12.9, а их содержимое — на рис. 12.10. Обратите внимание, что информация о статусе Рима (Rome) включена только в переменную отношения CS. Данное преобразование обратимо, поскольку переменная отношения SECOND может быть получена посредством соединения переменных отношения SC и CS по атрибуту CITY.

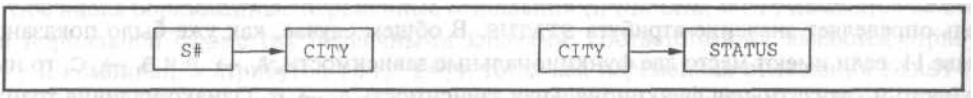


Рис. 12.9. Функциональные зависимости в переменных отношения SC и CS

SC

S#	CITY
S1	London
S2	Paris
S3	Paris
S4	London
S5	Athens

CS

CITY	STATUS
Athens	30
London	20
Paris	10
Rome	50

Аналог этой строки на рис. 12.8 отсутствует

Рис. 12.10. Данные в переменных отношения SC и CS

И на этот раз вполне очевидно, что подобное изменение структуры переменных отношения позволяет устранить все описанные выше проблемы в операциях обновления. Читателю предлагается самостоятельно разобраться в подробностях решения этих проблем. Сравнивая рис. 12.7 и 12.9, можно заметить, что благодаря дальнейшей декомпозиции удалось исключить транзитивную зависимость атрибута STATUS ОТ атрибута s#. Это позволило избавиться от всех существовавших трудностей. Интуитивно понятное объяснение состоит в том, что в переменной отношения SECOND атрибут STATUS соответствует не той сущности, которая идентифицируется первичным ключом отношения (т.е. номером поставщика), а содержит информацию о городе, в котором в настоящее время поставщик проводит свои деловые операции. Именно смешивание этих двух типов информации в одной переменной отношения приводило к возникновению описанных выше проблем.

Теперь можно дать определение третьей нормальной формы.

- **Третья нормальная форма** *(в определении предполагается наличие только одного потенциального ключа, который к тому же является первичным ключом отношения)*. Переменная отношения находится в третьей нормальной форме тогда и только тогда, когда она находится во второй нормальной форме и ни один неключевой атрибут не является транзитивно зависимым от ее первичного ключа. *Примечание.* Под этим подразумевается отсутствие в переменной отношения транзитивных зависимостей. Это означает, что в ней отсутствуют какие-либо *взаимные* зависимости в указанном выше смысле.

Переменные отношения SC и CS находятся в третьей нормальной форме, причем первичными ключами в них являются, соответственно, атрибуты {S#} и {CITY}. Переменная отношения SECOND не находится в третьей нормальной форме. Переменная отношения,

которая находится в 2НФ, но не находится в ЗНФ, всегда может быть преобразована в эквивалентный набор переменных отношения в ЗНФ. Как говорилось ранее, этот процесс обратим и, следовательно, никакая информация при подобном преобразовании не утрачивается. Однако результирующий набор отношений в ЗНФ способен содержать такую информацию[8], которая не могла быть представлена в исходной переменной отношения в 2НФ (например, сведения о том, что статус Рима соответствует 50).

Подводя итог сказанному, можно отметить, что второй этап нормализации состоит в создании проекций для устранения транзитивных зависимостей. Иначе говоря, пусть дана переменная отношения R, имеющая следующий вид.

```
R { A, B, C }
  PRIMARY KEY { A }
  /* Предполагается наличие функциональной зависимости B → */
```

Процедура нормализации предусматривает замену переменной отношения R следующими двумя проекциями, R1 и R2.

```
R1 { B, C }
     PRIMARY KEY { B }

R2 { A, B }
     PRIMARY KEY { A }
     FOREIGN KEY { B } REFERENCES R1
```

Переменная отношения R может быть восстановлена посредством соединения переменных отношения R1 и R2 по внешнему ключу и соответствующему ему первичному ключу этих переменных отношения.

В заключение следует подчеркнуть, что уровень нормализации переменной отношения определяется семантикой, а не конкретным значением этой переменной в определенный момент времени. Иначе говоря, по конкретному значению некоторой переменной отношения невозможно определить, находится ли она, например, в ЗНФ. Для этого необходимо также знать, какие функциональные зависимости определены в рассматриваемой переменной отношения. Следует также отметить, что даже зная о зависимостях в некоторой переменной отношения, нельзя на основании конкретного ее значения *доказать,* что она находится в ЗНФ. Самое лучшее, чего можно достичь в подобном случае, — это лишь продемонстрировать, что данное конкретное значение не нарушает никаких зависимостей, и, если это так, высказать предположение о том, что *рассматриваемое значение переменной отношения не противоречит гипотезе* о ее принадлежности к ЗНФ. Однако сам этот факт не гарантирует, что предложенная гипотеза верна.

12.4. СОХРАНЕНИЕ ЗАВИСИМОСТЕЙ

В процессе нормализации часто возникает ситуация, когда переменная отношения может быть подвергнута декомпозиции без потерь несколькими разными способами. Вновь обратимся к приведенной выше переменной отношения SECOND с функциональными

---

[8] Из этого следует, что комбинация переменных отношения "SECOND—SP" немного лучше представляет реальный мир по сравнению с переменной отношения FIRST, находящейся в 1НФ, а комбинация переменных отношения "SC-CS" — немного лучше по сравнению с переменной отношения SECOND, находящейся в2НФ.

зависимостями S# → CITY и CITY → STATUS и, следовательно, с еще одной транзитивной зависимостью S# →STATUS (на рис. 12.11 эта транзитивная зависимость показана пунктирной стрелкой). В разделе 12.3 отмечалось, что аномалии обновления, характерные для переменной отношения SECOND, можно предотвратить посредством ее декомпозиции с последующей заменой двумя проекциями в ЗНФ.

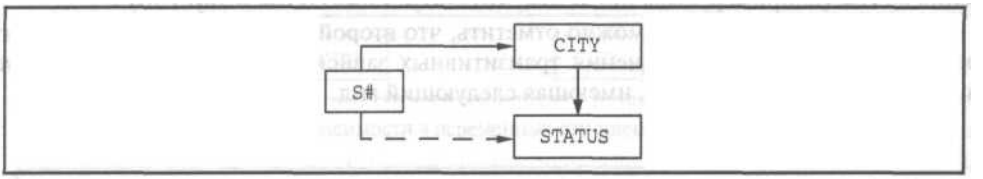


Рис. 12.11. Функциональные зависимости в переменной отношения SECOND

```
SC  { S#,  CITY  }
CS { CITY, STATUS }
```

Назовем эту декомпозицию "декомпозицией А". Ниже приведен еще один вариант декомпозиции (декомпозиция в).

```
SC { S#, CITY }
SS { S#, STATUS }
```

При этом проекции SC одинаковы и для варианта А, и для варианта в. Декомпозиция в также выполняется без потери информации, и обе ее проекции находятся в ЗНФ. Однако по некоторым причинам декомпозиция в является менее приемлемой, чем декомпозиция А. Например, после выполнения декомпозиции в по-прежнему будет невозможно ввести информацию о том, что некоторый город имеет определенный статус, не указав конкретного поставщика из этого города.

Рассмотрим этот пример подробнее. Прежде всего заметим, что зависимости, использованные для создания проекций в декомпозиции А, соответствуют *сплошными* стрелкам (см. рис. 12.11), тогда как одна из зависимостей, использованная для создания проекций в декомпозиции в, отмечена *пунктирной* стрелкой. В декомпозиции А обе проекции независимы одна от другой в том смысле, что обновления в каждой из них могут выполняться совершенно независимо[9]. Если гарантируется, что выполняемые обновления будут допустимы в контексте данной проекции (т.е. уникальность ее первичного ключа не нарушается), то *соединение этих двух проекций после обновления всегда будет иметь результатом допустимое значение переменной отношения* SECOND. Это следует понимать так, что при соединении не будут нарушаться ограничения, наложенные на функциональные зависимости в переменной отношения SECOND. Однако в случае декомпозиции в, вносимые в любую из двух проекций обновления должны тщательно контролироваться, чтобы можно было исключить возможные нарушения функциональной зависимости CITY → STATUS. (Нарушения могут иметь место, если два и более поставщиков находятся в одном и том же городе; в этом случае они должны иметь *один* статус. В качестве примера разберите случай, когда в декомпозиции в поставщик с номером S1 перемещается из Лондона в Париж.) Иначе говоря, две проекции декомпозиции в не являются независимыми друг от друга.

---

[9] Если не учитывать ограничение ссылочной целостности, которое связывает между собой переменные отношения SC и CS.

Основная проблема заключается в том, что в декомпозиции в функциональная зависимость CITY → STATUS превращается (в соответствии с терминологией главы 9) в *ограничение базы данных,* охватывающее две переменные отношения. (Следует отметить, что во многих современных программных продуктах подобные ограничения должны поддерживаться с помощью специального процедурного кода.) В противоположность этому, в декомпозиции А ограничением базы данных является *транзитивная* зависимость S# → STATUS, которая поддерживается автоматически, если обеспечена поддержка двух ограничений *переменных отношения: S#* → STATUS и CITY → STATUS. Реализовать эти ограничения очень просто, поскольку, по сути, они сводятся к поддержке уникальности значений первичных ключей в соответствующих переменных отношения.

Таким образом, концепция независимых проекций предоставляет критерий выбора одного из возможных вариантов декомпозиции. В частности, вариант декомпозиции, обеспечивающий *независимость* проекций в приведенном выше смысле, в общем случае предпочтительнее вариантов, в которых проекции будут зависимы. Риссанен (Rissanen) [12.6] показал, что проекции R1 и R2 переменной отношения R будут независимы в упомянутом выше смысле тогда и только тогда, когда соблюдаются следующие требования:

- каждая функциональная зависимость в переменной отношения R является логиче ским следствием функциональных зависимостей в ее проекция Rl и R2;
- общие атрибуты проекций R1 и R2 образуют потенциальный ключ по крайней ме ре для одной из этих двух проекций.

Рассмотрим заданные выше декомпозиции A и в. В декомпозиции А обе проекции независимы, поскольку их общий атрибут CITY является первичным ключом для переменной отношения CS и каждая функциональная зависимость переменной отношения SECOND либо представлена в одной из проекций, либо является логическим следствием других имеющихся в них ФЗ. В декомпозиции в, наоборот, две составляющие ее проекции не являются независимыми, поскольку функциональная зависимость CITY → STATUS не может быть выведена из ФЗ, существующих в этих проекциях, даже несмотря на то, что их общий атрибут S# является потенциальным ключом для обеих проекций.

Примечание. Третий вариант декомпозиции с заменой переменной отношения SECOND проекциями {S#, STATUS} и {CITY, STATUS} не является допустимой декомпозицией, поскольку сопровождается потерей информации. *(Упражнение.* Докажите это утверждение.)

В [12.6] переменная отношения, которая не может быть подвергнута декомпозиции с получением независимых проекций, называется **атомарной** (это — не очень удачный термин). Однако это вовсе не означает, что каждую переменную отношения, отличную от атомарной (в указанном смысле), следует обязательно разбивать на атомарные компоненты. Например, переменные отношения S и p из упоминавшейся выше базы данных поставщиков и деталей не являются атомарными, однако дальнейшая их декомпозиция имела бы мало смысла. Переменная отношения SP, наоборот, *является* атомарной.

Идея о том, что нормализация всегда должна предусматривать декомпозицию переменных отношения на независимые проекции (в определенном Риссаненом смысле), называется требованием **сохранения зависимостей.** В завершение настоящего раздела приведем несколько более точное определение этой концепции.

i 1. Пусть дана переменная отношения R, которая после выполнения всех этапов нормализации заменяется множеством переменных отношения Rl, R2, ..., Rn, являющихся проекциями переменной отношения R.

2. Пусть также задано множество функциональных зависимостей S, имеющих место в исходной переменной отношения R, и множество функциональных зависимостей S1, S2, ..., Sn, которые, соответственно, выполняются в переменных отношения R1, R2, ..., Rn.

. 3. Каждая функциональная зависимость в множестве Si относится только к атрибутам проекции Ri (где i=l, 2, 3, ..., п). В результате реализация ограничений (устанавливаемых существующими ФЗ) для любого данного множества Si намного упрощается. Однако в действительности необходимо реализовать все ограничения, определяемые исходным множеством функциональных зависимостей S. Следовательно, целесообразно выбрать такой вариант декомпозиции исходной переменной отношения на проекции Rl, R2, ..., Rn, при котором совместный эффект от реализации ограничений для отдельных множеств S1, S2, ..., Sn будет эквивалентен реализации всех ограничений для исходного множества функциональных зависимостей S. Иначе говоря, декомпозиция должна обеспечивать *сохранение зависимостей.*

4. Пусть S ' является объединением множества зависимостей S1, S2, ..., Sn. Об ратите внимание на то, что в общем случае равенство S' =S *не* выполняется. Для декомпозиции с сохранением зависимостей достаточно, чтобы были равны *замы кания* множеств S и S ' (понятие замыкания множества функциональных зависи мостей рассматривалось в разделе 11.4 главы 11).

5. В общем случае не существует эффективного метода вычисления замыкания S^+ для заданного множества функциональных зависимостей, поэтому задача вычис ления этих двух замыканий и проверки их на равенство фактически является не выполнимой.

   Тем не менее, существует эффективный метод проверки того, будет ли декомпозиция выполняться с сохранением зависимостей. Описание всех нюансов этого алгоритма выходит за рамки данной главы, но заинтересованный читатель сможет найти его в [8.13].

Ниже приведен состоящий из девяти этапов алгоритм, с помощью которого может быть выполнена декомпозиция без потерь произвольной переменной отношения R (с сохранением функциональных зависимостей) на множество D проекций, находящихся в ЗНФ. Предположим, что дано множество функциональных зависимостей S, удовлетворяемых в переменной отношения R. В таком случае декомпозиция может быть выполнена, как показано далее.

1. Инициализировать D значением пустого множества.
2. Пусть I является неприводимым покрытием для S.
3. Пусть x — множество атрибутов, присутствующих в левой части некоторой функ циональной зависимости X → У из I.

4. Пусть полным множеством функциональных зависимостей из I с левой частью X является $X \rightarrow Y1, X \rightarrow Y2, \ldots, X \rightarrow Yn$.
5. Пусть объединением Yl, Y2, ..., Yn является z.
6. Заменить множество D объединением множества D и проекции R по X и z.
7. Повторить шаги 4—6 для каждого отдельного X.
8. Пусть A1, A2, ..., An являются теми атрибутами R (если только они вообще имеются), которые все еще не охвачены этим алгоритмом (т.е. не включены ни в одну переменную отношения из D); заменить множество D объединением множе ства D и проекции R по A1, A2, ..., An.
9. Если ни одна переменная отношения из D не включает некоторый потенциальный ключ переменной отношения R, заменить D объединением D и проекции R по рас сматриваемому потенциальному ключу переменной отношения R.

12.5. НОРМАЛЬНАЯ ФОРМА БОЙСА-КОДДА

В этом разделе мы отменим применявшееся выше допущение о том, что каждая переменная отношения имеет только один потенциальный ключ (а именно— первичный ключ), и рассмотрим более общий случай. Дело в том, что первоначальное определение, данное Коддом для ЗНФ [11.6], не во всех случаях оказывается удовлетворительным. В частности, оно неадекватно при выполнении следующих условий, касающихся определенной переменной отношения:

1. переменная отношения имеет два (или больше) потенциальных ключа, таких, что
2. эти потенциальные ключи являются составными и
3. два или больше потенциальных ключей перекрываются (т.е. имеют по крайней мере один общий атрибут).

Поэтому впоследствии исходное определение ЗНФ было заменено более строгим определением Бойса—Кодда (Boyce—Codd) [12.2]. А поскольку это новое определение фактически задает нормальную форму, которая во всех отношениях сильнее по сравнению со старой ЗНФ, для нее было установлено собственное название[10] — *нормальная форма Бойса—Кодда* (или НФБК).

Примечание. Комбинация условий 1—3 на практике встречается не часто. Для любой переменной отношения, в которой не выполняются все эти три условия, ЗНФ и НФБК полностью эквивалентны.

Для объяснения концепции НФБК необходимо снова воспользоваться понятием *детерминанта,* введенным в главе 11 для обозначения левой части некоторой функциональной зависимости, а также понятием *тривиальной функциональной зависимости,* которое обозначает такую ФЗ, левая часть которой является надмножеством правой части. Дадим определение НФБК.

---

[10] На самом деле строгое определение "третьей" нормальной формы, эквивалентное определению нормальной формы Бойса-Кодда, впервые было дано Хитом (Heath) в 1971 году [11.4], поэтому данную форму следовало бы называть "нормальной формой Хита".

- **Нормальная форма Бойса—Кодца.** Переменная отношения находится в *нормальной форме Бойса-Кодда* тогда и только тогда, когда каждая ее нетривиальная и неприводимая слева функциональная зависимость имеет в качестве своего детерминанта некоторый потенциальный ключ.

Можно дать и другой, менее формальный вариант этого определения.

- **Нормальная форма Бойса—Кодда** *{неформальное определение).* Переменная отношения находится в *нормальной форме Бойса—Кодда* тогда и только тогда, когда детерминанты всех ее ФЗ являются потенциальными ключами.

Иначе говоря, на диаграмме функциональных зависимостей все стрелки должны начинаться только с элементов, представляющих потенциальные ключи. Выше уже было указано, что стрелка должна отходить от каждого потенциального ключа, а в соответствии с данным определением, *никакие другие* стрелки не допускаются и, следовательно, никакие стрелки не могут быть исключены с помощью описанной выше процедуры нормализации.

Примечание. Различие между двумя приведенными определениями **НФБК** состоит в том, что в менее формальном из них неявно подразумевается, что детерминанты "не слишком велики" и все ФЗ нетривиальны. Для простоты изложения далее в этой главе будут использованы такие же допущения, за исключением особо оговоренных случаев.

Следует также отметить, что определение НФБК концептуально проще, чем данное ранее определение ЗНФ, поскольку в нем нет явных ссылок на первую и вторую нормальные формы, а также не используется концепция транзитивной зависимости. Кроме того, хотя (как отмечалось выше) определение НФБК является во всех отношениях более сильным, чем определение ЗНФ, при его использовании любая переменная отношения может быть подвергнута декомпозиции без потерь информации на некоторое эквивалентное множество переменных отношения в НФБК.

Прежде чем рассматривать примеры переменных отношения с несколькими потенциальными ключами, покажем, что переменные отношения FIRST и SECOND, которые не находятся в ЗНФ, не находятся также в НФБК. Кроме того, покажем, что переменные отношения SP, SC и CS, которые находятся в ЗНФ, находятся также в **НФБК.** Переменная отношения FIRST содержит три детерминанта, а именно — S#, CITY и {S#, P#}, из которых только {S#, P#} является ее потенциальным ключом. Поэтому переменная отношения FIRST не находится в **НФБК.** Аналогичное утверждение верно для переменной отношения SECOND, поскольку детерминант CITY не является ее потенциальным ключом. С другой стороны, переменные отношения SP, SC и CS находятся в НФБК, поскольку в каждом случае единственный потенциальный ключ является единственным детерминантом для данной переменной отношения.

Теперь рассмотрим пример, содержащий два отдельных (т.е. неперекрывающихся) потенциальных ключа. Допустим, что в переменной отношения поставщиков S{S#, SNAME, STATUS, CITY} множества атрибутов {S#} и {SNAME} являются ее потенциальными ключами (т.е. в этом случае каждый поставщик имеет уникальный номер и уникальное имя). Также предположим (как, впрочем, и всюду в этой книге), что атрибуты STATUS и CITY являются взаимно независимыми, т.е. введенная выше просто для раскрытия темы раздела 12.3 функциональная зависимость CITY→ STATUS больше не соблюдается. Тогда диаграмма функциональных зависимостей будет иметь вид, представленный на рис. 12.12.

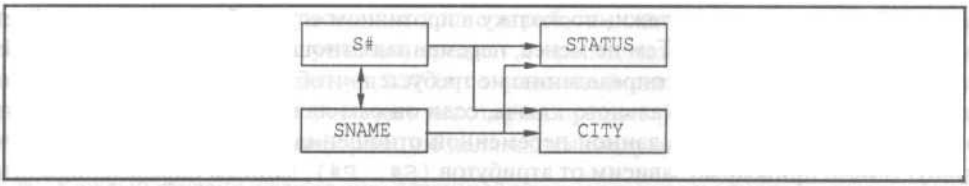


Рис. 12.12. Диаграмма функциональных зависимостей в переменной отношения S для случая, ко- : гда множество атрибутов {SNAME} является ее потенциальным ключом (и зависимость CITY—> STATUS не выполняется)

Можно видеть, что переменная отношения S находится в НФБК. Хотя ее диаграмма ФЗ существенно "сложнее" диаграммы ФЗ для переменной отношения в ЗНФ, в этом случае все детерминанты являются потенциальными ключами, поскольку все стрелки начинаются с потенциальных ключей. Таким образом, из данного примера можно сделать вывод, что наличие нескольких потенциальных ключей не обязательно должно рассматриваться как признак плохого проекта.

Теперь рассмотрим несколько примеров, в которых потенциальные ключи перекрываются. Два потенциальных ключа перекрываются, если они содержат по два или больше атрибутов и имеют хотя бы один общий атрибут.

Примечание. В соответствии с доводами, которые приводились в главе 9, в примерах этой главы из имеющихся потенциальных ключей первичный ключ выбираться не будет. Поэтому в представленных ниже таблицах никакие столбцы не будут обозначаться с помощью двойной линии.

В первом примере снова примем предположение, что имена поставщиков уникальны, и рассмотрим переменную отношения.

SSP { S#, SNAME, P#, QTY }

Потенциальными ключами в ней являются множества атрибутов {S#, P#} и {SNAME, P#}. Однако эта переменная отношения не находится в НФБК, так как она содержит два детерминанта, s# и SNAME, которые не являются ее потенциальными ключами (и {S#}, и {SNAME} —детерминанты, поскольку они определяют друг друга). Пример значения переменной отношения SSP показан на рис. 12.13.

SSP

S#	SNAME	P#	QTY
S1	Smith	P1	300
S1	Smith	P2	200
S1	Smith	P3	400
S1	Smith	P4	200
..	.....	..	...

Рис. 12.13. Пример значения переменной отношения SSP (показан не полностью)

Согласно этому рисунку, переменной отношения SSP свойственна некоторая доля избыточности, как и переменным отношения FIRST И SECOND (СМ. раздел 12.3), а также переменной отношения SCP (см. раздел 12.1), поэтому при ее использовании возникают такие же проблемы, которые были характерными для последних. Например, для изменения имени поставщика, имеющего номер S1, со Smith на Robinson необходимо найти

все относящиеся к нему кортежи, поскольку в противном случае база данных перейдет в противоречивое состояние. Тем не менее, переменная отношения SSP находится в ЗНФ, поскольку согласно старому определению не требуется, чтобы атрибут был неприводимо зависим от каждого потенциального ключа, если он сам является компонентом некоторого потенциального ключа данной переменной отношения. В результате тот факт, что атрибут SNAME приводимо зависим от атрибутов {S#, P#}, данным определением игнорируется.

Примечание. Под термином *ЗНФ* здесь подразумевается определение, данное Коддом в [11.6], а не упрощенное определение, данное нами в разделе 12.3.

Для решения указанной проблемы переменную отношения SSP следует разбить на две проекции, как показано ниже.

```
SS  {  S#,   SNAME
} SP  {  S#,   P#,
QTY  }
```

Однако можно выбрать и альтернативный вариант разбиения.

```
SS  {  S#,   SNAME  }
SP  {  SNAME,   P#,   QTY  }
```

Обратите внимание, что в этом примере показаны два варианта декомпозиции, и оба они в равной степени являются допустимыми, поскольку все входящие в них проекции находятся в НФБК.

Здесь следует сделать небольшое отступление и пояснить, что же происходит "в действительности". *Ясно,* что исходный вариант, состоящий из одной переменной отношения SSP, неудачен и возникающие в связи с этим проблемы вполне очевидны. Маловероятно, чтобы подобный проект предложил более или менее опытный разработчик баз данных, даже если он совершенно незнаком с концепцией НФБК (и т.д.). Простой здравый смысл подскажет нам, что вариант с двумя переменными отношения SS и SP, несомненно, лучше. Но что в данном случае подразумевается под "здравым смыслом" и какими *принципами* должен руководствоваться разработчик, отдавая предпочтение варианту с переменными отношения SS и SP, а не варианту с переменной отношения SSP?

Безусловно, ответ заключается в том, что таковыми являются именно принципы функциональной зависимости и нормальная форма Бойса—Кодда. Иначе говоря, рассматриваемые нами концепции (ФЗ, НФБК и все прочие формальные идеи, изложенные в этой и следующей главах) являются не чем иным, как *соображениями здравого смысла, записанными в формальном виде.* Сутью излагаемой здесь *теории нормализации* является поиск и формулирование этих принципов здравого смысла, что, конечно же, является весьма непростой задачей. Однако, если такая задача будет нами решена, найденные принципы могут быть положены в основу решений по *автоматизации* проектирования, т.е. можно будет написать программу, позволяющую выполнять проектирование с помощью компьютера. Критики методов нормализации обычно упускают этот момент из виду, совершенно справедливо заявляя, что данные идеи в основном представляют собой всего лишь продукт здравого смысла. Однако при этом не принимается во внимание, что возможность применения формальной и строгой формулировки вместо "соображений здравого смысла" уже сама по себе является значительным достижением.

Теперь вернемся к основной теме данного раздела и представим второй пример с перекрывающимися потенциальными ключами (следует предупредить читателя, что это всего лишь пример, поэтому не стоит задумываться над тем, что он далек от реальности).

Рассмотрим переменную отношения SJT с атрибутами s, J и т, которые обозначают, соответственно, студента, изучаемый предмет и преподавателя. Смысл каждого кортежа (s, j , t) переменной отношения SJT (в сокращенной системе обозначений) состоит в том, что некоторый студент s изучает некоторый предмет j на лекциях некоторого преподавателя t. При этом на информацию налагаются следующие два ограничения.

- Каждый студент изучает определенный предмет только у одного преподавателя.
- Каждый преподаватель ведет только один предмет, но каждый предмет ведут не сколько преподавателей.

На рис. 12.14 показан пример значения переменной отношения SJT.

SJT

S	J	T
Smith	Math	Prof. White
Smith	Physics	Prof. Green
Jones	Math	Prof. White
Jones	Physics	Prof. Brown

Рис. 12.14. Пример значения переменной отношения SJT

Какие функциональные зависимости существуют в этой переменной отношения? Из первого ограничения следует функциональная зависимость {S, J} → т, а из второго — функциональная зависимость T → J. Наконец, сам факт, что каждый предмет ведут несколько преподавателей, говорит о том, что функциональная зависимость J → т не соблюдается. Следовательно, диаграмма функциональных зависимостей переменной отношения SJT будет иметь вид, представленный на рис. 12.15.

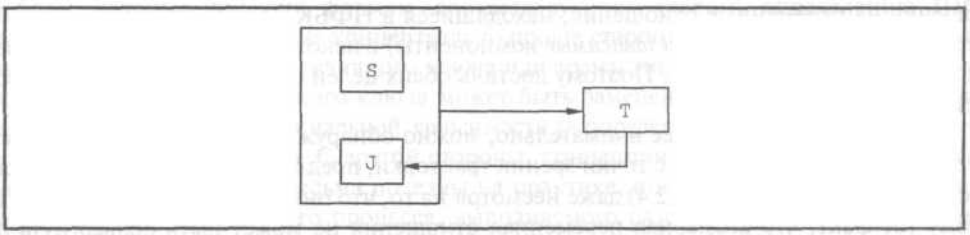


Рис. 12.15. Диаграмма функциональных зависимостей в переменной отношения SJT

И вновь в рассматриваемом примере присутствуют два перекрывающихся потенциальных ключа, а именно множество атрибутов {S, J} и множество атрибутов {S, T}. Как и в первом примере, исходная переменная отношения SJT находится в ЗНФ, но не в НФБК, в связи с чем для нее будут характерны некоторые аномалии обновления. Например, если потребуется удалить сведения о том, что студент Джонс (Jones) изучает физику, этого нельзя будет сделать, не утратив одновременно информацию о том, что профессор Браун (Prof . Brown) преподает этот предмет. Подобные трудности вызваны тем, что атрибут т является детерминантом, но не является потенциальным ключом. И вновь для решения указанной проблемы исходную переменную отношения SJT следует разбить на две проекции, каждая из которых будет находиться в НФБК.

ST { S, T }
TJ { T, J }

Предлагаем читателю в качестве упражнения определить значения этих двух переменных отношения (которые соответствуют данным, представленным на рис. 12.14), нарисовать для проверки соответствующую диаграмму ФЗ, доказать, что эти две вновь созданные проекции находятся в НФБК (объясните, какие атрибуты являются их потенциальными ключами), и проверить, действительно ли такая декомпозиция позволяет устранить все аномалии обновления.

Однако следует отметить, что все еще существует иная проблема. Суть ее в том, что декомпозиция исходного отношения на проекции ST и TJ позволяет исключить одни аномалии, но приводит к появлению других. Причиной является тот факт, что проекции ST и TJ не являются независимыми в том смысле, который был указан Риссаненом (см. раздел 12.4). Точнее говоря, функциональная зависимость

{S, J} → T

не может быть выведена из той единственной функциональной зависимости, которая присутствует в двух данных проекциях,

T → J

В результате две полученные проекции не могут обновляться независимо. Например, попытка вставить в переменную отношения ST кортеж для студента Смита (Smith) и профессора Брауна (Prof . Brown) должна быть отвергнута системой, поскольку профессор Браун преподает физику, а Смит уже обучается физике у профессора Грина (Prof. Green). Однако система не может обнаружить этот факт, не проверив содержимое переменной отношения TJ. Таким образом, мы пришли к неприятному выводу о том, что попытка достижения двух целей (а именно декомпозиции исходной переменной отношения на переменные отношения, находящиеся в НФБК, и декомпозиции исходной переменной отношения на *независимые* компоненты) в некоторых случаях может привести к конфликтной ситуации. Поэтому достичь обеих целей одновременно не всегда возможно.

Изучив этот пример более внимательно, можно обнаружить, что в действительности переменная отношения s JT с точки зрения трактовки, предложенной Риссаненом, является *атомарной* (см. раздел 12.4) даже несмотря на то, что она не находится в НФБК. Поэтому тот факт, что атомарная переменная отношения не может быть подвергнута декомпозиции на независимые компоненты, отнюдь не означает, что она вообще не может быть подвергнута декомпозиции (здесь под "декомпозицией" понимается декомпозиция без потерь). Таким образом, интуитивные соображения подсказывают, что концепцию атомарности нельзя назвать очень продуктивной, поскольку атомарность переменных отношения не может служить необходимым или достаточным условием создания хорошего проекта базы данных.

В качестве третьего и последнего примера переменной отношения с перекрывающимися потенциальными ключами рассмотрим переменную отношения EXAM с атрибутами S (студент), J (предмет) и P (позиция). Каждый кортеж (s, j , p) переменной отношения EXAM отражает сведения о том, что некоторый студент s экзаменуется по определенному предмету j и занимает определенную позицию p в экзаменационной ведомости. Дополнительно условимся, что в нашем примере имеет место следующее ограничение.

- Никакие два студента не могут занимать одну и ту же позицию в экзаменационной ведомости, которая относится к одному и тому же предмету.

В этом случае диаграмма функциональных зависимостей будет иметь вид, представленный на рис. 12.16.

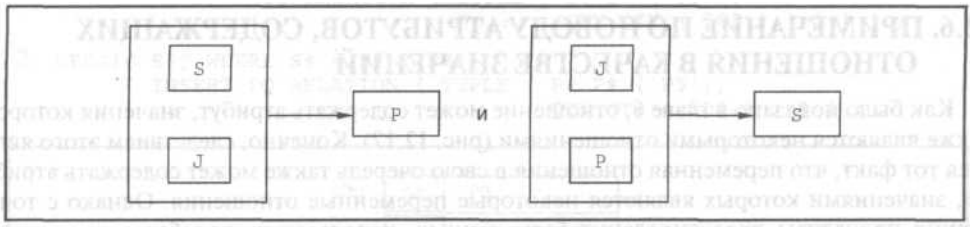


Рис. 12.16. Диаграмма функциональных зависимостей в переменной отношения EXAM

В этом примере вновь присутствуют два перекрывающихся потенциальных ключа, {S, J} и {J, P}, поскольку для каждого студента и предмета существует точно одна занимаемая позиция в соответствующем списке, а в каждом списке экзаменующихся по некоторому предмету каждую позицию занимает только один соответствующий студент. Однако такая переменная отношения находится в НФБК, поскольку указанные потенциальные ключи являются ее единственными детерминантами. Поэтому для данной переменной отношения не характерны какие-либо аномалии обновления, подобные упоминавшимся ранее в настоящей главе. *{Упражнение.* Проверьте правильность этого утверждения.) Таким образом, наличие перекрывающихся потенциальных ключей *не всегда* приводит к появлению проблем подобного рода.

В заключение следует подчеркнуть, что концепция НФБК позволяет избавиться от проблем, которые присущи формам, соответствующим старому определению ЗНФ. Более того, новое определение концептуально проще старого, так как в нем не используются понятия 1НФ, 2НФ, первичного ключа или транзитивной зависимости. Дополнительно понятие потенциального ключа может быть заменено ссылкой на более фундаментальное понятие функциональной зависимости (в определении, данном в [12.2], имеет место именно такая замена). С другой стороны, концепции первичного ключа, транзитивной зависимости и т.д. весьма полезны на практике, поскольку позволяют наметить схему некоторого пошагового процесса, выполняемого разработчиком для приведения произвольной переменной отношения к эквивалентному набору переменных отношения в НФБК.

Для использования в будущем приведем в конце этого раздела алгоритм, состоящий из четырех шагов, с помощью которого произвольная переменная отношения R может быть подвергнута декомпозиции без потерь на множество D проекций НФБК (но при этом не обязательно сохраняются все зависимости).

1. Инициализировать множество D так, чтобы в нем содержалась только переменная отношения R.
2. Для каждой переменной отношения т из множества D, не находящейся в НФБК, выполнить шаги 3 и 4.

3. Пусть X —> Y является функциональной зависимостью для т, которая нарушает требования НФБК.
4. Заменить переменную отношения т из множества D двумя ее проекциями: по ат рибутам X и Y и по всем атрибутам, кроме тех, что находятся BY.

12.6. ПРИМЕЧАНИЕ ПО ПОВОДУ АТРИБУТОВ, СОДЕРЖАЩИХ ОТНОШЕНИЯ В КАЧЕСТВЕ ЗНАЧЕНИЙ

Как было показано в главе 6, отношение может содержать атрибут, значения которого также являются некоторыми отношениями (рис. 12.17). Конечно, следствием этого является тот факт, что переменная отношения в свою очередь также может содержать атрибуты, значениями которых являются некоторые переменные отношения. Однако с точки зрения процедуры проектирования базы данных, использовать подобные переменные отношения обычно не рекомендуется, поскольку для них характерна *асимметричность* (не говоря уже о том, что их предикаты, как правило, весьма сложны), а эта асимметричность способна вызвать разные проблемы практического характера. Например, для случая, показанного на рис. 12.17, обработка сведений о поставщиках и деталях должна осуществляться асимметрично. Рассмотрим два приведенных ниже симметричных запроса.

1. Получить список номеров поставщиков (s#) детали с номером P1.
2. Получить список номеров деталей (P#), поставляемых поставщиком с номером S1.

Как показано ниже, эти два запроса имеют совершенно разные формулировки.

```
1. ( SPQ WHERE TUPLE { P# P# ('P1') } £ PQ { P# } ) { S# }
2. ( ( SPQ WHERE S# = S# ('S1') ) UNGROUP PQ ) { P# }
```

В данном примере предполагается, что SPQ — это переменная отношения, допустимыми значениями которой являются отношения в форме, аналогичной показанной на рис. 12.17. Следует отметить, что в данном случае не только наблюдается значительное различие между формулировками запросов, но и сами они становятся намного сложнее по сравнению со своими аналогами для переменной отношения SP.

Еще хуже обстоит дело с операциями обновления. Рассмотрим следующие две операции обновления.

1. Ввести сведения о новой поставке 500 штук деталей типа P5, выполняемой поставщиком с номером S6.
2. Ввести сведения о новой поставке 500 штук деталей типа P5, выполняемой поставщиком с номером S2.

При работе с обычной переменной отношения SP между двумя подобными обновлениями нет никакой принципиальной разницы, так как в обоих случаях речь идет о вставке в переменную отношения единственного кортежа. В противоположность этому, при работе с переменной отношения SPQ выполняемые обновления будут существенно

---

[11] Фактически подобные переменные отношения раньше даже не считались допустимыми и назывались ненормализованными, т.е. не находящимися в первой нормальной форме (см. также главу 6).

отличаться (не говоря уже о том, что и в данной ситуации *обе* эти операции будут намного сложнее, чем при работе с переменной отношения SP).

```
1. INSERT SPQ RELATION
       { TUPLE { S# S# ('S6'),
                 PQ RELATION { TUPLE { P# P# ( ' P5 ' ) ,
                                       QTY QTY ( 500 ) } } } } ;

2. UPDATE SPQ WHERE S# = S# ('S2')
       { INSERT PQ RELATION { TUPLE { P# P# { 'P5' ) ,
                                      QTY QTY ( 500 ) } } } ;
```

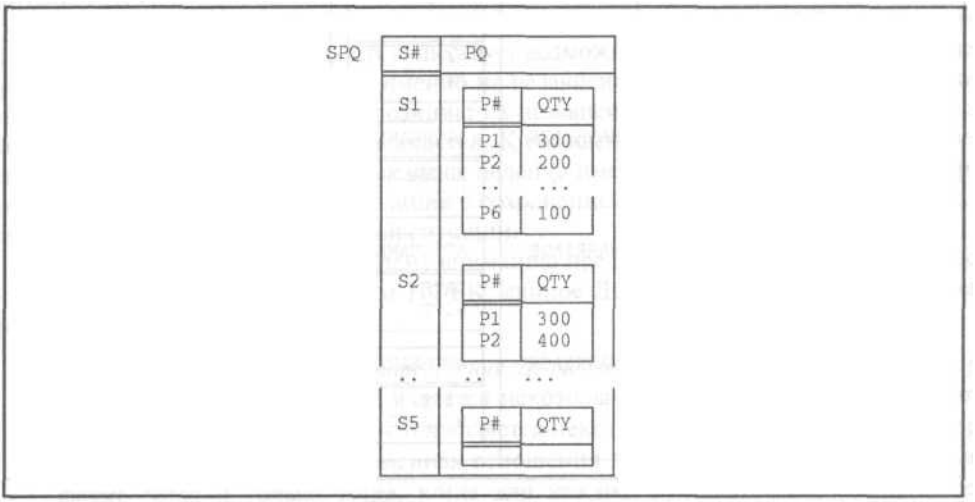


Рис. 12.17. Переменная отношения SPQ с атрибутом, содержащим в качестве значений другое отношение

Переменные отношения (по крайней мере, базовые переменные отношения) предпочтительнее создавать без использования подобных атрибутов, принимающих в качестве значений другие отношения, так как в этом случае они обладают более простой логической структурой, что существенно упрощает выполнение с ними различных операций. Однако это замечание следует воспринимать лишь как рекомендацию, но не как обязательное правило. На практике вполне могут возникать такие ситуации, когда имеет смысл использовать атрибут со значениями в виде отношений в некоторой переменной отношения и даже в базовой переменной отношения. Например, на рис. 12.18 показано (частично) возможное значение переменной отношения каталога RVK, в которой хранятся сведения о переменных отношения некоторой базы данных с указанием их потенциальных ключей. Атрибут ск в этой переменной отношения содержит значения в виде отношений. Причем он одновременно является компонентом единственного потенциального ключа переменной отношения RVK! Определение этой переменной отношения на языке Tutorial D может выглядеть, как показано ниже.

```
VAR RVK BASE RELATION
      { RVNAME NAME, CK RELATION { ATTRNAME NAME
        } } KEY { RVNAME, CK } ;
```

Примечание. В упр. 12.3 требуется найти способ устранения атрибутов, значениями которых являются отношения, если такое устранение является желательным (что обычно имеет место на практике)[12].

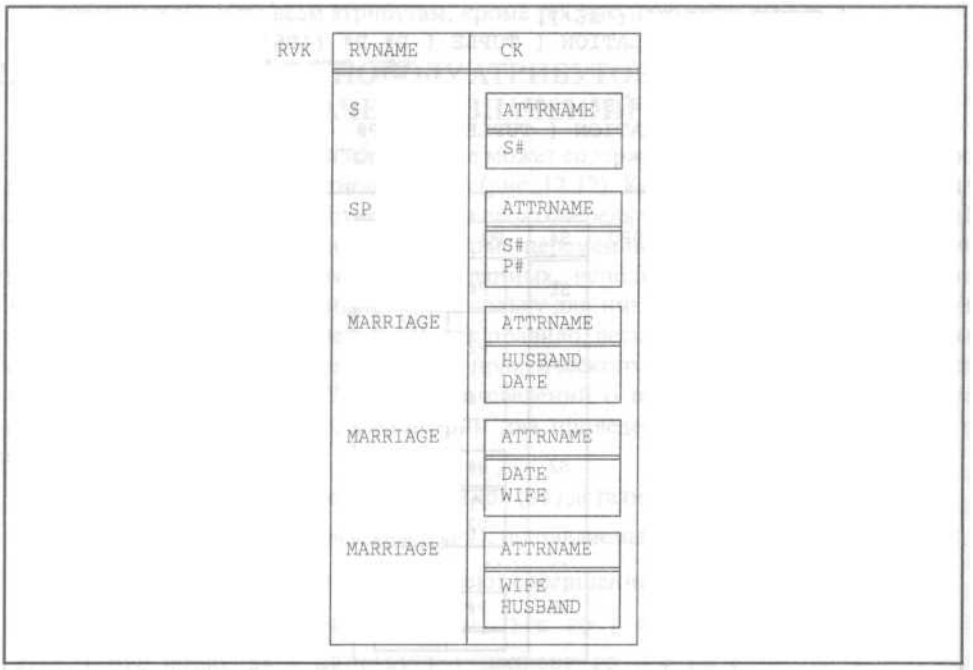


Рис. 12.18. Переменная отношения RVK, входящая в каталог некоторой базы данных

12.7. РЕЗЮМЕ

На этом заканчивается первая из двух глав, посвященных дальнейшей нормализации. В ней обсуждались концепции **первой, второй** и третьей **нормальных форм,** а также **нормальной формы Бойса-Кодца.** Вообще говоря, разные нормальные формы (включая четвертую и пятую нормальные формы, речь о которых пойдет в следующей главе) связаны между собой отношениями *упорядочения,* в том смысле, что каждая переменная отношения на некотором уровне нормализации соответствует требованиям всех более низких уровней нормализации, тогда как обратное утверждение неверно, т.е. на каждом уровне нормализации могут быть переменные отношения, которые не находятся на всех более высоких уровнях нормализации. Более того, всегда можно выполнить приведение к НФБК (а на самом деле к 5НФ), т.е. любую заданную переменную отношения всегда можно заменить эквивалентным набором переменных отношения, которые находятся в НФБК (или же в 5НФ). Такое приведение используется для того, чтобы **избежать избыточности** и, следовательно, возникновения некоторых аномалий обновления.

---

[12] И возможным! Следует отметить, что эту задачу *невозможно* решить в случае переменной отношения RVK, по крайней мере, непосредственно (т.е. без введения своего рода атрибута CKNAME, с указанием "имени потенциального ключа").

Процесс нормализации заключается в замене данной переменной отношения некоторым набором ее **проекций,** составленным таким образом, чтобы обратное соединение этих проекций позволяло вновь получить исходную переменную отношения. Иначе говоря, этот процесс является обратимым (т.е. декомпозиция всегда выполняется без потерь информации). Также было показано, что решающую роль в этом процессе играют функциональные зависимости. В частности, теорема Хита прямо утверждает, что если некоторая декомпозиция выполняется в соответствии с определенной ФЗ, то она будет выполнена без потерь. Такое положение дел может рассматриваться как еще одно подтверждение приведенного в главе 11 утверждения, что понятие ФЗ является если "не совсем фундаментальным, то весьма близким к этому".

В данной главе также обсуждалась концепция независимых проекций Риссанена и было указано, что в тех случаях, когда существует возможность выбора, исходную переменную отношения следует разбивать именно на независимые проекции, а не на проекции, зависимые одна от другой. Декомпозицию на независимые проекции принято называть декомпозицией с сохранением зависимостей. К сожалению, следует отметить, что стремление к достижению двух указанных выше целей (а именно к декомпозиции без потерь с приведением к НФБК и к декомпозиции с сохранением зависимостей) в некоторых случаях может привести к конфликтной ситуации.

В заключение данной главы вашему вниманию предлагается весьма изящное (и абсолютно точное) определение ЗНФ и НФБК, данное Дзаниоло (Zaniolo) [12.7]. Сначала приведем определение ЗНФ.

- Третья нормальная форма *(определение Дзаниоло).* Предположим, что дана переменная отношения R, что X является некоторым подмножеством атрибутов этой переменной отношения R и что A является некоторым отдельным атрибутом переменной отношения R. Переменная отношения R находится в третьей нормальной форме тогда и только тогда, когда для каждой функциональной зависимости $X \rightarrow A$ в переменной отношения R верно по крайней мере одно из следующих утверждений.

  1. Подмножество X включает атрибут A (т.е. данная ФЗ тривиальна).
  2. Подмножество X является суперключом переменной отношения R.
  3. Атрибут A входит в состав некоторого потенциального ключа переменной отношения R.

Определение **НФБК** можно получить из приведенного выше определения ЗНФ, просто исключив третье утверждение (из чего следует, что НФБК является более строгим ограничением по сравнению с ЗНФ). Именно третье утверждение является причиной того "недостатка" исходного определения третьей нормальной формы Коддом, который в конечном итоге стал причиной введения нормальной формы Бойса-Кодда.

УПРАЖНЕНИЯ

12.1. Докажите теорему Хита. Будет ли верна обратная ей теорема?

12.2. Иногда можно встретить утверждение, что каждая бинарная переменная отношения обязательно находится в НФБК. Верно ли это утверждение?

12.3. На рис. 12.19 показана структура информации о персонале компании, которая должна быть помещена в базу данных. Она представлена в том виде, который используется при работе с *иерархической* СУБД, подобной IMS (Information Management System). Эта структура удовлетворяет следующим правилам.

- В компании имеется несколько отделов.
- В каждом отделе (DEPARTMENT) есть некоторое количество сотрудников (EMPLOYEE), занятых в нескольких проектах (PROJECT) и размещающихся в нескольких офисах (OFFICE).
- Для каждого сотрудника предусмотрена ведомость выполненных работ (JOB) с информацией о заданиях, выполненных данным сотрудником.
- Для каждого такого задания предусмотрена ведомость заработной платы (SALARY HISTORY), содержащая перечень денежных сумм, полученных сотрудником за выполнение данного задания.
- В каждом офисе установлено несколько телефонов (PHONE).

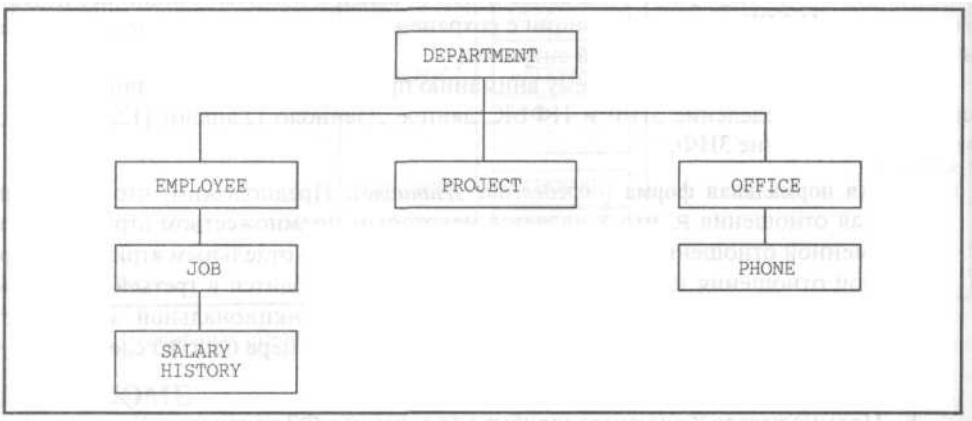


Рис. 12.19. Иерархическое представление структуры информации о персонале компании, которая должна храниться в базе данных

В базе данных должна храниться следующая информация.

- Для каждого отдела: номер отдела (уникальный), бюджет отдела и личный номер (уникальный) сотрудника, который является руководителем отдела.
- Для каждого сотрудника: личный номер сотрудника (уникальный), номер текущего проекта, номер офиса, номер телефона, а также название каждого выполненного задания с указанием дат и размеров всех выплат, проведенных в качестве оплаты за выполнение данного задания.
- Для каждого проекта: номер проекта (уникальный) и бюджет проекта.
- Для каждого офиса: номер офиса (уникальный), площадь помещения, номера (уникальные) всех установленных в нем телефонов.
- Спроектируйте соответствующее множество переменных отношения, необходимое для представления этой информации. Обоснуйте любые предположения,

которые будут сделаны вами в отношении существующих функциональных зависимостей.

12.4. В базе данных системы ввода заказов содержится информация о клиентах, товарах и заказах в соответствии с приведенным ниже описанием.

- Для каждого клиента:
- номер клиента (уникальный);
- адрес доставки (с учетом того, что каждый клиент может иметь несколько таких адресов);
- остаток не счете;
- лимит кредита;
- скидка.
- Для каждого заказа:
- информация в заголовке:
  - номер клиента;
  - адрес доставки;
  - дата выполнения заказа;
- строки расшифровки (в каждом заказе их может быть несколько):
  - номер товара;
  - заказанное количество.
- Для каждого товара:
- номер товара (уникальный);
- заводы-изготовители;
- запас товара на каждом заводе;
- минимальное допустимое количество товара на складе для каждого завода;
- описание товара.

Для внутреннего учета также вводится величина *неотгруженное количество.* Данная величина связана с каждой строкой расшифровки каждого заказа. Первоначально эта величина устанавливается равной заказанному количеству данного товара и последовательно уменьшается до нуля по мере частичного выполнения данной поставки. Составьте проект соответствующей базы данных и, как и в предыдущем упражнении, обоснуйте любые сделанные предположения о существующих функциональных зависимостях.

12.5. Предположим, что по условиям упр. 12.4 лишь очень незначительная часть клиентов, например один процент или даже меньше, имеет несколько адресов доставки. (Это обычная ситуация в реальной жизни, когда лишь незначительное число исключений— но достаточно важных— может не вписываться в предлагаемую общую структуру.) Можете ли вы в этом случае указать на какие-либо недостатки решения, которое было предложено вами для упр. 12.4? Какие усовершенствования целесообразно было бы внести для их устранения?

12.6. *{Модифицированная версия упр. 11.13.)* Переменная отношения TIMETABLE включает следующие атрибуты.

- D. День недели (1-5),

- P. Определенный период времени в течение дня (1—6).
- c. Номер аудитории.
- t. Имя преподавателя.
- S. Имя студента.
- L. Название лекции.

Кортеж (d,p,c,t,s,l) появляется в этой переменной отношения тогда и только тогда, когда в момент (d,p) некоторый студент s посещает лекцию l, которую читает преподаватель t в аудитории c. Можно предположить, что каждая лекция продолжается только в течение одного периода времени p и имеет название, уникальное по отношению ко всем другим лекциям, которые читаются на данной неделе. Сделайте структуру переменной отношения TIMETABLE более приемлемой.

12.7. *(Модифицированная версия упр. 11.14.)* Дана переменная отношения NADDR с атрибутами NAME (уникальное имя), STREET, CITY, STATE и ZIP. Примите предположения, что, во-первых, каждому почтовому индексу соответствует только один город и штат, а во-вторых, каждой улице, городу и штату — только один почтовый индекс. Находится ли переменная отношения NADDR в НФБК, 3НФ или 2НФ? Можно ли предложить улучшенный вариант этой переменной отношения?

12.8. Пусть SPQ — переменная отношения, значениями которой являются отношения в форме, показанной на рис. 12.17. Определите внешний предикат для SPQ.

СПИСОК ЛИТЕРАТУРЫ

Помимо приведенного здесь списка литературы, следует также обратить внимание на литературу к главе 11, особенно на оригинальную статью Кодда о первых трех нормальных формах [11.6].

12.1. Bernstein P.A. Synthesizing Third Normal Form Relations from Functional Dependencies //ACM TODS. — December 1976. — 1, № 4.

В этой главе обсуждались методы декомпозиции "больших" переменных отношения с преобразованием в "меньшие", т.е. имеющие меньше атрибутов, а в статье Бернштейна рассматривается обратная задача — создание "больших" переменных отношения (имеющих больше атрибутов) на основе "меньших", но эта задача фактически сформулирована в несколько иной форме. В ней поставлена проблема синтеза переменных отношения на основе заданных множеств атрибутов и соответствующих наборов ФЗ, с тем условием, что эти переменные отношения должны находиться в ЗНФ. Но поскольку атрибуты и функциональные зависимости не имеют смысла вне контекста некоторой переменной отношения, точнее и строже было бы представлять примитивные конструкции как бинарные переменные отношения, включающие ФЗ, а не просто как пару атрибутов и функциональную зависимость между ними.

Примечание. Точно так можно было бы рассмотреть заданное множество атрибутов и ФЗ как **универсальную переменную отношения** [13.20], удовлетворяющую заданному множеству зависимостей. В этом случае процесс "синтеза" можно заменить процессом *декомпозиции* этой универсальной переменной отношения на проекции в

ЗНФ. Однако далее в нашем обсуждении будет использоваться интерпретация на основе синтеза.

В такой ситуации процесс синтеза представляет собой процедуру создания л-арных переменных отношения на основе бинарных переменных отношения с заданным множеством ФЗ, связанных с этими переменными отношения, с учетом обязательного требования, чтобы все вновь созданные переменные отношения находились в ЗНФ. (Эта статья опубликована еще до того, как было определено понятие НФБК.) В данной статье также представлены алгоритмы выполнения этой задачи. Одним из возражений против указанного подхода (с которым Бернштейн согласился) является то, что выполняемые на основании предложенного алгоритма синтеза манипуляции являются чисто синтаксическими и не имеют никакого отношения к семантике. Например, третья из приведенных ниже функциональных зависимостей может быть, а может и не быть избыточной (т.е. может быть или не быть следствием первой и второй) в зависимости от смысла переменных отношения R, SИ T.

```
A → B (для переменной отношения R{A,B}) B → C (для переменной отношения S{B,C}) A → C (для переменной отношения T{A,C})
```

В качестве примера, в котором избыточность отсутствует, допустим, чтб атрибут A описывает личный номер сотрудника, атрибут в — номер офиса, атрибут C — номер отдела, отношение R содержит сведения об офисах, в которых работают сотрудники, отношение s — об отделах, которым принадлежат офисы, а отношение T— об отделах, в которых работают сотрудники. Рассмотрим случай, когда некоторый сотрудник работает в офисе не своего отдела. Согласно алгоритму синтеза, два атрибута с всегда представляют одно и то же (при этом имена переменных отношения фактически вообще не рассматриваются). Таким образом, предполагается применение некоторого внешнего механизма (т.е. вмешательство человека), позволяющего предотвратить семантически неверные манипуляции. В нашем примере следовало бы еще при определении исходных ФЗ использовать для атрибутов из переменных отношения S и T различные имена, например C1 и C2.

12.2. Codd E.F. Recent Investigations into Relational Data Base Systems // Proc. IFIP Congress. — Stockholm, Sweden, 1974. Эту статью можно также найти в других ис точниках.

Здесь приводится обзор других работ на данную тему, в частности, дается "улучшенное определение третьей нормальной формы", под которой фактически подразумевается нормальная форма, которая в наши дни известна под названием нормальной формы Бойса-Кодда. Среди других тем можно найти обсуждение *представлений* и *обновления представлений, подъязыков данных, обмена данными* и *сопутствующих исследований* (по состоянию на 1974 год).

12.3. Date C.J. A Normalization Problem // Relational Database Writings 1991-1994. Reading, Mass.: Addison-Wesley, 1995.

Согласно резюме этой статьи, в ней "рассматривается простая задача нормализации, которая используется для представления некоторых идей в области составления проекта базы данных и явного объявления ограничений целостности". Задача

сформулирована на основе простой базы данных некоторой авиакомпании с перечисленными ниже ФЗ, где приняты следующие обозначения: FLIGHT — название авиарейса, DESTINATION — место назначения, HOUR — время отправления, DAY — день недели, GATE — номер выхода для посадки на самолет, PILOT — летчик.

```
{ FLIGHT } → DESTINATION
{ FLIGHT } → HOUR
{ DAY, FLIGHT } → GATE
{ DAY, FLIGHT } → PILOT
{ DAY, HOUR, GATE } → DESTINATION
{ DAY, HOUR, GATE } → FLIGHT
{ DAY, HOUR, GATE } → PILOT
{ DAY, HOUR, PILOT } → DESTINATION
{ DAY, HOUR, PILOT } → FLIGHT
{ DAY, HOUR, PILOT } → GATE
```

Помимо всего прочего, этот пример — прекрасная иллюстрация того, что "приемлемый" проект базы данных вряд ли может быть создан только на основе принципов нормализации.

12.4. Heath I.J. Unacceptable File Operations in a Relational Database // Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control. — San Diego, Calif. — November 1971.

В работе дается определение ЗНФ, которое на самом деле является первым опубликованным определением *НФБК.* В ней также приводится доказательство теоремы, которая в разделе 12.2 была названа *теоремой Хита.* Следует отметить, что упомянутые в настоящей главе три этапа процедуры нормализации представляют собой практические приложения этой теоремы.

12.5. Kent W. A Simple Guide to Five Normal Forms in Relational Database Theory // CACM. - February 1983. - 26, № 2.

Эта публикация является первоисточником следующей весьма привлекательной характеристики ЗНФ (точнее, **НФБК): каждый атрибут должен представлять некоторый факт о ключе, только о ключе и не о чем ином, кроме ключа** (автор позволил себе немного ее перефразировать).

12.6. Rissanen J. Independent Components of Relations // ACM TODS. — December 1977. - 2, № 4.

12.7. Zaniolo C. A New Normal Form for the Design of Relational Database Schemata // ACM TODS. - September 1982. - 7, № 3.

Первоисточник изящных определений ЗНФ и НФБК, упомянутых в разделе 12.7 этой главы. Но основное назначение этой статьи заключалось в определении новой нормальной формы, *нормальной формы с элементарными ключами (НФЭК),* которая занимает промежуточное положение между ЗНФ и НФБК и "сохраняет все преимущества обеих этих форм", но лишена всех их недостатков (например, как указано в статье, ЗНФ является "слишком нестрогой", а НФБК "характеризуется вычислительной сложностью"). В статье также показано, что алгоритм Бернштейна [12.1] фактически приводит к генерации переменных отношения, которые находятся в НФЭК, но не в ЗНФ.

ГЛАВА 13

Дальнейшая нормализация: нормальные формы более высокого порядка

13.1. ВВЕДЕНИЕ

В предыдущей главе были изложены основные идеи дальнейшей нормализации вплоть до нормальной формы Бойса—Кодда (НФБК) включительно (т.е. до формы, которой можно достичь, используя понятие функциональной зависимости). В этой главе обсуждение вопроса дальнейшей нормализации завершается рассмотрением **четвертой** и **пятой** нормальных форм (4НФ и 5НФ). Как будет показано ниже, для определения 4НФ необходимо ввести понятие **многозначной** зависимости (МЗЗ), которое является обобщением понятия функциональной зависимости. Аналогично, для определения понятия 5НФ необходимо ввести новый тип зависимости, которая называется **зависимостью соединения** (ЗС). Она является обобщением понятия многозначной зависимости. В разделе 13.2 будут рассмотрены

понятия многозначной зависимости и 4НФ, в разделе 13.3 — понятия зависимости соединения и 5НФ; также будет разъяснено, почему 5НФ в некотором смысле можно считать *окончательной* нормальной формой. Сразу же следует отметить, что описание многозначной зависимости и зависимости соединения будет менее формальным и полным, чем описание функциональных зависимостей, данное в главе 11. Более подробную информацию заинтересованный читатель может найти в работах, представленных в списке рекомендуемой литературы в конце этой главы.

После описания основных понятий в разделе 13.4 дается обзор процедуры нормализации в целом, с дополнительными комментариями. Затем в разделе 13.5 кратко обсуждается понятие нормализации. В разделе 13.6 рассматривается еще один важный принцип проектирования — **ортогональное проектирование** (orthogonal design). Наконец, в разделе 13.7 коротко обсуждаются некоторые возможные направления будущих исследований в области нормализации, а заключительный раздел 13.8 представляет собой краткое резюме всей главы.

13.2. МНОГОЗНАЧНЫЕ ЗАВИСИМОСТИ И ЧЕТВЕРТАЯ НОРМАЛЬНАЯ ФОРМА

Пусть дана переменная отношения нстх (где н сокращенно обозначает "иерархический" — hierarchic), содержащая информацию о курсах обучения, преподавателях и учебниках. В этой переменной отношения атрибуты, описывающие преподавателей и учебники, принимают в качестве значений *отношения* (пример значения НСТХ приведен на рис. 13.1). Каждый кортеж переменной отношения НСТХ состоит из атрибута с названиями курсов (COURSE), а также атрибута-отношения с именами преподавателей (TEACHERS) и атрибута-отношения с названиями учебников (TEXTS) (на рис. 13.1 показаны два таких кортежа). Смысл каждого кортежа состоит в том, что соответствующий курс может читать любой из указанных преподавателей с использованием всех указанных учебников. Предположим, что для заданного курса с может быть определено произвольное количество соответствующих преподавателей m и учебников п (m > 0n п > 0). Более того, допустим, хотя этой не совсем реально, что преподаватели и рекомендуемые учебники совершенно не связаны друг с другом. Это означает, что независимо от того, кто преподает данный курс, всегда используется один и тот же набор учебников. Наконец, допустим, что определенный преподаватель или определенный учебник может быть связан с любым количеством курсов.

Пусть необходимо (как в разделе 12.6 главы 12) исключить атрибуты, принимающие отношения в качестве значений. Один из способов выполнения этого (но, возможно, не самый лучший; к этой теме мы вернемся в конце данного раздела) заключается в простой замене переменной отношения НСТХ переменной отношения СТХ с тремя *скалярными* атрибутами, COURSE, TEACHER и TEXT, как показано на рис. 13.2. Как видно из этого рисунка, каждый кортеж исходной переменной отношения нстх порождает *m* * *n* кортежей в переменной отношения СТХ, где *m* и *n* являются значениями кардинальности для отношений TEACHERS и TEXTS в данном кортеже переменной отношения НСТХ. Обратите внимание на то, что все атрибуты результирующей переменной отношения СТХ входят в состав ее ключа (в отличие от переменной отношения нстх, потенциальный ключ которой {COURSE} состоял из единственного атрибута).

Упражнение. Составьте реляционное выражение, с помощью которого можно получить переменную отношения стх из НСТХ.

Рис. 13.2. Набор значений данных в переменной отношения СТХ, эквивалентный

HCTX

COURSE	TEACHERS	TEXTS
Physics	TEACHER: Prof. Green, Prof. Brown	TEXT: Basic Mechanics, Principles of Optics
Math	TEACHER: Prof. Green	TEXT: Basic Mechanics, Vector Analysis, Trigonometry

Рис. 13.1. Пример значений данных в переменной отношения HCTX

CTX

COURSE	TEACHER	TEXT
Physics	Prof. Green	Basic Mechanics
Physics	Prof. Green	Principles of Optics
Physics	Prof. Brown	Basic Mechanics
Physics	Prof. Brown	Principles of Optics
Math	Prof. Green	Basic Mechanics
Math	Prof. Green	Vector Analysis
Math	Prof. Green	Trigonometry

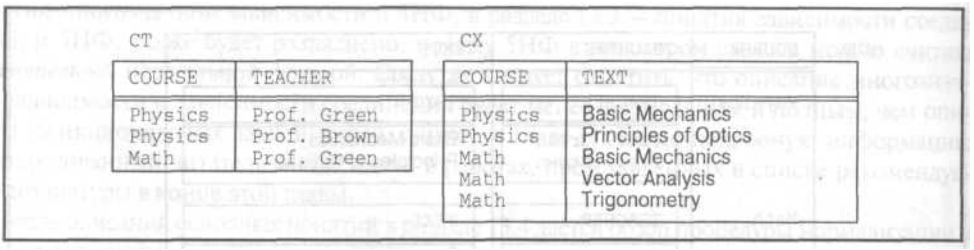
приведенному на рис. 13.1 примеру значений данных в переменной отношения HCTX

Данные, введенные в переменную отношения стх, имеют следующий смысл: кортеж (c, t, x) (в упрощенной системе обозначений) появляется в переменной отношения СТХ тогда и только тогда, когда курс с читается преподавателем t с использованием учебника x. Затем, принимая во внимание то, что для каждого курса указаны все возможные комбинации имен преподавателей и названий учебников, можно утверждать, что для переменной отношения стх верно следующее ограничение.

ЕСЛИ кортежи (c,t1,x1) и (c,t2,x2) присутствуют одновременно,

ТО кортежи (c, t1, x2) и (c, t2 , xl) также присутствуют одновременно.

Очевидно, что переменная отношения СТХ характеризуется значительной избыточностью, а это, как обычно, приводит к некоторым аномалиям обновления. Например, для добавления информации о том, что курс физики может читаться новым преподавателем, необходимо вставить *два* отдельных кортежа, по одному для каждого используемого учебника. Как можно избежать появления таких проблем? Вполне очевидно, что указанные проблемы вызваны тем фактом, что данные о преподавателях и учебниках полностью независимы друг от друга. Кроме того, легко обнаружить, что положение дел может быть улучшено путем декомпозиции переменной отношения стх на две ее проекции (например, с именами ст и сх), соответственно, по атрибутам {COURSE, TEACHER} и {COURSE, TEXT} (рис. 13.3).



CT

COURSE	TEACHER
Physics	Prof. Green
Physics	Prof. Brown
Math	Prof. Green

CX

COURSE	TEXT
Physics	Basic Mechanics
Physics	Principles of Optics
Math	Basic Mechanics
Math	Vector Analysis
Math	Trigonometry

Рис. 13.3. Значения данных переменных отношения CT и CX, которые соответствуют содержанию переменной отношения CTX, показанному на рис. 13.2

Если база данных имеет проект, показанный на рис. 13.3, то для добавления новой информации о том, что курс физики будет читаться новым преподавателем, достаточно вставить единственный кортеж в переменную отношения ст. (Отметим также, что переменная отношения CTX может быть восстановлена за счет обратного соединения проекций ст и сх, и потому данная декомпозиция выполнена без потерь.) Таким образом, вполне разумно было бы предположить, что для переменных отношения, подобных CTX, существует некий способ "дальнейшей нормализации".

На данном этапе читатель мог бы возразить, что с самого начала не было необходимости вводить избыточность в переменную отношения стх, поэтому указанные аномалии обновления также не были неизбежными. Точнее говоря, можно предположить, что для описания некоторого курса в переменную отношения CTX не обязательно включать все возможные комбинации "преподаватель—учебник". Например, двух кортежей вполне достаточно, чтобы показать, что курс физики читают два преподавателя с использованием двух учебников. Но проблема заключается в том, *какие* именно два кортежа следует выбрать? Любой вариант выбора приводит к получению переменной отношения с совершенно неочевидной интерпретацией и довольно странным характером обновления. (Попробуйте подобрать предикат для такой переменной отношения, т.е. задать критерии приемлемости операции обновления того или иного типа для данной переменной отношения.)

В связи с этим с *неформальной* точки зрения очевидно, что переменная отношения CTX спроектирована плохо и ее декомпозиция на проекции ст и сх является более удачным решением. Но проблема в данном случае заключается в том, что с *формальной* точки зрения это совсем не очевидно. Заметим, в частности, что переменная отношения стх вообще не имеет функциональных зависимостей (за исключением таких тривиальных, как COURSE → COURSE). Фактически переменная отношения CTX находится[1] в НФБК, поскольку, как отмечалось ранее, все ее атрибуты входят в состав ее ключа, а любая подобная переменная отношения обязательно находится в НФБК. (Обратите внимание на то, что и проекции ст и сх являются полностью ключевыми, а потому также находятся в **НФБК.)** Следовательно, изложенные в предыдущей главе идеи никак не могут помочь в разрешении этой проблемы.

"Проблемы", которые связаны с переменными отношения в **НФБК,** подобными переменной отношения стх, были замечены достаточно давно, и способы их разрешения

[1] Переменная отношения НСТХ также находится в НФБК, а фактически она дополнительно находится и в 4НФ, и в 5НФ (определения этих нормальных форм приведены ниже в данной главе).

также были вскоре после этого определены, по крайней мере, на интуитивном уровне. Однако эти идеи были сформулированы Фейгином (Fagin) в строгом теоретическом виде с использованием понятия **многозначной зависимости,** или **МЗЗ,** только в 1977 году [13.14]. Многозначную зависимость можно считать обобщением понятия функциональной зависимости в том смысле, что каждая функциональная зависимость является также многозначной зависимостью, но обратное утверждение неверно (поскольку существуют многозначные зависимости, которые не являются функциональными). В случае переменной отношения CTX имеют место две следующие многозначные зависимости.

```
COURSE  →→  TEACHER
COURSE  →→  TEXT
```

Обратите внимание на двойную стрелку, которая в многозначной зависимости A →→ B означает, что **в многозначно зависит от A** или **A многозначно определяет в.** Вначале рассмотрим первую из этих зависимостей, COURSE →→ TEACHER. На интуитивном уровне она означает, что, хотя для каждого курса не существует *одного* соответствующего только ему преподавателя, т.е. *не* выполняется *функциональная* зависимость COURSE → TEACHER, каждый курс имеет вполне определенное *множество* соответствующих преподавателей. Если говорить точнее, под понятием "вполне определенное множество" в нашем случае подразумевается, что для данного курса с и данного учебника x множество преподавателей t, соответствующее паре (c, x) переменной отношения CTX, зависит только от значения c, поскольку не имеет значения, какой именно учебник x будет выбран. Вторая многозначная зависимость, COURSE →→ TEXT, имеет аналогичную интерпретацию.

Ниже приведено формальное определение понятия многозначной зависимости.

- **Многозначная зависимость.** Пусть R - переменная отношения, а A, в и с являются произвольными подмножествами множества атрибутов переменной отношения R. Тогда подмножество в **многозначно зависит** от подмножества A, что символически выражается следующей записью

  A →→ B

  (читается как "A многозначно определяет в" или "A двойная стрелка в"), тогда и только тогда, когда в каждом допустимом значении R множество значений в, соответствующее заданной паре значений A, с, зависит только от значения A и не зависит от значения с.

Нетрудно показать (это описано в работе Фейгина [13.14]), что для данной переменной отношения R{A, в, C} многозначная зависимость A→→ B выполняется тогда и только тогда, когда выполняется также многозначная зависимость A →→ C. Таким образом, многозначные зависимости всегда образуют связанные пары, поэтому обычно их представляют вместе в символическом виде, как показано ниже.

A →→ B | C

Для рассматриваемого примера подобная запись будет иметь следующий вид.

COURSE →→ TEACHER | TEXT

Ранее уже утверждалось, что многозначные зависимости являются обобщениями функциональных зависимостей в том смысле, что каждая функциональная зависимость является многозначной. Точнее говоря, функциональная зависимость — это многозначная

зависимость, в которой множество зависимых значений, соответствующее заданном} значению детерминанта, всегда является одноэлементным множеством. Таким образом если A → в, то, безусловно, A →→ B.

Теперь, возвращаясь к исходной задаче с переменной отношения стх, можно отметить следующее: описанная ранее проблема с переменными отношения этого типа возникает из-за того, что они содержат многозначные зависимости, которые не являются функциональными. (Следует отметить совсем неочевидный факт, что именно наличие таких МЗЗ требует вставки *двух* кортежей, когда необходимо добавить сведения о новом преподавателе физики. Данные два кортежа необходимы для поддержания ограничения целостности, представленного этой МЗЗ.) Проекции ст и сх не содержат многозначных зависимостей, а потому они действительно представляют собой определенное усовершенствование исходной структуры. Поэтому было бы желательно заменить исходную переменную отношения стх двумя рассматриваемыми проекциями. Такое действие будет правомочным в соответствии с теоремой Фейгина [13.14], которая приведена ниже.

- **Теорема Фейгина.** Пусть A, в и C являются множествами атрибутов переменной от ношения R{A, B, C}. В таком случае переменная отношения R будет равна соеди нению ее проекций по атрибутам {A, B} и {A, C} тогда и только тогда, когда для переменной отношения R выполняется многозначная зависимость A →→ B | C.

(Обратите внимание, что эта теорема является более строгой версией теоремы Хита [12.4], которая описана в главе 12.) Теперь, следуя работе Фейгина [13.14], можно дать определение *четвертой нормальной формы.* (Эта нормальная форма получила такое название потому, что в момент ее появления НФБК все еще считалась *третьей* нормальной формой, как отмечено в главе 12.)

- **Четвертая нормальная форма.** Переменная отношения R находится в четвертой **нормальной форме** (4НФ) тогда и только тогда, когда в случае существования таких подмножеств A и B атрибутов этой переменной отношения R, для которых выпол няется нетривиальная многозначная зависимость A →→ в, все атрибуты пере менной отношения R также *функционально* зависят от атрибута A.

Примечание. Многозначная зависимость A →→ в является **тривиальной,** если A является надмножеством в или объединение AB атрибутов A и B составляет весь заголовок.

Иначе говоря, единственные нетривиальные зависимости (функциональные или многозначные) в переменной отношения R находятся в форме K —> X (т.е. имеет место функциональная зависимость некоторого другого атрибута x от определенного суперключа к). Это можно также сформулировать в следующей эквивалентной форме: переменная отношения R находится в 4НФ тогда и только тогда, когда она находится в НФБК и все многозначные зависимости в переменной отношения R фактически представляют собой функциональные зависимости от ее ключей. Обратите внимание на то, что, исходя из этого определения, нахождение в 4НФ предполагает обязательное нахождение в НФБК.

Переменная отношения СТХ не находится в 4НФ, поскольку содержит многозначную зависимость, которая вообще не является функциональной, не говоря уже о том, что последняя должна быть еще и функциональной зависимостью от ключа. Однако обе ее проекции, СТ и СХ, находятся в 4НФ. Следовательно, 4НФ обеспечивает лучшую структуру

данных по сравнению с НФБК, поскольку позволяет исключить некоторые нежелательные зависимости. Кроме того, в [13.14] Фейгин показал, что 4НФ всегда является достижимой, т.е. любая переменная отношения может быть подвергнута декомпозиции без потерь в эквивалентный набор переменных отношения в 4НФ. Однако, как показано в разделе 12.5 главы 12 на примере переменной отношения SJT, такая декомпозиция (или даже декомпозиция до НФБК) не всегда оказывается полезной и нужной.

Следует отметить, что хотя идеи Риссанена (Rissanen), изложенные в посвященной независимым проекциям работе [12.6], сформулированы с использованием функциональных зависимостей, они также справедливы в отношении многозначных зависимостей. Напомним, что в соответствии с этими идеями, переменную отношения R{A, B, C}, удовлетворяющую функциональным зависимостям A —> *B* и B —> *C*, желательно разбивать на проекции по атрибутам {A, B} и {B, C}, а не на проекции по атрибутам {A, B} и {A, C}. Это утверждение также будет верно, если вместо функциональных зависимостей A →B и B → с использовать многозначные зависимости A →→ B и B →→ C.

В заключение, как было обещано, вернемся к вопросу об устранении атрибутов со значениями в виде отношений, или сокращенно АО (атрибутов-отношений). Суть в том, что если приходится иметь дело с переменной отношения наподобие НСТХ, которая включает один или несколько независимых АО, то вместо простой замены этих АО скалярными атрибутами (как было сделано в начале данного раздела), а затем выполнения декомпозиции без потерь полученного результата, *лучше вначале разделить эти АО.* Например, в случае переменной отношения НСТХ прежде всего следует заменить исходную переменную отношения двумя ее проекциями нет {COURSE, TEACHERS} и нсх{COURSE, TEXTS}, где атрибуты TEACHERS и TEXTS все еще относятся к типу АО. Далее эти АО можно будет исключить из двух полученных проекций (с приведением их к НФБК, а фактически к 4НФ) обычным способом, и тогда "проблема", свойственная находящейся в НФБК переменной отношения СТХ, просто никогда не возникнет. Как видите, понятия многозначных зависимостей и 4НФ предоставляют формальное обоснование тех способов усовершенствования проекта, которые в противном случае были бы основаны исключительно на эмпирических правилах.

13.3. ЗАВИСИМОСТИ СОЕДИНЕНИЯ И ПЯТАЯ НОРМАЛЬНАЯ ФОРМА

До сих пор в настоящей главе и на протяжении всей предыдущей главы по умолчанию предполагалось, что единственной необходимой или допустимой операцией в процессе нормализации является замена переменной отношения по правилам декомпозиции без потерь *точно двумя* ее проекциями. Такое допущение нас вполне устраивало, пока речь не шла о 4НФ. Однако, хотя это может показаться удивительным, существуют переменные отношения, для которых нельзя выполнить декомпозицию без потерь на две проекции, но которые *можно* подвергнуть декомпозиции без потерь на три или большее количество проекций. Подобные переменные отношения обозначим не очень удачным, но достаточно удобным термином "n-декомпонуемая переменная отношения, или отношение". Такое название применяется к переменной отношения, если можно выполнить ее декомпозицию без потерь нап проекций, но не наш проекций, где 1 < m и m < п.

Примечание. Впервые возможность n-декомпонуемости для n > 2 была упомянута в работе Ахо (Aho), Бери (Beeri) и Ульмана (Ullman) [13.1], а частный случай для n = 3 был описан Николасом (Nicolas) [13.26].

В качестве примера рассмотрим переменную отношения SPJ из базы данных поставщиков, деталей и проектов, представленную на рис. 13.4 (в целях упрощения атрибут QTY исключен). Обратите внимание на то, что эта переменная отношения состоит только из ключевых атрибутов, не содержит нетривиальных функциональных и многозначных зависимостей и поэтому находится в 4НФ. Заметим также, что на этом рисунке показаны следующие компоненты.

1. Три бинарные проекции (SP, PJ и JS), соответствующие значению отношения SPJ, показанному в верхней части рисунка.
2. Результат соединения проекций SP и PJ ПО атрибуту p#.
3. Результат соединения этого результата с проекцией JS по комбинации атрибутов **J# И S#.**

Обратите внимание на то, что в результате первого соединения получается копия исходной переменной отношения SPJ с одним дополнительным (фиктивным) кортежем, а в результате второго соединения этот дополнительный кортеж исключается и восстанавливается первоначальное отношение SPJ. Иначе говоря, исходная переменная отношения SPJ является 3-декомпонуемой.

Примечание. Независимо от того, какая пара проекций будет выбрана для первого соединения, в итоге будет получен один результат, хотя промежуточные результаты будут в каждом случае разными.

Упражнение. Предлагаем читателю проверить это утверждение.

Далее отметим, что представленный на рис. 13.4 пример, безусловно, выражен в терминах *отношений,* а не *переменных отношения.* Однако 3-декомпонуемость переменной отношения SPJ может быть более фундаментальным и не зависящим от времени свойством (т.е. свойством, которое соблюдается при всех допустимых значениях данной переменной отношения), *если* данная переменная отношения удовлетворяет определенному, не зависящему от времени ограничению целостности. Для того чтобы понять, каким именно должно быть это ограничение, прежде всего отметим, что утверждение "переменная отношения SPJ равна соединению трех своих проекций SP, PJ и JS", полностью эквивалентно следующему утверждению:

- если пара (s1, p1) присутствует в SP
  и пара(p1, j1) присутствует в PJ, а
  пара (j1, s1) присутствует в JS,
- то тройка (s1, p1, j1) присутствует в SPJ.

Это верно, поскольку очевидно, что тройка (s1, p1, j1) обязательно присутствует в соединении проекций SP, PJ и JS. (Следует отметить, что обратное утверждение, т.е. если тройка (s1, p1, j1) присутствует в переменной отношения SPJ, то, например, пара (s1, p1) присутствует в проекции SP, является истинным для любой переменной отношения SPJ степени три.) Поскольку пара (s1, p1) присутствует в отношении SP тогда и только тогда, когда тройка (s1, p1, j2) присутствует в отношении SPJ для

некоторого значения j2 (аналогично для (p1, j1) и (j1, s1)), приведенное выше утверждение можно переписать в виде следующего ограничения, налагаемого на переменную отношения SPJ:

- если кортежи (s1, p1, j2),(s2, p1, j1),(s1, p2, jl) присутствуют в SPJ,
- то кортеж (s1, pi, j 1) также присутствует в SP J.

Если это утверждение выполняется всегда, т.е. для всех допустимых значений переменной отношения SPJ, то будет получено не зависящее от времени (хотя и несколько странное) ограничение для данной переменной отношения. Обратите внимание на циклический характер этого ограничения ("если значение s1 связано с p1 и p1 связано с j 1, а j 1 связано опять с s1, то s1, p1 и j1 должны находиться в одном кортеже"). *Переменная отношения будет n-декомпонуемой для* n>2 *тогда и только тогда, когда она удовлетворяет некоторому подобному циклическому ограничению (многостороннему, с количеством участников n).*

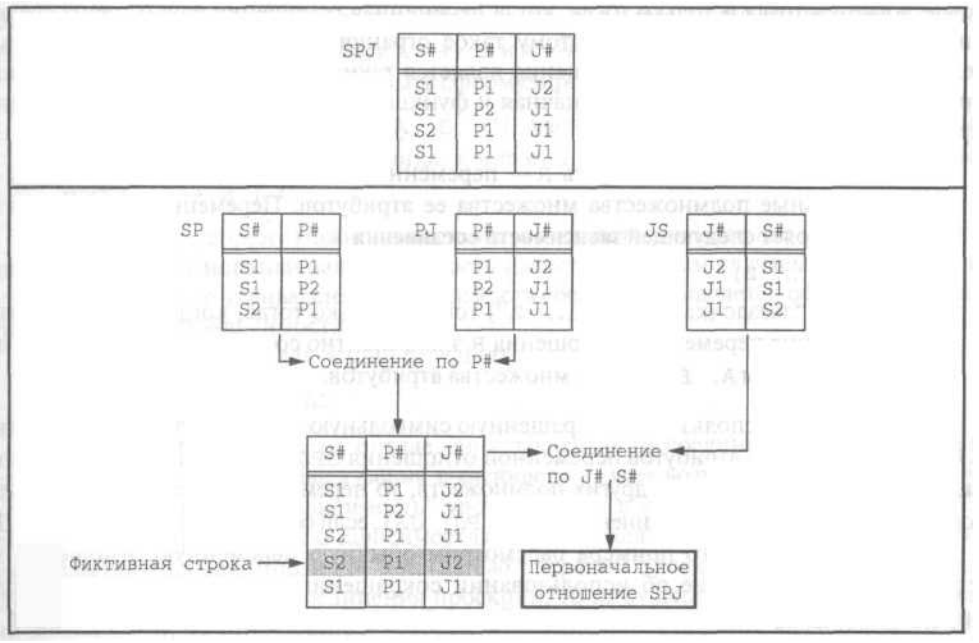


Рис. 13.4. Пример значения переменной отношения SPJ, которая может быть получена только в результате соединения всех трех ее бинарных проекций, но не л юбых двух из них

До конца этого раздела будем руководствоваться предположением, что переменная отношения SPJ действительно удовлетворяет этому не зависящему от времени ограничению (представленный на рис. 13.4 пример данных соответствует такой гипотезе). Такое ограничение будем кратко называть *ограничением 3-Д* (сокращение от 3-декомпонуемости). Что означает ограничение 3-Д с практической точки зрения? Для получения ответа на этот вопрос рассмотрим пример, в котором под такими ограничениями подразумевается, что *если* в реальном мире для переменной отношения SP J верны следующие утверждения:

а) *если* Смит поставляет гаечные ключи;

б) гаечные ключи используются в Манхэттенском проекте;

в) Смит является поставщиком для Манхэттенского проекта;

то

г) Смит поставляет гаечные ключи для Манхэттенского проекта.

Обратите внимание на то, что (как уже упоминалось в главе 1, раздел 1.3) из взятых в совокупности утверждений *а, б* и *в* обычно *не* следует утверждение *г.* Действительно, точно такой же пример был рассмотрен в главе 1 для демонстрации "дефекта соединения". Однако в данном частном случае следует отметить, *что никакого дефекта здесь не возникает,* поскольку существует дополнительное практическое ограничение 3-Д, имеющее место в реальном мире, благодаря чему вывод утверждения *г* на основе утверждений *а, бив* является вполне правомочным.

Возвращаясь к главной теме данного обсуждения, отметим, что ограничение 3-Д удовлетворяется тогда и только тогда, когда переменная отношения равнозначна соединению некоторых ее проекций, поэтому такое ограничение называется **зависимостью соединения** (ЗС). Зависимость соединения является таким же ограничением для данной переменной отношения, как многозначная и функциональная зависимости. Ниже дано определение этого понятия.

■ **Зависимость соединения.** Пусть R— переменная отношения, а A, B, ..., Z — произвольные подмножества множества ее атрибутов. Переменная отношения R удовлетворяет следующей **зависимости соединения**

***{A, B, …, Z}**

(читается "звездочка A, в, ..., Z") тогда и только тогда, когда любое допустимое значение переменной отношения R эквивалентно соединению ее проекций по подмножествам A, в, …, z множества атрибутов.

Например, если использовать сокращенную символьную запись SP для подмножества {S#, P#} множества атрибутов переменной отношения SPJ и аналогично использовать сокращения PJ и JS для двух других подмножеств, то переменная отношения SPJ будет удовлетворять зависимости соединения * {SP, P J, JS}, если соблюдается ограничение 3-Д.

В качестве еще одного примера рассмотрим обычную переменную отношения S. Если принято соглашение об использовании сокращения SN для обозначения подмножества {S#,SNAME} множества атрибутов S и аналогичное соглашение для ST и SC, то можно сказать, что переменная отношения S удовлетворяет зависимости соединения *{SN, ST, SC}.

Отсюда ясно, что переменная отношения SPJ с зависимостью соединения *{SP, PJ, JS} может быть 3-декомпонуемой. Однако возникает вопрос, *следует ли* выполнять такую декомпозицию? По всей видимости, следует, так как в связи с наличием зависимости соединения переменная отношения SPJ характеризуется многочисленными аномалиями обновления, которые можно устранить лишь с помощью ее 3-декомпозиции. Некоторые примеры подобных аномалий показаны на рис. 13.5. Предлагаем читателям в качестве упражнения самостоятельно ответить на вопрос, что произойдет после выполнения 3-декомпозиции.

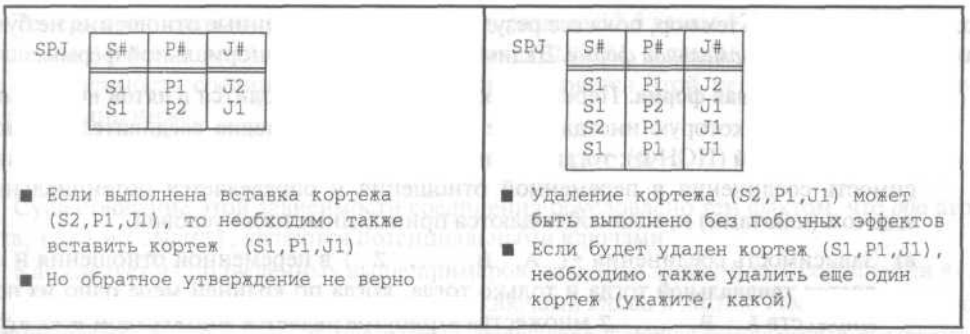


Рис. 13.5. Примеры аномалий обновления в переменной отношения SPJ

Теорема Фейгина (которая рассматривалась в разделе 13.2) гласит, что переменная отношения R{A, в, C} может быть декомпонована без потерь на проекции по атрибутам {A, B} и {A, c} тогда и только тогда, когда для переменной отношения R выполняются многозначные зависимости A →→ B и A →→ C.

Теперь *теорема Фейгина* может быть сформулирована иначе, как показано ниже.

- Переменная отношения R{A, в, C} удовлетворяет зависимости соединения *{AB, AC} тогда и только тогда, когда она удовлетворяет многозначной зависимости A →→ B | C.

Поскольку эту теорему можно использовать в качестве *определения* многозначной зависимости, либо многозначная зависимость является частным случаем зависимости соединения, либо (что эквивалентно) зависимость соединения является обобщением понятия многозначной зависимости.

Формально получим следующий результат.

A →→ B | C ≡ *{ AB, AC }

Примечание. Более того, из определения зависимости соединения непосредственно следует, что это **наиболее общая форма зависимости** из всех возможных (безусловно, имеется в виду, что термин "зависимость" употребляется в узко специальном смысле). Это означает, что если речь идет о зависимостях, касающихся переменных отношения, которые подвергаются декомпозиции с помощью операции проекции и обратной композиции с помощью операции соединения проекций, то не существует более высокой степени зависимости, по отношению к которой зависимость соединения является всего лишь частным случаем. (Но если ввести другие операции декомпозиции и композиции, то возможно появление других типов зависимостей, которые будут кратко описаны в разделе 13.7.)

Возвращаясь к рассматриваемому примеру, можно отметить, что проблема, связанная с использованием переменной отношения SPJ, состоит в том, что она подчиняется зависимости соединения, которая не является многозначной зависимостью, и поэтому также не является функциональной. *{Упражнение.* Объясните, *почему* такая ситуация рассматривается как проблема?) Можно также заметить, что возможно (и даже желательно) декомпоновать такую переменную отношения на меньшие компоненты, а именно — на проекции, определяемые зависимостью соединения. Данный процесс декомпозиции

может повторяться до тех пор, пока все результирующие переменные отношения не буду! находиться в *пятой нормальной форме.* Дадим определение этой нормальной формы.

- **Пятая нормальная форма.** Переменная отношения R находится в **пятой нормально» форме** (5НФ), которую иногда иначе называют **проекционно-соединительной нормальной формой** (ПСНФ), тогда и только тогда, когда каждая нетривиальная зависимость соединения в переменной отношения R определяется потенциальным ключом (ключами) R, если соблюдаются приведенные ниже условия.

  а) Зависимость соединения *{ A, в, ..., z } в переменной отношения R яв ляется **тривиальной** тогда и только тогда, когда по крайней мере одно из под множеств A, в, ..., z множества атрибутов является множеством всех атри бутов R.

  б) Зависимость соединения *{ A, в, ..., Z } в переменной отношения R **оп ределяется потенциальным ключом (ключами)** R тогда и только тогда, когда каж
  дое из подмножеств A, B, ..., Z множества атрибутов является суперключом
  для R.

Переменная отношения SPJ не находится в 5НФ. Она удовлетворяет некоторой зависимости соединения, а именно — ограничению 3-Д, которое, безусловно, не определяется ее единственным потенциальным ключом (этот ключ является комбинацией всех ее атрибутов). Иначе говоря, переменная отношения SPJ не находится в 5НФ, поскольку она *может* быть 3-декомпонована и возможность такой декомпозиции не обусловлена тем фактом, что комбинация атрибутов {s#, P#, J#} является ее потенциальным ключом. Напротив, после 3-декомпозиции три проекции SP, PJ и JS находятся в 5НФ, поскольку в них вообще нет нетривиальных зависимостей соединения.

Обратите внимание на тот факт, что любая переменная отношения в 5НФ безусловно находится также в 4НФ, поскольку многозначная зависимость является частным случаем зависимости соединения. Действительно, Фейгин в [13.15] показал, что любая многозначная зависимость, определяемая потенциальным ключом, на самом деле должна быть функциональной зависимостью, в которой потенциальный ключ является детерминантом. В той же работе Фейгин показал, что любая переменная отношения может быть подвергнута декомпозиции без потерь на эквивалентный набор переменных отношения в 5НФ, т.е. 5НФ всегда достижима.

Теперь более подробно рассмотрим вопрос о том, что означает утверждение "зависимость соединения определяется потенциальными ключами". Еще раз рассмотрим знакомую переменную отношения поставщиков s. Такая переменная отношения удовлетворяет нескольким зависимостям соединения, в частности следующей зависимэсти.

*{ { S#, SNAME, STATUS }, { S#, CITY } }

Это означает, что переменная отношения S эквивалентна соединению ее проекций по атрибутам {S#, SNAME, STATUS} и {S#, CITY}. Поэтому она может быть подвергнута декомпозиции без потерь на указанные проекции. (Этот факт означает не то, что она *должна* подвергаться подобной декомпозиции, а только то, что *может* быть ей подвергнута.) Существование данной зависимости соединения предполагается на основании того факта, что атрибут {S#} является потенциальным ключом данной переменной отношения (в действительности, это следует из теоремы Хита [12.4]).

Теперь предположим (как было сделано в разделе 12.5 главы 12), что переменная отношения S имеет второй потенциальный ключ, {SNAME}. В таком случае существует еще одна зависимость соединения, которая удовлетворяется этой переменной отношения, как показано ниже.

```
*{ { S#, SNAME }, { S#, STATUS }, { SNAME, CITY } }
```

Существование этой зависимости соединения обусловлено тем фактом, что *оба* атрибута, {S#} и {SNAME}, являются потенциальными ключами.

Как следует из приведенных выше примеров, заданная зависимость соединения *{A, B, ..., Z} определяется потенциальными ключами *тогда и только тогда, когда каждое подмножество* A, B, ..., Z *множества атрибутов фактически является суперключом для данной переменной отношения.* Таким образом, относительно заданной переменной отношения R можно утверждать, что она находится в 5НФ, только при условии, что известны все ее потенциальные ключи и **все зависимости соединения,** существующие в ней. Но сама по себе задача определения всех подобных зависимостей соединения может оказаться очень сложной. Это означает, что можно относительно легко определить функциональные и многозначные зависимости (поскольку они имеют довольно простую интерпретацию в реальном мире), но этого нельзя утверждать по отношению к зависимостям соединения (т.е. к таким зависимостям соединения, которые не являются многозначными и поэтому функциональными зависимостями), поскольку интуитивный смысл зависимостей соединения не всегда бывает очевидным. Следовательно, процедура определения того, что некоторая переменная отношения все еще находится в 4НФ, а не в 5НФ, и, таким образом, существует возможность ее дальнейшей выгодной декомпозиции, *все еще остается не вполне ясной.* Однако, как следует из опыта, подобные переменные отношения довольно редко встречаются на практике.

В заключение заметим, что, как следует из определения, 5НФ является **окончательной нормальной формой** по отношению к операциям проекции и соединения (что отражено в ее альтернативном названии — *проекционно-соединительная* нормальная форма). Таким образом, если переменная отношения находится в 5НФ, то **гарантируется, что она не содержит аномалий,** которые могут быть исключены посредством ее разбиения на проекции. (Безусловно, это замечание не означает, что такая переменная отношения свободна от аномалий; еще раз подчеркнем, что это означает *отсутствие в ней аномалий,* которые могут быть устранены с использованием операции проекции.) Если переменная отношения находится в 5НФ, то единственными в ней являются те зависимости соединения, которые определяются ее потенциальными ключами, и тогда единственными допустимыми декомпозициями будут декомпозиции, которые основаны на этих потенциальных ключах. (Каждая проекция в подобной декомпозиции будет состоять из одного или нескольких потенциальных ключей в сочетании с дополнительными атрибутами в количестве от нуля или больше.) Например, переменная отношения поставщиков s находится в 5НФ. Как упоминалось выше, эта переменная отношения *может* быть подвергнута дальнейшей декомпозиции без потерь, причем в нескольких вариантах, но каждая проекция в любом из этих вариантов по-прежнему будет содержать один из исходных потенциальных ключей. Следовательно, подобная декомпозиция не даст никаких дополнительных преимуществ.

13.4. ОБЩАЯ СХЕМА ПРОЦЕДУРЫ НОРМАЛИЗАЦИИ

До настоящего раздела в этой (и предшествующей) главе рассматривалась технология *декомпозиции без потерь,* предназначенная для использования в процессе проектирования базы данных. Основная идея состоит в следующем. Пусть дана некоторая переменная отношения R, представленная в 1НФ, в совокупности с набором определенных для нее функциональных зависимостей, многозначных зависимостей и зависимостей соединения. Задача заключается в систематическом разбиении исходной переменной отношения R на такой набор меньших (т.е. имеющих меньшую степень) переменных отношения, который в некотором заданном смысле будет эквивалентен переменной отношения R, но с определенной точки зрения будет также более предпочтительным[2]. Каждый этап процесса такого преобразования заключается в разбиении на проекции переменных отношения, полученных на предыдущем этапе. При этом на каждом этапе преобразования существующие ограничения используются для выбора тех проекций, которые будут получены в этот раз. Весь процесс можно неформально определить с помощью перечисленных ниже правил.

1. Переменную отношения в 1НФ следует разбить на такие проекции, которые по зволят исключить все функциональные зависимости, не являющиеся неприводи мыми. В результате будет получен набор переменных отношения в 2НФ.
2. Полученные переменные отношения в 2НФ следует разбить на такие проекции, которые позволят исключить все существующие транзитивные функциональные зависимости. В результате будет получен набор переменных отношения в ЗНФ.
3. Полученные переменные отношения в ЗНФ следует разбить на проекции, позво ляющие исключить все оставшиеся функциональные зависимости, в которых де терминанты не являются потенциальными ключами. В результате такого приведе ния будет получен набор переменных отношения в НФБК.

   Примечание. Правила 1—3 могут быть объединены в одно: "Исходную переменную отношения следует разбить на проекции, позволяющие исключить все функциональные зависимости, в которых детерминанты не являются потенциальными ключами".
4. Полученные переменные отношения в НФБК следует разбить на проекции, позво ляющие исключить все многозначные зависимости, которые не являются также функциональными. В результате будет получен набор переменных отношения в 4НФ. *Примечание.* На практике такие многозначные зависимости обычно исключаются *перед* выполнением этапов 1-3 (на этапе "устранения независимых МЗЗ"), как описано в разделе 13.2.
5. Полученные переменные отношения в 4НФ следует разбить на проекции, позво ляющие исключить все зависимости соединения, которые не определяются потен циальными ключами (хотя в данном случае в определение следовало бы добавить фразу "если удастся их выявить"). В результате будет получен набор переменных от ношения в 5НФ.

---

[2] Если переменная отношения R включает какие-либо (необходимые) атрибуты со значением в виде отношения, то предполагается, что такая ее структура была принята сознательно. Нежелательные атрибуты со значением в виде отношения могут быть устранены, как было описано в разделе 13.2.

По поводу приведенных выше правил можно сделать несколько дополнительных замечаний.

1. Процесс разбиения на проекции на каждом этапе должен быть выполнен без потерь и с сохранением зависимостей (там, где это возможно).
2. Обратите внимание, что существует довольно привлекательный набор следующих альтернативных определений НФБК, 4НФ и 5НФ (как было впервые отмечено Фейгином в [13.15]):
   - переменная отношения R находится в НФБК тогда и только тогда, когда каждая функциональная зависимость, удовлетворяемая переменной отношения R, определяется ее потенциальными ключами;
   - переменная отношения R находится в 4НФ тогда и только тогда, когда каждая многозначная зависимость, удовлетворяемая переменной отношения R, опре деляется ее потенциальными ключами;
   - переменная отношения R находится в 5НФ тогда и только тогда, когда каждая зависимость соединения, удовлетворяемая переменной отношения R, опреде ляется ее потенциальными ключами.

   Аномалии обновления, обсуждавшиеся в главе 12 и в предыдущих разделах данной главы, были вызваны именно теми функциональными зависимостями, многозначными зависимостями или зависимостями соединения, которые не определялись потенциальными ключами. (Подразумевается, что все упомянутые здесь функциональные, многозначные зависимости и зависимости соединения являются нетривиальными.)
3. Общее назначение процесса нормализации заключается в следующем:
   - исключение некоторых типов избыточности;
   - устранение некоторых аномалий обновления;
   - разработка проекта базы данных, который является достаточно "качествен ным" представлением реального мира, интуитивно понятен и может служить хорошей основой для последующего расширения;
   - упрощение процедуры применения необходимых ограничений целостности.

   Последний пункт данного списка следует рассмотреть отдельно. Общая идея (как отмечалось в других главах этой книги) состоит в том, что из *одних ограничений целостности следуют другие.* В качестве простейшего примера можно привести ограничение для суммы зарплаты, которая должна быть выше 10 000 долл., а следовательно, выше нуля. Таким образом, если из ограничения A следует ограничение в, то соблюдение *ограничения* A *автоматически влечет за собой соблюдение ограничения* B (поэтому даже нет необходимости явно задавать ограничение в, за исключением того, что оно может быть упомянуто в комментарии). Тогда приведение к 5НФ представляет собой простой способ наложения некоторых важных и весьма распространенных ограничений. Главное — обеспечить поддержку уникальности потенциальных ключей, после чего все зависимости соединения (а также все многозначные и функциональные зависимости) будут реализованы СУБД автоматически, поскольку все они определяются потенциальными ключами.

4. Необходимо вновь подчеркнуть тот факт, что данные рекомендации по поводу нормализации являются всего лишь рекомендациями и, вероятно, могут сущест вовать соображения, по которым нормализацию не следует выполнять до конца. Классическим примером ситуации, когда полная нормализация *не желательна,* является переменная отношения имени и адреса NADDR (см. упр. 12.7 главы 12), хотя следует признать, что этот пример не совсем убедителен. Как правило, нор мализацию рекомендуется выполнять полностью.
5. Необходимо еще раз повторить сделанное в главе 12 замечание о том, что понятия зависимости и дальнейшей нормализации по своему характеру являются семанти ческими, т.е. они связаны со *смыслом* данных, тогда как реляционная алгебра и ре ляционное исчисление, а также построенные на их основе языки наподобие SQL, наоборот, имеют дело со *значениями* данных и не требуют (да и не могут требовать) выполнения нормализации выше первого уровня. Рекомендации по выполнению дальнейшей нормализации должны рассматриваться прежде всего как методика, позволяющая разработчику базы данных (и, следовательно, ее пользователю) вы разить определенную часть семантики реального мира (пусть даже небольшую) в простой и понятной форме.
6. Исходя из сказанного выше, необходимо отметить, что идеи нормализации чрез вычайно полезны для проектирования баз данных, но они отнюдь не являются универсальным средством. Ниже перечислены некоторые причины подобного по ложения дел (этот список дополнен в [13.9]).

   - Нормализация (как упоминалось выше, в главе 9) действительно позволяет реализовать (в очень простой форме) определенные ограничения целостности, но на практике, помимо зависимостей соединения, функциональных и многозначных зависимостей, существуют и другие типы ограничений.
   - Декомпозиция может быть неуникальной (как правило, имеется несколько способов приведения заданного набора переменных отношения к 5НФ), но существует очень мало объективных критериев, позволяющих выбрать один из альтернативных вариантов декомпозиции.
   - Как упоминалось в разделе 12.5 (пример декомпозиции переменной отноше ния SJT), преследование одновременно двух целей (т.е. приведение к НФБК и сохранение зависимостей) в некоторых случаях приводит к конфликтной си туации.
   - Процедура нормализации позволяет избавиться от избыточности за счет раз биения на проекции, но не всякую избыточность можно устранить таким обра зом (это — "проблема переменной отношения CTXD", см. аннотацию к [13.13]).

   Следует также отметить, что хорошие методики нисходящего проектирования позволяют тем или иным конкретным способом создавать полностью нормализованный проект базы данных (см. главу 14).

13.5. ОБЩИЕ СВЕДЕНИЯ О ДЕНОРМАЛИЗАЦИИ

До сих пор в этой (и предыдущей) главе в основном предполагалось, что полная нормализация вплоть до 5НФ весьма желательна. Но на практике часто можно слышать утверждения, что для достижения высокой производительности системы иногда следует

выполнить денормализацию. При этом используются доводы, подобные перечисленным ниже.

1. Полная нормализация приводит к появлению большого количества логически не зависимых переменных отношения (и предполагается, что рассматриваемые пере менные отношения являются базовыми).
2. Большое количество логически независимых переменных отношения приводит к появлению большого количества отдельно хранимых физических файлов.
3. Большое количество отдельно хранимых физических файлов приводит к появле нию большого количества операций ввода-вывода.

Строго говоря, эти доводы, конечно же, не верны, поскольку (как многократно отмечалось в данной книге) в определении реляционной модели нигде не утверждается, что базовые переменные отношения должны находиться во взаимно однозначном соответствии с хранимыми файлами. Поэтому *денормализацию в случае необходимости следует выполнять на уровне хранимых файлов, но не на уровне базовых переменных отношения*[3]. Однако в некотором отношении эти доводы все же верны для современных продуктов SQL, поскольку именно в них эти два уровня не разделены в требуемой степени. В данном разделе понятие "денормализация" будет рассмотрено более подробно.

Примечание. Материал этого раздела в значительной степени основан на [13.6].

Общее определение денормализации

Напомним, что нормализация *переменной отношения R означает ее замену множеством* таких проекций Rl, R2, ..., Rn, что результатом обратного соединения проекций Rl, R2, ..., Rn обязательно будет значение R. Конечной целью нормализации является *сокращение степени избыточности данных* за счет приведения проекций Rl, R2, ..., Rn к максимально высокому уровню нормализации.

Теперь можно перейти к определению понятия денормализации. Пусть Rl, R2, Rn является множеством переменных отношения. Тогда **денормализацией** этих переменных отношения называется такая замена переменных отношения их соединением R, что для всех возможных значений i (где i = 1, ..., п) выполнение проекции R по атрибутам Ri обязательно снова приводит к созданию значений Ri. Конечной целью денормализации является *увеличение степени избыточности данных* за счет приведения переменной отношения R к более низкому уровню нормализации по сравнению с исходными переменными отношения Rl, R2, ..., Rn. Точнее, преследуется цель сократить количество соединений, которые потребуется выполнять в приложении на этапе прогона, поскольку (в действительности) некоторые из этих соединений уже выполнены заранее в составе работ по проектированию базы данных.

---

[3] Это замечание фактически является не совсем точным; денормализация — это операция с переменными отношения, а не с хранимыми файлами и поэтому не может применяться "на уровне хранимых файлов". Но предположение о том, что некоторый аналог денормализации может выполняться на уровне хранимых файлов, отнюдь не лишено смысла.

Рис. 13.6. Денормализованные данные о деталях и поставках

В качестве примера рассмотрим денормализацию переменных отношения деталей и поставок для получения переменной отношения[4] PSQ, представленной на рис. 13.6. Следует отметить, что переменная отношения PSQ находится в 1НФ, а не в 2НФ.

PSQ

P#	PNAME	COLOR	WEIGHT	CITY	S#	QTY
P1	Nut	Red	12.0	London	S1	300
P1	Nut	Red	12.0	London	S2	300
P2	Bolt	Green	17.0	Paris	S1	200
..	.....	.....	.....	.....	..	...
P6	Cog	Red	19.0	London	S1	100

Некоторые проблемы денормализации

Использование понятия денормализации связано с некоторыми вполне очевидными проблемами. Первая из них заключается в том, что начиная денормализацию, трудно сказать, когда ее следует прекратить. В случае выполнения нормализации существуют ясные логические критерии ее продолжения до тех пор, пока не будет достигнута самая высокая из возможных нормальных форм. Следует ли при выполнении денормализации стремиться к тому, чтобы достичь самой низкой из возможных нормальных форм? Безусловно, нет, поэтому не существует никаких *логических* критериев точного определения момента прекращения этого процесса. Иначе говоря, в случае денормализации прежний подход (применявшийся при нормализации), созданный на основании строго научной и логичной теории, заменяется чисто прагматическим и субъективным подходом.

Второе очевидное затруднение связано с проблемами избыточности и аномалиями обновления, которые возникают из-за того, что приходится иметь дело с не полностью нормализованными переменными отношения. Эти проблемы достаточно подробно обсуждались выше. Но менее очевидной является проблема *выборки* данных, т.е. денормализация может существенно усложнить выполнение некоторых запросов. Рассмотрим в качестве примера следующий запрос: "Найти средний вес всех деталей определенного цвета". При использовании обычного нормализованного проекта наиболее подходящая формулировка данного запроса будет выглядеть следующим образом.

```
SUMMARIZE P BY { COLOR } ADD AVG ( WEIGHT ) AS AVWT
```

Но при использовании денормализованного проекта, показанного на рис. 13.6, эта формулировка, которая показана ниже, становится несколько сложнее (не говоря уже о том, что в такой ситуации предполагается наличие по крайней мере одной поставки для каждой детали, что представляет собой достаточно смелое и в общем неверное допущение).

---

[4] При использовании нашего традиционного примера с переменными отношения *поставщиков* и *поставок,* в случае денормализации может возникнуть проблема, связанная с тем, что в результате соединения будет утрачена информация о поставщике с номером S5. По этой причине можно предположить, что в процессе денормализации следует использовать операции "внешних" соединений. Однако, как показано в главе 19, само по себе использование внешних соединений также сопровождается возникновением определенных проблем.

```
SUMMARIZE PSQ { P#, COLOR, WEIGHT } BY { COLOR
            } ADD AVG ( WEIGHT ) AS AVWT
```

(Обратите внимание на то, что во второй формулировке запрос, вероятно, будет к тому же выполняться с более низкой производительностью.) Иначе говоря, общепринятое мнение о том, что денормализация "хороша для выборки, но плоха для обновления", является неверным, как по соображениям практической применимости, так и по соображениям производительности.

Третья, и самая главная, проблема формулируется следующим образом. (Это относится к "правильной" денормализации, т.е. к денормализации, которая выполняется только на физическом уровне, а также к тому типу денормализации, которую иногда приходится осуществлять в современных продуктах SQL.) Когда речь идет о том, что денормализация "способствует достижению высокой производительности", фактически подразумевается, что она способствует достижению высокой *производительности некоторых конкретных приложений.* Любая выбранная физическая структура, которая прекрасно подходит для одних приложений с точки зрения их производительности, может оказаться совершенно непригодной для других. Например, предположим, что каждая базовая переменная отношения отображается на один физически хранимый файл, а каждый хранимый файл состоит из физически смежного набора хранимых записей, по одной для каждого кортежа соответствующей переменной отношения. Тогда применительно к данной структуре можно сделать приведенные ниже замечания.

- Допустим, что соединение отношений поставщиков, поставок и деталей представлено в виде одной переменной отношения, т.е. в виде одного хранимого файла. Тогда запрос "Получить сведения обо всех поставщиках деталей красного цвета" благодаря выбранной физической структуре будет, по-видимому, выполняться достаточно эффективно.
- Однако запрос "Получить сведения о поставщиках из Лондона" при такой физической структуре будет выполняться менее эффективно по сравнению со структурой, состоящей из трех отдельных базовых переменных отношения, каждая из которых отображена на физически отдельный хранимый файл. Дело в том, что в последнем случае все записи с данными о поставщиках являются физически смежными, а в первом случае данные будут распределены на большем участке устройства хранения и для их выборки потребуется существенно больше операций ввода—вывода. Аналогичные замечания можно высказать по отношению ко всем запросам, в которых доступ осуществляется только к данным о поставщиках, деталях или поставках по отдельности (без выполнения операции соединения).

13.6. ОРТОГОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ (НЕБОЛЬШОЕ ОТСТУПЛЕНИЕ ОТ ТЕМЫ)

В этом разделе, следуя описанию, приведенному в [13.12], кратко рассматривается еще один принцип проектирования баз данных, который напрямую не связан с принципом дальнейшей нормализации, но очень похож на него тем, что также является *научно обоснованным.* Он называется принципом ортогонального проектирования (principle of orthogonal design). Обратимся к рис. 13.7, на котором представлен, безусловно, плохой, но вполне допустимый проект представления данных о поставщиках. Здесь переменная отношения SA соответствует поставщикам, которые находятся в Париже, а переменная

отношения SB — поставщикам, которые либо не находятся в Париже, либо имеют статус выше 3 0 (т.е. предикаты этих переменных отношения имеют именно такой смысл). Как следует из данного рисунка, подобный проект характеризуется некоторой избыточностью, точнее говоря, в нем дважды представлен кортеж для поставщика с номером S3 (по одному в каждой переменной отношения), а такая избыточность также приводит к аномалиям обновления.

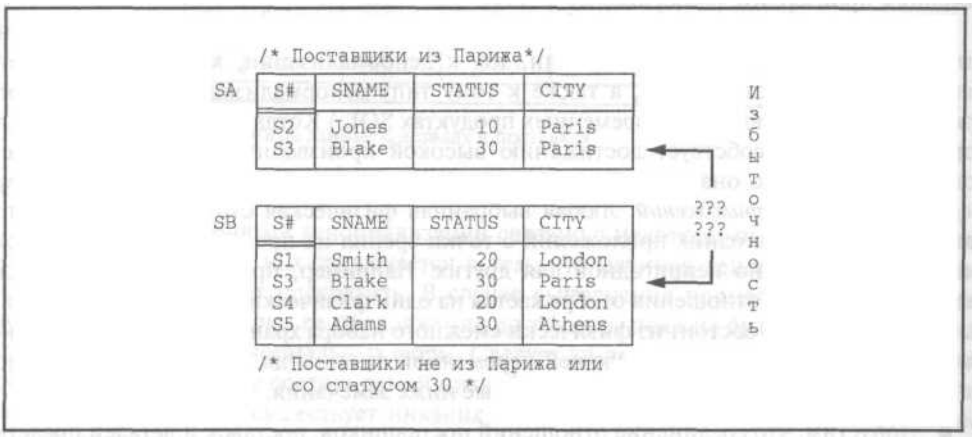


Рис. 13.7. Плохой, но вполне допустимый проект представления данных о поставщиках

Отметим, что данный кортеж с данными о поставщике S3 *должен* находиться в обеих переменных отношения. Допустим обратное, т.е. что он находится в переменной отношения SB, но отсутствует в переменной отношения SA. Применив допущение о замкнутости мира к переменной отношения SA, можно сделать вывод, что поставщик с номером S3 не находится в Париже. Но данные в переменной отношения SB свидетельствуют об обратном, т.е. о том, что он *находится* в Париже. Иначе говоря, мы получили противоречие, и, следовательно, база данных находится в противоречивом состоянии.

Недостаток показанного на рис. 13.7 проекта представления данных очевиден: один и тот же кортеж может дублироваться в каждой из двух переменных отношения. Иначе говоря, две переменные отношения имеют *перекрывающееся смысловое значение* и это приводит к тому, что один и тот же кортеж может удовлетворять предикатам обеих переменных отношения. Поэтому достаточно очевидным является правило, приведенное ниже.

- **Принцип ортогонального проектирования** *(исходная версия).* Никакие две переменные отношения в базе данных не должны иметь перекрывающихся смысловых значений.

Из этого следуют приведенные ниже выводы.

1. Как было отмечено в главе 10, с точки зрения пользователя *все* переменные отношения являются базовыми (за исключением представлений, которые определяются просто как сокращенные формы). Иначе говоря, этот принцип применим для проектирования всех "представимых", а не только "реальных" баз данных. Здесь нам вновь приходится иметь дело с *принципом относительности баз данных.* (Безусловно, аналогичные замечания применимы и к принципам нормализации.)

2. Обратите внимание на то, что две переменные отношения могут иметь перекрывающееся смысловое значение только в том случае, если они имеют одинаковые типы (т.е. одинаковые заголовки).
3. Принцип ортогонального проектирования означает, что вставка кортежа рассматривается как операция вставки кортежа *в базу данных,* а не в какую-то конкретную переменную отношения, поскольку существует не больше одной переменной отношения, предикату которой удовлетворяет этот кортеж.

Приведем несколько дополнительных замечаний по поводу последнего пункта. Безусловно, что в настоящее время при вставке кортежа обычно требуется указывать имя той переменной отношения R, в которую вставляется данный кортеж. Но это отнюдь не противоречит приведенным выше доводам, поскольку, по сути, имя R является всего лишь *сокращенным обозначением для соответствующего предиката* (допустим, PR). Действительно, команда вставки имеет такой смысл: INSERT *кортеж* t, *где* t *должен удовлетворять предикату* PR. Более того, переменная отношения R может быть представлением, определенным, например, с помощью выражения типа A UNION в. Как было сказано в главе 10, весьма желательно, чтобы системе было известно, куда вставлять новый кортеж— только в переменную отношения А, только в переменную отношения в или одновременно в обе эти переменные отношения.

На самом деле замечания, аналогичные приведенным выше, относятся ко всем типам операций, а не только к операциям INSERT. Во всех этих случаях указываемые имена переменных отношения в действительности являются лишь сокращенными обозначениями для их предикатов. *Это замечание вряд ли можно выразить более строго, просто сказав, что семантика данных представлена именно предикатами переменных отношения, а не их именами.*

Прежде чем завершить обсуждение принципа ортогонального проектирования, необходимо сделать одну важную поправку. На рис. 13.8 показан еще один пример явно плохого, но вполне допустимого проекта отношений с данными о поставщиках. В этом случае две переменные отношения сами по себе не имеют перекрывающегося смыслового значения, но их проекции по атрибутам {S#, SNAME}, безусловно, имеют такое значение. Вследствие этого попытка вставки некоторого кортежа, например, (S6, Lopez), в представление, определенное как объединение этих двух проекций, приведет к вставке кортежа (S6, Lopez, t) в переменную отношения SX и кортежа (S6, Lopez, c) — в переменную отношения SY (где t и c — соответствующие используемые по умолчанию значения). Ясно, что для устранения подобных проблем принцип ортогонального проектирования необходимо несколько расширить.

SX

S#	SNAME	STATUS
S1	Smith	20
S2	Jones	10
S3	Blake	30
S4	Clark	20
S5	Adams	30

SY

S#	SNAME	CITY
S1	Smith	London
S2	Jones	Paris
S3	Blake	Paris
S4	Clark	London
S5	Adams	Athens

Рис. 13.8. Еще один неудачный, но вполне допустимый вариант проекта отношений с данными о поставщиках

- **Принцип ортогонального проектирования** *(окончательная версия).* Пусть A и B являются двумя различными базовыми переменными отношения в некоторой базе данных. Тогда для переменных отношения A и в не должно существовать декомпозиций без потерь, соответственно, на такие проекции A1, A2, ..., Am и Bl, B2, ..., Bт, что некоторая проекция Ai в множестве проекций A1, A2, ..., Am и некоторая проекция Bj в множестве проекций Bl, B2, ..., Bт будут обладать перекрывающимися смысловыми значениями.

Из этого следуют приведенные ниже выводы.

1. Здесь термин *декомпозиция без потерь* означает именно то, что он означает всегда, т.е. декомпозицию на множество таких проекций, которые обладают следующими свойствами:
   - исходная переменная отношения может быть восстановлена за счет обратной операции соединения проекций;
   - ни одна из проекций в процессе такого восстановления не бывает избыточной (строго говоря, это второе условие не является необходимым для того, чтобы декомпозиция осуществлялась без потерь, но обычно оно желательно, как по казано в главе 12).
2. Данная версия принципа ортогонального проектирования охватывает предыду щую версию, поскольку единственная декомпозиция без потерь, которая всегда существует для любой переменной отношения R, является идентичной проекцией R (т.е. проекцией R по всем ее атрибутам).

Дополнительные замечания

Ниже приведены дополнительные замечания, касающиеся принципа ортогонального проектирования.

1. Прежде всего, отметим, что термин *ортогональность* отражает тот факт, что дан ный принцип проектирования фактически требует наличия в базовых переменных отношения взаимно независимых смысловых значений. Безусловно, что этот принцип продиктован здравым смыслом, но сам уже становится формализован ным выражением здравого смысла (как и принципы нормализации).
2. Предположим, что обычно рассматриваемую переменную отношения S с данными о поставщиках для улучшения структуры информации решено разделить на не сколько фрагментов. Тогда, в соответствии с принципом ортогонального проекти рования, необходимо обеспечить, чтобы полученные фрагменты не пересекались, в том смысле, что каждый кортеж с данными о некотором поставщике присутствовал не больше чем в одном из фрагментов. (А также, безусловно должно соблюдаться такое требование, чтобы после объединения этих фрагментов восстанавливалась первоначальная переменная отношения.) Назовем такое разбиение **ортогональной декомпозицией.**
3. Общее назначение ортогонального проектирования заключается в сокращении избыточности и, следовательно, в предотвращении аномалий обновления (при этом преследуются такие же цели, как и при использовании принципа нормализации). По сути, ортогональное проектирование дополняет нормализацию в том смысле,

что, выражаясь неформально, нормализация сокращает избыточность данных *внутри* переменных отношения, тогда как ортогональное проектирование сокращает избыточность данных *между* переменными отношения.

4. Несмотря на то, что принципы ортогональности очевидны с точки зрения здравого смысла, они часто игнорируются на практике (причем иногда их даже рекомендуют игнорировать). Например, в финансовых базах данных весьма распространен приведенный ниже проект структуры данных.

```
ACTIVITIES 2001{ ENTRY#,DESCRIPTION,    AMOUNT, NEW BAL }
ACTIVITIES 2 002{ ENTRY#,DESCRIPTION,    AMOUNT, NEW BAL }
ACTIVITIES 2 003{ ENTRY*,DESCRIPTION,    AMOUNT, NEW BAL }
ACTIVITIES 2004{ ENTRY*,DESCRIPTION,    AMOUNT, NEW BAL }
ACTIVITIES_2005{ ENTRY*,DESCRIPTION,    AMOUNT, NEW_BAL }
```

По сути, внесение смыслового значения в имена переменных отношения или других объектов нарушает *информационный принцип,* который гласит, что вся информация в базе данных должна быть явно представлена в виде значений данных и никак иначе.

5. Если A и B являются базовыми переменными отношения одного типа, то приверженность принципам ортогонального проектирования будет равносильна соблюдению приведенных ниже требований.

   - A UNION B. Всегда является объединением непересекающихся отношений.
   - A INTERSECT B. Всегда является пустым отношением.
   - A MINUS B. Всегда равно A.

13.7. ДРУГИЕ НОРМАЛЬНЫЕ ФОРМЫ

В этом разделе мы снова возвращаемся к теме нормализации. Прежде чем завершить обсуждение вопросов нормализации, следует напомнить сделанное в главе 12 замечание о том, что, помимо уже описанных, существуют и другие нормальные формы. Дело в том, что *теория нормализации* и связанные с ней вопросы (в настоящее время эту область обычно называют **теорией зависимостей)** развились в широкую самостоятельную область знаний. Исследования в данной области продолжаются и в настоящее время, причем довольно успешно (хотя в последнее время наблюдается некоторое снижение интереса к этой теме). Но более подробный обзор этих исследований выходит за рамки данной главы. Заинтересованный читатель найдет достаточно полный обзор полученных в этой области результатов в [13.18] (по состоянию на середину 1980-х годов), а более свежие обзоры можно найти в [11.1] и [11.3]. Ниже кратко описаны лишь некоторые из них.

1. **Доменно-ключевая нормальная форма (ДКНФ).** Эта форма была предложена Фейгином [13.16]. В отличие от рассмотренных выше нормальных форм, она не определяется в терминах функциональных зависимостей, многозначных зависимостей или зависимостей соединения. Вместо этого утверждается, что переменная отношения R находится в ДКНФ тогда и только тогда, когда каждое наложенное на нее ограничение является логическим следствием *ограничений доменов* и *ограничений ключей,* наложенных на данную переменную отношенияи.

- **Ограничение домена** в том смысле, в котором оно здесь употребляется, — это ограничение, предписывающее использование для определенного атрибута значений только из некоторого заданного домена. (В главе 9 это ограничение упоминается как ограничение *атрибута,* а не как ограничение типа, даже не смотря на то, что домены представляют собой типы.)
- **Ограничение ключа** — это ограничение, утверждающее, что некоторый атрибут или комбинация атрибутов представляет собой потенциальный ключ.

Концептуально реализация ограничений, которые установлены для переменной отношения, находящейся в ДКНФ, осуществляется очень просто, поскольку для этого достаточно реализовать поддержку ограничений домена (атрибута) и ключа, а все остальные ограничения будут приведены в действие автоматически. Обратите внимание на то, что под выражением "все остальные ограничения" подразумевается нечто большее, чем просто функциональные и многозначные зависимости или зависимости соединения. Это выражение фактически обозначает весь *предикат* данной переменной отношения.

Фейгин в [13.16] показал, что любая переменная отношения, находящаяся в ДКНФ, обязательно находится в 5НФ (а значит, в 4НФ и т.д.), а также в форме типа (3,3)НФ (подробнее о ней рассказывается ниже в пункте 2). Но не всегда можно привести переменную отношения к ДКНФ или получить ответ на вопрос о том, когда *может* быть выполнено такое приведение.

2. **Нормальная форма типа "сокращение—объединение".** Вновь обратимся к переменной отношения s с данными о поставщиках. Согласно описанной выше теории нормализации, эта переменная отношения находится в "приемлемой" нормальной форме, и действительно, она находится в 5НФ, поэтому не характеризуется аномалиями и не нуждается в дальнейшем разбиении на проекции для устранения аномалий. Но зачем хранить сведения обо всех поставщиках в одной переменной отношения? Может, было бы лучше разместить данные о поставщиках из Лондона в одном отношении (например, в отношении LS), из Парижа — в другом (например, в PS) и т.д.? Иначе говоря, может быть стоило бы рассмотреть возможность декомпозиции на основе некоторого **сокращения,** а не на основе проекции? Лучше или хуже будет структура данных, полученная в результате такой декомпозиции? (Фактически она всегда получается хуже, как показано в упр. 8.8 в главе 8, но классическая теория нормализации не может дать ответа на поставленные выше вопросы.)

   Другим направлением в исследованиях нормализации является применение декомпозиции на основе операций, отличных от проекции. В рассматриваемом выше примере, как уже упоминалось, операцией декомпозиции является (непересекающееся) сокращение, а соответствующей операцией композиции — (непересекающееся) **объединение.** Таким образом, вполне возможно создать "сократительно-объединительную" теорию нормализации, аналогичную, но ортогональную (независимую) относительно обсуждавшейся выше проекционно-соединительной теории нормализации[5]. Автору настоящей книги ничего не известно о достаточно

[5] Действительно, Фейгин в [13.15] назвал 5НФ *проекционно-соединительной* нормальной формой, поскольку это была именно такая нормальная форма, определяемая с помощью операций проекции и соединения.

развитых теориях подобного типа, но некоторые исходные идеи можно найти I статье Смита [13.32], где дано определение новой нормальной формы под названием **(3,3)НФ.** Подразумевается, что переменная отношения в (3,3)НФ уже находится в НФБК, однако не обязательно находится в 4НФ, так же как переменная отношения в 4НФ не обязательно находится в (3,3)НФ. Таким образом, как *v* предполагалось выше, приведение к форме типа (3,3)НФ является независимым по отношению к приведению к 4НФ (и 5НФ). Более подробно об этом можнс прочесть в [13.15] и [13.23]. К этой теме относится также принцип ортогонального проектирования [13.12] (поэтому ортогональные проекты можно в конечном итоге рассматривать как один из вариантов осуществления нормализации).

3. **Шестая нормальная форма.** Если речь идет о классических операциях проекции и соединения, то пятая нормальная форма является последней нормальной формой. Но, как будет показано в главе 23, возможно и желательно определить обобщенные версии этих операций, а следовательно, обобщенную форму зависимости соединения, поэтому и новую (шестую) нормальную форму, 6НФ. Следует отметить, что для нее целесообразно использовать название "шестая нормальная форма", поскольку 6НФ (в отличие от нормальных форм, описанных в пп. 1 и 2) фактически представляет собой еще один этап на пути от 1НФ к 2НФ, затем к следующей нормальной форме и т.д., вплоть до 5NF. Более того, все переменные отношения, находящиеся в 6НФ, обязательно находятся и в 5НФ. Дополнительное описание приведено в главе 23.

13.8. РЕЗЮМЕ

В этой главе завершается обсуждение **дальнейшей нормализации** (начатое в главе 12), включая рассмотрение **многозначных зависимостей,** являющихся обобщением понятия функциональных зависимостей, а также **зависимостей соединения,** являющихся обобщениями многозначных зависимостей. Ниже даны значительно упрощенные определения этих понятий.

- Переменная отношения R{A, в, C} удовлетворяет многозначной зависимости A →→ B | C тогда и только тогда, когда множество значений атрибута в, соот ветствующее заданной паре значений атрибутов AC, зависит только от значения атрибута A, и аналогично этому, множество значений атрибута с, соответствующее заданной паре значений атрибутов AB, зависит только от значения атрибута A. Та кая переменная отношения может быть подвергнута декомпозиции без потерь на проекции по атрибутам {A, B} и {A, C}, причем многозначные зависимости яв ляются необходимым и достаточным условием допустимости такой декомпозиции (теорема Фейгина).
- Некоторая переменная отношения удовлетворяет зависимости соединения *{A, B, ..., Z} тогда и только тогда, когда она равна соединению своих проекций по подмножествам A, B, ..., Z множества атрибутов. Очевидно, что такая пере менная отношения может быть подвергнута декомпозиции без потерь на указан ные проекции.

Переменная отношения находится в 4НФ в том случае, если все существующие в ней многозначные зависимости одновременно являются функциональными зависимостями

от ее суперключей. Переменная отношения находится в 5НФ (называемой также **проекционно-соединительной** нормальной формой — ПСНФ) тогда и только тогда, когда все существующие в ней зависимости соединения одновременно являются функциональными зависимостями от ее суперключей (т.е. если зависимость соединения представляет собой *{A, B, ..., Z}, то каждое из подмножеств A, B, ..., Z множества атрибутов является суперключом). Пятая нормальная форма (которая всегда достижима) является *окончательной нормальной формой* по отношению к операциям проекции и соединения.

В этой главе была также кратко описана общая схема **процедуры нормализации,** представленная в виде некоторой неформально описанной последовательности этапов с необходимыми комментариями (но следует напомнить читателю, что проектирование базы данных обычно не осуществляется с помощью этой процедуры). Затем было дано краткое описание **принципа ортогонального проектирования,** который неформально можно сформулировать следующим образом: никакие две переменные отношения не должны иметь проекций с перекрывающимися смысловыми значениями. Наконец, здесь в этой главе было дано краткое описание *некоторых дополнительных нормальных форм.*

В заключение следует отметить, что дальнейшие исследования в данной области являются чрезвычайно перспективными. Дело в том, что *теория дальнейшей нормализации,* которую теперь все чаще называют **теорией зависимостей,** представляет собой один из весьма субъективных разделов теории проектирования баз данных. Это означает, что она, к сожалению, ближе к искусству, чем к строгой методике. Для выработки последней необходимо найти более твердые принципы и разработать соответствующие рекомендации. Поэтому любой успех в дальнейшем развитии данной теоретической области представляет значительный интерес для исследователей.

УПРАЖНЕНИЯ

13.1. Рассматриваемые в этой главе переменные отношения cтх и SPJ (примеры их данных показаны на рис. 13.2 и 13.4) удовлетворяют некоторой многозначной зависимости и некоторой зависимости соединения, соответственно, которые не определяются потенциальными ключами данных переменных отношения. Выразите эту многозначную зависимость и эту зависимость соединения в виде ограничений целостности с помощью синтаксиса языка Tutorial D, применявшегося в главе 9. Приведите версии, основанные на использовании и реляционного исчисления, и реляционной алгебры.

13.2. Пусть C — это некоторое множество, а переменная отношения R{A, B} такова, что кортеж (a, b) содержится в R тогда и только тогда, когда a и b принадлежат множеству C. Какие функциональные и многозначные зависимости, а также зависимости соединения имеют место в переменной отношения R? В какой нормальной форме находится эта переменная отношения?

13.3. Пусть в некоторой базе данных содержится информация о торговых агентах, регионах сбыта и самих товарах. Каждый агент отвечает за сбыт в одном или нескольких регионах, а в каждом регионе имеется один или несколько торговых агентов. Аналогичным образом, каждый агент отвечает за сбыт одного или больше видов товаров, а за каждым товаром закреплен один или несколько торговых агентов. Каждый вид товара продается в каждом регионе, однако два торговых агента не могут продавать один и тот же товар в одном и том же регионе. Каждый торговый

агент продает один и тот же набор товаров в каждом регионе, за который он отвечает. Спроектируйте набор переменных отношения, отвечающий указанным требованиям к данным.

13.4. В разделе 12.5 главы 12 был приведен алгоритм декомпозиции без потерь для произвольной переменной отношения R с ее разбиением на множество переменных отношения в НФБК. Измените этот алгоритм таким образом, чтобы выполнялось аналогичное преобразование, но уже в **4НФ.**

13.5. *(Модифицированная версия упр. 13.3.)* Пусть в некоторой базе данных содержится информация о торговых агентах, регионах сбыта и самих товарах. Каждый торговый агент отвечает за сбыт товаров в одном или нескольких регионах, и в каждом регионе имеется один или несколько торговых агентов. Аналогичным образом, каждый торговый агент отвечает за сбыт одного или нескольких видов товаров, а каждый вид товара распространяется одним или несколькими торговыми агентами. Наконец, каждый вид товара продается в одном или нескольких регионах и в каждом регионе продается один или несколько видов товаров. Более того, если торговый агент R отвечает за сбыт в регионе A, товар P продается в регионе A и торговый агент R отвечает за сбыт товара p, то агент R продает товар P в регионе A. Спроектируйте набор переменных отношения, отвечающий указанным требованиям к данным.

13.6. Предположим, что данные о поставщиках представлены с помощью следующих двух переменных отношения sx и SY (как показано на рис. 13.8 в разделе 13.6).

```
SX { S#, SNAME, STATUS
} SY { S#, SNAME,
CITY }
```

Соответствует ли этот проект рекомендациям по нормализации, приведенным в данной и предшествующей главах? Обоснуйте ваш ответ.

СПИСОК ЛИТЕРАТУРЫ

13.1. Aho A.V., Beeri C, Ullman J. D. The Theory of Joins in Relational Databases // ACM TODS. — September 1979. — 4, № 3.

В этой статье было впервые указано, что могут существовать переменные отношения, которые не равны соединению любых двух своих проекций, но равны соединению трех или более проекций. Основное назначение статьи — представить алгоритм, который теперь называется алгоритмом **преследования** (chase), предназначенный для определения, является ли данная зависимость соединения логическим следствием данного набора функциональных зависимостей (а в [13.18] приведен пример **проблемы определения такого следствия).** Эта проблема эквивалентна проблеме определения того, будет ли данная декомпозиция выполняться без потерь для заданного набора функциональных зависимостей. В статье также обсуждается вопрос о расширении алгоритма для случая, когда заданные зависимости являются не функциональными, а многозначными.

13.2. Beeri C, Fagin R., Howard J.H. A Complete Axiomatization for Functional and Multivalued Dependencies // Proc. 1977 ACM SIGMOD Intern. Conf. on Management of Data. — Toronto, Canada. — August 1977.

В этой работе результаты работы Армстронга (Armstrong) [11.2] обобщаются и распространяются на многозначные и функциональные зависимости. В частности, приведен строгий и полный набор правил вывода для многозначных зависимостей.

1. **Дополнение.** Если множества атрибутов A, в, **с** совместно содержат все атрибуты переменной отношения и A является надмножеством пересечения в п C, то A →→ B тогда и только тогда, когда A →→ C.
2. **Рефлексивность.** Если **в** является подмножеством A, то A →→ B.
3. **Приращение.** Если A →→ в и с является подмножеством D, то AD →→BC.
4. **Транзитивность.** Если A →→ B и B →→ C, то A →→ C - B.

   Далее следуют дополнительные полезные правила вывода, которые могут быть получены на основании приведенных выше четырех правил.

5. **Псевдотранзитивность.** Если A →→ B и BC →→ D, то AC→→ D - BC.
6. **Объединение.** Если A →→ B и A →→ C, то A →→ BC.
7. **Декомпозиция.** Если A →→ BC, то A →→B ∩ C, A →→ B - СИА →→ C - B.

   Затем в этой статье рассматриваются два правила, касающиеся совместного применения многозначных и функциональных зависимостей.

8. **Репликация.** Если A →→B, то A →→ B.
9. **Слияние.** Если A →→ B, C → D, D является подмножеством в, а пересечение B ∩ C ПУСТО, ТО A → D.

   Правила Армстронга [11.2] вместе с правилами 1—4, 8 и 9 образуют строгий и полный набор правил вывода для функциональных и многозначных зависимостей. В статье дано также еще одно полезное правило, связывающее функциональные и многозначные зависимости.

10. Если A →→ B и AB → C,то A → C - B.

13.3. Brosda V., Vossen G. Update and Retrieval Through a Universal Schema Interface // ACM TODS. - December 1988. - 13, № 4.

В предыдущих попытках создания интерфейса *универсального отношения* (см. аннотацию к [13.20]) рассматривались только операции выборки данных. В этой статье предлагается также подход на основе операций обновления.

13.4. Carlson C.R., Kaplan R.S. A Generalized Access Path Model and Its Application to a Relational Data Base System // Proc. 1976 ACM SIGMOD Intern. Conf. on Management of Data. —Washington, D.C. —June 1976.

См. аннотацию к [13.20].

13.5. Date C.J. Will the Real Fourth Normal Form Please Stand Up? // C J. Date and Hugh Darwen. Relational Database Writings 1989—1991.— Reading, Mass.: Addison-Wesley, 1992.

В резюме к этой работе отмечается, что "существует несколько различных понятий в области проектирования баз данных, которые разные авторы называют одинаково — *четвертая нормальная форма* (4НФ). Назначение данной работы — прояснить

смысл этого понятия". Здесь, вероятно, следует добавить, что единственно правильное определение 4НФ приводится в настоящей главе... Не верьте никаким другим!

13.6. Date C.J. The Normal Is So... Interesting (в двух частях) // DBP&D. — November-December 1997. - 10, № 11-12.

Обсуждение нормализации в разделе 13.5 взято из этой работы. Кроме того, следует отметить некоторые дополнительные особенности.

- Даже в такой базе данных, которая используется только для чтения, необходи мо задавать ограничения целостности, поскольку они определяют смысл дан ных, а (как отмечается в разделе 13.4) *отказ* от денормализации предоставляет простой способ определения некоторых важных ограничений. Если же база данных используется *не только* для чтения, то отказ от денормализации ее дан ных предоставляет также простой способ *реализации* этих ограничений.
- Денормализация предполагает наличие повышенной избыточности данных, но (что противоречит широко распространенному ошибочному мнению) повы шенная избыточность данных не обязательно предполагает использование процедуры денормализации! По этому поводу многие авторы заблуждались и продолжают заблуждаться до сих пор.
- В общем случае следует придерживаться такого правила: денормализацию (на логическом уровне) следует использовать лишь в качестве последнего средства ("только в той ситуации, когда все другие методы себя исчерпали", как указано в [4.17]).

13.7. Date C.J. The Final Normal Form! (в двух частях) // DBP&D. — January/February 1998.-11, №1-2.

Учебное пособие по зависимостям соединения и 5НФ. Заголовок этой статьи уже небесспорен (см. главу 23).

13.8. Date C.J. What's Normal, Anyway? // DBP&D. — March 1998. — 11, № 3.

Обзор некоторых "патологических" примеров нормализации.

13.9. Date C.J. Normalization Is No Panacea // DBP&D. — April 1998. — 11, № 4.

Обзор некоторых проблем проектирования базы данных, когда применение теории нормализации *не дает* результата. Данную статью не следует рассматривать как критику этой теории.

13.10. Date C.J. Principles of Normalization // http: //www.BRCommunity.com (Februa ry 2003); http: / /www. dbdebunk. com (March 2003).

Краткий учебник по нормализации. Для справок ниже кратко описаны рассматриваемые в нем принципы.

1. Для любой переменной отношения, не находящейся в 5НФ, должна быть вы полнена декомпозиция на проекции, находящиеся в 5NF.
2. Должна обеспечиваться возможность восстановить первоначальную перемен ную отношения путем повторного соединения этих проекций.
3. В процессе декомпозиции должны сохраняться зависимости.

4. В процессе восстановления должна быть необходима каждая проекция.
5. *(Не такой безусловный принцип, как первые четыре.)* Прекращать нормализацию после того, как все переменные отношения будут находиться в 5НФ.

13.11. Date C.J., Fagin R. Simple Conditions for Guaranteeing Higher Normal Forms in Relational Databases // C. J. Date and Hugh Darwen. Relational Database Writings 1989—1991. — Reading, Mass.: Addison-Wesley, 1992. (Работа также опубликована в ACM TODS. - September 1992. - 17, № 3.)

В этой работе показано, что если переменная отношения R находится в 3НФ и все потенциальные ключи переменной отношения R простые (т.е. состоят из одного атрибута), то переменная отношения R уже находится в 5НФ. Иначе говоря, в таком случае не стоит беспокоиться о разных относительно сложных вопросах, связанных с многозначными зависимостями, зависимостями соединения, 4НФ и 5НФ, которые обсуждались в данной главе.

Примечание. В статье доказан также другой результат, а именно: если переменная отношения R находится в НФБК и хотя бы один из ее потенциальных ключей является простым, то переменная отношения R уже находится в 4НФ, но не обязательно в 5НФ.

13.12. Date C. J., McGovern D. A New Database Design Principle // C. J. Date. Relational Database Writings 1991-1994. — Reading, Mass.: Addison-Wesley, 1995.

13.13. Delobel C, Parker D.S. Functional and Multivalued Dependencies in a Relational Database and the Theory of Boolean Switching Functions // Tech. Report No. 142. — Dept. Maths. Appl. et Informatique, Univ. de Grenoble, France. — November 1978.

В этой работе описанные в [11.5] результаты распространяются на многозначные зависимости аналогично функциональным зависимостям.

13.14. Fagin R. Multivalued Dependencies and a New Normal Form for Relational Databases //ACM TODS. - September 1977. - 2, № 3.

В этой статье впервые определены такие понятия, как 4НФ и многозначная зависимость.

Примечание. В ней также рассматривается тема, касающаяся **внедренных** многозначных зависимостей. Допустим, что переменная отношения ctx, рассмотренная в этой главе, расширена дополнительным атрибутом DAYS, представляющим количество дней, затраченных на преподавание предмета по учебнику TEXT некоторым преподавателем TEACHER, ведущим некоторый курс обучения COURSE. Назовем такую расширенную переменную отношения именем CTXD и рассмотрим приведенный ниже пример ее данных.

CTXD

COURSE	TEACHER	TEXT	DAYS
Physics	Prof. Green	Basic Mechanics	5
Physics	Prof. Green	Principles of Optics	5
Physics	Prof. Brown	Basic Mechanics	6
Physics	Prof. Brown	Principles of Optics	4
Math	Prof. Green	Basic Mechanics	3
Math	Prof. Green	Vector Analysis	3
Math	Prof. Green	Trigonometry	4

В таком случае комбинация атрибутов {COURSE,TEACHER,TEXT} является потенциальным ключом этой переменной отношения, в которой имеет место следующая функциональная зависимость.

```
{ COURSE, TEACHER, TEXT } → DAYS
```

Можно заметить, что данная переменная отношения находится в 4НФ, поскольку не содержит никаких многозначных зависимостей, которые не являются одновременно и функциональными зависимостями. Однако она содержит две *внедренные* многозначные зависимости (атрибута TEACHER от атрибута COURSE и атрибута TEXT от атрибута COURSE). В переменной отношения R существует внедренная многозначная зависимость атрибута в от атрибута А, если в некоторой проекции переменной отношения R выполняется "обычная" многозначная зависимость AB. Обычная многозначная зависимость является частным случаем внедренной многозначной зависимости, но не все внедренные многозначные зависимости являются обычными многозначными зависимостями.

Как иллюстрируется в данном примере, внедренные многозначные зависимости так же указывают на наличие избыточности, как и обычные МЗЗ, однако эта избыточность (вообще говоря) не может быть устранена с помощью разбиения на проекции. Представленную выше переменную отношения CTXD нельзя подвергнуть декомпозиции без потерь (она фактически находится не только в 4НФ, но и в 5НФ), поскольку атрибут DAYS зависит от всех трех атрибутов, COURSE, TEACHER и TEXT, и потому не может присутствовать в отношении без какого-либо из указанных атрибутов. Значит, две внедренные многозначные зависимости следует рассматривать как дополнительные явно заданные ограничения для данной переменной отношения. Задачу уточнения деталей оставляем читателю в качестве упражнения.

13.15. Fagin R. Normal Forms and Relational Database Operators // Proc. 1979 ACM SIGMOD Intern. Conf. on Management of Data. — Boston, Mass. — May—June 1979.
В статье представлена концепция проекционно-соединительной нормальной формы (ПСНФ), или 5НФ. Однако содержимое статьи намного выходит за эти рамки. Она может рассматриваться как окончательное определение того, что можно назвать "классической" теорией нормализации, т.е. теорией декомпозиции без потерь, основанной на проекции как операторе декомпозиции и на естественном соединении как соответствующем операторе (ре)композиции.

13.16. Fagin R. A Normal Form for Relational Databases That Is Based on Domains and Keys // ACM TODS. - September 1981. - 6, № 3.

13.17. Fagin R. Acyclic Database Schemes (of Various Degrees): A Painless Introduction // IBM Research Report RJ3800. — April 1983. (Переиздано: Proc. CAAP83 8th Colloquium on Trees in Algebra and Programming: Springer-Verlag Lecture Notes in Computer Science No. 159 (eds. G. Ausiello and M. Protasi). — New York, N.Y.: Springer-Verlag, 1983.)

В разделе 13.3 этой главы было показано, как некоторая тройная переменная отношения SPJ, удовлетворяющая определенному циклическому ограничению, может быть подвергнута декомпозиции без потерь с образованием трех бинарных переменных отношения. Полученная в результате структура базы данных (в этой работе

она называется *схемой)* является циклической, поскольку каждая из трех переменных отношения имеет атрибут, общий с каждой из остальных двух переменных отношения. (Если структура описана в виде *гиперграфа,* в котором ребра представляют отдельные переменные отношения, а узел, соединяющий два ребра, представляет общие для этих двух переменных отношения атрибуты, то становится понятно, почему используется термин "циклическая структура".) Многие реальные структуры, наоборот, являются ациклическими. Они обладают большим количеством формальных свойств, которые не применимы для циклических структур. В этой статье Фейгин представляет и разъясняет некоторые из таких свойств. Ацикличность следует рассматривать с такой точки зрения: теория нормализации позволяет определить целесообразность реструктуризации *отдельной* переменной отношения, а теория ацикличности позволяет определить целесообразность реструктуризации *набора* переменных отношения.

13.18. Fagin R., Vardi M.Y. The Theory of Data Dependencies — A Survey// IBM Research Report RJ4321. — June 1984. (Переиздано: Mathematics of Information Processing // Proc. Symposia in Applied Mathematics 34, American Mathematical Society, 1986.)

В этой работе излагается краткая история развития теории зависимостей по состоянию на середину 1980-х годов (следует обратить внимание, что слово "зависимость" в данном случае относится *не только* к функциональным зависимостям). В частности, в статье подытожены основные достижения в трех отдельных областях этой теории с перечнями рекомендуемой литературы по каждой из тем. Эти три темы включают, во-первых, проблему логического вывода, во-вторых, универсальную реляционную модель и, в-третьих, ациклические схемы. Проблема логического вывода состоит в выяснении для заданного множества зависимостей D и некоторой отдельной зависимости d, будет ли зависимость «алогическим следствием множества зависимостей D. **Универсальная реляционная модель** и **ациклические схемы,** соответственно, кратко обсуждаются в комментариях к [13.20] и [13.17].

13.19. Fagin R., Mendelzon A.O., Ullman J.D. A Simplified Universal Relation Assumption and Its Properties // ACM TODS. — September 1982. — 7, № 3.

Здесь рассматривается предположение о том, что реальный мир всегда может быть представлен с помощью "универсального отношения" [13.20] (скорее, универсальной переменной отношения), которое удовлетворяет одной и только одной зависимости соединения вместе с некоторым множеством функциональных зависимостей, а также обсуждаются следствия такого предположения.

13.20. Kent W. Consequences of Assuming a Universal Relation // ACM TODS. — December 1981.-6, №4.

Здесь показано несколько способов определения понятия **универсальной переменной отношения.** Прежде всего, согласно процедуре нормализации, описанной в последних двух главах, предполагается возможность задания исходного универсального *отношения* (а точнее, универсальной переменной отношения), которое включает все атрибуты рассматриваемой базы данных. Затем демонстрируется, как такая переменная отношения может последовательно заменяться "меньшими" проекциями (т.е. отношениями меньшей степени) вплоть до достижения некоторой "приемлемой" структуры. Но является ли исходное предположение реальным

или справедливым? В работе [13.20] утверждается, что это не так с практической и теоретической точек зрения. В ответ на данную статью появилась публикация [13.33], а в ответ на последнюю — [13.21].

Еще одним проявлением концепции универсальной переменной отношения, более важным с практической точки зрения, является ее применение в качестве *интерфейса пользователя.* Основная идея здесь совершенно ясна и действительно весьма притягательна (на уровне интуитивного понимания): пользователям следует составлять запросы не в терминах переменных отношения и их соединений, а на основе одних только атрибутов, например так, как показано в приведенном ниже выражении, где требуется "получить статус всех поставщиков деталей красного цвета".

```
STATUS WHERE COLOR = 'Red'
```

В этой связи данная идея имеет две более или менее различающиеся интерпретации.

1. Одна возможность заключается в том, что система должна каким-то образом самостоятельно определить, по какому логическому пути нужно следовать для выполнения запроса (в частности, какие соединения ей необходимо создавать). Таков подход, предлагаемый в [13.4] (эта статья, по всей видимости, является первой статьей, посвященной возможности создания универсального интер фейса пользователя, хотя сам термин "универсальная переменная отношения" в ней не упоминается). Этот подход в значительной степени зависит от пра вильного именования атрибутов. Таким образом, например, для двух атрибутов номера поставщика (в переменных отношения, соответственно, S и SP) долж ны быть заданы *одинаковые* имена; и наоборот, для атрибутов города, в котором находится поставщик, и города, в котором хранятся детали (в переменных от ношения, соответственно, S и P), должны быть заданы *разные* имена. Если ка кое-либо из этих правил будет нарушено, то некоторые запросы могут быть выполнены не совсем корректно.
2. В другой, менее амбициозной, интерпретации все запросы рассматриваются в терминах *предопределенного* набора соединений — фактически предопределен ного представления, состоящего из соединения всех переменных отношения в рассматриваемой базе данных.

Хотя нет сомнений в том, что любой из представленных здесь подходов позволяет значительно упростить выражения при составлении многих практически применяемых запросов (причем тот или иной вариант данного подхода будет весьма уместен в клиентских приложениях, поддерживающих естественный язык общения), ясно, что в общем случае система должна поддерживать и средства явного указания логического пути доступа. Для подтверждения данного тезиса рассмотрим следующий запрос.

```
STATUS WHERE COLOR ≠ COLOR ('Red')
```

Что означает этот запрос: "Получить статус поставщиков деталей, имеющих цвет, отличный от красного" или "Получить статус всех поставщиков, которые не поставляют детали красного цвета"? Какой бы смысл из двух предложенных не был выбран системой, должен также существовать еще один способ формулировки

выбранного варианта запроса. (Можно сразу же указать альтернативную интерпретацию первого запроса, который рассматривался в данной аннотации: "Получить статус поставщиков *только* красных деталей".) В качестве третьего примера можно привести запрос "Получить имена поставщиков, находящихся в одном и том же городе", для которого, очевидно, придется явным образом задать соединение (поскольку в нем, неформально выражаясь, предусматривается соединение переменной отношения S с самой собой).

13.21. Kent W. The Universal Relation Revisited //ACM TODS. - December 1983. - 8, № 4.

13.22. Korth H.F. et al. System/U: A Database System Based on the Universal Relation Assumption//ACM TODS. — September 1984. — 9, № 3.

Здесь приводится описание теории, языка определения данных DDL, языка управления данными DML, а также практического воплощения экспериментальной системы на основе "универсального отношения", созданной в Станфордском университете.

13.23. Maier D., Ullman J.D. Fragments of Relations // Proc. 1983 SIGMOD Intern. Conf. on Management of Data. — San Jose, Calif. — May 1983.

13.24. Maier D., Ullman J.D., Vardi M.Y. On the Foundations of the Universal Relation Model // ACM TODS. — June 1984. — 9, № 2. (Более ранняя версия этой статьи под заголов ком "The Revenge of the JD" опубликована в Proc. 2nd ACM SIGFACT-SIGMOD Symposium on Principles of Database Systems. — Atlanta, Ga. — March 1983.)

Maier D., Ullman J.D. Maximal Objects and the Semantics of Universal Relation Databases //ACM TODS. — March 1983. — 8, № 1.

Максимальные объекты представляют собой подход к разрешению проблемы неоднозначности, возникающей в "универсальных реляционных" системах, базовая структура которых не является ациклической (подробности приводятся в [13.17]). Максимальный объект соответствует предварительно объявленному подмножеству атрибутов универсальной переменной отношения, для которого базовая структура будет ациклической. Такие объекты используются для интерпретации запросов, которые в противном случае были бы неоднозначными.

13.26. Nicolas J.M. Mutual Dependencies and Some Results on Undecomposable Relations // Proc. 4th Intern. Conf. on Very Large Data Bases. — Berlin, Federal German Republic. — September 1978.

В работе вводится концепция *взаимной зависимости,* которая в действительности представляет собой частный случай зависимости соединения, не являющейся многозначной или функциональной зависимостью, и включает точно три проекции (как в примере с "ограничением 3-Д", упомянутым в разделе 13.3). Однако эта концепция не имеет ничего общего с понятием взаимной зависимости, описанным в главе 12.

13.27. Osborn S.L. Towards a Universal Relation Interface // Proc. 5th Intern. Conf. on Very Large Data Bases. — Rio de Janeiro, Brazil. — October 1979.

В этой работе предполагается, что если две или больше последовательностей соединений в системе на основе "универсального отношения" генерируют потенциальный ответ на заданный запрос, то желательным откликом будет соединение всех таких потенциальных ответов. Приводятся алгоритмы для генерации всех таких последовательностей соединений.

13.28. Parker D.S., Delobel C. Algorithmic Applications for a New Result on Multivalued Dependencies // Proc. 5th Intern. Conf. on Very Large Data Bases. — Rio de Janeiro, Brazil. — October 1979.

Здесь описано применение результатов работы [13.13] для решения различных проблем, например для тестирования операции декомпозиции без потерь.

13.29. Sagiv Y., Delobel C, Parker D.S., Fagin R. An Equivalence between Relational Database Dependencies and a Subclass of Propositional Logic //JACM. — June 1981. — 28, № 3.

Это — сводка работ [11.8] и [13.30].

13.30. Sagiv Y., Fagin R. An Equivalence between Relational Database Dependencies and a Subclass of Propositional Logic // IBM Research Report RJ2500. — March 1979.

Здесь результаты работы [11.8], полученные для функциональных зависимостей, расширены с учетом многозначных зависимостей.

13.31. Sciore E. A Complete Axiomatization of Full Join Dependencies // JACM. — April 1982.-29, №2.

Здесь результаты работы [13.2], полученные для зависимостей соединения, расширены с учетом функциональных и многозначных зависимостей.

13.32. Smith J.M. A Normal Form for Abstract Syntax // Proc. 4th Intern. Conf. on Very Large Data Bases. — Berlin, Federal German Republic. — September 1978.

13.33. Ullman J.D. On Kent's Consequences of Assuming a Universal Relation // ACM TODS. - December 1983. - 8, № 4.

13.34. Ullman J.D. The U.R. Strikes Back // Proc. 1st ACM SIGFACT-SIGMOD Symposium on Principles of Database Systems. — Los Angeles, Calif. — March 1982.

ГЛАВА 14

Семантическое моделирование

14.1. ВВЕДЕНИЕ

Семантическое моделирование стало предметом интенсивных исследований с конца 1970-х годов. Основным побудительным мотивом подобных исследований (т.е. проблемой, которую пытались разрешить исследователи) был следующий факт. Дело в том, что системы баз данных обычно обладают весьма ограниченными сведениями о *смысле* хранящихся в них данных. Чаще всего они позволяют лишь манипулировать данными определенных простых типов и определяют некоторые простейшие ограничения целостности, наложенные на эти данные. Любая более сложная интерпретация возлагается на пользователя. Однако было бы замечательно, если бы системы могли обладать немного более широким объемом сведений[1] и несколько интеллектуальнее отвечать на запросы пользователя, а также поддерживать более сложные (т.е. более высокоуровневые) интерфейсы пользователя. Например, было бы замечательно, если бы система могла определять, что

---

[1] Следует отметить, что с нашей точки зрения система с поддержкой **предикатов** (обсуждавшихся в главе 9) *должна была бы* "понимать немного больше". Иначе говоря, можно было бы возразить, что описанная поддержка предикатов может восприниматься как удобное и эффективное основание для семантического моделирования. Однако, как это ни печально, большинство схем семантического моделирования не имеет никакого строгого обоснования, поскольку все они в той или иной степени являются произвольными (предложения в **[14.22]—[14.24]** остаются редкими исключениями). Возможно, такое состояние дел в будущем изменится, особенно благодаря растущему пониманию в мире коммерции важности **бизнес-правил** [9.21], [9.22]. Описанные в главе 9 внешние предикаты по сути как раз и являются такими "бизнес-правилами" [14.14].

вес детали и поставляемое количество являются хотя и числовыми, но все же *семантически* разными величинами. Это позволило бы системе поставить под сомнение или даже вовсе отвергнуть запрос, в котором соединение информации о деталях и поставках выполняется путем сравнения веса детали с поставляемым количеством.

Безусловно, к приведенному примеру прямое отношение имеет понятие типов (или *доменов).* Это служит прекрасной иллюстрацией того факта, что существующие модели данных все же не лишены семантических аспектов. В частности, домены, потенциальные и внешние ключи являются примерами семантических аспектов существующей реляционной модели, что непосредственно следует из ее определения. В качестве другого способа реализации семантики были разработаны различные "расширенные" модели данных, которые, тем не менее, несут лишь ненамного большую смысловую нагрузку, чем модели, предложенные ранее. Перефразировав Кодда [14.7], можно сказать, что задача представления смысла данных не имеет окончательного решения и можно ожидать непрекращающегося прогресса в этом направлении по мере углубления нашего понимания существующих проблем (или надеяться на это!). Термин **семантическая модель,** который часто используется по отношению к той или иной "расширенной" модели, в данном случае не совсем подходит, поскольку предполагает, что модель построена таким образом, чтобы выявить *всю* семантику рассматриваемой ситуации. С другой стороны, понятие **семантическое моделирование** *действительно* является довольно удачным названием целой области исследований, посвященных способам представления смыслового значения. В данной главе сначала представлено краткое введение в некоторые основные идеи этой области исследований, а затем подробно описан один из конкретных подходов, основанный на использовании модели "сущность—связь" (наиболее распространенный на практике).

Следует отметить, что семантическое моделирование имеет также несколько других названий, например, *моделирование данных, ER-моделирование, моделирование сущностей* и *объектное моделирование.* В настоящей книге по указанным ниже причинам предпочтение отдано термину *семантическое моделирование* (которое иногда называют также *концептуальным моделированием).*

- Термин *моделирование данных* не подходит, поскольку он конфликтует со сложившимся ранее определением термина *модель данных,* который обозначает формальную систему наподобие реляционной модели. К тому же термин *моделирование данных* способствует укреплению широко распространенного заблуждения, что модель данных (в приведенном выше смысле) охватывает *только* структуру данных. ***Примечание.*** Стоит напомнить (см. раздел 1.3 главы 1), что термин *модель данных* используется в литературе с двумя совершенно разными значениями: первое относится к модели данных вообще (реляционная модель является моделью данных именно в этом смысле), а второе — к модели представления перманентных данных *некоторого конкретного предприятия.* Здесь рассматриваемый термин в последнем смысле не используется, но следует иметь в виду, что он часто применяется в таком смысле другими авторами.
- Также неудачным является термин *ER-моделирование,* поскольку в нем неявно подразумевается существование только одного подхода к данной проблеме, тогда как на практике, безусловно, существует много различных подходов. Тем не менее, термин *ER-моделирование* хорошо известен, весьма популярен **и** принят в широком кругу специалистов.

- Относительно термина *моделирование сущностей* нет никаких значительных возражений за исключением того, что он кажется более узко специализированным по сравнению с термином *семантическое моделирование* и носит определенный смысловой акцент, который не совсем подходит в данном случае.
- Что касается термина *объектное моделирование,* то проблема здесь заключается в том, что термин *объект* в данном контексте, очевидно, является синонимом термина *сущность,* тогда как он используется в совершенно другом смысле и другом контексте (в частности, в контексте баз данных других типов, рассматриваемых в главе 25 этой книги). По мнению автора, именно этот факт (наличие у данного термина двух разных значений) явился причиной такого явления, как **Первое Серьезное Заблуждение** (First Great Blunder). Более подробно этот вопрос рассматривается в главе 26.

Возвращаясь к главной теме нашего обсуждения, следует остановиться на причинах включения данного материала в эту часть книги. *Идеи семантического моделирования могут быть полезны как средство проектирования базы данных даже при отсутствии их непосредственной поддержки в СУБД.* Аналогично тому, как идеи исходной реляционной модели использовались в качестве примитивного вспомогательного средства при проектировании баз данных задолго до появления любых коммерческих воплощений этой модели, так и идеи некоторой "расширенной" модели могут оказаться полезными в качестве вспомогательного средства проектирования базы данных, даже если никакого коммерческого воплощения этих идей еще не существует. Действительно, было бы справедливо отметить, что на момент написания этой книги идеи семантического моделирования оказали на проектирование баз данных *большое влияние* — было предложено несколько технологий проектирования, основанных на том или ином семантическом подходе. Поэтому в данной главе основной акцент будет сделан на применении идей семантического моделирования к задаче проектирования базы данных.

Материал этой главы упорядочен следующим образом. После данного введения, в разделе 14.2, описываются основные термины, используемые в семантическом моделировании. Затем, в разделе 14.3, обсуждается наиболее известная из расширенных моделей — предложенная Ченом (Chen) *модель "сущность-связь"* (или ER-модель; entity/relationship model — ER-model). Далее, в разделах 14.4 и 14.5, рассматривается применение этой модели для проектирования базы данных. (Другие модели кратко описаны в аннотациях к некоторым публикациям, упомянутым в списке рекомендуемой литературы.) После этого, в разделе 14.6, предлагается краткий анализ некоторых аспектов ER-модели, и наконец, в разделе 14.7, приводится краткое резюме.

14.2. ОБЩИЙ ПОДХОД

Общий подход к проблеме семантического моделирования характеризуется четырьмя основными этапами.

1. Прежде всего, попытаемся выявить некоторое множество *семантических* концепций (понятий), которые могут быть полезны при неформальном обсуждении рассматриваемой проблемы реального мира.
   - Например, можно согласиться с тем, что мир построен из **сущностей.** (Хотя невозможно с определенной точностью описать, что именно представляет собой

сущность, эта концепция, тем не менее, оказывается весьма полезной для описания реального мира, по крайней мере, на интуитивном уровне.)

- Развивая данную концепцию, можно допустить, что сущности могут быть с пользой классифицированы по разным **типам сущностей.** Например, можно предположить, что каждый отдельный работник является **экземпляром** неко торого универсального **типа** сущности с именем EMPLOYEE (Работник). Пре имущество такой классификации заключается в том, что все сущности опреде ленного типа будут обладать некоторыми общими **свойствами** (например, все работники получают зарплату), поэтому подобная классификация может при вести к некоторой (и совершенно очевидной) *систематизации представлений.* Например, в терминах реляционной теории выявленная общность может быть зафиксирована в заголовке переменной отношения.
- Более того, можно пойти еще дальше и согласиться с тем, что каждая сущность обладает неким особым свойством, предназначенным для ее *идентификации,* т.е. с тем, что каждая сущность обладает собственной **идентичностью.**
- Наконец, можно также предположить, что каждая сущность может быть связа на с другими сущностями с помощью некоторых **связей.**

Аналогичные рассуждения можно продолжать и дальше. Но здесь следует отметить, что все эти термины *(экземпляр сущности, тип сущности, свойство, связь* и т.д.) определены *не* совсем точно или формально, поскольку они являются всего лишь концепциями "реального мира", а не формальными терминами. Таким образом, данный первый этап является неформальным, тогда как приведенные ниже второй, третий и четвертый этапы, напротив, достаточно формальны.

2. Далее попытаемся определить набор соответствующих *символических* (т.е. фор мальных) **объектов,** которые могут использоваться для представления описанных выше семантических концепций. *(Примечание.* Здесь термин *объект* не использу ется в каком-то строго определенном смысле!) Например, *в расширенной реляцион ной модели* (RM/T) [14.7] вводится несколько особых видов отношений, которые называются *Е-* и *Р-отношениями.* Грубо говоря, Е-отношения (от "entity-relation") представляют сущности, а Р-отношения (от "property-relation") — свойства, одна ко Е- и Р-отношения, безусловно, имеют конкретные формальные определения, тогда как сами сущности и свойства их не имеют.
3. Кроме того, следует определить набор формальных общих **правил целостности** (или, используя терминологию главы 9, *метаограничений),* предназначенных для работы с такими формальными объектами. Например, RM/T-модель включает правило *целостности свойств,* которое гласит, что для каждого элемента Р-отношения должен существовать соответствующий ему элемент в Е-отношении (это отражает тот факт, что каждое свойство в базе данных должно быть свойством некоторой сущности).
4. Наконец, необходимо также определить набор формальных **операторов,** предназ наченных для манипулирования этими формальными объектами. Например, в RM/T-модели присутствует оператор PROPERTY, который можно использовать для соединения Е-отношения со всеми соответствующими ему Р-отношениями независимо от того, сколько их и какие им присвоены имена, т.е. оператор, позво ляющий собрать воедино все свойства любой сущности.

Описанные в пп. 2—4 объекты, правила и операторы совместно образуют расширенную модель данных, но только если эти конструкции действительно являются супермножеством конструкций одной из базовых моделей, например, такой как базовая реляционная модель. Однако на самом деле в данном контексте нет четкого различия между тем, что является расширенным и что базовым. Обратите особое внимание на то, *что правила и операторы являются такой же частью расширенной модели, как и объекты* (безусловно, это утверждение справедливо и для базовой реляционной модели). Тем не менее, с точки зрения проектирования баз данных операторы являются менее важной частью модели по сравнению с объектами и правилами целостности[2]. Поэтому, за исключением нескольких посвященных операторам комментариев, основное внимание в этой главе уделяется именно объектам и правилам.

Напомним, что на первом этапе была предпринята попытка выявить множество семантических концепций, которые были бы полезны для описания реального мира. Некоторые из этих концепций, а именно — сущности, свойства, связи и подтипы, представлены в табл. 14.1 с указанием неформального определения и нескольких типичных примеров. Обратите внимание на то, что все эти специально подобранные примеры иллюстрируют возможность рассмотрения одного и того же объекта реального мира одними пользователями в качестве сущности, другими — в качестве свойства, а третьими — в качестве связи. (Этот пример прекрасно демонстрирует, почему невозможно дать строгое определение такого термина, как *сущность.)* Одна из целей семантического моделирования (несомненно, полностью достигнутая) как раз и заключается в поддержке такой *гибкости интерпретации.*

Таблица 14.1. Некоторые полезные семантические концепции

Понятие	Неформальное определение	Примеры
СУЩНОСТЬ (Entity)	Некоторый различимый объект	Поставщик, деталь, поставка Работник, отдел, человек Произведение, концерт, оркестр, дирижер Заказ на поставку, серия заказов
СВОЙСТВО (Property)	Элемент информации, описывающий сущность	Номер поставщика, поставляемое количество Отдел работника, рост человека Категория концерта Дата заказа
СВЯЗЬ (Relationship)	Сущность, которая служит для обеспечения взаимодействия между двумя или несколькими другими сущностями	Поставка (поставщик–деталь) Должность (работник–отдел) Запись (произведение–оркестр–дирижер)
ПОДТИП (Subtype)	Сущность типа Y является подтипом сущности типа X тогда и только тогда, когда каждый экземпляр сущности типа Y обязательно является экземпляром сущности типа X	"Работник" является подтипом сущности "Человек" "Концерт" является подтипом сущности "Произведение"

[2] Если не считать того, что без операторов невозможно сформулировать сами правила.

Кроме того, следует отметить, что вполне возможно возникновение конфликтов между терминами, которые представлены в табл. 14.1 и используются на семантическом уровне, и терминами, которые используются в рамках выбранного формального подхода, например в реляционной модели. В частности, во многих схемах семантического моделирования вместо термина *свойство* используется термин *атрибут,* но при этом отнюдь не подразумевается, что такой атрибут представляет (или отображается на) то же самое, что и атрибут реляционного уровня. Еще одним (важным!) примером может быть то, что концепция *типа сущности,* как она понимается в ER-модели, отличается от концепции *типа,* которая рассматривалась в главе 5. Точнее говоря, подобным типам сущностей в реляционном проекте соответствуют *переменные отношения,* поэтому они, очевидно, не соответствуют реляционными типам *атрибута* (доменам). Однако по перечисленным ниже причинам они не полностью соответствуют и типам *отношений.*

1. На семантическом уровне некоторые *базовые* типы отношений, вероятно, будут соответствовать типам связей, а не типам сущностей.
2. Говоря упрощенно, некоторые *производные* типы отношений могут вообще ничему не соответствовать на семантическом уровне, а другие— могут.

Путаница при определении этих уровней (в частности, вследствие несогласованности используемой терминологии) как в прошлом, так и в настоящем часто является причиной дорогостоящих ошибок (см. раздел 26.2 главы 26).

В заключение следует отметить, что в главе 1 связи рассматривались как сущности особого рода, причем было указано, что подобным же образом они будут рассматриваться во всей книге. Кроме того, в главе 3 как одно из преимуществ реляционной модели отмечалось единство представления всех типов сущностей, включая связи, с помощью некоторого единообразного способа, а именно кортежей отношений. Тем не менее, концепция связей (так же, как и концепция сущностей) действительно полезна при описании реального мира на *интуитивном уровне.* Более того, представленный ниже, в разделах 14.3—14.5, подход к проектированию базы данных будет в значительной степени опираться на различия между сущностями и связями. Поэтому в нескольких следующих разделах будет принята терминология, предусматривающая разделение понятий сущностей и связей, однако в разделе 14.6 по этому вопросу будут представлены некоторые дополнительные замечания.

14.3. МОДЕЛЬ "СУЩНОСТЬ-СВЯЗЬ"

Как уже упоминалось в разделе 14.1, одним из наиболее известных и получивших широкое распространение методов семантического моделирования является метод построения модели **"сущность—связь"** (или ER-модели). Этот подход основан на использовании модели "сущность—связь", предложенной Ченом в 1976 году [14.6] и с тех пор неоднократно дополнявшейся как самим Ченом, так и многими другими исследователями (об этом можно прочесть, например, в [14.18], [14.45]—[14.47]). Дальнейшее обсуждение в настоящей главе в основном посвящено именно данному подходу. (Следует подчеркнуть, что модель "сущность-связь" является далеко не единственной "расширенной" моделью, кроме нее, было предложено очень много других моделей. В частности, в [14.6], [14.18], [14.30], [14.37] и особенно в [14.24] приведены общие вводные сведения по некоторым из них, а в [14.27] и [14.36] даны вводные обзоры по рассматриваемой теме.)

ER-модель включает аналоги всех семантических объектов, представленных в табл. 14.1, каждый из которых подробно рассматривается далее в этой главе. Прежде всего отметим, что в [14.6] была предложена не только сама ER-модель как таковая, но и соответствующая ей **технология построения диаграмм,** получивших название *ER-диаграммы.* Более подробно ER-диаграммы описываются в следующем разделе, а на рис. 14.1 показан простой пример подобной диаграммы, взятый из [14.6]. Это пример представления структуры данных некоторой вымышленной производственной компании (KnowWare, Inc.). Он будет полезен при изучении последующего материала данной главы и подробно описывается ниже. (Это — расширенная версия ER-диаграммы, приведенной на рис. 1.6 в главе 1.)

Примечание. Большинство понятий, рассматриваемых в следующих подразделах, хорошо известны тем, кто знаком с реляционной моделью. Но, как будет показано, между используемыми в этих двух моделях терминами существуют определенные различия.

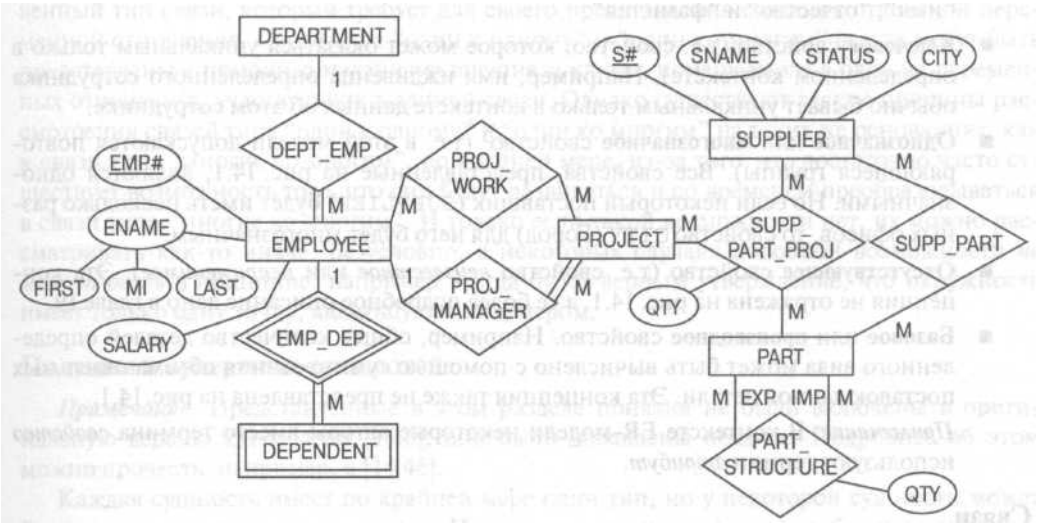


Рис. 14.1. Диаграмма модели "сущность—связь" (сокращенная версия для рассматриваемого примера)

Сущности

Работа [14.6] начинается с определения **сущности** (entity) как "понятия, которое может быть четко идентифицировано". **При** этом сущности подразделяются на **обычные** и **слабые.** Слабой называется такая сущность, существование которой зависит от другой сущности, т.е. она не может существовать, если этой другой сущности не существует. Например, на рис. 14.1 сущность DEPENDENT (Иждивенец работника) является слабой, поскольку она не может существовать (в контексте рассматриваемой базы данных), если не существует соответствующей сущности EMPLOYEE (Работник). В частности, если сведения о некотором работнике (сущность EMPLOYEE) будут удалены, то и сведения обо всех его иждивенцах (сущности DEPENDENT) также должны быть удалены. *Обычной* называется сущность, которая не является слабой. В данном примере сущность EMPLOYEE — обычная.

Примечание. Некоторые авторы вместо термина *обычная сущность* применяют термин *сильная сущность.*

Свойства

Сущности (и связи) обладают некоторыми **свойствами** (property). Все сущности или связи одного и того же типа обладают определенными общими свойствами. Например, все работники имеют табельный номер, имя, зарплату и т.д. *(Примечание.* В данном примере среди свойств работника сознательно не упоминается номер отдела; причина этого разъясняется в следующем подразделе.) Значения свойств каждого типа извлекаются из соответствующего **множества значений,** которое в реляционной терминологии называется доменом (или типом). Ниже перечислены некоторые характеристики свойств и указаны их особенности.

- **Простое** или **составное** свойство. Например, свойство "имя работника" может быть составным, если его значение составляется из значений простых свойств "имя", "отчество" и "фамилия".
- **Ключевое** свойство (т.е. свойство, которое может оказаться уникальным только в определенном контексте). Например, имя иждивенца определенного сотрудника обычно бывает уникальным только в контексте данных об этом сотруднике.
- **Однозначное** или **многозначное** свойство[3] (т.е. в этой модели допускаются повто ряющиеся группы). Все свойства, представленные на рис. 14.1, являются одно значными. Но если некоторый поставщик (SUPPLIER) будет иметь несколько раз ных офисов, то свойство CITY (Город) для него будет многозначным.
- Отсутствующее свойство (т.е. свойство *неизвестное* или *неприменимое).* Эта кон цепция не отражена на рис. 14.1, а ее более подробное описание дано в главе 19.
- **Базовое** или **производное** свойство. Например, общее количество деталей опреде ленного вида может быть вычислено с помощью суммирования объема отдельных поставок данной детали. Эта концепция также не представлена на рис. 14.1. *Примечание.* В контексте ER-модели некоторые авторы вместо термина *свойство* используют термин *атрибут.*

Связи

В [14.6] **связь** (relationship) определяется как "ассоциация, объединяющая несколько сущностей". Например, между отделами и работниками существует связь с именем DEPT_EMP. Она представляет тот факт, что в каждом отделе работает определенное количество работников. Так же, как и в отношении сущностей (см. главу 1), необходимо понимать принципиальную разницу между типами и экземплярами связей, однако в неформальном описании такими тонкостями можно пренебречь, что мы и будем неоднократно делать в будущем.

Сущности, включенные в связь, называются ее **участниками,** а количество участников связи называется ее **степенью.** (Следует отметить, что в данном случае определение термина *степень* отличается от его определения в реляционной модели.)

Пусть R является типом связи, включающей тип сущности E в качестве участника. Если каждый экземпляр сущности E участвует по крайней мере в одном экземпляре связи R, то участие сущности E в связи R называется **полным,** в противном случае — **частичным.**

---

[3] Определение этого термина приведено в подразделе "Атрибуты со значениями в виде отношения" раздела 6.4.

Например, если каждый работник обязательно должен относиться к определенному отделу, то участие сущности EMPLOYEE В СВЯЗИ между работниками и отделами (DEPT_EMP) является полным. С другой стороны, если допустима ситуация, когда в некотором отделе (например, вновь созданном) не будет ни одного работника, участие сущности DEPARTMENT В СВЯЗИ DEPT_EMP будет ЧасТИЧНЫМ.

Связи в модели "сущность-связь" могут иметь тип **"один к одному", "один ко многим"** (иначе может называться **"многие к одному")** или **"многие ко многим".** (Для упрощения изложения далее предполагается, что все связи являются двухсторонними, т.е. имеют степень *два,* хотя, безусловно, изложенные концепции и терминологию можно без труда расширить и на связи с более высокой степенью.) Здесь читатель, уже знакомый с основами реляционной модели, мог бы заметить, что именно тип связи "многие ко многим" является единственным типом, представляющим истинную связь, поскольку это единственный тип связи, который требует для своего представления создания отдельной переменной отношения. Связи типа "один к одному" и "один ко многим" всегда могут быть представлены с помощью механизма внешнего ключа, помещаемого в одну из переменных отношения, участвующих в данной связи. Однако существуют веские причины рассмотрения связей типа "один к одному" и "один ко многим" на таких же основаниях, как и связи типа "многие ко многим", по крайней мере, из-за того, что достаточно **часто** существует возможность того, что они будут развиваться и со временем преобразовываться в связи типа "многие ко многим". И только если такой возможности нет, их можно рассматривать как-то иначе. Безусловно, в некоторых случаях подобной возможности *не может быть* в принципе; например всегда будет верным утверждение, что окружность имеет только одну точку, являющуюся ее центром.

Подтипы и супертипы сущностей

Примечание. Представленные в этом разделе понятия не были включены в оригинальную версию ER-модели [14.6]; **они** были добавлены позднее. Подробнее об этом можно прочесть, например, в **[14.46].**

Каждая сущность имеет по крайней мере один тип, но у некоторой сущности может быть одновременно несколько **типов.** Например, если определенные работники являются программистами (и все программисты являются работниками), то можно сказать, что тип сущности PROGRAMMER (программист) представляет собой **подтип** типа сущности EMPLOYEE (работник). (Или, что эквивалентно, тип сущности EMPLOYEE является **супертипом** типа сущности PROGRAMMER.) Программисты автоматически обладают всеми свойствами работников, однако обратное утверждение неверно (например, для программистов может быть задано свойство "владение языком программирования", которое в общем случае не применимо ко всем работникам). Аналогичным образом, сущность PROGRAMMER автоматически участвует во всех связях, в которых участвует сущность EMPLOYEE, однако обратное утверждение также неверно (например, программисты могут входить в общество компьютерных специалистов, в которое прочие работники в общем случае не входят). Поэтому считается, что подтип **наследует** свойства и связи супертипа.

Обратите внимание на то, что одни программисты (сущность PROGRAMMER) могут быть прикладными программистами (сущность APPLICATION_PROGRAMMER), а другие — системными программистами (SYSTEM_PROGRAMMER). Исходя из этого, можно сказать, ЧТО ТИПЫ APPLICATION_PROGRAMMER И SYSTEM_PROGRAMMER являются **подтипами**

супертипа PROGRAMMER и т.д. Иначе говоря, сущность-подтип по-прежнему является типом сущности, поэтому может иметь собственные подтипы. Некоторый тип сущности, его непосредственные подтипы, подтипы этих подтипов и т.д. все вместе образуют **иерархию типов сущности,** пример которой представлен на рис. 14.2.

Здесь стоит подробно рассмотреть приведенные ниже особенности.

1. Поскольку детальное обсуждение иерархии и наследования типов будет отложено до главы 20, следует предупредить читателя, что в данной главе термин *тип* имеет такое же значение, как и в главе 5 (т.е. он *не* означает *тип сущности).*
2. Для читателей, знакомых с СУБД IMS (или какой-нибудь иной СУБД, в которой используется иерархическая структура данных), необходимо отметить, что иерар хии типов *не* следует путать с иерархиями данных. Например, на рис. 14.2 вовсе не подразумевается, что в расчете на одного работника (EMPLOYEE) имеется несколь ко соответствующих программистов (PROGRAMMER). Наоборот, для одного экзем пляра типа сущности EMPLOYEE существует *не больше* одного соответствующего экземпляра типа сущности PROGRAMMER, представляющего того же работника в роли программиста (как было бы, если бы этот рисунок представлял иерархию в стиле IMS).

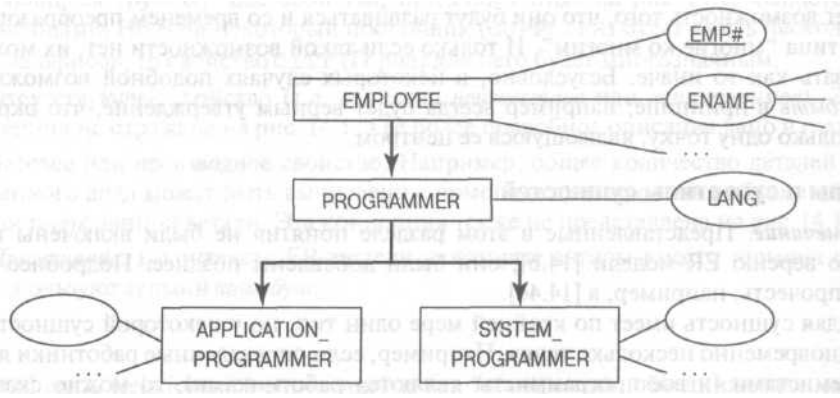


Рис. 14.2. Пример иерархии типов сущностей

На этом краткое обсуждение основных структурных особенностей ER-модели завершено, и можно перейти к рассмотрению ER-диаграмм.

14.4. ER-ДИАГРАММЫ

Как уже указывалось в предыдущем разделе, в [14.6] была не только введена сама модель "сущность—связь", но и представлена концепция **ER-диаграмм.** Такая диаграмма является методом представления логической структуры базы данных в графическом виде для более простого и наглядного отображения основных компонентов конкретного проекта базы данных (один рисунок порой стоит тысячи слов). Действительно, популярность методов ER-моделирования как подхода для проектирования баз данных, скорее всего, объясняется именно наличием подобной диаграммной технологии, а не чем-либо иным. Ниже правила создания ER-диаграмм поясняются на примерах, представленных на рис. 14.1 и 14.2.

Примечание. Так же, как и сама модель "сущность—связь", технология создания ER-диаграмм постоянно совершенствуется, поэтому в данном разделе будет описана та ее версия, которая отличается в некоторых важных аспектах от оригинальной методики, предложенной в [14.6].

Сущности

Каждый тип сущности на ER-диаграмме представляется в виде отдельного прямоугольника с указанным внутри именем сущности, причем прямоугольники сущностей слабых типов обозначаются двойной линией.

Соответствующие примеры приведены ниже (см. рис. 14.1).

- Обычные сущности:
- DEPARTMENT (Отдел);
- EMPLOYEE (Работник);
- SUPPLIER (Поставщик);
- PART (Деталь);
- PROJECT (Проект).
- Слабая сущность:
- DEPENDENT (Иждивенец).

Свойства

Свойства отображаются на ER-диаграмме в виде эллипсов, содержащих имена этих свойств. Эллипсы соединяются с соответствующей сущностью (или связью) сплошной линией. Контур эллипса изображается штриховой или пунктирной линией, если свойство производное, и двойной линией, если свойство многозначное. Если свойство составное, то составляющие его свойства показаны в виде других эллипсов, соединенных с эллипсом составного свойства с помощью дополнительных сплошных линий. Имена ключевых свойств обычно подчеркиваются, а множества значений не отображаются совсем.

Соответствующие примеры приведены ниже (см. рис. 14.1).

- Для сущности EMPLOYEE:
  EMP# (Табельный номер работника) — ключевое свойство;
  ENAME (Полное имя) — составное, состоящее из свойств FIRST (Имя),
  Mi (Отчество) и LAST (Фамилия);
  SALARY (Зарплата).
- Для сущности SUPPLIER:
  S# (Номер поставщика) — ключевое свойство;
  SNAME (Имя поставщика);
  STATUS (Статус поставщика);
  CITY (Город, в котором находится поставщик).
- Для связи SUPP_PART_PROJ:
  QTY (Количество).
- ДЛЯ СВЯЗИ PART_STRUCTURE:
  QTY (Количество).

Для экономии места другие свойства, представленные на рис. 14.1, не показаны.

Связи

Каждый тип связи показан на ER-диаграмме в виде ромба с именем связи внутри. Ромб рисуется двойной линией, если это связь между слабым типом сущности и типом сущности, от существования которого она зависит. Участники каждой связи соединяются с ромбом соответствующей связи сплошными линиями. Каждая такая линия содержит обозначение "1" или "M", которое указывает, относится ли связь к типу "один к одному", "один ко многим" или "многие ко многим"). Двойная линия обозначает полное участие в связи данной стороны.

Соответствующие примеры приведены ниже (см. рис. 14.1).

- DEPT_EMP (СВЯЗЬ ТИПа "ОДИН КО МНОГИМ" МеЖДУ СУЩНОСТЯМИ DEPARTMENT И EMPLOYEE).
- EMP_DEP (связь типа "один ко многим" между сущностью EMPLOYEE и сущностью слабого типа DEPENDENT).
- PROJ_WORK и PROJ_MANAGER (обе связи установлены между сущностями EMPLOYEE и PROJECT, причем первая имеет тип "многие ко многим", а вторая — "один ко многим").
- SUPP_PART_PROJ (связь типа "многие ко многим и ко многим" между сущностя ми SUPPLIER, PART И PROJECT).
- SUPP_PART (связь типа "многие ко многим" между сущностями SUPPLIER И PART).
- PART_STRUCTURE (связь типа "многие ко многим" между сущностями PART и PART).

Обратите внимание на то, что в последнем Случае две линии от PART К PART_STRUCTURE отличаются надписями с указанием различных выполняемых **ролей** (EXP и IMP, которые обозначают, соответственно, "разборку узла на детали" и "сборку узла из деталей"). Связь PART_STRUCTURE является типичным примером **рекурсивной связи.**

Подтипы и супертипы сущностей

Пусть тип сущности Y является подтипом типа сущности X. Тогда от прямоугольника Y к прямоугольнику x можно провести сплошную линию со стрелкой на конце возле Y. Эта линия представляет то, что иногда называют *связью принадлежности* (is-a relationship) (поскольку множество всех сущностей типа Y *является* ("is a") подмножеством множества всех сущностей типах).

Соответствующие примеры приведены ниже (см. рис. 14.2).

- ТИП СУЩНОСТИ PROGRAMMER ЯВЛЯеТСЯ ПОДТИПОМ ТИПЭ СУЩНОСТИ EMPLOYEE.
- Типы сущностей APPLICATION_PROGRAMMER И SYSTEM_PROGRAMMER ЯВЛЯЮТСЯ подтипами типа сущности PROGRAMMER.

14.5. ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ С ПОМОЩЬЮ МЕТОДА ER-МОДЕЛИРОВАНИЯ

В определенном смысле построенная в соответствии с описанными выше правилами ER-диаграмма *является* проектом базы данных. Если попытаться отобразить подобный проект[4] на некоторую формальную схему, соответствующую требованиям конкретной СУБД, то станет ясно, что обычная ER-диаграмма для этого недостаточно точна и в ней отсутствует описание множества важных деталей (особенно тех, которые относятся к целостности данных). Для иллюстрации этого утверждения рассмотрим, что происходит при попытке отобразить проект базы данных, показанный на рис. 14.1, на набор определений компонентов реляционной базы данных.

Обычные сущности

На рис. 14.1 показаны следующие обычные типы сущностей.

- DEPARTMENT.
- EMPLOYEE.
- SUPPLIER.
- PART.
- PROJECT.

Каждый обычный тип сущности отображается на базовую переменную отношения. Следовательно, рассматриваемая база данных будет содержать пять базовых переменных отношения, например, DEPT, EMP, s, P и J, соответствующих этим пяти типам сущности. Более того, каждое из базовых отношений будет иметь потенциальный ключ (DEPT#, EMP#, s#, P# и J#), соответствующий указанным на ER-диаграмме *ключевым свойствам.* Для определенности допустим, что в каждой из создаваемых переменных отношения соответствующий потенциальный ключ определяется как *первичный.* В качестве примера ниже приводится определение переменной отношенияБЕРТ (в сокращенном виде).

```
VAR DEPT BASE RELATION
       { DEPT# . . . , ... }
       PRIMARY KEY { DEPT#
       } ;
```

Читателю предлагается в качестве упражнения записать определения остальных четырех переменных отношения.

Примечание. Безусловно, в документации также должны быть зафиксированы определения доменов и допустимых множеств значений. Однако подробности здесь не даны, поскольку, как уже отмечалось, множества значений на ER-диаграммах не отображаются.

Связи типа "многие ко многим"

В рассматриваемом примере присутствуют следующие связи типа "многие ко многим" (или "многие ко многим и ко многим" и т.д.).

---

[4] В настоящее время существует множество инструментов, позволяющих выполнить такое отображение (например, с использованием ER-диаграммы для генерации соответствующего набора операторов CREATE TABLE на языке SQL).

- PROJ_WORK (между работниками и проектами).
- SUPP_PART (между поставщиками и деталями).
- SUPP_PART_PROJ (между поставщиками, деталями и проектами).
- PART_STRUCTURE (между деталями-узлами и деталями, входящими в их состав)

Каждая такая связь отображается и на базовую переменную отношения. Таким образом, вводятся еще четыре базовые переменные отношения, соответствующие четырем указанным связям. Допустим, что для связи SUPP_PART такой базовой переменной отношения является SP (уже знакомая нам переменная отношения поставок). Временно отложим описание ее первичного ключа и обратимся к *внешним* ключам этой переменной отношения, которые необходимы для идентификации участников данной связи.

```
VAR SP BASE RELATION SP
    { S# ... , P# ... , ... }

    FOREIGN KEY { S# } REFERENCES
    S FOREIGN KEY { P# } REFERENCES
    P ;
```

Ясно, что такая переменная отношения должна включать два внешних ключа (s# и P#), соответствующих двум участникам связи (сущностям SUPPLIER и PART), и эти внешние ключи должны ссылаться на соответствующие переменные отношения s и p. Более того, для каждого из внешних ключей должен быть задан подходящий набор *правил внешних ключей,* т.е. правило обновления UPDATE и правило удаления DELETE.

Примечание. Данные конкретные правила показаны здесь лишь в качестве иллюстрации (они являются не единственными возможными). Что еще более важно, необходимо учитывать, что какие бы правила не были определены, их нельзя вывести на основании ER-диаграммы или указать на этой диаграмме.

Ниже приведены правила, которые следует задать в случае базовой переменной отношения SP.

```
VAR SP BASE RELATION SP
    { S# ...,  P# ...,... }

     FOREIGN KEY { S# } REFERENCES S
            ON DELETE RESTRICT
            ON UPDATE CASCADE
     FOREIGN KEY { P# } REFERENCES
     P
            ON DELETE RESTRICT
            ON UPDATE CASCADE ;
```

Что можно сказать о первичном ключе этой переменной отношения? Одним из возможных способов его определения может быть применение комбинации внешних ключей, идентифицирующих участников соответствующей связи (в случае базовой переменной отношения SP ими являются атрибуты s# и P#). Это возможно, *если* данная комбинация имеет уникальное значение для каждого экземпляра данной связи (это условие может соблюдаться или не соблюдаться, но обычно оно соблюдается) и *если* разработчик базы данных не возражает против использования составных первичных ключей (на практике они в равной степени могут применяться и не применяться). В качестве альтернативного варианта первичного ключа можно использовать новый несоставной *замещающий* атрибут, допустим, "номер поставки" (подробные сведения приведены в [14.11] и [14.21]).

В данном примере будет использован первый из двух описанных выше вариантов. Таким образом, в определение SP необходимо добавить следующее предложение.

```
PRIMARY KEY { S#, P# }
```

В качестве упражнения читателю предлагается самостоятельно рассмотреть связи PROJ_WORK, PART_STRUCTURE И SUPP_PART_PROJ.

Связи типа "многие к одному"

В рассматриваемом примере присутствуют три связи типа"многие к одному".

- PROJ_MANAGER (между проектами и их руководителями).
- DEPT_EMP (между работниками и отделами).
- EMP_DEP (между иждивенцами и работниками).

Только в последней из трех связей участвует слабый тип сущности (DEPENDENT), тогда как в двух других участвуют только обычные типы сущностей. Связь со слабым типом сущности будет обсуждаться несколько позже, а сейчас рассмотрим какую-либо из двух других связей, например DEPT_EMP. В данном случае не требуется вводить никаких новых переменных отношения[5]. Вместо этого достаточно просто ввести приведенный ниже внешний ключ в переменную отношения, расположенную со стороны "многие" (EMP), который будет ссылаться на переменную отношения на стороне "один" (DEPT).

```
VAR EMP BASE RELATION
      { EMP# ..., DEPT# ...,
      ... } PRIMARY KEY { EMP#
      }
      FOREIGN KEY ( DEPT# ) REFERENCES
             DEPT ON DELETE ... ON UPDATE
             ... ;
```

В данном случае возможности определения правил удаления DELETE и обновления UPDATE точно такие же, как и в случае внешнего ключа, представляющего участника связи типа "многие ко многим" (в общем случае). Здесь вновь следует отметить, что они не показаны на данной ER-диаграмме.

Примечание. В рассматриваемом примере предполагается, что связи типа "один к одному" (которые не так уж часто встречаются на практике) следует рассматривать так же, как связи "многие к одному". Подробное описание особенностей отображения связей типа "один к одному" приведено в [14.8].

Слабые сущности

Связь между сущностью слабого типа и той сущностью, от которой она зависит, безусловно, является связью типа "многие к одному", как это уже отмечалось в предыдущем разделе. Однако правила удаления и обновления для этой связи *должны* выглядеть так, как показано ниже.

---

[5] Хотя, возможно, это имело бы некоторый смысл. Как было указано в разделе 14.3, могут существовать веские причины для такого рассмотрения определенных связей типа "многие к одному", будто на самом деле они имеют тип/'многие ко многим". Дополнительную информацию можно найти в части IV работы [19.19].

```
ON DELETE CASCADE
ON UPDATE CASCADE
```

Взятые в совокупности, эти правила выражают обязательную зависимость существования, что иллюстрируется следующим примером.

```
VAR DEPENDENT BASE
  RELATION { EMP# ..., ...
  }

  FOREIGN KEY ( EMP# ) REFERENCES
         EMP ON DELETE CASCADE ON
         UPDATE CASCADE ;
```

Что является первичным ключом данной переменной отношения? Как и в случае связей "многие ко многим", оказалось, что существует несколько вариантов. Одним из вариантов является комбинация внешнего ключа и ключевого свойства слабой сущности, представленного на ER-диаграмме, опять же, *если* разработчик базы данных не возражает против использования составных первичных ключей. Альтернативным вариантом первичного ключа является ключ на основе нового несоставного *замещающего* атрибута (подробные сведения также приведены в [14.11] и [14.21]). В рассматриваемом примере мы применим первый из двух приведенных выше вариантов, для чего добавим в определение базовой переменной отношения DEPENDENT следующее предложение.

```
PRIMARY KEY { EMP#, DEP_NAME }
```

Здесь DEP_NAME — имя иждивенца данного работника.

Свойства

Каждое показанное на ER-диаграмме свойство отображается на отдельный атрибут в соответствующей переменной отношения, за исключением случая многозначного свойства, для которого обычно приходится создавать новую переменную отношения (как описано в разделе 12.6 главы 12), поскольку атрибуты со значением в виде отношения обычно являются неудобными в использовании. Множества значений отображаются на типы простым и очевидным способом (в связи с тем, что множества значений, безусловно, и являются типами). Эти отображения тривиальны и не требуют дополнительного обсуждения в данном разделе. Но следует отметить, что на первых порах задача выбора подходящих множеств значений может оказаться не такой уж простой!

Супертипы и подтипы сущностей

Поскольку на рис. 14.1 не содержится никаких супертипов или подтипов, далее речь пойдет о примере, представленном на рис. 14.2. Рассмотрим типы сущностей EMPLOYEE и PROGRAMMER. Предположим для простоты, что программисты обладают навыками работы только с одним языком программирования[6] (т.е. свойство LANG является однозначным).

---

[6] Здесь, в частности, следует отметить, что мы не собираемся отображать типы сущностей EMPLOYEE и PROGRAMMER на какие-то конструкции наподобие "супертаблиц" или "подтаблиц". В этом заключается концептуальная трудность или, по крайней мере, ловушка — из того, что на ER-диаграмме тип сущности Y является подтипом типа сущности X, не следует, что реляционный аналог сущности Y является "подчиненным" реляционного аналога сущности X, и это действительно так. Более подробно данная тема рассматривается в [14.13].

- Супертип EMPLOYEE отображается на базовую переменную отношения, например EMP, обычным образом (т.е. так, как уже было описано выше).
- Подтип PROGRAMMER отображается на другую базовую переменную отношения, например PGMR, с таким же первичным ключом, что и у переменной отношения ее супертипа, но с другим набором атрибутов, соответствующим свойствам, кото рые применяются для описания только работников, являющихся программи стами (в нашем примере — LANG).

```
VAR PGMR BASE RELATION { EMP#
  ..., LANG ... } PRIMARY KEY {
  EMP# } ... ;
```

Более того, первичный ключ переменной отношения PGMR является также *внешним* ключом, который ссылается на переменную отношения EMP. Следовательно, приведенное выше определение необходимо соответствующим образом расширить (в частности, обратите внимание на определение правил удаления и обновления).

```
VAR PGMR BASE RELATION {
  EMP# ..., LANG ... }
  PRIMARY KEY { EMP# }
  FOREIGN KEY { EMP# } REFERENCES EMP
          ON DELETE CASCADE ON UPDATE
          CASCADE ;
```

- Нам также потребуется *представление,* например, с именем EMP_PGMR, являющее ся соединением переменных отношения супертипа и подтипа.

```
VAR EMP_PGMR VIEW
    EMP JOIN PGMR ;
```

Обратите внимание на то, что это соединение относится к типу "(нуль или один) к одному". Оно представляет собой соединение по потенциальному ключу и соответствующему внешнему ключу, и этот внешний ключ сам по себе является потенциальным ключом. В результате представление будет содержать сведения только о работниках, являющихся программистами.

Такая структура позволяет выполнять описанные ниже действия.

- С помощью базовой переменной отношения EMP можно получить доступ к тем свойствам, которые являются общими для всех работников (например, для вы борки данных или для их применения в некоторых ограничениях целостности).
- С помощью базовой переменной отношения PGMR МОЖНО получить доступ к свой ствам, характерным только для программистов.
- С помощью представления EMP_PGMR можно получить доступ ко *всему* набору свойств программистов.

1 В базовую переменную отношения EMP можно вставлять сведения о работниках, которые не являются программистами.

- Сведения о работниках-программистах можно вставлять в базу данных с помощью представления EM7P_PGMR.
- Сведения о любых работниках (программистах и не программистах) можно уда лить из базы данных, удалив их из базовой переменной отношения EMP, а сведения

о работниках-программистах можно удалить из базы данных, удалив их из представления EMP_PGMR.

- Свойства, общие для всех работников, можно обновлять в базовой переменной отношения EMP, а свойства только работников-программистов — обновлять и с помощью представления EMP_PGMR.
- Свойства, характерные только для программистов, можно обновлять в базовой переменной отношения PGMR.
- Статус сотрудника, не являющегося программистом, можно изменить на статус программиста за счет вставки сведений об этом сотруднике в базовую переменную отношения PGMR или же в представление EMP_PGMR.
- Статус программиста можно изменить на статус сотрудника, не являющегося программистом, за счет удаления сведений об этом программисте из базовой переменной отношения PGMR.

Предлагаем читателю самостоятельно рассмотреть другие типы сущностей, показанные на рис. 14.2 (APPLICATION_PROGRAMMER И SYSTEM_PROGRAMMER).

14.6. КРАТКИЙ АНАЛИЗ ER-МОДЕЛИ

В этом разделе кратко рассматриваются некоторые аспекты ER-модели. Большая часть излагаемого здесь материала взята из другой работы автора [14.9], в которой эта тема обсуждается подробнее. Дополнительные сведения и комментарии можно найти в аннотациях, помещенных в список рекомендуемой литературы к данной главе.

ER-модель как основа реляционной модели

Рассмотрим подход с использованием ER-модели с несколько иной точки зрения. Вполне очевидно, что идеи, ставшие отправной точкой для разработки этого подхода, во многом очень близки тем идеям, которые послужили Кодду *неформальной* основой при создании им исходной *формальной* реляционной модели. Как уже объяснялось в разделе 14.2, общий подход к разработке "расширенных" моделей включает четыре основных этапа.

1. Выявление полезных семантических концепций.
2. Определение формальных объектов.
3. Определение формальных правил поддержки целостности данных *(метаограничений).*
4. Определение формальных операторов.

Обратите внимание на то, что эти этапы вполне применимы и для разработки реляционной модели (а также любой другой формальной модели данных), а не только "расширенных" моделей, подобных ER-модели. Иначе говоря, для того чтобы создать формальную базовую реляционную модель, Кода прежде всего должен был выявить некоторые неформальные "полезные семантические концепции". Эти концепции в основе своей должны были быть близки идеям ER-моделирования или чему-то очень схожему с ними. Действительно, работы Кодда подтверждают это предположение, поскольку в его первой статье (в самой ранней версии работы [6.1], которая была опубликована в 1969 году), посвященной реляционной модели, имеются следующие строки.

"Множество сущностей заданного типа сущности можно рассматривать как отношение, и такое отношение мы будем называть ***отношением типа сущности...*** *Остальные отношения... между типами сущностей... называются* ***межсущностными отношениями...*** *Важнейшим свойством каждого межсущностного отношения является то, что [оно содержит по крайней мере два внешних ключа, которые] ссылаются на различные типы сущностей либо на общий тип сущности, выполняющий несколько ролей ".*

Здесь Кода явно предлагает использовать отношения для моделирования и *сущностей,* и *связей.* Но самое главное заключается в том (очень важное замечание!), что *отношения являются формальными объектами, а реляционная модель — формальной системой.* Ценность научного вклада Кодда состоит в том, что он нашел удачную *формальную* модель для определенных аспектов реального мира.

В противоположность этому, ER-модель *не* является формальной моделью (или, по крайней мере, является таковой не в первую очередь). Фактически она состоит из набора преимущественно неформальных концепций, соответствующих только первому из четырех приведенных выше этапов. (Более того, ее формальные аспекты на самом деле не очень значительно отличаются от соответствующих аспектов основной реляционной модели; обсуждение этого вопроса будет продолжено в следующем подразделе.) Для проектирования базы данных, безусловно, полезно иметь в своем распоряжении, помимо всего прочего, набор концепций, определенных на этапе 1. Однако несомненным остается тот факт, что проектирование базы данных не может быть завершено без применения формальных объектов и правил, представленных на этапах 2 и 3, а множество других задач и вовсе не может быть решено без использования формальных операторов, определяемых на этапе 4.

Следует отметить, что перечисленные выше замечания не следует рассматривать как стремление доказать бесполезность ER-модели; скорее всего, наоборот. Но одной этой модели недостаточно. Более того, несколько странным кажется тот факт, что первые описания неформальной ER-модели были опубликованы спустя несколько лет после опубликования описания *формальной* реляционной модели, изначально основанной (как мы видели) на некоторых идеях ER-модели.

Является ли ER-модель моделью данных

Из изложенного выше не совсем ясно, является ли ER-модель на самом деле моделью данных, по крайней мере, в указанном в настоящей книге смысле (т.е. формальной системой, включающей структурные аспекты, а также аспекты поддержания целостности и манипулирования данными). Безусловно, термин *ER-моделирование* обычно используется для обозначения процесса выбора только структуры базы данных[7], хотя выше, в разделах 14.3—14.5, рассматривались и некоторые аспекты целостности (они в основном относились к

[7] Фактически основная слабость здесь заключается в том, что ER-модель за исключением некоторых частных (но, по общему признанию, очень важных) случаев *совершенно непригодна* для работы с ограничениями целостности или бизнес-правилами. Процитируем типичное высказывание на этот счет из работы [14.32]: "Декларативные процессы очень сложны для того, чтобы их можно было выразить в виде части бизнес-модели, а потому они должны быть определены отдельно аналитиком-разработчиком". При этом все еще остается в силе такой довод, что проектирование базы данных должно быть процессом точного определения приемлемых ограничений (см. [9.21], [9.22] и [14.22]—[14.24]).

проблеме использования ключей). Но при более внимательном чтении работы [14.6] можно предположить, что ER-модель действительно является моделью данных, но такой, которая представляет собой лишь небольшое дополнение к реляционной модели (и, безусловно, не может заменить реляционную модель, как хотелось бы некоторым авторам). Это утверждение можно обосновать приведенными ниже доводами.

- Фундаментальный элемент данных в ER-модели, т.е. ее фундаментальный *фор мальный* объект, существующий в противоположность неформальным объектам *{сущностям, связям* и т.д.), — это отношение степени п.
- Операторы ER-модели в основном являются операторами реляционной алгебры. (Действительно, работу [14.6] нельзя назвать очень четко разъясняющей эту мысль, но в ней предлагается менее мощное по сравнению с реляционной алгеб рой множество операторов, среди которых, например, не существует объединения и явно заданного соединения.)
- Именно в области поддержки целостности ER-модель обладает некоторым (не большим) дополнительным набором функциональных средств, который отсут ствует в реляционной модели. Дело в том, что ER-модель содержит набор *встроен ных* правил целостности, соответствующих некоторым (но не всем) описанным в настоящей книге правилам удаления и обновления внешнего ключа. Таким образом, там, где для "чистой" реляционной системы потребовалось бы явно сформулировать некоторые правила удаления и обновления для внешних ключей, для ER-модели достаточно было указать, что данная переменная отношения пред ставляет определенный тип связи, — и соответствующие правила для внешних ключей были бы введены.

Сравнительный анализ сущностей и связей

В настоящей книге уже несколько раз отмечалось, что *связи* лучше всего рассматривать просто как сущности особого рода. И наоборот, обязательным условием использования ER-модели является то, что эти два понятия должны каким-то образом различаться. По мнению автора, любой подход, при котором преследуется такое разделение, обладает серьезными недостатками, поскольку, как отмечалось выше, в разделе 14.2, *один и тот же объект* может совершенно обоснованно рассматриваться как сущность одними пользователями и как связь — другими. В частности, рассмотрим приведенный ниже пример с заключением брака.

- С одной стороны, очевидно, что браком называется определенная связь между двумя людьми. В качестве примера можно привести запрос: "С кем вступила в брак Элизабет Тейлор в 1975 году?".
- С другой стороны, не менее очевидно, что понятие брака является самостоятельной сущностью. В качестве примера можно привести следующий запрос: "Сколько браков было заключено в этой церкви с апреля?".

Если в методике проектирования базы данных между сущностями и связями предполагается наличие определенных различий, то в лучшем случае эти две интерпретации будут рассматриваться асимметрично (т.е. запросы для *сущностей* и запросы для *связей* будут выглядеть совершенно по-разному). В худшем случае одна из этих интерпретаций вообще не будет поддерживаться (т.е. один из классов запросов сформулировать будет невозможно).

В качестве еще одной иллюстрации рассмотрим следующее утверждение из учебного пособия по ER-моделированию [14.22].

> *"Обычно на первом этапе в процессе разработки концептуальной схемы принято представлять некоторые связи в виде атрибутов [под этим подразумеваются именно внешние ключи]. Затем эти атрибуты преобразуются в связи по мере дальнейшей разработки проекта и углубления понимания его особенностей".*

Но что произойдет, если атрибут *станет* внешним ключом позже, т.е. если база данных будет доработана уже после того, как она использовалась в течение некоторого времени? Если эту идею логически развить до конца, то можно прийти к выводу, что при проектировании базы данных следует учитывать только связи и совсем не учитывать атрибуты. (На самом деле и такая позиция обладает некоторыми достоинствами; см. аннотацию к [14.23] в конце этой главы.)

Заключительные замечания

Помимо описанной в этой главе схемы ER-моделирования, существует много других семантических схем моделирования. Но большинство из них очень похожи одна на другую; в частности, многие из них можно характеризовать просто как тот или иной вариант графических обозначений для представления некоторых ограничений внешнего ключа, плюс несколько других дополнительных компонентов. Безусловно, подобные графические компоненты могут быть полезны для представления "всей картины в целом", но они слишком просты, чтобы с их помощью можно было выполнить все необходимые проектные работы[8]. В частности, как отмечалось выше, они обычно не подходят для определения общих ограничений целостности. Например, как можно было бы представить на ER-диаграмме наличие общей зависимости соединения?

14.7. РЕЗЮМЕ

Эта глава начиналась с краткого введения в общее семантическое моделирование. В целом, данный процесс состоит из четырех перечисленных ниже этапов, первый из которых является неформальным, а остальные — формальными.

1. Выявление полезных семантических концепций.
2. Определение формальных объектов.
3. Определение формальных правил поддержки целостности данных *(метаограничений).*
4. Определение формальных операторов.

В качестве примера полезных семантических концепций можно назвать концепции сущности, свойства, связи и подтипа.

Примечание. Следует особо подчеркнуть, что весьма вероятно появление терминологических конфликтов между неформальным уровнем семантического моделирования и

---

[8] Печально, но простые решения продолжают очень широко применяться в области информационной технологии даже тогда, когда они *чрезмерно* просты. В таких случаях можно согласиться с Эйнштейном, который однажды заметил, что "все вещи следует делать настолько простыми, насколько это возможно, *но не проще".*

базовым формальным системным уровнем и что наличие подобных конфликтов часто приводит к путанице, поэтому призываем читателя быть внимательным.

Основная цель исследований в области семантического моделирования состоит в том, чтобы сделать СУБД более "интеллектуальными". А более конкретная цель заключается в предоставлении некоторого систематического подхода к решению проблемы **проектирования базы данных. В** настоящей главе было описано применение одной из *семантических* моделей, предложенной Ченом для решения указанной проблемы, а именно — модели **"сущность-связь",** иначе называемой **ER-моделью.**

В связи со сказанным выше стоит повторить мысль о том, что первая статья об ER-моделировании [14.6] на самом деле содержала два различных, более или менее независимых, предложения: *саму* ER-модель, а также технологию **диаграммного ER-моделирования.** Как было отмечено в разделе 14.4, широкую популярность ER-модели, скорее всего, можно объяснить именно наличием этой технологии использования диаграмм, а не какой-либо другой причиной. Однако для использования технологии *ER-диаграмм* совсем не обязательно поддерживать все идеи этой *модели.* Данные диа граммы можно применять в качестве основы в *любой* методике проектирования, напри мер, в методике на основе RM/T-модели, описанной в [14.7]. Эта особенность часто упускают из виду при сравнении удобства использования ER-модели и других методик, разработанных для проектирования базы данных.

Сравним теперь идеи семантического моделирования (и ER-модели в частности) с методом нормализации, описанным в главах 12 и 13. Метод нормализации предусматривает приведение больших переменных отношения к набору переменных отношения меньшего размера. При этом предполагается, что на исходном этапе имеется небольшое количество больших переменных отношения, которые к завершающему этапу после выполнения определенных операций будут преобразованы в множество малых переменных отношения, т.е. произойдет преобразование больших отношений в малые (это, конечно же, *очень* приблизительная формулировка). Однако метод нормализации не дает никаких рекомендаций, касающихся того, каким именно образом могут быть получены исходные большие переменные отношения. Нисходящие методики, подобные описанной в настоящей главе, предназначены для решения именно этой проблемы, т.е. они позволяют отобразить некоторую предметную область реального мира на набор больших переменных отношения. Иначе говоря, эти два подхода (нисходящее проектирование и нормализация) *дополняют друг друга.* Таким образом, общая процедура проектирования базы данных включает два описанных ниже этапа.

1. Использование методов ER-моделирования (или другой аналогичной методики)[9] для создания "больших" переменных отношения, представляющих обычные сущ ности, слабые сущности и т.д.
2. Использование идей дальнейшей нормализации для разбиения созданных "больших" переменных отношения на "малые".

Однако на основании приведенного в данной главе обсуждения можно сделать вывод, что в общем случае семантическое моделирование не является такой же строгой и ясной дисциплиной, как методика дальнейшей нормализации, описанная в главах 12 и 13. Суть

[9] Сам автор предпочитает подход, в котором вначале записываются внешние предикаты, которые описывают предприятие, а затем эти предикаты непосредственно преобразуются во внутренние преди каты, как описано в главе 9.

в том, что (как было указано в начале настоящей главы) проектирование базы данных все еще является весьма субъективным занятием. Оно и не может быть объективным, поскольку существует сравнительно мало действительно строгих принципов, которые могут использоваться для решения этой задачи (немногие существующие на сегодняшний день принципы описаны в двух предыдущих главах). Изложенные в настоящей главе идеи можно рассматривать как чисто эмпирические рекомендации, которые действительно могут быть весьма полезными на практике.

В заключение следует упомянуть еще один очень важный момент. Несмотря на заявление, что эта область исследований все еще остается очень субъективной, в ней есть одна особая часть, в которой идеи семантического моделирования в настоящее время могут быть весьма уместными и полезными. Речь идет о словаре данных, который в определенном смысле можно рассматривать как "базу данных для разработчика баз данных". В словаре данных разработчик может хранить сведения о решениях, принятых в процессе проектирования базы данных [14.2]. Таким образом, изучение семантического моделирования может оказаться чрезвычайно полезным для проектирования системы управления словарем данных, поскольку в нем могут быть указаны типы сущностей, которые словарь должен поддерживать и описывать, например, категории сущностей (такие как обычные и слабые сущности в ER-модели), правила целостности (такие как полное или частичное участие в связи ER-модели), супертипы и подтипы сущностей и т.д.

УПРАЖНЕНИЯ

14.1. Что означает термин *семантическое моделирование?*

14.2. Перечислите четыре основных этапа разработки определения любой "расширенной" модели, подобной технологии ER-моделирования

14.3. Дайте определение следующим терминам ER-модели:

иерархия типов	свойство
ключевое свойство	связь
множество значений	слабая сущность
наследование	супертип, подтип
обычная сущность	сущность

14.4. Предположим, что в ER-диаграмме для поставщиков и деталей обусловлена "полная" степень участия деталей в поставках (т.е. указано, что каждая деталь должна поставляться не меньше чем одним поставщиком). Определите это ограничение:

а) на языке Tutorial D;

б) на языке SQL.

14.5. Приведите примеры связей следующих типов:

а) связь типа "многие ко многим", в которой один из участников является слабой сущностью;

б) связь типа "многие ко многим", в которой один из участников является другой связью;

в) связь типа "многие ко многим", имеющая собственный подтип;

г) подтип, связанный со слабой сущностью, которая не зависит от его супертипа.

14.6. Нарисуйте ER-диаграмму для базы данных повышения квалификации из упр. 9.7 главы 9.

14.7. Нарисуйте ER-диаграмму для базы данных, содержащей информацию о персонале компании, из упр. 12.3 главы 12. Используйте ее для вывода соответствующего на бора определений базовых переменных отношения.

14.8. Нарисуйте ER-диаграмму для базы данных системы ввода заказов из упр. 12.4 гла вы 12. Используйте ее для вывода соответствующего набора определений базовых "переменных отношения.

14.9. Нарисуйте ER-диаграмму для базы данных, содержащей информацию о сбыте, из упр. 13.3 главы 13. Используйте ее для вывода соответствующего набора определе ний базовых переменных отношения.

14.10. Нарисуйте ER-диаграмму для модифицированной базы данных о сбыте из упр. 13.5 главы 13. Используйте ее для вывода соответствующего набора определе ний базовых переменных отношения.

СПИСОК ЛИТЕРАТУРЫ

Обширность предлагаемого списка рекомендуемой литературы объясняется большим количеством конкурирующих методик проектирования баз данных, существующих в настоящее время в деловом и научном мире. Полное согласие в этой области еще не достигнуто, поэтому несмотря на то, что рассмотренная в данной главе методика ER-моделирования является наиболее распространенным подходом, не все специалисты с ней согласны. Действительно, следует отметить, что *самые распространенные* подходы не обязательно являются *самыми лучшими.* Также отметим, что многие коммерческие продукты представляют собой нечто большее, чем просто инструменты разработки баз данных. Часто наряду с предоставлением средств определения структуры базы данных (схемы) они позволяют генерировать целые приложения — пользовательские формы, логику приложений, триггерные процедуры и т.д. (В этой связи следует упомянуть книги Росса по бизнес-правилам [9.21], [9.22], а также [9.15].) В числе другие источников, которые относятся к теме настоящей главы, можно назвать отчет ISO по концептуальным схемам [2.3] и книгу Кента [2.4].

14.1. Abrial J.R. Data Semantics // J. W. KJimbie and K. L. Koffeman (eds.). Data Base Management. — Amsterdam, Netherlands: North-Holland; New York, N.Y.: Elsevier Science, 1974.

Одна из самых ранних публикаций в области семантического моделирования. Следующая цитата прекрасно передает общий дух этой статьи (многие специалисты согласятся с тем, что это относится и ко всей данной тематике в целом): "Совет читателю: если вы хотите найти в данной статье определение термина *семантика,* прекратите чтение, поскольку такого определения здесь нет".

14.2. Bernstein P. A. The Repository: A Modern Vision // DBP&D. — December 1996. — 9, №12.

Похоже, что во время написания этой статьи возникла тенденция к замене термина *словарь* термином *репозитарий.* Репозитарием называется СУБД, предназначенная для управления метаданными, причем не только для СУБД, но и для всех

видов программных инструментов. Приведем цитату из этой статьи Бернстейна: "Речь идет об инструментах проектирования, разработки и распространения программного обеспечения, а также инструментах управления проектами в области электроники, механики, создания Web-сайтов, а также подготовки многих других типов официальных документов, которые относятся к инженерно-конструкторской работе". Эта работа является вводным пособием для изучения концепций репозитария.

14.3. Blaha M., Premerlani W. Object-Oriented Modeling and Design for Database Applications. — Upper Saddle River, N.J.: Prentice-Hall, 1998.

В этой книге подробно описывается методология проектирования Object Modeling Technique (OMT). Ее можно рассматривать, как вариант ER-моделирования *(объекты* в ней соответствуют *сущностям* ER-модели), который представляет собой нечто большее, чем просто методику проектирования *базы данных.* См. также аннотацию к [14.37].

14.4. Booch G. Object-Oriented Design with Applications. — Redwood City, Calif.: Benjamin/ Cummings. — 1991.

См. аннотацию к [14.37].

14.5. Booch G., Rumbaugh J., Jacobson I. The Unified Modeling Language User Guide // Reading, Mass.: Addison-Wesley, 1999.

См. аннотацию к [14.37].

Примечание. Те же авторы выпустили еще две книги по указанной тематике (немного в ином освещении): *The Unified Modeling Language Reference Manual* (Rumbaugh, Jacobson, Booch) и *The Unified Software Development Process* (Jacobson, Booch, Rumbaugh). Обе эти книги были также опубликованы издательством Addison-Wesley в 1999 году.

14.6. Chen P. P.-S. The Entity-Relationship Model - Toward a Unified View of Data // ACM TODS.— March 1976.— 1, №1. (Переиздано: M. Stonebraker (ed.) Readings in Database Systems. — San Mateo, Calif.: Morgan Kaufmann, 1988.)

В статье впервые представлены ER-модель и ER-диаграмма. Как уже говорилось в настоящей главе, с тех пор данная модель претерпела значительные изменения, поскольку приведенные в этой первой статье объяснения и определения, конечно же, были не очень строгими и точными, поэтому нуждались в уточнении. (Одно из чаще всего высказываемых критических замечаний по отношению к ER-моделированию состоит в том, что термины модели не имеют единого и четкого определения, а интерпретируются различными способами. Безусловно, вся область изучения баз данных характеризуется наличием неточной и противоречивой терминологии, однако в наибольшей степени это относится к рассматриваемой здесь области.) Ниже приведено несколько примеров таких не точностей.

- В разделе 14.3 отмечается, что сущность определяется как "понятие, которое может быть четко идентифицировано", а связь — как "ассоциация сущностей". При этом сразу же возникает вопрос: "А является ли связь сущностью?". Ясно, что связь является "понятием, которое может быть четко идентифицировано", но из последующих разделов этой статьи следует, что термин

"сущность" предусмотрен для чего-то, определенно *не* являющегося "связью". Если все же допустить, что это так, то почему модель называется "моделью «сущность—связь»"? В статье все это определено не очень четко.

- Сущности и связи могут иметь *атрибуты* (в этой главе мы использовали тер мин "свойство"). В статье снова не дается четкого определения этому термину, поскольку сначала атрибут определяется как свойство, которое не является первичным ключом или его компонентом (в противоположность реляционно му определению), а затем он используется в стандартном реляционном смысле.
- Предполагается, что первичный ключ связи является комбинацией внешних ключей, которые идентифицируют сущности, включенные в состав данной связи (однако термин "внешний ключ" при этом не используется). Но это до пущение верно только для связей типа "многие ко многими", и то не всегда. Например, рассмотрим переменную отношения SPD{S#, P#,DATE,QTY}, со держащую сведения о поставках некоторых деталей некоторыми поставщика ми по некоторым датам. Предположим, что один и тот же поставщик может поставлять одну и ту же деталь несколько раз, но только по разным датам. То гда первичным ключом (или, по крайней мере, единственным потенциальным ключом) этой переменной отношения является комбинация атрибутов {S#, P#, DATE}, причем поставщики и детали могут быть описаны как сущно сти, а для дат этого сделать нельзя.

14.7. Codd E.F. Extending the Database Relational Model to Capture More Meaning // ACM TODS. - December 1979. - 4, № 4.

В статье представлена "расширенная" версия реляционной модели — RM/T-модель. Можно сразу же отметить несколько различий между RM/T- и ER-моделью. Во-первых, в RM/T-модели не делается никаких различий между сущностями и связями (связь рассматривается всего лишь как особый вид сущности). Во-вторых, структурные и целостные аспекты RM/T-модели более обширны и определены более четко, чем в ER-модели. В-третьих, RM/T-модель содержит несколько специальных операторов в дополнение к операторам базовой реляционной модели (хотя в этой области еще предстоит дополнительная работа). Функционирование RM/T-модели кратко описано ниже.

- Во-первых, сущности (включая "связи") представлены *E-* и *P-переменными отношения*[10], которые являются особыми видами отношений общего типа степени п. E-переменные отношения используются для указания на наличие некоторых сущностей, а P-переменные отношения — для указания некоторых свойств этих сущностей.
- Во-вторых, среди сущностей могут быть заданы разные связи; например, типы сущностей A и B могут образовывать ассоциацию (этот термин принят в RM/T-модели для связи типа "многие ко многим") или же тип сущности Y может быть объявлен подтипом другого типа сущности x. RM/T-модель включает формальную структуру каталога, предназначенную для предоставления системе сведений о существовании подобных связей. В результате система может

[10] В статье они называются E- и P-отношениями.

привести в действие различные **ограничения целостности,** которые обусловлены наличием таких связей.

- В-третьих, для манипулирования различными объектами RM/T-модели (E- и P-переменными отношения, переменными отношения каталога и т.д.) предусмотрены **операторы** высокого уровня.

В RM/T-модели имеются аналоги всех конструкций ER-модели (сущность, свойство, связь, подтип), которые перечислены в табл. 14.1. Точнее, в ней поддерживается **классификационная схема сущностей,** которая во многих случаях представляет наиболее значительный или, по крайней мере, наиболее очевидный аспект всей модели. В соответствии с этой схемой, сущности разделяются на три основные категории: *ядра, характеристики* и *ассоциации.*

- **Ядра.** Это сущности, характеризующиеся *независимым существованием.* Они представляют те объекты, "для описания которых и создается база данных". Иначе говоря, ядрами называются сущности, которые не являются ни характеристиками, ни ассоциациями (см. ниже).
- **Характеристики.** Это сущности, предназначенные для описания или "предоставления характеристики" некоторой другой сущности. Существование характеристик *зависит от существования* описываемых ими сущностей. Описываемая сущность может быть ядром, характеристикой или ассоциацией.
- **Ассоциации.** Это сущности, представляющие *связь типа "многие ко многим"* (или "многие ко многим, ко многим" и т.д.) между двумя или более сущностями. Участвующие в ассоциации сущности могут представлять собой ядра, характеристики или соединения.

Кроме того, необходимо отметить следующее.

- Сущности (независимо от их классификации) могут также иметьсвойства.
- В частности, любая сущность (опять-таки, независимо от ее классификации) может иметь свойство, предназначенное для **указания** на некоторую другую сущность. Такое *указание* представляет связь типа "многие к одному", имеющуюся между двумя сущностями.

Примечание. В исходной статье с определением RM/T-модели такие указания не рассматривались, а были предложены позже.

ИМ Поддерживаются **супертипы** и **подтипы** сущностей. Если сущность Y является подтипом сущности X, то сущность У является ядром, характеристикой или ассоциацией в зависимости от того, чем является сущность X: ядром, характеристикой или ассоциацией. Все перечисленные выше концепции можно соотнести с их аналогами в ER-модели следующим образом (несколько неформально). Ядро соответствует обычной сущности ER-модели, характеристика — слабой сущности, а ассоциация — связи (только для типа "многие ко многим"). Приведенные выше краткие сведения следует дополнить упоминанием о том, что в модели RM/T предусмотрена поддержка замещающих ключей (подробности приводятся в [14.21]), *временной размерности* (см. главу 23) и различных типов *операций агрегирования данных* (подробности приводятся в [14.40], [14.41]).

14.8. Date C.J. A Note on One-to-One Relationships // Relational Database Writings: 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

Это расширенное рассмотрение проблемы связей типа "один к одному", которая оказалась более сложной, чем может показаться на первый взгляд.

14.9. Date C.J. Entity/Relationship Modeling and the Relational Model // C. J. Date and Hugh Darwen. Relational Database Writings: 1989-1991.— Reading, Mass.: Addison-Wesley, 1992.

14.10. Date C.J. Don't Encode Information into Primary Keys! // C. J. Date and Hugh Darwen. Relational Database Writings: 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.
Здесь представлены неформальные доводы против использования того, что иногда называют "интеллектуальными ключами". К тому же следует упомянуть работу [14.11], в которой приведены рекомендации относительно внешних ключей.

14.11. Date C.J. Composite Keys // C. J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.
Цитата из аннотации к этой работе: "Собраны и проанализированы доводы *за* и *против* включения составных [ключей] в проект реляционной базы данных... и предложены рекомендации". В частности, в статье показано, что замещающие ключи [14.21] не всегда рекомендуется использовать.

14.12. Date C.J. A Database Design Dilemma?//http: //www. dpbd. com (January 1999).
На первый взгляд заданный тип сущности (например, "работник") может быть представлен в реляционной системе либо с помощью типа EMPLOYEE (т.е. доме на), либо с помощью переменной отношения EMPLOYEE. В этой короткой статье (которая основана на материале приложения С работы [3.3]) даются рекоменда ции по выбору одного из двух вариантов.

14.13. Date C.J. Subtables and Supertables // Приложение Е в [3.3].
Часто считается, что наследование типов сущностей в реляционном контексте может быть организовано с помощью так называемых *подтаблиц* и *супертаблиц,* где сущность-подтип отображается в подтаблицу, а сущность-супертип — в супер-таблицу. В частности, такой подход поддерживается в стандарте SQL (см. главу 26), а также в некоторых существующих продуктах. Но в [14.13] приведены весо-мые доводы против этой идеи.

14.14. Date C.J. Twelve Rules for Business Rules // http: //www.versata.com (May 1, 2000).
В этой статье предложено множество правил (или предписаний), которым должна соответствовать "качественная" система бизнес-правил.

14.15. Date C.J. Models, Models, Everywhere, Nor Any Time to Think // http: //www. dbdebunk. com (November 2000).

Термин *модель* в мире информационной технологии, особенно в сообществе поль-зователей баз данных, чрезвычайно перегружен (если не сказать большего). Эта статья написана в качестве предупреждения о том, какие наихудшие последствия могут быть с этим связаны, и в ней предпринята попытка показать, что следует (по меньшей мере!) дважды подумать, прежде чем использовать этот термин не по на-значению.

14.16. Date C.J. Basic Concepts in UML: A Request for Clarification // http://www.dbdebunk. com (December 2000/January 2001).

Эта статья, состоящая из двух частей, содержит результаты исследования и серьезного критического анализа универсального языка моделирования (Unified Modeling Language — UML). В ней особое внимание уделено языку объектных ограничений (Object Constraint Language — OCL). В части I в основном рассматривается "библия OCL" [14.49], а в части II - "библия UML" [14.5].

14.17. Dey D., Storey V.C., Barron T.M. Improving Database Design Through the Analysis of Relationships // ACM TODS. — December 1999. — 24, № 4.

14.18. Elmasri R., Navathe S.B. Fundamentals of Database Systems (3rd edition). — Redwood City, Calif.: Benjamin/Cummings, 2000.

В этом учебном пособии по управлению базами данных две главы посвящены использованию ER-методов для проектирования баз данных.

14.19. Embley D.W. Object Database Development: Concepts and Principles. — Reading, Mass.: Addison-Wesley, 1998.

В этой книге представлена методика проектирования на основе OSM-модели (Object-oriented Systems Model). Некоторые элементы OSM-модели похожи на ORM-модель **[14.22]—[14.24].**

14.20. Fleming C.C., Von Halle B. Handbook of Relational Database Design. — Reading, Mass.: Addison-Wesley, 1989.

Прекрасное практическое пособие по проектированию баз данных в реляционных системах с конкретными примерами реализации в среде СУБД DB2 компании IBM и DBC/1012 компании Teradata (теперь NCR). Описаны этапы как логического, так и физического проектирования, но следует предупредить читателя, что в этой книге термин "логическое проектирование" используется для описания того, что мы называем реляционным проектированием, а термин "реляционное проектирование" включает, по крайней мере, несколько аспектов того, что мы называем физическим проектированием!

14.21. Hall P., Owlett J., Todd S.J.P. Relations and Entities // G. M. Nijssen (ed.). Modelling in Data Base Management Systems. — Amsterdam, Netherlands: North-Holland; New York, N.Y.: Elsevier Science, 1975.

Это первая статья, в которой подробно рассмотрена идея замещающих ключей (позднее они были введены в состав модели RM/T [14.7]). *Замещающим* называется ключ в обычном реляционном смысле, но дополнительно обладающий перечисленными ниже специфическими свойствами.

- Он всегда состоит только из одного атрибута.
- Значениями этого атрибута являются *только* замещающие ключи (отсюда и их название), выбранные для ссылки на сущности, которые они обозначают. Иначе говоря, эти значения служат лишь для представления того факта, что соответствующие сущности действительно определены, но не несут никакой другой информации и никакой другой смысловой нагрузки.

- При вставке в базу данных новой сущности присваивается значение замещающего ключа, которое никогда прежде не использовалось, и никогда не будет использоваться, даже если данная сущность впоследствии будет удалена.

В идеале, значения замещающего ключа должны генерироваться системой, но сам способ их генерации (системный или пользовательский) не имеет никакого отношения к основной идее замещающих ключей.

Здесь, вероятно, стоит заметить, что замещающие ключи *не* являются (как полагают некоторые авторы) тем же, что и "идентификаторы кортежей" (или идентификаторы строк), поскольку последние идентифицируют кортежи, а замещающие ключи идентифицируют сущности, притом что никакой прямой связи типа "один к одному" между кортежами и сущностями не существует. (В частности, достаточно представить себе идентификаторы производных кортежей, возникших в результате выполнения некоторого произвольного запроса. В действительности, даже не совсем ясно, должны ли производные кортежи вообще иметь идентификаторы кортежей.) Более того, идентификаторы кортежей обладают дополнительными преимуществами в отношении производительности, чего не скажешь о замещающих ключах, поскольку доступ к кортежу через идентификатор кортежа происходит очень быстро (предполагается, что кортежи базового отношения отображаются непосредственно на физическую структуру хранения, а это действительно имеет место во многих современных программных продуктах). Кроме того, идентификаторы кортежей обычно скрыты от пользователя, тогда как замещающие ключи чаще всего — нет (согласно *информационному принципу),* т.е. идентификатор кортежа нельзя сохранить, как значение атрибута, а замещающий ключ— можно.

Короче говоря, замещающие ключи относятся к уровню модели, а идентификаторы кортежей — к уровню реализации.

14.22. Halpin T. Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design. San Francisco, Calif.: Morgan Kaufmann, 2001.

Подробное описание ORM-модели (см. аннотации к двум следующим работам). Среди многих других тем в этой книге обсуждаются связи между ORM- и ER-моделированием, с одной стороны, и языками ORM и UML, с другой.

Примечание. Эта книга представляет собой существенно пересмотренную и дополненную версию предыдущей книги того же автора, *Conceptual Schema and Relational Database Design* (2-е издание), опубликованной издательством Prentice Hall of Australia Pty., Ltd., 1995.

14.23. Halpin T. Business Roles and Object-Role Modeling // DBP&D. - October 1996. - 9, №10.

Великолепное введение в **объектно-ролевое моделирование** (Object-Role Modeling — ORM) [14.22]. Автор вначале приводит свое замечание о том, что "[в отличие от] ER-моделирования, для которого существуют десятки разных диалектов, ORM-моделирование имеет лишь несколько диалектов с незначительными различиями". *(Примечание.* Один из таких диалектов— NIAM-моделирование [14.34].) ORM-моделирование называется также *моделированием на основе фактов,* поскольку проектировщик начинает процесс моделирования с записи (на естественном языке

или с помощью специальных графических обозначений) ряда *элементарных* фактов (или, точнее, *типов* фактов), которые в совокупности характеризуют все особенности моделируемой организации. Ниже приводятся некоторые примеры таких типов фактов.

- Каждый работник имеет не больше одного имени.
- Каждый работник отчитывается о своей деятельности не больше чем перед одним работником.
- Если работник el отчитывается перед работником e2, то обратное (т.е. работ ник e2 отчитывается перед работником el) невозможно.
- Ни один работник не может одновременно руководить проектом и оценивать результаты его выполнения.

Очевидно, что типы фактов на самом деле представляют собой внешние *предикаты* или бизнес-правила. Как следует из названия этой статьи, подход к проектированию базы данных в ORM-моделировании очень близок к тем методам, которые предпочитают сторонники использования бизнес-правил и сам автор. В общем случае факты представляют собой *роли,* выполняемые *объектами* в связях (отсюда и название "объектно-ролевое моделирование"). Обратите внимание на то, что *объекты* в данном контексте представляют собой сущности, а не объекты в конкретном смысле этого термина, описанного в части VI данной книги, а связи между ними не обязательно являются двухсторонними. Однако факты действительно *элементарны,* т.е. их декомпозиция на меньшие факты невозможна.

Примечание. Основная идея о том, что база данных на концептуальном уровне должна содержать только элементарные (или неприводимые) факты, принадлежит Холлу (Holl), Оулетту (Owlett) и Тодцу (Todd) [14.21].

Заслуживает внимание то, что в ORM-моделировании нет понятия *атрибут*. В результате, как показано в этой статье, ORM-проекты концептуально проще и устойчивее, чем их аналоги, выполненные по методу RM-моделирования (в этой связи см. также аннотацию к [14.24]). Но атрибуты могут и должны появляться в проектах на основе ER-модели или определениях SQL, которые вырабатываются (автоматически) на основании проекта ORM.

В ORM-моделировании также особо подчеркивается использование *фактов-образцов* (т.е. *экземпляров* фактов-образцов, которые иначе можно было бы назвать *высказываниями)* в качестве способа проверки корректности проекта со стороны конечного пользователя. Как утверждается в данной статье, такой подход достаточно просто реализовать с помощью моделирования на основе фактов, но гораздо труднее осуществить при работе с ER-моделью.

Безусловно, существует много логически эквивалентных способов описания одного и того же предприятия и, соответственно, много логически эквивалентных ORM-схем. По этой причине ORM-моделирование включает набор *правил преобразования,* позволяющих выполнять преобразования с получением логически эквивалентных схем. Поэтому ORM-инструменты позволяют выполнять некоторую оптимизацию проектов в соответствии с требованиями разработчика. Как уже говорилось выше, с их помощью можно также вырабатывать схему в виде

ER-модели или определения SQL на основании схемы ORM, а также выполнять обратную процедуру восстановления схемы ORM из существующей схемы ER или SQL. В зависимости от используемой СУБД, созданная схема SQL может включать декларативные ограничения в стиле SQL; другой вариант может предусматривать реализацию таких ограничений с помощью триггеров или хранимых процедур. В отношении ограничений отметим, что ORM-модель, в отличие от ER-модели, *по определению* содержит "развитый язык для определения ограничений". (Однако в [14.24] автор признает, что не все бизнес-правила можно выразить с помощью принятых в ORM-модели *графических* обозначений. Для некоторых из них все же придется использовать текстовые формулировки.)

Наконец, ORM-модель, безусловно, может рассматриваться как высокоуровневое абстрактное представление базы данных (на самом деле следовало бы возразить, что оно, скорее, ближе к чистому, возможно, даже в некоторой степени строгому, *реляционному* представлению). Как таковая, эта модель может служить в качестве основы для непосредственного составления запросов. См. аннотацию к [14.24].

14.24. Halpin T. Conceptual Queries // Data Base Newsletter. — March-April 1998. — 26, № 2.

Цитата из аннотации к этой работе: "Формулировка нетривиальных запросов с помощью таких реляционных языков программирования, как SQL и QBE, может оказаться очень сложной задачей для конечных пользователей. *ConQuer*— это новый концептуальный язык создания запросов, который основан на объектно-ролевом моделировании и предоставляет простой и понятный пользователям способ составления запросов... В этой статье описаны преимущества [такого языка] по сравнению с традиционными языками запросов в отношении определения запросов и бизнес-правил".

Помимо всего прочего, в статье обсуждается приведенный ниже запрос на языке ConQuer, смысл которого состоит в следующем: "Извлечь данные о работниках, которые водят машину, и об отделах, в которых они работают".

Работник + -водит Машину + -работает в Отделе

Если сотрудник в общем случае умеет управлять произвольным количеством автомобилей, но имеет право работать только в одном отделе, то соответствующий проект SQL может состоять из двух таблиц, а код SQL— иметь следующий вид.

```
SELECT     DISTINCT     X1.EMPt,
X1.BRANCH» FROM EMPLOYEE AS X1,
DRIVES AS X2 WHERE   X1.EMP# =
X2.EMP# ;
```

Теперь предположим, что служащий получил возможность работать одновременно в нескольких отделах. Тогда соответствующий проект SQL должен включать не две, а три таблицы, а соответствующий код SQL будет выглядеть следующим образом.

```
SELECT DISTINCT X1.EMP#, X3.BRANCH#
FROM   EMPLOYEE AS X1, DRIVES AS X2, WORKS FOR AS X3
WHERE  X1.EMP# = X2.EMP# AND X1.EMP# = X3.EMP# ;
```

Однако формулировка на языке ConQuer никаких изменений не потребует.

Как видно из предыдущего примера, язык наподобие ConQuer может рассматриваться как весьма строгий вариант реализации логической независимости от данных. Для объяснения этого замечания придется несколько уточнить архитектуру ANSI/SPARC. Как говорилось в главе 2, логическая независимость от данных означает независимость от изменений концептуальной схемы, но дело в том, что в предыдущем примере не было сделано никаких изменений концептуальной схемы! Причина в том, что современные продукты SQL не поддерживают работу концептуальной схемой должным образом. Вместо этого они, скорее всего, поддерживают работу *со схемой SQL.* При этом схема SQL может рассматриваться как находящаяся на промежуточном уровне между истинным концептуальным уровнем и внутренним или физическим уровнем. Если ORM-инструмент позволяет определить истинную концептуальную схему и отобразить ее на схему SQL, то язык ConQuer может обеспечить требуемую независимость от изменений в схеме SQL (безусловно, за счет соответствующих изменений этого отображения).

Из статьи не совсем ясно, чем ограничивается выразительная мощь языка ConQuer. Автор статьи не дает прямого ответа на поставленный вопрос, но отмечает (и это достойно сожаления), что "этот язык позволяет идеально формулировать любые вопросы, которые имеют какое-то отношение к данному приложению, но на практике не всегда следует стремиться к этому идеалу" (эти слова автора немного перефразированы). К тому же автор заявляет, что "наиболее мощной функциональной возможностью языка ConQuer... является его способность выполнять корреляции с произвольным уровнем сложности", и приводит следующий пример.

```
^Работник1
   +-живет в Городе1
   +-родился в Стране1
   +-руководит работой Работника2, который
      +-живет в Городе1
      +-родился в Стране2 <> Стране1
```

Этот запрос означает: "Извлечь данные о сотрудниках, руководящих работой других сотрудников, которые живут в том же городе, что и их руководитель, но родились в другой стране". Автор предлагает попытаться сформулировать этот запрос на языке SQL!

Наконец, в отношении языка ConQuer и бизнес-правил автор говорит: "Хотя графические обозначения ORM-модели могут выразить больше бизнес-правил, чем [ER-модель], их все еще приходится дополнять текстовым описанием [для выражения некоторых ограничений]. В настоящее время продолжаются исследования, проводимые в направлении адаптации языка ConQuer для достижения указанной цели".

14.25. Hammer M.M., McLeod D. J. The Semantic Data Model: A Modelling Mechanism for Database Applications // Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data. — Austin, Texas. — May/June 1978.

Семантическая модель данных (Semantic Data Model — SDM) представляет собой другой формальный подход к проектированию базы данных. Как и в случае ER-моделирования, в этой модели основное внимание уделяется структурным аспектам и (до определенной степени) аспектам целостности и совсем немного внимания

(или вообще никакого) — аспектам манипулирования данными. Об этом также можно прочесть в [14.26] и [14.29].

14.26. Hammer M., McLeod D. Database Description with SDM: A Semantic Database Model //ACM TODS. - September 1981. - 6, № 3.

См. аннотацию к [14.25].

14.27. Hull R., King R. Semantic Database Modeling: Survey, Applications, and Research Issues // ACM Сотр. Surv. - September 1987. - 19, № 3.

Исчерпывающее учебное пособие по семантическому моделированию и относящейся к нему проблематике по состоянию на конец 1980 годов. Эта статья является прекрасной отправной точкой для изучения семантического моделирования и связанных с ним проблем. См. также [14.36].

14.28. Jacobson I., Christerson M., Jonsson P., Overgaard G. Object-Oriented Software Engineering (пересмотренное издание). — Reading. Mass.: Addison-Wesley, 1994.

В этой работе описывается метод проектирования Object-Oriented Software Engineering (OOSE). Так же, как и при использовании OMT-модели [14.3], компоненты OOSE (по крайней мере, имеющие отношение к базам данных) могут рассматриваться как вариант ER-модели (как и в случае с OMT-моделью, *OOSE-объекты* соответствуют *ER-сущностям).* Приведем следующую цитату: "Большинство используемых сегодня методов разработки как информационных, так и технических систем основано на функциональной декомпозиции этой системы и/или декомпозиции, управляемой данными. Такие подходы сильно отличаются от объектно-ориентированных методов, в которых данные и функции тесно связаны". Очевидно, что здесь автор указывает на существенное различие между объектным подходом и подходом, принятым при проектировании баз данных. *Предполагается,* что базы данных, по крайней мере, совместно используемые базы данных общего назначения, которым уделяется больше всего внимания со стороны специалистов в области баз данных, отделены от "функций", т.е. *предполагается,* что базы данных проектируются независимо от приложений, в которых они будут использоваться. Тем не менее, мы считаем, что специалисты в области объектно-ориентированного подхода употребляют термин "база данных" в непосредственной связи с *конкретным приложением,* а не как совместно используемую базу данных общего назначения (в этой связи см. обсуждение объектных баз данных в главе 25).

См. также аннотацию к [14.5], [14.16] и [14.37].

14.29. Jagannathan D. et al. SIM: A Database System Based on the Semantic Data Model // Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data. — Chicago, 111. — June 1988.

Описание коммерческой СУБД, которая построена на основе семантической модели данных, аналогичной семантической модели данных, предложенной Хаммером и Маклеодом в [14.25].

14.30. Keuffel W. Battle of the Modeling Techniques: A Look at the Three Most Popular Mo deling Notations for Distilling the Essence of Data // DBMS. — August 1996. — 9, № 9.

"Тремя наиболее популярными системами обозначения являются" ER-модель, метод NIAM (Natural-language Information Analysis Method — метод анализа информации на естественном языке) [14.34] и метод SOM (Semantic Object Modeling — семантическое объектное моделирование). Автор утверждает, что ER-модель является "предшественником" двух других моделей, но одновременно подвергает ее критике в связи с отсутствием необходимого формального обоснования. Например, по его словам, сущности, связи и атрибуты (т.е. свойства) "описываются без упоминания о том, как они были открыты". NIAM-метод является гораздо более строгим. Если строго следовать правилам этой модели, можно получить концептуальный проект, который "обладает гораздо большей целостностью", чем проекты, полученные на основе других методологий, хотя "некоторые разработчики могут счесть строгость NIAM-модели чрезмерно ограничивающей" (!). Что касается SOM-модели, то она "напоминает ER-модель... из-за [аналогичной] нестрогой формулировки определений сущностей, атрибутов и связей". Но она отличается от ER-модели тем, что в ней поддерживаются *групповые атрибуты* (т.е. на самом деле повторяющиеся группы), позволяющие одному "объекту" (т.е. сущности) содержать другие. (В ER-модели сущность может содержать повторяющиеся группы *атрибутов,* но не других *сущностей.)*

14.31. Mannila H., Raiha K.-J. The Design of Relational Databases // Reading, Mass.: Addison-Wesley, 1992.

Согласно предисловию, эта книга является "учебным пособием для студентов и справочником по проектированию реляционных баз данных". В ней, с одной стороны, описаны теория зависимостей и процедура нормализации, а с другой — ER-модель, причем в каждом случае изложение ведется с формальной точки зрения. Ниже приведен (неполный список) названий глав из этой книги, которые могут дать представление о ее содержании.

- Принципы проектирования.
- Ограничения целостности и зависимости.
- Свойства реляционных схем.
- Аксиоматизация зависимостей.
- Алгоритмы решения задач проектирования.
- Соответствия между ER-диаграммами и схемами реляционных баз данных.
- Преобразования схем.
- Применение типовых баз данных в проектах.

Описанные в книге технологии были воплощены ее авторами в виде коммерчески распространяемого инструмента проектирования *Design By Example* (проектирование по образцу).

14.32. Moriarty T. Enterprise View (постоянная колонка в журнале) // DBP&D. — August 1997. - 10, № 8.

В этой работе описан коммерческий инструмент проектирования и разработки приложений Usoft (www.usoft.com), который позволяет определять бизнес-

правила с помощью синтаксиса, подобного SQL, а затем использовать их для генерации приложения (включая создание определения базы данных).

14.33. Nijssen G.M., Duke D.J., Twine S.M. The Entity-Relationship Data Model Considered Harmful // Proc. 6th Symposium on Empirical Foundations of Information and Software Sciences. — Atlanta, Ga., 1988.

"Может ли использование ER-модели привести к пагубным последствиям?" Мы можем дать несколько подтверждений положительного ответа на этот вопрос, включая приведенные ниже.

- Неправильное понимание различий между типами и переменными отношения (см. обсуждение первого серьезного заблуждения в главе 26).
- Странные манипуляции с подтаблицами и супертаблицами (см. [14.13], атакже главу 26).
- Широко распространенное нежелание признавать *принцип относительности базы данных* (см. главу 10).
- . Широко распространенное мнение о том, что между сущностями и связями есть или должно быть различие (о чем было сказано в настоящей главе).

В [14.33] приведены факты, которые еще больше усугубляют описанную выше мрачную картину. В ней утверждается, что перечень недостатков ER-модели включает перечисленные ниже недостатки.

- Модель предлагает несколько перекрывающихся способов представления структур данных, что чрезмерно усложняет процесс проектирования.
- Нет никаких критериев выбора одного из нескольких альтернативных представлений, что на практике может иметь весьма нежелательное следствие — необходимость внесения изменений в уже завершенный проект при изменении тех или иных обстоятельств.
- Имеется слишком мало способов представления требований поддержки цело стности данных, что делает неосуществимыми некоторые аспекты процесса проектирования ("[действительно] ограничения могут быть формально выра жены с помощью более общих систем обозначения, например таких, как логика предикатов. [Однако] полагать, что это есть обоснованный предлог для исклю чения [ограничений] из модели данных, — все равно что полагать, что некий язык программирования очень хорош, [даже несмотря на то, что] для реализа ции всех тех функций, которые нельзя прямо выразить с помощью данного языка, вам предлагают воспользоваться процедурами на языке ассемблера!").
- Вопреки популярному мнению, ER-модель не является хорошим средством общения конечных пользователей и профессионалов в области баз данных.
- ER-модель нарушает принцип концептуализации: "Концептуальная схема долж на... включать [только] концептуально приемлемые аспекты предметной об ласти, как статические, так и динамические, поэтому исключать все аспекты (внешнего и внутреннего) представления данных, физической организации данных и доступа к ним, [а также] все аспекты конкретного представления внешнего пользователя, например форматы сообщений, структуры данных и

т.д." [2.3]. Действительно, авторы этой работы считают, что ER-модель "по сути представляет собой другое воплощение" старой сетевой модели CODASYL. "Можно ли считать основной причиной широкого распространения ER-модели в сообществе специалистов в области [баз данных] то, что в ней особое внимание уделяется структурам реализации?"

В этой статье также указаны многочисленные более мелкие недостатки ER-модели. Затем в качестве возможного перспективного варианта в ней предлагается альтернативная методология NIAM [14.34]. В частности, в статье подчеркивается, что в NIAM нет излишнего разделения на атрибуты и связи, которое характерно для ER-модели.

14.34. Olle T.W., Sol H.G., Verrijn-Stuart A.A. (eds.). Information Systems Design Methodologies: A Comparative Review. — Amsterdam, Netherlands: North-Holland; New York, N.Y.: Elsevier Science, 1982.

Доклады конференции IFIP Working Group 8.1, в которых описаны 13 различных методик и результаты их применения для решения стандартной тестовой задачи. Среди прочих рассматривается и методика NIAM [14.33]. Эта работа, вероятно, была одной из первых работ по методике NIAM. В книгу включены также обзоры некоторых предложенных подходов, включая NIAM.

14.35. Papazoglou M.P. Unraveling the Semantics of Conceptual Schemas // CACM. — September 1995. - 38, № 9.

В этой работе предлагается подход на основе того, что можно назвать *запросами к метаданным,* т.е. на основе запросов, которые обращены к смыслу данных (а не к значениям данных в базе данных), или, иначе говоря, запросов к самой концептуальной схеме. Примером такого запроса является следующий запрос: "Что такое постоянный работник?".

14.36. Peckham J., Maryanski F. Semantic Data Models // ACM Comp. Surv. — September, 1988.-20, №3.

Еще один вводный обзор (см. также [14.27]).

14.37. Reed P. Unified Modeling Language Takes Shape // DBMS. - July 1998. — 11, № 8.

Универсальный язык моделирования UML (Unified Modeling Language) представляет собой еще одну графическую систему обозначений, предназначенную для проектирования и разработки приложений (иначе говоря, она позволяет разрабатывать приложения с помощью диаграмм, по меньшей мере, частично). Она также может использоваться для разработки схем SQL.

Примечание. В то время, когда был впервые создан язык UML, предполагалось, что он вскоре приобретет большое коммерческое значение, отчасти благодаря тому, что он был принят в качестве стандарта группой OMG (Object Management Group) (к тому же этот язык в целом характеризуется заметной объектной ориентацией). Но теперь создается впечатление, что степень распространения этого языка намного меньше по сравнению с той, которая предполагалась в самом начале, хотя и нельзя отрицать, что он поддерживается во многих коммерческих программных продуктах.

Как бы то ни было, язык UML обеспечивает моделирование данных и процессов (в этом отношении он превосходит по своим возможностям ER-моделирование), но ограничениям целостности в нем не уделяется достаточного внимания. (В разделе работы [14.37] с названием "От моделей к коду— бизнес-правила" термин *декларативный* вообще не упоминается! Основное внимание, скорее, сконцентрировано на генерации *процедурного кода приложения* для реализации "процессов". Вот цитата из этой работы: "UML формализует то, что уже давно знали практики в области объектного подхода — объекты реального мира лучше всего моделировать с помощью самодостаточных сущностей, которые содержат и данные, и функциональные средства". А в другом месте этой работы отмечено: "С исторической точки зрения очевидно, что формальное разделение данных и функций привело к тому, что усилия по созданию программного обеспечения имели преходящую ценность". Эти замечания могут быть верны в отношении приложения, но совсем не очевидно, что это верно в отношении баз данных, например, см. [25.25].)

Язык UML появился на основе ранней работы Буча с описанием метода Буча *(Booch method)* [14.4], работы Румбау с описанием OMT-модели [14.3] и работы Якобсона с описанием OOSE-метода [14.28]. См. также [14.5] и [14.16].

14.38. Schmid H. A., Swenson J. R. On the Semantics of the Relational Data Base Model // Proc. 1975 ACM SIGMOD Int. Conf. on Management of Data. — San Jose, Calif. - May 1975.

В этой работе предложена "базовая семантическая модель", которая предшествовала работе Чена с описанием ER-модели [14.6], но была во многом подобна этой модели (безусловно, за исключением используемой терминологии, поскольку Шмид и Свенсон применяли термины *независимый объект, зависимый объект* и *ассоциация* вместо терминов Чена *обычная сущность, слабая сущность* и *связь,* соответственно).

14.39. Sowa J.F. Conceptual Structures: Information Processing in Mind and Machine.— Reading, Mass.: Addison-Wesley, 1984.

Эта книга посвящена не самим системам баз данных, а скорее общей проблеме представления и обработки знаний. Но некоторые ее части непосредственно относятся к теме настоящей главы. (Последующие замечания основаны на докладе автора рассматриваемой книги, сделанном им в 1990 году по поводу применения "концептуальных структур" для семантического моделирования.) Основная проблема использования ER-диаграмм (и связанных с ней формальных методов) заключается в том, что они являются строго менее мощными, чем *формальная логика.* В результате они не могут справиться с некоторыми важными аспектами проектирования (например, не допускают явного использования кванторов, входящих в состав большинства ограничений целостности), с которыми *может* справиться формальная логика. (Кванторы были предложены Фреге (Frege) в 1879 году, что позволяет утверждать, что ER-диаграммы представляют "разновидность логики по состоянию до 1879 года"!) Но формальная логика гораздо сложнее воспринимается при чтении. Как утверждает автор рассматриваемой книги, "исчисление предикатов является языком ассемблера для представления знаний". *Концептуальные графы* — это достаточно удобные для чтения и строгие графические

обозначения, представляющие всю логику в целом. Следовательно (согласно утверждениям этого автора), они гораздо больше подходят для семантического моделирования, чем ER-диаграммы и им подобные модели.

14.40. Smith J.M., Smith D.C.P. Database Abstractions: Aggregation // CACM. — June 1977. - 20, № 6.

См. аннотацию к [14.41].

14.41. Smith J.M., Smith D.C.P. Database Abstractions: Aggregation and Generalization // ACM TODS. - June 1977. - 2, № 2.

Идеи, высказанные в [14.40] и [14.41], оказали значительное влияние на формулировку положений RM/T-модели [14.7], особенно в области определения подтипов и супертипов сущностей.

14.42. Storey V.C. Understanding Semantic Relationships // The, VLDB Journal. — October 1993. - 2, № 4.

В аннотации к настоящей статье говорится: "Семантические модели данных были разработаны сообществом исследователей баз данных с использованием таких абстракций, как подтип, агрегирование и ассоциация. Помимо этих хорошо известных понятий, дополнительные семантические понятия были введены исследователями в таких дисциплинах, как лингвистика, логика и психология познания. В статье исследуются некоторые из этих дополнительных понятий и обсуждается их влияние на проектирование баз данных".

14.43. Sundgren B. The Infological Approach to Data Bases // J. W. Klimbie and K. L. Koffeman (eds.). Data Base Management. — Amsterdam, Netherlands: North-Holland; New York, N.Y.: Elsevier Science, 1974.

Информационно-логическим (infological) называется один из ранних подходов к семантическому моделированию, который долгие годы успешно использовался для проектирования баз данных в Скандинавии.

14.44. Tasker D. Fourth Generation Data: A Guide to Data Analysis for New and Old Systems. — Sydney, Australia: Prentice-Hall of Australia Pty., Ltd., 1989.

Прекрасное практическое пособие по проектированию баз данных, в котором внимание уделяется главным образом отдельным элементам данных (по сути, речь идет о типах). Элементы данных подразделяются на три основные разновидности: именные, количественные и описательные. *Именные элементы* определяют сущности и в реляционном смысле соответствуют ключам. *Количественные элементы* представляют количество, величину или позицию на некоторой шкале (возможно, на шкале времени) и могут участвовать в обычных арифметических операциях. Все остальные элементы данных относятся к *описательным*. (Безусловно, это краткое описание не может дать полного представления обо всей классификационной схеме.) В книге подробно обсуждается каждый из перечисленных типов элементов данных. Эти описания не всегда можно назвать "реляционно чистыми", поскольку, например, использованное автором понятие "домена" не вполне отвечает реляционному смыслу этого термина. Однако в книге содержится достаточно материала, имеющего большое практическое значение.

14.45. Teorey T.J., Fry J.P. Design of Database Structures.— Englewood Cliffs, N.J.: Prentice-Hall, 1982.

Это учебник по всем вопросам проектирования баз данных, который разделен на пять частей: введение, концептуальное проектирование, практическое проектирование (т.е. преобразование концептуального проектирования в конструкции, которые можно применить для конкретной СУБД), физическое проектирование и вопросы специализированного проектирования.

14.46. Teorey T.J., Yang D., Fry J.P. A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model // ACM Сотр. Surv. — June 1986. — 18, №2.

В представленную в этой работе "расширенную ER-модель" добавлена поддержка иерархий типов сущностей, пустых значений (глава 19) и связей, включающих больше двух участников.

14.47. Teorey T.J. Database Modeling and Design: The Entity-Relationship Approach (3-е из дание) // San Mateo, Calif.: Morgan Kaufmann, 1998.

Более современный учебник с описанием применения концепций ER-модели и "расширенной" ER-модели [14.46] для проектирования базы данных.

14.48. Wand Y., Storey V.C., Weber R. An Ontological Analysis of the Relationship Construct in Conceptual Modeling // ACM TODS. — December 1999. — 24, № 4.

14.49. Warmer J., Kleppe A. The Object Constraint Language: Precise Modeling with UML. — Reading, Mass.: Addison-Wesley, 1999.

См. аннотацию к [14.16].

ЧАСТЬ IV

УПРАВЛЕНИЕ ТРАНЗАКЦИЯМИ

Эта часть книги состоит из двух глав, которые посвящены тесно связанным темам: восстановлению и параллельности. Обе они являются аспектами более общей темы управления транзакциями, но по методическим соображениям их желательно рассматривать отдельно.

Восстановление и параллельность, или, точнее, *управление* восстановлением и параллельностью, связаны с общим понятием **защиты данных,** т.е. предохранения данных от утраты или повреждения. В частности, риск потери данных обусловлен описанными ниже причинами.

- Во время выполнения некоторых программ может произойти аварийный останов системы, в результате чего база данных может оказаться в непредсказуемом со стоянии.
- При одновременном ("параллельном") выполнении возможен конфликт между двумя программами из-за обращения к одним и тем же данным, что приведет к получению неправильных результатов как внутри базы данных, так и на выходе из нее.

В главе 15 рассматривается восстановление, а в главе 16 — параллельность.

ГЛАВА 15

Восстановление

15.1. ВВЕДЕНИЕ

Как уже говорилось во введении к настоящей части, эта и следующая главы посвящены восстановлению и параллельности. Данные понятия очень тесно связаны и составляют части более общей темы *управления транзакциями.* Но с методической точки зрения желательно рассматривать эти темы по возможности отдельно, поэтому данная глава в основном посвящена теме восстановления, а параллельность будет рассмотрена в главе 16 (однако вопросы параллельности время от времени рассматриваются и в данной главе, особенно в разделе 15.4).

Восстановление в системе баз данных означает, в первую очередь, восстановление самой базы данных, т.е. возвращение базы данных в определенное состояние, которое считается правильным, после некоторого сбоя, в результате которого текущее состояние становится неправильным или, по крайней мере, достаточно неопределенным (понятие правильного состояния базы данных будет точно сформулировано в следующем разделе). Основной принцип, на котором строится подобное восстановление, достаточно прост и может быть выражен одним словом — **избыточность.** (Эта избыточность предусматривается на физическом уровне. По причинам, подробно описанным в других главах данной книги, такую избыточность, естественно, не следует предусматривать на логическом

уровне.) Иначе говоря, убедиться в том, что база данных действительно может быть восстановлена, можно, получив гарантии, что любая часть содержащейся в базе данных информации может быть реконструирована из другой информации, избыточно сохраняемой где-то в системе.

Прежде чем перейти к дальнейшему изложению, необходимо подчеркнуть, что принципы восстановления (а в действительности, и принципы обработки транзакций в целом) в значительной степени не зависят от того, какой является базовая система: реляционной или какой-либо еще. (С другой стороны, исторически сложилась такая ситуация, что большая часть теоретических исследований в области обработки транзакций была выполнена и продолжает выполняться именно в реляционном контексте.) Нужно также отметить, что это весьма обширная область, и мы сможем познакомить читателя только с наиболее важными и основополагающими принципами. Для более углубленного изучения этой темы можно обратиться к источникам, указанным в списке литературы в конце данной главы (в частности, заслуживает особого внимания [15.12]).

План этой главы выглядит следующим образом. После короткого введения в разделах 15.2 и 15.3 описываются фундаментальные понятия *транзакции* и *восстановления транзакции* (т.е. восстановления базы данных после неудачного завершения какой-либо транзакции). Затем в разделе 15.4 более глубоко обсуждается проблема восстановления *системы* (после одновременного нарушения процессов выполнения всех текущих транзакций, вызванного аварийным остановом системы). После этого в разделе 15.5 кратко рассматривается восстановление *носителей* (после какого-либо физического отказа внешнего устройства, на котором хранится база данных, например, из-за поломки головок дискового накопителя). Далее в разделе 15.6 описывается исключительно важная проблема *двухфазной фиксации транзакций,* а в разделе 15.7 рассматриваются точки сохранения. В разделе 15.8 описаны соответствующие средства языка SQL. Наконец, в разделе 15.9 приводятся краткое резюме и несколько заключительных замечаний.

Еще одно, последнее предварительное замечание: во всей этой главе предполагается, что речь идет о "крупной" (разделяемой, многопользовательской) среде базы данных. В "небольших" (неразделяемых, однопользовательских) СУБД поддержка восстановления обычно развита слабо или вообще не предусмотрена; вместо этого задача восстановления в такой системе, как правило, возлагается на самого пользователя. Под этим подразумевается, что пользователь должен периодически создавать резервные копии базы данных, а в случае отказа повторно вносить необходимые изменения вручную.

15.2. ТРАНЗАКЦИИ

Транзакция — это логическая единица работы; она начинается с выполнения операции BEGIN TRANSACTION и заканчивается операцией COMMIT или ROLLBACK. На рис. 15.1 показан псевдокод транзакции, которая предназначена для перечисления суммы 100 долл. со счета 123 на счет 456. Вполне очевидно, что операция перевода денег с одного счета на другой, которая по самой своей сути является неразрывной, фактически требует выполнения в базе данных двух отдельных операций обновления. Более того, сама база данных на этапе между этими двумя обновлениями находится в недопустимом состоянии, в том смысле, что она не отражает действительное состояние дел в реальном мире; вполне очевидно, что в банковской практике перевод денег с одного счета на другой не должен влиять на суммарное количество денежных средств на рассматриваемых счетах, а в данном примере после выполнения первого обновления сумма в 100 долл. на время "исчезает" из

базы данных. Поэтому такая логическая единица работы, как транзакция, не обязательно связана с выполнением только одной операции в базе данных. Транзакция, как правило, чаще всего состоит из последовательности подобных операций, и назначением такой последовательности является преобразование одного действительного состояния базы данных в другое такое состояние; при этом не обязательно требуется, чтобы на всех промежуточных этапах база данных находилась в допустимом состоянии.

```
    BEGIN TRANSACTION ;

    UPDATE ACC 123 ( BALANCE := BALANCE - $100 ) ;
    IF <возникла ошибка> THEN GO TO UNDO ; END IF ;

    UPDATE ACC 456 ( BALANCE := BALANCE + $100 ) ;
    IF <возникла ошибка> THEN GO TO UNDO ; END IF ;

    COMMIT ;                          /* Успешное завершение    * /
    GO TO FINISH ;

UNDO :
    ROLLBACK ;                         /* Неудачное завершение */

FINISH :
    RETURN ;
```

Рис. 15.1. Пример транзакции (псевдокод)

Итак, вполне очевидно, что в данном примере следует избегать такой ситуации, когда одно из обновлений будет выполнено, а другое — нет, поскольку в результате база данных останется в недопустимом состоянии. В идеале желательно иметь абсолютную гарантию того, что будут выполнены оба обновления. Но, к сожалению, действительно предоставить любую подобную гарантию невозможно, поскольку всегда имеется вероятность, что события будут развиваться по наихудшему сценарию, мало того, это может произойти в самый неблагоприятный момент. Например, на этапе перехода от одного обновления к другому может произойти аварийный останов системы[1], во время выполнения второго обновления может возникнуть арифметическое переполнение и т.д. Но система, которая обеспечивает **управление транзакциями,** хотя и не предоставляет подобную гарантию, вполне может ее заменить другими средствами. А именно, система с поддержкой транзакций гарантирует, что если в транзакции будут выполнены некоторые обновления, а затем возникнет отказ еще до того, как транзакция достигнет своего завершения, то все эти обновления будут отменены. Поэтому транзакция либо полностью выполняется, либо полностью отменяется (т.е. создается такая ситуация, как если бы эта транзакция вообще не выполнялась). Благодаря этому последовательность операций, которая по самой своей сути не является неразрывной, может быть преобразована в такую последовательность, которая внешне кажется неразрывной.

---

[1] Аварийный останов системы, который прерывает выполнение всех транзакций, действующих в системе во время его возникновения, называется также глобальным отказом или отказом системы, а отказ, такой как арифметическое переполнение, который влияет только на одну транзакцию, называется локальным отказом. Дополнительная информация на эту тему приведена, соответственно, в разделах 15.4 и 15.3.

Компонент системы, который обеспечивает такую неразрывность (или подобие неразрывности), называется **диспетчером транзакций** (для его обозначения применяются также термины **монитор обработки транзакций** или **TP-монитор),** а в основе организации его работы лежат операции COMMIT и ROLLBACK, которые описаны ниже.

- Оператор COMMIT (Зафиксировать) сигнализирует об *успешном* окончании транзакции. Он сообщает диспетчеру транзакций, что логическая единица работы успешно завершена, база данных вновь находится (или будет находиться после выполнения этого оператора) в непротиворечивом состоянии, а все обновления, выполненные данной логической единицей работы, теперь могут быть *зафиксированы,* т.е. внесены в базу данных.
- Оператор ROLLBACK (Выполнить откат) сигнализирует о *неудачном* окончании транзакции. Он сообщает диспетчеру транзакций, что произошла какая-то ошибка, база данных находится в противоречивом состоянии и следует осуществить *откат* всех проведенных при выполнении этой транзакции обновлений, т.е. они *должны* быть отменены.

Таким образом, в приведенном примере оператор COMMIT должен быть выполнен, если оба обновления прошли успешно, после чего внесенные в базу данных изменения станут постоянными. Если что-то произошло не так, например, если обновление было прервано каким-либо условием ошибки, то выполняется оператор ROLLBACK и любые проведенные до сих пор изменения отменяются.

На основании простого примера, приведенного на рис. 15.1, можно сделать ряд важных выводов, описанных ниже.

- **Неявно заданный оператор ROLLBACK.** В данном примере явно предусмотрено выполнение проверок на наличие ошибок и явный вызов на выполнение оператора ROLLBACK в случае обнаружения ошибки. Но не следует предполагать (и само это предположение не имеет смысла), что транзакции всегда включают явно заданные проверки на случай всех возможных ошибок. Поэтому система должна вызывать на выполнение неявно заданный оператор ROLLBACK для всех транзакций, которые по какой-то причине окончились неудачей и не достигли этапа планового завершения (здесь под *плановым завершением* подразумевается явно заданный оператор COMMIT или ROLLBACK).
- **Обработка сообщений.** Типичная транзакция не только вносит (или пытается внести) обновления в базу данных, но и передает конечному пользователю те или иные сообщения, позволяющие ему узнать, что происходит. Например, можно предусмотреть передачу сообщения "Перевод денег со счета на счет выполнен", если достигнут оператор COMMIT, или сообщение "Ошибка — перевод денег со счета на счет не выполнен" в противном случае. Обработка сообщений, в свою очередь, требует применения дополнительных средств восстановления. Более подробную информацию по этой теме можно найти в [15.12].
- **Журнал восстановления.** На первый взгляд не всегда очевидно, как можно обеспечить отмену обновления. Ответ на этот вопрос состоит в том, что в системе ведется **журнал** на ленте или (чаще всего) на диске, в котором регистрируются подробные сведения обо всех обновлениях, в частности, значения обновляемых объектов

(например кортежей) до и после каждого обновления, иногда называемые *образами данных до и после обновления.* Поэтому, если возникает необходимость отменить некоторые конкретные обновления, система использует соответствующую запись журнала для восстановления предыдущего значения обновленного объекта.

Примечание. **В** действительности это описание слишком упрощено. На практике журнал состоит из двух частей — активной (или оперативной) части и архивной (или автономной) части. *Оперативной* называется та часть журнала, которая используется в процессе текущей эксплуатации системы для регистрации подробных сведений об обновлениях по мере их выполнения, и обычно хранится на диске. После заполнения пространства памяти, отведенного для оперативной части журнала, ее содержимое передается в *автономную* часть, которая теперь может быть записана на ленту (поскольку всегда обрабатывается последовательно).

- **Неразрывность операций.** Система должна гарантировать, что выполнение отдельных операций (весь процесс их выполнения) должно происходить неразрывно. Такое требование становится особенно важным в реляционных системах, поскольку в них операции выполняются на уровне множества и обычно затрагивают одновременно большое количество кортежей, поэтому нельзя допустить, чтобы такие операции оканчивались неудачей в ходе выполнения и оставляли базу данных в противоречивом состоянии (например, в таком состоянии, что некоторые кортежи обновлены, а другие — нет). Иными словами, если в ходе выполнения подобного оператора возникла ошибка, то база данных должна остаться полностью в неизменном состоянии. Кроме того, как описано в главах 9 и 10, это условие остается в силе, даже если рассматриваемая операция неявно требует выполнения дополнительных обновлений (например, из-за того, что применяется правило каскадного удаления или происходит обновление представления, в котором предусмотрено соединение таблиц).
- **Выполнение программы как последовательности транзакций.** Необходимо обратить особое внимание на то, что операторы COMMIT и ROLLBACK завершают транзакцию, а не прикладную программу. Как правило, процесс прогона отдельной программы состоит из ряда транзакций, которые следуют одна за другой, как показано на рис. 15.2.
- **Отсутствие вложенных транзакций.** Если явно не указано иное, то в данной главе предполагается, что в прикладной программе оператор BEGIN TRANSACTION может быть вызван на выполнение, только если в настоящее время в этой программе не выполняется какая-либо другая транзакция. Иными словами, транзакция не может содержать в себе вложенную транзакцию. Но необходимо отметить, что в следующей главе это предположение будет пересмотрено.
- **Допустимость.** По определению база данных должна всегда находиться, по меньшей мере, в непротиворечивом состоянии. Под понятием *непротиворечивости,* согласно главе 9 (а также общепринятому толкованию этого термина), мы подразумеваем *отсутствие нарушений каких-либо известных ограничений целостности.* Следует также отметить, что непротиворечивость базы данных, рассматриваемая с точки зрения такого определения данного термина, обеспечивается средствами самой СУБД. Из этого следует (также по определению), что транзакции всегда переводят базу данных из одного непротиворечивого состояния в другое. Но одной

непротиворечивости недостаточно; база данных должна отвечать требованиям допустимости, а не только непротиворечивости! Например, в случае транзакции, показанной на рис. 15.1, требуется, чтобы суммарное количество денег на счетах 123 и 456 в результате транзакции не изменилось. Но было бы неразумно ставить перед собой задачу по подготовке ограничения целостности, которое соответствовало бы такому требованию (объясните, почему), и поэтому нельзя рассчитывать на то, что СУБД сможет следить за соблюдением данного требования. В действительности, как уже было сказано в главе 9, непротиворечивость не подразумевает правильность. Поэтому, к сожалению, все, что мы можем сделать (и чего мы вообще можем добиться с помощью СУБД), это просто принять предположение, что транзакции являются правильными, в том смысле, что они достоверно отражают лишь те операции реального мира, которые были реализованы с их помощью. Точнее, мы предполагаем, что если т — транзакция, которая переводит базу данных из состояния D1 в состояние D2 и состояние D1 является правильным, то D2 также является правильным[2]. Но еще раз отметим, что это желаемое свойство не может быть обеспечено самой системой (как было указано в главе 9, система может обеспечить не истинность, а только непротиворечивость).

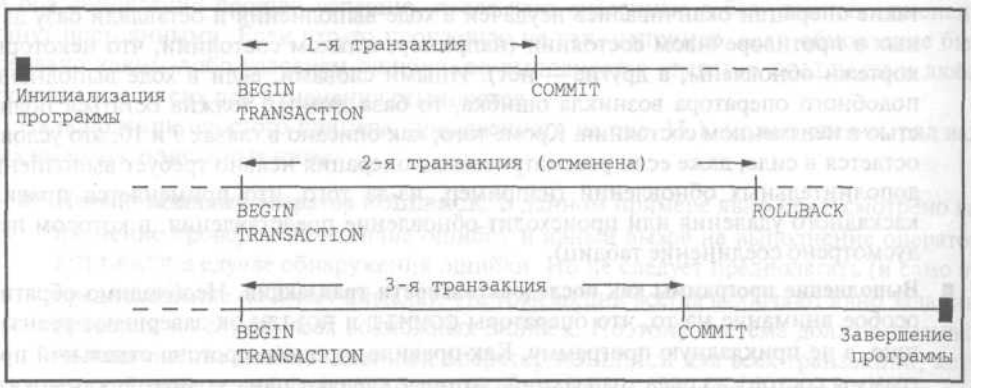


Рис. 15.2. Диаграмма, на которой процесс выполнения программы показан как ряд транзакций

- Множественное присваивание. Как было указано в главе 5 (в соответствии с доводами, приведенными в [3.3]), в базе данных необходимо предусмотреть поддержку множественной формы присваивания, которая позволяет выполнять одновременно любое количество отдельных операций присваивания (т.е. обновлений). Например, две отдельные операции UPDATE, показанные на рис. 15.1, могут быть заменены следующим одним оператором (обратите внимание на применяемый в нем разделитель в виде запятой).

[2] Следует также обратить особое внимание на то, что это утверждение должно распространяться на все возможные правильные состояния D1. Очевидно, что транзакция Т может фактически не "достоверно отражать лишь те операции реального мира, которые были с помощью нее реализованы", и вместе с тем обеспечивать переход в правильное состояние D2 из некоторого конкретного состояния D1. Но нас это не вполне устраивает, поскольку правильность транзакции должна быть гарантированной, а не являться просто результатом случайного стечения обстоятельств.

```
UPDATE ACC 123 { BALANCE := BALANCE - $100
} , UPDATE ACC 456 { BALANCE := BALANCE +
$100 } ;
```

Теперь эта транзакция будет включать только одну операцию обновления, поэтому ее воздействие на базу данных будет неразрывным по определению и операторы BEGIN TRANSACTION, COMMIT и ROLLBACK больше не потребуются (по крайней мере, в данном конкретном случае). Но, как было отмечено в главе 5, современные программные продукты не поддерживают множественное присваивание (или поддерживают эту операцию не полностью). Поэтому по практическим соображениям до конца данной главы возможность применения множественного присваивания не рассматривается. (Безусловно, при проведении оригинальной исследовательской работы, посвященной транзакциям, возможность применения множественного присваивания фактически даже не рассматривалась. Дополнительная информация на эту тему приведена в разделе 16.10.)

15.3. ВОССТАНОВЛЕНИЕ ТРАНЗАКЦИИ

Транзакция начинается с выполнения оператора BEGIN TRANSACTION и заканчивается выполнением оператора COMMIT или ROLLBACK. Оператор COMMIT устанавливает так называемую **точку фиксации** (которую называют также **точкой синхронизации** — syncpoint, особенно в ранее созданных системах). Точка фиксации соответствует (успешному) окончанию логической единицы работы и, следовательно, точке, в которой база данных находится (или будет находиться после фиксации) в непротиворечивом состоянии. В противовес этому, после выполнения оператора ROLLBACK база данных вновь возвращается в состояние, в котором она была в момент начала обработки оператора BEGIN TRANSACTION, т.е. в предыдущую точку фиксации.

Примечание. Выражение *предыдущая точка фиксации* является вполне допустимым даже в случае выполнения в программе первой транзакции, при условии, что первый оператор BEGIN TRANSACTION по умолчанию устанавливает в программе первоначальную точку фиксации. Следует также отметить, что в этом контексте термин *база данных* фактически означает доступную для транзакции часть базы данных. Другие транзакции могут выполняться параллельно с данной транзакцией и вносить изменения в иные части базы данных. Поэтому *общая база данных* может и *не* быть в полностью непротиворечивом состоянии в момент фиксации конкретной транзакции. Но, как описано в разделе 15.1, в данной главе не учитывается возможность параллельного выполнения транзакций, поскольку такое упрощение существенно не влияет на рассматриваемый вопрос.

Ниже перечислены случаи установки точки фиксации.

1. Как описано в разделе 15.2, все обновления, совершенные программой с момента установки предыдущей точки фиксации, выполнены, т.е. получили статус *постоянных* в том смысле, что гарантирована запись их результатов в базу данных. До достижения точки фиксации все выполненные транзакцией обновления могут рассматриваться *только как намеченные* в том смысле, что они могут быть отменены (т.е. существует возможность их отката). Но гарантируется, что с момента фиксации обновление уже никогда не будет отменено[3] (это и есть определение понятия *фиксации).*

---

[3] Под этим подразумевается, что такая отмена не произойдет, кроме как в результате явных действий пользователя. (Зафиксированное обновление фактически может быть также отменено с помощью неявного действия системы, если допускается применение вложенных транзакций [15.15], но это последняя возможность в данной главе не рассматривается.)

2. Утрачена вся информация о позиционировании базы данных **и** сняты все блокировки кортежей. Определение понятия *позиционирования базы данных* в этом контексте основано на том, что в любой заданный момент выполняющаяся программа обычно адресуется к конкретным кортежам в базе данных (например, с помощью определенных *курсоров* в случае языка SQL, как показано в главе 4). Эта адресуемость в точке фиксации теряется. Понятие *блокировка кортежей* рассматривается в следующей главе.

Примечание. В некоторых системах имеется опция, с помощью которой программа может сохранять адресуемость к некоторым кортежам (а значит, сохранять некоторые кортежи заблокированными) от одной транзакции к другой; фактически такая возможность была введена в стандарте SQL в 1999 году. Более подробно речь об этом пойдет ниже, в разделе 15.8.

Примечание. Здесь п. 2 также действителен, если транзакция завершается оператором ROLLBACK, а не COMMIT (за исключением замечания о возможности сохранения некоторой адресуемости и, следовательно, соответствующей блокировки кортежей). К п. 1 это, безусловно, не относится.

Из всего сказанного выше следует, что транзакции — это **не** только логические единицы работы, но и единицы **восстановления.** Это означает, что при успешном завершении транзакции система гарантирует, что выполненные ею обновления будут существовать в базе данных постоянно, даже если в следующий момент в системе произойдет аварийный останов. Вполне возможно, что в системе произойдет сбой после успешного выполнения оператора COMMIT, но перед тем, как обновления будут физически записаны в базу данных (они все еще могут оставаться в буфере оперативной памяти[4] и, таким образом, могут быть утеряны в момент сбоя системы). Даже если подобное произойдет, процедура перезапуска системы все равно должна зарегистрировать эти обновления в базе данных; чтобы узнать о том, действительно ли эти значения были записаны в базу данных, можно ознакомиться с соответствующими записями в журнале. (Из этого следует, что физическая запись в журнал регистрации должна быть внесена *перед* завершением операции COMMIT. Это важное правило называется **правилом опережающей записи в журнал** — write-ahead log rule.) Процедура перезапуска позволяет восстановить любые успешно завершенные транзакции, обновления которых не были физически записаны во вторичную память до возникновения сбоя системы. Следовательно, как и указывалось ранее, транзакция действительно является единицей восстановления.

Примечание. В следующей главе будет показано, что транзакции являются также единицами организации *параллельной работы.*

На данном этапе необходимо рассмотреть несколько вопросов реализации. Интуитивно должно быть ясно, что практическое осуществление задачи поддержки транзакций становится проще, если соблюдаются оба из описанных ниже условий.

- Данные об обновлениях базы данных хранятся в буферах оперативной памяти и не записываются физически на диск до фиксации транзакции. Таким образом, если транзакция оканчивается неудачей, то нет необходимости отменять какие-либо обновления, записанные на диске.

---

[4] *Буферами* называются промежуточные области в оперативной памяти для данных, которые должны передаваться (в том или ином направлении) между базой данных, хранящейся на физическом носителе, и некоторой выполняемой транзакцией.

- Обновления базы данных физически записываются на диск в процессе выполнения оператора COMMIT транзакции. Таким образом, если в дальнейшем произойдет аварийный останов системы, то можно быть уверенным в том, что нет необходимости повторно выполнять какие-либо операции обновления на диске.

Но на практике (как правило) ни одно из этих требований не соблюдается. Во-первых, буфера могут просто оказаться недостаточно большими, а это означает, что потребуется записывать на диск обновления транзакции A еще до того, как произойдет фиксация транзакции A с помощью оператора COMMIT. Такая необходимость может, например, возникнуть в связи с тем, что потребуется освободить место для обновлений транзакции в (в подобных ситуациях принято говорить, что транзакция в **вытесняет** транзакцию A из буферного пространства). Во-вторых, физическая (или **принудительная)** запись обновлений на диск во время выполнения оператора COMMIT может оказаться очень неэффективной; например, если все 100 последовательных транзакций обновляют один и тот же объект, то принудительная запись означает, что должно быть выполнено 100 физических операций записи, когда вполне достаточно одной. По этим причинам в применяемых на практике диспетчерах транзакций обычно предусмотрено так называемое правило *конфискации и отказа от принудительной записи* (steal/no-force), а это, вполне очевидно, весьма значительно усложняет реализацию. Дополнительные сведения об этом выходят за рамки данной книги.

Итак, как уже было сказано, правило опережающей записи в журнал означает, что обработка оператора COMMIT может закончиться только после того, как будет выполнена физическая запись в журнал. Безусловно, это утверждение остается в силе, но теперь его можно расширить и уточнить, как описано ниже.

- Запись журнала, касающаяся конкретного обновления базы данных, должна быть физически внесена в журнал еще до того, как само обновление будет физически внесено в базу данных.
- Все другие записи журнала, касающиеся данной конкретной транзакции, должны быть физически внесены в журнал еще до того, как в журнал будет физически вне сена запись с оператором COMMIT, которая относится к указанной транзакции.
- Обработка оператора COMMIT для данной транзакции не должна завершаться до тех пор, пока в журнал не будет физически внесена запись с оператором COMMIT, который относится к этой транзакции[5].

[5] В некоторых системах предусмотрен принудительный вывод записи журнала с оператором COMMIT на диск сразу после получения запроса на выполнение оператора COMMIT, а в других системах происходит ожидание до тех пор (например), пока буфер не заполнится, и только после этого выполняется принудительный вывод этой записи на диск. Последний из этих методов называется *групповой фиксацией* (group commit) на том основании, что его применение обычно влечет за собой одновременную фиксацию нескольких транзакций; при использовании такого метода количество операций ввода—вывода сокращается, но продолжительность выполнения некоторых транзакций увеличивается.

Свойства ACID транзакций

Как и в [15.14], можно подытожить материал этого и предыдущего разделов, сделав заключение, что транзакции обладают (или должны обладать!) четырьмя важными свойствами: *неразрывностью* (atomicity), *правильностью*[6] (correctness), *изолированностью* (isolation) и *устойчивостью* (durability). Этот набор свойств принято называть **свойствами ACID** (по первым буквам их английских названий).

- **Неразрывность.** Транзакции неразрывны (выполняются по принципу "все или ни чего").
- **Правильность.** Транзакции преобразуют базу данных из одного правильного со стояния в другое; при этом правильность не обязательно должна обеспечиваться на всех промежуточных этапах.
- **Изолированность.** Транзакции изолированы одна от другой. Таким образом, даже если будет запущено множество транзакций, работающих параллельно, результаты любых операций обновления, выполняемых отдельной транзакцией, будут скрыты от всех остальных транзакций до тех пор, пока эта транзакция не будет зафиксиро вана. Иначе говоря, для любых отдельных транзакций A и B справедливо следую щее утверждение: транзакция A сможет получить результаты выполненных тран закцией в обновлений только после фиксации транзакции в, а транзакция в смо жет получить результаты выполненных транзакцией A обновлений только после фиксации транзакции A.
- **Устойчивость.** После того как транзакция зафиксирована, выполненные ею обновления сохраняются в базе данных на постоянной основе, даже если в дальнейшем произойдет аварийный останов системы.

15.4. ВОССТАНОВЛЕНИЕ СИСТЕМЫ

Система должна быть готова к восстановлению не только после локальных отказов, подобных возникновению условия переполнения при выполнении операции в пределах определенной транзакции, но и после глобальных нарушений, подобных отключению питания. Локальное нарушение по определению влияет только на ту транзакцию, в которой оно, собственно говоря, и произошло. Подобные нарушения уже обсуждались выше, в разделах 15.2 и 15.3. Глобальное нарушение воздействует сразу на все транзакции, выполнявшиеся в момент его возникновения, и поэтому приводит к более значительным для системы последствиям. В этом и следующем разделе кратко описано, какие действия необходимо выполнить в процессе восстановления после глобального отказа системы. Такие отказы подразделяются на две основные категории, которые описаны ниже.

- **Отказы** системы (например, сбои в питании) воздействуют на все выполняющиеся в данный момент транзакции, но не приводят к физическому повреждению базы данных. Отказ системы иногда также называют *мягким аварийным остановом.*

---

[6] В литературе (в частности в [15.14]) чаще всего вместо термина "правильность" (correctness) применяется термин "совместимость" (consistency), но в этой главе уже было сказано, по каким причинам автор предпочитает именно первый термин.

- **Отказы носителей** (например, поломка головки дискового накопителя), которые могут представлять угрозу для физического состояния всей базы данных или какой-либо ее части, способны воздействовать, по крайней мере, на те транзакции, которые используют поврежденную часть базы данных. Отказ носителя иногда называют также *жестким аварийным остановом.*

В этом разделе будут рассмотрены отказы системы, а в разделе 15.5 — отказы носителей.

Критическим моментом в отказе системы является *потеря содержимого основной (оперативной) памяти* (в частности, буферов базы данных). Поскольку точное состояние любой выполнявшейся в момент отказа системы транзакции остается неизвестным, такая транзакция не может быть успешно завершена. Поэтому при перезапуске системы любая такая транзакция будет *отменена* (т.е. будет выполнен ее откат).

Более того, при перезапуске системы, возможно, потребуется (как указывалось в разделе 15.3) *повторно выполнить* некоторые транзакции, которые были успешно завершены до аварийного останова, но выполненные ими обновления еще не были перенесены из буферов оперативной памяти в физическую базу данных во вторичной памяти.

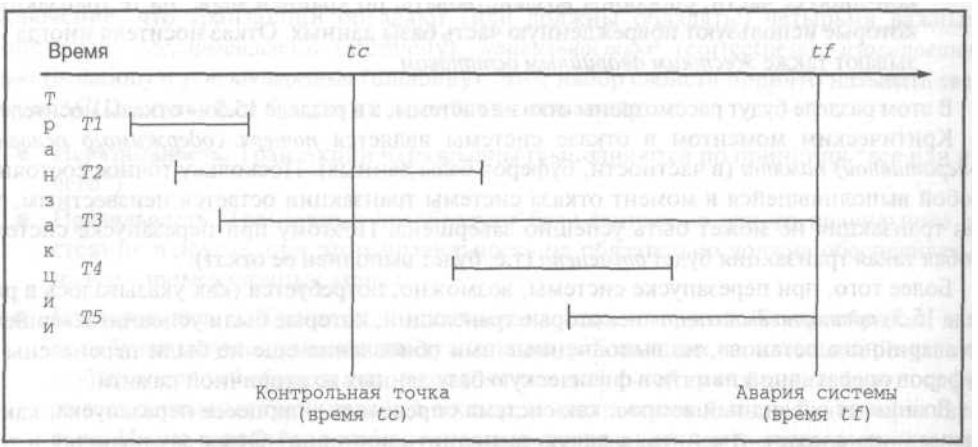
Возникает очевидный вопрос: как система определяет в процессе перезапуска, какую транзакцию следует отменить, а какую выполнить повторно? Ответ заключается в том, что система автоматически создает *контрольные точки* с некоторым наперед заданным интервалом (обычно, когда в журнале накапливается определенное число записей). Для создания контрольной точки требуется, во-первых, выполнить принудительное сохранение содержимого буферов оперативной памяти в физической базе данных, и, во-вторых, осуществить принудительное сохранение специальной записи **контрольной точки** в журнале на физическом носителе. **Запись контрольной точки** содержит список всех транзакций, выполняемых в тот момент, когда создавалась контрольная точка. Для того чтобы ознакомиться с тем, как используется эта информация, рассмотрим рис. 15.3, который можно описать словами следующим образом (ось времени на этом рисунке направлена слева направо).

- Отказ системы произошел в момент времени tf.
- Ближайшая к моменту tf контрольная точка была создана в момент времени tc.
- Транзакция типа T1 успешно завершена до момента времени tc.
- Транзакция типа T2 начата до момента tc и успешно завершена после момента времени tc, но до момента времени tf.
- Транзакция типа T3 также начата до момента времени tc, но не завершена к мо менту времени tf.
- Транзакция типа T4 начата после момента времени tc и успешно завершена до момента времени tf.
- Наконец, транзакция типа T5 также начата после момента tc, но не завершена к моменту времени tf.

Очевидно, что при перезапуске системы транзакции типов T3 и T5 должны быть отменены, а транзакции типа T2 и T4 — выполнены повторно. Отметим, что транзакции типа T1 вообще не участвуют в процессе перезапуска, поскольку выполненные ими обновления были принудительно внесены в физическую базу данных к моменту времени tc как часть процедуры создания этой контрольной точки. Обратите внимание также на то,

Рис. 1S.3. Пять возможных вариантов выполнения транзакций

что транзакции, завершившиеся неудачно (т.е. для них был выполнен откат) до момента времени tf, также не будут учитываться в процессе перезапуска (объясните, почему).



... Следовательно, во время перезапуска система прежде всего выполняет описанную ниже процедуру для выявления всех транзакций типовТ2-Т5.

1. Создаются два списка транзакций; назовем их UNDO (Отменить) и REDO (Выполнить повторно).
2. В список UNDO заносятся все транзакции, упомянутые в последней из существующих записей контрольной точки, а список REDO пока остается пустым.
3. В журнале регистрации поиск начинается с записи контрольной точки и происходит в прямом направлении.
4. Если в журнале регистрации обнаружена запись BEGIN TRANSACTION с указанием о начале выполнения некоторой транзакции т, то эта транзакция добавляется в СПИСОК UNDO.
5. Если в журнале регистрации обнаружена запись COMMIT, свидетельствующая об окончании выполнения некоторой транзакции т, эта транзакция переносится из СПИСКа UNDO В СПИСОК REDO.
6. По достижении конца файла журнала регистрации списки UNDO и REDO анализируются для выявления, соответственно, транзакций типов Т3 и Т5, а также типов Т2 и Т4.

После этого система просматривает журнал регистрации в обратном направлении, выполняя откат транзакций из списка UNDO, а затем вновь просматривает журнал в прямом направлении, повторно выполняя транзакции из списка REDO.

Примечание. Восстановление согласованного состояния базы данных посредством отмены выполненных операций иногда называется *обратным восстановлением.* Аналогичным образом, восстановление согласованного состояния базы данных за счет повторного выполнения уже выполненных действий называется *прямым восстановлением.* Необходимо учитывать, что при прямом восстановлении повторное внесение обновлений

происходит в том порядке, в котором они были выполнены первоначально, а при обратном восстановлении отмена обновлений происходит в обратном порядке.

Система будет готова к дальнейшей работе тогда (и только тогда), когда такая восстановительная работа будет завершена.

Алгоритм ARIES

Безусловно, приведенное выше описание процедуры восстановления системы чрезмерно упрощено[7]. В частности, следует отметить, что в этой процедуре операции отмены внесенных обновлений (называемые также *откатом)* выполняются перед операциями повторного внесения изменений (которые принято также называть *накатом).* На первых порах во многих системах восстановление было организовано именно таким образом, но для повышения эффективности в современных системах эти действия происходят в обратной последовательности. В настоящее время в большинстве систем фактически используется схема, называемая ARIES [15.20], или другая организация работы, весьма похожая на эту схему, в которой операции наката в действительности осуществляются в первую очередь. Выполнение процедур алгоритма ARIES подразделяется на описанные ниже три основных этапа.

1. Анализ. Формирование списков REDO (накат) и UNDO (откат).
2. Накат. Начиная с позиции журнала, которая определена на этапе анализа, восста новление базы данных до того состояния, в котором она находилась во время ава рийного останова.
3. Откат. Отмена результатов внесения изменений теми транзакциями, фиксация которых не была выполнена.

Следует отметить, что принцип "накат перед откатом" предусматривает повторное внесение изменений, сделанных теми транзакциями, которые не были зафиксированы и для этих изменений в дальнейшем требуется снова выполнять откат. Отчасти по этой причине этап наката в алгоритме ARIES часто называют *повторением истории* [15.21] (или *повторением прежних ошибок).* Заслуживает также внимания тот факт, что в алгоритме ARIES предусматривается запись в журнал всех операций, выполняемых на этапе отката, поэтому, если в ходе выполнения данной процедуры перезапуска снова произойдет аварийный останов системы (а это отнюдь нельзя исключить), то обновления, для которых уже был выполнен откат, не будут отменены повторно при следующем перезапуске.

Аббревиатура ARIES расшифровывается как "Algorithms for Recovery and Isolation Exploiting Semantics" (Алгоритмы восстановления и изоляции на основе семантики).

---

[7] Кроме всего прочего, в нем предполагается, что восстановление всегда возможно! Вполне очевидно, что если никакие транзакции не выполнялись параллельно в момент аварии, то все они могут быть восстановлены. Но если в базе данных предусмотрена возможность параллельного выполнения транзакций, то организация ее работы в значительной степени усложняется и приходится тщательно следить за тем, чтобы выполняемые в ней операции исключали возможность восстановления. Эта тема рассматривается более подробно в следующей главе.

15.5. ВОССТАНОВЛЕНИЕ НОСИТЕЛЕЙ

Примечание. Тема восстановления носителей представляет собой нечто совершенно самостоятельное и не имеющее отношения к транзакциям и восстановлению системы после сбоев. Она включена в данное обсуждение только для создания полной картины.

Как уже отмечалось в разделе 15.4, *отказы носителей* — это нарушения наподобие поломки головок дискового накопителя или отказа контроллера дисков, когда некоторая часть базы данных разрушается физически. Восстановление после такого нарушения включает загрузку (или *восстановление)* базы данных с резервной копии (или из *дампа)* и последующее использование журнала (как его активной, так и архивной частей) для повторного выполнения всех транзакций, успешно завершенных в системе с момента создания данной резервной копии (прямое восстановление). При этом нет никакой необходимости отменять транзакции, которые выполнялись в момент отказа носителей, поскольку по определению все обновления, выполненные этими транзакциями, уже полностью отменены (фактически просто утрачены).

Таким образом, процедура восстановления носителей подразумевает наличие в системе *утилиты копирования/восстановления* (или выгрузки/загрузки). Функция копирования этой утилиты используется для создания резервной копии в установленные моменты. Такие копии могут сохраняться либо на ленте, либо на других устройствах архивирования. Совсем не обязательно, чтобы они создавались на устройствах хранения с прямым доступом. Функция восстановления утилиты используется в случае отказа носителя для воссоздания базы данных с требуемой резервной копии.

15.6. ДВУХФАЗНАЯ ФИКСАЦИЯ

Примечание. При первом чтении этот раздел можно пропустить.

В данном разделе кратко рассматривается очень важное усовершенствование основной концепции фиксации/отката транзакций, а именно — **двухфазная фиксация.** Это средство организации работы требуется в тех условиях, когда определенная транзакция может взаимодействовать с несколькими независимыми *диспетчерами ресурсов,* каждый из которых распоряжается собственным набором восстанавливаемых ресурсов и поддерживает собственный журнал восстановления[8]. Например, допустим, что транзакция, выполняемая на мэйнфрейме IBM, модифицирует как базу данных СУБД IMS, так и базу данных СУБД DB2 (подобные транзакции применяются на практике довольно часто). Если транзакция завершается успешно, то *все* выполненные ею обновления, как в базе данных СУБД IMS, так и в базе данных СУБД DB2, должны быть зафиксированы. В противном случае для *всех* внесенных обновлений должен быть выполнен откат. Иначе говоря, недопустима ситуация, когда обновления информации в базе данных СУБД IMS зафиксированы, а для обновлений информации в базе данных СУБД DB2 выполнен откат, или наоборот. Суть в том, что в подобном случае транзакция перестанет быть неразрывной (выполняемой по принципу "все или ничего").

Отсюда следует, что для транзакции не имеет смысла выполнять, например, оператор COMMIT в СУБД IMS и оператор ROLLBACK в СУБД DB2. Даже если один и тот же оператор будет выдан для обеих СУБД, в системе все равно может произойти аварийный останов

---

[8] В частности, это важно в контексте распределенных систем баз данных, а потому данный вопрос более подробно рассматривается в главе 21.

между завершением этих двух операций, и полученные результаты окажутся неудовлетворительными. Вместо этого транзакция должна выдать одну "глобальную", или **общесистемную,** команду COMMIT (или ROLLBACK). Выполнением таких глобальных операций фиксации или отката управляет системный компонент, называемый **координатором.** Его задача состоит в обеспечении того, что оба диспетчера ресурсов (т.е. СУБД IMS и СУБД DB2 в данном примере) *согласованно* выполнят фиксацию или откат тех обновлений, за которые они ответственны. Более того, он должен обеспечивать такую гарантию даже в том случае, *если отказ системы произошел до завершения всего процесса.* Все это достигается за счет использования протокола двухфазной фиксации.

Ниже приведено описание последовательности работы координатора. Для упрощения примем предположение, что транзакция в базе данных выполнена успешно, т.е. выдана общесистемная команда COMMIT, а не ROLLBACK. После получения запроса на выполнение команды COMMIT координатор осуществляет описанный ниже двухфазный процесс.

1. **Подготовка.** Первая фаза начинается с выдачи координатором всем диспетчерам ресурсов указания подготовиться к завершению транзакции *тем или иным способом.* На практике это означает, что каждый **участник** процесса (т.е. каждый диспетчер ресурсов) должен принудительно выгрузить все записи журнала регистрации для используемых транзакцией локальных ресурсов в собственный физический жур нал регистрации (т.е. из первичной во вторичную энергонезависимую память). Теперь, что бы ни случилось, диспетчер ресурсов будет иметь в своем распоряже нии *постоянную запись* о работе, выполненной им в процессе обработки данной транзакции, поэтому в случае необходимости сможет зафиксировать или отменить выполненные обновления. Если принудительная выгрузка прошла успешно, дис петчер ресурсов отвечает координатору, что "подготовка завершилась успешно". В противном случае он посылает противоположное по смыслу сообщение — "подготовка окончилась неудачей".
2. **Фиксация.** Вторая фаза наступает после того, как координатор получит соответ ствующие ответы от всех участников транзакций. Сначала он принудительно выгружает записи о завершаемой транзакции в собственный физический журнал регистрации, регистрируя свое решение относительно этой транзакции. Если все поступившие ответы были положительными, координатор принимает реше ние глобально зафиксировать данную транзакцию. Если же поступил хотя бы один отрицательный ответ, то для транзакции будет выполнен глобальный откат. Затем координатор каким-либо способом информирует каждого из участников транзакции о своем решении, *и каждый участник, согласно поступившей инструк ции, должен локально зафиксировать транзакцию или выполнить ее откат.* Обрати те внимание на то, что каждый участник должен делать то, что ему указал коорди натор в ходе выполнения второй фазы; в этом и состоит суть данного протокола. Обратите также внимание на то, что переход от фазы 1 к фазе 2 начинается именно с появления записи об этом решении в физическом журнале регистрации коорди натора.

Теперь, если система дает сбой в какой-либо точке всего этого процесса в целом, процедура перезапуска будет искать запись о принятом решении в журнале регистрации координатора. Если такая запись будет обнаружена, можно будет установить, какое решение было принято до остановки, и продолжить процесс двухфазной фиксации. Если подобная

запись не будет обнаружена, предполагается, что было принято решение об откате данной транзакции и, следовательно, процесс так или иначе завершится, *Примечание.* Следует подчеркнуть, что если координатор и участники выполняют свою работу на различных компьютерах, что типично для распределенной системы (глава 21), то ошибка в работе координатора может привести к тому, что некий участник достаточно долго будет ожидать поступления сведений о принятом координатором решении. В течение всего *времени ожидания* любое из обновлений, произведенное транзакцией в базе данных этого участника, должно быть скрыто от других транзакций (иначе говоря, эти обновления должны быть *заблокированы,* о чем речь пойдет в следующей главе). Отметим, что диспетчер передачи данных (Data Communications Manager), или сокращенно DC-диспетчер (см. главу 2), также может рассматриваться как диспетчер ресурсов в описанном выше смысле. Это означает, что сообщения можно считать такими же восстанавливаемыми ресурсами, как и саму базу данных, а диспетчер передачи данных должен быть способен участвовать в процессе двухфазной фиксации. Для дальнейшего изучения этого вопроса, а также общей идеи двухфазной фиксации обратитесь к [15.12].

15.7. ТОЧКИ СОХРАНЕНИЯ (ОТСТУПЛЕНИЕ ОТ ОСНОВНОЙ ТЕМЫ)

Выше было описано, что при обычных условиях транзакции не могут быть вложены одна в другую. Но иногда возникает необходимость предусмотреть разбиение транзакций на меньшие *субтранзакции.* Такая задача может быть решена, но только частично, поскольку предусмотрена лишь возможность устанавливать в процессе выполнения транзакции промежуточные точки сохранения и в дальнейшем выполнять откат к заранее установленной точке сохранения (если это потребуется), чтобы не возобновлять всю работу по выполнению транзакции с самого начала. Механизм создания точек сохранения, действующий по указанному принципу, фактически был включен в самые первые версии некоторых систем, в том числе Ingres (в коммерческий программный продукт, а не прототип) и System R. Кроме того, это средство было введено в стандарт SQL в 1999 году. Но следует отметить, что операция установки точки сохранения не является аналогичной операции COMMIT; обновления, внесенные в ходе транзакции, все еще остаются невидимыми для других транзакций до тех пор, пока не будет успешно выполнена операция COMMIT в текущей транзакции. Дополнительные сведения по этой теме приведены в разделе 15.8.

15.8. ПОДДЕРЖКА ЯЗЫКА SQL

В этом разделе средства поддержки работы с транзакциями в языке SQL (в частности, процедуры восстановления на основе транзакций) рассматриваются в соответствии с общими принципами, описанными в предыдущих разделах. Прежде всего, в системе гарантируется, что выполнение большинства операторов SQL происходит неразрывно (единственными исключениями являются CALL и RETURN). Кроме того, как было описано в главе 4, в языке SQL предусмотрены непосредственные аналоги операторов BEGIN TRANSACTION, COMMIT и ROLLBACK, которые называются в нем, соответственно, START TRANSACTION, COMMIT WORK и ROLLBACK WORK. Ниже показан синтаксис оператора START TRANSACTION.

```
START TRANSACTION <option commalist> ;
```

Здесь параметр *<option commalist>,* как показано ниже, определяет режим доступа, уровень изоляции или то и другое (описание четвертого варианта, который касается определения размера диагностической области, выходит за рамки данной книги).

- Режим доступа может обозначаться ключевым словом READ ONLY или READ WRITE. Если ни одно из них не указано, то по умолчанию применяется ключевое слово READ WRITE (при условии, что не определен уровень изоляции READ UNCOMMITTED, так как в этом случае по умолчанию применяется READ ONLY). Если задано ключевое слово READ WRITE, то уровень изоляции не должен принимать значение READ UNCOMMITTED.
- Определение уровня изоляции принимает форму ISOLATION LEVEL *<isolation>,* где параметр *<isolation>* может иметь значение READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ или SERIALIZABLE. Дополнительная информация по этой теме приведена в главе 16.

Синтаксис операторов COMMIT и ROLLBACK приведен ниже.

```
COMMIT [ WORK ]  [ AND [ NO ] CHAIN ]
; ROLLBACK [ WORK ]  [ AND [ NO ]
CHAIN ] ;
```

Ключевое слово WORK является необязательным. Применение ключевого слова AND CHAIN обеспечивает автоматическое выполнение оператора START TRANSACTION после вызова оператора COMMIT (с таким же параметром *<option commalist>,* как и у предыдущего оператора START TRANSACTION); ключевое слово AND NO CHAIN задано по умолчанию. Оператор CLOSE выполняется автоматически для каждого открытого курсора (что приводит к потере позиционирования всех курсоров в базе данных), за исключением курсоров, объявленных с ключевым словом WITH HOLD (это относится только к оператору COMMIT).

Примечание. Хотя об этом не было сказано в главе 4, ключевое слово WITH HOLD является одной из опций в объявлении курсора. Курсор, объявленный как WITH HOLD, не закрывается автоматически после выполнения оператора COMMIT, а остается открытым и имеет такое позиционирование, что следующий оператор FETCH переведет его на следующую по порядку строку. Благодаря этому исключается необходимость разрабатывать сложный код повторного позиционирования курсора, который бы в противном случае потребовался в следующем операторе OPEN.

Кроме того, в языке SQL поддерживаются точки сохранения. Приведенный ниже оператор создает точку сохранения с именем, указанным пользователем (которое известно только в пределах данной транзакции).

SAVEPOINT *<savepoint name>* ;

Следующий оператор отменяет все обновления, внесенные в базу данных со времени создания указанной точки сохранения.

ROLLBACK TO *oavepoint name>* ;

Ниже приведен оператор, который удаляет указанную точку сохранения, а это означает, что с этого времени нельзя будет выполнить оператор ROLLBACK для возврата к данной точке сохранения.

RELEASE <*savepoint name*> ;

Все точки сохранения уничтожаются автоматически после завершения транзакции.

15.9. РЕЗЮМЕ

В этой главе кратко представлена необходимая информация об управлении **транзакциями.** Транзакция — это **логическая единица работы,** а также **единица восстановления** (кроме того, единица параллельности; подробности приводятся в главе 16). Транзакции обладают такими **свойствами ACID,** как **неразрывность, правильность** (которое в литературе чаще называется **совместимостью), изолированность** и **устойчивость. Управление транзакциями** предусматривает решение задачи организации их выполнения таким образом, чтобы соблюдение всех этих важнейших свойств абсолютно гарантировалось (за исключением правильности). Проще говоря, общая цель функционирования всей системы может быть определена как **надежное выполнение транзакций.**

Транзакции начинаются с операции **BEGIN TRANSACTION** и оканчиваются после выполнения операции **COMMIT** (в случае *успешного* завершения) или **ROLLBACK** (в случае *неудачного* завершения). Оператор COMMIT устанавливает **точку фиксации** (при этом обновления записываются в базу данных). Выполнение оператора ROLLBACK возвращает базу данных в предыдущую точку фиксации (все внесенные обновления отменяются). Если транзакция не достигает запланированного завершения, система автоматически выполняет для нее операцию ROLLBACK **(восстановление транзакции).** Для получения возможности отмены (отката) и повторного применения (наката) обновлений в системе ведется **журнал восстановления.** Важно отметить, что все записи регистрации хода выполнения некоторой транзакции должны быть физически занесены в журнал *до* выполнения для этой транзакции операции COMMIT **(правило опережающей записи в журнал).**

Если происходит аварийный останов системы, то после перезапуска система должна, во-первых, **повторно внести** все изменения, выполненные в транзакциях, успешно завершенных до этого останова, и во-вторых, **отменить** все обновления, сделанные в транзакциях, которые начали выполняться, но не были завершены до останова. Необходимо гарантировать сохранение свойств ACID транзакций даже в случае сбоя системы. Эти операции по **восстановлению системы** осуществляются в составе процедуры **перезапуска системы** (которая иначе называется процедурой *перезапуска/восстановления).* Анализируя последнюю имеющуюся в журнале **запись контрольной точки,** система определяет, какую работу необходимо выполнить повторно, а какую отменить. Записи контрольной точки заносятся в журнал через установленное время.

Система также обеспечивает **восстановление носителей,** что реализуется путем восстановления базы данных с заранее созданной резервной копии (из **дампа)** с последующим использованием журнала для повторного выполнения всей работы, законченной в базе данных после создания указанной резервной копии. Для поддержки функции восстановления носителей используются **утилиты создания/восстановления** резервной копии.

Системы, позволяющие транзакциям взаимодействовать с двумя (или несколькими) различными **диспетчерами ресурсов** (например, с двумя разными СУБД или с СУБД и диспетчером передачи данных), должны использовать протокол, называемый **протоколом двухфазной фиксации** (или какую-то версию этого протокола), для того, чтобы эти системы поддерживали свойство неразрывности транзакций. Две фазы этого протокола включают, во-первых, фазу **подготовки,** в которой **координатор** дает указание всем участникам

"подготовиться к выполнению либо фиксации, либо отката", и во-вторых, фазу **фиксации,** в которой координатор (после того как все участники дали ожидаемый от них ответ во время предыдущей фазы) инструктирует участников, что можно выполнить окончательную фиксацию данной транзакции (или ее откат в случае получения хотя бы одного отрицательного ответа).

В завершение этой главы кратко описаны **точки сохранения** и приведен обзор средств восстановления, предусмотренных в языке SQL. В частности, в ней представлен оператор **START TRANSACTION** языка SQL, который дает возможность пользователю указать **режим доступа** и **уровень изоляции транзакции.**

И еще одно, последнее замечание. В принципе, в данной главе речь идет о среде прикладного программирования. Тем не менее, все описанные концепции применимы и к пользовательской среде (хотя на этом уровне они могут быть в большей степени скрытыми). Например, программные продукты SQL обычно позволяют пользователю вводить операторы SQL интерактивно, с терминала. Обычно каждый такой оператор SQL, введенный в интерактивном режиме, обрабатывается как отдельная транзакция — по умолчанию система будет автоматически выполнять операцию COMMIT от имени пользователя сразу после успешного выполнения заданного оператора SQL (или же оператор ROLLBACK, если выполнение оканчивается неудачей). Но в некоторых системах пользователи могут запрещать такие автоматические операторы COMMIT и вместо этого выполнять целый ряд операторов SQL (за которыми следует явно заданный оператор COMMIT или ROLLBACK) как единую транзакцию. Но такая практика обычно не рекомендуется, так как в подобном случае часть базы данных может быть заблокирована и на длительное время стать недоступной для других пользователей (глава 16). Более того, в такой операционной среде для конечных пользователей возможно возникновение ситуации *взаимной блокировки,* что является еще одним доводом в пользу отказа от подобного режима работы (подробные сведения приведены в главе 16).

УПРАЖНЕНИЯ

15.1. Системы баз данных не позволяют транзакции фиксировать изменения только **в** отдельных базах данных (или переменных отношения и т.д.), т.е. без одновременного фиксирования этих изменений во всех других базах данных (или переменных отношения и т.д.). Объясните, почему.

15.2. Обычно не допускается, чтобы транзакции были вложены одна в другую. Объяс ните, почему.

15.3. Сформулируйте определение правила опережающей записи в журнал. Для чего оно нужно?

15.4. Каков смысл описанных ниже требований с точки зрения восстановления?

а) Принудительная запись буферов базы данных во время операцииCOMMIT.

б) Запрещение физической записи буферов в базу данных до выполнения опера ции COMMIT.

15.5. Изложите суть протокола двухфазной фиксации и опишите последствия отказа во время каждой из его двух фаз отдельно для координатора и участника.

15.6. Используя базу данных поставщиков и деталей, разработайте программу SQL для выборки и распечатки сведений обо всех деталях в порядке их номеров детали, удаляя каждую десятую строку и начиная каждую новую транзакцию после каждой десятой строки. Предположим, что для внешнего ключа таблицы деталей по отношению к данным о поставщиках для операции DELETE задано правило CASCADE (т.е. в рамках этого упражнения можно игнорировать поставки). *Примечание.* В данном случае предложено предоставить решение с помощью языка SQL, поэтому при подготовке ответа можно использовать механизм курсоров SQL.

СПИСОК ЛИТЕРАТУРЫ

15.1. Bernstein P.A. Transaction Processing Monitors // CACM. — November 1990. — 33, № 11.

Как было описано в данной главе, *TP-монитор* — это сокращенное обозначение диспетчера транзакций. Эта статья является хорошим неформальным введением в структуру и функции диспетчеров обработки транзакций. Приведем цитату из данной статьи: *"TP-система* является интегрированным набором продуктов, которые... включают аппаратное обеспечение, такое как процессоры, оперативная память, диски, контроллеры обмена данными, а также программное обеспечение, такое как операционные системы, СУБД, компьютерные сети и TP-мониторы. Интеграция этих средств в основном достигается благодаря TP-мониторам".

15.2. Bernstein P.A., Hadzilacos V., Goodman N. Concurrency Control and Recovery in Database Systems. — Reading, Mass.: Addison-Wesley, 1987.

В этом учебнике рассматриваются темы не только восстановления (как следует из его названия), но и управления транзакциями в целом, причем намного шире, чем в настоящей главе.

15.3. Bilris A. et al. ASSET: A System for Supporting Extended Transactions // Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data, Minneapolis, Minn. — May 1994.

Основные рекомендации в области управления транзакциями, изложенные в этой и следующей главах, оказались чрезмерно жесткими для некоторых типов приложений (особенно для тех, где широко применяются интерактивные средства), поэтому для устранения данного недостатка было предложено несколько *расширенных моделей обработки транзакций* [15.16]. Но ко времени написания настоящей книги ни одно из этих предложений не показало своего очевидного превосходства над другими. В результате, как указано в данной статье, "разработчики баз данных [не спешат] включать какую-либо из подобных моделей в свои программные продукты".

Основная область применения системы ASSET выглядит немного иначе. Вместо еще одной модели управления транзакциями в ней предлагается набор примитивных операторов (включая обычный оператор COMMIT, а также некоторые новые операторы), которые могут быть использованы для "создания определяемых пользователем моделей транзакций, которые подходят для конкретных приложений". В частности, в статье показано, что система ASSET может использоваться для

определения "вложенных транзакций, разделяемых транзакций, хроник и других расширенных моделей транзакций, описанных в литературе".

15.4. Bjork L.A. Recovery Scenario for a DB/DC System // Proc. ACM National Conference. — Atlanta, Ga. —August 1973.

Эта статья и связанная с ней статья Дэвиса (Davies) [15.7] представляют собой, наверное, наиболее ранние теоретические работы в области восстановления.

15.5. Cms R.A. Data Recovery in IBM DATABASE 2 // IBM Sys. J. - 1984. - 23, № 2.

Здесь подробно описан механизм восстановления, используемый в системе DB2 (в том виде, в каком он был реализован в первой версии этого программного продукта), в частности, показано, как в системе DB2 происходит возобновление работы после аварийного останова этой системы в ходе восстановления, в то время как некоторая транзакция находится в процессе отката. При решении этой проблемы необходимо убедиться в том, что незафиксированные обновления транзакции, для которой выполнялся откат, будут действительно отменены (эта проблема является в определенном смысле обратной по отношению к проблеме потерянного обновления, которая рассматривается в главе 16).

15.6. Date *C.i.* Distributed Database: A Closer Look // C.J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

В разделе 15.6 этой главы описан *базисный* протокол двухфазной фиксации, для которого можно предложить некоторые усовершенствования. Например, если участник р отвечает координатору с в фазе 1, что он не совершает обновлений в данной транзакции (т.е. работает в режиме *только чтения),* то С может просто игнорировать Р в фазе 2. Соответственно, если все участники в фазе 1 сообщают, что они работают в режиме *только чтения,* фаза 2 может быть полностью исключена (дополнительные сведения по этой теме приведены в главе 21).

Возможны также и другие усовершенствования. В статье приводится вводное описание некоторых из них. В частности, описаны протоколы *предположительной фиксации* и *предположительного отката* (улучшенные версии базисного протокола), модель *дерева процессов* (когда участнику необходимо отвечать за выполнение определенной части транзакции в качестве координатора), а также ситуация, при которой во время пересылки подтверждения от участника к координатору происходит *сбой в системе связи.*

Примечание. Хотя обсуждение в основном ведется в контексте распределенных систем, большинство обсуждаемых концепций фактически может иметь более широкое применение. Еще раз отметим, что расширенное обсуждение некоторых из этих вопросов приведено в главе 21 и особенно в [21.13].

15.7. Davies CT., Jr. Recovery Semantics for a DB/DC System // Proc. ACM National Conf. — Atlanta, Ga. — August 1973.

См. аннотацию к [15.8].

15.8. Davies CT., Jr. Recovery Data Processing Spheres of Control // IBM Sys. J. — 1978. — 17, № 2.

Понятие *сфер управления* (Spheres of Control) было первой попыткой изучить и формализовать то, что в дальнейшем стало называться *управлением транзакциями.* Сфера управления — это абстракция, представляющая часть работы, которая внешне может выглядеть как неразрывная. В отличие от транзакций, которые поддерживаются в большинстве современных систем, сферы управления могут вкладываться одна в другую на произвольную глубину.

15.9. Garcia-Molina H., Salem K. Sagas // Proc. 1987 ACM SIGMOD Intern. Conf. on Management of Data. — San Francisco, Calif. — May 1987.

Одной из важных проблем, связанных с использованием средств управлением транзакциями, которые описаны в настоящей главе, является то, что они предназначены для работы с транзакциями, имеющими очень короткую продолжительность (милли- или даже микросекунды). Если же транзакция длится долго (часы, дни, недели), то, во-первых, в случае ее отката приходится отменять очень большой объем работы, и во-вторых, даже если все пройдет успешно, она все же будет удерживать ресурсы системы (данные, хранящиеся в базе данных и т.д.) неопределенно долгое время, тем самым блокируя доступ к ним других пользователей (этот вопрос подробнее обсуждается в главе 16). К сожалению, наблюдается тенденция к увеличению продолжительности выполнения многих практически применяемых транзакций, особенно в таких новейших областях, как проектирование и разработка программного и аппаратного обеспечения.

Хроники представляют собой попытку решения этой проблемы. **Хроника** (saga) — это последовательность коротких транзакций (в обычном понимании этого термина), для которых со стороны системы гарантируется, что *либо* все транзакции данной последовательности будут выполнены успешно, *либо* будут выполнены некоторые **компенсационные транзакции** [15.7], предназначенные для отмены результатов успешно выполненных транзакций в случае незавершенного выполнения всей хроники в целом (таким образом, система будет приведена в состояние, существовавшее до начала выполнения хроники). Например, в банковских системах для транзакции "Зачислить 100 долл. на счет А" компенсационной транзакцией очевидно будет "Списать 100 долл. со счета А". Расширение оператора COMMIT позволяет пользователю информировать систему о том, что в случае отмены только что завершенной транзакции должна быть запущена определенная компенсационная транзакция. Обратите внимание на то, что в идеальном случае компенсационная транзакция никогда не должна завершаться откатом!

15.10. Gray J. Notes on Data Base Operating System // R. Bayer, R. M. Graham, and G. Seegmuller (eds.). Operating Systems: An Advanced Course (Springer-Verlag Lecture Notes in Computer Science 60). — New York, N.Y.: Springer-Verlag, 1978. Эта статья опубликована также в виде отчета: IBM Research Report RJ 2188. — February 1978.

Это один из первых (и, безусловно, один из наиболее доступных для восприятия) источников сведений по управлению транзакциями. В нем впервые опубликовано общедоступное описание протокола двухфазной фиксации. Очевидно, что эта работа не является такой же всеобъемлющей, как более новые работы [15.12], но мы все еще рекомендуем с ней ознакомиться.

15.11. Gray J. The Transaction Concept: Virtues and Limitations // Proc. 7th Intern. Conf. on Very Large Data Bases. — Cannes, France. — 1981.

В этой работе приводится краткое описание проблем, связанных с транзакциями, включая вопросы их реализации.

15.12. Gray J., Reuter A. Transaction Processing: Concepts and Techniques. — San Mateo, Calif: Morgan Kaufmann, 1993.

Если какая-либо публикация в области информатики и заслуживает эпитета *классическая,* то это, наверное, именно данная книга. Ее объем может показаться просто устрашающим, однако авторам удалось пролить свет даже на самые сложные аспекты обсуждаемой темы и одновременно сделать чтение предлагаемого материала достаточно увлекательным. В предисловии авторы заявляют о своем намерении "помочь... в решении реальных проблем"; книга "прагматична и очень подробно освещает основные проблемы управления транзакциями"; в ней "представлены фрагменты программ, демонстрирующие... основные алгоритмы и структуры данных"; но она не является "лишь справочником". Вопреки последнему утверждению, эта книга представляет собой достаточно полное руководство поданной теме и ей, очевидно, суждено стать стандартным пособием в данной области. Настоятельно рекомендуем ее прочесть.

15.13. Gray J. et al. The Recovery Manager of the System R Data Manager // ACM Comp. Surv. — June 1981. — 13, № 2.

Работы [15.13], [15.19] связаны с описанием особенностей средств восстановления системы System R (одной из первых систем, обладающей такими свойствами). В [15.13] дается краткий обзор всей подсистемы восстановления, а в [15.19] подробно описывается ее конкретный аспект, связанный с использованием механизма *теневых страниц.*

Harder T., Reuter A. Principles of Transaction-Oriented Database Recovery // ACM Comp. Surv. — December 1983. — 15, № 4.

В этой статье впервые было определено сокращение *свойства ACID.* В ней дается очень четкое и подробное описание принципов восстановления, приведена продуманная терминология для описания схем восстановления и методов ведения журналов, даны классификация и описание множества существующих систем в соответствии с этой терминологией. Приведены также интересные эмпирические данные, показывающие частоту возникновения и типичное "приемлемое" время восстановления для трех видов сбоев (локальных, системных и сбоев носителей) в больших системах.

Тип отказа	Частота возникновения	Время восстановления
Локальный	10–100 раз в минуту	Примерно равно времени выполнения транзакции
Системный	Несколько раз в неделю	Несколько минут
Сбой носителей	Один или два раза в год	1–2 часа

15.15. Harder T., Rothermel K. Concepts for Transaction Recovery in Nested Transactions // Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data. — San Francisco, Calif. - May 1987.

В статье предложена схема, согласно которой любая транзакция может иметь субтранзакции более низкого уровня (где субтранзакция также является транзакцией в полном смысле этого слова и может иметь *субсубтранзакции* и т.д.). Транзакция, не являющаяся субтранзакцией любой другой транзакции, называется транзакцией верхнего уровня и должна обладать обычными свойствами ACID, а транзакция, представляющая собой субтранзакцию некоторой другой транзакции, должна обладать только свойствами неразрывности и изолированности. Дополнительные сведения по этой теме приведены в разделе 16.10.

15.16. Korth H.F. The Double Life of the Transaction Abstraction: Fundamental Principle and Evolving System Concept (invited talk) // Proc. 21st Int. Conf. On Very Large Data Bases. — Zurich, Switzerland. — September 1995.

Превосходный краткий обзор путей, по которым может идти развитие понятия *транзакции* для того, чтобы оно удовлетворяло требованиям новых классов приложений.

15.17. Korth H.F., Levy E., Silberschatz A. A Formal Approach to Recovery by Compensating Transactions // Proc. 16th Intern. Conf. on Very Large Data Bases. — Brisbane, Australia. — 1990.

В этой статье приведено формальное определение понятия компенсационных транзакций, которые используются в хрониках [15.9] и в других случаях для *отмены* зафиксированных (а также незафиксированных) транзакций.

15.18. Lomet D., Turtle M.R. Redo Recovery after System Crashes // Proc. 21st Int. Conf. On Very Large Data Bases. — Zurich, Switzerland. — September 1995.

В этой работе дан строгий и тщательный анализ процесса восстановления по принципу наката (т.е. прямого восстановления). "[Хотя] процесс восстановления путем наката является всего лишь одной из форм восстановления, он имеет... большое значение [поскольку является существенной частью всего процесса восстановления] и должен помочь в разрешении самых сложных проблем". Авторы утверждают, что проведенный ими анализ позволяет лучше понять недостатки существующих реализаций и может способствовать значительному усовершенствованию систем восстановления.

15.19. Lorie R. A. Physical Integrity in a Large Segmented Database //ACM TODS. — March 1977.-2, №1.

Как уже было указано в аннотации к [15.13], в этой статье рассматривается механизм *теневых страниц* — один из наиболее важных аспектов подсистемы восстановления System R. (Отметим, что термин *целостность* (integrity) в заглавии статьи имеет значение, несколько отличное от описанного в главе 9.) Идея механизма теневых страниц очень проста. Когда незафиксированное обновление записывается в базу данных, система не перезаписывает существующую страницу, а сохраняет новую где-нибудь на диске. Поэтому старая страница становится "тенью" для новой

страницы. Фиксация обновления сводится к корректировке различных указателей, чтобы они указывали на новую страницу, и уничтожению теневой страницы. И наоборот, откат обновления — это переустановка указателей на теневую страницу и уничтожение новой.

Несмотря на кажущуюся простоту, механизм теневых страниц имеет один серьезный недостаток, заключающийся в разрушении любой физической кластеризации, ранее установленной для данных, и потому этот механизм не был перенесен в систему DB2 [15.5], хотя и применялся в системе SQL/DS [4.14].

15.20. Mohan C, Haderle D., Lindsay B., Pirahesh H., Schwartz P. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging//ACM TODS. - March 1992. - 17, № 1.

Это первая статья, посвященная алгоритму ARIES. Ко времени публикации данной статьи алгоритм ARIES был реализован "в различной степени" в нескольких коммерческих и экспериментальных системах, включая, в частности, DB2. Вот несколько перефразированная цитата из этой статьи: "Решения [проблемы управления транзакциями] можно оценить с помощью нескольких критериев: степень параллельности, поддерживаемая в пределах одной страницы и нескольких страниц; сложность применяемых при этом алгоритмов; потребность в пространстве внешней памяти, а также памяти для данных и для журнала; издержки, обусловленные многочисленными синхронными и асинхронными операциями ввода—вывода, которые необходимо выполнить во время перезапуска/восстановления и нормальной работы; типы поддерживаемых функциональных средств (частичного отката транзакций и т.д.); объем вычислений, выполняемых во время перезапуска/восстановления; степень параллельности вычислений, выполняемых во время перезапуска/восстановления; степень отката транзакций, вызванных ситуациями взаимной блокировки в системе; ограничения, накладываемые на хранимые данные (например, требования уникальности ключей для всех записей, ограничения максимального размера объектов до размера страницы и т.д.); способность поддерживать новые режимы блокировки, в которых допускается параллельное выполнение (с учетом коммутативности и других свойств) таких операций, как пополнение/сокращение объема одних и тех же данных с помощью различных транзакций и т.д. По всем этим критериям [ARIES] характеризуется очень высокими показателями".

Со времени создания метода ARIES появилось множество его усовершенствований, разработано и описано в литературе несколько специализированных версий: ARIES/CSA (для систем с архитектурой "клиент/сервер"), ARIES/IM (для управления индексами), ARIES/KVL (сокращение от Key Value Locking) — блокировка значения ключа, применяемая на индексах, ARIES/NT (для вложенных транзакций) и т.д. См. также [15.21].

15.21. Mohan C. Repeating History Beyond ARIES // Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland. — September 1999.

Параллельность

16.1. ВВЕДЕНИЕ

Термин **параллельность** обозначает такое свойство СУБД, которое, как правило, позволяет одновременно обращаться с помощью многих транзакций к одной и той же базе данных. Очевидно, что в системе, обладающей таким свойством, требуется определенный механизм управления, который позволяет добиться того, чтобы параллельно выполняемые транзакции не нарушали работу друг друга. (Примеры нарушений в работе различных типов, которые могут возникать в отсутствие подобных средств управления, приведены в разделе 16.2.) В настоящей главе дано подробное описание данной темы. Она имеет указанную ниже структуру.

- Как уже было отмечено, в разделе 16.2 рассматриваются некоторые проблемы, которые могут возникнуть, если не предусмотрены подходящие средства управления параллельным доступом.
- В разделе 16.3 представлен широко применяемый механизм устранения подобных проблем, называемый *блокировкой.* (Блокировка — не единственный возможный механизм управления параллельным доступом, но он находит гораздо более широ кое применение на практике по сравнению с другими. Некоторые другие средства описаны в аннотациях к литературе в конце данной главы.)
- Затем в разделе 16.4 показано, как может использоваться блокировка для решения проблем, описанных в разделе 16.2.
- К сожалению, сама блокировка может стать причиной проблем, и одной из наи более известных является *взаимоблокировка.* Описанию этой темы посвящен раздел 16.5.
- В разделе 16.6 рассматривается понятие *упорядочиваемости,* которое является об щепризнанным формальным критерием правильной организации работы в этой области.
- В разделе 16.7 описано, как связана тема организации параллельной работы с те мой предыдущей главы — восстановлением.
- После этого в разделах 16.8 и 16.9 представлены два значительных усовершенство вания основных принципов блокировки — применение уровней изоляции и наме ченных блокировок.
- В разделе 16.10 приведены довольно неожиданные и несколько скептические за мечания на тему так называемых свойств ACID транзакций.
- В разделе 16.11 описаны соответствующие средства языка SQL.
- Наконец, в разделе 16.12 приведено резюме.

В завершение этого вступительного раздела приведем ряд общих замечаний. Во-первых, способ организации параллельного выполнения, как и восстановления, в значительной степени зависит от того, относится ли соответствующая система к типу реляционных систем или к другому типу (но важно отметить, что основная часть ранних теоретических работ в этой области, как и в области восстановления, была выполнена именно в реляционном контексте, поскольку он позволяет достичь "большей определенности" [16.6]). Во-вторых, параллельность, как и восстановление, — это очень обширная тема, и в этой главе мы можем рассчитывать лишь на то, что в ней удастся изложить наиболее важные и фундаментальные идеи. Некоторые более сложные аспекты этой темы кратко рассматриваются в некоторых аннотациях к списку литературы в конце данной главы.

16.2. ТРИ ПРОБЛЕМЫ ОРГАНИЗАЦИИ ПАРАЛЛЕЛЬНОЙ РАБОТЫ

Вначале рассмотрим некоторые из проблем, с которыми должен помочь справиться любой механизм управления параллельным выполнением. По сути, возникают три основные ситуации, в которых могут происходить нарушения работы. Это такие ситуации, что транзакция, будучи правильной сама по себе в том смысле, который определен в главе 15, может, тем не менее, выработать неправильный ответ, если в ее работу каким-то образом вмешиваются другие транзакции. При этом необходимо учитывать важное замечание,

что и сами транзакции, нарушающие ее работу, также рассматриваются как правильные (в соответствии с обычно принятым допущением), поэтому причиной получения общего неправильного результата становится неконтролируемое чередование операций двух транзакций, которые сами по себе являются правильными. Ниже перечислены три возникающие при этом проблемы.

- Проблема потерянного обновления.
- Проблема зависимости от незафиксированных результатов.
- Проблема анализа несовместимости.

Рассмотрим каждую из этих проблем по очереди.

Проблема потерянного обновления

Рассмотрим ситуацию, показанную на рис. 16.1. События, представленные на этом рисунке, развиваются следующим образом: транзакция А выполняет выборку некоторого кортежа t в момент времени tl, транзакция в выполняет выборку того же кортежа t в момент времени t2, транзакция А обновляет кортеж в момент времени t3 (с учетом тех значений, которые он имел во время tl), а транзакция в обновляет тот же кортеж в момент времени t4 (с учетом тех значений, которые он имел во время t2, которые оставались такими же, как и во время tl). В момент времени t4 происходит потеря обновления, внесенного транзакцией А, поскольку транзакция в перезаписывает его, даже не проверяя.

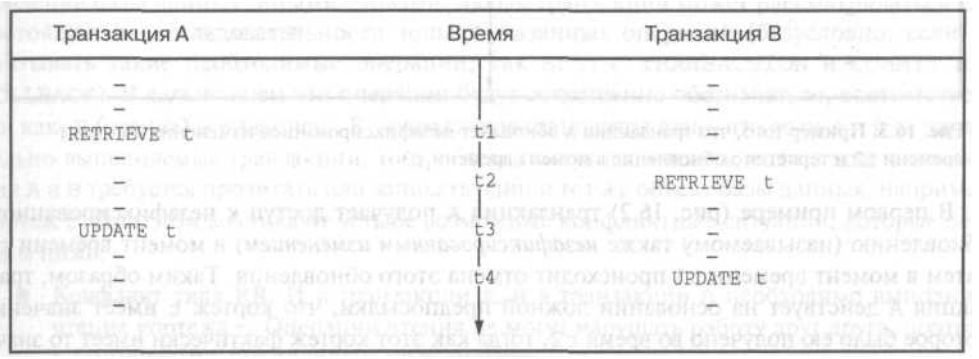


Рис. 16.1. Пример того, как транзакция А теряет обновление в момент времени t4

Примечание. Здесь и далее в этой главе принято соглашение об использовании термина *обновление кортежа* вместо *обновление значения переменной кортежа.*

Проблема зависимости от незафиксированных результатов возникает, если одной транзакции разрешено выполнять выборку (или, что еще хуже, обновление) кортежа, который был обновлен другой транзакцией, но это обновление еще не было зафиксировано другой транзакцией. А поскольку это обновление еще не зафиксировано, то всегда существует вероятность, что оно так и не будет зафиксировано, а вместо этого произойдет

откат, и в этом случае первая транзакция будет работать с данными, которых больше не существует (и которые в определенном смысле и не существовали). Рассмотрим в качестве примеров рис. 16.2 и 16.3.

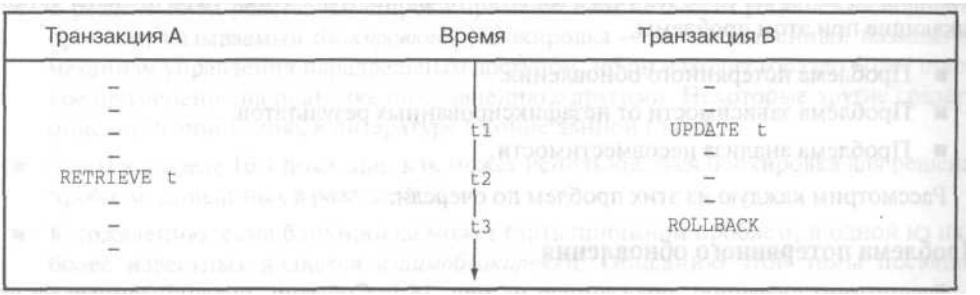


Рис. 16.2. Пример того, что транзакция A становится зависимой от незафиксированного обновления в момент времени t2

Рис. 16.3. Пример того, что транзакция A обновляет незафиксированное изменение в момент времени t2 и теряет это обновление в момент времени t3

В первом примере (рис. 16.2) транзакция A получает доступ к незафиксированному обновлению (называемому также *незафиксированным изменением)* в момент времени t2. Затем в момент времени t3 происходит отмена этого обновления. Таким образом, транзакция A действует на основании ложной предпосылки, что кортеж t имеет значение, которое было ею получено во время t2, тогда как этот кортеж фактически имеет то значение, которое он имел к моменту времени tl. В связи с этим транзакция A вполне может выработать неправильные результаты. Кстати, следует отметить, что откат транзакции в вполне может произойти не из-за ошибки в в; например, он может произойти из-за аварийного останова системы. (A если транзакция A к этому времени уже будет завершена, то такой останов не повлечет за собой выполнение отката также и для A.)

Во втором примере (рис. 16.3) события развиваются по еще более худшему сценарию. Транзакция A становится не только зависимой от незафиксированного изменения в момент времени t2, но и фактически теряет свое обновление во время t3, поскольку откат в момент времени t3 вызывает восстановление кортежа t до того значения, которое он имел к моменту времени tl, т.е. это еще один вариант проблемы потерянного обновления.

Проблема анализа несовместимости

Рассмотрим рис. 16.4, где показаны две транзакции, A и в, оперирующие с кортежами некоторого счета (ACCount — ACC). Транзакция A подсчитывает остатки на счетах, а транзакция B переводит сумму 10 со счета 3 на счет 1. Очевидно, что результат, выработанный транзакцией A, — 110, является неправильным; если бы в ходе своего дальнейшего выполнения транзакция A снова записала этот результат в базу данных, то фактически оставила бы базу данных в противоречивом состоянии[1]. По сути, транзакция A обнаружила несовместимое состояние базы данных и поэтому выполнила анализ несовместимости. Обратите внимание на различие между этим примером и предыдущим: в данном случае не возникает проблема зависимости транзакции A от незафиксированного изменения, поскольку в зафиксировала все свои обновления еще до того, как A прочитала значение ACC 3.

Примечание. Кстати, следует отметить, что вместо термина *анализ несовместимости* следовало бы применять *анализ неправильности.* Но мы по традиции придерживаемся прежнего термина.

Более подробное описание рассматриваемых проблем '

Примечание. Этот подраздел при первом чтении можно пропустить.

В данном подразделе описанные выше проблемы рассматриваются немного более подробно. Очевидно, что с точки зрения организации параллельной работы наибольший интерес представляют такие операции, как выборка информации из базы данных и обновление базы данных; иными словами, любая транзакция может рассматриваться как состоящая из последовательности только указанных операций (безусловно, если не учитывать такие необходимые операции, как BEGIN TRANSACTION И COMMIT ИЛИ ROLLBACK). В дальнейшем эти операции будут сокращенно обозначаться, соответственно, как R (чтение) и w (запись). В таком случае становится ясно, что если A и в — параллельно выполняемые транзакции, то проблемы могут возникнуть, если в ходе выполнения A и B требуется прочитать или записать один и тот же объект базы данных, например, кортеж t. При этом возникают четыре возможные конфликтные ситуации, которые описаны ниже.

- Конфликт типа RR. И в транзакции A, и в транзакции в необходимо выполнить чтение кортежа t. Операции чтения не могут нарушать работу друг друга, поэтому в данной ситуации проблема не возникает.
- Конфликт типа RW. В транзакции A выполняется чтение кортежа t, а затем в тран закции в возникает необходимость записать кортеж t. Если транзакции в будет разрешено выполнить эту запись, то (как показано на рис. 16.4) может возникнуть проблема анализа несовместимости, поэтому можно утверждать, что проблема анализа несовместимости вызвана конфликтами типа RW.

  Примечание. Если транзакция B выполняет свою операцию записи, а затем транзакция A снова считывает значение t, то последняя обнаруживает значение, отличное

[1] Если мы допускаем такую возможность (т.е. запись указанного результата обратно в базу данных), то, естественно, должны также допустить, что не заданы ограничения целостности, которые исключили бы возможность такой записи.

от того, что было прочитано раньше; такое стечение обстоятельств принято обозначать (не совсем точно) как **неповторяемое чтение,** поэтому проблема неповторяемого чтения также вызывается конфликтом типа RW.

- **Конфликт типа WR.** В транзакции A выполняется запись кортежа t, а затем в транзакции **в** возникает необходимость прочитать t. Если транзакции в будет разрешено выполнить эту операцию чтения, то (как показано на рис. 16.2, с учетом того, что транзакции A и B поменялись ролями) может возникнуть проблема зависимости от незафиксированных обновлений; таким образом, можно утверждать, что проблема зависимости от незафиксированных обновлений вызывается конфликтами типа WR.

  Примечание. Чтение в транзакции в, если оно будет разрешено, называется **грязным чтением** (dirty read).

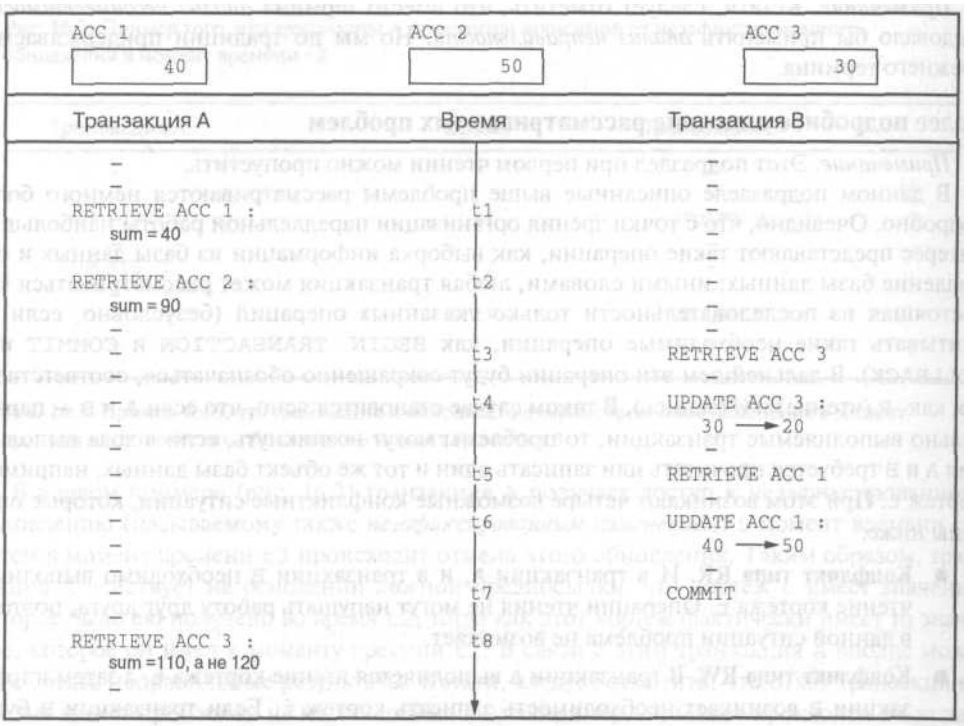


Рис. 16.4. Пример того, что транзакция A выполняет анализ несовместимости

- **Конфликт типа** WW. В транзакции A выполняется запись кортежа t, а затем необходимость выполнить запись кортежа t возникает в транзакции в. Если транзакции B будет разрешено выполнить эту запись, то (как показано на рис. 16.1, а также на рис. 16.3) может возникнуть проблема потерянного обновления; таким образом, можно утверждать, что проблема потерянного обновления вызвана конфликтами типа WW.

Примечание. Запись в транзакции в, если она будет разрешена, называется **грязной записью** (dirty write).

16.3. БЛОКИРОВКА

Как было указано в разделе 16.1, все проблемы, описанные в разделе 16.2, могут быть устранены с помощью механизма управления параллельным выполнением, называемого блокировкой. В его основе лежит простая идея — если для некоторой транзакции A требуется гарантия, чтобы определенный объект, в котором она заинтересована (как правило, кортеж базы данных), не изменился каким-то образом без ее ведома (как описано выше), она приобретает блокировку на этот объект (как принято называть соответствующую операцию). Неформально выражаясь, следствием приобретения блокировки является то, что к рассматриваемому объекту, условно говоря, "блокируются доступ других транзакций", и поэтому, в частности, предотвращается возможность внесения ими изменений. Благодаря этому транзакция A может продолжать свои операции обработки в полной уверенности в том, что рассматриваемый объект останется в определенном состоянии до тех пор, пока он требуется для этой транзакции.

Ниже приведено более подробное описание того, по какому принципу работает блокировка.

1. Прежде всего, предположим, что в системе поддерживаются блокировки двух ти пов: исключительные блокировки (блокировки X — exclusive) и разделяемые блоки ровки (блокировки S — shared), которые определены, как указано в следующих двух абзацах.

   Примечание. Блокировки X и S иногда именуются, соответственно, блокировками записи и блокировками чтения. Если не указано иное, то в данной главе предполагается, что блокировки X и S являются единственными доступными типами блокировок; описание других типов блокировок приведено в разделе 16.9. Кроме того, если не указано иное, предполагается, что единственными типами объектов базы данных, которые могут быть заблокированы, являются кортежи; описание других возможностей также приведено в разделе 16.9.
2. Если транзакция A владеет исключительной блокировкой (X), то запрос от неко торой другой транзакции в на получение блокировки кортежа t любого типа не может быть немедленно удовлетворен.
3. Если транзакция A владеет разделяемой блокировкой (S) кортежа t, то выполня ются следующие условия:

   - запрос некоторой другой транзакции в на получение блокировки X кортежа t не может быть немедленно удовлетворен;
   - запрос некоторой другой транзакции в на получение блокировки S кортежа t может и должен быть немедленно удовлетворен (это означает, что с этого вре мени транзакция в также будет владеть блокировкой S кортежа).

Эти правила можно успешно подытожить с помощью матрицы совместимости типов блокировок (рис. 16.5). Такая матрица интерпретируется следующим образом: рассмотрим некоторый кортеж t и предположим, что транзакция A в настоящее время владеет блокировкой на t, которая обозначена одной из записей в заголовках столбцов (дефис

обозначает, что блокировка отсутствует), а также предположим, что некоторая другая транзакция в выдает запрос на получение блокировки t, которая обозначена одной из записей в заголовках строк (для полноты здесь также предусмотрен случай "отсутствия блокировки"). В этой матрице "N" указывает на **конфликт** (запрос транзакции **в** не может быть немедленно удовлетворен), а "Y" указывает на **совместимость** (запрос транзакции в может и должен быть немедленно удовлетворен). Очевидно, что эта матрица является симметричной.

	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

Рис. 16.5. Матрица совместимости
'для блокировок типов X и S

Теперь перейдем к описанию **протокола доступа к данным,** или **протокола блокировки,**
который позволяет использовать только что описанные блокировки X и S для обеспечения того, чтобы никогда не возникали проблемы, описанные в разделе 16.2.

1. Транзакция, в которой требуется выполнить выборку кортежа, должна вначале приобрести блокировку S на этот кортеж.
2. Транзакция, в которой требуется выполнить обновление кортежа, должна вначале приобрести X блокировку на этот кортеж. В ином случае, если она уже владеет блокировкой S на этом кортеже, как происходит в ситуации, когда выполняется последовательность операций выборки и обновления (RETRIEVE и UPDATE), то эта транзакция должна, как принято называть такое действие, *расширить,* или *повысить уровень* блокировки S до уровня X.

   Примечание. В этот момент необходимо прервать изложение, чтобы пояснить, что запросы на получение блокировок обычно бывают неявными — операция *выборки кортежа* неявно запрашивает блокировку S на соответствующий кортеж, а операция *обновления кортежа* неявно запрашивает блокировку X (или неявно запрашивает расширение существующей блокировки S до уровня X) на соответствующий кортеж. Кроме того, в этом описании, как обычно, под операцией обновления подразумеваются любые операции вставки (INSERT) и удаления (DELETE), а также обновления (UPDATE) как таковые, но рассматриваемые правила требуют определенного уточнения с учетом особенностей операторов INSERT и DELETE. В данном разделе дополнительные сведения по этому вопросу не приведены.
3. Если запрос на блокировку от транзакции **в** не может быть немедленно удовлетво рен из-за того, что он конфликтует с блокировкой, которой уже владеет транзак ция А, то в переходит в **состояние ожидания.** Транзакция **в** ожидает до тех пор, по ка не появится возможность удовлетворить ее запрос на блокировку, а это может произойти не раньше, чем транзакцияА освободит блокировку.

   Примечание. Выше было применено выражение "не раньше, чем", поскольку после освобождения блокировки транзакцией А может быть удовлетворен другой запрос на

блокировку соответствующего кортежа, но это будет не запрос транзакции B, поскольку освобождения блокировки могут также ожидать другие транзакции. Безусловно, система должна обеспечить, чтобы транзакция в не ожидала до бесконечности (такая ситуация называется **активным тупиком** — livelock, или **истощением ресурсов** — starvation). Один из простых предотвращения такой ситуации состоит в том, чтобы запросы на блокировку обслуживались в порядке простой очереди ("первым поступил—первым обслуживается"). 4. Блокировки X освобождаются по завершении транзакции (COMMIT или ROLLBACK). Блокировки S также обычно освобождаются в это время (по крайней мере, такого предположения мы будем придерживаться до раздела 16.8).

Описанный выше протокол называется **строгим протоколом двухфазной блокировки.** Он рассматривается более подробно в разделе 16.6. В частности, в этом разделе указано, почему он получил такое название.

16.4. ДАЛЬНЕЙШЕЕ ОПИСАНИЕ ТРЕХ ПРОБЛЕМ ОРГАНИЗАЦИИ ПАРАЛЛЕЛЬНОЙ РАБОТЫ

Теперь мы имеем возможность рассмотреть, каким образом с помощью строгого протокола двухфазной блокировки решаются три проблемы, описанные в разделе 16.2. В этом разделе они снова рассматриваются по очереди.

Проблема потерянного обновления

Рис. 16.6 представляет собой модифицированную версию рис. 16.1. На нем показано, что произойдет при выполнении чередующихся операций данного рисунка в условиях применения строгого протокола двухфазной блокировки. Выполнение операции UPDATE транзакцией A в момент времени t3 не допускается, поскольку эта операция представляет собой неявный запрос на блокировку X кортежа t, а такой запрос конфликтует с блокировкой S, которой уже владеет транзакция в, поэтому A переходит в состояние ожидания.

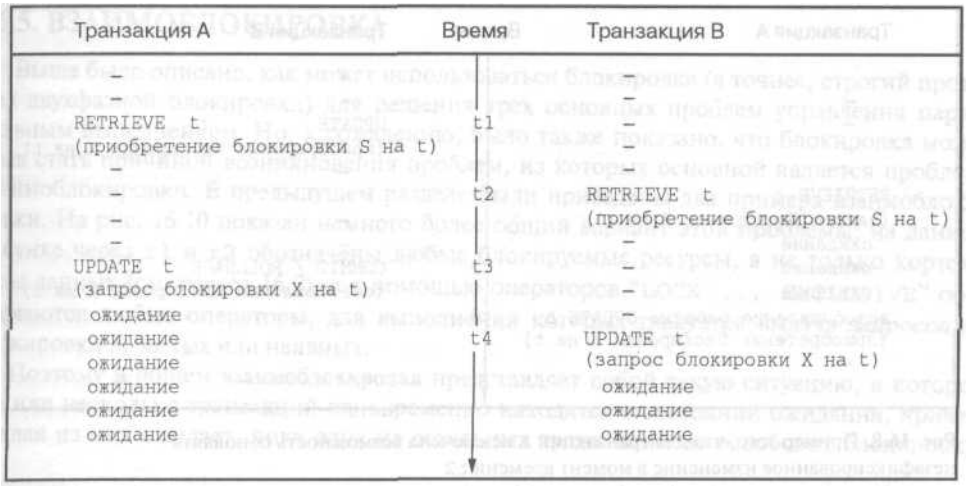


Рис. 16.6. Потеря обновления не определяется, но в момент времени t4 возникает

По аналогичным причинам транзакция в переходит в состояние ожидания в момент времени t4. При таких обстоятельствах обе транзакции не могут выполнять свои действия, поэтому проблема потери какого-либо обновления не возникает. Итак, проблема потерянного обновления свелась к другой проблеме! Но, по крайней мере, решена первоначальная проблема. Новая проблема называется *взаимоблокировкой,* и она рассматривается в разделе 16.5.

Проблема зависимости от незафиксированного обновления

На рис. 16.7 и 16.8, соответственно, приведены модифицированные версии рис. 16.2 и 16.3. На них показано, что произойдет при выполнении чередующихся операций, показанных на этих рисунках, в условиях применения строгого протокола двухфазной блокировки.

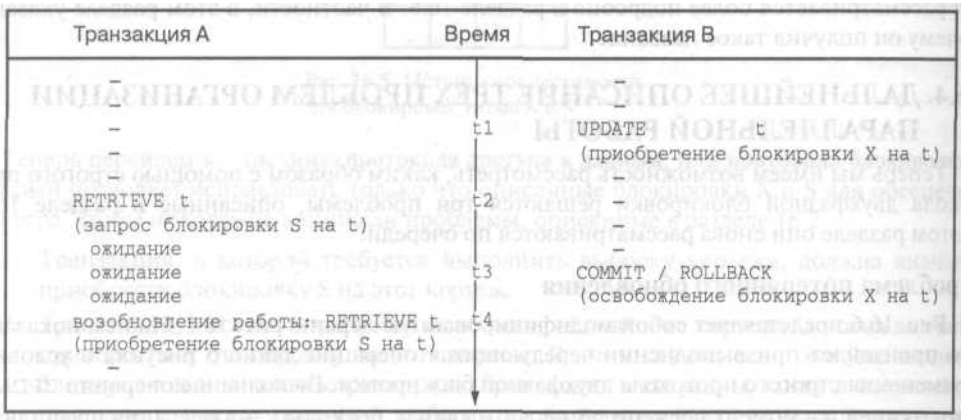


Рис. 16.7. Пример того, что для транзакции A исключена возможность считывать незафиксированное изменение в момент времени t2

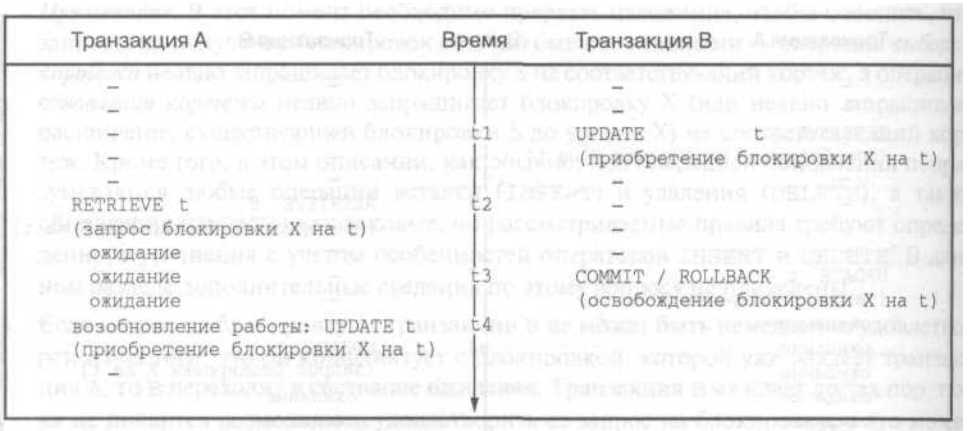


Рис. 16.8. Пример того, что для транзакции A исключена возможность обновлять незафиксированное изменение в момент времени t2

В обоих случаях выполнение операций транзакцией А в момент времени t2 (RETRIEVE на рис. 16.7 и UPDATE на рис. 16.8) не допускается, поскольку они представляют собой неявный запрос на блокировку кортежа t, а любой такой запрос конфликтует с блокировкой X, которой уже владеет транзакция в, поэтому А переходит в состояние ожидания. Эта транзакция остается в состоянии ожидания до тех пор, пока в не достигнет своего завершения (в результате выполнения либо оператора COMMIT, либо оператора ROLLBACK). После этого блокировка транзакции в освобождается и транзакция А получает возможность продолжить работу. В этот момент транзакция А обнаруживает уже зафиксированное значение (либо значение, предшествовавшее выполнению транзакции в, если произошел ее откат, либо значение, полученное после выполнения транзакции в, в ином случае). Так или иначе, успешное выполнение транзакции А больше не зависит от незафиксированного обновления, поэтому первоначальная проблема решена.

Проблема анализа несовместимости

На рис. 16.9 приведена модифицированная версия рис. 16.4. На этом рисунке показано, что произойдет при чередующемся выполнении операций данного рисунка в условиях применения строгого протокола двухфазной блокировки. Выполнение операции UPDATE транзакцией в в момент времени t6 не допускается, поскольку она представляет собой неявный запрос на блокировку X кортежа ACC 1, а такой запрос конфликтует с блокировкой S, которой уже владеет транзакция А, поэтому транзакция в переходит в состояние ожидания. Аналогичным образом, выполнение операции RETRIEVE транзакцией А в момент времени t7 также не допускается, поскольку она представляет собой неявный запрос на блокировку S кортежа ACC 3, а такой запрос конфликтует с блокировкой X, которой уже владеет транзакция в, поэтому транзакция А также переходит в состояние ожидания. Поэтому первоначальная проблема (в данном случае проблема анализа несовместимости) снова решена, но это привело к возникновению взаимоблокировки. Еще раз отметим, что взаимоблокировка рассматривается в разделе 16.5.

16.5. ВЗАИМОБЛОКИРОВКА

Выше было описано, как может использоваться блокировка (а точнее, строгий протокол двухфазной блокировки) для решения трех основных проблем управления параллельным выполнением. Но, к сожалению, было также показано, что блокировка может сама стать причиной возникновения проблем, из которых основной является проблема взаимоблокировки. В предыдущем разделе были приведены два примера взаимоблокировки. На рис. 16.10 показан немного более общий вариант этой проблемы; на данном рисунке через rl и r2 обозначены любые блокируемые ресурсы, а не только кортежи базы данных (см. раздел 16.9), а с помощью операторов "LOCK . . . EXCLUSIVE" обозначаются любые операторы, для выполнения которых требуется выдача запросов на блокировки X, явных или неявных.

Поэтому в общем взаимоблокировка представляет собой такую ситуацию, в которой две или несколько транзакций одновременно находятся в состоянии ожидания, причем каждая из них ожидает, пока одна из остальных транзакций не освободит блокировку,

чтобы получить возможность продолжить свою работу[2]. На рис. 16.10 показана взаимоблокировка, в которой участвуют две транзакции, но возможны также такие ситуации (по крайней мере, в принципе), когда во взаимоблокировке участвуют три, четыре или большее количество транзакций. Однако эксперименты, проведенные в системе System R, показывают, что на практике взаимоблокировки, в которых участвуют больше двух транзакций, почти никогда не возникают [16.9].

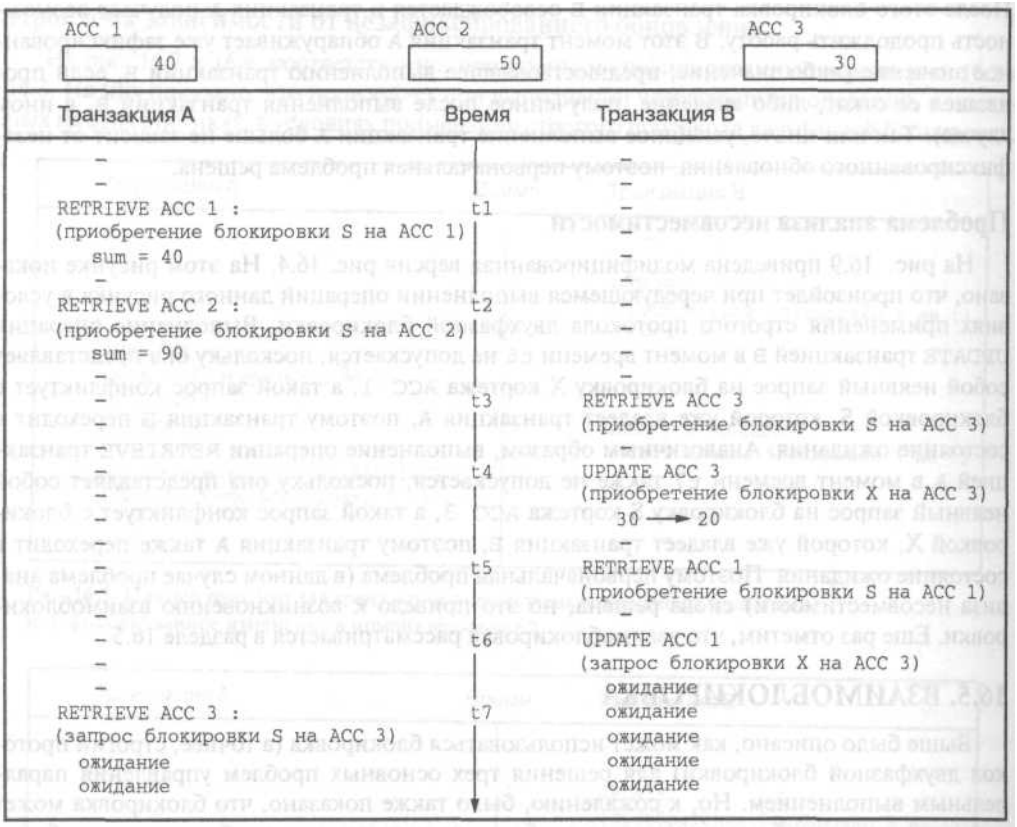


Рис. 16.9. Проблема анализа несовместимости исключается, но в момент времени t7 возникает взаимоблокировка

Если произошла взаимоблокировка, желательно, чтобы система обнаружила ее и разорвала. Для обнаружения взаимоблокировки необходимо определить наличие цикла в **графе ожидания** (Wait-For Graph). Так называется граф, который, неформально выражаясь, показывает "кто кого ожидает" (см. упр. 16.4). Чтобы разорвать взаимоблокировку, необходимо выбрать одну из транзакций, участвующих во взаимоблокировке (т.е. одну из транзакций, которые входят в состав цикла в графе ожидания), в качестве "жертвы" и

[2] Иногда в литературе для обозначения взаимоблокировки применяется также довольно красочный термин "смертельные объятья" (deadly embrace).

выполнить ее откат, что позволяет освободить ее блокировки и дать возможность другим транзакциям продолжить свою работу.

Примечание. Фактически на практике не все системы обнаруживают взаимоблокировки как таковые; в некоторых системах используется лишь механизм тайм-аутов и просто предполагается, что если некоторая транзакция не выполняет никакой работы в течение некоторого заранее установленного периода времени, то она находится в состоянии взаимоблокировки.

Рис. 16.10. Пример взаимоблокировки

Кстати, следует отметить, что *неудачное завершение* работы транзакции, выбранной в качестве жертвы, и последующий ее откат происходят не по вине самой этой транзакции. В некоторых системах осуществляется автоматический перезапуск такой транзакции с самого начала на основании предположения о том, что условия, которыми прежде было вызвано возникновение взаимоблокировки, вероятно, больше не повторятся. В других системах просто предусмотрена передача в ответ приложению кода исключения "deadlock victim" (жертва взаимоблокировки); выбор способа дальнейших действий для правильного выхода из этой ситуации предоставляется самому приложению. Разумеется, что с точки зрения прикладного программиста первый из этих двух подходов является более предпочтительным. А если даже сам программист иногда захочет принять участие в устранении указанной проблемы, он имеет для этого все возможности, но по очевидным причинам всегда желательно скрывать ее существование от конечного пользоштеля.

Предотвращение взаимоблокировок

Некоторые подходы к устранению взаимоблокировок основаны на том, что существует возможность модифицировать протокол блокировки различными способами, с тем чтобы полностью избежать взаимоблокировок, а не ожидать их возникновения, после чего их устранять (как это делается в большинстве систем). Один из подобных подходов кратко рассматривается в данном разделе. Этот подход реализован в двух версиях, называемых "ожидание-отмена" и "отмена-ожидание", и был впервые предложен для использования в распределенных системах [16.19], но может также применяться в централизованных системах. Краткое описание рассматриваемого подхода к предотвращению взаимоблокировок приведено ниже.

- Каждая транзакция обозначается отметкой времени ее начала (которая должна быть уникальной).
- Если транзакция А запрашивает блокировку на кортеже, который уже заблокиро ван транзакцией в, то выполняются описанные ниже действия в зависимости от применяемого варианта.

  - "Ожидание-отмена" (Wait-Die). Если выполнение транзакции А началось рань ше, чем в, А переходит в состояние ожидания; в противном случае происходит ее *отмена.* Это означает, что осуществляется откат и перезапуск транзакции А.
  - "Отмена—ожидание" (Wound-Wait). Если выполнение транзакции А началось позже, чем в, она переходит в состояние ожидания; в противном случае, она *отменяет* В. Это означает, что происходит откат и перезапуск транзакции В.

- Если транзакция должна быть перезапущена, ей после запуска присваивается ее первоначальная отметка времени.

Следует отметить, что первый компонент названия используемой версии *(ожидание* или *отмена)* в любом случае указывает, что произойдет, если выполнение транзакции А началось раньше, чем В. Вполне очевидно, что в версии "ожидание-отмена" множество всех ожидающих транзакций состоит из транзакций, начавшихся раньше и ожидающих завершения работы тех транзакций, которые были начаты позже, а в версии "отмена-ожидание" множество всех ожидающих транзакций состоит из транзакций, начавшихся позже и ожидающих завершения работы тех транзакций, которые были начаты раньше. Вполне очевидно, что при использовании любой из этих версий взаимоблокировка не может вообще возникнуть. Кроме того, можно легко убедиться в том, что в конечном итоге гарантировано успешное завершение работы каждой транзакции. Это означает, что возникновение активного тупика невозможно (ни одна из транзакций не должна ожидать до бесконечности), к тому же ни одна транзакция не должна перезапускаться снова и снова бесконечное количество раз. Основным недостатком этого подхода (при использовании любой из версий) является то, что в нем приходится выполнять слишком много откатов.

16.6. УПОРЯДОЧИВАЕМОСТЬ

Выше в данной главе была заложена основа для изучения крайне важного понятия упорядочиваемости, к чему мы теперь приступаем. Упорядочиваемость — это общепринятый *критерий правильной организации* чередующегося выполнения множества транзакций; иными словами, такая организация выполнения считается правильной тогда и только тогда, когда она является упорядочиваемой[3]. Любая конкретная процедура организации выполнения заданного множества транзакций является упорядочиваемой (а следовательно, *правильной)* тогда и только тогда, когда она эквивалентна некоторой процедуре последовательного выполнения тех же транзакций (т.е. гарантирует достижение таких

[3] В литературе фактически можно найти определения двух видов упорядочиваемости — упорядочиваемость по конфликтам (conflict serializability) и упорядочиваемость по просмотру (view serializability). Но понятие упорядочиваемости по просмотру почти не имеет практического значения и поэтому термин "упорядочиваемость" применяется как обозначение именно упорядочиваемости по конфликтам. Дополнительную информацию по этой теме можно, например, найти в [16.21].

же результатов). Пояснения к некоторым терминам, которые использовались в данном определении, приведены ниже.

- *Последовательным выполнением транзакций* называется такая организация их вы полнения, при которой транзакции выполняются одна за другой в некоторой по следовательности.
- Под *гарантированным достижением таких же результатов* подразумевается то, что рассматриваемый вариант чередующегося выполнения транзакций и вариант их последовательного выполнения всегда вырабатывают одни и те же результаты независимо от начального состояния базы данных.

Ниже дано дополнительное обоснование применяемого определения упорядочиваемого множества транзакций.

1. Предполагается, что отдельные транзакции являются правильными; это означает, что они по определению переводят базу данных из одного правильного состояния в другое правильное состояние, как описано в главе 15.
2. Таким образом, любая последовательность транзакций, на основании которой может быть организовано выполнение транзакций одна за другой, также является правильной (поскольку предполагается, что если отдельные транзакции не зависят друг от друга, то может быть выбрана *любая* последовательность их выполнения).
3. Поэтому имеет смысл ввести определение правильной организации чередующего ся выполнения операций отдельных транзакций, причем только такой организа ции выполнения, которая эквивалентна некоторой последовательной организации выполнения отдельных транзакций (т.е. организация выполнения может быть правильной тогда и только тогда, когда она является упорядочиваемой). В этой формулировке заслуживает особого внимания выражение "тогда и только тогда"! Дело в том, что некоторый вариант организации чередующегося выполнения мо жет быть неупорядочиваемым и при этом все равно приводить к результату, кото рый окажется правильным в некотором определенном начальном состоянии базы данных (см. упр. 16.3), но этого недостаточно, поскольку правильная организация чередующегося выполнения должна гарантироваться (т.е. быть независимой от конкретных состояний базы данных), а не возникать лишь по стечению обстоя тельств.

Снова обратившись к примерам, приведенным в разделе 16.2 (рис. 16.1-16.4), можно убедиться, что в каждом из рассматриваемых случаев проблема состояла именно в том, что применяемая процедура организации чередующегося выполнения транзакций не была упорядочиваемой. Это означает, что в этих вариантах выполнение "вначале А, затем В" или "вначале в, затем А" не приводит к получению одинаковых результатов. Кроме того, анализ материала, изложенного в разделе 16.4, под этим новым углом зрения показывает, что действие строгого протокола двухфазной блокировки сводится именно к тому, что он в каждом случае обеспечивает соблюдение принципов упорядочиваемости. На рис. 16.7 и 16.8 принятая организация чередующегося выполнения транзакций эквивалентна варианту "вначале в, затем А". На рис. 16.6 и 16.9 показано, что происходит взаимоблокировка, а это означает, что для одной из двух транзакций должен быть выполнен откат (и, предположительно, повторный запуск на выполнение в какой-то последующий

момент времени). Если будет выполнен откат транзакции А, то данный вариант организации чередующегося выполнения снова становится эквивалентным варианту "вначале в, затем А".

Терминология. Если дано множество транзакций, то любая процедура организации выполнения этих транзакций, с их чередованием или без чередования, называется **графиком.** Если транзакции выполняются одна за другой, без чередования, то применяемый при этом график называется **последовательным.** График, не являющийся последовательным, называется **чередующимся** (или просто *непоследовательным).* Два графика называются **эквивалентными** тогда и только тогда, когда они гарантируют выработку одних и тех же результатов, независимо от начального состояния базы данных. Таким образом, график называется *упорядочиваемым* и *правильным* тогда и только тогда, когда он эквивалентен некоторому последовательному графику.

Следует отметить, что два разных последовательных графика, в которых участвуют одни и те же транзакции, вполне могут вырабатывать разные результаты, и это при том, что оба графика являются правильными. Это означает, что два разных чередующихся правильных графика, в которых участвуют одни и те же транзакции, также могут вырабатывать различные результаты. В качестве примера рассмотрим транзакцию А, смысл которой сводится к тому, чтобы "сложить 1 с х", и транзакцию в, которая имеет своей целью "удвоить х" (где х — некоторый элемент в базе данных). Предположим также, что начальное значение х равно 10. В таком случае последовательный график "А, затем в" приводит к получению результата х = 22, а последовательный график "вначале в, затем А" приводит к получению результатах = 21. Оба эти результата в равной степени являются правильными, и поэтому любой график, который гарантирует эквивалентность последовательным графикам "вначале А, затем В" или "вначале в, затем А", также является правильным.

Понятие упорядочиваемое™ было впервые введено (хотя и не под таким названием) Эсвараном (Eswaran) и др. в [16.6]. Кроме того, в той же статье была доказана важная теорема, называемая **теоремой двухфазной блокировки,** которую мы будем рассматривать в приведенной ниже формулировке[4].

Если все транзакции подчиняются протоколу двухфазной блокировки, то все возможные чередующиеся графики являются упорядочиваемыми.

Протокол двухфазной блокировки, в свою очередь, формулируется следующим образом:

- прежде чем выполнить операцию с любым объектом (например, кортежем базы данных), транзакция должна приобрести блокировку на этот объект;
- после освобождения любой блокировки транзакция больше не должна приобре тать какие-либо дополнительные блокировки.

Таким образом, любая транзакция, которая подчиняется этому протоколу, имеет две фазы: фаза приобретения блокировок (или фаза *расширения)* и фаза освобождения блокировок (или фаза *сужения).*

[4] Двухфазная блокировка не имеет ничего общего с двухфазной фиксацией; просто эти два термина имеют внешнее сходство.

Примечание. На практике фаза сужения часто сводится к единственной операции COMMIT или ROLLBACK, выполняемой в конце транзакции (к этой теме мы вернемся в разделах 16.7 **и** 16.8). Если соблюдается такое условие, то данный протокол становится равносильным *строгой* версии, которая была описана в разделе 16.3.

Рассматриваемое здесь понятие упорядочиваемости оказывает значительную помощь **в** анализе этой области, которая может стать источником многих затруднений, поэтому в данном разделе приведено еще несколько дополнительных замечаний на эту тему. Предположим, что I — чередующийся график, который охватывает некоторое множество транзакций T1, T2, . . ., тп. Если график I является упорядочиваемым, то существует по меньшей мере один последовательный график S, который включает множество транзакций T1, T2, . . ., Тп, такой что I эквивалентен S. График s называется **упорядочением** графика I.

Теперь предположим, что Ti и tj — две любые разные транзакции в множестве T1, T2, . . ., Тп. Допустим, что Ti предшествует Tj в упорядочении S. Поэтому в чередующемся графике I достигнутый эффект организации чередующегося выполнения должен быть таким, как если бы транзакция Ti действительно выполнялась перед Tj. Иными словами, неформальной, но очень удобной характеристикой упорядочиваемости является соблюдение того требования, что если А и в — любые две транзакции, участвующие в некотором упорядочиваемом графике, то в этом графике либо А логически предшествует в, либо в логически предшествует А; это означает, что **либо в может воспользоваться результатами выполнения А, либо А — результатами выполнения в.** (Если А вырабатывает выходные результаты x, y, . . . , z и в получает в качестве входных любые из данных x, y, . . ., z, то в в любом случае получает эти данные либо так, как если бы все они были выведены транзакцией А, либо все они были такими, как перед выводом из транзакции А (т.е. перед ее выполнением), а не в виде смеси результатов этих двух типов.) И наоборот, если эффект действия чередующегося графика не сводится к такой ситуации, как если бы А выполнялась перед в или В выполнялась перед А, то этот график нельзя считать упорядочиваемым и правильным.

Наконец, необходимо подчеркнуть такую мысль, что если некоторая транзакция А не является двухфазной (т.е. не подчиняется протоколу двухфазной блокировки), то **всегда** можно сформировать некоторую другую транзакцию в, которая может выполняться, чередуясь с А таким образом, что в результате будет получен общий график, не являющийся упорядочиваемым и правильным. Отметим, что в целях уменьшения конкуренции за ресурсы и повышения за счет этого производительности и пропускной способности, в системах, применяемых на практике, обычно разрешено формирование транзакций, не являющихся двухфазными; это означает, что транзакции могут преждевременно *освобождать блокировки* (до выполнения оператора COMMIT), а затем приобретать другие блокировки. Но должно быть вполне очевидно, что предложение использовать такие транзакции является довольно рискованным; при этом, допуская применение транзакция А, не являющейся двухфазной, можно лишь надеяться на то, что в системе не будет выполняться одновременно с А транзакция в, которая обращается к тем же данным (поскольку при возникновении указанной ситуации система становится способной вырабатывать неправильные ответы).

16.7. ДАЛЬНЕЙШЕЕ ОПИСАНИЕ ПРОБЛЕМЫ ВОССТАНОВЛЕНИЯ

Если применяется последовательный график, то очевидно, что отдельные транзакции в нем являются восстанавливаемыми, поскольку любая из таких транзакций в случае необходимости всегда может быть отменена и/или выполнена повторно с использованием методов, описанных в предыдущей главе. Но далеко не очевидно, продолжают ли транзакции оставаться восстанавливаемыми, если для них разрешено выполнение с чередованием операций. В действительности проблема зависимости от незафиксированного выполнения, описанная в разделе 16.2, может стать причиной нарушения восстанавливаемости, как будет показано в данном разделе.

На время предположим, что не применяется никакой протокол блокировки (как и в разделе 16.2) и поэтому, в частности, что транзакции не должны никогда ожидать для того, чтобы приобрести блокировку. Теперь рассмотрим рис. 16.11, который представляет собой модифицированную версию рис. 16.2 (различие между ними состоит в том, что теперь транзакция А выполняет фиксацию до того, как произойдет откат транзакции в). Проблема здесь заключается в следующем: для того чтобы можно было выполнить запрос ROLLBACK транзакции в и возвратить базу данных в такое состояние, как если бы в вообще не выполнялась, необходимо также выполнить откат транзакции А, поскольку в ней использовалось одно из обновлений транзакции в. Но откат транзакции А невозможен, поскольку А уже зафиксирована. Таким образом, график, показанный на этом рисунке, является невосстановимым.

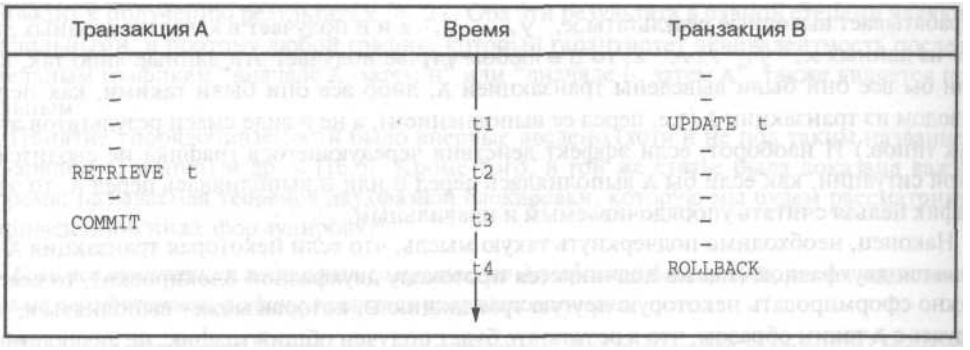


Рис. 16.11. Пример невосстановимого графика

Ниже приведено условие достаточности того, чтобы график был восстановимым [15.2].

Если в транзакции А используются какие-либо из обновлений транзакции в, то фиксация А не должна осуществляться до завершения работы в.

Очевидно, что применяемый механизм управления параллельной работой (а если для организации параллельной работы используется блокировка, то протокол блокировки) должен гарантировать, чтобы все графики были восстановимыми в указанном смысле.

Но на изложенном выше требовании не исчерпываются все задачи, стоящие перед этим протоколом. Теперь предположим, что протокол блокировки введен в действие, но в рассматриваемом протоколе используется нестрогая форма двухфазной блокировки, согласно которой транзакция может освобождать блокировки еще до своего завершения.

Далее рассмотрим рис. 16.12, который представляет собой модифицированную версию рис. 16.11 (различия между ними состоят в том, что в данном случае в транзакции A не выполняется фиксация до завершения транзакции B, но транзакция в освобождает свою блокировку кортежа t *преждевременно).* Как показано на рис. 16.11, для того, чтобы выполнить запрос транзакции в на выполнение операции ROLLBACK и вернуться к такой ситуации, как если бы в никогда не выполнялась, необходимо произвести также откат транзакции A, поскольку в ней использовалось одно из обновлений транзакции B. Более того, может быть осуществлен и откат транзакции A, поскольку A еще не была зафиксирована. Но почти наверняка применение каскадных откатов, осуществляемых таким образом, является нежелательным; в частности, вполне очевидно, что если допускается каскадное распространение отката одной транзакции, которое влечет за собой откат другой транзакции, то можно быть готовым к тому, что придется столкнуться с *цепочками каскадных откатов* произвольной длины. Иными словами, недостаток графика, показанного на этом рисунке, состоит в том, что он не гарантирует отсутствия каскадных откатов.

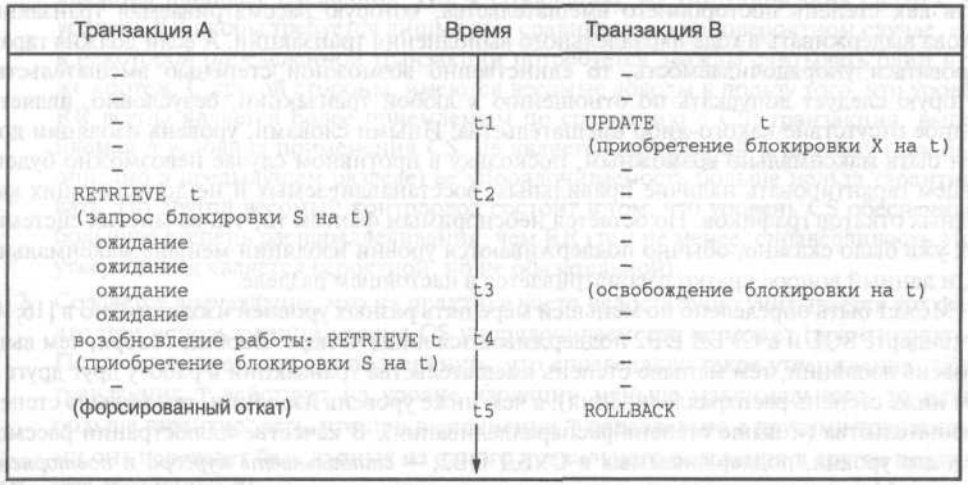


Рис. 16.12. Пример графика, который может стать причиной каскадного отката

Ниже приведено условие достаточности того, чтобы график гарантировал отсутствие каскадных откатов [15.2].

Если в транзакции A используются какие-либо обновления транзакции B, то транзакция A не должна использовать эти данные до завершения транзакции B.

Строгий протокол двухфазной блокировки гарантирует выполнение всех графиков без каскадных откатов. Именно поэтому этот протокол используется в подавляющем большинстве систем[5]. Кроме того, можно легко убедиться в том, что график, гарантирующий отсутствие каскадных откатов, должен также обязательно быть восстановимым графиком, в том смысле этого термина, который был определен выше.

[5] Фактически строгий протокол двухфазной блокировки в определенной степени является слишком строгим, поскольку на практике нет необходимости удерживать блокировки S до конца транзакции, при условии, что сама транзакция продолжает оставаться двухфазной.

16.8. УРОВНИ ИЗОЛЯЦИИ

Упорядочиваемость гарантирует изолированность транзакций, в той трактовке этого термина, которая применяется при описании свойств ACID. Одним из непосредственных и весьма благоприятных следствий из этого факта является то, что если все графики — упорядочиваемые, то прикладной программист, разрабатывая код для любой конкретной транзакции А, не должен обращать абсолютно никакого внимания на тот факт, что одновременно с ней в системе может выполняться некоторая другая транзакция в. Но можно также утверждать, что протоколы, применяемые для обеспечения упорядочиваемое™, снижают степень распараллеливания, или общую производительность системы, до неприемлемых уровней. Поэтому на практике в системах обычно поддерживаются целый ряд других уровней "изоляции" (этот термин заключен в кавычки, поскольку любой уровень ниже максимального означает, что транзакция в конечном итоге не является в полном смысле слова изолированной от других, как показано ниже).

Уровень изоляции применительно к данной конкретной транзакции можно определить как степень постороннего вмешательства, которую рассматриваемая транзакция готова выдерживать в ходе параллельного выполнения транзакций. А если должна гарантироваться упорядочиваемость, то единственно возможной степенью вмешательства, которую следует допускать по отношению к любой транзакции, безусловно, является полное отсутствие какого-либо вмешательства! Иными словами, уровень изоляции должен быть максимально возможным, поскольку в противном случае невозможно будет в общем гарантировать наличие правильных, восстанавливаемых и не допускающих каскадных откатов графиков. Но остается неоспоримым фактом то, что во многих системах, как уже было сказано, обычно поддерживаются уровни изоляции меньше максимального, и данный вопрос кратко рассматривается в настоящем разделе.

Может быть определено по меньшей мере пять разных уровней изоляции, но в [16.10], в стандарте SQL и в СУБД DB2 поддерживаются лишь четыре. Вообще говоря, чем выше уровень изоляции, тем меньше степень вмешательства транзакций в работу друг друга (и тем ниже степень распараллеливания), а чем ниже уровень изоляции, тем больше степень вмешательства (и выше степень распараллеливания). В качестве иллюстрации рассмотрим два уровня, поддерживаемых в СУБД DB2, — *стабильность курсора* и *повторяемое чтение.* Максимальным уровнем является повторяемое чтение (Repeatable Read — RR); если все транзакции действуют на этом уровне, то все графики становятся упорядочиваемыми. В отличие от этого, при использовании другого уровня, а именно стабильности курсора (Cursor Stability — CS) соблюдаются следующие условия:

- если транзакция А получает возможность обратиться к некоторому кортежу[6] t и в результате
- приобретает блокировку на t, а затем
- освобождает структуры адресации, применяемые для доступа к" кортежу t без его обновления, и поэтому

---

[6] Такая возможность в транзакции появляется благодаря такой установке курсора, когда он указывает на кортеж (как было описано в главе 4), отсюда и происходит название "стабильность курсора". Но в целях уточнения необходимо указать, что в СУБД DB2 приобретаемая блокировка T1 кортежа t фактически является блокировкой "обновления" (update — U), а не блокировкой S (см. [4.21]).

- не расширяет свою блокировку до уровня X, то
- данная блокировка может быть освобождена без ожидания конца транзакции.

Но следует учитывать, что теперь некоторая другая транзакция в может обновить кортеж t и зафиксировать изменение. Если в дальнейшем транзакция A возвратится на некоторый предыдущий этап своей работы и снова прочитает кортеж t (следует отметить, что при этом происходит нарушение протокола двухфазной блокировки!), то обнаружит внесенное изменение и поэтому фактически столкнется с несовместимым состоянием базы данных. С другой стороны, при повторяемом чтении (RR) все блокировки кортежей (а не только блокировки X) удерживаются до конца транзакции и поэтому только что указанная проблема не возникает.

Из этого следуют приведенные ниже выводы.

1. Указанная проблема является не единственной, которая может возникнуть при использовании уровня CS; мы здесь привели ее описание лишь потому, что она является наиболее очевидной. Но, к сожалению, из этого описания следует, что уровень RR якобы требуется лишь в том сравнительно маловероятном случае, если в некоторой определенной транзакции потребуется дважды считывать один и тот же кортеж. С другой стороны, имеются весомые доводы в пользу того, что уровень RR всегда является более приемлемым по сравнению с CS; транзакция, выпол няемая в условиях применения CS, не является двухфазной и поэтому (как было описано в предыдущем разделе) ее упорядочиваемость больше нельзя гарантиро вать. Но другой весомый контрдовод состоит в том, что уровень CS обеспечивает большую степень распараллеливания, чем RR (тем не менее, справедливость этого утверждения является вероятной, но не обязательной).
2. Создается впечатление, что на практике часто недостаточно учитывается тот факт, что при использовании уровня CS упорядочиваемость не может гарантироваться. Поэтому следует еще раз подчеркнуть, что справедливо такое утверждение: "Если транзакция т действует на уровне изоляции меньше максимального, то нельзя больше гарантировать, что при выполнении т параллельно с другими транзакция ми она переведет базу данных из одного правильного состояния в другое правиль ное состояние".
3. В любой реализации, которая поддерживает любой уровень изоляции ниже мак симального, должны быть, как правило, предусмотрены некоторые средства яв ного управления параллельностью, позволяющие пользователям создавать свои приложения так, чтобы гарантировалась безопасность эксплуатации базы данных в отсутствие подобных гарантий со стороны самой системы (обычно явно задан ные операторы LOCK). Например, в СУБД DB2 предусмотрен явно заданный опе ратор LOCK TABLE, который дает возможность пользователям, работающим на уровне меньше максимального, приобретать явные блокировки, превосходящие те, которые приобретаются в DB2 автоматически для соблюдения требований те кущего уровня. (Кстати, следует отметить, что в стандарте SQL не предусмотрены подобные явно заданные механизмы управления параллельностью, как описано в разделе 16.11.)

Кроме того, необходимо учитывать, что приведенное выше описание уровня RR как максимального уровня изоляции относится к реализации повторяемого чтения в СУБД

DB2. К сожалению, в стандарте SQL термин *повторяемое чтение* используется для обозначения уровня изоляции, который является строго более низким, чем максимальный уровень (об этом также сказано в разделе 16.11).

Фантомы

Одной из особых проблем, которая может возникнуть, если транзакции действуют на уровне изоляции меньше максимального, является так называемая *проблема фантомов*. Рассмотрим приведенный ниже пример (он является довольно надуманным, но вполне позволяет проиллюстрировать рассматриваемое понятие).

- Прежде всего, предположим, что в транзакции A вычисляется средний остаток на всех счетах, принадлежащих клиенту Джо. Допустим, что в настоящее время имеются три таких счета и на каждом из них остаток составляет 100 долл. Поэтому транзакция A просматривает три счета, в ходе своего выполнения приобретает на них разделяемые блокировки и получает результат (100 долл.).
- А теперь предположим, что выполняется параллельная транзакция в, в результате которой в базу данных вводится еще один счет клиента Джо, с остатком 200 долл. Для определенности допустим, что новый счет вводится после того, как в транзакции A было вычислено среднее значение, равное 100 долл. Предположим также, что сразу же после введения нового счета в транзакции в происходит фиксация (и освобождение исключительной блокировки нового счета, которой она владела).
- Затем допустим, что было принято решение снова воспользоваться транзакцией A для просмотра счетов клиента Джо, подсчета их количества и суммирования их остатков, а затем деления суммы остатков на количество счетов (возможно, потому что пользователь захотел определить, действительно ли полученное раньше среднее значение равно сумме, деленной на количество). На данный момент обнаруживаются четыре счета вместо трех, а полученный результат равен 125 долл. вместо 100 долл.!

Итак, в данном случае в обоих транзакциях применялась строгая двухфазная блокировка и все же случилось что-то неправильное, а именно: в транзакции A было обнаружено то, чего не существовало при первом ее выполнении, — фантом. Вследствие этого упорядочиваемость была нарушена (очевидно, что в данном случае чередующееся выполнение не эквивалентно ни последовательности "вначале A, затем в", ни последовательности "вначале в, затем A").

Но заслуживает особого внимания то, что рассматриваемая здесь проблема не имеет ничего общего с двухфазной блокировкой как таковой. Скорее, эта проблема состоит в том, что в транзакции A не было заблокировано то, что логически должно было быть заблокировано; вместо блокировки лишь трех имеющихся счетов клиента Джо в ней фактически нужно было заблокировать множество всех счетов, принадлежащих Джо или, иными словами, счетов, соответствующих предикату "владелец счета — Джо" (см. [16.6] и [16.13])[7]. Если бы можно было обеспечить такую возможность, то транзакции в пришлось бы перейти в состояние ожидания после осуществления в ней попытки ввести новый

[7] Эту мысль можно также выразить таким образом, что в транзакции A необходимо было заблокировать саму возможность отрицания того факта, что Джо не принадлежат какие-либо иные счета, кроме рассматриваемых.

счет (поскольку в, безусловно, пришлось бы затребовать блокировку на этот новый счет, а такая блокировка вошла бы в противоречие с блокировкой, принадлежащей транзакции А). Хотя по причинам, описанным в [15.12], в большинстве современных систем не поддерживается блокировка предикатов как таковая, в них все равно, как правило, удается избежать возникновения *фантомов* благодаря блокировке *пути доступа,* который используется для обращения к рассматриваемым данным. Например, если в случае использования тех счетов, которые принадлежат Джо, путем доступа является индекс на имени клиента, то система может заблокировать запись в этом индексе, относящуюся к клиенту Джо. Такая блокировка предотвращает возникновение фантомов, поскольку для создания такого фантома требуется обновление пути доступа (в данном примере записи индекса) и поэтому приобретение блокировки X на такой путь доступа. Дополнительные сведения на эту тему приведены в [15.12].

16.9. НАМЕЧЕННЫЕ БЛОКИРОВКИ

До этого времени в основном предполагалось, что единицей измерения объема данных, применяемых в целях блокировки, является отдельный кортеж. Но, в принципе, нет никаких оснований, по которым блокировки нельзя было бы применять к большим или меньшим единицам данных, например, ко всей переменной отношения, или даже ко всей базе данных, или (переходя в другую крайность) к отдельному компоненту конкретного кортежа. В данном случае речь идет о степени детализации **блокировки** [16.10], [16.11]. Как и обычно, здесь приходится идти на компромисс, поскольку чем тоньше детализация, тем больше степень распараллеливания, а чем она грубее, тем меньше блокировок приходится устанавливать и проверять, что способствует также снижению издержек. Например, если транзакция установила блокировку X на всю переменную отношения, то нет необходимости устанавливать блокировки X на отдельных кортежах в этой переменной отношения; с другой стороны, ни одна из параллельно выполняемых транзакций вообще не сможет получить каких-либо блокировок на данной переменной отношения или на кортежах в этой переменной отношения.

Предположим, что некоторая транзакция т фактически запрашивает блокировку X на некоторой переменной отношения R. После получения запроса от т система должна иметь возможность определить, имеют ли уже какие-либо другие транзакции блокировку на любом кортеже R; если так оно и есть, то запрос транзакции т в настоящее время не может быть удовлетворен. Но может ли система обнаружить подобный конфликт? Безусловно, нежелательно, чтобы приходилось проверять каждый кортеж в переменной отношения R для определения того, является ли какой-либо из них в настоящее время заблокированным какой-то другой транзакцией, или рассматривать все существующие блокировки, чтобы узнать, не относится ли какая-либо из них к одному из кортежей в переменной отношения R. Вместо этого вводится **протокол намеченной блокировки,** согласно которому ни одной транзакции не разрешается приобрести блокировку на кортеже перед тем, как будет вначале приобретена блокировка (а, возможно, лишь намечена такая блокировка, как описано в следующем абзаце) на переменную отношения, которая ее содержит. Поэтому в данном примере обнаружение конфликтов становится сравнительно простой задачей и сводится к определению того, имеет ли какая-либо транзакция конфликтующую блокировку на уровне переменной отношения.

Итак, фактически изложенное выше уже показывает, что блокировки X и S имеют смысл не только для отдельных кортежей, но и для целых переменных отношения. Согласно рекомендациям, изложенным в [16.10], [16.11], введем три дополнительных типа блокировок, называемых **намеченными блокировками,** которые также имеют смысл для переменных отношения, но не для отдельных кортежей: **намеченные разделяемые** (Intent Shared — IS) блокировки, **намеченные исключительные** (Intent Exclusive — IX) блокировки и **разделяемые намеченные исключительные** (Shared Intent Exclusive — SIX) блокировки. Эти новые типы блокировок могут быть определены неформально, как описано ниже. (Предполагается, что транзакция т затребовала блокировку указанного типа на переменной отношения R; для полноты включены также определения типов X и S.)

- **Намеченная разделяемая блокировка (IS). В** транзакции т намечается установка блокировок S на отдельных кортежах переменной отношения R для того, чтобы гарантировать постоянство этих кортежей в процессе их обработки.
- **Намеченная исключительная блокировка (IX).** То же, что и **IS,** наряду с тем, что т может обновлять отдельные кортежи в R и поэтому устанавливать блокировки X на этих кортежах.
- **Разделяемая блокировка** (S). Транзакция т может допускать параллельное приме нение других транзакций чтения, но не параллельное применение других транзак ций обновления в переменной отношения R (сама транзакция т не обновляет ни одного из кортежей BR).
- **Разделяемая намеченная исключительная блокировка (SIX).** Представляет собой сочетание S и IX; это означает, что т допускает присутствие параллельно выполняемых транзакций чтения, но не параллельно выполняемых транзакций обновления в R, но наряду с этим в транзакции т могут обновляться отдельные кортежи в R и поэтому должны устанавливаться блокировки X на эти кортежи.
- **Исключительная блокировка (X).** Транзакция т вообще не допускает выполнения какого-либо параллельного с ней доступа к R (в самой транзакции Т отдельные кортежи R могут обновляться или не обновляться).

Формальные определения этих пяти типов блокировок иллюстрируются расширенной версией матрицы совместимости типов блокировок, которая впервые рассматривалась в разделе 16.3 (рис. 16.13).

	X	SIX	IX	S	IS	—
X	N	N	N	N	N	Y
SIX	N	N	N	N	Y	Y
IX	N	N	Y	N	Y	Y
S	N	N	N	Y	Y	Y
IS	N	Y	Y	Y	Y	Y
—	Y	Y	Y	Y	Y	Y

Рис. 16.13. Матрица совместимости, расширенная с учетом намеченных блокировок

Ниже приведена более точная формулировка определения *протокола намеченной блокировки.*

1. Прежде чем любая конкретная транзакция сможет приобрести блокировку S на указанном кортеже, она должна вначале приобрести блокировку IS или более сильную блокировку (как описано ниже) на переменную отношения, содержащую этот кортеж.
2. Прежде чем любая конкретная транзакция сможет приобрести блокировку X на указанном кортеже, она должна вначале приобрести блокировку IX или более сильную блокировку (как описано ниже) на переменную отношения, содержащую этот кортеж.

(Но следует отметить, что это — еще не полное определение. См. аннотацию к [16.10].) Термин *сильная блокировка,* применяемый в приведенном выше определении протокола, который полностью формулируется как *относительно более сильная блокировка,* можно объяснить следующим образом. Рассмотрим **граф предшествования,** приведенный на рис. 16.14. Принято считать, что блокировка типа L2 сильнее (т.е. находится выше в этом графе), чем блокировка типа L1 тогда и только тогда, когда при наличии обозначения "N" (конфликт) в столбце блокировки L1 в матрице совместимости для заданной строки имеется также обозначение "N" в столбце блокировки L2 для той же строки (см. рис. 16.13). Следует отметить, что запрос на блокировку, оканчивающейся неудачей при использовании блокировки определенного типа, безусловно, также окончится неудачей при запросе блокировки более сильного типа (и из этого факта следует, что всегда безопасно использовать блокировки такого типа, который является более сильным по сравнению со строго необходимым). Заслуживает также внимания то, что ни блокировка S, ни блокировка IX не являются более сильными по сравнению с другими.

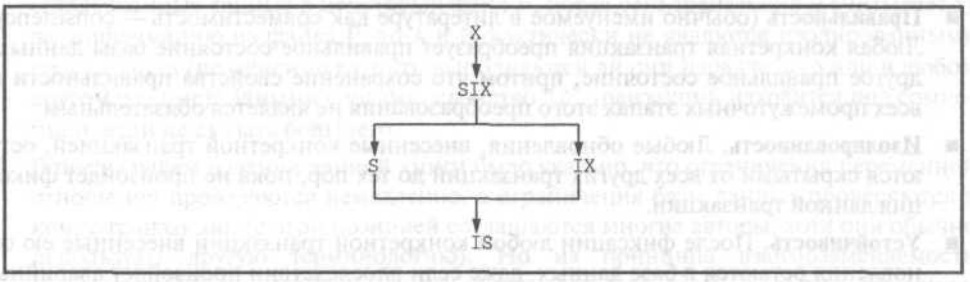


Рис. 16.14. Граф предшествования типов блокировок

Необходимо также указать, что на практике блокировки переменных отношения, приобретаемые с помощью протокола намеченной блокировки, обычно приобретаются неявно. Например, если речь идет о транзакциях, предусматривающих только чтение, то система по всей вероятности должна неявно приобретать блокировки IS на всех переменных отношения, к которым получают доступ эти транзакции. А если речь идет о транзакциях обновления, то система вместо этого должна, по-видимому, приобретать блокировки IX. Но в системе, вероятно, должен быть предусмотрен явно заданный оператор LOCK определенного типа для того, чтобы была возможность приобретать в транзакциях

блокировки S, X или SIX на уровне переменной отношения, если они потребуются. В частности, такой оператор поддерживается в СУБД DB2 (только для блокировок S и X, а не для блокировок SIX).

В завершение этого раздела отметим, что во многих системах предусмотрена возможность эскалации блокировок. Это средство представляет собой попытку достичь равновесия между конфликтующими требованиями повышения степени распараллеливания и снижения издержек на управление блокировками. Основная его идея состоит в том, что после достижения некоторого заранее заданного порогового значения система автоматически заменяет коллекцию блокировок с тонкой степенью детализации единственной блокировкой с более грубой детализацией, например, путем оценки мощности множества отдельных блокировок S уровня кортежа и преобразования в блокировку S той блокировки IS, которая установлена на переменной отношения, содержащей отдельных кортежи с блокировками S. Создается впечатление, что этот метод достаточно хорошо проявляет себя на практике [16.9].

16.10. КРИТИКА ПОДХОДА, ОСНОВАННОГО НА ИСПОЛЬЗОВАНИИ СВОЙСТВ ACID

Как было сказано в главе 15, в настоящей главе будет продолжено описание свойств ACID транзакций. В действительности, у автора сложилось довольно нестандартное мнение по этой теме, как вскоре станет очевидным.

Вначале напомним, что ACID — это сокращенное обозначение таких свойств транзакций, как неразрывность, правильность, изолированность и устойчивость (atomicity-correctness-isolation-durability). Ниже эти свойства кратко описаны повторно.

- Неразрывность. Любая конкретная транзакция действует по принципу "все или ничего".
- Правильность (обычно именуемое в литературе как совместимость — consistency). Любая конкретная транзакция преобразует правильное состояние базы данных в другое правильное состояние, притом что сохранение свойства правильности на всех промежуточных этапах этого преобразования не является обязательным.
- Изолированность. Любые обновления, внесенные конкретной транзакцией, остаются скрытыми от всех других транзакций до тех пор, пока не произойдет фиксация данной транзакции.
- Устойчивость. После фиксации любой конкретной транзакции внесенные ею обновления остаются в базе данных, даже если впоследствии произойдет аварийный останов системы.

Итак, на первый взгляд за сокращением ACID стоят весьма желаемые свойства, но так ли действительно выглядят представляемые ими понятия при более внимательном исследовании? В данном разделе приведены некоторые свидетельства, которые показывают, что в общем ответ на этот вопрос является отрицательным.

Немедленная проверка ограничений

Начнем с того, что может показаться отступлением от основной темы, — с обоснования нашей позиции, впервые явно сформулированной в главе 9, о том, что все ограничения целостности должны проверяться немедленно (т.е. по окончании выполнения оператора), а

не откладываться до конца транзакции. В основе этой позиции лежат по меньшей мере четыре причины, которые описаны ниже.

1. Как известно, любая база данных может рассматриваться как коллекция высказы ваний (которые в соответствии с принятым соглашением считаются истинными). А если есть хоть какая-то вероятность, что эта коллекция будет включать любые несогласованные, противоречивые высказывания, то все эти предположения ста новятся бессмысленными. Мы не должны ни в коем случае доверять ответам, по лученным из противоречивой базы данных; фактически из подобной базы данных можно вообще получить абсолютно любой ответ (доказательство этого факта дано в аннотации к [9.16] в главе 9). Хотя свойство изолированности (или кратко свой ство "i" — isolation) транзакций может означать, что с любой конкретной несогла сованностью может столкнуться не больше чем одна транзакция, неоспоримым фактом остается то, что все равно с этой несогласованностью столкнется данная конкретная транзакция, поэтому может выработать неправильные ответы. Имен но поэтому прежде всего необходимо ввести в действие ограничения — не по той причине, что путем изоляции следует исключить возможность обнаруживать эти несогласованности больше чем в одной транзакции одновременно, а потому, что с несогласованностями вообще нельзя смириться.
2. Так или иначе, нельзя гарантировать, что любая конкретная несогласованность (при условии, что допускается их наличие в базе данных) будет обнаружена только одной транзакцией. Подлинной гарантией того, что транзакции остаются изоли рованными друг от друга, может стать только соблюдение в них определенных протоколов, причем эти протоколы не должны вводиться принудительно (а в дей ствительности их невозможно ввести принудительно). Например, если транзакция A обнаруживает противоречивое состояние базы данных и поэтому записывает не согласованные данные в некоторый файл F, после чего транзакция в считывает ту же информацию из файла F, то A и в фактически не являются изолированными друг от друга (независимо от того, выполняются ли они параллельно или в любом другом режиме)[8]. Иными словами, свойство "I" транзакций находится под сомне нием, если не сказать большего.
3. В предыдущем издании данной книги было указано, что ограничения переменной отношения проверяются немедленно, а ограничения базы данных проверяются в конце транзакции (с этой позицией соглашаются многие авторы, хотя они обычно используют другую терминологию). Но из принципа взаимозаменяемости (базовых и производных переменных отношения — см. главу 9) следует, что одно и то же практическое ограничение может стать ограничением переменной отноше ния в одном проекте базы данных и ограничением базы данных — в другом! По скольку вполне очевидно, что ограничения переменной отношения должны про веряться немедленно, из этого следует, что и ограничения базы данных также должны проверяться немедленно.

---

[8] В действительности, эта проблема возникает, даже если транзакция A не обнаруживает несогласо ванного состояния базы данных; все еще остается возможность того, что транзакция A сама записала несогласованные данные в определенный файл, который впоследствии считывается транзакцией B.

4. Для обеспечения возможности выполнять *семантическую оптимизацию* требуется, чтобы база данных была согласованной постоянно, а не только на границах транзакций.

   Примечание. Семантическая оптимизация — это метод использования ограничений целостности для упрощения запросов в целях повышения производительности. Очевидно, что если ограничения не удовлетворяются, то и результаты упрощений будут неправильными. Дополнительная информация по этой теме приведена в главе 18.

Безусловно, *здравый смысл* подсказывает, что проверку ограничений базы данных почти наверняка приходится откладывать на более позднее время. В качестве простейшего примера предположим, что на базу данных поставщиков и деталей распространяется ограничение "Поставщик si и деталь P1 находятся в одном том же городе". Если поставщик S1 переезжает, скажем, из Лондона в Париж, то деталь P1 также необходимо перевезти из Лондона в Париж. Обычно применяемое решение этой проблемы состоит в том, что оба обновления заключаются в одну транзакцию, как показано ниже.

```
BEGIN TRANSACTION      ;
UPDATE S WHERE S#      = S# ('S1') { CITY := 'Paris' } ;
UPDATE P WHERE P#      = P# ('P1') { CITY :=
'Paris' } ;
COMMIT ;
```

В этом обычном решении указанное ограничение проверяется при выполнении оператора COMMIT, а база данных на этапе между двумя операциями UPDATE остается несогласованной. В частности, следует отметить, что если бы в этой транзакции, где выполняются операторы UPDATE, был задан вопрос "Находятся ли поставщик S1 и деталь P1 в одном и том же городе?" на этапе между выполнением этих двух операторов UPDATE, TO был бы получен отрицательный ответ.

Однако следует напомнить, что автор настаивает на необходимости поддержки операции множественного присваивания, которая позволяет выполнить сразу несколько присваиваний в виде одной операции (т.е. в одном операторе) без осуществления какой-либо проверки целостности до того, как будут произведены все рассматриваемые присваивания. Напомним также, что операции INSERT, DELETE и UPDATE являются просто сокращенными обозначениями некоторых операций присваивания. Поэтому в данном примере мы должны иметь возможность осуществлять желаемое обновление в одной операции следующим образом.

```
UPDATE S WHERE S# = S# ('S1') { CITY := 'Paris'
} , UPDATE P WHERE P# = P# ('P1') { CITY :=
'Paris' } ;
```

В таком случае никакие проверки целостности не выполняются до завершения обработки обоих операторов UPDATE (т.е., как принято условно обозначать эту ситуацию, "до достижения точки с запятой"). Следует также отметить, что в данном случае исключена вероятность того, что указанная транзакция обнаружит противоречивое состояние базы данных на этапе между выполнением двух операторов UPDATE, поскольку теперь понятие "этап работы между двумя операторами UPDATE" не имеет смысла.

Из этого примера следует, что если бы поддерживалась операция множественного присваивания, то не было бы необходимости осуществлять отложенную проверку в традиционном смысле этого термина (т.е. выполнять ее как проверку, которая отложена до конца транзакции).

Теперь перейдем к анализу свойств ACID как таковых. Но будет проще провести такой анализ, рассматривая данные свойства в последовательности C-I-D-A (Correctness—Integrity-Durability-Atomicity).

Правильность

Автор уже изложил свои соображения (в главе 15) относительно того, почему он предпочитает использовать указанный здесь термин *правильность* вместо более общепринятого термина *согласованность*. Создается впечатление, что в литературе эти два понятия приравниваются. Например, ниже приведена цитата из глоссария к книге Грея (Gray) и Рейтера (Reuter) [15.12].

Согласованность. Правильность.

Кроме того, в той же книге дано приведенное ниже определение свойства согласованности транзакций.

Согласованность. Транзакция — это правильное преобразование состояния. С этим состоянием связаны действия, объединенные в группу, которые не нарушают каких-либо ограничений целостности. Для этого требуется, чтобы транзакция представляла собой правильную программу [именно так!].

Но если ограничения целостности всегда проверяются немедленно, то база данных всегда является согласованной (но не обязательно правильной!) и транзакции всегда заведомо преобразуют базу данных из одного согласованного состояния в другое согласованное состояние.

Поэтому, если буква C в аббревиатуре ACID обозначает согласованность (consistency), то смысл этого свойства является тривиальным[9], а если оно обозначает правильность (correctness), то его нельзя достичь, принудительно вводя какие-либо протоколы (см. п. 2 в подразделе "Немедленная проверка ограничений"). Поэтому, так или иначе, это свойство фактически является бессмысленным, по меньшей мере, с формальной точки зрения. Как уже было сказано, сам автор предпочитает рассматривать букву C как сокращенное обозначение свойства правильности, но тщательный анализ показывает, что так называемое *свойство правильности* фактически следует считать не свойством как таковым, а всего лишь благим пожеланием.

Изолированность

Теперь рассмотрим свойство изолированности. Как уже было указано в данном разделе, в подразделе "Немедленная проверка ограничений", это свойство также является довольно сомнительным. Но, по крайней мере, нельзя отрицать того, что если каждая транзакция действует по такому же принципу, как если бы она была единственной транзакцией в системе, то безукоризненно действующий механизм управления параллельным выполнением гарантирует изолированность (и упорядочиваемость) транзакций. Тем не менее, для того чтобы транзакция могла действовать "как если бы она была единственной

[9] В действительности, он был бы тривиальным, даже если бы проверка ограничений не осуществлялась немедленно, поскольку все равно выполнялся бы откат транзакции, а это было бы равносильно тому, что она не выполнялась в случае нарушения какого-либо ограничения. Иными словами, все равно остается в силе такое условие, что транзакции оказывают долговременное воздействие на базу данных, только если они не нарушают никаких ограничений.

транзакцией в системе", она, кроме всего прочего, должна соответствовать перечисленным ниже требованиям.

- Не предпринимать попыток (преднамеренных или иных) вступать во взаимодей ствие с другими транзакциями (выполняемыми параллельно или иными).
- Не допускать даже возможности влияния того, что в системе могут существовать другие транзакции (задавая уровень изоляции меньше максимального).

Итак, в действительности свойство изолированности также в большей степени напоминает благие пожелания, чем абсолютную гарантию. Более того, в практически применяемых системах обычно предусмотрены явно заданные механизмы (а именно уровни изоляции меньше максимального), действие которых сводится именно к тому, что они нивелируют свойство изолированности.

Устойчивость

Теперь рассмотрим свойство устойчивости. Это свойство является достижимым благодаря использованию механизма восстановления системы, при условии, что отсутствует вложенность транзакций. И действительно, до настоящего времени предполагалось, что это требование соблюдается. Но допустим, что поддерживается вложенность транзакций. А именно: предположим, что транзакция в вложена в транзакцию А и происходит показанная ниже последовательность событий.

```
BEGIN TRANSACTION (транзакция A) ;

   BEGIN TRANSACTION(транзакция
  B) ;  транзакция B обновляет
  кортеж t ; COMMIT (транзакция
  B) ;

ROLLBACK (транзакция A) ;
```

Если выполняется оператор ROLLBACK транзакции А, то фактически выполняется также откат транзакции в (поскольку в в действительности входит в состав A). Поэтому воздействие транзакции в на базу данных не является *устойчивым;* по сути, выполнение оператора ROLLBACK транзакции А вызывает восстановление того значения кортежа t, которое он имел до выполнения транзакции А. Иными словами, соблюдение свойства устойчивости больше нельзя гарантировать, по меньшей мере, для транзакций, подобных транзакции в, рассматриваемой в данном примере, которая вложена в некоторую другую транзакцию.

А теперь отметим, что многие авторы, начиная с Дэвиса (Davies) [15.8], фактически предложили предусмотреть возможность вкладывать транзакции в той форме, которая показана в приведенном выше примере. А в [15.15] утверждается, что такая поддержка желательна по меньшей мере по трем причинам: распараллеливание работы между транзакциями, управление восстановлением с учетом нескольких транзакций и повышение модульности системы. Но, как показывает данный пример, в системе с подобной поддержкой операторы COMMIT, выполненные во внутренней транзакции, фиксируют обновления, внесенные в этой транзакции, но только на следующем, более внешнем уровне. По сути, внешняя транзакция обладает правом вето на выполнение операторов COMMIT внутренними транзакциями, поскольку если внешняя транзакция выполняет откат, происходит также откат и внутренней транзакции. В данном примере оператор

COMMIT транзакции в является таковым только для транзакции A, но не для внешнего мира, а фактически этот оператор COMMIT впоследствии отменяется (происходит его откат).

Примечание. Следует отметить, что вложенные транзакции могут рассматриваться как обобщение понятия точек сохранения. Точки сохранения позволяют структурировать транзакции как линейные последовательности действий, выполняемые одна за другой (и откат к началу любого предыдущего действия в данной последовательности может произойти в любое время). В отличие от этого, вложение позволяет структурировать транзакции рекурсивно, как иерархию действий, которые могут выполняться одновременно. Иными словами, являются справедливыми приведенные ниже утверждения.

- Оператор BEGIN TRANSACTION расширяется с учетом поддержки *субтранзакций* (т.е., если оператор BEGIN TRANSACTION вызывается на выполнение, притом что уже функционирует некоторая транзакция, то он запускает дочернюю транзакцию).
- Оператор COMMIT *осуществляет фиксацию,* но только в области определения роди тельской транзакции (если транзакция, в которой выполнен этот оператор, явля ется дочерней).
- Оператор ROLLBACK отменяет выполненную работу, но только вплоть до начала данной конкретной транзакции (включая сами дочерние транзакции, дочерние транзакции этих дочерних транзакций и т.д., но не включая родительскую тран закцию, если она имеется).

Теперь, вернувшись к главной теме данного обсуждения, можно убедиться в том, что свойство устойчивости транзакций является применимым только на самом внешнем уровне (иными словами, применяется только к транзакциям, не вложенным в другие транзакции)[10]. Поэтому в общем мы убедились в том, что это свойство также не гарантируется на все 100%.

Неразрывность

Наконец, рассмотрим свойство неразрывности. Как и свойство устойчивости, данное свойство гарантируется механизмом восстановления системы (даже во вложенных транзакциях). Наши возражения против использования этого свойства несколько отличаются от приведенных выше возражений. А именно: достаточно просто отметить, что если бы система поддерживала множественное присваивание, то не нужно было бы требовать от транзакций как таковых, чтобы они обладали свойством неразрывности; вместо этого, было бы достаточно требовать это только от операторов. Кроме того, все равно необходимо соблюдать условие, чтобы операторы также обладали свойством неразрывности, по причинам, уже подробно описанным в других стандартах[11].

---

[10] В [15.15] указано, что такое же утверждение является справедливым и по отношению к свойству согласованности, поскольку в этой статье, как и в большей части литературы, посвященной данной теме, предполагается, что самая внешняя транзакция вовсе не обязательно должна обеспечивать согласован ность на промежуточных этапах. Но мы не разделяем эту позицию по уже описанным выше причинам.

[11] Кстати, следует отметить, что свойством неразрывности, в частности, обладают многие операто ры, описанные в стандарте SQL.

Заключительные замечания

Мы можем подытожить этот раздел с помощью приведенных ниже довольно риторических вопросов.

- *Действительно ли транзакция является единицей работы?* Да, но только если не поддерживается множественное присваивание.
- *Действительно ли она является единицей восстановления ?* Тот же ответ.
- *Действительно ли она является единицей параллельности* ?Тот же ответ.[3]
- *Действительно ли она является единицей целостности?* Да, но только если не соблюдается требование, что "проверка всех ограничений должна осуществляться немедленно".

Примечание. Приведенные здесь ответы являются таковыми несмотря на многие замечания, сделанные в предыдущих главах (а также в предыдущих изданиях данной книги), которые в большей степени следовали *здравому смыслу,* накопленному в этой области знаний.

Поэтому в целом наш вывод состоит в том, что понятие транзакции является значимым в большей степени с практической точки зрения, чем с теоретической. Автор просит понять его правильно, что этот вывод не следует рассматривать как стремление перечеркнуть всю работу, сделанную в этой области! У него не вызывают ничего, кроме глубокого уважения, те многочисленные изящные и полезные результаты, которые получены в течение больше чем 25 лет исследований в области управления транзакциями. Он просто отмечает, что теперь мы обладаем гораздо лучшим пониманием некоторых предпосылок, на которых были основаны эти исследования, — в частности, лучшим пониманием важнейшей роли ограничений целостности, а также признанием того, что необходима поддержка множественного присваивания как примитивного оператора. И действительно, было бы удивительно, если бы смена предпосылок не повлекла за собой пересмотр заключений, сделанных на их основании.

16.11. СРЕДСТВА ЯЗЫКА SQL

В стандарте SQL не предусмотрены какие-либо явно заданные средства блокировки; фактически в нем вообще не упоминается блокировка как таковая[12]. Но этот стандарт требует, чтобы в его реализации были предусмотрены обычные гарантии, касающиеся взаимного вмешательства (или, скорее, его отсутствия) между одновременно выполняемыми транзакциями. Что еще более важно, в этом стандарте требуется, чтобы обновления, внесенные любой конкретной транзакцией T1, не становились бы доступными для любой другой транзакции T2 до тех пор (или только после того), пока не произойдет фиксация транзакции T1.

Примечание. Для соблюдения изложенных выше требований необходимо, чтобы все транзакции выполнялись на уровне изоляции READ COMMITTED, REPEATABLE READ или SERIALIZABLE (см. следующий абзац). К транзакциям, выполняемым на уровне

[12] Это упущение сделано преднамеренно — идея заключается в том, что разработчики системы должны быть вправе использовать любые механизмы управления параллельным выполнением, при условии, что эти механизмы позволяют реализовать желаемые функциональные возможности.

READ UNCOMMITTED, предъявляются особые требования, поскольку им, во-первых, разрешено выполнять *грязное чтение,* но, во-вторых, они обязательно должны относиться к типу READ ONLY (поскольку если бы было разрешено применять в этом случае транзакции типа READ WRITE, то восстанавливаемость больше нельзя было бы гарантировать).

Теперь напомним, что уровни изолированности SQL определяются в операторе START TRANSACTION (как было сказано в главе 15). Для этого предусмотрены четыре **ВОЗМОЖНОСТИ[13]— SERIALIZABLE, REPEATABLE READ, READ COMMITTED И READ UNCOMMITTED.** По умолчанию применяется значение SERIALIZABLE; если определена любая из трех других возможностей, то в конкретной реализации разрешено назначать некоторый более высокий уровень, где понятие *более высокий* определено в терминах упорядочения SERIALIZABLE > REPEATABLE READ > READ COMMITTED > READ UNCOMMITTED.

Если все транзакции выполняются на уровне изоляции SERIALIZABLE (применяемом по умолчанию), то гарантирована упорядочиваемость процедуры чередующегося выполнения любого множества параллельных транзакций. Но если любая транзакция выполняется на более низком уровне изоляции, то нарушение упорядочиваемости может произойти по самым различным причинам. В стандарте определено три вида нарушений: грязное чтение, неповторяемое чтение и фантомы (первые два из них описаны в разделе 16.2, а третий — в разделе 16.8), а различные уровни изоляции определены в терминах нарушений, которые в них допускаются[14]. Итоговые сведения об этих нарушениях приведены в табл. 16.1 (здесь "Y" обозначает, что нарушение может возникнуть, а"N" — нет).

Таблица 16.1. Уровни изоляции SQL

Уровень изоляции	Грязное чтение	Неповторяемое чтение	Фантомы
READ UNCOMMITTED	Y	Y	Y
READ COMMITTED	N	Y	Y
REPEATABLE READ	N	N	Y
SERIALIZABLE	N	N	N

В завершение этого раздела напомним, что ключевое слово REPEATABLE READ, которое определено в стандарте SQL, и опция *повторяемого чтения* (Repeatable Read — RR), определенная в СУБД DB2, не являются одинаковыми. В действительности, опция RR В СУБД DB2 аналогична ключевому слову SERIALIZABLE этого стандарта.

16.12. РЕЗЮМЕ

В этой главе были описаны способы **управления параллельностью.** Вначале рассматривались три проблемы, возникающие из-за отсутствия такого управления при чередующемся выполнении параллельных транзакций, а именно: **проблема потерянного обновления,**

---

[13] В данном случае ключевое слово SERIALIZABLE не совсем подходит, поскольку предполагается, что упорядочиваемыми должны быть графики, а не транзакции. Более подходящим термином был бы просто TWO PHASE, который означает, что транзакция подчиняется (или будет вынуждена подчиняться) протоколу двухфазной блокировки.

[14] Однако см. [16.2] и [16.14].

проблема **зависимости от незафиксированных результатов** и проблема **анализа несогласованности.** Все эти проблемы возникают из-за того, что график запуска не является **упорядочиваемым,** т.е. он не эквивалентен некоторому последовательному графику запуска этих же транзакций.

Наиболее распространенным методом разрешения указанных проблем является использование механизма **блокировки.** Существует два основных типа блокировок: разделяемая (блокировка S) и **исключительная** (блокировка X). Если транзакция устанавливает блокировку S для некоторого объекта, то другие транзакции также смогут установить блокировку S для этого же объекта, однако установить для него блокировку X им не удастся. Если одна транзакция устанавливает блокировку X для некоторого объекта, то никакая другая транзакция не сможет установить для этого объекта никакой другой блокировки. Специальный протокол использования этих блокировок позволяет устранить упомянутую выше проблему потерянного обновления и другие возможные проблемы. В соответствии с данным протоколом, блокировка S устанавливается для всех считываемых объектов, а блокировка X — для всех обновляемых объектов, причем все установленные блокировки сохраняются до окончания выполнения транзакции. С помощью этого протокола можно обеспечить упорядочиваемость графика выполнения транзакций.

Описанный протокол является строгой формой **протокола двухфазной блокировки. В теореме двухфазной блокировки** утверждается, что если выполнение всех транзакций подчиняется этому протоколу, то любой из возможных графиков запуска будет упорядочиваемым. При построении такого графика запуска гарантируется, что если A и **в** являются двумя транзакциями данного графика, то либо транзакция A получит доступ к результатам выполнения транзакции в, либо транзакция **в** получит доступ к результатам выполнения транзакции A. К сожалению, применение протокола двухфазной блокировки может привести к возникновению **взаимоблокировки,** несмотря на то, что этот протокол гарантирует **восстанавливаемость** транзакций и обеспечивает создание графиков, которые не становятся причиной **каскадных откатов.** Для устранения **взаимоблокировки** следует выбрать одну из конфликтующих транзакций в качестве **транзакции-жертвы** и отменить с выполнением отката всех внесенных ею изменений (в результате чего происходит освобождение всех ее блокировок).

В общем случае нельзя гарантировать безопасность выполнения набора транзакций, если используемый график — не полностью упорядочиваемый. Однако в некоторых системах для повышения производительности и пропускной способности предусмотрена возможность чередующегося выполнения нескольких транзакций с некоторым установленным **уровнем изоляции,** что на самом деле является не совсем безопасным решением. В этой главе был описан один такой *небезопасный уровень,* т.е. уровень **стабильности курсора** (с применением терминологии, принятой в СУБД DB2, хотя в стандарте языка SQL аналогичный уровень называется READ COMMITTED).

Далее вкратце было рассмотрено понятие **степени детализации блокировок** и связанная с ним идея **намеченной блокировки.** Предполагается, что перед установкой транзакцией блокировки определенного типа для некоторого объекта, например для кортежа базы данных, ей следует установить соответствующую намеченную блокировку, по меньшей мере, *для родительского* объекта (в случае кортежа — для переменной отношения, в которой содержится кортеж). На практике намеченные блокировки обычно устанавливаются неявно таким же образом, как блокировки S и X кортежей. Однако рекомендуется предусмотреть **оператор типа** LOCK для **явной установки** (в случае необходимости) блокировок,

более сильных, чем те, которые устанавливаются системой неявно (хотя в стандарте SQL такой механизм не предусмотрен).

Затем в этой главе под новым углом зрения были проанализированы так называемые свойства ACID транзакций и сделан вывод, что, вопреки широко распространенному мнению, эти свойства являются не такими уж бесспорными. Наконец, в данной главе были кратко описаны средства управления параллельностью, предусмотренные в языке SQL. В стандарте языка SQL средства явной установки блокировки не предусмотрены вообще, однако в нем поддерживаются различные уровни изоляции (SERIALIZABLE, **REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED), которые в СУБД обычно**
реализуются неявно заданными средствами блокировки.

УПРАЖНЕНИЯ

16.1. Дайте определение понятия*упорядочиваемости.*

16.2. Дайте полное определение:

а) протокола двухфазной блокировки; !

б) теоремы двухфазной блокировки.

Подробно опишите, каким образом с помощью двухфазной блокировки разрешаются конфликты RW, WR и WW.

16.3. Пусть даны транзакции Tl, T2 и T3, которые выполняют следующие операции:

- T1 — сложить 1 с A;
- T2 — удвоить значение A;
- T3 — вывести значение A на экран, а затем поместить в A значение 1

(здесь A — некоторый числовой элемент в базе данных).

а) Допустим, что разрешено параллельное выполнение транзакций Tl, T2 и T3. Перечислите все возможные результаты их выполнения, если начальное значение A равно нулю.

б) Допустим, что транзакции Tl, T2 и T3 имеют показанную ниже внутреннюю структуру.

T1	T2	T3
R1. RETRIEVE A INTO a1 ;	R2. RETRIEVE A INTO a2 ;	R3. RETRIEVE A INTO a3 ;
a1 := a1 + 1 ;	a2 := a2 * 2 ;	Вывести значение a3 на экран ;
U1. UPDATE A FROM a1 ;	U2. UPDATE A FROM a2 ;	U3. UPDATE A FROM 1 ;

Сколько существует возможных графиков запуска, если эти транзакции выполняются *без* каких-либо блокировок?

в) Существуют ли какие-либо чередующиеся графики запуска, которые для заданного начального значения A (нуль) приводят к получению *правильного* результата, но не допускают упорядочения?

г) Существуют ли какие-либо графики запуска, которые действительно допускают упорядочение, но не могут быть реализованы, если все три транзакции подчиняются протоколу двухфазной блокировки?

16.4. Ниже приведена последовательность выполнения различных событий в графике запуска с транзакциями T1, T2, ..., T12 и объектами базы данныхA, в, ..., H.

```
время tO  ......
время tl (Tl)  : RETRIEVE A
; время t2 (T2) : RETRIEVE
B ;
  ... (Tl) : RETRIEVE C ;
  ... (T4) : RETRIEVE D ;
  ... (T5) : RETRIEVE A ;
  ... (T2) : RETRIEVE E ;
  ... (T2)  : UPDATE E ;
  ... (T3) : RETRIEVE F ;
  ... (T2) : RETRIEVE F ;
  ... (T5) : UPDATE A ;
  ...(Tl) : COMMIT ;
  ... (T6) : RETRIEVE A ;
  ... (T5) : ROLLBACK ;
  ... (T6) : RETRIEVE C ;
  ... (T6) : UPDATE C ;
  ... (T7) : RETRIEVE G ;
  ... (T8) : RETRIEVE H ;
  ... (T9) : RETRIEVE G ;
  ... (T9) : UPDATE G ;
  ...(T8) : RETRIEVE E ;
  ... (T7) : COMMIT ;
  ... (T9) : RETRIEVE H ;
  ... (T3) : RETRIEVE G ;
  ... (T10) : RETRIEVE A ;
  ... (T9) : UPDATE H ;
  ... (T6) : COMMIT ;
   ...       (T11) : RETRIEVE C ;
  ... (T12) : RETRIEVE D ;
  ... (T12) : RETRIEVE C ;
  ... (T2) : UPDATE F ;
  ... (T11) : UPDATE C ;
  ... (T12) : RETRIEVE A ;
  ... (T10) : UPDATE A ;
  ... (T12) : UPDATE D ;
  ... (T4) : RETRIEVE G ;
время t3 6  ......
```

Предположим, что для выполнения оператора RETRIEVE i (считать i) требуется (в случае успеха) установить блокировку S для объекта i, а для выполнения оператора UPDATE i (обновить i) требуется (в случае успеха) установить для объекта i блокировку X. Допустим, что все блокировки сохраняются до конца выполнения транзакции. Начертите граф ожидания (показав на нем, "кто кого ожидает"), который соответствует состоянию дел в момент времени t3 6. Возникнет ли к этому моменту ситуация взаимоблокировки?

16.5. Еще раз обратимся к проблемам параллельности, представленным на рис. 16.1-16.4. Что произойдет в каждом из этих случаев, если все транзакции будут выполняться на уровне изоляции *стабильность курсора* (CS), а не на уровне *повторяемое чтения* (RR)?

Примечание. Здесь обозначения CS и RR относятся к уровням изоляции DB2, которые описаны в разделе 16.8.

16.6. Дайте формальные и неформальные определения пяти типов блокировок: X, S, IX, IS и SIX.

16.7. Сформулируйте понятие относительной силы блокировок и нарисуйте соответствующую диаграмму их предшествования.

16.8. Сформулируйте определение протокола намеченной блокировки, а также поясните основное назначение этого протокола.

16.9. В стандарте языка SQL определяются три проблемы параллельности: *грязное чтение, неповторяемое чтение* и *фантомы.* Как они соотносятся с тремя проблемами параллельности, описанными в разделе 16.2?

16.10. Кратко опишите возможный механизм реализации протокола управления параллельностью на основе поддержки многих версий, который рассматривается в аннотации к [16.1].

СПИСОК ЛИТЕРАТУРЫ

В этом списке следовало бы также упомянуть работы [15.2], [15.10] и особенно [15.12] из главы 15.

16.1. Bayer R., Heller M., Reiser A. Parallelism and Recovery in Database Systems // ACM TODS. - June 1980. - 5, № 2.

Как уже говорилось в главе 15, новые прикладные области (например, проектирование и разработка программного и аппаратного обеспечения) часто выдвигают сложные требования к обработке данных, которые нельзя полностью удовлетворить с помощью классических схем управления выполнением транзакций, описанных в этой и предыдущей главах. Основная проблема заключается в том, что сложные транзакции в этих прикладных областях могут продолжаться в течение часов или даже суток вместо нескольких миллисекунд, что типично для большинства традиционных систем обработки данных. Это может привести к перечисленным ниже последствиям.

1. Полный откат транзакции к самому началу может вызвать потерю слишком большого объема уже проделанной работы, что совершенно неприемлемо.
2. Использование обычных блокировок может иметь следствием появление чрезмерных задержек, вызванных ожиданием снятия блокировок требуемых элементов, что также неприемлемо.

В этой работе описывается один из методов устранения подобных недостатков (см. также [16.8], [16.12], [16.15] и [16.20]), а именно— метод управления параллельностью, называемый *параллельным выполнением с применением многих версий* (multi-version locking); он известен также под названием *чтения с применением*

многих версий (multi-version read) и уже используется в нескольких коммерческих программных продуктах. Основное преимущество этого метода заключается в том, что операции чтения выполняются без вынужденного ожидания, т.е. любое количество операций чтения и *одна операция записи* могут выполняться для одного и того же объекта одновременно. Точнее говоря, выполняются приведенные ниже правила.

- Операция чтения всегда выполняется без задержки (как только что говорилось).
- Операции чтения никогда не задерживают выполнение операций обнсвления.
- Откат транзакций, выполняющих только чтение, не имеет смысла и никогда не осуществляется.
- Ситуация взаимной блокировки может возникнуть только между транзакция ми обновления.

Указанные преимущества особенно важны для распределенных систем (см. главу 21), в которых на выполнение обновления данных может потребоваться значительное время, поэтому может чрезмерно затянуться выполнение запросов, включающих только операции чтения (и наоборот). Основную идею данного метода можно сформулировать, как описано ниже.

- Если транзакция в пытается выполнить операцию чтения объекта, для кото рого в данный момент выполняется обновление в пределах транзакции А, то транзакции в предоставляется доступ к *предварительно зафиксированной* версии этого объекта. Такая версия обычно хранится в каком-то особом месте систе мы (как правило, в журнале регистрации) и предназначается для восстановле ния системы.
- Если транзакция в пытается выполнить операцию обновления объекта, кото рый в данный момент считывается транзакцией А, то транзакции в предостав ляется доступ к этому объекту, тогда как транзакции А предоставляется доступ к некоторой версии этого объекта (которая в действительности является его предыдущей версией).
- Если транзакция в пытается выполнить операцию обновления объекта, кото рый в данный момент обновляется транзакцией А, то транзакция в переходит в состояние ожидания[15] (таким образом, как отмечалось выше, в системе может возникнуть ситуация взаимоблокировки, для устранения которой потребуется принудительный откат одной из транзакций).

Безусловно, этот метод предусматривает наличие соответствующих средств управления, необходимых для обеспечения того, что каждая транзакция всегда будет "иметь дело" с согласованным состоянием базы данных.

16.2. Berenson H. et al. A Critique of ANSI SQL Isolation Levels // Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data. — San Jose, Calif. — May 1995.

[15] Иначе говоря, конфликты WW все еще могут возникать, и здесь предполагается, что для их устранения используется механизм блокировки. Однако для устранения конфликтов, безусловно, могут использоваться и другие методы, например временные отметки [16.3].

В этой статье критически оценивается предпринятая в стандарте "ANSI SQL" попытка охарактеризовать уровни изоляции на основе нарушений упорядочиваемости (см. раздел 16.11). "[Эти] определения не способны должным образом охарактеризовать несколько широко распространенных уровней изоляции, включая стандартные способы реализации блокировок для упомянутых уровней". В частности, в этой статье подчеркивается, что данный стандарт не способен запретить *грязную запись (си.* раздел 16.2).

Похоже, что это действительно так, поскольку в данном стандарте *грязная запись* явно не запрещена. В нем фактически изложены приведенные ниже требования (здесь они немного перефразированы).

- "Выполнение параллельных транзакций на уровне изоляции SERIALIZABLE гарантированно является упорядочиваемым". Иначе говоря, если все транзакции выполняются на уровне изоляции SERIALIZABLE, в реализации *требуется* запретить операции *грязной записи,* поскольку они, безусловно, будут нарушать упорядочиваемость.
- "Четыре уровня изоляции гарантируют, что... никакие обновления не будут потеряны". Это утверждение на самом деле является лишь благим пожеланием, поскольку сами определения четырех уровней изоляции не дают подобных гарантий. Однако оно указывает на то, что разработчики стандарта действительно стремились запретить операции *грязной записи.*
- *т* "Изменения, выполненные одной транзакцией, не могут быть восприняты другими транзакциями [за исключением транзакций на уровне изоляции READ UNCOMMITTED] до тех пор, пока результаты выполнения первоначальной транзакции не будут зафиксированы". Что в данном контексте означает слово *восприняты'!* Может ли транзакция обновить часть *грязных данных* без их *восприятия?*

См. также [16.14].

16.3. Bernstein P.A., Goodman N. Timestamp-Based Algorithms for Concurrency Control in Distributed Database Systems // Proc. 6th Int. Conf. on Very Large Data Bases. — Montreal, Canada. — October 1980.

В работе рассматриваются некоторые методы управления параллельностью, в основу которых положены не блокировки, а временные отметки. Основная идея заключается в том, что если транзакция А начинает выполняться раньше транзакции в, то система ведет себя так, как будто выполнение транзакции А полностью завершилось еще до начала выполнения транзакции в (как это имеет место в классическом последовательном графике). Из этого следуют приведенные ниже выюды.

1. Для транзакции А не должны быть доступны любые обновления, выполняемые транзакцией в.
2. Для транзакции А не должно быть разрешено обновлять объект, уже считанный транзакцией в.

Эти два требования могут быть выполнены следующим образом. Для каждого запроса к базе данных система сравнивает временную отметку транзакции этого запроса с временной отметкой транзакции, выполнившей последнюю выборку или обновление запрашиваемого кортежа. В случае конфликта транзакция запроса

может быть перезапущена с новой временной отметкой (так же, как и при использовании так называемых *оптимистических* методов [16.16]).

Как следует из заголовка этой работы, данная методика была изначально создана в контексте распределенной системы (где использование блокировки сопряжено с возникновением нежелательных издержек из-за отправки сообщений для проверки и установки блокировок). Однако такая методика почти наверняка не подходит для нераспределенных систем. Кроме того, существует достаточно скептическая оценка практичности ее использования даже для распределенных систем. Одной очевидной проблемой является то, что каждый кортеж должен содержать временную отметку последней транзакции, которая *считывала* данные этого кортежа (а также временную отметку последней транзакции, которая обновляла данные этого кортежа). В таком случае подразумевается, что каждая операция чтения становится операцией записи! Еще одна проблема состоит в том, что транзакция в не должна иметь доступ к каким-либо обновлениям транзакции А до тех пор, пока в последней не будет выполнена фиксация; из этого следует, что (по сути) на обновления транзакции А накладывается *исключительная блокировка* до фиксации каких-либо данных. Действительно, в [15.12] утверждается, что такая схема с применением временных отметок на самом деле является вырожденным случаем схем оптимистического метода управления параллельным выполнением [16.16], для которых, в свою очередь, характерны собственные проблемы.

Примечание. Одно из понятий, широко обсуждавшихся в литературе, — *правило записи Томаса* [16.22] — фактически представляет собой одно из усовершенствований описанной выше схемы; оно основано на той идее, что некоторые операции обновления можно пропустить, поскольку они уже являются устаревшими (запрос на уровне пользователя удовлетворяется, а физическое обновление не выполняется).

16.4. Blasgen M.W., Gray J.N., Mitoma M., Price T.G. The Convoy Phenomenon // ACM Operating Systems Review. — April 1979. — 13, № 2.

Проблема образования вереницы (convoy) возникает при использовании механизма блокировок для интенсивно применяемых элементов данных. Примером может служить установка блокировки перед выполнением операции записи в журнал регистрации в системах с *приоритетным планированием.*

Примечание. Здесь понятие планирования относится к проблеме распределения процессорного времени по выполняемым транзакциям, а не к чередованию операций, принадлежащих различным транзакциям, что обсуждалось ранее в настоящей главе.

Эта проблема может быть пояснена следующим образом. Пусть транзакция т устанавливает блокировку некоторых интенсивно используемых элементов данных, после чего выполнение этой транзакции прекращается системным планировщиком, т.е. транзакция переходит в состояние ожидания, например, из-за того, что время отведенное ей для выполнения, истекло. Такая ситуация способствует образованию *вереницы* транзакций, каждая из которых ожидает продолжения его выполнения и уже успела установить блокировку некоторых элементов интенсивно используемых данных. Когда транзакция Т выйдет из подобного состояния ожидания она должна будет завершить обработку и снять блокировку используемых

ею элементов данных. Однако именно из-за того, что заблокированными являются интенсивно используемые данные, велика вероятность того, что транзакция T выйдет из состояния ожидания слишком рано, еще до того, как иная транзакция завершит работу с другим требуемым транзакции т элементом данных. Поэтому транзакция т не сможет продолжить выполнение своих операций и уже по собственной инициативе переведет себя в состояние ожидания.

Суть проблемы заключается в том, что в большинстве случаев (но не во всех) планирование является частью функций базовой операционной системы, а не СУБД, поэтому оно организуется на основе совершенно других допущений. По наблюдениям авторов этой работы, однажды образовавшаяся вереница проявляет тенденцию к стабильности. В результате система оказывается в состоянии постоянной *пробуксовки блокировок* (lock thrashing) и основная часть процессорного времени расходуется на переключение с одного процесса на другой, а не на выполнение какой-либо полезной работы. В качестве возможного решения этой проблемы, помимо замены способа планирования, можно предложить установку блокировок в произвольном порядке, а не на основе схемы "первым поступил-первым обработан".

16.5. Blott S., Korth H.F. An Almost-Serial Protocol for Transaction Execution in Main-Memory Database Systems // Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong. — August 2002.

В статье предложен механизм обеспечения упорядочиваемое™ для систем с оперативной памятью, который позволяет полностью избежать необходимости использования блокировок.

16.6. Eswaran K.P., Gray J.N., Lorie R.A., Traiger I.L. The Notions of Consistency and Predicate Locks in a Data Base System // CACM. — November 1976. — 19, № 11.

В этой работе впервые дано строгое теоретическое описание основ управления параллельностью.

16.7. Franaszek P., Robinson J.T. Limitations on Concurrency in Transaction Processing // ACM TODS. - March 1985. - 10, № 1.

См. аннотацию к [16.16].

16.8. Franaszek P., Robinson J.T., Thomasian A. Concurrency Control for High Contention Environments //ACM TODS. - June 1992. - 17, № 2.

В статье сделано заявление, что в будущем системы обработки транзакций, вероятно, будут обеспечивать гораздо большую степень параллельности, чем современные системы (по разным причинам). Поэтому в них будут гораздо чаще возникать конфликтные ситуации на уровне данных. Затем авторы представляют "несколько концепций управления параллельностью [без использования блокировки] и методов планирования транзакций, которые применимы к средам с большим количеством конфликтных ситуаций". Применимость такого подхода обоснована экспериментами с различными имитационными моделями и, по словам авторов, этот подход "может обеспечить значительные преимущества".

16.9. Gray J.N. Experience with the System R Lock Manager // IBM San Jose Research Laboratory internal memo, 1980.

Эта работа представляет собой набор заметок, а не законченную статью, причем они в настоящее время уже несколько устарели. Тем не менее, в ней содержится несколько интересных утверждений, включая приведенные ниже.

- Использование блокировок связано с издержками в размере 10% при работе с транзакциями, выполняемыми в интерактивном режиме, и в размере 1% при работе с транзакциями в пакетном режиме.
- Желательно поддерживать разные уровни детализации блокировок, ■ На практике хорошо себя показали средства автоматической эскалации уровня блокировок.
- Метод с использованием повторяемого чтения (RR) является более эффектив ным, а также более безопасным по сравнению с методом, основанным на ста бильности курсоров (CS).
- Ситуации взаимоблокировки очень редко возникают на практике **и** никогда не включают более двух транзакций.
- Почти все ситуации взаимоблокировки (97%) можно устранить с помощью блокировок U, что и предусмотрено в СУБД DB2, но не в системе System R.

Примечание. В этом определении предполагается, что блокировки U совместимы с блокировками S, но не с другими блокировками U и, безусловно, не с блокировками X. Более подробные сведения приводятся в [4.21].)

16.10. Gray J.N., Lorie R.A., Putzolu G.R. Granularity of Locks in a Large Shared Data Base// Proc. 1st Int. Conf. on Very Large Data Bases. — Framingham, Mass. — September 1975.

В статье впервые описана концепция **намеченной блокировки.** Как объясняется в разделе 16.8, понятие *степень детализации* (granularity) относится к размеру блокируемых объектов. Поскольку очевидно, что разные транзакции имеют различные характеристики и требования, желательно, чтобы в системе существовал целый набор возможных степеней детализации блокировок (что, в принципе, реализовано во многих системах). В этой статье представлена методика воплощения нескольких степеней детализации блокировок на основе механизма намеченной блокировки в подобной системе.

Поскольку в данной главе было дано весьма упрощенное описание протокола намеченной блокировки, ниже он будет рассмотрен несколько подробнее. Во-первых, стоит отметить, что блокируемыми объектами могут быть не только переменные отношения и кортежи, как предполагалось прежде. Во-вторых, типы блокируемых объектов могут в общем случае не обладать строгой иерархической структурой. Присутствие индексов или других структур доступа будет лишь означать, что систему объектов следует рассматривать как *ориентированный ациклический граф.* Например, база данных поставщиков и деталей может содержать как хранимую переменную отношения деталей P, так и индекс, например XP, построенный для атрибута p# этой переменной отношения. Для выборки кортежей переменной отношения P следует открыть всю базу данных, а затем *либо* перейти непосредственно к этому отношению и выполнить последовательный поиск, *либо* использовать индекс XP для прямого перехода к искомым кортежам переменной отношения P. Таким образом, кортежи переменной отношения P в этом графе имеют два

родительских объекта, P и XP, для каждого из которых *родительским* объектом является база данных.

Теперь формулировку этого протокола можно представить в более обобщенной форме.

- При установке блокировки X для данного объекта для всех его дочерних объек тов неявным образом устанавливается блокировка X.
- При установке блокировки S или SIX для данного объекта для всех его дочерних объектов неявным образом устанавливается блокировка S.
- Прежде чем транзакция сможет потребовать установить блокировку S или IS для заданного объекта, она должна установить блокировку IS (или более силь ную) по крайней мере для одного из его родительских объектов.
- Прежде чем транзакция сможет потребовать установить блокировку X, IX или SIX для заданного объекта, она должна установить блокировку IX (или более сильную) для всех родительских объектов этого объекта.
- Прежде чем транзакция получит право освободить блокировку данного объек та, она должна вначале освободить все блокировки, установленные ею для до черних объектов данного объекта.

На практике использование этого протокола не сопровождается слишком большими издержками, как это может показаться на первый взгляд, поскольку в любой заданный момент транзакция, весьма вероятно, уже будет обладать почти всеми необходимыми ей блокировками. Например, блокировка IX для всей базы данных, вероятно, будет установлена непосредственно во время инициализации программы и эта блокировка будет сохраняться для всех транзакций, выполняемых на протяжении всей работы данной программы.

16.11. Gray J.N., Lorie R.A., Putzolu G.R., Traiger I.L. Granularity of Locks and Degrees of Consistency in a Shared Data Base // Proc. IFIP TC-2 Working Conf. on Modeling in Data Base Management Systems (ed. G. M. Nijssen). — Amsterdam, Netherlands: North-Holland/New York, N.Y.: Elsevier Science, 1976.

В статье впервые введено понятие уровней изоляции (под названием *степени согласованности* — degrees of consistency)[16].

16.12. Harder T., Rothermel K. Concurrency Control Issues in Nested Transactions // The VLDB Journal. — January 1993. — 2, № 1.

Как уже отмечалось в разделе "Список литературы" главы 15, несколькими различными авторами была предложена идея **вложенных транзакций.** В этой статье предлагается соответствующий набор протоколов блокировки для подобных транзакций.

16.13. Jordan J.R., Banerjee J., Batman R.B. Precision Locks // Proc. 1981 ACM SIGMOD Int. Conf. on Management of Data. — Ann Arbor, Mich. — April/May 1981.

---

[16] Этот термин нельзя назвать очень удачным. Данные могут быть только согласованными или несогласованными. Таким образом, термин, который указывает на то, что могут быть "степени" согласованности, звучит так, как будто ставится под сомнение сам этот тезис. Создается впечатление, что теория, лежащая в основе понятия "степеней согласованности", была разработана еще до того, как удалось достичь четкого понимания фундаментальной важности понятия целостности данных (или их "согласованности").

Точная блокировка является схемой блокировки на уровне кортежей, которая гарантирует блокировку только нужных кортежей (для достижения упорядочиваемости), включая фантомы. На самом деле эта форма блокировки называется блокировкой *предикатов* (см. раздел 16.8, а также [16.6]). Она основана на следующих действиях. Во-первых, при проверке запросов на обновление система определяет, удовлетворяет ли вставляемый или удаляемый кортеж сделанному ранее запросу на выборку данных в некоторой параллельной транзакции. Во-вторых, при проверке запросов на выборку система определяет, удовлетворяет ли вставляемый или удаляемый в некоторой параллельной транзакции кортеж данному запросу на выборку. Эта схема не только изящна, но и, как заявляют ее авторы, обладает более высокой производительностью, чем обычные методы (в которых блокируется чрезмерно много объектов).

16.14. Kempster T., Stirling C, Thanisch P. Diluting ACID // ACM SIGMOD Record.— December 1999. - 28, № 4.

Эту статью лучше было бы назвать не "Разведение", а "Концентрирование ACID" (если ACID рассматривается не как аббревиатура, а как одно слово — кислота)! Кроме всего прочего, в ней утверждается, что обычные механизмы управления параллельным выполнением исключают возможность создания некоторых упорядочиваемых графиков (как сказано в статье, "изолированность фактически является достаточным, но не необходимым условием для упорядочиваемое™"). Как и в стандарте SQL и в [16.2], в этой статье уровни изоляции определены в терминах нарушений упорядочиваемости, но эти определения являются более точными и охватывают больше упорядочиваемых графиков по сравнению с предыдущими попытками. В данной статье также отмечена одна ошибка (касающаяся фантомов) в [16.2].

16.15. Korth H.F., Speegle G. Formal Aspects of Concurrency Control in Long-Duration Transaction Systems Using the NT/PV Model // ACM TODS. - September 1994.. - 19, №3.

Как уже отмечалось в литературе (см., например, [15.3], [15.9], [15.16] и [15.17]), упорядочиваемость часто рассматривается как чрезмерно строгое требование, которое накладывается на некоторые виды систем обработки транзакций, особенно в новых прикладных областях, включающих взаимодействие с людьми, а значит, и транзакции большой длительности. В данной статье представлена новая модель обработки транзакций NT/PV (Nested Transactions with Predicates and Views — вложенные транзакции с предикатами и представлениями), позволяющая решить эти проблемы. В статье также показано, что стандартная модель обработки транзакций с упорядочением является лишь частным случаем; после чего дается определение "новых и более полезных классов правильности". Здесь утверждается, что новая модель обеспечивает "необходимую основу для решения проблем, связанных с долговременными транзакциями".

16.16. Kung H.T., Robinson J.T. On Optimistic Methods for Concurrency Control // ACM TODS. -June 1981. -6, №2.

Схемы блокировки рассматриваются как *пессимистические* в случае, когда делается предположение (худшее из возможных) о том, что все данные, необходимые для заданной транзакции, могут также потребоваться для параллельно выполняемой

транзакции и поэтому их следует заблокировать. В отличие от этого, в **оптимистических** схемах (называемых также схемами *сертификации* или *подтверждения)* делается противоположное допущение о том, что такие конфликтные ситуации весьма маловероятны. Поэтому в рассматриваемых схемах допускается беспрепятственное выполнение всех транзакций. И только во время фиксации проверяется наличие конфликтных ситуаций. Если такие конфликты имели место, то вызвавшие их транзакции просто перезапускаются. Никакие обновления не записываются в базу данных до успешного завершения процедуры фиксации, поэтому при повторном запуске транзакций не требуется отменять какие-либо операции обновления. Позднее в [16.7] было показано, что при некоторых разумных предположениях оптимистические методы демонстрируют определенные присущие им преимущества по сравнению с традиционными методами блокировки для некоторого уровня параллельности (т.е. некоторого числа одновременно выполняемых транзакций), который они способны поддерживать. Предполагается, что оптимистические методы могут быть наиболее полезными в системах с большим количеством параллельных процессоров. (В [15.12], наоборот, утверждается, что оптимистические методы в общем случае на самом деле *хуже* методов блокировки в некоторых "горячих" ситуациях, т.е. в ситуациях, когда объект данных обновляется очень часто и несколькими различными транзакциями. Более подробную информацию о методах, эффективных при работе в подобных ситуациях, можно найти в [16.17].)

16.17. O'Neil P.E. The Escrow Transactional Method // ACM TODS. — December 1986. — 11, №4.,

Рассмотрим следующий весьма простой пример. Предположим, что в базе данных содержится объект данных тс, представляющий *имеющуюся на руках наличность,* и допустим, что почти каждая транзакция в системе обновляет данные в объекте ТС с некоторым уменьшением их значения (соответствующим некоторому списанию денежных средств со счета). Объект ТС представляет собой пример *горячей точки,* т.е. элемента базы данных, который используется большинством транзакций в системе. Образно говоря, при традиционной блокировке *горячая точка* может очень скоро превратиться в узкое место всей системы, поэтому в таких ситуациях было бы весьма неразумно использовать традиционную блокировку. Если начальное значение ТС равно 10 млн. долл., а каждая отдельная транзакция приводит к его уменьшению в среднем на 10 долл., то в целом может быть выполнено около миллиона таких транзакций. Причем для этого *потребовалось бы выполнить в произвольном порядке около миллиона операций вычитания.* В таком случае действительно вовсе нет необходимости в использовании традиционной блокировки для ТС. Вместо этого следует убедиться, что текущее значение достаточно велико для успешного выполнения данной операции вычитания, а затем выполнить обновление значения. (Если выполнение такой транзакции не будет успешно завершено, то последнюю операцию вычитания следует отменить.)

Метод **депонирования** применяется в ситуациях, подобных описанной выше, т.е. когда обновления имеют некоторую конкретную, а не произвольную форму. В такой системе должен быть определен особый вид оператора обновления (например, "уменьшить на х тогда и только тогда, когда текущее значение больше, чем у"). В этом случае обновление можно было бы выполнить, помещая уменьшенное

значение х в некоторый третий объект-депонент, для его извлечения оттуда в конце выполнения транзакции (это изменение фиксируется, если в конце транзакции выполняется оператор COMMIT, а если транзакция оканчивается оператором ROLLBACK, эта сумма снова добавляется к первоначальной).

В статье описывается несколько ситуаций, в которых можно использовать метод депонирования. Одним из примеров коммерческого продукта, в котором поддерживается данный метод, является информационная система IMS Fast Path фирмы IBM. Следует отметить, что этот метод может рассматриваться как частный случай оптимистического управления параллельностью [16.16]. Но заслуживает внимания то, что данный случай и должен рассматриваться как частный, поэтому для него крайне необходимо предусмотреть специальные операторы обновлгния.

16.18. Papadimitriou C. The Theory of Database Concurrency Control. — Rockville, Md.: Computer Science Press, 1986.

Учебное пособие, в котором особое внимание уделяется формальной теории.

16.19. Rosencrantz D.J., Stearns R.E., Lewis II P.M. System Level Concurrency Control for Distributed Database Systems //ACM TODS. — June 1978. — 3, № 2.

16.20. Salem K., Garcia-Molina H., Shands J. Altruistic Locking // ACM TODS. — March 1994._ 19, № 1.

В этой статье предлагается расширение протокола двухфазной блокировки, в соответствии с которым транзакция А, завершившая работу с заблокированным элементом данных (который не может быть разблокирован из-за ограничений протокола двухфазной блокировки), все же *возвращает* этот элемент системе, тем самым позволяя некоторой другой транзакции в установить для него блокировку. В этом случае говорят, что транзакция в *идет вслед за* транзакцией А. Здесь определены протоколы, например, позволяющие предотвратить получение некоторой транзакцией любых обновлений, внесенных той транзакцией, которая следовала за ней. Показано, что альтруистическая блокировка (происхождение этого термина связано с тем фактом, что *возвращение* данных приносит пользу другим транзакциям, а не транзакции, отдающей другим свои ресурсы) обеспечивает более высокую степень параллельности, чем обычный протокол двухфазной блокировки, особенно когда некоторые из транзакций продолжаются достаточно долго.

16.21. Silberschatz A., Korth H.F., Sudarshan S. Database System Concepts (4th ed.). — New York, N.Y.: McGraw-Hill, 2002.

' В этом учебнике по вопросам управления базой данных содержится формальное описание тематики управления транзакциями (с учетом требований восстановления и организации параллельной работы).

16.22. Thomas R.H. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases//ACM TODS. — June 1979. — 4, № 2.

См. аннотацию к [16.3].

16.23. Thomasian A.Concurrency Control: Methods, Performance, and Analysis // ACM Сотр. Surv. — March 1998. — 30, № 1.

Подробное исследование производительности различных алгоритмов управления параллельной работой.

ЧАСТЬ V

ДОПОЛНИТЕЛЬНЫЕ ТЕМЫ

В части II этой книги было сказано, что реляционная модель является основой современной технологии баз данных, и это действительно так. Но это только *основа,* и, помимо описанной в части II реляционной модели, в технологии баз данных есть много других компонентов. Как студентам, так и профессионалам необходимо освоить множество дополнительных концепций и технологий, чтобы полностью овладеть этой областью знаний (последнее вполне очевидно из обсуждений в частях III и IV данной книги). Теперь нам предстоит сосредоточить внимание на ряде важных тем, которые будут рассматриваться в такой последовательности.

- Защита данных (глава 17).
- Оптимизация (глава 18).
- Отсутствующая информация (глава 19).
- Наследование типов (глава 20).
- Распределенные базы данных (глава 21).
- Поддержка принятия решений (глава 22).
- Хронологические базы данных (глава 23).
- Логические системы управления базами данных (глава 24).

Выбранная последовательность глав носит несколько произвольный характер, однако при их написании предполагалось, что эти главы (может быть, не все) будут изучаться именно в таком порядке.

ГЛАВА 17

Защита данных

17.1. ВВЕДЕНИЕ

Вопросы защиты данных часто рассматриваются вместе с вопросами поддержки целостности данных (по крайней мере, в неформальном контексте), хотя на самом деле это совершенно разные понятия. Термин *защита* (security) относится к защищенности данных от несанкционированного доступа, изменения или умышленного разрушения, а *целостность*— к точности или достоверности данных'. Эти термины можно определить, как показано ниже.

- Под **защитой данных** подразумевается предотвращение доступа к ним со стороны
  несанкционированных пользователей.
- Под **поддержкой целостности данных** подразумевается предотвращение их разру
  шения при доступе со стороны *санкционированных* пользователей (!).

Иначе говоря, защита данных означает получение гарантий, что пользователям *разрешено* выполнять те действия, которые они пытаются выполнить, а поддержка целостности означает получение гарантий, что действия, которые пользователи пытаются выполнить, будут допустимыми.

---

[1] Как указано в главе 9, в данном контексте термин *достоверность* или, скорее, *правильность,* должен быть в действительности приведен в кавычках.

Между этими понятиями есть, безусловно, некоторое сходство, поскольку как при обеспечении защиты данных, так и при обеспечении поддержки их целостности система вынуждена проверять, не нарушаются ли при выполняемых пользователем действиях некоторые установленные *ограничения.* Эти ограничения формулируются (обычно администратором базы данных) на некотором подходящем языке и сохраняются в системном каталоге. Причем в обоих случаях СУБД должна каким-то образом отслеживать все выполняемые пользователем действия и проверять их соответствие установленным ограничениям. В данной главе речь пойдет о защите данных, а вопросы целостности уже рассматривались в главе 9.

Примечание. Основная причина, по которой эти две темы обсуждаются раздельно, состоит в том, что целостность данных является фундаментальным понятием, тогда как защита данных — понятием вторичным, несмотря на его большую практическую важность (особенно в наши дни повсеместного распространения Internet, электронной коммерции и соответствующих средств доступа).

Ниже описаны многочисленные аспекты проблемы защиты данных.

- Правовые, общественные и этические аспекты (например, имеет ли некоторое ли цо легальное основание запрашивать, скажем, информацию о выделенном клиен ту кредите).
- Физические условия (например, запирается ли помещение с компьютерами или терминалами на замок либо оно охраняется другим способом).
- Организационные вопросы (например, каким образом на предприятии, являю щемся владельцем системы, принимается решение о том, кому разрешено иметь доступ к тем или иным данным).
- Вопросы управления (например, как в случае организации защиты системы от не санкционированного доступа по схеме паролей обеспечивается секретность ис пользуемых паролей и как часто они меняются).
- Аппаратные средства защиты (например, имеет ли используемое вычислительное оборудование встроенные функции защиты, подобные ключам защиты хранимой информации или привилегированному режиму управления).
- Возможности операционной системы (например, стирает ли используемая опера ционная система содержимое оперативной памяти и дисковых файлов после прекращения работы с ними, и каким образом обрабатывается журнал восста новления).
- Аспекты, имеющие отношение непосредственно к самой СУБД (например, под держивает ли используемая СУБД концепцию владельца данных).

По очевидным причинам обсуждение этой обширной темы в данной главе будет ограничено в основном вопросами, относящимися к последнему пункту приведенного выше списка.

В современных СУБД обычно поддерживается один из двух широко распространенных методов организации защиты данных — *избирательный* (discretionary) или *мандатный* (mandatory), а иногда оба этих метода. В обоих случаях единица данных (или *объект данных)* для которой организуется защита, может выбираться из широкого диапазона, от всей базы данных до конкретных компонентов отдельных кортежей. Различия между двумя указанными методами кратко описаны ниже.

- В случае **избирательного** контроля каждому пользователю обычно предоставляются различные *права доступа* (иначе называемые **привилегиями,** или **полномочиями)** к разным объектам. Более того, разные пользователи, как правило, обладают раз ными правами доступа к одному и тому же объекту. (Например, пользователю U1 может быть разрешен доступ к объекту А, но запрещен доступ к объекту в, тогда как пользователю U2 может быть разрешен доступ к объекту в, но запрещен доступ к объекту А.) Поэтому избирательные схемы характеризуются значительной гиб костью.
- В случае **мандатного** контроля, наоборот, каждому объекту данных назначается не который **классификационный уровень,** а каждому пользователю присваивается не который уровень **допуска.** В результате право доступа к объекту данных получают только те пользователи, которые имеют соответствующий уровень допуска. Ман датные схемы обычно имеют иерархическую структуру и поэтому являются более жесткими. (Если пользователь U1 имеет доступ к объекту А, но не имеет доступа к объекту в, то в схеме защиты объект в должен будет располагаться на более высоком уровне, чем объект А, а значит, не может существовать никакого пользователя U2, который будет иметь доступ к объектув, но не будет иметь доступа к объекту А.)

Избирательные схемы подробно обсуждаются ниже, в разделе 17.2, а мандатные схемы — в разделе 17.3.

Независимо от того, какая схема используется (избирательная или мандатная), все решения относительно предоставления пользователям прав на выполнение тех или иных операций с теми или иными объектами должны приниматься исключительно управленческим персоналом. Поэтому все эти вопросы выходят за пределы возможностей самой СУБД, и все, что она способна сделать в данной ситуации, — привести в действие решения, которые будут приняты на другом уровне. Исходя из этих соображений, можно определить приведенные ниже условия.

- Принятые организационные решения должны быть доведены до сведения систе мы (т.е. представлены как **ограничения защиты,** выраженные с помощью некото рого языка описания требований защиты) и должны быть ей постоянно доступны (храниться в системном каталоге).
- Очевидно, что в системе должны существовать определенные средства проверки поступающих запросов на получение доступа по отношению к установленным правилам защиты. (Здесь под понятием *запрос на получение доступа* подразумева ется конкретная комбинация *запрашиваемой операции, запрашиваемого объекта* и *запрашивающего пользователя.)* Обычно такая проверка выполняется **подсистемой защиты** СУБД, которую иногда называют также **подсистемой авторизации.**
- Для принятия решения о том, какие именно установленные ограничения защиты применимы к данному запросу на получение доступа, система должна быть способна установить *источник* этого запроса, т.е. суметь опознать *запрашивающего пользова теля.* Поэтому при подключении к системе от пользователя обычно требуется вве сти не только свой **идентификатор** (чтобы указать, кто он такой), но и **пароль** (чтобы подтвердить, что он именно тот, за кого себя выдает). Предполагается, что пароль известен только системе и тем лицам, которые имеют право применять дан ный идентификатор пользователя. Процесс проверки пароля (т.е. проверки того, что пользователи являются теми, за какого себя выдают) называется **аутентификацией,**

Примечание. Следует отметить, что в настоящее время существуют намного более сложные методы аутентификации по сравнению с простой проверкой паролей, в которых для аутентификации применяется целый ряд биометрических устройств: приборы для чтения отпечатков пальцев, сканеры радужной оболочки, анализаторы геометрических характеристик ладони, приборы проверки голоса, устройства распознавания подписей и т.д. Все эти устройства могут эффективно использоваться для проверки "персональных характеристик, которые никто не может подделать" [17.6].

Кстати, в отношении идентификаторов пользователей следует заметить, что один и тот же идентификатор может совместно применяться для целого ряда разных пользователей, входящих в состав некоторой группы. Таким образом, система может поддерживать **группы пользователей** (называемые также *ролями),* обеспечивая одинаковые права доступа для всех ее членов, например, для всех работников бухгалтерского отдела. Кроме того, операции добавления новых пользователей в группу или их удаления из нее можно выполнять независимо от операций задания привилегий доступа для этой группы на те или иные объекты. В этой связи следует обратить внимание на работу [17.11], в которой описана система с *вложенными* пользовательскими группами. Приведем цитату из этой работы: "Способность классифицировать пользователей в виде иерархии групп представляет собой мощный инструмент администрирования больших систем с тысячами пользователей и объектов". Но следует отметить, что наиболее удобным местом для регистрации информации о том, какие пользователи относятся к тем или иным группам, является также системный каталог (или, возможно, сама база данных), причем и эта информация должна быть, безусловно, объектом применения подходящих средств защиты.

17.2. ИЗБИРАТЕЛЬНАЯ СХЕМА УПРАВЛЕНИЯ ДОСТУПОМ

Повторяя сказанное выше, следует еще раз отметить, что во многих СУБД поддерживается либо избирательная, либо мандатная схема управления доступом, либо оба типа доступа одновременно. Однако точнее все же будет сказать, что в действительности в большинстве СУБД поддерживается только избирательная схема доступа и лишь в некоторых — только мандатная. Поскольку на практике избирательная схема доступа встречается гораздо чаще, основное внимание в этой главе уделено именно ей.

Как уже отмечалось, для определения избирательных ограничений защиты необходимо использовать некоторый язык. По вполне очевидным причинам гораздо легче указать то, что *разрешается,* чем то, что *не разрешается.* Поэтому в подобных языках обычно поддерживается определение не самих ограничений защиты, а **полномочий,** которые по своей сути противоположны ограничениям защиты (т.е. они разрешают какие-либо действия, а не запрещают их). Дадим краткое описание гипотетического языка определения полномочий[2], воспользовавшись следующим примером.

```
AUTHORITY SA3
   GRANT RETRIEVE { S#, SNAME, CITY }, DELETE
   ON S
   TO Jim, Fred, Mary ;
```

[2] В современной версии языка Tutorial D [3.3] намеренно не предусмотрено никаких средств определения полномочий, однако используемый в данном разделе гипотетический язык можно рассматривать как "выдержанный в духе" языка Tutorial D.

Этот пример иллюстрирует тот факт, что в общем случае полномочия доступа включают четыре описанных ниже компонента.

1. Имя (в данном примере SA3, "suppliers authority three" — полномочия поставщика с номером 3). Устанавливаемые полномочия будут зарегистрированы в системном каталоге под этим именем.
2. Одна или несколько привилегий, задаваемых в конструкции GRANT.
3. Задаваемое в конструкции ON имя переменной отношения, к которой применяются полномочия.
4. Множество пользователей (точнее, *идентификаторов пользователей),* которым предоставляются указанные привилегии применительно к указанной переменной отношения, задаваемой с помощью фразы TO.

Ниже приводится общий синтаксис оператора определения полномочий.

```
AUTHORITY <authority name>
   GRANT <privilege commalist>
   ON <relvar name>
   TO <user ID commalist> ;
```

Пояснения. В приведенном выше определении смысл таких параметров, как имя полномочия *<authority name>,* имя переменной отношения *<relvar name>* и разделенный запятыми список идентификаторов пользователей *<user ID commalist>,* понятен без дополнительных комментариев. Добавим только, что значение ALL в данном контексте может применяться для обозначения сразу всех известных пользователей. В качестве элементов списка в параметре *<privilege>* с обозначением полномочия разрешается применять следующие значения.

```
RETRIEVE [      {  <attribute name commalist>  }  ] •
INSERT [ {      <attribute name commalist>  }  ]
DELETE
UPDATE [ {      <attribute name commalist>  }  ]
ALL
```

Смысл привилегий RETRIEVE (выборка, без уточнения), INSERT (вставка, без уточнения), DELETE (удаление) и UPDATE (обновление, без уточнения) очевиден без дополнительных пояснений (возможно, что это не совсем так; следует отметить, что привилегия RETRIEVE требуется также просто для указания соответствующего объекта, например, в определении представления или в ограничении целостности, а также для выборки как таковой). Если с ключевым словом RETRIEVE задан разделенный запятыми список атрибутов *<attribute name commalist>,* то эта привилегия распространяется только на указанные атрибуты. Разделенные запятыми списки имен атрибутов в определениях привилегий INSERT и UPDATE имеют тот же смысл. Спецификация ALL является сокращенной записью предоставления *всех* привилегий — RETRIEVE (по всем атрибутам), INSERT (по всем атрибутам), DELETE и UPDATE (по всем атрибутам).

Примечание. Для простоты здесь игнорируется проблема применения некоторых особых привилегий для выполнения общих реляционных операций присваивания. Мы также ограничим рассмотрение операциями *манипулирования данными,* хотя на практике, безусловно, существует много других операций, для которых следовало бы проверять наличие соответствующих полномочий (например, определение и удаление переменных

отношения, а также определение и удаление самих полномочий). Подробное описание этих операций исключено для экономии места.

Что произойдет, если пользователь попытается выполнить с объектом операцию, не имея на это необходимых полномочий? В простейшем случае достаточно просто отвергнуть эту попытку (безусловно, предоставив пользователю соответствующее диагностическое сообщение). На практике такой вариант применяется наиболее часто, поэтому его можно понимать как используемый по умолчанию. Однако в более сложных ситуациях может оказаться полезным использовать некоторое другое действие, например завершить выполнение программы или заблокировать клавиатуру пользователя. Также может иметь смысл фиксация всех подобных попыток в специальном журнале регистрации (т.е. *отслеживание угроз).* Впоследствии эти данные могут оказаться полезными для выявления попыток обнаружения брешей в системе защиты, а также следов возможного нелегального проникновения в систему (вопросы ведения такого *контрольного журнала,* позволяющего следить за всеми процессами в системе, подробно обсуждаются в конце этого раздела).

Безусловно, необходимо также предусмотреть способ отмены существующих полномочий, как показано ниже.

DROP AUTHORITY *<authority name>* ;

Для простоты изложения предположим, что при удалении некоторой переменной отношения автоматически удаляются и все полномочия, предоставленные по отношению к ней.

Ниже представлено несколько примеров определения полномочий; большинство этих примеров не нуждается в дополнительных пояснениях.

1. AUTHORITY EX1
```
GRANT RETRIEVE ( P#, PNAME, WEIGHT )
ON P
TO Jacques, Anne, Charley ;
```
Пользователи Jacques, Anne и Charley получают право доступа к указанному *вертикальному подмножеству* данных в базовой переменной отношения P. Таким образом, это — пример предоставления полномочий, **не зависящих от значений** данных.

2. AUTHORITY EX2
```
GRANT RETRIEVE, DELETE, UPDATE (SNAME, STATUS)
ON LS
TO Dan, Misha ;
```
Упоминаемая здесь переменная отношения LS является представлением (с данными о поставщиках из Лондона; см. рис. 10.4 в главе 10). Пользователи Dan и Misha получают доступ к *горизонтальному подмножеству* данных в базовой переменной отношения S. Это — пример предоставления полномочий, **зависящих** от **значений** данных. Обратите внимание на то, что хотя пользователи Dan и Misha могут удалить (DELETE) некоторые кортежи поставщиков (через представление LS), они не могут вставить (INSERT) их, а также не имеют права обновлять (UPDATE) атрибут s# или CITY.

3.
```
VAR SSPPO VIEW
 (S JOIN SP JOIN (P WHERE CITY = 'Oslo') { P# } )
                                  { ALL BUT P#, QTY } ;

AUTHORITY EX3
   GRANT RETRIEVE
   ON SSPPO TO
   Lars ;
```

Это еще один пример полномочий, зависящих от значений данных: пользователь Lars получает доступ к информации о поставщиках, но только **о** тех, котррые поставляют детали, хранящиеся в Осло.

4.
```
VAR SSQ VIEW
    SUMMARIZE SP PER S { S# } ADD SUM (QTY) AS SQ ;

AUTHORITY EX4
GRANT RETRIEVE
ON SSQ TO Fidel
;
```

Пользователь Fidel получает право просматривать итоговые данные по каждому поставщику, но не имеет доступа к данным о каждой поставке в отдельности. Таким образом, пользователь Fidel сможет просматривать в базе только результаты **статистической обработки** данных.

5.
```
AUTHORITY EX5
   GRANT RETRIEVE, UPDATE     (STATUS)
   ON S
   WHEN DAY () IN ('Мои',     'Tue', 'Wed', 'Thu', 'Fri')
        AND NOW () > TIME     '09:00:00'
        AND NOW () < TIME
                         '17:00:00''
   TO   ACCOUNTING ;
```

Здесь синтаксис определения полномочий AUTHORITY расширен для включения конструкции WHEN, позволяющей задать некоторые *контекстные элементы управления.* Дополнительно предполагается, что в системе предусмотрены два *нуль-арных оператора* (так называются операторы, не имеющие явно заданных операндов), которые называются DAY() и NOW O и имеют очевидную интерпретацию. Определение полномочий EX5 гарантирует, что значение статуса поставщика может быть изменено любым пользователем из группы ACCOUNTING (работники бухгалтерского отдела) только в рабочие дни и в рабочее время. Это — пример так называемого **контекстно-зависимого** полномочия, поскольку поступивший запрос на доступ будет или не будет нарушать установленные ограничения в зависимости от состояния контекста (в данном случае это комбинация дня недели и времени суток).

Ниже приведены примеры других встроенных функций, которые могут быть полезны при определении контекстно-зависимых полномочий.

- TODAY (). Возвращает текущую дату
- USER (). Возвращает идентификатор пользователя, выдавшего запрос.
- TERMINAL (). Возвращает идентификатор терминала, с которого поступил запрос.

С концептуальной точки зрения несколько полномочий объединяются одно с другим по принципу связи логических утверждений с помощью оператора "ИЛИ". Иначе говоря, поступивший запрос на получение доступа (включающий комбинацию запрашиваемой операции, запрашиваемого объекта и имени запрашивающего пользователя) принимается тогда и только тогда, когда он удовлетворяет хотя бы одному из существующих полномочий. Например, если в одном определении полномочий утверждается, что пользователю Nancy разрешается считывать сведения о цвете деталей, а в другом определении указывается, что этот же пользователь имеет право доступа к сведениям о весе деталей, это не означает, что ему разрешается считывать сведения о цвете и весе деталей одновременно (для этого потребуется определить дополнительное полномочие).

В заключение необходимо отметить следующее: выше предполагалось, хотя явно нигде не утверждалось, что пользователи могут выполнять только те действия, которые им позволено выполнять предоставленными полномочиями. При таком подходе к организации защиты все, что не позволено явным образом, неявно запрещено!

Модификация запроса

Для иллюстрации некоторых представленных выше идей целесообразно описать аспекты защиты данных, реализованные в прототипе системы University Ingres, а также особенности используемого в ней языка запросов QUEL. В этой реализации был принят довольно интересный подход к решению проблем защиты, заключающийся в том, что любой запрос на языке QLJEL перед выполнением автоматически модифицировался таким образом, чтобы предотвратить любые возможные нарушения установленных ограничений защиты. В качестве примера предположим, что пользователю и разрешено считывать данные только о тех деталях, которые хранятся в Лондоне. Это ограничение задается следующим выражением.

```
DEFINE PERMIT RETRIEVE ON P TO U
      WHERE P.CITY = "London"
```

(Оператор DEFINE PERMIT будет подробно описан ниже.) Теперь предположим, что пользователь и вводит приведенный ниже запрос на языке QUEL.

```
RETRIEVE ( P.P#, P.WEIGHT )
WHERE      P.COLOR = "Red"
```

Используя приведенное выше *разрешение* (PERMIT), которое хранится в системном каталоге и определяет права доступа пользователя и к переменной отношения p, система автоматически преобразует приведенный выше запрос в запрос следующего вида.

```
RETRIEVE ( P.P#, P.WEIGHT )
WHERE      P.COLOR = "Red" AND
P.CITY  = "London"
```

Безусловно, модифицированный подобным образом запрос вряд ли сможет нарушить установленное ограничение защиты. К тому же процесс модификации происходит совершенно *незаметно,* т.е. пользователь и не получает информации о том, что фактически система выполнила запрос, который несколько отличается от исходного. Дело в том, что сокрытие этой информации может быть само по себе достаточно важным (например, пользователю и не следует даже знать о том, что существуют детали, которые хранятся не в Лондоне).

Кратко описанный выше процесс **модификации запроса** полностью идентичен методу, используемому для реализации представлений [10.12], а также (особенно в случае прототипа системы Ingres) ограничений целостности [9.23]. Таким образом, важное преимущество данного подхода заключается в том, что его можно реализовать достаточно просто (в основном благодаря тому, что необходимый для этого программный код уже присутствует в системе). Другое его преимущество — сравнительно высокая эффективность, так как увеличение издержек, связанных с реализацией ограничений защиты, происходит в основном во время компиляции, а не во время прогона программы. Еще одно преимущество данного подхода состоит в том, что при его использовании отсутствуют определенные неудобства, свойственные подходу, описанному ранее, когда для одного пользователя требовалось задавать разные привилегии при обращении к разным частям одной и той же переменной отношения (см. раздел 17.6, где дано пояснение к сказанному).

Единственный *недостаток* этого подхода заключается в том, что не всем ограничениям защиты можно придать столь простую форму. В качестве простейшего контрпримера предположим, что пользователю U запрещен доступ к переменной отношения P. Тогда любая достаточно простая *модифицированная* форма приведенного выше определения полномочий на выборку данных RETRIEVE непременно создаст иллюзию, что переменной отношения P просто не существует. В результате система обязательно выдаст сообщение об ошибке приблизительно следующего содержания: "Вам не разрешен доступ к указанной переменной отношения". (Или, вероятно, система просто *скроет истину* и сообщит: "Такой переменной отношения не существует".) Но самый лучший способ заключается в том, чтобы сообщить пользователю: "Либо такой переменной отношения не существует, либо вы не имеете права доступа к ней".

В общем виде синтаксис оператора определения полномочий DEFINE PERMIT выглядит следующим образом.

```
DEFINE PERMIT «operation name commalist>
       ON <relvar name> [ ( <attribute name commalist> )
       ] TO <user ID>
     [ AT <terminal ID commalist> ]
     [ FROM <time> TO <time> ]
     [ ON <day> TO <day> }
     [ WHERE <bool exp> ]
```

Таким образом, концептуально синтаксис оператора определения полномочий DEFINE PERMIT очень похож на синтаксис оператора определения полномочий AUTHORITY, за исключением того, что он обеспечивает применение дополнительных условий в конструкции WHERE (все конструкции AT, FROM и ON могут быть заменены конструкцией WHEN). Ниже приведен пример подобного определения.

```
DEFINE PERMIT RETRIEVE, APPEND,
       REPLACE ON S ( S#, CITY ) TO
       Joe AT TTA4
       FROM 9:00 TO 17:00
       ON Sat TO Sun
       WHERE S.STATUS < 50
       AND S.S# = SP.P#
       AND SP.P# = P.P#
       AND P.COLOR = "Red"
```

Примечание. Операторы добавления (APPEND) и замены (REPLACE) языка QUEL являются аналогами операторов вставки (INSERT) и обновления (UPDATE), соответственно.

Контрольный журнал

Важно понимать, что непреодолимых систем защиты не бывает. Настойчивый потенциальный нарушитель всегда сможет преодолеть все установленные системы контроля, особенно если он рассчитывает получить за это достаточно высокое вознаграждение. Поэтому при работе с очень важными данными или при выполнении ответственных операций возникает необходимость организации **контрольного журнала** (audit trail), в который вносится информация обо всех событиях, происходящих в системе. Допустим, например, что обнаружена противоречивость полученных результатов, а это может свидетельствовать о злонамеренном искажении информации, хранящейся в базе данных. В этом случае для прояснения ситуации могут использоваться записи контрольного журнала, которые позволят либо подтвердить тот факт, что все процессы находятся под контролем, либо обнаружить вмешательство и распознать нарушителя.

Контрольный журнал, по сути, представляет собой особого рода файл **или** базу данных, в которую СУБД автоматически помещает сведения обо всех операциях, выполненных пользователями при работе с основной базой данных. В одних системах данные контрольного журнала могут физически записываться в файл журнала восстановления (см. главу 15), а в других системах эти сведения помещаются в отдельный файл. В любом случае, пользователям должен быть предоставлен механизм доступа к информации контрольного журнала, желательно с помощью средств обычного языка реляционных запросов (разумеется, при условии, что эти запросы санкционированы!). Типичная запись в файле контрольного журнала может содержать такую информацию:

- сам запрос (исходный текст запроса);
- номер терминала, с которого была затребована операция;
- имя пользователя, затребовавшего операцию;
- дата и время запуска операции;
- переменные отношения, кортежи и атрибуты, вовлечённые в процесс выполнения операции;
- исходные значения изменяемых данных (старые значения);
- модифицированные значения данных (новые значения).

Как уже упоминалось, даже сам факт, что в системе ведется контрольный журнал, в определенных ситуациях способен отпугнуть потенциального нарушителя.

17.3. МАНДАТНАЯ СХЕМА УПРАВЛЕНИЯ ДОСТУПОМ

Методы мандатного управления доступом применяются к тем базам данных, в которых хранимая информация имеет достаточно статичную и жесткую структуру, что свойственно, например, некоторым военным или правительственным организациям. Как отмечалось выше, в разделе 17.1, основная идея состоит в том, что каждому объекту данных присваивается некоторый **классификационный уровень** (classification level) (или требуемый **гриф секретности,** например "Совершенно секретно", "Секретно", "Для служебного пользования" и т.д.), а каждому пользователю предоставляется **уровень допуска** (clearance level) с градациями, аналогичными существующим классификационным уровням. Предполагается, что эти уровни образуют строгую иерархическую систему (например,

"Совершенно секретно" > "Секретно" > "Для служебного пользования" и т.д.)- Тогда исходя из этих положений можно сформулировать два очень простых правила, впервые предложенные Беллом (Bell) иЛа-Падулой (LaPadula) [17.3].

1. Пользователь i может выполнить выборку данных объекта j только в том случае, если его уровень допуска больше классификационного уровня объекта j или равен ему *{простое свойство безопасности* — simple security property).
2. Пользователь i может модифицировать объект j только в том случае, если его уро вень допуска равен классификационному уровню объекта j *(звездное свойство* — star property).

Первое правило достаточно очевидно, тогда как второе требует дополнительных пояснений. Прежде всего, следует отметить, что иным образом второе правило можно сформулировать так: "По определению любая информация, записанная пользователем i, автоматически приобретает классификационный уровень, который равен уровню допуска пользователя i". Подобное правило необходимо, например, для того, чтобы предотвратить запись секретных данных, выполняемую пользователем с уровнем допуска "Секретно", в файл с меньшим уровнем классификации, что нарушит всю систему секретности.

Примечание. В отношении только операций собственно *записи* (INSERT), во втором правиле достаточно было бы потребовать, чтобы уровень допуска пользователя i был *меньше* классификационного уровня объекта j *или равен* ему, и именно это правило часто приводится в литературе. Но тогда пользователи могли бы записывать то, что потом сами не смогли бы прочесть! (Хотя бывает так, что кое-кто с трудом может прочесть даже собственный рукописный текст... Возможно, применение более слабой формы второго правила все-таки не совсем лишено смысла.)

Особенно большое внимание методам мандатного управления доступом стало уделяться в начале 1990-х годов. Дело в том, что согласно требованиям Министерства обороны США, любая используемая в этом ведомстве СУБД должна непременно поддерживать схему мандатного управления доступом. Поэтому разработчикам СУБД пришлось вступить в соперничество за скорейшую разработку методов такого управления. Обязательные требования к этой схеме были изложены в двух важных публикациях Министерства обороны, получивших неформальные названия **"Оранжевая" книга** (Orange Book) [17.21] и **"Сиреневая" книга** (Lavender Book) [17.22]. В "Оранжевой" книге перечислен набор требований защиты для некоторой *надежной вычислительной базы* (Trusted Computing Base — TCB), а в "Сиреневой" книге дается интерпретация этих требований в отношении систем баз данных.

Определенные в "Оранжевой" и "Сиреневой" книгах методы мандатного управления доступом на самом деле являются частью более общей классификации уровней защиты, которые в очень сжатой форме излагаются ниже. Прежде всего, в этих документах определяются четыре **класса безопасности** — A, **B,** C и D. Грубо говоря, класс D среди них наименее безопасен, класс C — более безопасен, чем класс D, и т.д. Говорят, что класс D обеспечивает *минимальную,* класс C — *избирательную,* класс B — *мандатную* и класс A — *проверенную* защиту. Ниже кратко рассматриваются классы C, **B** и **A.**

- **Избирательная защита.** Класс C делится на два подкласса, C1 и C2 (где подкласс C1 *менее* безопасен, чем подкласс C2), каждый из которых поддерживает избирательное управление доступом, в том смысле, что управление доступом осуществляется по

усмотрению *владельца* данных (точно так, как описано выше, в разделе 17.2). Кроме того, эти классы характеризуются указанными ниже особенностями.

- В соответствии с требованиями класса C1, необходимо отделить права владения от прав доступа, т.е. наряду с поддержкой концепции совместного доступа к данным пользователям разрешается иметь собственные защищенные данные.
- В соответствии с требованиями класса C2, необходимо дополнительно организовать поддержку учетных записей, построенную на основе процедур применения подписей, аудита и изоляции ресурсов.

- **Мандатная защита.** Класс B включает требования к методам мандатного управления доступом и делится натри подкласса — B1, B2 и B3 (где B1 является наименее, а B3 — наиболее безопасным подклассом). Определение указанных подклассов приведены ниже.
  - В соответствии с требованиями класса B1, необходимо организовать *защиту с использованием меток* (это означает, что каждый объект данных в системе должен иметь метку с обозначением присвоенного ему классификационного уровня — "Секретно", "Для служебного пользования" и т.д.). Дополнительно требуется также предусмотреть *неформальное* описание правил защиты.
  - В соответствии с требованиями класса B2, дополнительно требуется предусмотреть *формальное* описание правил защиты. Кроме того, необходимо обеспечить обнаружение и исключение *каналов утечки информации.* Таким каналом может быть, например, возможность логического вывода ответа на недопустимый запрос из ответа на допустимый запрос (см. раздел 17.4) или возможность раскрытия секретных сведений на основании данных о времени, которое затрачивается на выполнение некоторых допустимых вычислений (см. аннотацию к [17.14]).
  - В соответствии с требованиями класса B3, необходимо дополнительно обеспечить поддержку аудита и восстановления данных, а также назначить *администратора защиты.*
- **Проверенная защита.** Класс A является наиболее безопасным, и, согласно его требованиям, необходимо математическое доказательство того, что выбранный механизм защиты является приемлемым и обеспечивает адекватную поддержку установленных ограничений защиты (!).

В настоящее время некоторые коммерческие СУБД поддерживают мандатную схему защиты уровня B1. Кроме того, они обычно поддерживают избирательную схему защиты уровня C2.

Терминология. СУБД, в которых поддерживается мандатная схема защиты, часто называют **системами с многоуровневой защитой** [17.15], [17.18], [17.23] (см. следующий подраздел, "Многоуровневая защита"). В этом же смысле иногда используется термин **система, заслуживающая доверия** (trusted system) [17.19], [17.21], [17.22].

Многоуровневая защита

Допустим, что требуется применить идеи мандатной схемы управления доступом к переменной отношения поставщиков S. Для определенности и простоты будем считать, что единицей данных, на уровне которой требуется контролировать доступ, является отдельный кортеж этой переменной отношения. Тогда каждый кортеж должен быть отмечен

Рис. 17.1. Переменная отношения S с присвоенными значениями классификационного уровня (пример)

соответствующим классификационным уровнем, например так, как показано на рис. 17.1. (Здесь значение 4 в столбце LEVEL означает уровень "Совершенно секретно", 3 — "Секретно", 2 — "Для служебного пользования".)

S

S#	SNAME	STATUS	CITY	LEVEL
S1	Smith	20	London	2
S2	Jones	10	Paris	3
S3	Blake	30	Paris	2
S4	Clark	20	London	4
S5	Adams	30	Athens	3

Теперь предположим, что пользователи из и U2 имеют уровни доступа 3 ("Секретно") и 2 ("Для служебного пользования"), соответственно. Тогда переменная отношения S для этих пользователей будет выглядеть по-разному! Запрос на выборку сведений обо всех поставщиках со стороны пользователя из возвратит четыре кортежа с данными о поставщиках с номерами S1, S2, S3 и S5. Аналогичный запрос со стороны пользователя U2 возвратит два кортежа с данными о поставщиках с номерами S1 и S3. Более того, в результатах выполнения *обоих* запросов будут отсутствовать сведения о поставщике с номером S4.

Разобраться в приведенных выше утверждениях можно, применив метод *модификации запроса.* Рассмотрим приведенный ниже запрос ("Получить сведения о поставщиках из Лондона").

```
S WHERE CITY = 'London'
```

Система модифицирует этот запрос и приводит его к следующему виду.

```
S WHERE CITY = 'London' AND LEVEL < допуск пользователя
```

Аналогичные соображения будут справедливы и по отношению к операциям обновления. Например, пользователь из не знает о существовании кортежа для поставщика с номером S4, поэтому приведенная ниже команда INSERT может показаться ему вполне приемлемой.

```
INSERT INTO S RELATION { TUPLE { S#     S# ( 'S4' ),
                                 SNAME  NAME ( 'Baker' ),
                                 STATUS 25,
                                 CITY   'Rome' } } ;
```

Система не должна отвергать команду INSERT, поскольку в этом случае пользователь U3 в конечном счете узнает о существовании поставщика с номером S4. Такую команду система примет, но модифицирует ее и приведет к следующему виду.

```
INSERT INTO S RELATION { TUPLE {S#      S#('S4'),
                                 SNAME  NAME (
                                 'Baker' )> STATUS 25,
                                 CITY   'Rome', LEVEL
                                 3 } } ;
```

Обратите внимание на то, что в данном случае первичным ключом для переменной отношения поставщиков является уже не атрибут {S#}, а комбинация атрибутов {S#,LEVEL}.

Примечание. Для простоты предполагается, что существует только один потенциальный ключ, который в этом случае можно рассматривать как *первичный* ключ.

Еще о терминологии. Модифицированная указанным образом переменная отношения поставщиков является примером *многоуровневой переменной отношения.* Та особенность, что *одни и те же* данные по-разному выглядят для разных пользователей, называется *поликонкретизацией* (polyinstantiation). В приведенном выше примере с оператором INSERT запрос на извлечение данных о поставщике с номером S4 возвратит разные результаты для пользователя U4 с правом доступа к совершенно секретным материалам и для пользователя U3, имеющего лишь право доступа к секретным материалам. Третий результат, отличный от двух предыдущих, будет предоставлен пользователю U2, обладающему правом доступа к материалам для служебного пользования.

Операторы обновления (UPDATE) и удаления (DELETE) обрабатываются системой аналогично (здесь мы не будем их обсуждать, поскольку более подробно они рассматриваются в некоторых работах, перечисленных в списке литературы для этой главы).

Вопрос. Как вы думаете, не нарушают ли эти подходы упомянутый выше *информационный принцип!* Обоснуйте свой ответ.

17.4. СТАТИСТИЧЕСКИЕ БАЗЫ ДАННЫХ

Статистической (в приведенном здесь контексте) называется база данных, в которой допускаются запросы с агрегированием данных (суммированием, вычислением среднего значения и т.д.), но не допускаются запросы по отношению к элементарным данным. Например, в статистической базе данных разрешается выдача запроса "Какова средняя зарплата служащих?", тогда как выдача запроса "Какова зарплата служащего Мэри?" запрещена.

Проблема статистических баз данных заключается в том, что иногда с помощью логических заключений на основе выполнения разрешенных запросов можно вывести ответ, который прямо может быть получен только с помощью запрещенного запроса. Как указывается в [17.8], "Обобщенные значения содержат следы исходной информации, и она может быть восстановлена злоумышленником после соответствующей обработки достаточного количества этих обобщенных значений. Такой процесс называется *дедуктивным формированием конфиденциальной информации с помощью логического вывода".* Следует отметить то, что эта проблема, к сожалению, становится все более и более важной по мере того, как использование *хранилищ данных* (см. главу 22) получает все большее распространение.

Рассмотрим более подробный пример. Предположим, что в базе данных содержится только одна переменная отношения, STATS, представленная на рис. 17.2. Предположим также для простоты, что все атрибуты определены на основе примитивных типов данных (т.е. числового и строкового). Далее предположим, что некоторому пользователю и в этой базе данных разрешено выполнять только статистические запросы, но он поставил себе целью узнать размер зарплаты работника по имени Alf. Наконец предположим, что из других источников пользователь и узнал, что Alf — программист мужского пола. Проанализируем представленные ниже запросы[3].

---

[3] Для сокращения объема текста все запросы данного раздела представлены на языке Tutorial D в сокращенной форме. Например, в запросе 1 выражение COUNT (x) следовало бы представить в более полной форме (скажем) как EXTEND TABLE_DEE ADD COUNT(X) AS RESULT1.

```
1. WITH ( STATS WHERE SEX = 'M' AND .
          OCCUPATION =  'Programmer'   )
   AS  X  : COUNT  (  X  )
```

Результат: 1.

```
2. WITH  (  STATS  WHERE  SEX  =  'M'  AND
          OCCUPATION = 'Programmer' ) AS X
   : SUM  (  X,  SALARY  )
```

Результат: 50 000.

NAME	SEX	CHILDREN	OCCUPATION	SALARY	TAX	AUDITS
Alf	M	3	Programmer	50K	10K	3
Bea	F	2	Physician	130K	10K	0
Cyn	F	0	Programmer	56K	18K	1
Dee	F	2	Builder	60K	12K	1
Ern	M	2	Clerk	44K	4K	0
Fay	F	1	Artist	30K	0K	0
Guy	M	0	Lawyer	190K	0K	0
Hal	M	3	Homemaker	44K	2K	0
Ivy	F	4	Programmer	64K	10K	1
Joy	F	1	Programmer	60K	20K	1

Рис. 17.2. Переменная отношения STATS (примеры значений)

Очевидно, что защита базы данных была нарушена, хотя пользователь и пряменя только разрешенные ему статистические запросы. Как показано в примере, если пользователь сумеет найти такое логическое выражение, которое позволит ему идентифицировать некоторую личность, то информация о данной личности окажется незащищенной. По этой причине система должна отвергать любые запросы, для которых кардинальность обобщаемого множества окажется меньше некоторого установленного минимального значения Ь. Это также означает, что система должна отвергать запросы, кардинальность обобщаемого множества которых превосходит значение п-Ь (где п — кардинальность исходной переменной отношения). Дело в том, что можно привести еще один пример нарушения защиты этой базы данных, осуществляемого посредством ввода показанной ниже последовательности запросов 3-6.

```
3. COUNT  (  STATS  )
```

Результат: 12.

```
4. WITH  (  STATS  WHERE  NOT  (  SEX  =  'M'  AND
          OCCUPATION =  'Programmer'   )   )
   AS  X   : COUNT  (  X  )
```

Результат: 11; 12 - 11 = 1.

```
5. SUM  (  STATS,  SALARY  )
```

Результат: 728 000.

```
6. WITH  (  STATS  WHERE  NOT  (  SEX  =  'M'  AND
          OCCUPATION =  'Programmer'  )  )  AS  X
   :  SUM  (  X,  SALARY  )
```

Результат: 678 000; 728 000 - 678 000 = 50 000.

К сожалению, можно легко показать, что в общем нарушения защиты нельзя предотвратить, просто разрешая выполнение лишь таких запросов, в которых агрегируемое множество имеет кардинальность с в диапазоне b < c < п - Ь. Еще раз обратимся к примеру на рис. 17.2. Если b = 2, то в таком случае будут разрешены запросы только с кардинальностью с, находящейся в диапазоне 2 < с < 8. Это означает, что следующее логическое выражение использовать нельзя.

```
SEX = 'M' AND OCCUPATION = 'Programmer'
```

Теперь рассмотрим приведенную ниже последовательность запросов.

```
7. WITH ( STATS WHERE SEX = 'M' ) AS
   X :
   COUNT ( X )
```

Результат: 4.

```
8. WITH ( STATS WHERE SEX = 'M' AND NOT
                ( OCCUPATION. = 'Programmer' ) ) AS X :
   COUNT ( X )
```

Результат: 3.

На основании запросов 7 и 8 пользователь и может сделать вывод, что существует только один мужчина-программист, следовательно, это и есть Alf (поскольку пользователь и уже знает из других источников, что сотрудник по имени Alf является мужчиной и программистом). Зарплата этого сотрудника может быть определена следующим образом.

```
9. WITH ( STATS WHERE SEX = 'M' ) AS
   X :
   SUM ( X, SALARY )
```

Результат: 328 000.

```
10. WITH ( STATS WHERE SEX = 'M' AND NOT
                 ( OCCUPATION = 'Programmer' ) ) AS X
    : SUM ( X, SALARY )
```

Результат: 278 000; 328 000 - 278 000 = 50 000.

Логическое выражение SEX = 'M' AND OCCUPATION = ' Programmer' называется **индивидуальным** средством **слежения** (individual tracker) для человека по имени Alf [17.8], поскольку оно позволяет отыскать информацию об отдельном человеке по имени Alf. В общем случае, если пользователю известно некоторое логическое выражение BE, идентифицирующее некоторого человека I, и если логическое выражение BE может быть выражено в виде BE1 AND BE2, то логическое выражение BE1 AND NOT BE2 является **индивидуальным средством слежения** для человека с идентификатором I (при условии, что в системе разрешено использовать выражения BE1 и BE1 AND NOT BE2, т.е. оба они приводят к результату, кардинальность с которого находится в диапазоне b < c < п - b). Причина этого заключается в том, что определенное с помощью BE множество идентично разности между множеством, определенным с помощью BE1, и множеством, определенным с помощью BE1 AND NOT BE2, что наглядно представлено на рис. 17.3.

```
{ X : BE }  ≡ { x : BE1 AND BE2 }
            ≡ {x : BE1 } MINUS { X : BE1 AND NOT BE2 }
```

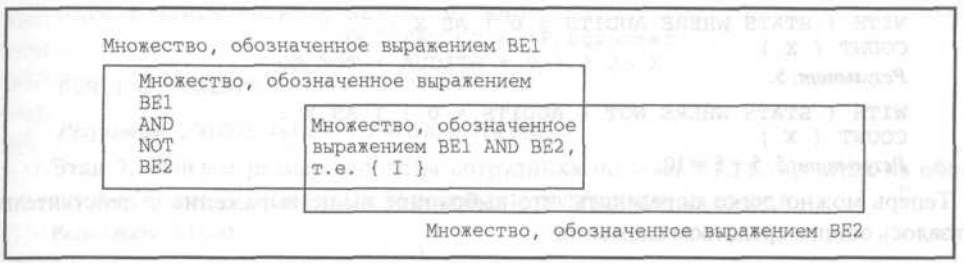


Рис. 17.3. Индивидуальное средство слежения BE1 AND NOT BE2

В [17.8] упомянутые здесь идеи обобщены и показано, что *практически для любой* статистической базы данных всегда могут быть определены общие средства слежения (в отличие от множества средств слежения за личными данными). Общее средство слежения (general tracker) — это логическое выражение, которое может быть использовано для поиска ответа на *любой* запрещенный запрос, т.е. запрос, включающий недопустимое логическое выражение. (В противоположность этому, индивидуальное средство слежения работает только на основе запросов, включающих *конкретные* запрещенные выражения.) Фактически оказывается, что любое логическое выражение с результирующим набором данных с кардинальностью c в диапазоне $2b < c < n - 2b$ является общим средством слежения (здесь b должно быть меньше, чем n/4, что обычно всегда выполняется на практике). Как только такое средство слежения будет найдено, сразу же можно будет получить ответ на запрос, включающий запрещенное логическое выражение BE, что наглядно показано в следующем примере. (Для определенности рассмотрим случай, в котором кардинальность результирующего множества логического выражения BE меньше b. Случай, когда b больше n - b, обрабатывается аналогично.) Обратите внимание на то, что по определению t является общим средством слежения тогда и только тогда, когда NOT t также является общим средством слежения.

Пример. Предположим снова, что b = 2; тогда общим средством слежения будет любое логическое выражение с кардинальностью результирующего множества c в диапазоне $4 < c < 6$. Еще раз предположим, что пользователю u известно из внешних источников, что сотрудник по имени Alf является мужчиной и программистом, т.е. запрещенное логическое выражение BE выглядит так, как показано ниже.

```
SEX = 'M' AND OCCUPATION = 'Programmer'
```

Допустим также, что пользователь и хочет узнать размер зарплаты сотрудника Alf. В этом случае общее средство слежения придется применить дважды: сначала, чтобы удостовериться, что BE однозначно идентифицирует сотрудника по имени Alf (этапы 2—4), а затем непосредственно для того, чтобы определить размер зарплаты сотрудника Alf (этапы 5-7).

Этап 1. Попробуем найти выражение для общего средства слежения t. В качестве первого приближения выберем следующее выражение.

```
AUDITS = 0
```

Этап 2. С помощью выражений t и NOT t определим общее количество сотрудников, сведения о которых содержатся в базе данных.

```
WITH ( STATS WHERE AUDITS = 0 ) AS
X : COUNT ( X )
```

Результат: 5.

```
WITH ( STATS WHERE NOT ( AUDITS = 0 ) ) AS
X : COUNT ( X )
```

Результат: 5; 5 + 5 = 10. Теперь можно легко определить, что выбранное выше выражение **т** действительно оказалось общим средством слежения.

Этап 3. Найдем результат сложения общего количества сотрудников в базе данных с количеством сотрудников, для которых удовлетворяется неразрешенное выражение BE, для чего используем выражения BE OR T и BE OR NOT T.

```
WITH ( STATS WHERE ( SEX = 'M' AND
     OCCUPATION = 'Programmer' OR AUDITS =
0 ) AS X : COUNT ( X )
```

Результат: 6.

```
WITH ( STATS WHERE ( SEX = 'M' AND
     OCCUPATION = 'Programmer' OR NOT (
AUDITS = 0 ) ) AS X : COUNT ( X )
```

Результат: 5; 6 + 5 = 11.

Этап 4. На основании полученных выше результатов можно сделать заключение, что количество сотрудников, к которым относится логическое выражение BE, равно единице (результат этапа 3 минус результат этапа 2), т.е. логическое выражение BE однозначно идентифицирует сотрудника по имени Alf.

Теперь на этапах 5 и 6 повторим запросы, использованные на этапах 2 и 3, но вместо [ии COUNT укажем функцию SUM.

Этап 5. С помощью выражений **т** и NOT T найдем размер зарплаты всех сотрудников в базе данных.

```
WITH ( STATS WHERE AUDITS = 0 ) AS X
: SUM ( X, SALARY )
```

Результат: 438 000.

```
WITH ( STATS WHERE NOT ( AUDITS = 0 ) ) AS
X : SUM ( X, SALARY )
```

Результат: 290 000; 438 000 + 290 000 = 728 000.

Этап 6. Найдем размер зарплаты сотрудника по имени Al f плюс размер зарплаты всех сотрудников, используя выражения BE OR T и BE OR NOT T.

```
WITH ( STATS WHERE ( SEX = 'M' AND
     OCCUPATION = 'Programmer' OR AUDITS =
0 ) AS X : SUM ( X, SALARY )
```

Результат: 488 000.

```
WITH ( STATS WHERE ( SEX = 'M' AND
     OCCUPATION = 'Programmer' OR NOT (
AUDITS = 0 ) ) AS X : SUM ( X, SALARY )
```

Результат: 290 000; 488 000 + 290 000 = 778 000.

Этап 7. Найдем размер зарплаты сотрудника по имени Alf, вычитая из общей суммы (полученной на этапе 5) результат, полученный на этапе 6. *Результат:* 50 000.

Принцип действия показанного ниже общего средства слежения схематически представлен на рис. 17.4.

```
{ X : BE } ≡ ( { X : BE OR T } UNION { X : BE OR NOT T } )
                                MINUS { X : T OR NOT T }
```

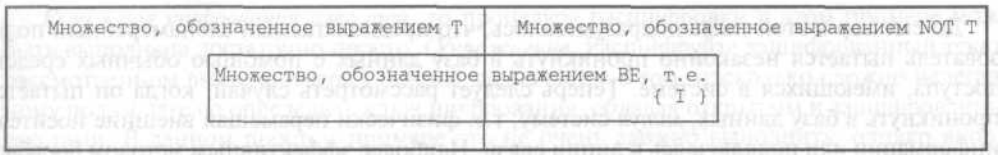


Рис. 17.4. Принцип функционирования общего средства слежения T

Если исходное предположение оказалось неверным (т.е. т не является общим средством слежения), тогда одно или оба выражения (BE OR T) И (BE OR NOT T) МОГУТ оказаться недопустимыми. Например, если кардинальности для результирующих наборов выражений BE и т равны р и q, соответственно, гдер < b и Ь < q < 2b, то вполне возможно, что кардинальность результирующего множества для (BE OR NOT T) больше, чем п - Ь. В такой ситуации необходимо выбрать другой вариант выражения для общего средства слежения и повторить попытку. В [17.8] предполагается, что на практике не так уж сложно найти подходящее выражение для общего средства слежения. В приведенном здесь частном примере исходный вариант сразу *оказался* общим средством слежения (кардинальность его результирующего набора данных равна 5) и оба запроса на этапе 3 также оказались допустимыми.

Обобщим полученные результаты. *Практически всегда* существует выражение, являющееся общим средством слежения, найти которое так же легко, как и использовать. Действительно, выражение для общего средства слежения достаточно быстро можно просто угадать путем перебора нескольких вариантов [17.8]. Даже в тех случаях, когда не существует выражения для общего средства слежения, могут быть найдены, как показано в [17.8], специфические средства слежения, предназначенные для конкретных запросов. Трудно избежать общего заключения, что безопасность статистических баз данных остается насущной проблемой.

Что в таком случае можно сделать? Некоторые варианты решения этой проблемы уже предлагались в литературе, но все же пока нельзя сказать определенно, является ли какой-либо из них удовлетворительным во всех отношениях. Например, в одном из вариантов предлагается организовать *обмен данными* (data swapping), т.е. обмен значениями атрибутов между кортежами, осуществляемый таким образом, чтобы поддерживалась лишь статистическая точность. При этом даже если злоумышленнику удастся идентифицировать отдельное значение (например, некоторое значение зарплаты), то у него не будет

способа узнать, какому именно кортежу (в нашем примере — сотруднику) оно принадлежит. Сложность этого подхода заключается в необходимости отыскать множество тех записей, между которыми можно будет организовать обмен значениями соответствующим образом. Подобные затруднения имеют место и при использовании большинства других предложенных методов. В данный момент, видимо, придется согласиться с выводами, приведенными в [17.8], о том, что "методы нарушения защиты данных просты и не связаны с большими расходами. Поэтому требование обеспечения полной секретности конфиденциальной информации несовместимо с требованием возможности вычисления точных статистических показателей для произвольных подмножеств данных в базе. По крайней мере одно из этих требований должно быть снято прежде, чем можно будет доверять гарантиям обеспечения секретности".

17.5. ШИФРОВАНИЕ ДАННЫХ

До сих пор в этой главе подразумевалось, что предполагаемый злонамеренный пользователь пытается незаконно проникнуть в базу данных с помощью обычных средств доступа, имеющихся в системе. Теперь следует рассмотреть случай, когда он пытается проникнуть в базу данных, *минуя* систему, т.е. физически перемещая внешние носители информации или подключаясь к линии связи. Наиболее эффективным методом борьбы с такими угрозами является **шифрование данных,** т.е. хранение и передача особо важных данных в зашифрованном виде.

Для обсуждения основных концепций шифрования данных следует ввести некоторые новые понятия. Исходные (незашифрованные) данные называются **открытым текстом.** Открытый текст **шифруется** с помощью специального **алгоритма шифрования.** В качестве входных данных для такого алгоритма выступают открытый текст и **ключ шифрования,** а в качестве выходных— преобразованная форма открытого текста, которая называется **шифрованным текстом.** Детали алгоритма шифрования могут быть опубликованы или, по крайней мере, могут не утаиваться, но ключ шифрования ни в коем случае не разглашается. Именно зашифрованный текст, непонятный всем, кто не обладает ключом шифрования, хранится в базе данных и передается по линии связи.

Пример. Пусть в качестве открытого текста дана следующая строка.

```
AS KINGFISHERS CATCH FIRE
```

(Здесь для простоты изложения предполагается, что данные состоят только из пробелов и прописных символов.) Кроме того, допустим, что ключом шифрования является следующая строка.

ELIOT

Ниже описыва7ется используемый алгоритм шифрования.

1. Разбейте открытый текст на блоки, длина которых равна длине ключа шифрования.

   ```
   AS+KI NGFIS HERS+ CATCH +FIRE
   ```

   (Здесь пробелы обозначены знаком "+".)

2. Замените каждый символ открытого текста целым числом в диапазоне 00-26, используя для пробела число 00, для A — число 01,..., для Z — число 26. В результате получится следующая строка цифр.

   ```
   0119001109 1407060919 0805181900 0301200308 0006091805
   ```

3. Повторите п. 2 для ключа шифрования, в результате чего получится следующая строка цифр.
   ```
   0512091520
   ```
4. Теперь значения, помещенные вместо каждого символа в Каждом блоке открытого текста, просуммируйте с соответствующими значениями, подставленными вместо символов ключа шифрования, и для каждой суммы из указанных двух значений определите и запишите остаток от деления на 27.
   ```
   0119001109 1407060919 0805181900 0301200308 0006091805
   0512091520 0512091520 0512091520 0512091520 0512091520
   0604092602 1919152412 1317000720 0813021801 0518180625
   ```
5. Замените каждое число в нижней строке п. 4 соответствующим текстовым символом.
   ```
   FDIZB SSOXL MQ+GT HMBRA ERRFY
   ```

Если ключ шифрования известен, то процедура расшифровки в этом примере может быть выполнена достаточно просто. *(Упражнение.* Расшифруйте зашифрованный текст в рассмотренном выше примере.) Вопрос заключается в том, насколько сложно нелегальному пользователю определить ключ шифрования, обладая открытым и зашифрованным текстами. В данном простом примере это не очень сложно выполнить, однако вполне очевидно, что можно разработать и более сложные схемы шифрования. В идеале, схема шифрования должна быть такой, чтобы усилия, затраченные на ее расшифровку, во много раз превышали полученную при этом выгоду. (Фактически это замечание применимо ко всем аспектам проблемы обеспечения безопасности, т.е. стоимость осуществления попыток взлома системы защиты должна быть значительно выше потенциальной выгоды от этого.) Конечной целью поиска таких схем следует считать схему, для которой сам ее *разработчик,* обладая открытым и зашифрованным вариантами одной и той же части текста, не в состоянии определить ключ и, следовательно, расшифровать другую часть зашифрованного текста.

Стандарт шифрования данных

Приведенный выше пример основан на использовании процедуры **подстановки:** ключ шифрования применялся для того, чтобы определить, какой символ зашифрованного текста следует *подставить* вместо данного символа открытого текста. Подстановка — один из двух основных традиционно используемых методов шифрования, причем в качестве второго выступает процедура **перестановки,** когда символы открытого текста просто переставляются в некоторой другой последовательности. Ни один из этих способов не является безопасным сам по себе, но алгоритмы, построенные на основе их комбинации, способны обеспечить достаточно высокую степень безопасности. Одним из таких алгоритмов является **Data Encryption Standard** (DES), разработанный фирмой IBM и принятый в 1977 году в качестве Федерального стандарта шифрования данных США [17.20].

Согласно этому стандарту, открытый текст делится на блоки по 64 бита и каждый блок шифруется с помощью 64-битового ключа (в действительности этот ключ состоит из 56 битов данных и 8 битов четности, так что существует не 2^{64}, а только 2^{56} возможных ключей). Сначала блок шифруется путем перестановки, затем переставленные данные блока подвергаются обработке посредством подстановки, включающей 16 последовательных сложных шагов, после этого к данным еще раз применяется обратная начальной процедура перестановки, которая и приводит к получению окончательного результата

шифрования. Подстановка на i-м шаге контролируется не самим исходным ключом шифрования к, а ключом Ki, который вычисляется на основе значений К и i. Более подробно этот материал излагается в [17.20].

Стандартом шифрования данных предусматривается, что алгоритм расшифровки идентичен алгоритму шифрования, но ключи Ki применяются в обратном порядке.

Но по мере увеличения быстродействия и производительности компьютеров, стандарт DES все чаще стал подвергаться критике на основании того, что в нем используется сравнительно короткие (56-битовые) ключи. Поэтому 2000 году Федеральным правительством США был принят стандарт **Advanced Encryption Standard** (AES), основанный на так называемом алгоритме Рейндаля (Rijndael) [17.5], в котором используются ключи длиной 128, 192 или 256 битов. Применение даже 128-битовых ключей означает, что новый стандарт является значительно более безопасным по сравнению со старым; согласно [26.34], если "когда-то будет создан компьютер, действующий достаточно быстро для того, чтобы взломать шифр DES за одну секунду, то этому компьютеру придется работать примерно 149 млрд. лет, чтобы взломать 128-битовый ключ AES" (цитата немного перефразирована). Дополнительные сведения по этой теме приведены в [17.5].

Шифрование на основе открытого ключа

Как уже было сказано, стандарт шифрования данных DES не является абсолютно безопасным; стандарт AES является более защищенным, но, тем не менее, многие специалисты полагают, что подобные схемы могут быть взломаны с применением одной лишь "грубой силы", без использования каких-либо сложных методов расшифровки. Многие специалисты считают также, что по сравнению с самыми современными методами, построенными на основе использования **открытого ключа,** все другие методы выглядят технологически устаревшими. В методах с использованием открытого ключа внешнему миру доступны как алгоритм шифрования, так и *ключ шифрования,* поэтому любой желающий может преобразовать открытый текст в зашифрованный. Но соответствующий **ключ расшифровки** хранится в секрете (в методах открытого ключа используются *два* ключа: один — для шифрования, другой — для расшифровки). Более того, ключ расшифровки не может быть просто выведен из ключа шифрования, поэтому лицо, выполнившее шифрование исходного текста, не сможет его расшифровать, не имея доступа к соответствующей информации.

Первоначальная идея использования такого метода принадлежит Диффи (Diffie) и Хеллману (Hellman) [17.9], однако здесь для демонстрации подобных методов будет описан наиболее известный подход, разработанный Райвестом (Rivest), Шамиром (Shamir) и Адлеманом (Adleman) [17.17]. В основу этого подхода, названного *схемой RSA* (по первым буквам фамилий его авторов), положены описанные ниже два факта.

1. Существует быстрый алгоритм определения того, является ли данное число простым.
2. Не существует быстрого алгоритма разложения данного составного числа (т.е. числа, которое не является простым) на простые множители.

В **[17.12]** приведен пример, в котором для определения, является ли число из 130 цифр простым, потребовалось 7 мин вычислений на некотором компьютере, тогда как для поиска (на том же компьютере) двух простых множителей числа, получаемого

умножением двух простых чисел, состоящих из 63 цифр, потребуется около *40 квадриллионов лет*[4] (1 квадриллион = 1 000 000 000 000 000).

Схема RSA предусматривает выполнение приведенной ниже последовательности действий.

1. Выберите два произвольных и разных больших простых числа p и *q,* а затем вычислите их произведение *г = р * д.*
2. Выберите произвольное большое целое число e, которое является взаимно простым (т.е. не имеет общих множителей) по отношению к произведению (p-1) * (д-1). Это целое число e и будет служить ключом шифрования. *Примечание.* Вы брать число e достаточно просто, например, для него подойдет любое простое число, которое больше чисел p и *д.*
3. Выберите в качестве ключа расшифровки число d, которое является *обратным относительно умножения* на e по модулю (p-1) * (д-1), т.е. такое число, для которого выполняется следующее соотношение:

   d * e = 1 modulo (p-1) * (д-1)

   Алгоритм вычисления cd для заданных чисел e, p и *д* достаточно прост и полностью приведен в [17.17].
4. При этом огласке могут быть преданы числа r и e, но не d.
5. Для шифрования части открытого текста *p* (который для простоты рассматривается как целое число меньше r) заменим его зашифрованным текстом c согласно следующей формуле:

   $C = P^e \text{modulo } r$
6. Для расшифровки части зашифрованного текста C замените его открытым текстом *P,* определяемым согласно следующей формуле:

   $P = d \text{ modulo } r$

В [17.17] доказывается работоспособность этой схемы, т.е. возможность расшифровки текста c с использованием числа d для восстановления исходного открытого текста *p.* Однако, как упоминалось ранее, вычисление d для известных чисел r и e (а не p и q) практически неосуществимо. Поэтому любой пользователь может зашифровать открытый текст, но расшифровать зашифрованный текст смогут только санкционированные пользователи (т.е. те, что знают число d).

Для иллюстрации работы этого метода рассмотрим приведенный ниже простой пример, в котором по очевидным причинам используются очень небольшие целые числа.

Пример. Пусть p = 3, *д* = 5; тогда r = 15, а произведение (p-1) * (д-1) = 8. Пусть e = 11 (простое число больше p и д). Вычислим d согласно следующей формуле.

---

[4] Несмотря на это, и в отношении надежности схемы RSA уже возникают некоторые сомнения. Работа [17.12] была опубликована в 1977 году, а в 1990 году Ленстра (Lenstra) и Манассе (Manasse) смогли успешно разложить на простые множители число из 155 цифр [17.24]; они оценили, что выполненные вычисления, в которых участвовала тысяча компьютеров, эквивалентны расчету на одном компьютере со скоростью 1 млн инструкций в секунду в течение 273 лет. Данное число из 155 цифр было девятым числом Ферма: $2^{2^7}+1$ (обратите внимание, что 512 = 2'). См. также работу [17.14], в которой сообщается о совершенно другом (и тоже успешном!) подходе к расшифровке алгоритма RSA.

$$d * 11 = 1 \text{ modulo } 8$$

В результате получим d = 3. Теперь допустим, что открытый текст *p* состоит из целого числа 13; тогда зашифрованный текст Сбудет иметь следующий вид:

$$C = P^e \text{ modulo } r = 13^n \text{ modulo } 15$$
$$= 1\ 792\ 160\ 394\ 037 \text{ modulo } 15$$
$$= 7$$

Теперь исходный открытый текст *P* может быть получен с помощью такой обратной процедуры:

$$P - C^d \text{ modulo } r$$
$$= 7^3 \text{ modulo } 15$$
$$=3\ 43 \text{ modulo}$$
$$15\ .\ = 13$$

Поскольку числа e и *d* взаимно обратны, в методах шифрования на основе открытого ключа предусмотрена также возможность **подписывать** отсылаемые сообщения, что позволит получателю иметь уверенность в том, что данное сообщение было получено именно от того лица, которое объявлено его отправителем (т.е. такая подпись не может быть подделана). Допустим, что пользователи *А И В* общаются друг с другом с помощью метода шифрования на основе открытого ключа. Пусть они предали гласности алгоритм шифрования (в том числе оба они указали соответствующий ключ шифрования), но хранят в секрете даже друг от друга алгоритм и ключ расшифровки. Пусть алгоритмы шифрования сообщений *ECA* и *ECB* используются для шифрования сообщений, посылаемых пользователям *Аи в,* соответственно, а отвечающими им алгоритмами расшифровки являются алгоритмы *DCA* и *DCB.* Следует отметить, что алгоритмы шифрования и расшифровки *ECA* и *DCA,* как и алгоритмы *ECB* и *DCB,* являются взаимно обратными.

Теперь предположим, что пользователь *А* хочет отослать фрагмент открытого текста *p* пользователю *в.* Вместо вычисления результата *ECB (P)* и отправки его пользователю в, пользователь *А* сначала применяет для открытого текста *P* алгоритм *расшифровки DCA,* а затем зашифровывает полученный результат и в зашифрованном виде с передает его пользователю *в,* как показано ниже.

$$C = ECB\ (\ DCA\ (\ P\ )\ )$$

После получения зашифрованного текста с пользователь в применяет сначала алгоритм расшифровки *DCB,* а затем — алгоритм *шифрования ECA,* что позволяет ему в результате получить следующий открытый текст *P.*

```
ECA ( DCB ( C )  )
  = ECA ( DCB (    ECB            ( DCA ( P ) ) ) )
  = ECA ( DCA (    P )            )  /* поскольку DCB и ECB взаимно
  исключаются */
  = P                  /* поскольку  ECA и DCA взаимно исключаются
  */
```

Таким образом, пользователь В знает, что полученное им сообщение действительно пришло от пользователя *А,* поскольку алгоритм шифрования *ECA* приведет к получению результата Р, только если в процессе шифрования применялся алгоритм расшифровки *DCA,* а этот алгоритм известен только пользователю *А.* Поэтому никто не сможет подделать подпись пользователя *А,* даже пользователь В.

17.6. СРЕДСТВА ЯЗЫКА SQL

В действующем стандарте языка SQL предусматривается поддержка только избирательного метода управления доступом. Она строится на основе двух более или менее независимых функций языка SQL. Первая из них является **механизмом представлений,** который может быть использован для сокрытия важных данных от несанкционированных пользователей. Вторая функция называется **подсистемой авторизации** и позволяет одним пользователям, обладающим определенными правами доступа, избирательно и динамически предоставлять эти полномочия другим пользователям, а также отзывать предоставленные ими полномочия в случае необходимости. Ниже эти функции языка SQL обсуждаются более подробно.

Представления и защита данных

Продемонстрируем использование представлений для организации защиты данных с помощью записи на языке SQL примеров 2—4 из раздела 17.2.

2.
```
CREATE VIEW LS AS
       SELECT S.S#, S.SNAME, S.STATUS, S.CITY
       FROM   S
       WHERE  S, CITY = 'London' ;
```
Это представление определяет данные, по отношению к которым будут предоставлены некоторые привилегии. Само же предоставление привилегий осуществляется с помощью операторов GRANT.
```
GRANT SELECT, DELETE, UPDATE ( SNAME, STATUS )
ON    LS
TO    Dan, Misha ;
```
Здесь следует отметить то, что, поскольку привилегии задаются с помощью оператора GRANT, а не с помощью гипотетического оператора CREATE AUTHORITY, привилегиям в языке SQL *имена не присваиваются.* (Хотя для ограничений целостности, наоборот, предусматривается присвоение имен, как показано в главе9.) '

3.
```
CREATE VIEW SSPPO AS
       SELECT S.S#, S.SNAME, S.STATUS,
       S.CITY FROM   S WHERE  EXISTS
            ( SELECT * FROM SP
              WHERE EXISTS
                   ( SELECT * FROM P
                     WHERE  S.S#  =  SP.S#  AND
                     SP.P#    =    P.P#    AND
                     P.CITY = 'Oslo' ) ) ;
```
Собственно предоставление привилегий выполняется с помощью следующего оператора GRANT.
```
GRANT SELECT ON SSPPO TO Lars ;
```

4.
```
CREATE VIEW SSQ AS
       SELECT S.S#,   ( SELECT SUM (SP.QTY)
                        FROM SP
                        WHERE SP.S# = S.S# ) AS
       SQ FROM   S ;
```

Собственно предоставление привилегий выполняется с помощью следующего оператора GRANT.

```
GRANT SELECT ON SSQ TO Fidel ;
```

В примере 5 из раздела 17.2 демонстрируется предоставление *контекстно-зависимых* полномочий. В языке SQL поддерживается несколько нуль-арных встроенных операторов (CURRENT_USER, CURRENT_DATE, CURRENT_TIME И Т.Д.), каждый из которых, поМИМО всего прочего, может быть использован и для определения контекстно-зависимых представлений. (Но в этом языке не поддерживается аналог оператора DAY (), который использовался в первоначальном варианте примера 5.) Поэтому ниже приведен немного упрощенный вариант этого примера.

```
CREATE VIEW S NINE TO FIVE AS
       SELECT S.S#, S.NAME, S.STATUS,
       S.CITY FROM   S
       WHERE  CURRENT TIME > TIME
       '09:00:00' AND    CURRENT TIME <
       TIME '17:00:00'
```

Соответствующий оператор GRANT выглядит следующим образом.

```
GRANT SELECT, UPDATE ( STATUS )
ON    S NINE TO FIVE TO
ACCOUNTING ;
```

Обратите внимание на то, что представление S_NINE_TO_FIVE демонстрирует очень странный тип представления, так как его содержимое меняется со временем, причем даже в том случае, когда исходные данные в базе не изменялись. *{Упражнение.* Определите, каковым является соответствующий предикат.) Более того, представление, в определение которого встроен оператор CURRENT_USER, может (и, скорее всего, обязательно будет) иметь разное содержание для различных пользователей. Такие *представления* действительно отличаются от традиционных представлений, поскольку фактически они являются *параметризованными.* Предпочтительнее было бы, по крайней мере, концептуально, разрешить пользователям определять собственные (потенциально параметризованные) функции со значениями в виде отношения и рассматривать *представления,* подобные S_NINE_TO_FIVE, в качестве особых случаев таких функций.

Как бы то ни было, приведенные выше примеры хорошо иллюстрируют тот факт, что механизм представлений косвенно обеспечивает важные функции защиты данных ("косвенно" потому, что этот механизм включен в систему для достижения совсем других целей). Более того, значительная часть процедуры контроля прав доступа (в том числе зависящая от конкретных значений) может быть выполнена еще на этапе компиляции, а не на этапе прогона, что позволяет достичь существенного выигрыша в производительности. Тем не менее, подходу с использованием представлений для организации защиты также свойственны определенные недостатки. В частности, если некоторому пользователю в одно и то же время потребуются разные права доступа в отношении разных подмножеств данных одной и той же таблицы, то это приведет к сложностям. В качестве примера можно привести сложную реализацию приложения, в котором пользователю разрешается просматривать и распечатывать данные о деталях из Лондона и дополнительно разрешается обновлять данные о некоторых из них (например, только о деталях красного цвета) непосредственно в процессе просмотра.

Операторы GRANT и REVOKE

Механизм представлений языка SQL позволяет концептуально разделять базу данных на части таким образом, чтобы важная информация была скрыта от несанкционированных пользователей. Но этот механизм не позволяет указывать тот набор операций, которые разрешается применять в отношении данных частей *санкционированным* пользователям. Подобная задача (как было показано выше) решается с помощью оператора **GRANT,** который ниже рассматривается более подробно (но некоторые наиболее специализированные средства этого оператора не описаны).

Прежде всего, создателю любого объекта автоматически предоставляются все привилегии в отношении этого объекта. Например, создателю базовой таблицы т автоматически разрешается осуществлять выборку (SELECT), вставку (INSERT), удаление (DELETE), обновление (UPDATE) таблицы т и ссылаться (REFERENCES) на нее, а также определять для нее триггеры (TRIGGER)[5]. Ниже эти привилегии рассматриваются более подробно. Более того, привилегии в каждом случае даются *с правом их передачи,* т.е. обладатель некоторых полномочий в подобном случае имеет право предоставить их другим пользователям.

Ниже приводится общий синтаксис оператора GRANT.

```
GRANT <privilege
      commalist> ON
      <object>
      TO <user ID
      commalist> [ WITH
      GRANT OPTION ] ;
```

Пояснения

1. Допустимыми значениями параметра <*privilege*> могут быть ключевые слова USAGE (использование), UNDER (определение подтаблицы), SELECT (выборка), INSERT (вставка), DELETE (удаление), UPDATE (обновление), REFERENCES (ссылка), TRIGGER (определение триггера) и EXECUTE (выполнение). Каждая из привилегий SELECT, INSERT, UPDATE и REFERENCES может быть определена на уровне столбца.

   Примечание. Может быть также задана привилегия ALL PRIVILEGES, но она имеет не такую простую семантику, как может показаться на первый взгляд (см. [4.20]).

   - Привилегия USAGE на тип, определяемый пользователем, требуется для ис пользования этого типа.
   - Привилегия UNDER может потребоваться, во-первых, на конкретный тип, оп ределяемый пользователем, для получения возможности создания подтипа этого типа, и, во-вторых, на конкретную таблицу, для создания подтаблицы этой таблицы (см., соответственно, главы 20 и 26).
   - Привилегии SELECT, INSERT, DELETE и UPDATE не требуют дополнительных пояснений.
   - Привилегия REFERENCES на конкретную таблицу требуется, если нужно со слаться на эту таблицу в ограничении целостности (под этим подразумевается

[5] Желательно было бы также определить привилегию на создание подтаблиц (UNDER) в тех случаях, где она имеет смысл, но в стандарте SQL она не упоминается.

любое ограничение данного типа, а не только ограничение ссылочной целостности).

- Привилегия TRIGGER на базовую таблицу требуется для создания триггера для данной таблицы.
- Привилегия EXECUTE на конкретную процедуру **SQL** требуется для вызова этой процедуры.

2. К числу допустимых параметров с обозначения объекта <*objec t*> относятся TYPE <*type name*>, TABLE <*table name*> и (применительно к ключевому слову EXECUTE) определенная конструкция, называемая обозначением конкретной процедуры <*specific routine designator*>, описание которой выходит за рамки настоящей книги.

   Примечание. **В** данном контексте, в отличие от многих других случаев применения ключевого слова TABLE в языке **SQL,** область действия этого ключевого слова (которое фактически является необязательным) охватывает не только базовые таблицы, но и представления.

3. Параметр с обозначением разделенного запятыми списка идентификаторов пользователей <*user ID commalist*> может быть заменен специальным ключевым словом PUBLIC, которое обозначает всех пользователей, известных системе.

   Примечание. Язык SQL поддерживает также определяемые пользователем роли; в качестве примера можно указать роль ACCOUNTING, под которой подразумеваются все пользователи бухгалтерского отдела. Любой роли после ее создания могут быть назначены привилегии, по такому же принципу, как если бы она представляла собой обычный идентификатор пользователя. Кроме того, сами роли могут быть предоставлены, как и привилегии, либо идентификаторам пользователей, либо другим ролям. Иными словами, в языке SQL роли выполняют функции механизма поддержки групп пользователей (см. раздел 17.1).

4. Ключевое слово WITH GRANT OPTION, если оно задано, обозначает, что указанная привилегия на указанный объект предоставлена указанному пользователю с правом предоставления другому пользователю; это означает, что, как было описано выше, пользователь, имеющий такое право, может предоставлять привилегии на данный объект другому пользователю (пользователям). Разумеется, пользователь, применяющий оператор GRANT, может задавать в нем ключевое слово WITH GRANT OPTION лишь при том условии, что он сам обладает необходимыми привилегиями.

Если пользователь А наделяет некоторыми полномочиями пользователя в, то впоследствии он может отозвать эти полномочия у пользователя в. Отзыв полномочий выполняется с помощью оператора **REVOKE** с приведенным ниже синтаксисом.

```
REVOKE [ GRANT OPTION FOR ] <privilege commalist>
                            ON <object>
                            FROM <user ID
                            comma.list> <behavior>
                            ;
```

Здесь ключевое слово GRANT OPTION FOR означает, что отменяется только право передачи указанных привилегий, предоставленное ранее этим пользователям. Значения

параметров <*privilege commalist*>, <*object*> и <*user ID commalist*> аналогичны значениям параметров оператора GRANT. Значениями параметра <*behavior*> могут быть ключевые слова RESTRICT (ОТКЛОНИТЬ **в** случае нарушения) и CASCADE (выполнить каскадно) (как обычно). Ниже приведены некоторые примеры использования этого оператора.

```
1. REVOKE SELECT ON S FROM Jacques, Anne, Charley RESTRICT ;
2. REVOKE SELECT, DELETE, UPDATE ( SNAME, STATUS )
          ON LS FROM Dan, Misha CASCADE ;
3. REVOKE SELECT ON SSPPO FROM Lars RESTRICT ;
4. REVOKE SELECT ON SSQ FROM Fidel RESTRICT ;
```

Рассмотрим назначение ключевых слов RESTRICT и CASCADE. Допустим, что р является некоторой привилегией для некоторого объекта и пользователь А предоставляет привилегию р пользователю в, который в свою очередь предоставляет ее пользователю С. Что произойдет, если теперь пользователь А отменит привилегию р для пользователя в? На данный момент предположим, что оператор REVOKE выполняется успешно. Если эта отмена состоится, то привилегия р у пользователя С останется *зависшей* (orphaned), поскольку она была производной от привилегии р пользователя в, который уже ею **не** обладает. Основное назначение ключевых слов RESTRICT И CASCADE СОСТОИТ В предотвращении ситуаций возникновения зависших привилегий. При задании ключевого слова RESTRICT запрещается выполнять операцию отмены привилегии, если она приводит к появлению зависшей привилегии. Ключевое слово CASCADE указывает на необходимость каскадной отмены всех привилегий, производных от отменяемой.

Наконец, при удалении типа таблицы, столбца или процедуры автоматически отменяются также все привилегии на удаленный объект, предоставленные в отношении этого объекта со стороны **всех** пользователей.

17.7. РЕЗЮМЕ

В этой главе рассматривались различные аспекты проблемы **защиты** базы данных. Во введении были показаны различия между понятиями *защиты* и *целостности.* Итак, под защитой подразумевается контроль за тем, разрешено ли санкционированным пользователям выполнять предпринимаемые ими действия, тогда как под целостностью понимается проверка допустимости этих действий. Иначе говоря, под защитой понимается *защита данных от несанкционированного доступа.*

Защита обеспечивается **подсистемой защиты** СУБД, проверяющей соответствие всех поступающих запросов существующим **ограничениям защиты** (или чаще всего полномочиям), которые хранятся в системном каталоге. Сначала были рассмотрены **избирательные** схемы защиты, в которых доступ к конкретному объекту определялся владельцем объекта по своему усмотрению. Для каждого ограничения безопасности в избирательной схеме задаются **имя,** множество **привилегий** (RETRIEVE, INSERT и т.д.), соответствующая **переменная отношения** (т.е. данные, к которым применимы указанные ограничения) и множество **пользователей.** Такие правила могут применяться для организации управления как **зависящего,** так и **не зависящего от конкретных значений,** а также для **статистически обобщенного** и **контекстно-зависимого** управления. Для регистрации попыток нарушения защиты может использоваться **контрольный журнал. В** данной главе **было** представлено

краткое описание метода реализации избирательной схемы защиты на основе механизма **модификации запроса.** Впервые эта технология была применена в прототипе системы Ingres с использованием средств языка QUEL.

Далее кратко рассматривались методы **мандатного** управления, согласно которым каждый объект должен обладать некоторым **классификационным уровнем,** а каждому пользователю должен быть присвоен определенный уровень **допуска.** Помимо описания правил доступа для такой схемы, кратко рассматривалась классификация мер безопасности, регламентированная Министерством обороны США в "Оранжевой" и "Сиреневой" книгах, а также кратко приводились идеи создания **многоуровневых переменных отношения** и методы **поликонкретизации.**

Затем обсуждались особенности работы со **статистическими базами данных.** *Статистической* называется база данных, которая содержит большое количество отдельных конфиденциальных сведений и пользователям которой может предоставляться только некоторая статистически обобщенная информация. Было показано, что защита такой базы данных легко может быть нарушена с помощью специальных выражений — **средств слежения.** (Этот факт должен вызывать определенную тревогу в связи с возрастающим интересом к хранилищам данных, подробное описание которых приводится в главе 22.)

Также были описаны методы **шифрования данных** с использованием методов **подстановки** и **перестановки,** приведены разъяснения в отношении стандартов шифрования данных **Data Encryption Standard (DES)** и **Advanced Encryption Standard** (AES), а также кратко рассмотрены методы шифрования с помощью **открытых ключей. В** частности, был приведен простой пример использования схемы **RSA** на основе ключей, представляющих собой простые числа. Кроме того, была описана концепция применения **цифровых подписей.**

После этого были кратко описаны средства защиты языка SQL. В частности, обсуждалось использование **представлений** для сокрытия информации и применение операторов **GRANT** и **REVOKE** для управления наборами привилегий, предоставленных конкретным пользователям в отношении различных объектов базы данных (в основном это базовые таблицы и представления).

В заключение, вероятно, стоит отметить, что от СУБД, предоставляющей большое количество функций защиты, каждую из которых можно легко преодолеть, будет мало пользы. Например, в СУБД DB2 данные базы физически хранятся в файлах операционной системы. Следовательно, любые механизмы защиты в системе DB2 были бы совершенно бесполезны, если бы было возможно получить доступ к этим файлам с помощью обычной программы, использующей обычные средства операционной системы. Поэтому СУБД DB2 согласованно взаимодействует с другими параллельно существующими системами (например, с базовой операционной системой), что дает ей гарантии общей защищенности системы. Подробное изложение этого аспекта защиты выходит за рамки данной главы, однако указанную особенность все же стоило упомянуть.

УПРАЖНЕНИЯ

17.1. Пусть базовая переменная отношения STATS выглядит так, как показано в разделе 17.4.

```
STATS { NAME, SEX, CHILDREN, OCCUPATION, SALARY, TAX,
      AUDITS } KEY { NAME }
```

Используя гипотетический язык, представленный в разделе 17.2, запишите определения ограничений защиты для предоставления перечисленных ниже привилегий:

а) пользователь Ford обладает правом выборки (RETRIEVE) из всей переменной отношения;

б) пользователь Smith обладает правом вставки (INSERT) и удаления (DELETE) данных для всей переменной отношения;

в) каждый пользователь обладает правом выборки (RETRIEVE) кортежа (толькр) со своими личными данными;

г) пользователь Nash обладает правом выборки (RETRIEVE) из всей переменной отношения и правом обновления (UPDATE) атрибутов SALARY и TAX (и только этих атрибутов);

д) пользователь Todd обладает правом выборки (RETRIEVE) атрибутов USERID, SALARY и TAX (и только этих атрибутов);

е) пользователь Ward обладает такими же правами выборки (RETRIEVE), как и пользователь Todd, а также правом обновления (UPDATE) атрибутов SALARY и TAX (и только этих атрибутов);

ж) пользователь Pope (Священник) обладает всеми правами (выборки RETRIEVE, вставки INSERT, удаления DELETE и обновления UPDATE), но только в отношении кортежей с данными о проповедниках;

з) пользователь Jones обладает правом удаления (DELETE) кортежей с данными о лицах, занимающих *неспециализированные должности,* т.е. такие должности, которые занимают более 10 работников;

и) пользователь King обладает правом выборки (RETRIEVE) данных о минимальной и максимальной зарплатах для каждого из существующих типов рабочих мест.

17.2. Как следует расширить синтаксис определений полномочий AUTHORITY для включения в него средств управления такими операциями, как создание и удаление базовых переменных отношения, создание и удаление представлений, создание и удаление полномочий и т.д.?

17.3. Еще раз обратимся к рис. 17.2 и предположим, что из посторонних источников получены сведения о том, будто лицо по имени Hal является домохозяйкой, которая имеет не меньше двух детей. Запишите последовательность статистических запросов, позволяющих определить размер выплачиваемых ею налогов, используя подходящее индивидуальное средство слежения. Предположим (как и в разделе 17.4), что в системе не разрешены запросы, имеющие результирующие множества с кардинальностью меньше 2 или больше 8.

17.4. Повторите упр. 17.3, но с использованием общего средства слежения вместо индивидуального.

17.5. Расшифруйте приведенный ниже зашифрованный текст, созданный по тому же принципу, который был использован в примере из настоящей главы для строки AS KINGFISHERS CATCH FIRE, НО С ПОМОЩЬЮ ДРУГОГО ПЯТИСИМВОЛЬНОГО КЛЮЧА шифрования.

```
F N W A
L J P V
J C F P
E X E
```

A B W N
E A У E
I P S U
S V D

17.6. Испытайте схему шифрования RSA на основе открытого ключа с p = 7, g = 5 и e = 17 для открытого текста *p* = 3.

17.7. Какие, по вашему мнению, проблемы реализации или другие сложности могут возникнуть при использовании шифрования?

17.8. Приведите решения упр. 17.1 на языке SQL.

17.9. Составьте на языке SQL предложения для удаления привилегий, установленных в предыдущем упражнении.

СПИСОК ЛИТЕРАТУРЫ

17.1. Agrawal R., Kiernan J., Srikant R., Xu Y. Hippocratic Databases // Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong. — August 2002.

Приведем цитату из резюме: "[Мы] утверждаем, что одним из основных требований к будущим системам баз данных должно быть наличие средств, позволяющих возложить на них ответственность за конфиденциальность данных, которыми они управляют... В настоящей работе сформулированы ... основные принципы обеспечения конфиденциальности ... для систем баз данных".

17.2. Agrawal R., Kiernan J. Watermarking Relational Databases // Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong. — August 2002.

Предложена схема ввода в данные своего рода *водяных знаков,* позволяющая обнаруживать пиратские копии (по сути, выявлять случаи нарушения авторских прав).

17.3. Bell D.E., La Padula L.J. Secure Computer Systems: Mathematical Foundations and Model // MITRE Technical Report M74-244. - May 1974.

17.4. Bouganim L., Pucheral P. Chip-Secured Data Access: Confidential Data on Untrusted Servers // Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong. — August 2002.

Приведем цитату из резюме: "[Данная схема] обеспечивает конфиденциальность данных и управление личными привилегиями с помощью компонента защиты, основанного на клиентском приложении, которое действует в качестве посредника между клиентом и зашифрованной базой данных. Код этого компонента встроен в смарт-карту для предотвращения подделки". Эта схема позволяет решить классическую проблему поддержки зашифрованных баз данных, которая состоит в том, что данные в индексах должны быть в формате открытого текста.

17.5. Daemen J., Rijnen V. The Block Cipher Rijndael // J.-J. Quisquater and B. Schneier (eds.). Smart Card Research and Applications (Springer-Verlag Lecture Notes in Computer Science 1820). New York, NY: Springer-Verlag, 2000.

17.6. Daly J. Fingerprinting a Computer Security Code // Computerworld. — July 27, 1992.

17.7. Denning D.E. Cryptography and Data Security. — Reading, Mass.: Addison-Wesley, 1983.

17.8. Denning D.E., Denning P.J. Data Security // ACM Сотр. Surv. — September 1979. - 11, №3.

Прекрасное учебное пособие, посвященное средствам защиты, с описанием избирательных и мандатных методов управления доступом (здесь они называются *потоковыми методами управления),* методов шифрования данных и *контроля возможности логического вывода* (эта проблема характерна для статистических баз данных).

17.9. Dine W., Hellman M.E. New Directions in Cryptography // IEEE Transactions on Information Theory. — November 1976. — IT-22.

17.10. Fagin R. On an Authorization Mechanism //ACM TODS. — September 1978. — 3, № 3.

Подробное изложение поправок к [17.13] о некоторых условиях, способных привести к отмене привилегии, которую нельзя отменить с помощью механизма, описанного в [17.13]. В настоящей статье исправлена указанная ошибка.

17.11. Gagliardi R., Lapis C, Lindsay B. A Flexible and Efficient Database Authorization Facility//IBM Research Report RJ6826. — May 11, 1989.

17.12. Gardner M. A New Kind of Cipher That Would Take Millions of Years 'to Break // Scientific American. — August 1977. — 237, № 2.

Прекрасное неформальное введение в теорию шифрования с использованием открытых ключей, хотя заявление, приведенное в названии (о том, что для раскрытия рассматриваемого шифра потребуются миллионы лет), несколько преувеличено [17.14], [17.24].

17.13. Griffiths P.P., Wade B.W. An Authorization Mechanism for a Relational Data Base System //ACM TODS. - September 1976. - 1, № 3.

Описание механизмов предоставления (GRANT) и отмены (REVOKE) привилегий, впервые предложенных для системы System R. Впоследствии эта схема была включена в стандарт языка SQL, но немного в иной (и гораздо более усложненной) форме.

Hawkes N. Breaking into the Internet // London Times. — March 18, 1996.

Описание *взлома* опытным специалистом в области компьютерных технологий схемы RSA посредством измерения продолжительности шифрования сообщений системой: "электронный эквивалент определения комбинации цифр замка путем наблюдения за тем, как при открытии сейфа поворачиваются циферблаты, и измерения продолжительности каждого движения".

17.15. Jajodia S., Sandhu R. Toward a Multilevel Secure Relational Data Model // Proc. 1991 ACM SIGMOD Int. Conf. on Management of Data. - Denver, Col. - June 1991.

Как объяснялось в разделе 17.3, *многоуровневая структура* в контексте средств защиты относится к системе, в которой поддерживаются методы мандатного управления доступом. В статье утверждается, что вся текущая деятельность в этой области в основном выполняется спонтанно, поскольку не существует единого мнения в отношении главных концепций. Кроме того, в этой работе предложен подход к разработке формальных принципов многоуровневых систем.

17.16. Lempel A. Cryptology in Transition //ACM Сотр. Surv. — December 1979. — 11, № 4 (Special Issue on Cryptology).

Прекрасное учебное пособие по шифрованию и связанным с ним вопросам.

17.17. Rivest R.L., Shamir A., Adleman L. A Method for Obtaining Digital Signatures and Public Key Cryptosystems // CACM. - February 1978. - 21, № 3.

17.18. Smith K., Winslett M. Entity Modeling in the MLS Relational Model // Proc. 18th Int. Conf. on Very Large Data Bases. — Vancouver, Canada. — August 1992.

Аббревиатура MLS в заголовке статьи означает *многоуровневую защиту* (Multilevel Secure — MLS) [17.15]. В статье внимание акцентируется на значении баз данных с такой защитой и предложена конструкция нового типа (BELIEVED BY) для определения операций выборки и обновления по отношению к некоторому особому состоянию базы данных, которое воспринимается некоторым пользователем или *которому пользователь доверяет.* В статье утверждается, что этот подход поможет разрешить несколько проблем, существовавших при использовании прежних подходов. См. также [17.23].

17.19. Thuraisingham B. Current Status of R&D in Trusted Database Management Systems// ACM SIGMOD Record. - September 1992. - 21, № 3.

Краткий обзор и обширный набор ссылок на работы о *заслуживающих доверия* или многоуровневых системах (по состоянию на начало 1990-х годов).

17.20. U.S. Department of Commerce/National Bureau of Standards. Data Encryption Standard. — Federal Information Processing Standards Publication 46. — January 15, 1977.

В работе дано определение официально принятого стандарта шифрования данных (Data Encryption Standard — DES). Этот алгоритм шифрования/расшифровки (см. раздел 17.5) может быть реализован и на аппаратном уровне, например в микропроцессоре. Устройства с таким микропроцессором демонстрируют весьма высокую скорость обработки данных. В настоящее время существует целый ряд подобных коммерческих продуктов.

17.21. U.S. Department of Defense. Trusted Computer System Evaluation Criteria ("Оранжевая" книга). — Document № DoD 5200-28-STD, DoD National Computer Security Center. — December 1985.

17.22. U.S. National Computer Security Center. Trusted Database Management System Interpretation ("Сиреневая" книга).— Document № NCSC-TG-201, Version 1.— April 1991.

17.23. Winslett M., Smith K., Qian X. Formal Query Languages for Secure Relational Databases // ACM TODS. - December 1994. - 19, № 4.

Продолжение работы [17.18].

17.24. Wolf R. How Safe Is Computer Data? A Lot of Factors Govern the Answer // San Jose Mercury News. — July 5, 1990.

ГЛАВА 18

Оптимизация

18.1. ВВЕДЕНИЕ

Для реляционных систем оптимизация представляет собой как проблему, так и благоприятную возможность. Проблема состоит в том, что для достижения приемлемого уровня производительности оптимизация в подобных системах просто *необходима.* Причем одной из сильных сторон и несомненных достоинств реляционного подхода является то, что реляционные выражения реализуются и оптимизируются на достаточно высоком семантическом уровне. В противоположность этому, в нереляционных системах, где запросы пользователей выражаются на более низком семантическом уровне, любая "оптимизация" должна выполняться самим пользователем вручную (здесь термин "оптимизация" взят в кавычки, поскольку обычно он употребляется для обозначения *автоматической,* а не ручной оптимизации). В подобных системах пользователь (а не система) определяет, какие именно низкоуровневые операции должны быть выполнены и в какой последовательности. И если пользователь принял неправильное решение, то система не способна исправить положение. Отметим также, что для работы в подобных системах пользователь должен обладать некоторыми навыками в программировании, иначе он не сможет достаточно полно применять средства этих систем.

Преимущество автоматической оптимизации заключается в том, что пользователь может не задумываться над наилучшим способом выражения своих запросов (т.е. над тем, как следует сформулировать запрос, чтобы система выполнила его с максимальной производительностью). Но и это далеко не все. Вполне вероятно, что оптимизатор сформулирует запрос *лучше,* чем сам пользователь. Для подобного утверждения есть несколько веских причин. Ниже приведены лишь некоторые из них.

1. Хороший оптимизатор обладает большим объемом полезной информации, которая для пользователя обычно недоступна. Говоря конкретнее, оптимизатор владеет определенными **статистическими** данными (в частности о кардинальности переменных отношения и т.д.), в том числе следующими:
   - количество различных значений каждого типа;
   - текущее количество кортежей в каждой базовой переменной отношения;
   - текущее количество различающихся значений для каждого атрибута в каждой базовой переменной отношения;
   - количество вхождений каждого значения в каждом из атрибутов и т.п.

   (Вся эта информация хранится в системном каталоге, о чем речь пойдет ниже в этой главе, в разделе 18.5.) Благодаря наличию таких данных оптимизатор способен более точно оценить эффективность любой возможной стратегии реализации конкретного запроса. Исходя из полученных результатов он сможет выбрать наилучшую стратегию реализации запроса.
2. Если со временем статистические показатели базы данных значительно изменятся (например, в результате ее физической реорганизации), то при реализации запроса может оказаться целесообразнее использовать иную стратегию, отличную от применявшейся ранее. Другими словами, возникнет необходимость в повторной оптимизации, или **реоптимизации.** В реляционных системах процесс реоптимизации вполне тривиален и сводится к простой повторной обработке исходного запроса системным оптимизатором. В нереляционных же системах выполнение реоптимизации, как правило, требует корректировки программы и, вполне возможно, может оказаться вообще невыполнимым.
3. Оптимизатор — это *программа,* которая по определению "всегда доводит начатую работу до конца", в отличие от человека. Оптимизатор способен рассматривать буквально сотни различных стратегий реализации конкретного запроса, в то время как программист едва ли проанализирует более трех-четырех возможных стратегий (по крайней мере, достаточно глубоко).
4. В оптимизаторе реализованы знания и опыт *лучших из лучших* программистов, в результате чего эти знания и опыт становятся доступными *для всех.* Это позволяет применять ограниченный набор ресурсов, предоставленный широкому кругу пользователей, наиболее эффективно и экономично.

Приведенные выше соображения служат убедительным доказательством сделанного в начале данного раздела заявления о том, что оптимизируемость (т.е. возможность оптимизации запросов) является *сильной стороной* реляционных систем.

Общее назначение оптимизатора состоит в выборе наиболее эффективной стратегии вычисления реляционного выражения. В этой главе описаны некоторые фундаментальные

принципы и методы, применяемые в процессе оптимизации. После обсуждения вступительного примера, приведенного в разделе 18.2, в разделе 18.3 предложен обзор принципов работы оптимизатора. Затем в разделе 18.4 обсуждается один из важнейших аспектов процедуры оптимизации — *преобразование выражений* (или *перезапись запроса).* Далее в разделе 18.5 кратко излагается вопрос о *статистических показателях базы данных.* В разделе 18.6 дается более подробное описание одного из конкретных методов оптимизации, известного как *декомпозиция запросов.* Затем в разделе 18.7 обсуждается вопрос о том, как в действительности реализуются некоторые реляционные операторы (например оператор соединения и др.), и кратко описывается использование рассматривавшихся в разделе 18.5 статистических показателей для вычисления стоимостных оценок. Наконец, в разделе 18.8 приводится краткое резюме по всему материалу данной главы.

Еще одно вводное замечание. Достаточно часто данную тему связывают с оптимизацией *запросов.* Однако подобный термин несколько неточен, поскольку выражение, которое нужно оптимизировать *(запрос),* на самом деле может формироваться в контексте, отличном от интерактивной выборки информации из базы данных. В частности, оно может быть частью операции обновления, а не операции выборки, понимаемой под запросом. Более того, сам по себе термин *оптимизация* является несколько преувеличенным, так как обычно не существует гарантий, что выбранная стратегия реализации действительно *оптимальна* в некотором объективном смысле. На практике под *оптимизированной* стратегией реализации обычно понимается просто *улучшенный* вариант исходного неоптимизированного выражения. (Тем не менее, в некоторых немногочисленных случаях можно вполне обоснованно утверждать, что выбранная стратегия реализации будет действительно оптимальной в определенном смысле [18.30]. См. также приложение А.)

18.2. ПРИМЕР ВЫПОЛНЕНИЯ ОПТИМИЗАЦИИ

Начнем изложение с простого примера (он уже кратко рассматривался в разделе 7.6 главы 7), дающего представление о поразительных результатах, которых можно достичь с помощью *оптимизации.* Рассмотрим следующий запрос: "Определить имена поставщиков детали с номером P2". Алгебраическая запись этого запроса такова:

```
( ( SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

Предположим, что в базе данных содержится информация о 100 поставщиках и 10 000 поставок деталей, из которых только 50 включают партии деталей с номером P2. Предположим для простоты, что переменные отношения s и SP сохраняются на диске как два отдельно хранимых файла, в каждой записи которых помещается по одному кортежу данных. В этом случае, если система будет вычислять данное выражение *непосредственно* (т.е. вообще без оптимизации), последовательность выполняемых операций становится такой, как описано ниже.

1. **Соединение переменных отношения SP и S (по атрибуту S#).** При выполнении этой операции потребуется считать информацию о 10 000 поставок партий деталей и 10 000 раз считать информацию о 100 поставщиках (по одному разу для каждой поставки деталей из 10 000). В результате будет получен промежуточный набор данных, содержащий 10 000 соединенных кортежей. Этот набор данных записывается на диск (предположим, что для размещения промежуточного результата в основной (оперативной) памяти не хватает места).

2. **Выборка из полученного на этапе 1 результата кортежей с данными о детали с номе ром P2.** На этом этапе выполняется чтение 10 000 соединенных кортежей обратно в оперативную память, причем полученный результат состоит только из 50 корте жей, которые, по нашему предположению, вполне могут поместиться в оператив ной памяти.
3. **Выполнение проекции по атрибуту SNAME результата, полученного на этапе 2. На** этом этапе формируется результирующий набор исходного запроса (состоящий максимум из 50 кортежей, которые вполне могут быть размещены в оперативной памяти).

Представленная ниже процедура полностью эквивалентна описанной выше в том смысле, что она обязательно приведет к тому же конечному результату, но **он** будет получен более эффективным способом.

1. **Выборка из переменной отношения sp кортежей с данными только о детали с номе ром P2.** На этом этапе выполняется чтение 10 000 кортежей и создается результи рующий набор, состоящий только из 50 кортежей, который, как мы предполагаем, может поместиться в оперативной памяти.
2. **Соединение полученного на этапе 1 результата с переменной отношения S (по атрибу ту s#).** На этом этапе выполняется считывание данных обо всех 100 поставщиках (но только один раз, а не по одному разу для каждой поставки партии деталей P2, так как данные обо всех поставленных партиях деталей с номером P2 уже находят ся в оперативной памяти). Результат содержит 50 соединенных кортежей (которые также помещаются в оперативной памяти).
3. **Выполнение проекции по атрибуту SNAME результата, полученного на этапе 2** (анало гично этапу 3 предыдущей последовательности действий). Требуемый результат (не более 50 кортежей) помещается в оперативной памяти.

В первой из показанных процедур в целом выполняется 1 030 000 операций ввода-вывода кортежей, в то время как во второй процедуре выполняется только 10 100 операций ввода-вывода. Итак, совершенно очевидно, что если принять в качестве меры оценки производительности количество выполненных операций ввода—вывода кортежей, то вторая процедура больше чем в 100 раз эффективнее первой. Кроме того, вполне понятно, что предпочтительнее реализовать данный запрос именно с помощью второй процедуры, а не первой!

Примечание. На практике мерой оценки производительности служит количество операций ввода—вывода *страниц,* а не отдельных кортежей, но для упрощения предположим, что каждый кортеж занимает отдельную страницу.

Приведенный пример показывает, что следствием даже незначительных изменений в алгоритме реализации (выполнения выборки, а затем соединения, вместо соединения и последующей выборки) может быть существенное увеличение производительности (в сотни раз). Производительность повысится еще больше, если переменная отношения SP будет индексирована или хэширована по атрибуту P#. В этом случае количество кортежей, считываемых на этапе 1 второй процедуры, уменьшится с 10 000 всего лишь до 50, в результате чего вся процедура окажется в 7 000 раз эффективнее ее исходного варианта. Аналогично этому, применение индекса или хэш-функции для доступа к атрибуту S. S# позволит уменьшить количество операций ввода—вывода кортежей на этапе 2 со 100 до 50,

в результате чего процедура вычисления запроса окажется более чем в 10 000 раз эффективнее исходного варианта. Это означает, что если на вычисление исходного варианта реализации запроса потребуется *три часа,* то оптимизированная версия этого же запроса будет выполнена примерно за *одну секунду.* К тому же, безусловно, возможны и многие другие улучшения.

Несмотря на то, что приведенный выше пример достаточно прост, он весьма наглядно демонстрирует необходимость использования оптимизации. Кроме того, он показывает вероятные улучшения, которые могут применяться на практике. В следующем разделе используется более систематический подход к решению проблемы оптимизации. В частности, в нем описано, как общая проблема может быть разделена на последовательность из нескольких более или менее независимых подзадач. Это позволит нам перейти к рассмотрению отдельных стратегий и приемов оптимизации, обсуждаемых в следующих разделах.

18.3. ОПТИМИЗАЦИЯ ЗАПРОСОВ

Рассмотрим четыре стадии процесса оптимизации запросов, который схематически представлен на рис. 18.1.

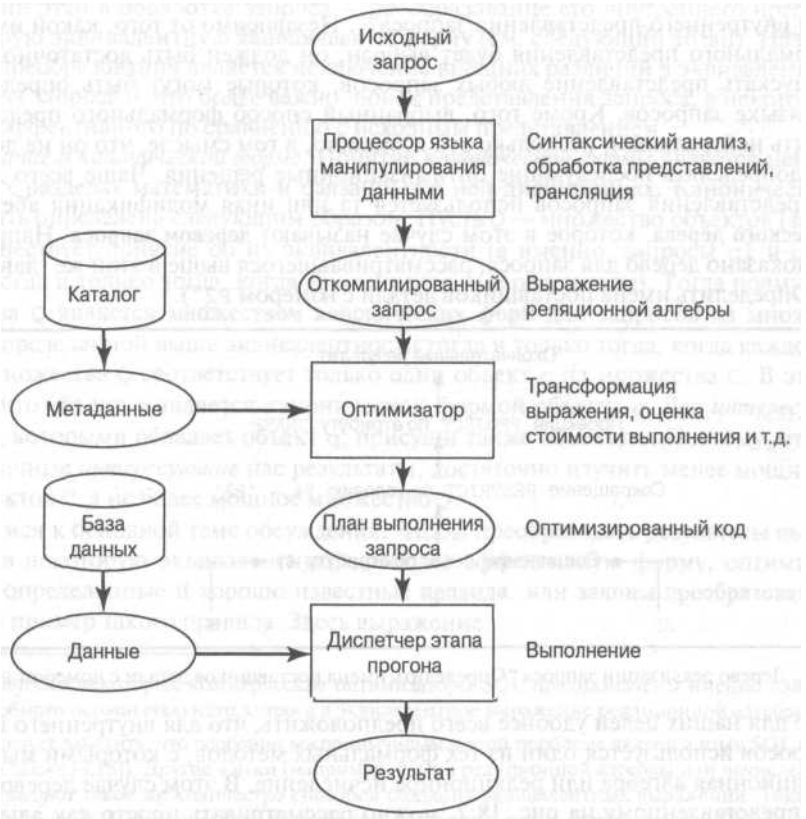


Рис. 18.1. Общая схема процесса оптимизации запроса

1. Преобразование запроса во внутреннюю форму.
2. Преобразование запроса в каноническую форму.
3. Выбор потенциальных низкоуровневых процедур.
4. Генерация различных вариантов планов вычисления запроса и выбор плана с ми нимальными затратами.

Перейдем к подробному рассмотрению каждой стадии процесса оптимизации.

Стадия 1. Преобразование запроса во внутреннюю форму

На этой стадии выполняется преобразование первоначального запроса в некоторое внутреннее представление, более удобное для машинной обработки. В результате из рассмотрения полностью исключаются конструкции сугубо внешнего уровня (например, разнообразные варианты конкретного синтаксиса используемого языка запросов) и готовится почва для последующих стадий оптимизации.

Примечание. Обработка представлений (т.е. процесс замены ссылок на представления выражениями, определяющими соответствующие представления) также выполняется на этом этапе.

Возникает очевидный вопрос: "Какое формальное представление должно использоваться для внутреннего представления запроса?". Независимо от того, какой именно вариант формального представления будет выбран, он должен быть достаточно полным, чтобы допускать представление любых запросов, которые могут быть определены на внешнем языке запросов. Кроме того, выбранный способ формального представления должен быть нейтральным, насколько это возможно, в том смысле, что он не должен заранее предопределять последующие оптимизационные решения. Чаще всего для внутреннего представления запросов используется та или иная модификация **абстрактного синтаксического дерева,** которое в этом случае называют **деревом запроса.** Например, на рис. 18.2 показано дерево для запроса, рассматривавшегося выше в этой же главе в разделе 18.2 ("Определить имена поставщиков детали с номером P2").

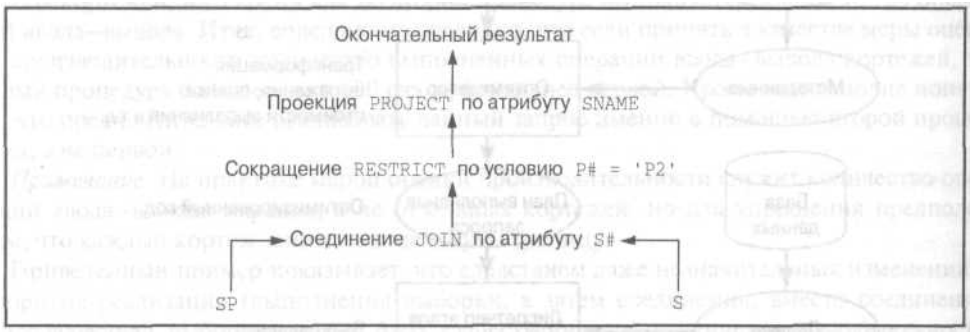


Рис. 18.2. Дерево реализации запроса "Определить имена поставщиков детали с номером P2"

Однако для наших целей удобнее всего предположить, что для внутреннего представления запросов используется один из тех формальных методов, с которыми мы уже знакомы: реляционная алгебра или реляционное исчисление. В этом случае дерево запроса, подобное представленному на рис. 18.2, можно рассматривать просто как альтернативный схематический вариант представления некоторого выражения, записанного в системе

обозначений одного из двух предложенных выше формальных методов (реляционная алгебра или реляционное исчисление). Предположим, что выбранный формальный метод — реляционная алгебра. С этого момента будем считать, что внутренним представлением дерева запроса, показанного на рис. 18.2, является следующее алгебраическое выражение[1]:

```
( { SP JOIN S ) WHERE P# = P# ('P2') ) { SNAME }
```

Стадия 2. Преобразование запроса в каноническую форму

На этой стадии оптимизатор выполняет несколько операций оптимизации, которые *гарантированно являются приемлемыми* независимо от реальных значений данных и существующих путей доступа к ним. Суть в том, что обычно реляционные языки позволяют сформулировать любые запросы (за исключением разве что простейших) несколькими разными (по крайней мере, внешне) способами. Например, даже простой запрос "Определить имена поставщиков детали с номером P2" на языке SQL может быть записан буквально десятками способов[2], не считая таких тривиальных вариантов, как замена A = в на в = A или p AND q на q AND p. Производительность вычисления запроса не должна зависеть от формы записи запроса, которую выбрал пользователь. Поэтому следующий этап в обработке запроса — преобразование его внутреннего представления в некоторую эквивалентную каноническую **форму** (см. следующий абзац). Назначением данного преобразования является исключение внешних различий в эквивалентных представлениях запроса и, что более важно, поиск представления запроса, в некотором смысле более эффективного по сравнению с исходным представлением.

Замечание о канонической форме. Понятие *канонической формы* является центральным во многих разделах математики и связанных с ней дисциплинах. Каноническая форма может быть определена следующим образом. Пусть Q — множество объектов (запросов) и пусть существует понятие об их эквивалентности (а именно, запросы ql и q2 эквивалентны тогда и только тогда, когда дают идентичные результаты). Тогда подмножество C множества Q является **множеством канонических форм** для запросов из множества Q в смысле определенной выше эквивалентности тогда и только тогда, когда каждому объекту q из множества Q соответствует только один объект с из множества с. В этом случае говорят, что объект с является канонической формой объекта q. Все *интересующие* нас свойства, которыми обладает объект q, присущи также объекту с. Поэтому, чтобы доказать различные *интересующие* нас результаты, достаточно изучить менее мощное множество объектов С, а не более мощное множество Q.

Вернемся к основной теме обсуждения. Чтобы преобразовать результаты выполнения стадии 1 в некоторую эквивалентную, но более эффективную форму, оптимизатор использует определенные и хорошо известные **правила,** или **законы преобразования.** Ниже приведен пример такого правила. Здесь выражение

---

[1] Фактически некоторые коммерческие оптимизаторы SQL предназначены именно для преобразования подобного первоначального запроса в эквивалентное выражение реляционной алгебры.

[2] Но следует заметить, что особенно восприимчивым к этой проблеме является язык SQL (см. упр. 8.12 в главе 8, а также [4.18]). Другие языки (например, языки реляционной алгебры или исчисления) обычно не предоставляют такое же количество способов создания эквивалентных выражений. Такая излишняя "гибкость" языка SQL на самом деле значительно усложняет жизнь *разработчикам СУБД* (не говоря уже о пользователях), поскольку осуществление функций оптимизатора становится намного сложнее.

(A JOIN B) WHERE *ограничение, распространяющееся на* A

может быть преобразовано в эквивалентное, но более эффективное выражение

(A WHERE *ограничение, распространяющееся на* A) JOIN B

Подобное преобразование кратко рассматривалось в разделе 7.6 главы 7. Кроме того, оно было описано выше, при обсуждении примера в разделе 18.2, который ясно продемонстрировал, зачем нужны подобные преобразования. Многие другие правила преобразования описываются ниже, в разделе 18.4.

Стадия 3. Выбор потенциальных низкоуровневых процедур

После преобразования внутренней формы запроса в более подходящую (каноническую) форму, оптимизатор должен определить, как следует выполнять запрос, представленный в этой канонической форме. На данной стадии принимаются во внимание такие факторы, как наличие индексов и других физических путей доступа, статистическое распределение существующих значений данных, физическая кластеризация хранимых данных и т.п. Обратите внимание на то, что на стадиях 1 и 2 этим аспектам совсем не уделялось внимания.

Основная стратегия состоит в том, чтобы рассматривать выражение запроса как серию низкоуровневых операций[3], которые в определенной степени зависят одна от другой. Ниже приведен пример такой взаимной зависимости. При программной реализации операции проекции обычно требуется, чтобы считываемые кортежи следовали в определенном порядке; это позволяет исключить дублирующиеся результирующие кортежи. В свою очередь, это означает, что операция, выполняемая непосредственно перед операцией проекции, должна выдавать выходные кортежи именно в такой, требуемой последовательности.

Для каждой низкоуровневой операции (а также, возможно, для нескольких часто встречающихся комбинаций подобных операций) оптимизатор имеет набор низкоуровневых процедур реализации. Например, существует набор процедур для реализации операции сокращения: по условию равенства определенному значению потенциального ключа; на основе индексированного атрибута, по которому выполняется сокращение; на основе хэшированного атрибута и т.д. Примеры таких процедур приведены ниже, в разделе 18.7 (см. также [18.7], [18.12]).

С каждой процедурой связана параметризованная формула стоимости, позволяющая оценить *стоимость* выполнения процедуры (т.е. уровень затрат, требуемых для ее выполнения). Чаще всего стоимость определяется путем подсчета количества необходимых дисковых операций ввода-вывода, но некоторые системы учитывают также время использования процессора и другие факторы. Эти формулы стоимости применяются на стадии 4 (см. следующий подраздел). В [18.7]—[18.12] обсуждаются и анализируются формулы стоимости для различных процедур реализации при разных исходных предположениях (см. также раздел 18.7).

Далее, используя сохраняемую в каталоге информацию о состоянии базы данных (существующие индексы, текущую кардинальность переменных отношения и т.п.) и сведения

[3] Безусловно, что в данном контексте понятие уровня является относительным! По сути, упомянутые здесь операторы "низкого уровня" являются операторами реляционной алгебры (соединение, ограничение и т.д.), а они обычно рассматриваются как операторы высокого уровня.

о взаимных зависимостях, упоминавшихся выше, оптимизатор выбирает одну или несколько процедур-кандидатов для каждой низкоуровневой операции в запросе. Этот процесс обычно называют **выбором пути доступа** (см. также [18.33]).

Примечание. Следует отметить, что в [18.33] термин **выбор пути доступа** используется для ссылки на определенные в данной главе стадии 3 и 4, а не только на стадию 3. Действительно, на практике очень трудно разграничить, где заканчивается одна стадия и начинается другая; просто стадия 3 более или менее плавно переходит в стадию4.

Стадия 4. Генерация различных вариантов планов вычисления запроса и выбор плана с минимальными затратами

На последней стадии процесса оптимизации создается набор потенциальных **планов запроса,** после чего осуществляется выбор лучшего (т.е. наименее дорогостоящего) плана выполнения запроса. Каждый план выполнения формируется как комбинация некоторого набора потенциальных процедур реализации, причем каждой низкоуровневой операции в запросе соответствует одна процедура. Отметим, что обычно для поступившего запроса может существовать много планов выполнения запроса (возможно, даже слишком много). На практике, вероятно, не стоит генерировать все возможные планы запроса, так как одни из них могут быть комбинаторными версиями других планов выполнения этого же запроса, и сам процесс выбора наиболее эффективного плана может стать чрезмерно дорогостоящим. При выборе плана с наименьшей стоимостью рекомендуется (возможно, даже требуется) руководствоваться некоторыми эвристическими правилами, позволяющими ограничить количество анализируемых планов запросов разумными пределами, но см. [18.53]. Практику ограничения количества запросов разумными пределами иначе называют *сокращением пространства поиска,* поскольку ее можно рассматривать и как уменьшение диапазона анализируемых оптимизатором вариантов *{пространства поиска)* до контролируемых пределов.

Несомненно, для выбора плана с наименьшей стоимостью необходим метод определения стоимости любого возможного плана. По сути, стоимость плана — это просто сумма стоимостей отдельных процедур, входящих в его состав. Поэтому задача оптимизатора сводится к вычислению формул стоимости для каждой отдельной процедуры. Основная проблема состоит в том, что формулы стоимости выполнения процедуры зависят от размера отношения (или отношений), которое обрабатывает эта процедура. Поскольку все запросы, за исключением самых простых, требуют создания некоторых промежуточных результатов выполнения (по меньшей мере, концептуально), оптимизатор должен быть способен оценить размер этих промежуточных результатов и использовать полученные значения при вычислении формул стоимости. К сожалению, размеры промежуточных наборов данных сильно зависят от конкретных значений хранимых данных, поэтому точная оценка стоимости может оказаться достаточно сложной задачей. В [18.2], [18.3] обсуждаются некоторые подходы к решению этой проблемы и приводятся ссылки на другие публикации.

18.4. ПРЕОБРАЗОВАНИЕ ВЫРАЖЕНИЙ

В этом разделе обсуждаются правила (или законы) преобразования, которые могут оказаться полезными на стадии 2 процесса оптимизации. Подбор примеров и выяснение, в чем именно эти правила могут быть полезны при оптимизации, оставляем в качестве упражнений для читателя.

Безусловно, следует учитывать, что в результате применения к какому-либо выражению одного правила преобразования может быть получено выражение, к которому применимо другое правило преобразования. Например, среди исходных запросов не очень часто встречаются запросы, сформулированные с использованием двух последовательно выполняемых операций проекции (речь об этом пойдет ниже; см. второе правило из подраздела "Операция: сокращения и проекции"). Однако при преобразовании запросов подобные выражения достаточно часто возникают как результат применения других правил преобразования. (В этом смысле очень важным случаем является обработка представлений. Например, рассмотрим запрос "Выбрать все названия городов в представлении v", где представление V определено как проекция переменной отношения поставщиков по атрибутам S# и CITY.) Иначе говоря, начиная с исходного выражения, оптимизатор будет шаг за шагом применять различные правила преобразования до тех пор, пока не будет получено выражение, которое, согласно встроенным в оптимизатор эвристическим правилам, будет считаться *оптимальным* для рассматриваемого запроса.

Операции сокращения и проекции

Начнем с правил преобразования, которые включают только операции сокращения и проекции.

1. Последовательность операций сокращения одного и того же отношения может быть заменена единственной операцией сокращения этого отношения (причем ус ловные выражения всех исходных операций с помощью операций AND объединя ются в одно условное выражение). Например, выражение

   (A WHERE pi) WHERE p2

   эквивалентно следующему выражению:

   A WHERE pi AND p2

   Такое преобразование желательно, поскольку первоначальный вариант требует двух проходов в процессе обработки переменной отношения A, тогда как преобразованный вариант требует только одного прохода.

2. В последовательности операций проекции для одного и того же отношения можно игнорировать все проекции, кроме последней. Таким образом, выражение

   (A { ac11 }) { ac12 }

   эквивалентно следующему выражению:

   A { ac12 }

   Здесь ac11иac12 — разделенные запятыми списки имен атрибутов.

   Безусловно, чтобы исходное выражение имело смысл, каждый атрибут, присутствующий в проекции ac12, обязательно должен присутствовать и в проекции acll.

3. Операцию сокращения для результата операции проекции можно преобразовать в операцию проекции для результата операции сокращения. Например, выражние

   (A { acl }) WHERE p

   эквивалентно следующему выражению:

   (A WHERE p) { acl }

Как правило, операции сокращения целесообразно выполнять перед операциями проекции, поскольку операции сокращения обычно приводят к уменьшению объема тех данных, которые будут являться входными для операций проекции. Следовательно, в этом случае уменьшается количество данных, которые потребуется сортировать для исключения возможных дублирующихся записей, образующихся в процессе выполнения операций проекции.

Распределительный закон

Правило преобразования, которое использовано в примере, приведённом выше, в разделе 18.2 (преобразование операции соединения с последующей операцией сокращения в операцию сокращения, за которой следует операция соединения), на самом деле является частным случаем более общего правила, называемого *распределительным* законом. Говорят, что унарный оператор f **распределяется** по бинарной операции o тогда и только тогда, когда для всех A и B выполняется следующее тождество:

f (A O B) ≡ f (A) O f (B)

Например, в обычной арифметике операция SQRT (извлечение квадратного корня из неотрицательного числа) распределяется по операции умножения, так как для всех Аи B выполняется приведенное ниже тождество:

SQRT (A * B) ≡ SQRT (A) * SQRT (B)

Следовательно, выполняя преобразование выражений, оптимизатор арифметических выражений всегда может заменить одну часть этого равенства другой. В качестве обратного примера можно привести утверждение, что операция SQRT не распределяется по операции сложения, так как в общем квадратный корень из суммы A + в не равен сумме квадратных корней из A и B.

В реляционной алгебре операция сокращения распределяется по операциям **объединения, пересечения** и **разности.** Она также распределяется по операции **соединения,** но лишь тогда и только тогда, когда условие операции сокращения состоит (в самом сложном случае) из двух простых условий сокращения[4], соединенных операцией AND — по одному условию сокращения для каждого операнда операции соединения. В примере, рассматриваемом в разделе 18.2, сформулированное выше условие соблюдено (условие сокращения — очень простое и относится лишь к одному из операндов), благодаря чему стало возможным применить распределительный закон для замены заданного выражения более эффективным эквивалентом. В силу этого закона, операция сокращения была выполнена раньше остальных операций. Предварительное выполнение операций сокращения почти всегда себя оправдывает, поскольку приводит к существенному уменьшению количества кортежей, обрабатываемых следующей операцией, а также, возможно, к уменьшению количества кортежей на выходе этой операции.

Ниже приведено несколько примеров более специфических случаев применения распределительного закона, на этот раз для операции **проекции.** Во-первых, операция проекции распределяется по операциям **объединения** и **пересечения (**но не по операции, **разности);**

[4] Определение термина "простое условие сокращения" приведено в подразделе **"Операция ограни-**чения" раздела 7.4 главы 7.

```
( A UNION B ) { acl } ≡ A { ac1 } UNION B { acl }
( A INTERSECT B ) { acl } ≡ A { acl } INTERSECT B { acl }
```

Здесь A и B должны иметь одинаковые типы.

Во-вторых, операция проекции распределяется также по операции **соединения,** при условии, что в результате операции проекции сохраняются все атрибуты соединения, поэтому справедливо следующее тождество:

```
( A JOIN B ) { acl } ≡ ( A { acl1 } ) JOIN ( B { ac12 } )
```

Здесь acll — объединение атрибутов соединения и тех атрибутов acl, которые присутствуют только в Л, а ac12 — объединение атрибутов соединения и тех атрибутов acl, которые присутствуют только в в.

Эти законы можно использовать для организации предварительного выполнения операций проекции, что обычно себя оправдывает по тем же причинам, как и в случае операций сокращения.

Коммутативность и ассоциативность

Законы *коммутативности* и *ассоциативности* представляют собой два еще более важных и общих правила преобразования. Говорят, что бинарная операция о является **коммутативной** тогда и только тогда, когда для всех Л и *в* истинно следующее тождество:

A O B ≡ B O A

Например, в обычной арифметике операции умножения и сложения коммутативны, а операции деления и вычитания — нет. В реляционной алгебре коммутативными являются операции **объединения, пересечения** и **соединения,** а операции **разности** и **деления** таковыми не являются. Например, если запрос включает соединение двух отношений, A и B, то коммутативность означает, что безразлично, какое из отношений A и B выбрано в качестве *внешнего* или *внутреннего.* Следовательно, при вычислении конкретного соединения системе предоставлено право выбора в качестве *внешнего* отношения, например, меньшего из двух отношений (см. раздел 18.7). Перейдем к ассоциативности. Принято считать, что бинарная операция о является **ассоциативной** тогда и только тогда, когда для всех А, в и С истинно следующее тождество:

A O (B O C) ≡ (A O B) O C

Например, в обычной арифметике умножение и сложение — ассоциативные операции, а деление и вычитание — нет. В реляционной алгебре ассоциативными являются операции **объединения, пересечения** и **соединения,** а операции **разности** и **деления** таковыми не являются. Так, например, если в запросе используется соединение трех отношений, А, в и с, то из законов коммутативности и ассоциативности следует, что не имеет значения, в каком порядке будет выполняться соединение отношений. Поэтому в процессе вычисления подобного соединения системе предоставляется право выбора наиболее эффективной последовательности из всех существующих.

Идемпотентность и поглощение

Еще одним важным общим правилом является закон *идемпотентности.* Бинарную операцию о называют **идемпотентной** тогда и только тогда, когда для любого А выполняется следующее тождество:

A O A ≡ A '

Можно ожидать, что применение закона идемпотентности окажется полезным в процессе преобразования выражений. В реляционной алгебре операции **объединения, пересечения** и **соединения** являются идемпотентными, а операции деления и **разности** — нет.

Операции объединения и пересечения подчиняются также приведенным ниже полезным правилам поглощения.

```
A UNION ( A INTERSECT B ) ≡ A
A INTERSECT ( A UNION B ) ≡ A
```

Вычисляемые выражения

Объектом применения законов преобразования являются не только реляционные выражения. Например, выше уже упоминалось, что некоторые законы преобразования применимы и к *арифметическим* выражениям. Вот конкретный пример. Вследствие того, что операция умножения"*" распределяется по операции сложения "+", выражение

```
A * B + A * C
```

можно преобразовать в следующее выражение:

```
A * ( B + C )
```

Оптимизатор реляционных выражений должен иметь информацию о возможности подобных преобразований, так как ему приходится иметь дело с вычисляемыми выражениями такого рода в контексте операций расширения и агрегирования.

Следует отметить, что приведенный пример иллюстрирует несколько более общую форму распределительного закона. Выше было дано определение понятия распределяемое™ по отношению к *унарным* операциям, распределяемым по *бинарным* операциям. Однако в данном случае обе операции, "*" и "+", являются *бинарными.* Говорят, что бинарная операция 5 **распределяется** по бинарной операции о тогда и только тогда, когда для всех А, В и С истинно следующее тождество:

```
A 6 ( B O C ) ≡ ( A 6 B ) O ( A 5 C )
```

(В приведенном выше арифметическом примере 5 можно рассматривать как "*", а о — как"+".)

Логические выражения

Перейдем к обсуждению *логических* выражений (иначе называемых *булевыми* или *условными* выражениями), результатами вычисления которых могут быть только значения *истина* и *ложь.* Предположим, что А и в — это атрибуты двух различных отношений, тогда следующее логическое выражение

```
A > B AND B > 3
```

безусловно, эквивалентно выражению

```
A > B AND B > 3 AND A > 3
```

и потому может быть преобразовано в это выражение.

Данная эквивалентность основана на том, что операция ">" является транзитивной. Отметим, что подобное преобразование весьма полезно, поскольку позволяет системе выполнить дополнительную операцию сокращения (по А) до выполнения операции по

условию "больше", которое следует из сравнения "A > в". Повторим сказанное выше. Предварительное выполнение операций сокращения чаще всего оправдывает себя, поэтому будет полезной и способность системы **логически выводить** информацию о предварительно применимых операциях сокращения (что и было показано в данном примере).

Примечание. Этот прием реализован в различных коммерческих продуктах, включая СУБД DB2 (в которой его называют *транзитивным замыканием предикатов),* а также СУБД Ingres.

Ниже приведен еще один пример. Вследствие того, что операция OR распределяется по операции AND, логическое выражение

```
A > B OR ( C = D AND E < F )
```

можно преобразовать в следующее выражение:

```
( A > B OR C = D ) AND ( A > B OR E < F )
```

Этот пример демонстрирует другой общий закон: "Любое логическое выражение может быть преобразовано в эквивалентное выражение, представленное в **конъюнктивной нормальной форме (КНФ)".** Выражение в КНФ в общем случае имеет следующий вид:

```
Cl AND C2 AND ... AND Cn
```

Здесь все термы Cl, C2, ..., Cп — это логические выражения (называемые **конъюнктами),** в которых не используется логическая операция AND. Преимущество КНФ состоит в том, что выражение в КНФ *истинно* только в том случае, если истинны *все* его конъюнкты. И наоборот, выражение в **КНФ** *ложно,* если *ложью* является результат вычисления хотя бы *одного* конъюнкта. Поскольку логическая операция AND коммутативна (выражение A AND в эквивалентно выражению BAND л), оптимизатор может вычислять отдельные конъюнкты в любом порядке, в частности, по возрастанию их сложности (начиная с самых простых). Как только будет найден конъюнкт, результатом вычисления которого является *ложь,* процесс вычисления выражения в КНФ можно прекратить. Более того, в системах с параллельной обработкой возможно параллельное вычисление каждого из конъюнктов [18.56]—[18.58]. Опять же, как только будет найден первый конъюнкт, результатом вычисления которого является значение *ложь,* процесс вычисления выражения в КНФ прекращается.

Из данного и предыдущего подраздела следует, что оптимизатор должен иметь информацию о том, как общие свойства, такие как распределенность, применяются не только к **реляционным** операциям, таким как соединение, но и к операторам **сравнения,** например ">", к **логическим** операторам, например AND и OR, к **арифметическим** операторам, например "+" и т.п.

Семантические преобразования

Рассмотрим следующее выражение.

```
( SP JOIN S ) { P# }
```

Данное соединение относится к соединениям типа *"внешний ключ—соответствующий потенциальный ключ".* В нем внешнему ключу в переменной отношения SP ставится в соответствие потенциальный ключ в переменной отношения S. Поэтому каждый кортеж в переменной отношения SP соединяется с определенным кортежем в переменной отношения S. Таким образом, из каждого кортежа в переменной отношения SP в общий результат поступает значение атрибута р#. Другими словами, соединение

можно вообще не выполнять! Рассматриваемое выражение можно заменить следующим выражением.

```
SP { P# }
```

Тем не менее, будьте внимательны — подобное преобразование применимо *только* в силу семантики (смысла) конкретной ситуации. В общем случае каждый операнд в операции соединения вносит в соединение кортежи, которые не имеют соответствующих им кортежей в других операндах, и поэтому некоторые кортежи не вносят свой вклад в общий результат. Таким образом, чаще всего преобразования подобного типа будут недопустимыми. Однако в рассматриваемом конкретном случае каждый кортеж в переменной отношения SP обязательно имеет соответствующий ему кортеж в переменной отношения S согласно установленному ограничению целостности (фактически это ограничение *ссылочной* целостности), которое гласит, что каждая поставка деталей должна быть связана с определенным поставщиком. Следовательно, учитывая это замечание, можно утверждать, что в данном случае рассматриваемое преобразование вполне допустимо.

Преобразование, которое является допустимым только в силу того, что имеется конкретное установленное ограничение целостности, называют семантическим преобразованием [18.25], а оптимизацию, достигаемую в результате подобных преобразований,— семантической оптимизацией. Семантическую оптимизацию можно определить как процесс преобразования одного запроса в другой, качественно отличный запрос, который, тем не менее, гарантирует результат, идентичный результату исходного запроса, благодаря тому, что обрабатываемые данные удовлетворяют определенному ограничению целостности.

Важно понимать, что, в принципе, *любое ограничение целостности* может быть использовано для семантической оптимизации. Другими словами, этот прием не ограничен использованием только ограничений ссылочной целостности, как в данном случае. Например, предположим, что в базе данных поставщиков и деталей установлено ограничение, согласно которому все детали красного цвета должны храниться в Лондоне. Рассмотрим следующий запрос.

> *Определить всех поставщиков, которые поставляют только детали красного цвета и находятся в том же городе, где хранится по меньшей мере одна поставляемая ими деталь.*

Это довольно сложный запрос! Но в силу рассмотренного ограничения целостности его можно привести к такому виду.

> *Определить всех поставщиков из Лондона, поставляющих детали только красного цвета.*

Примечание. Насколько известно автору, семантическая оптимизация в должной мере используется лишь в немногих современных коммерческих продуктах. Но, в принципе, семантическая оптимизация способна обеспечивать значительное повышение производительности (весьма вероятно, намного превышающее то, которое в настоящее время может быть достигнуто с помощью любых традиционных приемов оптимизации). Более подробно идея семантической оптимизации изложена в [18.13], [18.26]—[18.28] и особенно — в [18.25].

Заключительные замечания

В завершение хотелось бы подчеркнуть фундаментальную важность реляционного свойства замкнутости в отношении всех аспектов, обсуждавшихся в этом разделе. Наличие реляционного свойства замкнутости означает, что можно применять вложенные выражения, а это, в свою очередь, означает, что единственный запрос может быть представлен единственным выражением вместо некоторой процедуры, состоящей из нескольких выражений, и поэтому нет необходимости в анализе потоков. Вложенные выражения рекурсивно определяются в терминах подвыражений, что позволяет оптимизатору использовать множество стратегий вычисления по принципу "разделяй и властвуй" (см. раздел 18.6). И конечно же, при отсутствии реляционного свойства замкнутости не имели бы смысла общие законы, такие как правило дистрибутивности и т.п.

18.5. СТАТИСТИЧЕСКИЕ ПОКАЗАТЕЛИ БАЗЫ ДАННЫХ

На стадиях 3 и 4 общего процесса оптимизации (называемых *стадиями выбора пути доступа)* используются **статистические показатели базы данных,** хранящиеся в ее каталоге (дополнительные сведения о том, как используются эти **статистические показатели,** приведены в разделе 18.7). В демонстрационных целях ниже кратко рассматриваются (с небольшими дополнительными комментариями) некоторые из основных статистических показателей, используемых в двух коммерческих продуктах, — СУБД DB2 и Ingres. Приведем некоторые из основных статистических показателей[5], применяемых в СУБД DB2.

- Для каждой базовой таблицы фиксируются следующие показатели:
  - кардинальность;
  - количество страниц, занятых таблицей;
  - доля табличного пространства, занимаемого таблицей.
- Для каждого *столбца* каждой базовой таблицы фиксируются следующие показатели:
  - количество различных значений в столбце;
  - второе наибольшее значение в столбце;
  - второе наименьшее значение в столбце;
  - десять значений в столбце (только для индексированных столбцов), которые встречаются чаще всего, а также количество вхождений каждого из этих значений.
- Для каждого *индекса* фиксируются следующие показатели:
  - индикатор, указывающий, является ли индекс кластеризованным (т.е. индексом, в котором логический порядок значений ключа совпадает с физическим порядком размещения этих значений на диске);
  - для кластеризованных индексов — доля индексированной таблицы, находящейся в кластеризующей последовательности;

[5] Поскольку СУБД DB2 и Ingres относятся к категории систем с поддержкой SQL, в них вместо терминов *переменная* отношения и *атрибут* используются термины *таблица* и *столбец.* В связи с этим указанные термины используются и в настоящем разделе. Кроме того, следует отметить, что в обоих продуктах фактически подразумевается, что базовые таблицы отображаются непосредственно на хранимые таблицы.

- количество листовых страниц в индексе;
- количество уровней в индексе.

Примечание. Перечисленные выше статистические показатели не обновляются *в реальном масштабе времени* (т.е. при каждом обновлении базы данных) из-за больших издержек, которые потребовались бы при использовании такого подхода. Вместо этого статистические показатели обновляются избирательно, с помощью системной утилиты RUNSTATS, которая запускается по требованию администратора базы данных, например после реорганизации базы данных. Аналогичное утверждение применимо и к большинству других коммерческих продуктов (но не ко всем), в том числе к системе Ingres (см. следующий абзац), где соответствующая утилита называется OPTIMIZEDB.

Перечислим некоторые из основных статистических показателей базы данных, накапливаемых в СУБД Ingres.

Примечание. В системе Ingres индекс рассматривается как частный случай хранимой таблицы. Поэтому приведенные ниже статистические показатели для базовых таблиц и столбцов вычисляются также для индексов.

- Для каждой *базовой таблицы* фиксируются следующие показатели:
  - кардинальность;
  - количество первичных страниц для таблицы;
  - количество страниц переполнения для таблицы.
- Для каждого *столбца* в каждой базовой таблице фиксируются следующие показатели:
  - количество различных значений в столбце;
  - максимальное, минимальное и среднее значения для столбца;
  - реальные значения в столбце и частота их вхождений.

18.6. СТРАТЕГИЯ ОРГАНИЗАЦИИ РАБОТЫ ПО ПРИНЦИПУ "РАЗДЕЛЯЙ И ВЛАСТВУЙ"

Как уже упоминалось выше, в конце раздела 18.4, реляционные выражения рекурсивно определяются в терминах подвыражений, что позволяет оптимизатору применять различные стратегии оптимизации по принципу "разделяй и властвуй". Отметим, что использование подобных стратегий особенно привлекательно в средах, поддерживающих параллельные вычисления, в частности, в распределенных системах, в которых различные части запроса могут выполняться параллельно на разных процессорах [18.56]—[18.58]. В данном разделе рассматривается одна из подобных стратегий, получившая название декомпозиция запросов. Впервые она била применена в прототипе системы Ingres [18.34], [18.35].

Основная идея метода декомпозиции запросов состоит в том, что запрос, включающий большое количество переменных области значений[6], разбивается на последовательность запросов меньшего размера обычно с одной или двумя такими переменными в каждом. Требуемая декомпозиция достигается с помощью методов *отделения* и *подстановки кортежей,* которые описаны ниже.

---

[6] Напомним, что в языке запросов QUEL системы INGRES используются средства реляционного исчисления.

- Отделение — это процесс удаления из запроса такого компонента, который имеет только одну общую переменную с остальной частью запроса.
- **Подстановка кортежа** — это процесс подстановки значения одной переменной в запросе (один кортеж за одну операцию).

Использование метода отделения всегда предпочтительней использования метода подстановки кортежей во всех случаях, когда существует возможность выбора. Тем не менее, рано или поздно декомпозиция с помощью метода отделения обязательно приведет к разбиению запроса на множество меньших запросов, которые больше нельзя будет подвергнуть декомпозиции с помощью этого метода, после чего придется обратиться к методу подстановки кортежей.

Ниже рассмотрен пример декомпозиции (основанный на примере из [18.34]). Словесное выражение этого запроса имеет следующий вид: "Определить имена поставщиков из Лондона, поставляющих некоторые детали красного цвета весом меньше 25 фунтов в количестве больше 200 штук". Приведем формулировку этого запроса на языке QUEL, на которую далее будем ссылаться, как на формулировку запроса Q0.

```
Q0: RETRIEVE ( S.SNAME) WHERE S.CITY   = 'London'
                        AND   S.S#     = SP.S#
                        AND   SP.QTY   > 200
                        AND   SP.P#    = P.P#
                        AND   P.COLOR  = 'Red'
                        AND   P.WEIGHT < 25
```

Здесь переменными области значений являются s, P и SP, причем каждая из них распространяется на всю базовую переменную отношения с тем же именем.

Исходя из последних двух сравнений в выражении запроса можно заключить, что нас интересуют только детали красного цвета весом меньше 25 фунтов. Поэтому можно отделить подзапрос с одной переменной (на самом деле являющийся проекцией сокращения), в котором используется переменная P.

```
D1: RETRIEVE INTO P'  (P.P#) WHERE P.COLOR  = 'Red'
                           AND   P.WEIGHT < 25
```

Этот запрос с единственной переменной может быть отделен, поскольку в нем присутствует только одна переменная (а именно — переменная P), совместно используемая с остальной частью запроса. Так как запрос D1 связан с остальной частью исходного запроса через атрибут P# (в условии сравнения SP. P# = P. P#), атрибут P# должен входить в *кортеж-прототип* (см. главу 8) отделенного запроса. Иными словами, *отделенный запрос* предназначен для выборки номеров только тех деталей красного цвета, которые весят меньше 25 фунтов. Отделенный запрос обозначим как запрос D1, результатом выполнения которого является временная переменная отношения p'. (Назначение конструкции INTO состоит в том, чтобы создать новую переменную отношения P' с единственным атрибутом P#. Эта переменная отношения создается автоматически и содержит результат выполнения операции RETRIEVE.) Наконец, заменим ссылки на переменную отношения P в сокращенной версии запроса Q0 ссылками на переменную отношения p'. Эту новую сокращенную версию исходного запроса обозначим как запрос Q1 (этот запрос приведен ниже).

```
Q1: RETRIEVE (S.SNAME) WHERE S.CITY =     'London'
                       AND   S.S#   =SP.S#
                       AND   SP.QTY >200
                       AND   SP.P#  =P1.P#
```

Еще раз применим аналогичный прием к запросу Q1, отделив от него запрос, в котором используется единственная переменная SP. Присвоим отделенному запросу имя D2, а оставшуюся после Отделения часть запроса Q1 назовем Q2, как показано ниже.

```
D2: RETRIEVE INTO SP' (SP.S#, SP.P#) WHERE SP.QTY > 200

Q2: RETRIEVE (S.SNAME) WHERE S.CITY = 'London'
                       AND   S.S#   =
                       SP'.S# AND   SP'.P#
                       = P'.P#
```

Далее тем же способом отделим запрос с единственной переменной S.

```
D3: RETRIEVE INTO S'  (S.S#, S.SNAME) WHERE S.CITY = 'London'

Q3: RETRIEVE (S'.SNAME) WHERE S'.S#  = SP'.S#
                       AND   SP'.P# = P'.P#
```

Наконец, отделим еще один запрос, в котором используются переменные SP' и P'.

```
D4: RETRIEVE INTO SP''  (SP'.S#) WHERE SP'.P# = P'.P#

Q4: RETRIEVE (S'.SNAME) WHERE S'.S# = SP''.S#
```

В результате исходный запрос Q0 оказался разбитым на три запроса с одной переменной, Dl, D2 и D3 (каждый из которых является проекцией сокращения), и два запроса с двумя переменными, D4 и Q4 (каждый из которых является проекцией соединения). Сложившуюся в результате ситуацию можно схематично представить в виде древовидной структуры, показанной на рис. 18.3.

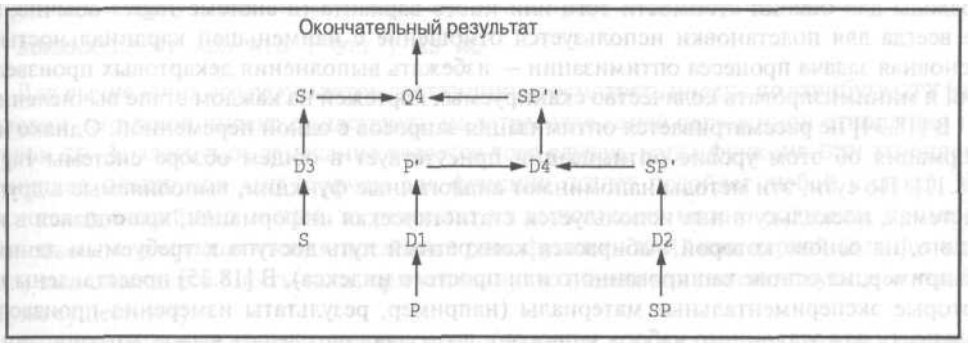


Рис. 18.3. Дерево декомпозиции запроса Q0

Эту схему можно интерпретировать, как описано ниже.

- В запросах Dl, D2 и D3 в качестве входных данных используются переменные отношения P, SP и S (а точнее, те отношения, которые являются текущими значениями переменных отношения p, SP и S), соответственно, которые и помещают результат своего выполнения во временные переменные отношения P', SP' и S ', соответственно.s

- Далее выполняется запрос D4, использующий в качестве входных данных временные переменные отношения P' и SP[1] и помещающий результат своего выполнения во временную переменную отношения SP' '.
- Наконец, выполняется запрос Q4, использующий в качестве входных данных переменные отношения S и SP", результат выполнения которого и является результатом выполнения исходного запроса.

Обратите внимание на то, что запросы Dl, D2 и D3 полностью независимы один от другого и их можно обрабатывать в любом порядке (даже параллельно). Запросы D3 и D4 также можно обрабатывать в любом порядке, но только после получения результатов выполнения запросов D1 и D2. Но запросы D4 и Q4 нельзя подвергнуть дальнейшей декомпозиции, поэтому их следует обрабатывать с помощью метода подстановки кортежей (что обычно означает необходимость применения *последовательного поиска,* называемого также подходом, основанном на использовании "грубой силы", *поиска по индексу* или *хэшированного поиска*— см. раздел 18.7). В качестве примера рассмотрим выполнение запроса Q4. Обратившись к обычному набору данных, используемых в этой книге в качестве примеров, можно определить, что множество номеров поставщиков в атрибуте SP' ' . s# будет выглядеть так: {SI, S2, S4 }. Каждое из этих трех значений по очереди должно быть подставлено в атрибут SP' ' . s#. В результате запрос Q4 примет такой вид, как будто он был записан следующим образом.

```
RETRIEVE   (S'.SNAME)   WHERE
                  S'.S#   =   'SI'
                  OR     S[1].S#   =   'S2'
                  OR     S'-S#   =   'S4'
```

В [18.34] приведены алгоритмы разбиения исходного запроса на меньшие запросы и выбора переменных для подстановки кортежей. Именно от выбора переменных во многом зависят фактические результаты оптимизации; в [18.34] приведены эвристические подходы для оценки стоимости того или иного варианта (в системе Ingres обычно, но не всегда для подстановки используется отношение с наименьшей кардинальностью). Основная задача процесса оптимизации — избежать выполнения декартовых произведений и минимизировать количество сканируемых кортежей на каждом этапе вычисления.

В [18.34] не рассматривается оптимизация запросов с одной переменной. Однако информация об этом уровне оптимизации присутствует в общем обзоре системы Ingres [18.11]. По сути, эти методы напоминают аналогичные функции, выполняемые в других системах, поскольку в них используется статистическая информация, хранящаяся в каталоге, на основе которой выбирается конкретный путь доступа к требуемым данным (например, на основе хэшированного или простого индекса). В [18.35] представлены некоторые экспериментальные материалы (например, результаты измерения производительности для эталонного набора запросов), позволяющие сделать вывод, что описанные выше методы оптимизации СУБД Ingres дают хорошие результаты и достаточно эффективны при применении на практике. Ниже приведены некоторые выводы, взятые из этой работы.

1. Метод отделения — это лучшая из методик, которую можно применять на первом этапе оптимизации.
2. Если на первом этапе *необходимо* выполнять подстановку кортежей, то для этого лучше всего использовать переменную, по которой выполняется соединение.

3. Если к одной из переменных в запросе с двумя переменными применялась подстановка кортежей, то оптимальная тактика заключается в формировании *динамически,* по мере необходимости, простого или хэшированного индекса по атрибуту соединения, принадлежащему другому отношению (такой метод фактически применяется в СУБД Ingres).

18.7. РЕАЛИЗАЦИЯ РЕЛЯЦИОННЫХ ОПЕРАТОРОВ

В этом разделе представлено краткое описание некоторых очевидных методов реализации отдельных реляционных операторов, в частности оператора соединения. Включение данного материала в книгу было вызвано стремлением развеять ту "таинственность", которая присуща описанию процесса оптимизации. Обсуждаемые далее методы соответствуют механизмам, которые в разделе 18.3 были названы *низкоуровневыми процедурами реализации.*

Примечание. Более сложные приемы реализации описаны в аннотациях к определенным работам, ссылки на которые приведены в конце главы. См. также приложение А.

Для простоты предположим, что кортежи и отношения физически хранятся на диске так, как они организованы логически. Далее будет рассмотрено выполнение операторов проекции, соединения и агрегирования, причем под операторами агрегирования будут подразумеваться оба указанных ниже основных варианта.

1. Операнд PER вообще не определяет атрибутов ("PER TABLE_DEE").
2. Операнд PER, по меньшей мере, определяет один атрибут.

Первый вариант достаточно прост. Как правило, в этом случае выполняется просмотр всего отношения, для которого выполняется агрегирование, за исключением варианта, когда агрегируемый атрибут является индексируемым, поскольку при этом требуемый результат можно вычислить исключительно с помощью индекса, без необходимости обращения к самому отношению. Например, рассмотрим следующее выражение.

```
SUMMARIZE SP ADD AVG ( QTY ) AS AQ
```

Для вычисления его результатов достаточно просмотреть индекс по атрибуту QTY (при условии, что такой индекс существует), не затрагивая *самой* переменной отношения поставок SP. Аналогичное замечание касается того случая, когда функция SUM заменяется функцией COUNT или AVG (причем для функции COUNT подойдет любой индекс). Что касается функций MAX и MIN, то результат их выполнения можно получить, применив *единственную операцию* чтения последней (для функции MAX) или первой (для функции MIN) записи индекса (опять-таки, при условии, что индекс для соответствующего атрибута уже существует).

В последней части данного раздела предполагается, что в качестве операции *агрегирования* рассматривается именно операция, описанная в случае 2. Ниже показан пример операции агрегирования второго типа.

```
SUMMARIZE SP PER P { P# } ADD SUM ( QTY ) AS TOTQTY
```

С точки зрения пользователя операции проекции, соединения и агрегирования (второго типа), безусловно, абсолютно не похожи одна на другую. Но с точки зрения реализации они имеют некоторое сходство, поскольку в каждом из случаев система должна сгруппировать кортежи на основе общих значений указанных атрибутов. В случае операции

проекции подобная группировка позволяет системе исключить кортежи-дубликаты, в случае операции соединения — найти соответствующие друг другу кортежи, а в случае операции агрегирования — вычислить отдельные агрегированные значения для каждой группы. Ниже перечислены несколько методов подобной группировки ксртежей.

1. Последовательный просмотр (или метод "грубой силы").
2. Поиск по индексу.
3. Поиск с помощью хэш-функции.
4. Слияние.
5. Хэширование.
6. Сочетания методов 1-5.

На рис. 18.4—18.8 приведены псевдокоды процедур реализации перечисленных методов для операции соединения (операции проекции и агрегирования оставлены читателю в качестве упражнения). На этих рисунках используются следующие обозначения. Прежде всего, R и S — это отношения, которые должны быть соединены, а с — их общий атрибут (возможно, составной). Предполагается, что возможен последовательный доступ к кортежам обоих отношений R и s по одному за одну операцию. Эти кортежи в последовательности доступа будут обозначаться как R[1], R[2], ..., R[m] и S[1], S[2], ..., s [n], соответственно. Для соединенного кортежа, составленного из атрибутов кортежей R[i] и S[j], будет использоваться обозначение R[i] * S[j]. Наконец, переменную отношения R будем считать **внешней,** а переменную отношения S — **внутренней** (поскольку они управляют внешним и внутренним циклами просмотра, соответственно).

Последовательный просмотр

Метод последовательного просмотра (или "грубой силы") иногда также называют *простым методом.* В этом случае рассматриваются все возможные комбинации кортежей (т.е. каждый кортеж отношения R проверяется в сочетании с каждым кортежем отношения S, как показано на рис. 18.4).

Примечание. Метод последовательного просмотра часто называют *методом вложенных циклов,* но это название нельзя считать приемлемым, поскольку вложенные циклы используются в реализациях всех названных выше методов.

```
do i := 1 to m ;                /* Внешний цикл     */
   do j := 1 to n ;             /* Внутренний цикл */
      if R[i].C = S[j].C then
      добавить соединяемый кортеж R[i] * S[j] к результату ;
   end ;
end ;
```

Рис. 18.4. Метод последовательного просмотра

Проанализируем затраты, связанные с использованием метода последовательного просмотра.

Примечание. Здесь мы ограничимся только анализом затрат на выполнение операций ввода-вывода, хотя на практике может потребоваться также учесть другие параметры (например, затраты процессорного времени).

Прежде всего, ясно, что для реализации этого метода потребуется всего m + (n * m) операций чтения кортежей. А как в отношении операций записи кортежей? Иначе говоря, какова кардинальность результата операции соединения? (Количество операций записи кортежа будет равно кардинальности результирующего отношения, если последнее требуется записывать на диск.)

- В частном, но достаточно важном случае соединения типа "многие к одному" (например, соединение типа "внешний ключ к соответствующему потенциально му ключу") кардинальность результирующего отношения почти всегда точно сов падает с *величиной т или п в зависимости от того, какое из отношений, R или S,* расположено на стороне внешнего ключа рассматриваемого соединения.
- Теперь рассмотрим более общий случай соединения типа "многие ко многим". Пусть dCR — количество различающихся значений атрибута с отношения R, по которому выполняется соединение, а dCS имеет тот же смысл, но для отношения S. Если предположить, что значения атрибутов *распределены равномерно* (т.е. любое значение атрибута с может встретиться с той же вероятностью, что и другие значе ния), то для данного кортежа отношения R существует n/dCS соответствующих кортежей в отношении S со значением атрибута с, равным значению этого атрибу та в рассматриваемом кортеже из отношения R. Таким образом, общее количество кортежей в соединении (т.е. кардинальность результирующего отношения) будет равно (т * n) /dCS. Или, если повторить изложенные рассуждения, начав с кор тежа в отношении S, то кардинальность результирующего отношения составит (п * m)/dCR. Эти две оценки будут различными, если dCS * dCR, т.е. если в отношении R существуют значения атрибута с, которые не встречаются в отношении s, и наоборот. В этом случае следует использовать меньшую из оценок.

Безусловно, как было отмечено в разделе 18.2, на практике применяются операции ввода-вывода *страниц,* а не *кортежей.* Поэтому предположим, что в отношениях R и S на странице помещается, соответственно, pR и pS кортежей (т.е. отношения занимают m/pR и n/ps страниц, соответственно). Теперь легко увидеть, что процедура, псевдокод которой показан на рис. 18.4, выполнит (m/pR) + (m*n) /pS операций чтения страниц. Аналогичным образом, если поменять ролями отношения R и S (отношение S считать внешним, а R— внутренним), количество операций чтения страниц составит (n/pS) + (n*m) /pR.

Для примера предположим, что m = 100, п = 10 000, pR = 1 и pS = 10. Тогда результатами вычисления последних двух формул будут значения 100 100и1 001 000 операций чтения страниц, соответственно.

Заключение. В описанном подходе с применением последовательного просмотра в качестве внешнего отношения желательно использовать меньшее из двух исходных отношений (в данном случае понятие *меньшее* относится к количеству занимаемых страниц внешней памяти).

Завершая краткое обсуждение метода последовательного просмотра, заметим, что этот прием является наихудшим из всех. В нем предполагается, что для отношения S не предусмотрен индекс, и не применяется хэш-функция для доступа к атрибуту соединения с. Эксперименты, проведенные Биттоном (Bitton) и др. [18.6], показали, что если это предположение (отсутствие индексов и хэш-функции) действительно имеет место, то

методы обработки можно улучшить, формируя индекс или применяя хэш-функцию динамически и продолжая обработку запроса по методу соединения с поиском по индексу или с помощью хэш-функции (см. следующие два подраздела). Как упоминается в конце предыдущего раздела, эта идея поддерживается и в [18.35].

Поиск по индексу

Теперь рассмотрим ситуацию, в которой для атрибута C внутреннего отношения S существует индекс X (рис. 18.5). Преимущество поиска по индексу по сравнению с методом последовательного просмотра состоит в том, что благодаря наличию индекса X для данного кортежа внешнего отношения R возможен переход непосредственно к соответствующему кортежу внутреннего отношения S. Общее количество операций чтения кортежей из отношений R и s будет равно кардинальности результирующего отношения операции соединения. Общее количество операций чтения страниц для отношений R и S при наихудшем предположении, что каждая операция чтения кортежа из отношения S требует обращения к отдельной странице, составит (m/pR) + (mn/dCS).

Но если кортежи отношения S хранятся в порядке значений атрибута соединения C, то количество операций чтения страницы сокращается до значения (m/pR) + (mn/dCS) /pS. Воспользовавшись теми же примерами и значениями, что и выше (т = 100, п = 10 000, pR = 1, pS = 10), и предположив, что dCS = 100, получим в результате вычисления двух последних формул значения 10 100 и 1100, соответственно. Разница между полученными значениями показывает, что кортежи хранимых отношений целесообразно размещать в "подходящей" физической последовательности [18.7].

```
/* Предполагается наличие индекса X, сформированного по атрибуту S.C */

do i := 1 to m ;                    /* Внешний цикл */
   /* Допустим, что имеется k записей индекса X[1], ..., X[k] */
   /* с индексированным значением атрибута = R[i].C */
   do j := 1 to k ;                 /* Внутренний цикл */
      /* Допустим, что кортежем отношения S, индексированным*/
      /* с помощью X[j], является S[j] */
      добавить соединяемый кортеж R[i] * S[j] к результату ;
   end ;
end ;
```

Рис. 18.5. Поиск по индексу

Однако при оценке затрат следует учитывать и издержки, связанные с выполнением операций чтения в самом индексе X. Наихудшим является предположение, что при поиске соответствующих кортежей в отношении S для каждого кортежа в отношении R потребуется выполнить *непредвиденный* поиск по индексу, требующий чтения страницы для каждого уровня индекса. Для индекса, обладающего x уровнями, операция поиска добавит к общему количеству операций чтения страницы еще m*x операций. На практике x обычно имеет значение 3 или меньше. (Более того, весьма вероятно, что верхний уровень индекса на протяжении всей обработки данных будет находиться в оперативной памяти, что значительно сократит количество операций чтения страниц.)

Поиск с помощью хэш-функции

Поиск с помощью хэш-функции аналогичен поиску по индексу, за исключением того, что в качестве *быстрого пути доступа* к значениям атрибута соединения S. С внутреннего отношения S вместо индекса используется хэш-функция (рис. 18.6). Определение оценки затрат, связанных с выполнением данного метода, оставляем в качестве упражнения для читателя.

Метод слияния

В методе слияния предполагается, что кортежи отношений R и S физически хранятся во внешней памяти в последовательности значений атрибута с, по которому выполняется соединение. Если данное предположение отвечает действительности, то два отношения можно будет просматривать в их физической последовательности, причем обе операции просмотра можно синхронизировать, в результате чего соединение может быть выполнено за один проход по данным (это утверждение истинно, по крайней мере, для соединений типа "один ко многим", но не всегда истинно для соединений типа "многие ко многим"). Несомненно, подобный метод будет оптимальным при соблюдении указанных допущений, поскольку каждая страница данных считывается всего один раз (рис. 18.7). Другими словами, количество операций чтения страниц составит (m/pR) + (n/pS).

```
/* Предполагается, что на основе атрибута S.C создана хэш-таблица H */

do i := 1 to m ;                    /* Внешний цикл */
   k := hash (R[i].C) ;
   /* Допустим, что имеется h кортежей S[1], ..., S[h], хранимых */
   /* в записях хэш-таблицы H[k] */
   do j := 1 to h ;                 /* Внутренний цикл */
      if S[j].C = R[i].C then
      добавить соединяемый кортеж R[i] * S[j] к результату ;
   end ;
end ;
```

Рис. 18.6. Поиск с помощью хэш-функции

На основании этого можно сделать приведенные ниже заключения.

- Физическая кластеризация логически связанных данных является одним из важ нейших факторов, влияющих на производительность системы, т.е. весьма жела тельно проводить кластеризацию данных так, чтобы они соответствовали соеди нениям, наиболее важным для предприятия [18.7].
- Если подобная кластеризация отсутствует, то может применяться сортировка не посредственно во время выполнения запроса какого-то одного или обоих отноше ний произвольным способом с последующим соединением методом слияния на этапе прогона. (Безусловно, назначение подобной сортировки состоит в динами ческом создании требуемой кластеризации.) На этот метод обычно ссылаются, что **вполне логично, как** на метод **сортировки-слияния** [18.8].

```
/* Предполагается, что оба отношения, R и S, отсортированы по атрибуту C ; */
/* при разработке приведенного ниже кода принято предположение,            */
/* что выполняется соединение "многие ко многим"; более простой случай     */
/* соединения "один ко многим" оставлен в качестве упражнения              */

r := 1 ;
s := 1 ;
do while r ≤ m and s ≤ n ;                    /* Внешний цикл */
   v := R[r].C ;
   do j := s by 1 while S[j].C < v ;
   end ;
   s := j ;
   do j := s by 1 while S[j].C = v ;          /* Главный внутренний цикл */
      do i := r by 1 while R[i].C = v ;
         добавить соединяемый кортеж R[i] * S[j] к результату ;
      end ;
   end ;
   s := j ;
   do i := r by 1 while R[i].C = v ;
   end ;
   r := i ;
end ;
```

Рис. 18.7. Метод слияния (для соединений типа "многие ко многим")

Хэширование

Как и метод слияния, метод хэширования позволяет обойтись одним проходом по каждому из двух соединяемых отношений (рис. 18.8). В результате первого прохода строится хэш-таблица для отношения S по значениям атрибута соединения S.C. Записи в этой таблице содержат значение атрибута соединения (а также, возможно, значения других атрибутов) и указатель на соответствующий кортеж, хранящийся на диске. Во время второго прохода сканируются кортежи отношения R и применяется та же хэш-функция для значений атрибута соединения R. C. Когда по вычисленному для кортежа отношения R значению хэш-функции в хэш-таблице обнаруживается один или несколько соответствующих ему кортежей отношения S, алгоритм проверяет, действительно ли равны значения R.C и S.C, и, если это так, вырабатывает соответствующий кортеж (или кортежи) результирующего соединения. Основным преимуществом данного метода по сравнению с методом слияния является то, что кортежи отношений R и S можно хранить в произвольной последовательности, а значит, не требуется их предварительно сортировать.

```
/* Сформировать хэш-таблицу H по атрибуту S.C */

do j := 1 to n ;
   k := hash (S[j].C) ;
   добавить S[j] к записи хэш-таблицы H[k] ;
end ;
/* Теперь выполнить поиск в R с помощью хэш-функции */
```

Рис. 18.8. Метод хэширования

Как и в случае метода поиска с помощью хэш-функции, определение оценок затрат, связанных с использованием хэширования, оставляем в качестве упражнения для читателя.

18.8. РЕЗЮМЕ

Эта глава начинается с утверждения, что для реляционных систем оптимизация является как *проблемой*, так и *благоприятной возможностью*. Фактически оптимизация является сильной стороной таких систем, причем по целому ряду причин. Реляционная система с хорошим оптимизатором будет функционировать намного лучше, чем нереляционная. Приведенный вступительный пример дает ясное представление о тех поразительных результатах, которых можно достичь благодаря оптимизации (иногда эффективность выполнения запроса повышается в соотношении 10000:1).

Процесс оптимизации в общем случае включает четыре описанные ниже последовательные стадии.

- Представление запроса во **внутренней форме** (обычно это **дерево запроса** или **абстрактное синтаксическое дерево,** однако подобные представления вполне можно рассматривать как специфическую внутреннюю форму для выражений реляционной алгебры или реляционного исчисления).
- Преобразование в **каноническую форму** с помощью различных **законов преобразования.**
- Выбор подходящих **низкоуровневых процедур** реализации различных операторов **в** каноническом представлении запроса.
- Генерация **планов запроса** и выбор плана с наименьшими затратами, оцениваемыми с помощью формул **стоимости** и статистических показателей **базы данных.**

В этой главе далее обсуждались общие законы **распределения, коммутативности** и **ассоциативности** и их применение к реляционным операторам, таким как соединение (кроме того, рассматривалось применение этих законов к **арифметическим** и **логическим** операторам, а также к операторам **сравнения).** Затрагивались и другие общие законы — идемпотентности и **поглощения.** Затем обсуждались некоторые специальные преобразования операторов **сокращения** и **проекции.** После этого была представлена важная идея **семантических** преобразований, т.е. преобразований запросов, которые основаны на сведениях об **ограничениях целостности,** хранящихся в системе.

В качестве иллюстрации был сделан краткий обзор статистических показателей базы данных, используемых в СУБД DB2 и Ingres. Далее обсуждалась одна из стратегий, построенных по принципу "разделяй и властвуй", — **декомпозиция запросов,** впервые реализованная в прототипе системы Ingres. Было также отмечено, что стратегии, построенные по принципу "разделяй и властвуй", весьма привлекательны для систем с поддержкой параллельных вычислений и распределенных систем.

Наконец, рассматривались некоторые **методы реализации** отдельных реляционных операторов, в частности оператора **соединения.** Был представлен псевдокод алгоритмов для пяти методов выполнения операции соединения: **метод последовательного просмотра, поиск по индексу, поиск с помощью хэш-функции, метод слияния** (включая метод **сортировки—слияния)** и **метод хэширования.** Также кратко рассматривались стоимостные характеристики описанных методов.

В заключение хотелось бы отметить, что, к сожалению, многие современные программные продукты обладают некоторыми особенностями, **препятствующими оптимизации,** о которых пользователь должен, по крайней мере, знать (даже если в большинстве случаев **с** этим ничего нельзя поделать). Особенностями, препятствующими оптимизации, называются

такие функции рассматриваемых СУБД, которые не позволяют оптимизатору выполнять свою работу так, как она могла быть выполнена в ином случае (т.е. при отсутствии этих препятствий). К числу таких особенностей, препятствующих оптимизации, можно, например, отнести *дубликаты строк* (см. [6.6]), *трехзначную логику* (см. главу 19) и *реализацию трехзначной логики в языке* SQL (см. [19.6] и [19.10]).

В данной главе задачи оптимизации рассматривались так, как они обычно трактуются и реализуются в СУБД; иными словами, здесь показано, какие *эмпирические подходы* в основном применяются в данной области. Но в последнее время появились реализации принципиально нового подхода к организации работы СУБД, а этот подход, по сути, ставит под сомнение многие предположения, лежащие в основе указанных эмпирических подходов. Вследствие этого многие аспекты общего процесса оптимизации могут быть упрощены (а в некоторых случаях даже полностью исключены). К ним относятся перечисленные ниже и многие другие аспекты.

- Применение метода выбора пути доступа с учетом стоимости (стадии 3 и 4 указан ного процесса).
- Применение индексов и других обычных путей доступа.
- Осуществление выбора между компиляцией и интерпретацией запросов к базе данных.
- Применение алгоритмов для реализации реляционных операторов.

Дополнительная информация по этой теме приведена в приложении А.

УПРАЖНЕНИЯ

18.1.Одни пары приведенных здесь выражений в контексте базы данных поставщиков, деталей и проектов эквивалентны, а другие — нет. Какие пары выражений действительно эквивалентны?

```
а1) S JOIN ( ( P JOIN J ) WHERE CITY = 'London' ).

а2) ( P WHERE CITY = 'London' ) JOIN ( J JOIN S ) .

б1) ( S MINUS ( ( S JOIN SPJ ) WHERE P# = P# ('P2') )
              { S#, SNAME, STATUS, CITY } ) { S#, CITY } .

б2) S { S#, CITY } MINUS
     ( S { S#, CITY } JOIN

     ( SPJ WHERE P# = P# CP2[1]) ) ) { S#, CITY }.

в1) ( S { CITY } MINUS P { CITY } ) MINUS J { CITY }.

в2) ( S { CITY } MINUS J { CITY } )

      MINUS ( P { CITY } MINUS J { CITY } ) .

Г1) ( J { CITY } INTERSECT P { CITY } ) UNION S { CITY }.

Г2) J { CITY ) INTERSECT ( S { CITY } UNION P { CITY } ) .

Д1) ( ( SPJ WHERE S# = S# ('S1') ) UNION ( SPJ WHERE P# = P# ('P1') ) )
   INTERSECT ( ( SPJ WHERE J# = J# ('J1') )
               UNION ( SPJ. WHERE S# = S# ('S1') ) ) .
```

```
Д2) ( SPJ WHERE S# = S# ('S1[1]') ) UNION
    ( ( SPJ WHERE P# = P# ('P1') )
     INTERSECT ( SPJ WHERE J# = J# ('J1')
     ) ).

e1) ( S WHERE CITY = 'London' ) UNION ( S WHERE STATUS >
10 ). e1) S WHERE CITY = 'London' AND STATUS > 10.

Ж1)( S { S# } INTERSECT ( SPJ WHERE J# = J# ('J1') ) {
     S# } ) UNION ( S WHERE CITY = 'London' ) { S# }.

Ж2)S { S# } INTERSECT ( ( SPJ WHERE J# = J# ('J1') ) { S# }
                UNION ( S WHERE CITY = 'London' ) { S# } ).

31) ( SPJ WHERE J# = J# ('Jl') ) { S# }
    MINUS ( SPJ WHERE P# = P# ('P1') ) { S# }.

32) ( ( SPJ WHERE J# = J# ('J1') )
      MINUS ( SPJ WHERE P# = P# {'P1') ) ) { S# }.

И1) S JOIN ( P { CITY } MINUS J { CITY } ) .

И2) ( S JOIN P { CITY } ) MINUS ( S JOIN J { CITY } ).
```

18.2. Докажите, что операции соединения, объединения и пересечения являются ком мутативными, а операция разности — нет.

18.3. Докажите, что операции соединения, объединения и пересечения являются ассо циативными, а операция разности — нет.

18.4. Докажите, что:

а) объединение распределяется по пересечению;

б) пересечение распределяется по объединению.

18.5. Докажите законы поглощения.

18.6. Докажите следующее.

а) Операция сокращения безусловно распределяема по объединению, пересече нию и разности, а также условно распределяема по соединению.

б) Операция проекции безусловно распределяема по объединению и пересече нию, условно распределяема по соединению, а также не распределяема по раз ности.

в) Сформулируйте соответствующие критерии для условной распределяемости. "

18.7. Дополните изложенные в разделе 18.4 правила преобразования так, чтобы учиты вались операции расширения EXTEND и агрегирования SUMMARIZE.

18.8. Можно ли сформулировать какие-либо полезные правила преобразований для операции реляционного деления?

18.9. Сформулируйте подходящий набор правил преобразования условных выражений, в которых используются операторы AND, OR И NOT. Примером таких правил может служить закон коммутативности для оператора AND, т.е. утверждение, что выраже ние A AND в тождественно выражению в AND A.

18.10. Распространите этот ответ на предыдущее упражнение, включив в сферу его действия логические выражения, в которых используются кванторы EXISTS и FORALL. Примером может служить изложенное в главе *8 (раздел 8.2) правило,* позволяющее преобразовывать выражения с квантором FORALL в выражения с отрицаемым квантором EXISTS.

18.11. Ниже перечислены ограничения целостности для базы данных поставщиков, деталей и проектов (которые сформулированы на основе упражнений к главе 9).

- Единственными допустимыми городами являются Лондон (' London'), Париж ('Paris'), Р[им] ('Rome'), Афины ('Athens'), Осло ('Oslo'), Стокгольм (' Stockholm'), Мадрид ('Madrid') и Амстердам ('Amsterdam').
- Никакие два проекта не могут быть размещены в одном и том же городе.
- В любой момент в Афинах может находиться не больше одного постшщика.
- Ни одна поставка по количеству не может превышать удвоенное среднее зна чение количества по всем поставкам.
- Поставщик с наибольшим статусом не может находиться в одном городе с по ставщиком с наименьшим статусом.
- Каждый проект должен находиться в городе, в котором есть хотя бы один по ставщик для этого проекта.
- Должна существовать по крайней мере одна деталь красного цвета.^
- Среднее значение статуса поставщика должно быть больше 19.
- Каждый поставщик из Лондона должен поставлять деталь с номером Р2.
- Хотя бы одна деталь красного цвета должна весить меньше 50 фунтов.
- Поставщики из Лондона должны поставлять больше различных видов деталей, чем поставщики из Парижа.
- Поставщики из Лондона должны поставлять больше деталей (по общему коли честву), чем поставщики из Парижа.

Ниже перечислены простые запросы к этой базе данных:

а) выбрать сведения о поставщиках, которые не поставляют деталь с номером Р2;

б) выбрать сведения о поставщиках, которые не поставляют детали для какого-либо проекта в том же городе, где они находятся;

в) выбрать сведения о таких поставщиках, для которых не существует других по ставщиков, поставляющих меньше видов деталей;

г) выбрать сведения о поставщиках из Осло, которые поставляютTto крайней ме ре два разных вида деталей из Парижа и по крайней мере для двух разных про ектов в Стокгольме;

д) выбрать сведения о парах поставщиков из одного города, поставляющих пары *деталей из* одного я того же города;

е) выбрать сведения о парах поставщиков из одного города, поставляющих детали для пар проектов, находящихся в одном городе;

ж) выбрать сведения о деталях, поставляемых хотя бы для одного проекта, но только теми поставщиками, которые размещены не в том городе, где находится сам проект;

з) выбрать сведения о поставщиках наибольшего количества различных типов деталей.

Используйте приведенные выше ограничения целостности для преобразования данных запросов в более простую форму (но все еще на обычном языке — от вас не требуется выполнять это упражнение *неформальном языке).*

18.12. Исследуйте доступную вам СУБД. Какие преобразования выражений выполняет эта система? Осуществляет ли она какие-либо семантические преобразования?

18.13. Попробуйте провести следующий эксперимент. Возьмите простой запрос, например "Выбрать имена поставщиков детали с номером P2", и запишите его всеми известными вам способами с помощью любого доступного языка запросов (возможно, это будет язык SQL). Создайте и заполните данными подходящую тестовую базу данных, выполните все версии запроса и определите время выполнения каждой версии. Если полученные значения будут сильно различаться, значит, экспериментально доказано, что оптимизатор системы не полностью справляется с задачей преобразования выражений. Повторите этот эксперимент с различными исходными запросами. Если возможно, повторите этот же эксперимент в разных СУБД.

*Примечание. Б*езусловно, разные версии запроса должны давать идентичные результаты. Если же результаты различаются, то, возможно, вы совершили ошибку или ошибка существует в оптимизаторе рассматриваемой СУБД. Если обнаружена ошибка в оптимизаторе, сообщите об этом разработчику СУБД!

18.14. Исследуйте любую доступную вам СУБД. Какие статистические показатели базы данных поддерживает эта СУБД? Как обновляются эти статистические показатели — динамически или с помощью какой-либо утилиты? Если статистические показатели обновляются с помощью утилиты, то как называется эта утилита? Как часто она запускается? Насколько выборочно ее действие (предусмотрена ли возможность при запуске утилиты с определенными параметрами обновлять только определенную статистику)?

18.15. В разделе 18.5 было сказано, что в состав статистических показателей, поддерживаемых СУБД DB2, входят второе наибольшее и второе наименьшее значения каждого столбца в каждой базовой таблице. Как вы считаете, почему выбраны не максимальное и минимальное значения, а именно значения, стоящие на *втором месте за ними"!*

18.16. В некоторых коммерческих продуктах пользователю разрешается передавать оптимизатору так называемые *подсказки* (hint). Например, в СУБД DB2 спецификация OPTIMIZE FOR *n* ROWS в объявлении курсора SQL означает, что пользователь предполагает извлечь с помощью данного курсора не больше *n* строк (i.e. оператор FETCH будет выполнен для данного курсора не больше *n* раз). Такая спецификация иногда может позволить оптимизатору выбрать более эффективный путь доступа, по крайней мере, в случае, когда пользователь действительно выполняет

оператор FETCH не больше *n* раз. Оправдано ли, по вашему мнению, применение таких подсказок? Обоснуйте свой ответ.

18.17. Разработайте набор процедур реализации для операций сокращения и проекции (руководствуясь приведенными в разделе 18.7 псевдокодами процедур для опера ции соединения). Выведите соответствующий набор формул стоимости для этих процедур, предположив, что в данном случае нас интересует только количество операций ввода—вывода (т.е. не старайтесь учитывать в своих формулах степень загрузки процессора и других компонентов системы). Сформулируйте и докажите любые утверждения, принятые в процессе вывода этих формул.

18.18. Прочитайте приложение А и выскажите свое мнение по поводу прочитанного.

СПИСОК ЛИТЕРАТУРЫ

Оптимизация представляет собой чрезвычайно большую и постоянно развивающуюся область исследований. Приведенный ниже список представляет собой относительно небольшую выборку из обширной литературы по оптимизации и связанным с ней вопросам. Условно он разбит на описанные ниже группы.

- **В [18.1]—[18.6]** содержится обзор общих проблем оптимизации или введение в эту
- В [18.7]—[18.14] рассматриваются эффективные методы реализации отдельных ре ляционных операций, таких как соединение и агрегирование.
- В **[18.15]—[** 18.32] представлены различные приемы преобразования выражений, описанные в разделе 18.4. В частности, в [18.25]—[18.28] рассматриваются *семан тические* преобразования.
- **В [** 18.33]—**[18.43]** обсуждаются приемы, использованные в System R, СУБД DB2 и СУБД Ingres, а также общая проблема оптимизации вложенных запросов в стиле языка SQL.
- В [1**8.44]—[18.62]** содержится описание многочисленных методов, приемов и идей, подлежащих исследованию в будущем, и т.п. В частности, в **[18.55]—[18.58]** рас смотрено влияние параллельной обработки на вопросы оптимизации.

Примечание. Из данного списка умышленно исключены публикации, в которых рассматривается оптимизация в распределенных базах данных и системах поддержки принятия решений (см., соответственно, главы 21 и 21).

18.1. Kim W., Reiner D.S., Batory D.S. (eds.). Query Processing In Database Systems. — New York, N.Y.: Springer-Verlag, 1985.

Это антология работ, посвященных общим проблемам обработки запросов (не только работ по оптимизации). Книга состоит из вступительного обзора статей Джарке (Jarke), Коха (Koch) и Шмидта (Schmidt), который перекликается с публикацией [18.2], но не повторяет ее. Далее следует ряд работ, описывающих обработку запросов в различных контекстах: распределенные базы данных, гетерогенные СУБД, проблемы обновления представлений (работа [10.8] является единственной статьей в этом разделе), нетрадиционные приложения (например CAD/CAM), оптимизация многооператорных запросов [18.47], машины баз данных и физическое проектирование базы данных.

18.2. Jarke M., Koch J. Query Optimization in Database Systems // ACM Comp. Surv. — June 1984. - 16, № 2.

Отличное учебное пособие. Представляет обобщенную систему взглядов по проблеме вычисления запросов, подобную приведенной в разделе 18.3 данной главы, но базирующуюся на реляционном исчислении, а не на алгебре. После этого в рамках описанной системы взглядов обсуждается множество методов оптимизации: синтаксические и семантические преобразования, реализация низкоуровневых операций и алгоритмы генерации планов запроса и выбора плана с наименьшей стоимостью. Приводится также обширный набор правил синтаксических преобразований для выражений исчисления. Имеется достаточно большой список литературы (без аннотаций). Но отметим, что после 1984 года по данной теме было опубликовано на порядок больше работ, чем до 1984 года.

В работе также кратко обсуждаются другие смежные проблемы, такие как оптимизация высокоуровневых языков запросов (т.е. языков, более мощных по сравнению с реляционной алгеброй или исчислением) и оптимизация в среде распределенных баз данных, а также роль машин баз данных с точки зрения оптимизации.

18.3. Graefe G. Query Evaluation Techniques for Large Databases // ACM Comp. Surv. — June 1993.-25, №2.

Еще одно прекрасное и более новое учебное пособие (по сравнению с [18.2]), с обширным списком рекомендуемой литературы. Приведем цитату из резюме: "В этом обзоре представлены основы проектирования и реализации инструментов выполнения запросов... В нем описан широкий круг практических методов оценки запросов ... включая итеративное выполнение сложных планов оценки запросов, показаны аналогии между алгоритмами сопоставления множеств на основе сортировки и хэширования, определены типы параллельного выполнения запросов и их реализации, а также перечислены специальные операторы для новых областей применения баз данных". Рекомендуется прочесть эту работу.

18.4. Palermo F.P. A Data Base Search Problem // J.T. Todd (ed.), Information Systems: COINS IV. - New York, NY: Plenum Press, 1974.

Одна из наиболее ранних работ по оптимизации (фактически уже ставшая классической). В этой работе берется произвольное выражение реляционного исчисления, к нему применяется алгоритм приведения Кодда для преобразования выражения в эквивалентное алгебраическое выражение (см. главу 8), а затем вводится несколько усовершенствований этого алгоритма, включая перечисленные ниже.

- Никакой кортеж не должен извлекаться дважды.
- Ненужные значения удаляются из кортежа во время его извлечения. Здесь под ненужными значениями подразумеваются либо значения атрибутов, не упомя нутых в запросе, либо значения, используемые только для организации выбор ки. Этот процесс эквивалентен проекции отношения по некоторым *нужным* атрибутам, поэтому позволяет сократить не только пространство, выделяемое для каждого кортежа, но и количество кортежей, с которыми ведется работа (в общем случае).

- Метод, используемый для создания результирующего отношения, основан на принципе наименьшего роста, поэтому оно растет очень медленно. Данный метод приводит к сокращению количества выполняемых сравнений и объема памяти для хранения промежуточных результатов.
- Для создания соединений применяется эффективный метод, основанный, во-первых, на динамическом разложении значений, используемых в соединяемых термах (например, таких как S. s# = SP. s#), на *полусоединения,* которые фактически являются некоторой разновидностью динамически создаваемого вторичного индекса (полусоединения Палермо отличаются от полусоединений, описанных в главе 7), и, во-вторых, на использовании внутреннего представления каждого соединения, называемого *косвенным соединением* (indirect join), в котором для идентификации кортежей, участвующих в соединении, применяются внутренние идентификаторы кортежей. Эти методы предназначены для сокращения количества просмотров, необходимых при формировании соединения, за счет обеспечения того, чтобы соединяемые кортежи каждого члена соединения были логически упорядочены в соответствии со значениями атрибутов соединения. Благодаря этому допускается динамическое определение *наилучшей* последовательности операций доступа к нужным отношениям.

18.5. Poess M., Floyd C. New TPC Benchmarks for Decision Support and Web Commerce // ACM SIGMOD. — December 2000. — Record 29, No. 4.

Сокращение TPC обозначает Совет по обработке транзакций (Transaction Processing Council). Это независимая организация, которая в течение многих лет разрабатывала эталонные тесты, ставших для отрасли обработки данных стандартными. Тест TPC-C (который составлен на основе модели системы ввода заказов) является эталонным тестом для измерения производительности систем OLTP. Эталонные тесты TPC-H и TPC-R предназначены для систем поддержки принятия решений; они были разработаны, соответственно, для измерения производительности произвольных запросов (TPC-H) и плановых отчетов (TPC-R). Эталонный тест TPC-W предназначен для измерения производительности приложений, применяемых в среде электронной коммерции. Дополнительную информацию, включая многочисленные результаты фактического применения эталонных тестов, можно найти по адресуhttp: / /www. tpc . org.

18.6. Bitton D., DeWitt D.J., Turbyfill C. Benchmarking Database Systems: A Systematic Approach // Proc. 9th Int. Conf. on Very Large Data Bases. — Florence, Italy. — October-November 1983.

Здесь впервые описано то, что сегодня называют "Висконсинским эталонным тестом" (поскольку этот тест был разработан авторами данной работы в университете штата Висконсин). В описываемом тесте используется набор отношений с точно указанными значениями атрибутов, по которым выполняются проекции. При его выполнении определяется производительность конкретно заданных алгебраических операций на указанных отношениях (например, различных видов проекций, использующих атрибуты с разной степенью дублирования значений). Таким образом, измеряется эффективность работы оптимизатора системы на описанных выше основных операциях.

18.7. Blasgen M.W., Eswaran K.P. Storage and Access in Relational Databases// IBM Sys. J. — 1977.-16, №4.

Представлены результаты сравнения различных методов обработки запросов, в которых используются операции сокращения, проекции и соединения, на основе их стоимости, выраженной как количество операций ввода-вывода. Основные компоненты этих методов реализованы в системе System R [18.33].

18.8. Merrett T.H. Why Sort/Merge Gives the Best Implementation of the Natural Join // ACM SIGMOD Record. - January 1983. - 13, № 2.

Представлен набор интуитивно понятных доводов в пользу утверждения, что с помощью сортировки-слияния можно достичь самых эффективных результатов. В основном, эти доводы сводятся к следующему:

а) операция соединения будет наиболее эффективной, если оба отношения будут отсортированы по значениям атрибута, по которому выполняется соединение (поскольку в данном случае, как показано выше, в разделе 18.7, слияние — дос таточно эффективный метод оптимизации, а выборка каждой страницы вы полняется только один раз, что действительно является оптимальным);

б) стоимость сортировки отношений в необходимом порядке (на достаточно мощных компьютерах) будет, вероятно, меньше стоимости любой схемы, в ко торой не требуется упорядоченность кортежей в обоих отношениях.

Тем не менее, автор допускает, что его позиция весьма радикальна и могут существовать исключения. В частности, одно из отношений может быть настолько мало (например, в случае, когда оно является результатом предыдущей операции сокращения), что прямой доступ ко второму отношению с помощью индекса или хэш-функции будет иметь меньшую стоимость, чем сортировка второго отношения. **В [18.9]—[18.1**1] приведены дополнительные примеры, в которых метод сортировки-слияния на практике является не лучшим методом обработки.

18.9. Sacco G.M. Fragmentation: A Technique for Efficient Query Processing // ACM TODS. - June 1986. - 11, № 2.

Представлен один из методов, использующих принцип "разделяй и властвуй" для выполнения операции соединения. Согласно этому методу, соединяемые отношения рекурсивно разбиваются на непересекающиеся подмножества кортежей (фрагменты) и в этих фрагментах выполняется ряд последовательных просмотров. В отличие от метода сортировки—слияния, этот метод не требует предварительной сортировки отношений. В работе показано, что фрагментация всегда работает эффективнее сортировки-слияния, когда последний метод требует предварительной сортировки обоих отношений, и иногда работает лучше, даже если метод сортировки-слияния требует предварительной сортировки только одного (большего) отношения. Автор утверждает, что данный метод можно применяться и для других операций, таких как пересечение и разность.

18.10. Shapiro L.D. Join Processing in Database Systems with Large Main Memories // ACM TODS. - September 1986. - 11, № 3.

Представлено три новых алгоритма хэширования при выполнении операции соединения. Один из них "особенно эффективен при наличии оперативной памяти с размером, значительно превышающим размер одного из соединяемых отношений".

Алгоритмы работают благодаря разбиению отношений на непересекающиеся части (т.е. выборки), которые можно обрабатывать в оперативной памяти. Автор заявляет, что, исходя из наблюдаемого в наши дни снижения цен на первичную память, методы на основе хэширования могут стать одними из лучших.

18.11. Negri M., Pelagatti G. Distributive Join: A New Algorithm for Joining Relations // ACM TODS.-December 1991.- 16, №4.

Здесь представлен другой метод соединения, использующий принцип "разделяй и властвуй", который по словам авторов "... базируется на идее о том, что... не нужно полностью сортировать оба отношения... Достаточно отсортировать одно отношение полностью, а другое — лишь частично, избежав таким образом расходов на выполнение полной сортировки второго отношения". При частичной сортировке рассматриваемая переменная отношения разбивается на последовательность неотсортированных подмножеств кортежей PI, P2, ..., Pn (по аналогии с методом Сакко [18.9], за исключением того, что в методе Сакко вместо сортировки используется хэширование), обладающих следующим свойством: MAX(P[i]) < MIN(P [i + 1]) для всех i (I, 2, ..., n-1). Утверждается, что данный метод работает эффективнее метода сортировки—слияния.

18.12. Graefe C, Cole R.L. Fast Algorithms for Universal Quantification in Large Databases // ACM TODS. - June 1995. - 20, № 2.

Квантор всеобщности (FORALL) в языке SQL непосредственно не поддерживается, поэтому он не поддерживается и в современных коммерческих СУБД, хотя чрезвычайно важен для формулировки широкого класса запросов. В этой статье описываются и сравниваются "три известных алгоритма и один недавно предложенный алгоритм реляционного деления, который является алгебраическим оператором, включающим квантор всеобщности". В ней также показано, что новый алгоритм работает "с той же скоростью, с какой выполнение хэшированного полусоединения позволяет получить оценку значения квантора существования для тех же отношений" (эти цитаты немного переформулированы). Помимо всего прочего, авторы делают вывод, что квантор FORALL необходимо включить в пользовательский язык, поскольку большинство оптимизаторов "не распознает его косвенные формулировки на языке SQL".

18.13. Simmen D., Shekita E., Malkemus T. Fundamental Techniques for Order Optimization // Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data.— Montreal, Canada. —June 1996.

В этой работе представлены методы оптимизации или исключения сортировок. Они частично основаны на работе Дарвена [11.7] и именно в таком виде реализованы в СУБД DB2.

18.14. Manku G.S., Rajagopalan S., Lindsay B.G. Approximate Medians and Other Quantities in One Pass and with Limited Memory // Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data. — Seattle, Wash. — June 1998.

18.15. Galindo-Legaria C.A., Joshi M.M. Orthogonal Optimization of Subqueries and Aggregation // Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. — May 2001.

18.16. Smith J.M., Yen-Tang Chang P. Optimizing the Performance of a Relational Algebra Database Interface // CACM. — October 1975. — 18, № 10.

Описан алгоритм оптимизации, использовавшийся в языке SQUIRAL (Smart Query Interface for a Relational Algebra). Этот алгоритм предусматривает перечисленные ниже действия.

- Преобразование исходного алгебраического выражения в эквивалентную, но более эффективную последовательность операций в соответствии с принципа ми, обсуждавшимися выше, в разделе 18.4.
- Привязка отдельных операций в преобразованном выражении к определенным процессам и использование параллельной и конвейерной обработки для вы полнения операций.
- Координация порядка сортировки в промежуточных отношениях, передавае мых между процессами.
- Использование индексов и попытки локализации страничных ссылок.

Эта статья и работа [18.17] были, вероятно, одними из первых работ, посвященных преобразованиям выражений.

. Hall P.A.V. Optimization of a Single Relational Expression in a Relational Data Base System // IBM J. R&D. - May 1976. - 20, № 3.

Описаны некоторые методы оптимизации, используемые в системе PRTV [7.9], которая, как и язык SQUIRAL [18.16], начинает обработку запроса с преобразования данного алгебраического выражения в некоторую более эффективную форму. Особенностью системы PRTV является то, что она не вычисляет каждое выражение автоматически, как только оно получено. Вместо этого, выражения комбинируются с полученными ранее в более сложное выражение и вычисление откладывается до последнего момента (см. обсуждение пошаговой формулировки запроса в главе 7, раздел 7.5). Таким образом, "единственное реляционное выражение" (о котором говорится в заголовке статьи) на самом деле представляет целую последовательность операций пользователя. Данные методы оптимизации подобны методу оптимизации языка SQUIRAL, но превосходят его по некоторым характеристикам. Эти методы оптимизации включают перечисленные ниже (в порядке их применения).

- Операции сокращения выполняются настолько рано, насколько это возможно.
- Последовательность проекций комбинируется в одну операцию проекции.
- Избыточные операции исключаются.
- Выражения, использующие пустые отношения и тривиальные условия, упро щаются.
- Общие выражения выносятся за скобки.

В заключении работы изложены результаты некоторых экспериментов, а также указаны направления дальнейших исследований.

18.18. Jarke M, Koch J. Range Nesting: A Fast Method to Evaluate Quantified Queries // Proc. 1983 ACM SIGMOD Int. Conf. on Management of Data. — San Jose, Calif. — May 1983.

Предложен вариант реляционного исчисления, в котором допускается применение некоторых дополнительных (и полезных) правил синтаксического преобразования. Представлены алгоритмы вычисления выражений в описанном исчислении. (Это исчисление фактически очень напоминает исчисление кортежей, рассматриваемое в главе 8.) Кроме того, описана оптимизация в предложенном исчислении отдельного класса выражений, называемых *идеальными вложенными выражениями.* Описаны методы преобразования в идеальные выражения довольно сложных запросов (в частности запросов, использующих квантор FORALL). Авторы показывают, что в идеальные выражения можно преобразовать большую часть запросов, используемых на практике.

18.19. Chaudhuri S., Shim K. Including Group-By in Query Optimization // Proc. 20th Int. Conf. on Very Large Data Bases. — Santiago, Chile. — September 1994."

18.20. Makinouchi A., Tezuka M., Kitakami H., Adachi S. The Optimization Strategy for Query Evaluation in RDB/V1 // Proc. 7th Int. Conf. on Very Large Data Bases — Cannes, France. — September 1981/

Система RDB/V1 является прототипом будущего программного продукта AIM/RDB компании Fujitsu. В публикации описаны методы оптимизации, используемые в данном прототипе. Они кратко сравниваются с аналогичными методами, применяемыми в прототипах систем Ingres и System R. Один из методов является новым и состоит в использовании динамически полученных значений MAX и MIN для выполнения дополнительных выборок. Подобный метод приводит к упрощению процесса выбора порядка соединения и повышает производительность самой операции соединения. В качестве простого примера для последнего утверждения предположим, что нужно соединить отношения поставщиков S и деталей P по атрибуту CITY. Сначала отношение s сортируется по атрибуту CITY. В процессе сортировки для атрибута S.CITY определяются максимальное и минимальное значения, скажем, HIGH и LOW. Тогда для уменьшения количества кортежей отношения P, которые потребуется проверять в процессе выполнения соединения, можно использовать следующую выборку.

```
LOW < P.CITY AND P.CITY < HIGH
```

18.21. Pirahesh H., Hellerstein J.M., Hasan W. Extensible Rule Based Query Rewrite Optimization in Starburst // Proc. 1992 ACM SIGMOD Int. Conf. on Management of Data. — San Diego, Calif. — June 1992.

Как отмечалось в разделе 18.1, для метода преобразования выражений используется еще одно название — *перезапись запросов* (query rewrite). По мнению авторов, несколько неожиданно то, что в современных реляционных СУБД такому преобразованию уделяется мало внимания (по крайней мере, по состоянию на 1992 год). В работе описан механизм преобразования выражений прототипа системы Starburst корпорации IBM [18.48], [26.19], [26.23], а также [26.29] и [26.30]. Пользователи с соответствующей квалификацией могут в любое время добавить в систему новые правила преобразования выражений (вот почему в заголовке статьи применяется слово *расширяемый* — extensible).

18.22. Mumick I.S., Finkelstein S.J., Pirahesh H., Ramakrishnan R. Magic is Relevant // Proc. 1990 ACM SIGMOD Int. Conf. on Management Data. — Atlantic City, N.J. — May 1990.

Неудачный термин "magic" ("магический" — под этим подразумевается "не обусловленный синтаксической структурой") используется в данной публикации для ссылок на метод оптимизации, который изначально создавался для обработки запросов (в частности, рекурсивных), записанных на языке логических баз данных Datalog (глава 24). Данная работа распространяет этот подход на условные реляционные системы; в ней утверждается на основе экспериментальных измерений, что он часто эффективнее традиционных методов оптимизации (отметим, что запрос не обязательно должен быть рекурсивным, чтобы можно было применить к нему описанный метод). Основная идея метода состоит в декомпозиции данного запроса на несколько меньших запросов, которые определяют множество *вспомогательных отношений* (нечто похожее на метод декомпозиции запросов, рассмотренный в разделе 18.6) таким образом, чтобы их можно было использовать для фильтрации кортежей, не относящихся к данному запросу. Ниже приведен пример запроса, который базируется на примере из данной работы (выраженный с помощью реляционного исчисления).

```
{ EX.ENAME }
    WHERE EX.JOB = 'Clerk' AND
        EX.SAL > AVG ( EY WHERE EY.DEPT# = EX.DEPT#, SAL )
```

("Выбрать имена клерков ('Clerk'), чья зарплата больше средней зарплаты по отделу".) Если этот запрос выполнять *прямолинейно* (т.е. именно так, как он записан), система будет проверять отношение, содержащее данные о сотрудниках, кортеж за кортежем, и поэтому вычислять среднюю зарплату для любого отдела, в котором работает больше одного клерка, несколько раз. Традиционный оптимизатор разобьет исходный запрос на два следующих, меньших запроса.

```
WITH { EX.DEPT#,
       AVG ( EY WHERE EY.DEPT# =
             EX.DEPT#, SAL ) AS ASAL } AS T1 :

{ EMP.ENAME } WHERE EMP.JOB = 'Clerk' AND    '..
                    EXISTS T1 ( EMP.DEPT# = T1. t>EPT#
                                AND EMP.SALARY > T1.ASAL
                                )
```

Теперь ни для одного отдела средняя зарплата не будет вычисляться больше одного раза, но будут вычисляться некоторые не относящиеся к запросу средние значения, а именно для тех отделов, в которых не работают клерки. Предложенный в этой работе "магический" подход исключает оба нежелательных варианта, повторное вычисление средней зарплаты и обработку не относящихся к запросу кортежей, за счет формирования *вспомогательных* отношений, как показано ниже.

```
/* Первое вспомогательное отношение : данные об именах, отделах
*/
/* и зарплатах клерков                                          */
WITH ( { EMP.ENAME, EMP.DEPT*, EMP.SAL }
        WHERE EMP.JOB = 'Clerk' ) AS T1 : /* Второе
вспомогательное отношение : данные об отделах, в которых */
```

```
/* работают клерки
WITH { T1.DEPT# } AS T2 :

/* Третье вспомогательное отношение : данные об отделах, в
которых */
/*работают клерки и соответствующих средних зарплатах        */
WITH ( { T2.DEPT#,
         AVG ( EMP WHERE
               EMP.DEPT# = T2.DEPT#, SAL ) AS ASAL } ) AS T3 :

/* Результирующее отношение */
{ T1.ENAME } WHERE EXISTS T3 ( T1.DEPT# = T3.DEPT# AND
                               T1.SAL > T3.ASAL )
```

"Магический" подход состоит в точном определении того, какие именно вспомогательные отношения требуется создать.

(См. [18.23], [18.24], а также список литературы в главе 24, где имеются другие упоминания о "магическом" подходе.)

18.23. Mumick I.S., Pirahesh H. Implementation of Magic in Starburst // Proc. 1994 ACM SIGMOD Int. Conf. on Management of Data. — Minneapolis, Minn. — May 1994.

18.24. Mumick I.S., Finkelstein S.J., Pirahesh H., Ramakrishnan R. Magic Conditions//ACM TODS. - March 1996. - 21, № 1.

18.25. King J.J. QUIST: A System for Semantic Query Optimization in Relational Databases // Proc. 7th Int. Conf. on Very Large Data Bases. — Cannes, France. — September 1981.

Представлена идея семантической оптимизации (см. раздел 18.4). В статье описана экспериментальная система QUIST (QUery Improvement through Semantic Transformation — усовершенствование запросов с помощью семантических преобразований), способная выполнять подобные преобразования.

18.26. Shenoy ST., Ozsoyoglu Z.M. A System for Semantic Query Optimization // Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data.— San Francisco, Calif — May/June 1987.

Эта публикация дополняет работу Кинга (King) [18.25], представляя схему, которая динамически выбирает из потенциально очень большого множества ограничений целостности только те ограничения, которые, по-видимому, будут применимы для обработки данного запроса. Рассматриваемые ограничения целостности разделены на два типа: *ограничения логического вывода* и *ограничения подмножеств*. Подобные ограничения используются для преобразования запросов посредством исключения избыточных выборок и соединений и введения дополнительных выборок для индексированных атрибутов. Случаи, когда результат запроса может быть получен непосредственно из ограничений целостности, также эффективно обрабатываются с помощью изложенного метода.

18.27. Siegel M., Sciore E., Salveter S. A Method for Automatic Rule Derivation to Support Semantic Query Optimization // ACM TODS. — December 1992. — 17, № 4.

Как было показано в этой главе, в разделе 18.4, в методах семантической оптимизации для преобразования запросов используются ограничения целостности. Тем не менее, существует несколько описанных ниже проблем, связанных с этой идеей.

- Как оптимизатор может определить, какое преобразование будет эффектив ным (т.е. какое преобразование сделает запрос более эффективным)?
- Некоторые ограничения целостности не очень полезны для решения задач оп тимизации. Например, ограничение, которое требует, чтобы вес детали был больше нуля, важно для целостности базы данных, но по сути бесполезно для целей оптимизации. Как оптимизатор сможет отличить полезные ограничения целостности от бесполезных?
- Некоторые утверждения могут быть справедливы для определенных состояний базы данных (даже для большинства состояний) и поэтому будут полезными для целей оптимизации, но все же, строго говоря, их нельзя рассматривать как ограничения целостности. Примером может служить утверждение вида EMP. AGE < 50 (возраст сотрудников — не больше 50 лет), которое не является ограничением целостности как таковым (поскольку могут быть сотрудники и старше 50 лет), но в данный момент в компании может действительно не быть сотрудников старше 50 лет.

В этой публикации описана архитектура системы, в которой учтены перечисленные проблемы.

18.28. Chakravarthy U.S., Grant J., Minker J. Logic-Based Approach to Semantic Query Optimization // ACM TODS. — June 1990. — 15, № 2.

Цитата из резюме: "В нескольких предыдущих работах авторами была описана и доказана правильность метода семантической оптимизации запросов... В этой работе обобщаются основные результаты из предыдущих работ и особое внимание обращено на методы и их применимость к оптимизации реляционных запросов. Дополнительно в данной работе показано, каким образом описываемый метод подводит итоги и обобщает результаты более ранних работ в области семантической оптимизации. (Кроме того, отмечается], как методы семантической оптимизации запросов могут быть распространены на рекурсивные запросы и ограничения целостности, содержащие дизъюнкции, отрицания и рекурсию".

18.29. Cheng Q. et al. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database // Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland. — September 1999.

18.30. Aho A.V., Sagiv Y., Ullman J.D. Efficient Optimization of a Class of Relational Expressions // ACM TODS. - December 1979. - 4, № 4.

Класс реляционных выражений, упоминаемый в заголовке этой работы, содержит выражения, использующие только операции сокращения по условию равенства (которые в этой статье называются *выборками),* проекции и естественного соединения. Этот класс выражений иногда называют *SPJ-выражениями* (от англ. "selection", "projection", "join"— *выборка, проекция, соединение).* SPJ-выражения соответствуют запросам в реляционном исчислении, в которых логические выражения <bool exp> в конструкции WHERE содержат только операторы сравнения на равенство, операторы AND и кванторы EXISTS. В работе предложена новая конструкция, получившая название табло (tableau), которая представляет собой удобное средство символического представления *SPJ-выражений.* Табло — это двухмерный массив, в котором столбцы соответствуют атрибутам, а строки — условиям, в

частности *условиям принадлежности,* которые гласят, что конкретный кортеж (субкортеж) принадлежит конкретному отношению. Строки табло логически связываются посредством размещения общих символов в соответствующих строках. Например, приведенное ниже табло представляет запрос "Выбрать статус (f 1) поставщиков (Ы) из Лондона, которые поставляют некоторые детали красного цвета (b2)".

S#	STATUS	CITY	P#	COLOR	
	f1				
b1	f1	'London'			- S (поставщики)
b1			b2		- SP (поставки)
			b2	'Red'	- P (детали)

Верхняя строка в табло представляет все атрибуты, которые используются в запросе, следующая строка — это *итоги* (она соответствует кортежу-прототипу в запросе, определенном в реляционном исчислении, или заключительной проекции в алгебраическом запросе), а оставшиеся строки (как уже говорилось) представляют условия принадлежности. В данном примере эти строки прокомментированы посредством указания относящихся к запросу отношений (точнее, переменных отношения). Обратите внимание на то, что b используется для ссылок на связанные переменные, а f — для ссылки на свободные переменные. Итоговая строка содержит только переменные типа f.

Табло — это еще один вариант канонической формулировки для представления запросов (см. раздел 18.3), однако оно не является достаточно общим, чтобы представить все возможные реляционные выражения. (Фактически табло можно рассматривать как синтаксическую разновидность языка Query-By-Example (QBE), хотя и строго менее мощную, чем сам язык QBE.) В работе представлены алгоритмы преобразования одного табло в другое, семантически эквивалентное табло, в котором количество строк уменьшено до минимума. Поскольку количество строк (не считая двух верхних специальных строк) на единицу больше количества соединений в соответствующем SPJ-выражении, полученное табло представляет оптимальную форму запроса, хотя и в очень специальном смысле (минимальное количество соединений). (В приведенном выше примере количество соединений уже минимально, поэтому подобная оптимизация не дает никакого эффекта.) После этого полученное минимальное табло можно преобразовать обратно в другое представление для последующей дополнительной оптимизации.

Идея минимизации количества соединений также применима к запросам, сформулированным в терминах представлений, которые сформированы на основе операции соединения, в частности, к запросам, сформулированным в терминах *универсальных отношений* (см. список литературы в конце главы 13). Например, предположим, что пользователю предложено представление V, определенное как соединение отношений S (поставщиков) и SP (поставок) по атрибуту s#, и пользователь вводит следующий запрос.

v { p# }

Прямой алгоритм обработки представлений преобразует примитивный запрос в представленный ниже.

```
( SP JOIN S ) { P# }
```

Тем не менее, как было показано в разделе 18.4, приведенный ниже запрос дает тот же результат, хотя он вовсе не использует операции соединения (т.е. количество соединений в исходном запросе минимизировано).

```
SP { P# }
```

Поэтому следует отметить, что поскольку алгоритмы сжатия табло, описанные в данной работе, учитывают все явно выраженные функциональные зависимости между атрибутами, они являются примером сокращенного метода *семантической* оптимизации.

18.31. Sagiv Y., Yannakakis M. Equivalences Among Relational Expressions with the Union and Difference Operators // JACM. — October 1980. — 27, № 4.

В этой статье идеи работы [18.30] дополнены таким образом, чтобы их можно было применять к запросам, использующим операции объединения и разности.

18.32. Levy A.Y., Mumick I.S., Sagiv Y. Query Optimization by Predicate Move-Around // Proc. 20th Int. Conf. on Very Large Data Bases. — Santiago, Chile — September 1994.

18.33. Selinger P.G. et al. Access Path Selection in a Relational Database System // Proc. 1979 ACM SIGMOD Int. Conf. on Management of Data. — Boston, Mass. — May/June 1979.

В этой важной статье обсуждаются некоторые методы оптимизации, используемые в системе System R. В системе System R запрос представляет собой выражение SQL, поэтому состоит из набора блоков SELECT-FROM-WHERE *(блоков запроса).* Одни блоки могут быть вложены в другие. Оптимизатор System R сначала определяет порядок, в котором будут вычисляться блоки запроса, а затем пытается свести к минимуму общую стоимость запроса посредством выбора реализации с наименьшей стоимостью для каждого отдельного блока запроса. Обратите внимание на то, что данная стратегия (сначала — выбор порядка блоков, а затем — оптимизация отдельных блоков) означает, что конкретный план выполнения всего запроса никогда не рассматривается, и это приводит к *сужению пространства поиска* (см. замечания по этому поводу почти в самом конце раздела 18.3).

Примечание. В случае вложенных блоков оптимизатор вычисляет блоки в порядке их вложения, который указал пользователь, т.е., грубо говоря, внутренний блок выполняется первым. **В [18.37]—[** 18.43] можно найти критические замечания и дальнейшее обсуждение данной стратегии.

Для каждого блока запроса существует, по сути, два описанных ниже варианта, которые следует проанализировать (первый из них можно рассматривать как частный случай второго).

1. Для блока, в котором используется только выборка и (или) проекция единственного отношения, оптимизатор применяет статистические данные из каталога совместно с формулами (представленными в этой публикации) для немедленной оценки размера результата и стоимости низкоуровневых операций, а также для выбора стратегии формирования выборки и (или) проекции.

2. Для блоков, в которых используются два или больше отношений, соединённых с помощью операции соединения, возможно, с локальными выборками и (или) проекциями отдельных отношений, оптимизатор, во-первых, рассматривает каждое отдельное отношение, как в варианте 1, и во-вторых, определяет последовательность выполнения соединений. Эти две операции не являются независимыми одна от другой. Итак, для отдельного отношения A может быть выбрана определенная стратегия доступа (например, на основе некоторого индекса), поскольку она позволяет получать кортежи из A в том порядке, который удобен для выполнения последующего соединения отношения A с другим отношением в.

Соединения реализуются с помощью метода сортировки—слияния, поиска по индексу или метода последовательного просмотра. В работе особое значение автор придал тому, что при вычислении, например, вложенного соединения (A JOIN B) JOIN C не обязательно полностью вычислять вложенное соединение A JOIN B перед соединением его результата и отношения C. Наоборот, каждый кортеж соединения A JOIN в сразу после вычисления передается процессу, соединяющему этот кортеж с кортежами из отношения C. Поэтому может никогда не потребоваться материализация отношения A JOIN B полностью. (Эта идея *конвейерной* обработки кратко обсуждалась в главе 3, раздел 3.2. См. также [18.16] и [18.58].)

Эта работа включает несколько замечаний о стоимости самого процесса оптимизации. Для соединения двух отношений стоимость приблизительно равна стоимости 5—20 операций выборки. При многократном выполнении оптимизированного запроса это довольно незначительные расходы. (Отметим, что система System R является компилирующей, поэтому однажды оптимизированный запрос может выполняться сотни и даже тысячи раз. В действительности, подход, основанный на использовании компиляции, был впервые реализован именно в этой системе.) В работе утверждается, что оптимизация сложных запросов требует "всего нескольких тысяч байтов пространства для хранения и нескольких десятых долей секунды" на компьютере IBM System 370 Model 158. "Соединение восьми таблиц было оптимизировано за несколько секунд".

18.34. Wong E., Youssefi K. Decomposition — A Strategy for Query Processing // ACM TODS. - September 1976. - 1, № 3.

18.35. Youssefi K., Wong E. Query Processing in a Relational Database Management System // Proc. 5th Int. Conf. on Very Large Data Bases. — Rio De Janeiro, Brazil. — September 1979.

18.36. Rowe L.A., Stonebraker M. The Commercial Ingres Epilogue // [8.10].

Коммерческая СУБД Commercial Ingres — это программный продукт, созданный на основе прототипа University Ingres. Ниже перечислены некоторые различия между оптимизаторами систем Commercial Ingres и University Ingres.

1. Оптимизатор University использует инкрементное планирование, т.е. определяет, что нужно сделать сначала, и делает это, затем определяет, что делать дальше, исходя из размера полученного промежуточного результата, и т.д.Оптимизатор Commercial определяет полный план перед началом выполнения на основе оценок размеров промежуточных результатов.

2. Оптимизатор University обрабатывает запросы с двумя переменными (например, соединения) с помощью метода подстановки кортежей, обсуждавшегося в разделе 18.6. Оптимизатор Commercial поддерживает несколько мощных методов обработки подобных запросов, включая, в частности, метод сортировки-слияния, описанный в разделе 18.7.
3. Оптимизатор Commercial поддерживает более сложный набор статистических показателей по сравнению с оптимизатором University.
4. Оптимизатор University осуществляет инкрементное планирование (см. п. 1). Оптимизатор Commercial выполняет более обширный поиск. Тем не менее, поиск прекращается, если на оптимизацию будет затрачено время, превышающее наилучшую оценку времени выполнения запроса (в противном случае выполнение оптимизации не даст никаких преимуществ).
5. Оптимизатор Commercial рассматривает все возможные комбинации индексов, все возможные последовательности соединения и "все доступные методы выполнения соединения — сортировку—слияние, частичную сортировку—слияние, поиск с помощью хэш-функции, поиск с помощью метода доступа ISAM, поиск по B-дереву и метод последовательного просмотра" (см. раздел 18.7).

18.37. Kim W. On Optimizing an SQL-Like Nested Query // ACM TODS. — September 1982.-7, №3.

См. аннотацию к [18.41].

18.38. Kiessling W. On Semantic Reefs and Efficient Processing of Correlation Queries with Aggregates // Proc. 1 lth Int. Conf. on Very Large Data Bases. — Stockholm, Sweden. — August 1985.

См. аннотацию к [18.41].

18.39. Ganski R.A., Wong H.K.T. Optimization of Nested SQL Queries Revisited // Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data. — San Francisko, Calif. — May 1987.

См. аннотацию к [18.41].

18.40. von Biiltzingsloewen G. Translating and Optimizing SQL Queries Having Aggregates // Proc. 13th Int. Conf. on Very Large Data Bases.— Brighton, England.— September 1987.

См. аннотацию к [18.41].

18.41. Muralikrishna M. Improved Unnesting Algorithms for Join Aggregate SQL Queries // Proc. 18th Int. Conf. on Very Large Data Bases. — Vancouver, Canada. — August 1992.

Язык SQL позволяет создавать *вложенные подзапросы,* т.е. блоки SELECT-FROM-WHERE, вложенные внутрь других подобных блоков (см. главу 8). Эта конструкция порождает некоторые трудности при реализации. Рассмотрим следующий запрос SQL ("Определить имена поставщиков детали с номером P2"), на который далее будем ссылаться, как на запрос Q1.

```
SELECT S.SNAME
FROM S
```

```
WHERE S.S# IN
          ( SELECT SP.S# FROM SP WHERE
            SP.P# = P# ('P2') ) ;
```

В системе System R [18.33] этот запрос будет реализован следующим образом. Сначала будет вычислен внутренний блок и создана временная таблица т, содержащая *номера* требуемых поставщиков. После этого кортеж за кортежем будет проверена вся таблица S и для каждого выбранного из нее кортежа будет просматриваться таблица т в целях определения, содержится ли в ней соответствующий номер поставщика. Эта стратегия достаточно неэффективна (поскольку таблица т не имеет индекса).

Теперь рассмотрим следующий запрос Q2.

```
SELECT S.SNAME
FROM S, SP
WHERE S.S# = SP.S#
AND SP.P# = P# ('P2') ;
```

Легко установить, что он семантически идентичен предыдущему, но в системе System R предусмотрена дополнительная стратегия реализации данного запроса. В частности, если таблицы S и SP окажутся физически сохраняемыми в последовательности номеров поставщиков, то система осуществит соединение с помощью слияния, которое будет выполняться весьма эффективно. Предположив, что два рассмотренных выше запроса логически эквивалентны, но второй вариант более удобен с точки зрения эффективности реализации, мы придем к выводу, что возможность преобразования запросов, подобных Q1, в запросы, подобные Q2, заслуживает более глубокого изучения. Эта возможность и является предметом обсуждения в [18.37]—[18.43].

Ким (Kim) [18.37] был первым, кто обратил внимание на эту проблему. В его работе идентифицировано пять типов вложенных запросов и описаны соответствующие алгоритмы преобразования. В ней приводятся результаты экспериментов, которые показывают, что предложенные алгоритмы повышают производительность обработки вложенных запросов (обычно на один—два порядка).

Затем Кисслинг (Kiessling) [18.38] показал, что алгоритмы Кима работают неправильно, если в списке SELECT атрибутов выборки вложенного запроса (на любом уровне) содержится оператор COUNT (алгоритм Кима неправильно обрабатывает случаи, когда фактический параметр оператора COUNT является пустым множеством). Термин "семантические рифы" в названии работы Кисслинга указывает на сложности, с которыми сталкивается пользователь, работающий на языке SQL, при попытке получить правильные и непротиворечивые результаты подобных запросов. Более того, Кисслинг показал, что алгоритмы Кима усовершенствовать непросто (поскольку "похоже, не существует единого способа выполнить данное преобразование эффективно и правильно при любых условиях").

В работе Гански (Ganski) и Вонга (Wong) [18.39] описано решение проблемы, обнаруженной Кисслингом. Оно состоит в использовании в преобразованной версии запроса *внешнего соединения* (глава 19) вместо обычного внутреннего соединения.

(По мнению автора данной книги, это усовершенствование не вполне удовлетворительно, поскольку оно вводит нежелательную зависимость от упорядочения среди операторов в преобразованном запросе.) В данной работе указана еще одна ошибка в оригинальной работе Кима, которая устраняется аналогичным способом. Но способы преобразования, описанные в этой работе, сами не лишены ошибок. Одни ошибки связаны с проблемой дубликатов строк (печально известные "семантические рифы"), другие — с изъянами в реализации квантора EXISTS в языке SQL (см. главу 19).

В [18.40] предпринята попытка сформулировать всю проблему на теоретической основе (основной проблемой предыдущих работ являлось то, что, как обнаружили различные авторы, семантическое и синтаксическое поведение вложений и агрегирующих функций в языке SQL продумано недостаточно глубоко). В ней также определены расширенные версии реляционного исчисления и реляционной алгебры (расширения касались действий над агрегированными и пустыми значениями) и доказана эквивалентность этих двух расширенных формулировок (при этом использовался более элегантный, новый метод доказательства по сравнению с ранее опубликованным методом). Затем семантика языка SQL представлена с помощью отображения конструкций этого языка на расширенное исчисление, определенное в работе. Тем не менее, необходимо отметить приведенные ниже замечания.

1. Обсуждаемый диалект языка SQL ближе к диалектам, используемым в коммерческих продуктах, по сравнению с диалектами языка SQL, которые используются в [18.37]—[18.39]. Однако этот диалект еще не стал полностью общепринятым. Он не включает оператор UNION И прямой поддержки операторов типа "=ALL" и ">ALL" (см. приложение Б), а трактовка *неизвестных* истинных значений (см. главу 19) отличается от трактовки, обусловленной стандартом языка SQL (на самом деле эта трактовка даже лучше).
2. В данной работе для *технического упрощения* не рассматриваются методы исключения дубликатов кортежей. Но влияние такого упрощения не совсем ясно просматривается, исходя из того, что (как было показано ранее) наличие или отсутствие дубликатов кортежей может значительно повлиять на правильность или другие характеристики некоторых преобразований [6.6].

Наконец, в [18.41] утверждается, что в некоторых случаях оригинальные алгоритмы Кима [18.37], несмотря на неправильность, могут быть эффективнее *общей стратегии,* изложенной в [18.39]. Поэтому автор работы [18.41] предлагает альтернативный способ исправления ошибок в алгоритмах Кима. К тому же в этой работе представлены некоторые улучшения рассматриваемых алгоритмов.

18.42. Baekgaard L., Mark L. Incremental Computation of Nested Relational Query Expressions //ACM TODS. — June 1995. - 20, № 2.

Еще одна статья об оптимизации запросов, включающих подзапросы в стиле языка SQL, в частности *коррелированные* подзапросы. Предлагаемая стратегия включает преобразование исходного запроса в эквивалентный запрос без вложений с последующей инкрементной оценкой полученной версии запроса без вложений.

"Для поддержки первого этапа разработан очень краткий алгоритм преобразования одного алгебраического выражения в другое... В [преобразованном] выражении широко используется оператор MINUS. Для реализации второго этапа предложен и проанализирован эффективный алгоритм инкрементной оценки операций MINUS". Термин *инкрементное вычисление* означает, что для оценки данного запроса могут использоваться ранее вычисленные результаты.

18.43. Rao J., Ross K.A. Using Invariants: A New Strategy for Correlated Queries // Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data. — Seattle, Wash. — June 1998.

Еще одна статья об оптимизации запросов, в том числе подзапросов в стиле языка SQL.

18.44. Warren D.H.D. Efficient Processing of Interactive Relational Database Queries Expressed in Logic // Proc. 7th Int. Conf. on Very Large Data Bases. — Cannes, France. — September 1981.

Представлен подход к оптимизации запросов на принципиально иной основе, а именно с применением формальной логики. Описаны методы, которые используются в экспериментальной СУБД, основанной на применении языка Prolog. Создается впечатление, что эти методы очень похожи на методы, используемые в системе System R, хотя разработаны они были совершенно независимо и с несколько иными целями. В работе указывается, что в отличие от обычных языков запросов, таких как QUEL и SQL, языки, основанные на логике (подобные языку Prolog), позволяют записывать запросы так, чтобы выделить следующее:

- логические цели, которые являются основными компонентами запроса;
- логические переменные, связывающие эти компоненты;
- порядок достижения логических целей, который является критическим момен том с точки зрения реализации.

Из этого следует, что подобные языки весьма удобны в качестве основы для выполнения оптимизации. Действительно, такой язык можно рассматривать в качестве еще одного кандидата для внутреннего представления запроса, исходно сформулированного с помощью какого-либо другого языка (см. раздел 18.3).

18.45. Ioannidis Y.E., Wong E. Query Optimization by Simulated Annealing // Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data. — San Francisco, Calif. — May 1987.

С увеличением числа отношений, используемых в запросе, количество возможных планов выполнения этого запроса растет экспоненциально. В традиционных коммерческих приложениях количество отношений в запросе обычно невелико и, следовательно, количество потенциальных планов выполнения запроса *(пространство поиска)* остается в разумных пределах. Тем не менее, в современных приложениях количество отношений в запросе может быть достаточно большим (см. главу 22). Более того, в приложениях нового типа ощущается необходимость *глобальной* (т.е. для многих запросов) оптимизации [18.47] и поддержки рекурсивных запросов. Эти функции также значительно увеличивают пространство поиска. Исчерпывающий поиск в таких условиях становится неэффективным, и возникает настоятельная необходимость в применении методов сужения пространства поиска.

В публикации содержатся ссылки на более ранние работы по проблеме оптимизации запросов с большим количеством отношений и многозапросной оптимизации. Однако в данной работе утверждается, что для оптимизации рекурсивных запросов не разработано ни одного подходящего алгоритма, после чего приводится алгоритм, который, как утверждают авторы, применим даже на пространствах поиска большого размера. В частности, показано, как применять предложенный алгоритм в случае рекурсивных запросов. Данный алгоритм (называемый *эмуляцией отжига* — simulated annealing, поскольку он имитирует процесс отжига, с помощью которого выращивают кристаллы, сначала прогревая жидкий раствор, а затем постепенно его охлаждая) является вероятностным алгоритмом поиска экстремума, который может успешно применяться для решения проблемы оптимизации в других контекстах.

18.46. Swami A., Gupta A. Optimization of Large Join Queries // Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data. — Chicago, 111. — June 1988.

Общая проблема определения оптимального порядка выполнения соединений в запросах, использующих большое количество отношений (что характерно для дедуктивных баз данных, описываемых в главе 24), является комбинаторно сложной. В работе представлен сравнительный анализ нескольких алгоритмов, предназначенных для решения этой проблемы: обход возмущений (perturbation walk), псевдослучайная генерация данных, последовательные улучшения, последовательные эвристики и эмуляция отжига (simulated annealing) [18.45] (немного "поэзии" в предмет обсуждения, который выглядит достаточно прозаично, добавляют сами названия методов). Согласно результатам проведенного анализа, алгоритм последовательных улучшений превосходит все остальные алгоритмы, в частности, алгоритм эмуляции отжига *сам по себе* не очень эффективен для больших запросов с использованием соединений.

18.47. Sellis T.K. Multiple-Query Optimization //ACM TODS. - March 1988. - 13, № 1.

Классическое исследование оптимизации, в котором автор сосредоточился на проблеме оптимизации отдельных изолированных реляционных выражений. Но в будущем возможность оптимизации нескольких отдельных запросов как одного модуля станет, вероятно, весьма важной. Одной из причин такого состояния дел стало то, что единственный запущенный запрос на верхнем уровне может быть преобразован в несколько запросов на реляционном уровне. В работе приведен следующий пример. Выдача на естественном языке запроса: "Хорошо ли оплачивается работа Майка?" может привести к выполнению трех отдельных приведенных ниже реляционных запросов.

- Зарабатывает ли Майк больше 75 тыс. долл.?
- Зарабатывает ли Майк больше 60 тыс. долл. и обладает ли он опытом работы меньше 5 лет?
- Зарабатывает ли Майк больше 45 тыс. долл. и обладает ли он опытом работы меньше 3 лет?

Этот пример демонстрирует, что наборы связанных запросов обычно совместно используют некоторые общие подвыражения, поэтому подобные наборы запросов требуют глобальной оптимизации.

В работе рассматриваются запросы, в которых используются только конъюнкции выборок и (или) соединения по эквивалентности, а также излагаются экспериментальные результаты и предлагаются направления дальнейших исследований.

18.48. Lohman G.M. Grammar-Like Functional Rules for Representing Query Optimization Alternatives // Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data. - Chicago, 111. - June 1988.

В некоторых случаях реляционные оптимизаторы можно рассматривать как экспертные системы. Тем не менее, исторически сложилось так, что правила, управляющие процессом оптимизации, встраиваются непосредственно в процедурный код, а не записываются отдельно в декларативном виде. Вследствие этого расширение оптимизатора для внедрения новых методов оптимизации — непростая задача. Будущие системы баз данных (глава 26), видимо, еще больше усугубят эту проблему, поскольку явно просматривается необходимость индивидуальной установки дополнительных наборов правил, расширяющих возможности оптимизатора для поддержки определяемых пользователем специальных типов данных. Поэтому разные исследователи предложили структурировать оптимизатор как типичную экспертную систему с использованием явно выраженных декларативных правил.

Однако эта идея страдает недостаточной производительностью. В частности, на любой стадии обработки запроса может применяться большое количество правил, и поиск подходящего правила может потребовать достаточно сложных вычислений. В этой работе представлен альтернативный подход (в данный момент используемый в прототипе системы Starburst — см. [18.21], атакже [26.19], [26.23], [26.29] ,и [26.30]), в котором правила определяются посредством продукционных правил грамматики, подобных правилах грамматики формальных языков. Эти правила, называемые правилами STAR (STrategy Alternative Rules —альтернативные стратегические правила), позволяют осуществлять рекурсивное формирование планов выполнения запросов из других планов и *операторов плана нижнего уровня* (LOwLEvel Plan OPerator— LOLEPOP), которые являются базовыми операциями над отношениями, такими как соединение, сортировка и т.п. Каждый из операторов LOLEPOP имеет свои *подвиды.* Например, оператор LOLEPOP соединения имеет разновидность сортировки-слияния, разновидность хэширования и т.д. В работе утверждается, что изложенный выше подход имеет множество преимуществ: правила (STAR) легко понятны тем, кто будет создавать новые правила; .процесс выбора правила, которое нужно применить в конкретной ситуации, проще и эффективнее по сравнению с традиционным подходом, применяемым в экспертных системах; кроме того, достигнута возможность расширяемости.

18.49. Nakano R. Translation with Optimization from Relational Calculus to Relational Algebra Having Aggregate Functions//ACM TODS. — December 1990. — 15, № 4.

Как показано в главе 8 (раздел 8.4), запросы на языке, базирующемся на реляционном исчислении, можно реализовать посредством преобразования исходного запроса в эквивалентное алгебраическое выражение с последующей оптимизацией полученного алгебраического выражения, которая завершается выполнением этого оптимизированного выражения. Автор предлагает схему объединения первого и второго этапов, т.е. преобразование данного выражения реляционного исчислении

непосредственно в *оптимальное* выражение в реляционной алгебре. Утверждается, что эта схема "более эффективна и перспективна... так как сложное алгебраическое выражение оптимизировать весьма непросто". В процессе оптимизации используются некоторые *эвристические* преобразования, сформированные на основе экспертных знаний об эквивалентности некоторых выражений исчисления и алгебры.

18.50. Whang K-Y., Krishnamurthy R. Query Optimization in a Memory-Resident Domain Relational Calculus Database System //ACM TODS. — March 1990. — 15, № 1.

Наиболее дорогостоящей составляющей в обработке запросов (в среде с достаточ но большой оперативной памятью, как предполагается в данной работе) является вычисление логических выражений. Поэтому в подобной среде целью оптимиза ции является минимизация количества таких вычислений.

18.51. Freytag J.C., Goodman N. On the Translation of Relational Queries into Iterative Programs//ACM TODS. - March 1989. - 14, № 1.

Представлены методы непосредственной компиляции реляционных выражений в выполняемый код на таком языке, как C или Pascal. Отметим, что этот подход отличается от подхода, изложенного в данной главе, где оптимизатор для создания плана выполнения запроса комбинировал предварительно написанные (параметризованные) фрагменты кода.

18.52. Опо К., Lohman G.M. Measuring the Complexity of Join Enumeration in Query Optimization // Proc. 16th Int. Conf. on Very Large Data Bases. — Brisbane, Australia. — August 1990.

Поскольку операция соединения в своей основе является бинарной, оптимизатор должен разбить соединение N отношений (N > 2) на последовательность бинарных соединений. Большинство оптимизаторов выполняет эту операцию в строго *вложенном* порядке. Оптимизаторы выбирают пару отношений, которые будут соединены первыми, после чего соединяют результат с третьим отношением и т.д. Другими словами, выражение, подобное A JOIN B JOIN c JOIN D, будет рассматриваться как ((D JOIN в) JOIN C) JOIN A, но ни в коем случае не как (A JOIN D) JOIN (в JOIN C). Более того, традиционные оптимизаторы разрабатываются так, чтобы вообще исключить выполнение операции декартова произведения. Показано, что оба этих метода могут рассматриваться как способы *сужения пространства поиска,* поэтому для выбора последовательности соединения все еще нужны эвристические методы.

В данной публикации также содержится описание других аспектов применения оптимизатора прототипа системы IBM Starburst (см [18.21], [18.48], [26.19], [26.23], [26.29] и [26.30]. Автор аргументирует это тем, что приведенные варианты тактики в некоторых случаях неуместны, поэтому возникает необходимость в *адаптивном* оптимизаторе, который можно заставить использовать разные варианты тактики для различных запросов.

18.53. Vance B., Maier D. Rapid Bushy Join-Order Optimization with cartesian Products // Proc. 1996 ACM S1GMOD Int. Conf. on Management of Data. — Montreal, Canada. — June 1996.

Как сказано в аннотации к [18.52], оптимизаторы стремятся *сократить пространство поиска* (помимо всего прочего) за счет исключения планов выполнения запросов, в которых используется декартово произведение. В этой статье показано, что поиск во всем пространстве "допустим в большей степени, чем это признавалось прежде", а исключение декартова произведения не всегда желательно (в этой связи см. обсуждение *звездообразного соединения* в главе 22). Согласно утверждениям авторов, основными результатами данной статьи являются полное отделение проблемы определения порядка соединения от проблемы анализа предикатов и представление *новых технологий реализации* методов выбора порядка соединения.

18.54. Ioannidis Y.E., Ng R.T., Shim K., Sellis T.K. Parametric Query Optimization // Proc. 18th Int. Conf. on Very Large Data Bases. — Vancouver, Canada. — August 1992.

Рассмотрим следующий запрос.

EMP WHERE SALARY > <salary>

Здесь параметр <salary> (зарплата) задается во время выполнения запроса. Предположим, что атрибут SALARY обладает индексом. В таком случае справедливы приведенные ниже утверждения.

Если параметр <salary> имеет значение 10тыс. долл. в месяц, то лучшим способом реализации запроса является индекс (так как можно предположить, что для большинства сотрудников не выполняется условие выборки). ■ Если параметр <salary> имеет значение 1 тыс. долл. в месяц, то лучшим способом реализации запроса будет последовательный просмотр (так как можно предположить, что практически для всех сотрудников будет выполняться условие выборки).

Данный пример демонстрирует утверждение о том, что некоторые решения об оптимизации лучше принимать на этапе прогона даже в компилирующих системах. В работе исследована возможность генерации множеств планов выполнения запросов во время компиляции (каждый план является *оптимальным* для некоторого подмножества из множества всех возможных значений параметров, применяемых на этапе прогона) и выбора соответствующего плана на этапе прогона, когда реальные значения параметров уже известны. В частности, автор обращает внимание на один конкретный параметр — размер пространства буфера для запроса. Результаты экспериментов показали, что данный подход незначительно замедляет процесс оптимизации и почти не приводит к снижению качества генерируемых планов запросов. Поэтому автор утверждает, что данный подход может существенно повысить производительность обработки запросов. "Значительный выигрыш в производительности можно получить от использования планов запросов, непосредственно связанных с определенными значениями параметров".

18.55. Kabra N., DeWitt D.J. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans // Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data. - Seattle, Wash. — June 1998.

18.56. Gray J. Parallel Database Systems 101 // Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data. — San Jose, Calif. — May 1995.

Это не научная статья, а расширенная аннотация для презентации учебного посо бия. По сути, основная идея параллельных систем заключается в разбиении боль шой задачи на несколько малых задач, которые могут решаться одновременно, что способствует общему повышению производительности (в том числе пропускной способности и времени отклика). В частности, реляционные системы баз данных прекрасно подходят для распараллеливания обработки исходя из самой сути реля ционной модели. Дело в том, что концептуально легко выполняются следующие действия: разбиение отношения несколькими разными способами на несколько подчиненных отношений; разбиение реляционного выражения на несколько под выражений также несколькими разными способами. В духе заголовка этой статьи следует отметить некоторые важные концепции параллельных систем баз данных. Прежде всего, сама по себе архитектура используемого аппаратного обеспечения должна предполагать возможность параллельной работы. Ниже перечислены три основных типа такой архитектуры, каждый из которых предполагает наличие не скольких процессоров, нескольких жестких дисков, а также сети для обмена дан ными.

- *С разделением оперативной памяти.* Сеть позволяет всем процессорам обра щаться к общей оперативной памяти.
- *С разделением дисковой памяти.* Каждый процессор обладает собственной опе ративной памятью, но сеть позволяет всем процессорам обращаться ко всем дискам.
- *Без разделения.* Каждый процессор обладает собственной оперативной и диско вой памятью, но сеть позволяет всем процессорам обмениваться данными ме жду собой.

На практике обычно используется архитектура *без разделения,* по крайней мере, для больших систем (поскольку в двух других подходах при увеличении количества процессоров очень скоро начинают возникать конфликтные ситуации). Если говорить точнее, то архитектура *без разделения* обеспечивает **линейное ускорение,** т.е. повышение времени отклика в N раз при увеличении вычислительной мощности аппаратного обеспечения в N раз, и **линейное масштабирование** (scalability), т.е. время отклика остается постоянным при увеличении вычислительной мощности аппаратного обеспечения и объема данных в одинаковое количество раз. Ниже представлены некоторые способы *секционирования данных* (data partitioning), т.е. разбиения отношения r на секции или подчиненные отношения и распределение этих секций междуп различными процессорами.

- *Секционирование по диапазону* (range partitioning). Отношение r делится на не пересекающиеся секции 1, 2, ... п на основе значений некоторого под множества s атрибутов отношения r (концептуально отношение r отсортиро вано по подмножеству значений атрибута s, а результат разделен на п секций равного размера). Секция i при этом соответствует процессору i. Данный под ход очень удобен для запросов с выборками на основе условий равенства или соответствия диапазону для подмножества атрибутов отношения s.

■ *Хэш-секционирование* (hash partitioning). Каждый кортеж t отношения r соответствует процессору h (t), где h — некоторая хэш-функция. Этот метод прекрасно подходит для запросов с выборками на основе условий равенства для одного или нескольких хэш-атрибутов, а также для запросов с последовательным доступом ко всему отношению r.

Циклическое секционирование (round-robin partitioning). Концептуально отношение r отсортировано некоторым образом, а i-й кортеж в отсортированном результате соответствует процессору с номером, вычисленным как результат деления i по модулю n. Этот подход прекрасно подходит для запросов с последовательным доступом ко всему отношению r.

Распараллеливание обработки можно применять для выполнения отдельной операции (распараллеливание между операциями), для выполнения разных операций внутри одного запроса (распараллеливание внутри операций или внутри запросов) и для выполнения разных запросов (распараллеливание между запросами). Учебное руководство по всем этим вариантам приведено в [18.3], а в [18.57] и в [18.58] рассматриваются некоторые конкретные методы и алгоритмы. Следует отметить, что параллельная версия *хэшированного соединения* (см. раздел 18.7) особенно эффективна и широко используется на практике.

18.57. Bitton D., Boral H., DeWitt D.J., Wilkinson K. Parallel Algorithms for the Execution of Relational Database Operations//ACM TODS. — September 1983. — 8, № 3.

Здесь описаны алгоритмы реализации операций сортировки, проекции, соединения, агрегирования и обновления в многопроцессорной среде. Представлены также общие формулы стоимости, учитывающие операции ввода—вывода, загрузку процессора и обмен сообщениями. Эти формулы можно адаптировать к различным многопроцессорным архитектурам.

18.58. Hasan W., Motwani R. Optimization Algorithms for Exploiting the Parallelism-Communication Tradeoff in Pipelined Parallelism // Proc. 20th Int. Conf. on Very Large Data Bases. — Santiago, Chile. — September 1994.

18.59. Kossmann D., Stocker K. Iterative Dynamic Programming: A New Class of Optimization Algorithms//ACM TODS. - March 2000. - 25, №1.

18.60. Godfrey P., Gryz J., Zuzarte C. Exploiting Constraint-Like Data Characterizations in Query Optimization // Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. — May 2001.

18.61. Deutsch A., Poppa L., Tannen V. Physical Data Independence, Constraints, and Optimization with Universal Plans // Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland. — September 1999.

18.62. Stillger M., Lohman G., Markl V., Kandil M. LEO— DB2's LEarning Optimizer// Proc. 27th Int. Conf. on Very Large Data Bases, Rome, Italy.— September 2001.

ГЛАВА 19

Отсутствующая информация

19.1. ВВЕДЕНИЕ

В повседневной жизни часто приходится сталкиваться с проблемой отсутствия некоторой информации. Весьма типичны ситуации, когда, например, "дата рождения не известна", "имя докладчика будет объявлено дополнительно", "адрес лица в данный момент не известен" и т.д. Поэтому в системах баз данных должен существовать механизм обработки подобных ситуаций. На практике наиболее типичный подход к решению этой проблемы (используемый, в частности, в языке SQL, а значит, и в большинстве коммерческих продуктов) основан на применении **неопределенных значений** (NULL-значений) и **трехзначной логики.** Например, вес детали, скажем, с номером P7, может быть не известен, поэтому упрощенно можно сказать, что ее вес "является неопределенным". В более точном смысле это выражение означает следующее: а) известно, что деталь существует; б) несомненно, деталь имеет вес; в) ее вес нам не известен.

Рассмотрим этот пример более подробно. Ясно, что мы не можем поместить истинное значение веса детали в атрибут WEIGHT кортежа, описывающего деталь с номером P7. Следовательно, все, что остается, — *пометить* или как-то *обозначить,* что значение атрибута WEIGHT этого кортежа является неопределенным. В дальнейшем наличие подобной метки будет интерпретироваться как указание, что истинное значение атрибута не известно. Для удобства можно неформально сказать, что атрибут WEIGHT "содержит неопределенное значение" или что значение этого атрибута "равно NULL". На практике подобные

выражения действительно широко используются. Но следует четко понимать, что подобные выражения неформальны и несколько неточны, поскольку говоря, что значение атрибута WEIGHT в кортеже является неопределенным или равно NULL, мы на самом деле имеем в виду, что *кортеж вообще не содержит никакого значения атрибута* WEIGHT. Поэтому широко распространенные выражения "неопределенное значение" и "значение NULL" использовать не рекомендуется, так как это "неопределенное значение (NULL)" на самом деле не значение, а всего лишь некая отметка или обозначение.

В следующем разделе будет показано, что при сравнении любых скалярных значений, в которых какой-либо из операндов не определен (содержит NULL), вместо значения *true* (истина) или *false* (ложь) будет получено логическое значение *unknown* (не известно). Причиной такого состояния дел является принятая нами интерпретация обозначения NULL как указателя "значение не известно". Если значение переменной А не известно, то не известен и результат любого ее сравнения, например вида А > в, причем независимо от значения в (даже если предположить, что значение переменной в также не известно). В частности, следует отметить, что два неопределенных значения (NULL) нельзя считать равными одно другому, поэтому, если обе переменные, А и в, содержат неопределенные значения (NULL), результатом сравнения А = в всегда будет *unknown,* а не *true.* Однако эти переменные не считаются и неравными, т.е. результатом сравнения А не =В также будет *unknown.* На этом и основано понятие трехзначной логики, поскольку концепция неопределенных значений, по крайней мере, в общепринятом смысле, неизбежно приводит нас к необходимости использования логики с тремя логическими значениями: *true (истина), false* (ложь) и *unknown* (не известно).

Прежде чем продолжить обсуждение, автор данной книги считает необходимым еще раз заявить, что он полностью разделяет мнение многих авторов о том, что неопределенные значения (NULL) и трехзначная логика являются ошибочными понятиями и им нет места в четких формальных системах, подобных реляционной модели. Например, утверждение о том, что некоторый кортеж с данными о детали вообще не содержит значения WEIGHT, по определению равносильно утверждению, что рассматриваемый кортеж в конечном итоге не является кортежем с данными о детали; равным образом, это равносильно утверждению, что рассматриваемый кортеж не является конкретизацией соответствующего предиката. Фактически такой "с позволения сказать, кортеж" просто не является кортежем! Это можно легко проверить, еще раз прочитав определение термина *кортеж* в главе 6. Истина заключается в том, что даже сама попытка точно определить, что собой представляет схема, в которой применяются неопределенные значения, позволяет сразу же показать, почему идея неопределенных значений не совсем обоснована. Вследствие этого ее также трудно достаточно логично объяснить. Как указано в [11.10], "она приобретает хоть какой-то смысл, только если вы на многое закрываете глаза и не слишком утруждаете себя размышлениями".

Тем не менее, не стоит лишать читателя возможности обсудить неопределенные значения и трехзначную логику, поэтому в настоящее издание и включена данная глава.

---

[1] Во всей данной книге для представления истинностных значений используются ключевые слова, состоящие полностью из прописных букв. В отличие от этого, в данной главе истинностные значения представлены с помощью прописных букв и выделены курсивом (в основном для того, чтобы принятый стиль соответствовал другим публикациям автора по той же теме).

План главы таков. Непосредственно после введения в разделе 19.2 без излишнего недоверия и критики описываются основные идеи, на которых базируется концепция неопределенных значений и трехзначной логики. (Суть в том, что невозможно строго и справедливо критиковать любые идеи без их предварительного описания.) Затем в разделе 19.3 обсуждаются некоторые более важные следствия изложенных идей и предпринимается попытка обосновать мнение автора о том, что понятие неопределенных значений является ошибочным. В разделе 19.4 рассматриваются следствия наличия неопределенных значений в первичных и внешних ключах. Далее, в разделе 19.5, будет сделано отступление от основной темы в целях описания чаще всего встречающихся операций в контексте неопределенных значений и трехзначной логики — операций *внешнего соединения.* В разделе 19.6 в очень сжатой форме описывается альтернативный способ обработки отсутствующей информации с использованием *специальных значений.* В разделе 19.7 кратко рассматриваются аспекты языка SQL, имеющие отношение **к** обсуждаемой проблеме. Наконец, в разделе 19.8 приводится краткое резюме.

Еще одно предварительное замечание. Существует множество причин, не позволяющих поместить истинное значение в тот или иной атрибут кортежа, и неизвестность этого значения — только одна из возможных причин. Среди других причин следует назвать такие, как "значение не применимо", "значение не существует", "значение не определено", "значение не задано" и т.д. [19.5][2]. Действительно, в [6.2] Кодд предложил использовать в реляционной модели не один, а два вида неопределенных значений: одно из них —"значение не известно", а второе — "значение не применимо". В результате в предложенной Коддом реляционной модели используется уже не трех-, а четырехзначная **логика.** Автор данной книги возражает против подобных предложений в этой и во всех других своих работах (например, в [19.5]). В данной главе мы сосредоточимся только на одном виде неопределенных значений, а именно — на неопределенных данных с неизвестным значением. На него мы достаточно часто, но не всегда, будем ссылаться с помощью аббревиатуры **UNK** (сокращение от "unknown" — "неизвестно").

19.2. ОБЗОР КОНЦЕПЦИИ ТРЕХЗНАЧНОЙ ЛОГИКИ

В этом разделе описываются принципиальные компоненты концепции трехзначной логики применительно к проблеме отсутствующей информации. Начнем обсуждение (в двух следующих подразделах) с рассмотрения влияния неопределенных значений (т.е. UNK) на вычисление логических выражений.

Логические операторы

Выше уже отмечалось, что результатом операций сравнения скаляров, в которых хотя бы один из операндов является величиной UNK, будет логическое значение *unknown,* а не значение *true* или *false.* Поэтому в таких случаях приходится иметь дело с трехзначной

---

[2] Однако следует особо отметить, что в любом из перечисленных выше случаев как таковой "отсутствующей информации" нет. Например, если к сотруднику ' Joe' понятие комиссионного воз награждения "не применимо", значит, к нему не применим сам принцип выплаты комиссионных и, таким образом, здесь нет никакой отсутствующей информации. (Тем не менее, если кортеж с описанием сотрудника 'Joe' содержит "неприменимое неопределенное значение" в атрибуте комиссионного вознаграждения, этот кортеж вообще не является кортежем с описанием сотрудника, поскольку он не является корректной реализацией предиката "Описание сотрудника". В действительности, он даже не является кортежем.)

логикой. Третьим логическим значением здесь является значение *unknown,* на которое мы достаточно часто, но не постоянно, будем ссылаться с помощью сокращения *unk.* Ниже приведены таблицы истинности для операторов AND, OR и NOT в трехзначной логике (в таблицах используются следующие сокращения: t — *true, f—false, u — unk).*

AND	t	u	f
t	t	u	f
u	u	u	f
f	f	f	f

OR	t	u	f
t	t	t	t
u	t	u	u
f	t	u	f

NOT	
t	f
u	u
f	t

Предположим, что A = 3, в = 4 и переменная с имеет значение UNK. Тогда приведенные ниже выражения будут иметь следующие результаты (которые показаны справа).

```
A > B AND B > C  :      false
A > B OR B > C : unk
A < B OR B < C : true
NOT ( A = C )   : unk
```

, Тем не менее, для реализации трехзначной логики одних только операторов AND, OR и NOT недостаточно [19.11]. Еще одним важным оператором является оператор MAYBE (возможно) [19.5]. Таблица истинности данного оператора показана ниже.

MAYBE	
t	f
u	t
f	f

Чтобы продемонстрировать необходимость в использовании оператора MAYBE, рассмотрим запрос: "Получить сведения о сотрудниках с годовой зарплатой меньше 50 тыс. долл., которые могут быть (но это точно не известно) программистами и родились до 18 января 1971 года". С помощью оператора MAYBE данный запрос можно достаточно кратко записать в следующем виде[3].

```
EMP WHERE MAYBE ( JOB = 'Programmer' AND
                  DOB < DATE ('1971-1-18')
                  AND SALARY < 50000.00 )
```

(Здесь предполагается, что атрибуты JOB, DOB и SALARY переменной отношения EMP имеют типы CHAR, DATE и RATIONAL, соответственно.) Но без оператора MAYBE ЭТОТ же запрос будет выглядеть следующим образом.

```
EMP WHERE    ( JOB = 'Programmer'
               OR IS UNK ( JOB ) )
   AND      ( DOB < DATE ('1971-1-18')
   OR IS UNK ( DOB ) )              (
   SALARY < 50000.00
             OR IS UNK ( SALARY ) )
    AND NOT ( JOB = 'Programmer'
    AND
              DOB < DATE ('1971-1-18') AND
                   SALARY < 50000.00 )
```

[3] Чтобы составить примеры, приведенные в данной главе, необходимо было принять допущение, что язык Tutorial D включает средства поддержки значений UNK и трехзначной логики. Фактически такая поддержка, разумеется, не предусмотрена.

Здесь предполагается существование другого истинностного оператора **IS_UNK,** который принимает в качестве параметра единственный скалярный операнд и возвращает значение *true,* если этот операнд равен UNK, или значение *false* — в противном случае. (Кстати, следует отметить, что определенная версия оператора **IS_DNK** необходима и для операндов, отличных от скалярных. Но автор не предпринимает попытку определить здесь подобную конструкцию, поскольку решение этой задачи связано со слишком большими сложностями, к тому же он вообще не испытывает доверия к утверждению о необходимости поддержки трехзначной логики.)

Не следует воспринимать изложенное выше как подтверждение того, что оператор MAYBE является *единственным* дополнительным логическим оператором, который необходим для поддержки трехзначной логики. На практике, например, весьма полезным может оказаться оператор TRUE_OR_MAYBE (который возвращает *true,* если его операнд равен *true* или *unk,* а в противном случае возвращает *false),* как показано в [19.5]. См. также аннотацию к [19.11] в разделе "Дополнительная литература".

Кванторы

Вопреки тому несомненному факту, что для представления большинства примеров в этой книге используется реляционная алгебра, а не исчисление, нам все же следует рассмотреть влияние трехзначной логики на вычисление кванторов реляционного исчисления EXISTS и FORALL. Как говорилось в главе 8, кванторы EXISTS И FORALL определяются как итерационные конструкции из операторов OR и AND, соответственно. Иначе говоря, если r — отношение с кортежами tl, t2, ..., tm, V — переменная области значений, которая принимает значения из данного отношения r , а p (V) — логическое выражение, в котором V используется как свободная переменная, то выражение

```
EXISTS V ( p ( V ) )
```

по определению эквивалентно следующему выражению.

```
false OR p ( tl ) OR ... OR p ( tm )
```

Аналогичным образом, выражение

```
FORALL V ( p ( V ) )
```

по определению эквивалентно следующему выражению.

```
true AND p ( tl ) AND ... AND p ( tm )
```

Что произойдет, если для некоторого i значение p(ti) будет равно *unk?* Например, пусть отношение r содержит следующие кортежи.

```
( 1, 2, 3 )
( 1 , 2 , U N K )
( UNK, UNK, UNK )
```

Предположим для простоты, что тремя атрибутами в указанном выше порядке слева направо являются атрибуты A, в и с, соответственно, и каждый из них имеет тип INTEGER. Ниже приведены примеры различных выражений и результаты их вычисления.

```
EXISTS v  ( v.c > 1 )                : true
EXISTS V ( V.B > 2 )                 : unk
EXISTS V ( MAYBE ( V.A > 3 ) )       : true
EXISTS V ( IS UNK ( V.C ) )          : true

FORALL V ( V.A > 1 )                 : false
FORALL V ( V.B > 1 )                 : unk
FORALL V ( MAYBE ( V.C > 1 ) )       : false
```

Другие скалярные операторы

Рассмотрим следующее числовое выражение.

```
WEIGHT * 454
```

Здесь атрибут WEIGHT представляет вес некоторой детали. Что будет, если вес рассматриваемой детали является величиной UNK? Каким будет значение всего этого выражения? Ответ состоит в том, что значением всего выражения также должна быть величина UNK. В общем случае, если *хотя бы один* из операндов арифметического выражения является величиной UNK, результатом вычисления всего выражения также будет величина UNK. Таким образом, например, если атрибут WEIGHT содержит величину UNK, результатом вычисления всех приведенных ниже выражений также будет величина UNK.

- `WEIGHT + 454`
- `454 + WEIGHT`
- `+ WEIGHT`
- `WEIGHT - 454`
- `454 - WEIGHT`
- `- WEIGHT`
- `WEIGHT * 454`
- `454 * WEIGHT`
- `WEIGHT / 454`
- `454 / WEIGHT`

Примечание. Вероятно, следует сразу же отметить, что подобная трактовка числовых выражений может привести к появлению некоторых аномалий. Например, выражение WEIGHT - WEIGHT, значением которого должен быть нуль, на самом деле будет иметь значение UNK, а результатом вычисления выражения WEIGHT/0 будет также величина UNK вместо обычного в данном случае сообщения об ошибке деления на нуль (безусловно, в обоих выражениях предполагается, что атрибут WEIGHT содержит величину UNK). Такие аномалии будут игнорироваться до особого упоминания.

Аналогичные рассуждения применимы ко всем другим скалярным типам и операторам, за исключением логических операторов (рассматриваемых в двух предыдущих подразделах), оператора IS_UNK, описанного выше, и рассматриваемого ниже оператора IF_UNK. Таким образом, для символьных строк выражение A | | в возвращает UNK, если A равен UNK, либо в равен UNK, либо оба равны UNK. (Снова следует отметить существование аномальных случаев, рассмотрение которых мы здесь опустим.)

Оператор **IF_UNK** получает на входе два скалярных операнда и возвращает значение первого операнда, если это не величина UNK. Если первый операнд является величиной UNK, то оператор IF_UNK возвращает значение второго операнда. Другими словами, это — оператор преобразования величины UNK в некоторое значение, отличное от UNK. Например, предположим, что значение UNK разрешено использовать для помещения в атрибут CITY переменной отношения поставщиков S. Рассмотрим следующее выражение.

```
EXTEND S ADD IF_UNK ( CITY, 'City unknown' ) AS SCITY
```

В результате его вычисления будет получена переменная отношения, в которой значение атрибута SCITY равно 'City unknown' для каждого поставщика, город которого в переменной отношения s определен как UNK.

Следует отметить, что оператор IF_UNK можно определить **с** помощью оператора IS_UNK. Точнее говоря, выражение

```
IF_UNK ( exp1, exp2 )
```

где параметры expl и exp2 должны иметь одинаковые типы, эквивалентно следующему выражению.

```
IF IS_UNK ( expl ) THEN exp2 ELSE expl END IF
```

UNK — это не *unк*

Очень важно понимать, что величина UNK (неопределенное значение вида "значение не известно") и значение *unк* (логическое значение *unknown)* — это **не** одно **и** то же[4]. Действительно, данное положение дел является прямым следствием того, что *unк* — это значение (точнее, истинностное значение), в то время как величина UNK вообще не является значением. Сформулируем сказанное выше более точно. Предположим, что X является переменной логического типа (BOOLEAN). Тогда она может принимать одно из следующих значений: *true, false* или *unк.* Таким образом, выражение "X равно *unк"* означает в точности то, что значение переменной *X* известно и равно *unк.* В отличие от этого, выражение "X равно UNK" означает, что значение переменной **x не известно.**

Проблема принадлежности величины UNK к типу

Из того факта, что шгк *не* является значением, прямо следует заключение, что типы (т.е. множества *значении)* не могут содержать величину UNK. Действительно, *если бы* типы могли содержать неопределенные значения какого-либо вида, то проверка ограничений целостности для них всегда завершалась бы с положительным результатом! Но, поскольку типы фактически *не могут* содержать величину UNK, "отношения", допускающие присутствие величины UNK, являются чем угодно, но только **не реляционными отношениями.** Причем это верно как для отношений, определение которых дано в главе 6, так и для отношений, отвечающих оригинальному определению Кодда [6.1]. К обсуждению этого важного вопроса мы еще вернемся позже.

Реляционные операторы

Рассмотрим влияние величины ШГК на операторы реляционной алгебры. Для простоты ограничимся пятью примитивными операторами: произведение, сокращение, проекция, объединение и разность (воздействие величины UNK на другие операторы можно определить на основе заключений о ее воздействии на пять перечисленных операторов).

Прежде всего, отметим, что на операцию **произведения** величина UNK никакого влия ния не оказывает.

Далее, операцию сокращения следует несколько переформулировать, потребовав, чтобы возвращались только кортежи, для которых условие сокращения имеет значение *true,* т.е. эта операция не должна включать в результат кортежи, для которых условие сокращения принимает значение *false* или *unк.*

[4] Тем не менее, в стандарте SQL они рассматриваются как эквивалентные понятия (см. раздел 19.7).

Примечание. Использование именно этой формулировки неявно предполагалось в примере с оператором MAYBE, приведенном выше, в подразделе "Логические операторы" данного раздела.

Теперь рассмотрим операцию **проекции.** Для получения проекции отношения требуется, кроме всего прочего, исключить дубликаты кортежей. В двухзначной логике два кортежа tl и t2 считаются дубликатами тогда и только тогда, когда они имеют одинаковые атрибуты A1, A2, . . . , An и для всех i (i = 1, 2, . . . , п) значение Ai в кортеже tl равно значению Ai в кортеже t2. Однако в трехзначной логике некоторые из этих значений атрибутов могут содержать величину UNK, а величина UNK (как было сказано выше) не равна *никакой* другой величине, даже самой себе. Следует ли из этого заключить, что кортеж, который содержит величину UNK, не может быть дубликатом другого кортежа и даже самого себя?

Согласно Кодду, ответ на этот вопрос *отрицателен:* две величины UNK, даже если они не равны между собой, считаются дубликатами одна другой в целях исключения дубликатов кортежей [14.7][5]. Это очевидное противоречие объясняется следующим образом:

> *"...проверка на равенство в процессе исключения дубликатов кортежей... выполняется на более низком уровне детализации по сравнению с проверкой на равенство при оценке условий выборки. Поэтому здесь можно применить другое правило ".*

Автор предоставляет читателю право решать, является ли такое обоснование приемлемым. В любом случае, давайте согласимся сейчас с этим обоснованием, а заодно примем приведенное ниже определение.

- Кортежи tl и t2 являются **дубликатами** один другого тогда **и** только тогда, когда они имеют одинаковые атрибуты A1, A2, . . ., An и для всех i (i = 1, 2, . . ., п), либо значение Ai в кортеже tl равно значению Ai в кортеже t2, либо оба значения Ai в кортежах tl и t2 равны UNK.

Теперь, исходя из этого расширенного определения дубликатов кортежей, можно сохранить прежнее определение операции проекции в неизменном виде. Но следует отметить, что больше нельзя утверждать, что следующие высказывания эквивалентны:

- **tl = t2;**
- tl и t2 являются дубликатами друг друга.

Объединение также подразумевает исключение избыточных дубликатов кортежей, и для этой операции также может быть применено приведенное выше определение дубликатов кортежей. Таким образом, объединение отношений r1 и r2 одного **и** того же типа можно определить как отношение r того же типа, тело которого состоит из всех возможных кортежей t, таких что кортеж t является дубликатом некоторого кортежа в отношении rl или в отношении r2 (или в них обоих одновременно).

Наконец, операция **разности** определяется аналогичным образом (хотя она и не требует исключения дубликатов кортежей). Иначе говоря, кортеж t попадает в результат

---

[5] Работа [14.7] была первой работой Кодда, в которой проблема отсутствия информации рассматривалась достаточно подробно, хотя описание этой проблемы не являлось основной задачей статьи (см. главу 14). Помимо всего прочего, в статье предлагаются версии "возможного" 9-соединения, 8-выборки (т.е. сокращения), операторы деления (см. упр. 19.4), а также "внешние" версии операций соединения, пересечения, вычитания, 8-соединения и естественного соединения (см. раздел 19.5).

операции r1 MINUS r2 тогда и только тогда, когда этот кортеж является дубликатом какого-либо кортежа в отношении rl, но не является дубликатом ни одного из кортежей в отношении r2. (Для полноты картины отметим, что операция **пересечения** определяется аналогично, хотя она, безусловно не относится к типу примитивных. Это означает, что кортеж t попадает в результат операции rl INTERSECT r2 тогда и только тогда, когда этот кортеж является дубликатом какого-либо кортежа в отношении rl и одновременно дубликатом какого-либо кортежа в отношении r2.)

Операции обновления

Здесь следует упомянуть два описанных ниже основных момента.

1. Если атрибут A отношения R может содержать величину UNK и если вставляемый в отношение R с помощью оператора INSERT кортеж не содержит значения для ат рибута A, то система автоматически поместит величину UNK на место отсутствую щего значения (безусловно, здесь предполагается, что для A не определено задан ное по умолчанию значение, отличное от UNK.) Если атрибут A в переменной отношения R не допускает присутствия величины UNK, то попытка поместить в R кортеж (с помощью операции INSERT или UPDATE), в котором в атрибут A поме щено значение UNK, приведет к ошибке.
2. Попытка поместить в отношении R дубликат кортежа (с помощью операции INSERT или UPDATE) обычно является ошибкой. В данном случае определение дубликатов кортежей взято из предыдущего подраздела.

Ограничения целостности

Как указывалось в главе 9, ограничение целостности неформально можно рассматривать как логическое выражение, результат вычисления которого не должен быть равен *false.* Следовательно, ограничение целостности не будет нарушено, если результат его вычисления будет равен значению *unk* (на самом деле, это неявно предполагалось в замечаниях, приведенных выше в настоящем разделе в отношении ограничений типов). Безусловно, в этом случае точнее было бы сказать, что нам ничего *не известно* о том, будет ли нарушено данное ограничение целостности. Но с известной долей приближения вполне можно утверждать, что если в конструкции WHERE значение *unk* рассматривается как *false,* то в ограничениях целостности оно рассматривается как *true* (и здесь в обоих случаях применяются несколько неформальные утверждения).

19.3. НЕКОТОРЫЕ СЛЕДСТВИЯ ИЗЛОЖЕННОЙ СХЕМЫ

Использование подхода с трехзначной логикой, описанной в предыдущем разделе, имеет ряд логических следствий, причем некоторые из них совсем не очевидны. В данном разделе обсуждаются эти следствия и их значения.

Преобразование выражений

Прежде всего, отметим, что выражения, которые в двухзначной логике всегда возвращают значение *true,* в трехзначной логике не обязательно будут всегда давать тот же результат. Ниже приведено несколько примеров с комментариями, но следует иметь в виду, что этот список далеко не полный.

- *Сравнение* x = x *не обязательно в результате даст true.*
  В двухзначной логике любая переменная x всегда равна самой себе. В трехзначной логике переменная x не равна самой себе, если она содержит величину UNK.
- *Логическое выражение* p OR NOT (p) *не обязательно в результате даст true.*
  Здесь p — это некоторое логическое выражение. В двухзначной логике выражение p OR NOT(p) всегда имеет значение *true* (т.е. истинно), независимо от значения p. Но в трехзначной логике, если p равно *unk,* общее выражение сводится к *unk* OR NOT (*unk),* т.е. к выражению *unk* OR *unk,* что в свою очередь упрощается до зна чения *unk,* а не *true.* Этот частный пример демонстрирует хорошо известное, но сложное для понимания свойство трехзначной логики, которое можно описать следующим образом. Если выполнить два запроса, "Получить сведения обо всех поставщиках из Лондона" и "Получить сведения обо всех поставщиках не из Лон дона", а затем объединить результаты обоих запросов, то *не* обязательно будут по лучены сведения обо *всех* поставщиках. Чтобы получить список всех поставщиков, к двум запросам нужно добавить еще один: "Получить сведения обо всех постав щиках, которые, *возможно* (may be), находятся в Лондоне" (иными словами, в трехзначной логике выражением, которое всегда принимает значение *true,* т.е. яв ляется аналогом выражения p OR NOT (p) двухзначной логики, служит выраже ние p OR NOT(p) OR MAYBE (p)).
  Поэтому имеет смысл исследовать приведенный выше пример немного более подробно. Суть его, безусловно, состоит в том, что в реальном мире состояния "находится в Лондоне" и "находится не в Лондоне" являются взаимоисключающими и покрывают весь спектр возможностей. Однако в базе данных содержится *не* сам реальный мир, а лишь **знания** о **реальном мире,** и эти знания характеризуются тремя, а не двумя возможными состояниями. В рассматриваемом здесь примере это следующие состояния: "место нахождения известно, и это Лондон", "место нахождения известно, и это не Лондон", "место нахождения не известно". Более того, как показано в [19.6], очевидно, что систему базы данных нельзя опрашивать о состоянии реального мира, ей можно задавать вопросы только о тех знаниях о реальном мире, которые представлены в базе данных в виде значений. Такое "противоестественное" свойство, приведенное в примере, порождено тем, что пользователь мыслит в терминах реального мира, а система функционирует, опираясь исключительно на *свои знания* об этом реальном мире. (Тем не менее, автор данной книги подозревает, что описанное несоответствие между предметными областями — это всего лишь ловушка, в которую очень легко попасть. Отметим, что *каждый отдельный запрос,* описанный в предыдущих главах (в примерах, упражнениях и т.п.), был сформулирован в терминах "реального мира", а не в терминах "знаний о реальном мире". И в этом смысле данная книга вовсе не является каким-то исключением.)
- *Вычисление логического выражения r* JOIN r *не обязательно в результате даст r.*
  В двухзначной логике соединение отношения r с самим собой всегда дает в результате исходное отношение r (т.е. операция естественного соединения является *идемпотентной).* Однако в трехзначной логике кортеж, содержащий величину UNK в любой из позиций, не будет соединен сам с собой, поскольку (согласно [14.7]) операция соединения, в отличие от операции объединения, предусматривает проверку

на равенство "в стиле выборки", а не проверку на равенство "в стиле исключения дубликатов кортежей" (именно так!?).

- *Операция* INTERSECT *больше не является частным случаем операции* JOIN. Это заключение является следствием того факта, что, опять же, операция соединения предусматривает проверку на равенство в стиле выборки, тогда как операция пересечения основана на проверке на равенство в стиле исключения дубликатов.
- *Из равенства A* = B AND B = *C вовсе не обязательно следует равенство A* = C. Развернутая иллюстрация этого следствия приведена в подразделе "Пример с ба зой данных отделов и сотрудников".

Таким образом, эквивалентности, допустимые в двухзначной логике, не являются эквивалентностями в трехзначной логике. Одно из наиболее серьезных следствий подобных несоответствий таково. Как правило, в основе различных **законов преобразования,** которые используются для преобразования запросов в более эффективную форму, лежит простая эквивалентность вида r JOIN r = r (см. главу 18). Более того, эти законы используются не только *системой* (в процессе оптимизации), но и *пользователями* (когда они пытаются отыскать "наилучший" способ записи конкретного запроса). Если же исходные эквивалентности становятся недействительными, то построенные на них законы преобразования также являются недействительными. А если законы преобразования недействительны, то и выполняемые на их основе преобразования также недействительны. В свою очередь, недействительные преобразования запросов приводят к получению **неправильных ответов** от системы.

Пример с базой данных отделов и сотрудников

Для иллюстрации проблемы недействительных преобразований рассмотрим более подробно конкретный пример. (Этот пример взят из [19.9]; по некоторым незначимым для данного случая причинам он построен на использовании реляционного исчисления, а не реляционной алгебры). Предположим, что имеется простая база данных отделов и сотрудников, показанная на рис. 19.1.

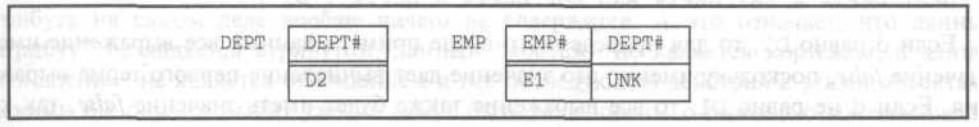


Рис. 19.1. База данных отделов и сотрудников Рассмотрим следующее выражение,

которое может являться частью некоторого запроса.

```
DEPT.DEPT# = EMP.DEPT# AND EMP.DEPT# = DEPT# ('Dl')
```

Здесь DEPT и EMP являются неявно заданными переменными области значений. Для всех кортежей в базе данных приведенное условие сводится к выражению *unk* AND *unk,* т.е. просто к значению *unk.* Тем не менее, "хороший" оптимизатор, исходя из того, что если *A* = *B* и *B* = *C,* то *A* = C, добавит к исходному условию дополнительный терм сокращения *A* - *C* (как описано в главе 18, раздел 18.4). В результате будет получено следующее выражение.

```
DEPT.DEPT# = EMP.DEPT# AND EMP.DEPT# = DEPT# ('D1')
                    AND DEPT.DEPT# = DEPT# ('Dl')
```

Это измененное выражение теперь сведется к выражению *unk* AND *unk* AND *false* или просто к значению *false* (для двух рассматриваемых кортежей в базе данных). В этом свете рассмотрим такой запрос.

```
  EMP.EMP# WHERE EXISTS DEPT ( NOT ( DEPT.DEPT# =
EMP.DEPT# AND EMP.DEPT# = DEPT# ('D1') ) )
```

В результате его выполнения *будет возвращен код сотрудника* E1, *если этот запрос решено "оптимизировать " в изложенном смысле, и не будет возвращено ничего, если решено его "не оптимизировать".* Безусловно, это означает, что проведенная "оптимизация" не корректна. Таким образом, показано, что некоторые методы оптимизации, которые были, безусловно, допустимыми и полезными в обычной двухзначной логике, становятся недопустимыми в трехзначной логике.

Попробуем оценить влияние сказанного выше на способы расширения систем с двухзначной логикой в отношении поддержки трехзначной логики. В лучшем случае подобное расширение потребует некоторой переработки существующей системы, поскольку отдельные фрагменты кода оптимизатора станут работать неправильно, причем в худшем случае оптимизатор будет вносить явные ошибки в результаты запросов. Более общий подход предполагает анализ последствий расширения систем, поддерживающих n-значную логику, до систем поддержки (n+1)-значной логики для любых n > 1; аналогичные сложности возникают для любого дискретного значения n.

Проблема интерпретации

Теперь исследуем приведенный выше пример базы данных отделов и сотрудников более тщательно. Поскольку сотрудник с номером E1 в действительности приписан к конкретному отделу, величина UNK представляет некоторое реальное значение d. В данном случае значение d либо равно D1, либо нет. Если справедливо первое предположение, то проанализируем значение следующего выражения.

```
DEPT.DEPT# = EMP.DEPT# AND EMP.DEPT# = DEPT# ('Dl')
```

Если d равно Dl, то для приведенного выше примера данных все выражение имеет значение *false,* поскольку именно это значение дает вычисление первого терма выражения. Если d не равно D1, то все выражение также будет иметь значение *false,* так как именно это значение будет получено при вычислении второго терма выражения — EMP. DEPT# = DEPT# (' Dl'). Иначе говоря, в реальном мире исходное выражение всегда имеет значение *false,* **независимо от того, какое реальное значение представлено величиной UNK.** Отсюда следует вывод, что результат, правильный в контексте реального мира, и результат, правильный в трехзначной логике, — это не одно и то же! Иначе говоря, трехзначная логика не отражает состояние реального мира, т.е. можно считать, что трехзначная логика не является адекватным инструментом для **интерпретации** состояния реального мира!

Примечание. Эта проблема интерпретации является далеко не единственной проблемой, возникающей в связи с использованием неопределенных значений и трехзначной логики (подробное обсуждение аналогичных вопросов можно найти в [19.1]-[19.11]).

Она даже не является наиболее фундаментальной (см. следующий подраздел). Но, по мнению автора, именно эта проблема имеет наибольшее практическое значение и вызывает наибольший интерес.

Дополнительные сведения о предикатах

Предположим, что отношение, являющееся текущим значением переменной отношения EMP, содержит только два кортежа: (E2,D2) и (E1,UNK). Первое соответствует утверждению: "В отделе с номером D2 есть сотрудник с номером E2", а второе — "Существует сотрудник с номером E1". (Напомним, что утверждение "Кортеж включает величину UNK" означает, что на самом деле этот кортеж вообще не содержит никакого значения в данной позиции кортежа. Таким образом, кортеж (E1,UNK), если такое сомнительное обозначение вообще можно назвать кортежем, на самом деле следует рассматривать как кортеж (E1).) Иначе говоря, эти два кортежа являются конкретизациями *двух разных предикатов,* а все "отношение" является не отношением, а лишь своего рода некоторым объединением (причем нереляционным объединением!) двух разных отношений с двумя разными заголовками (в данном частном случае).

Теперь можно предположить, что сложившуюся ситуацию можно было бы исправить, утверждая, что на самом деле существует всего один предикат, содержащий оператор OR, как показано ниже.

"Существует сотрудник с номером E#, работающий в отделе с номером D#"
ОД *"Существует сотрудник с номером Eё".*

Обратите внимание на то, что теперь (следуя допущению о замкнутости мира) отношение должно содержать "кортеж" вида (Ei,UNK) для всех сотрудников Ei! Страшно даже подумать о результатах обобщения подобной попытки исправления дел на тот случай, когда "отношение", как говорится, "содержит величину UNK" в нескольких разных "атрибутах". (В любом случае, полученное подобным образом "отношение" вовсе не будет являться отношением, что будет показано в следующем абзаце.)

Перефразируем сказанное выше. Если значение данного атрибута в данном кортеже данного отношения "представлено величиной UNK", то (повторим это еще раз) в позиции атрибута на самом деле **вообще ничего не содержится.** А это означает, что данный "атрибут" не является атрибутом, данный "кортеж" не является кортежем, а данное "отношение" не является отношением и все последующие действия с этими объектами (какими бы они ни были) уже не могут быть обоснованы с помощью математически строгой реляционной теории. Иначе говоря, использование величины UNK и трехзначной логики *противоречит самим основам реляционной модели.*

19.4. ОТСУТСТВУЮЩИЕ ЗНАЧЕНИЯ И КЛЮЧИ

Примечание. Далее вместо термина UNK мы будем использовать более традиционную терминологию, т.е. термин NULL.

Вопреки всему сказанному в предыдущем разделе, на практике неопределенные значения (NULL) и трехзначная логика широко поддерживаются в большинстве современных программных продуктов. Резонно предположить, что такая поддержка имеет очень важные последствия, особенно в отношении ключей. Именно это и будет предметом дальнейшего краткого рассмотрения в этом разделе.

Первичные ключи

Как уже отмечалось в разделе 9.10, исторически сложилось так, что в реляционной модели, по крайней мере, в базовых переменных отношения, из всего набора потенциальных ключей в качестве *первичного* ключа переменной отношения должен быть выбран один (и только один) потенциальный ключ. Все остальные потенциальные ключи (если таковые имеются) считаются *альтернативными.* Помимо концепции первичного ключа, в эту модель по некоторым историческим причинам входит следующее *метаограничение,* или правило *(правило поддержки целостности сущности).*

- **Правило поддержки** целостности **сущности.** Ни один компонент первичного ключа любой базовой переменной отношения не может содержать неопределенные значения (NULL).

Обоснование этого ограничения построено на такой последовательности доводов: а) кортежи базовых переменных отношения представляют сущности реального мира; б) сущности реального мира всегда должны допускать возможность их идентификации (по определению); в) следовательно, их аналоги в базе данных также должны допускать возможность идентификации; г) в базе данных значения первичного ключа используются **в** качестве идентификаторов; д) следовательно, значение любого первичного ключа не может быть "неопределенным".

В связи с этим приведем ряд соображений.

1. Прежде всего, часто ошибочно считается, что скрытый между строк смысл ограничения целостности сущности заключается в том, что "значения первичного ключа должны быть уникальными", но дело обстоит иначе. (Действительно, значения первичного ключа должны быть уникальными, но это следует из самого определения первичного ключа.)
2. Из этого следует вывод, что *альтернативные* ключи, по-видимому, могут содержать неопределенные значения (NULL). Но если некий ключ АК является альтернативным ключом, в котором допускаются неопределенные значения (NULL), то он не может использоваться в качестве первичного ключа, поскольку для него нарушается требование сохранения целостности сущности. Тогда в каком смысле ключ АК является потенциальным? И наоборот, если выдвинуть требование, что альтернативные ключи также не могут содержать неопределенные значения (NULL), то правило сохранения целостности сущности будет относиться ко *всем потенциальным ключам,* а не только к первичным ключам. В любом из этих двух вариантов указанное правило выглядит не вполне приемлемым.
3. Наконец, отметим, что правило сохранения целостности сущности применимо только для *базовых переменных отношения,* тогда как другие переменные отношения, по-видимому, могут содержать первичные ключи с неопределенными значениями (NULL). В качестве тривиального и очевидного примера рассмотрим проекцию переменной отношения R по атрибуту А, в котором допускается присутствие неопределенных значений (NULL). Очевидно, что правило сохранения целостности сущности нарушает *принцип взаимозаменяемости* (для базовых и производных переменных отношения). По мнению автора, это может послужить серьезным доводом для отказа от него (данная концепция будет отвергнута нами независимо от включения неопределенных значений (NULL)).

Теперь допустим, что мы отказались от идеи применять неопределенные значения (NULL) и для представления недостающей информации вместо них применили *специальные значения*[6] аналогично тому, как это делается в реальном мире (подробные сведения приведены в разделе 19.6). Тогда модифицированная версия правила сохранения целостности сущности будет иметь следующий вид: "Никакой компонент первичного ключа любой базовой переменной отношения не может содержать подобных специальных значений". Это требование может использоваться как *рекомендация,* но *не является* нерушимым законом (так же, как идеи дальнейшей нормализации служат в качестве рекомендаций, а не строгих законов). На рис. 19.2 приведен пример базовой переменной отношения SURVEY, для которой может потребоваться нарушить эту рекомендацию. В ней представлены результаты опроса сотрудников о размере их зарплаты, которые включают среднее, максимальное и минимальное значения для группы сотрудников с определенным годом рождения. (Здесь атрибут BIRTHYEAR является первичным ключом.) Кортеж со специальным значением "????" атрибута BIRTHYEAR представляет тех служащих, которые отказались ответить на вопрос о годе рождения.

SURVEY

BIRTHYEAR	AVGSAL	MAXSAL	MINSAL
1960	85K	130K	33K
1961	82K	125K	32K
1962	77K	99K	32K
1963	78K	97K	35K
....	...	...	...
1970	29K	35K	12K
????	56K	117K	20K

Рис. 19.2. Пример значений данных в **базовой переменной отношения** SURVEY

Внешние ключи

Еще раз обратимся к базе данных отделов и сотрудников, содержимое которой показано на рис. 19.1. Возможно, вы не обратили на это внимания, однако в свое время автор намеренно не указал, что на данном рисунке атрибут DEPT# переменной отношения EMP является внешним ключом. Предположим теперь, что это так. Сразу становится понятно следующее: требуется уточнить формулировку определения ограничений ссылочной целостности с учетом того, что внешние ключи могут содержать неопределенные значения[7] (NULL), а это очевидно противоречит исходной формулировке данного ограничения, приведенной в главе 9.

- **Ограничение ссылочной целостности** *(исходная формулировка).* База данных не должна содержать никаких несогласованных значений внешних ключей.

В действительности, это определение можно оставить в неизменном виде, расширив соответствующим образом определение термина *несогласованные значения внешнего ключа.* А именно, назовем несогласованным значением внешнего ключа такое его **непустое**

---

[6] Часто необоснованно называемые значениями, заданными по умолчанию **([19.12]).**

[7] Следует отметить, что такие неопределенные значения явились бы нарушением указанного правила, даже если бы в соответствующей переменной отношения, на которую указывает внешний ключ, был кортеж с неопределенным значением соответствующего потенциального ключа!

(т.е. отличное от NULL) значение в ссылающейся на него переменной отношения, для которого не существует согласованного значения потенциального ключа в переменной отношения, на которую сделана ссылка.

Из этого следуют приведенные ниже выводы.

1. Запрещение или разрешение на применение неопределенных значений (NULL) в данном внешнем ключе должно задаваться в определении базы данных. (Безусловно, в действительности это требование справедливо в отношении всех атрибутов в це лом, независимо от того, являются ли они частью некоторого внешнего ключа.)
2. Возможность присутствия неопределенных значений (NULL) во внешних ключах требует введения нового типа ссылочного действия, SET NULL, которое можно будет указывать в правиле удаления DELETE или обновления UPDATE при опреде лении внешних ключей, например, следующим образом.

```
VAR SP BASE RELATION { ... } ...
    FOREIGN KEY { S# } REFERENCES S
                        ON DELETE SET
                       NULL ON UPDATE
                         SET NULL ;
```

При использовании этих спецификаций операция DELETE в переменной отношения поставщиков приведет к помещению неопределенных значений (NULL) в атрибут внешнего ключа всех кортежей с данными о соответствующих поставках, и только после этого сведения об указанных поставщиках будут удалены. Аналогичным образом, операция UPDATE для атрибута s# в переменной отношения поставщиков вызовет помещение неопределенных значений (NULL) в атрибут внешнего ключа всех кортежей с данными о соответствующих поставках, и только после этого сведения об указанных поставщиках будут обновлены. ***Примечание.*** Спецификация SET NULL может быть указана только для тех внешних ключей, в которых допускается наличие неопределенных значений (NULL). 3. Наконец, следует отметить, что "необходимости" разрешить присутствие неопределенных значений (NULL) во внешних ключах вполне можно избежать за счет соответствующего проектирования базы данных [19.19]. Еще раз обратимся к примеру с отделами и сотрудниками. Возможна ситуация, когда некоторые сотрудники действительно не относятся ни к одному из отделов. Но тогда, как уже предполагалось в конце предыдущего раздела, логичнее было бы не включать атрибут номера отдела DEPT# в переменную отношения EMP, а создать отдельную переменную отношения (скажем) ED с атрибутами EMP# и DEPT#, предназначенную для представления того факта, что указанный сотрудник работает в данном отделе. Если некоторый сотрудник не относится ни к одному отделу, ситуацию легко можно представить, не включая кортеж для этого сотрудника из переменной отношения ED.

19.5. ВНЕШНЕЕ СОЕДИНЕНИЕ (ОТСТУПЛЕНИЕ ОТ ОСНОВНОЙ ТЕМЫ)

В этом разделе делается некоторое отступление от основной темы главы в целях обсуждения часто используемой операции **внешнего соединения** [19.3], [19.4], [19.7], [19.14], [19.15]. Внешнее соединение— это расширенная форма обычного или *внутреннего* соединения. Внешнее соединение отличается от внутреннего тем, что кортежи одного из

отношений, не имеющие соответствия среди кортежей другого отношения, появляются в результирующем отношении с неопределенными значениями (NULL) во всех позициях атрибутов второго отношения, вместо того чтобы быть просто проигнорированными, как в обычном соединении. Этот оператор не является примитивным. Например, для построения внешнего соединения по номеру поставщика переменных отношения поставщиков и поставок можно использовать приведенное ниже выражение (в демонстрационных целях предположим, что NULL является допустимым скалярным выражением).

```
 ( S JOIN SP )
  UNION
( EXTEND ( ( S { S# } MINUS SP { S# } )
JOIN S ) j  ADD { NULL AS P#, NULL AS QTY )
)
```

Результат будет содержать кортежи и для поставщиков, не выполнивших ни одной поставки. В этих кортежах будут содержаться неопределенные значения (NULL) в позициях атрибутов P# и QTY.

Исследуем данный пример более подробно. Рассмотрим рис. 19.3. На нем вверху показаны значения данных в исходных переменных отношения S и SP, в средней части показан результат обычного внутреннего соединения, а в нижней части — результат соответствующего внешнего соединения. Как следует из этого рисунка, во внутреннем соединении (выражаясь очень неформально!) "теряется информация" о поставщиках, не выполнивших ни одной поставки деталей (в данном примере это поставщик с номером S5), в то время как внешнее соединение "сохраняет" ее. Действительно, имеющееся различие и является причиной использования внешнего соединения.

Таким образом, внешнее соединение предназначено для решения весьма важной проблемы, заключающейся в потере информации при внутреннем соединении. Некоторые авторы высказывают мнение, что система должна обеспечивать прямую и явную поддержку внешних соединений, вместо того чтобы требовать от пользователя при формировании запроса прибегать к различным ухищрениям для достижения желаемого результата. В частности, в [6.2] приведено мнение о том, что внешние соединения должны быть неотъемлемой частью реляционной модели. Тем не менее, в этой книге автор не поддерживает эту позицию, в частности, по приведенным ниже причинам.

- Прежде всего, безусловно, операция внешнего соединения использует неопреде ленные значения (NULL), а автор настоящей книги всегда выступает против этого по целому ряду описанных ранее основательных причин.
- Кроме того, обратите внимание на то, что существует несколько разновидностей операции внешнего соединения — левое, правое и полное внешнее *в-соединение,* а также левое, правое и полное внешнее *естественное* соединение. (Левое соедине ние сохраняет в результирующем отношении информацию из левого операнда, правое — из правого операнда, а полное — из обоих операндов.) На рис. 19.3 пока зан пример левого соединения, точнее, левого внешнего естественного соедине ния. Более того, обратите внимание на то, что не существует достаточно простого способа формирования внешнего естественного соединения на основе внешнего в-соединения [19.7]. В результате не ясно, какой именно оператор внешнего со единения должна явно поддерживать система.

- Далее, вопрос о внешнем соединении ни в коем случае нельзя считать столь же тривиальным, как можно полагать на основании простого примера, который приведен на рис. 19.3. На практике, как показано в [19.7], операции внешнего соединения присущ ряд "неприятных свойств", которые в своей совокупности не позволяют эффективно реализовать ее поддержку в существующих языках, в частности в языке SQL. Предпринятые в некоторых коммерческих СУБД попытки решить эту проблему оказались неудачными, т.е. в них так и не удалось справиться с "неприятными свойствами" (см. [19.7], где данный вопрос описан подробнее).

S

S#	SNAME	STATUS	CITY
S2	Jones	10	Paris
S5	Adams	30	Athens

SP

S#	P#	QTY
S2	P1	300
S2	P2	400

Обычное (внутреннее) соединение:

S#	SNAME	STATUS	CITY	P#	QTY
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400

"Теряет" информацию о поставщике S5

Внешнее соединение:

S#	SNAME	STATUS	CITY	P#	QTY
S2	Jones	10	Paris	P1	300
S2	Jones	10	Paris	P2	400
S5	Adams	30	Athens	UNK	UNK

"Сохраняет" информацию о поставщике S5

Рис. 19.3. Сравнение внутреннего и внешнего соединений (пример)

- Наконец, уже было высказано мнение о том, что *атрибуты со значениями в виде отношений* представляют собой альтернативный подход, позволяющий радикально устранить все эти проблемы. Такой подход исключает необходимость использования неопределенных значений (NULL) и операций внешнего соединения и в целом, по мнению автора, представляет собой более изящное решение (не говоря уже о том, что это — реляционное решение, которое не приводит к нарушению реляционной модели). Например, для показанных на рис. 19.3 данных приведенное ниже выражение позволяет получить результат, представленный на рис. 19.4.

```
WITH ( S RENAME S# AS X ) AS Y :
     ( EXTEND Y ADD ( SP WHERE S# = X ) AS PQ ) RENAME X AS S#
```

В частности, пустое множество деталей, поставляемых поставщиком с номером S5, на рис. 19.4 представлено именно в виде пустого множества, а не в виде каких-то странных неопределенных значений (NULL), как это было в случае, показанном на рис. 19.3. Представление пустого множества в виде пустого множества, похоже, является удачной идеей. Действительно, при должной поддержке атрибутов, значениями которых могут быть отношения, *операция внешнего соединения может оказаться совершенно излишней.*

При более внимательном изучении этого вопроса можно заметить, что имеет место проблема *интерпретации* неопределенных значений (NULL), которые появляются в результирующем отношении операции внешнего соединения. Например, что представляют собой неопределенные значения, показанные на рис. 19.3? Несомненно, они не означают, что "значение не известно" или "значение не применимо". На самом деле, единственная логически обоснованная интерпретация их смысла состоит в следующем: "Значением является пустое множество". Более подробно эта проблема также обсуждается в [19.7].

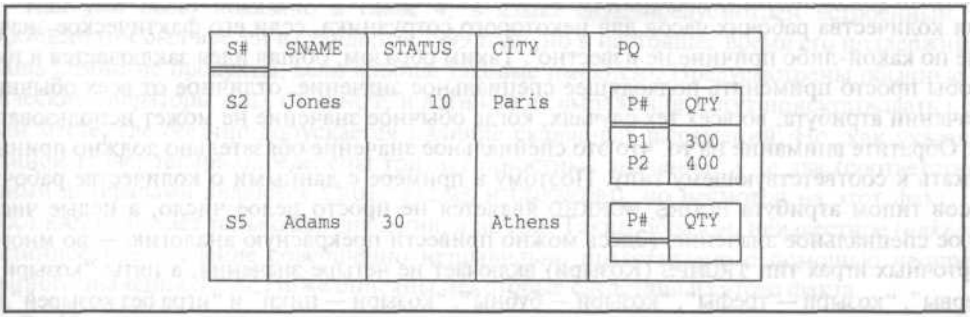


Рис. 19.4. Сохранение информации о поставщике с номером S5 (лучший способ)

В заключение раздела отметим, что существует возможность определить "внешние" версии других операторов реляционной алгебры, в частности операторов объединения, пересечения и разности [14.7]. Например, в [6.2] по меньшей мере один из них *{внешнее объединение)* рассматривается как неотъемлемая часть реляционной модели. Подобные версии операторов позволяют выполнять объединение двух отношений и другие операции, даже если данные отношения не совместимы по типу. Основной принцип работы таких операторов состоит в расширении каждого из операндов за счет добавления атрибутов, свойственных другому операнду, с помещением неопределенных значений (NULL) в каждый такой атрибут каждого кортежа (после чего операнды становятся совместимыми по типу). Затем выполняется обычная операция объединения, пересечения или разности[8]. Тем не менее, мы не будем подробно обсуждать эти операции по приведенным ниже причинам.

- Внешнее пересечение гарантированно возвращает пустое отношение, за исключе нием частного случая, когда исходные отношения сразу совместимы по типу. Но в такой ситуации внешнее пересечение вырождается до обычного пересечгния.
- Внешняя разность гарантированно возвращает первый операнд (дополненный не определенными значениями), за исключением частного случая, когда исходные отношения сразу совместимы по типу. Но в такой ситуации внешняя разность вы рождается до обычной разности.
- *Основными* проблемами, характерными для операции внешнего объединения, яв ляются проблемы интерпретации; причем они даже сложнее проблем интерпрета ции., имеющих место в случае операции внешнего соединения.. *(См..[1*.9.2, для. более детального изучения.)

[8] Данное объяснение относится к исходным определениям этих операций в [14.6]; в работе [6.2] эти определения немного изменились.

19.6. СПЕЦИАЛЬНЫЕ ЗНАЧЕНИЯ

Как показано выше, введение неопределенных значений (NULL) приводит к разрушению реляционной модели, которая великолепно обходилась без них в течение десяти лет с момента ее создания в 1969 году [6.1] и вплоть до введения этих значений в 1979 году [14.7].

Теперь предположим, как показано в разделе 19.4, что понятие *неопределенных значений* в контексте представления отсутствующей информации заменено понятием *специальных значений.* Следует отметить, что в реальном мире мы обычно пользуемся именно специальными значениями. Например, специальное значение "?" используется для обозначения количества рабочих часов для некоторого сотрудника, если его фактическое значение по какой-либо причине не известно[9]. Таким образом, общая идея заключается в том, чтобы просто применять подходящее специальное значение, отличное от всех обычных значений атрибута, во всех тех случаях, когда обычное значение не может использоваться. Обратите внимание на то, что это специальное значение обязательно должно принадлежать к соответствующему типу. Поэтому в примере с данными о количестве рабочих часов типом атрибута HOURS_WORKED является не просто целое число, а целые числа плюс специальное значение. (Здесь можно привести прекрасную аналогию — во многих карточных играх тип TRUMPS (Козыри) включает не четыре значения, а пять: "козыри — червы", "козыри — трефы", "козыри — бубны", "козыри — пики" и "игра без козырей".)

Безусловно, следует признать, что изложенная выше схема не очень изящна, но она обладает явным преимуществом, поскольку *не подрывает логических основ реляционной модели.* В остальной части этой книги мы будем просто игнорировать возможность поддержки неопределенных значений (NULL), за исключением некоторых случаев, характерных для контекста языка SQL, когда те или иные ссылки на неопределенные значения (NULL) будут неизбежны. Более подробно вопросы об использовании схемы, основанной на специальных значениях, рассматриваются в **[19.12].**

19.7. СРЕДСТВА ЯЗЫКА SQL

Поддержка неопределенных значений (NULL) и трехзначной логики в языке SQL отражает весь широкий спектр подходов, описанных в предыдущих разделах. Так, например, когда в языке SQL условие WHERE применяется к некоторой таблице т, при этом исключаются из рассмотрения все строки таблицы т, для которых указанное в конструкции WHERE выражение принимает значение *false* или *unk* (т.е. не *true).* Аналогичным образом, когда к результату выполнения некоторой операции группирования, представленному таблицей G, применяется конструкция HAVING, из дальнейшего рассмотрения исключаются все группы кортежей таблицы G, для которых указанное в конструкции HAVING выражение принимает значение *false* или *unk* (т.е. не *true)*[10]. Из этого следует, что мы просто обратили внимание читателя на некоторые средства поддержки трехзначной логики, характерные для языка SQL как такового, а не являющиеся неотъемлемой частью описанного выше подхода, основанного на использовании трехзначной лэгики.

---

[9] Обратите внимание, что для этого *не* используется неопределенное значение (NULL). В реальном мире никакого понятия неопределенного значения вообще не существует ([19.12]).

[D] В языке SQL сгруппированной называется таблица, которая создается в результате выполнения конструкции GROUP BY (возможно, неявной). А после выполнения соответствующего оператора SELECT такая таблица сводится к обычной, "несгруппированной" таблице.

Примечание. Все последствия поддержки неопределенных значений (NULL) в языке SQL оценить очень сложно; фактически, хотя уже было сказано, что в языке SQL в целом поддерживается трехзначная логика, истина состоит в том, что в вопросах этой поддержки допущены многочисленные ошибки, как вскоре будет показано. Дополнительную информацию можно найти в официальных документах стандарта SQL [4.23] и в учебном издании [4.20].

Типы данных

Как уже было показано в главе 4, в языке SQL предусмотрен встроенный тип BOOLEAN (он был введен в стандарт в 1999 году, но в настоящее время его поддерживают лишь немногие продукты, если вообще таковые имеются). Предусмотрены обычные логические операторы AND, OR и NOT, и логические выражения могут присутствовать в любом месте, где обычно допускается наличие скалярных выражений. Но, как указано в данной главе, теперь определены три истинностных значения, а не два (соответствующими литералами являются TRUE, FALSE и UNKNOWN), но несмотря на этот факт, тип BOOLEAN включает только два значения, а не три. Таким образом, неизвестное *{unknown)* истинностное значение совершенно неправильно представлено с помощью неопределенного значения NULL! Ниже описаны некоторые следствия из этого факта.

- Присваивание значения UNKNOWN переменной в типа BOOLEAN фактически равно сильно присваиванию этой переменной неопределенного значения.
- После такого присваивания операция сравнения в = UNKNOWN не дает в результа те *true* (или, скорее, TRUE); вместо этого результатом становится неопределенное значение.
- В действительности, операция сравнения в = UNKNOWN всегда приводит к полу чению неопределенного значения, независимо от значения в! Дело в том, что та кая операция логически эквивалентна операции сравнения "в = NULL" (которая не рассматривается как синтаксически допустимая).

Для того чтобы понять серьезность указанных недостатков, необходимо обратиться к аналогии с числовым типом, в котором для представления нуля использовалось бы неопределенное значение вместо нулевого.

Теперь предположим, что т — нескалярный тип или структурированный тип (и в данном контексте не играет роли, рассматриваются ли структурированные типы как скалярные или нескалярные). Чтобы уточнить описание, предположим, что т представляет собой именно строковый тип, а переменная V является переменной типа т. В таком случае, безусловно, существует логическое различие между самой переменной V, имеющей неопределенное значение, и переменной V, имеющей по меньшей мере один компонент (т.е. поле), который содержит неопределенное значение. Действительно, сама переменная V не обязательно имеет неопределенное значение", даже если неопределенными являются все ее компоненты! Тем не менее, по-видимому, обоснованным является утверждение (хотя в стандарте нет явных указаний на этот счет), что если значение v не определено, то все компоненты этой переменной также рассматриваются как имеющие неопределенное значение. Поэтому, если значение V не является неопределенным, но эта

[11] В действительности, в языке SQL и эта ситуация трактуется неправильно; см. описание конструкции IS [NOT] NULL в подразделе "Логические выражения".

переменная содержит по меньшей мере один неопределенный компонент, то в результате сравнения V = V будет получено неопределенное значение и, тем не менее, выражение V IS NULL будет равно FALSE. Но, в общем, можно хотя бы утверждать, что если выражение ((V = V) is NOT TRUE) is TRUE равно TRUE, то переменная vявляется либо неопределенной, либо содержит неопределенный компонент.

Базовые таблицы

Как описано в разделе 6.6 главы 6, для столбцов в базовых таблицах обычно предусмотрены соответствующие значения, применяемые *по умолчанию;* они часто определяются (явно или неявно), как неопределенные значения NULL. Более того, столбцы в таблицах всегда *разрешают* использование неопределенных значений, если условие запрета их использования не будет указано явно (например, в виде фразы NOT NULL).

Из указанного выше следует, что если мы твердо придерживаемся декларируемых принципов, то действительно обязаны задавать ключевое слово NOT NULL, явно или неявно, для каждого столбца базовый таблицы в каждом примере SQL, приведенном выше в данной книге. Но, по меньшей мере, это требование будет соблюдаться во всех примерах SQL начиная с этого момента. Однако следует отметить, что применение ключевого слова NOT NULL неявно подразумевается для любого столбца, указанного в спецификации PRIMARY KEY.

Табличные выражения

Напомним замечание из главы 8 (раздел 8.7) о том, что явная поддержка операции JOIN была введена в стандарт языка SQL в 1992 году. Более того, если перед ключевым словом JOIN указан один из префиксов LEFT, RIGHT или FULL (С необязательным ключевым словом OUTER в каждом случае), рассматриваемое соединение является *внешним.* Ниже приведено несколько примеров.

```
S LEFT JOIN SP ON S.S# =
SP.S# S LEFT JOIN SP USING (
S# ) S LEFT NATURAL JOIN SP
```

Три приведенных выражения эквивалентны, но первое приводит к созданию таблицы с двумя идентичными столбцами s#, а второе и третье — таблицы с одним таким столбцом.

Язык SQL поддерживает также аппроксимацию внешнего объединения, которую называют *объединяющим соединением* (эта операция была введена в спецификации SQL: 1992, но должна быть удалена в SQL:2003). Однако подробное обсуждение этого вопроса выходит за рамки данной главы.

Логические выражения

Не удивительно то, что в языке SQL логические выражения испытывают сильное влияние неопределенных значений и трехзначной логики. Остановимся на рассмотрении лишь нескольких наиболее важных частных случаев, которые описаны ниже.

- *Проверка наличия неопределенного значения.* В языке SQL предусмотрены два специальных оператора сравнения, IS NULL и IS NOT NULL, предназначенных для проверки наличия или отсутствия неопределенных значений. Синтаксис использования этих операторов показан ниже.

<row value constructor IS [NOT] NULL

Если в выражении *<row value constructor* с определением конструктора значения строки "конструируется" строка со степенью один, то в языке SQL это выражение рассматривается так, как будто оно фактически указывает на значение, содержащееся в этой строке, а не на саму строку как таковую; в противном случае это выражение рассматривается как указывающее на строку. Но в последнем случае считается, что строка имеет неопределенное значение тогда и только тогда, когда каждый ее компонент является неопределенным, а не имеет неопределенного значения — тогда и только тогда, когда каждый ее компонент имеет значение, отличное от неопределенного! Одним следствием этой ошибки является то, что если r — строка с двумя компонентами, то выражения r IS NOT NULL и NOT (r IS NULL) *не являются* эквивалентными; первое из них эквивалентно выражению c1 is NOT NULL AND c2 is NOT NULL, а второе — выражению c 1 IS NOT NULL OR c2 IS NOT NULL. Еще одним следствием из указанного является то, что если строка r одновременно включает и компоненты с неопределенными, и компоненты с определенными значениями, то сама строка r, безусловно, не может рассматриваться ни как имеющая неопределенное значение, ни как имеющая определенное значение.

- *Проверка наличия значений* true, false *и* unknown. Если p — это заключенное в круглые скобки логическое выражение (фактически в некоторых случаях можно обойтись без круглых скобок, но они никогда не помешают), то следующие операторы также являются логическими выражениями.

```
p IS [ NOT ] TRUE p IS
[ NOT ] FALSE p IS [
NOT ] UNKNOWN
```

Значения этих выражений показаны в приведенной ниже таблице истинности.

P	*true*	*false*	*unk*
p IS TRUE	*true*	*false*	*false*
p IS NOT TRUE	*false*	*true*	*true*
p IS FALSE	*false*	*true*	*false*
p IS NOT FALSE	*true*	*false*	*true*
p IS UNKNOWN	*false*	*false*	*true*
p IS NOT UNKNOWN	*true*	*true*	*false*

Обратите внимание на то, что выражения p IS NOT TRUE и NOT p не являются эквивалентными.

Примечание. Выражение p IS UNKNOWN соответствует описанному выше оператору MAYBE (p). Если считать, что в языке SQL для представления значения *unk* используются неопределенные значения, то данное выражение эквивалентно также выражению p IS NULL.

- Условия EXISTS. Оператор EXISTS языка SQL не является аналогичным квантору существования в трехзначной логике, поскольку он всегда возвращает значение *true* или *false,* но не *unk,* даже если *unk* — логически правильный ответ. А именно, этот оператор возвращает *false,* если таблица, заданная в нем в качестве

фактического параметра, пуста, и true — в ином случае (Поэтому он иногда возвращает *true,* тогда как *unк*— логически правильный ответ). Дополнительная информация на эту тему приведена в [19.6].

- Условия UNIQUE. Неформально говоря, условия UNIQUE служат для проверки того, что указанная таблица не содержит дубликатов строк (именно так!). Точнее, выражение UNIQUE (< *table exp>)* возвращает true, если таблица, обозначенная с помощью параметра < *table exp>,* не включает двух разных строк, скажем, rl и r2, таких что операция сравнения rl = r2 возвращает true; в противном случае данное выражение возвращает *false.* Поэтому UNIQUE, как и EXISTS, иногда возвращает true, даже когда *unк* — логически правильный ответ.
- Условия DISTINCT. Условия DISTINCT, вообще говоря, предназначены для проверки того, не являются ли две строки дубликатами друг друга. Обозначим две рассматриваемые строки как Left и Right; строки Left и Right должны иметь одинаковую степень, скажем, п. Допустим, что i-ми компонентами Left и Right, соответственно, являются Li и Ri (i = 1, 2, . . ., п); все компоненты Li и Ri должны быть такими, что операция сравнения Li = Ri является действительной. В таком случае следующее выражение

  ```
  Left IS DISTINCT FROM Right
  ```

  возвращает *false,* если для всех i либо операция сравнения "Li = Ri" возвращает true, либо оба значения, Li и Ri, являются неопределенными; в противном случае указанное выражение возвращает true.

Другие скалярные выражения

И снова рассмотрим лишь несколько важных частных случаев, которые описаны ниже.

- "Литералы". Ключевое слово NULL может иногда использоваться как своего рода литеральное представление неопределенного значения (например, в операторе INSERT), но не во всех контекстах; в стандарте SQL указано, что "ключевое слово NULL ... может применяться для обозначения неопределенного значения... в некоторых контекстах, [но] повсеместное использование этого литерала не допускается" [4.23]. В частности, следует отметить, что ключевое слово NULL нельзя применять для обозначения операнда простого оператора сравнения; например, выражение "WHERE x = NULL" является .недопустимым (правильная форма, безусловно, WHERE X IS NULL).
- COALESCE. Оператор COALESCE — это аналог предложенного автором оператора IF_UNK в языке SQL. Точнее, выражение COALESCE (x,y, . . . , z) возвращает неопределенное значение, если все его параметры x, y, . . ., z возвращают неопределенное значение; в противном случае оно возвращает первый свой операнд, отличный от неопределенного значения.
- Агрегирующие операторы. Агрегирующие операторы SQL (SUM, AVG и т.д.) не действуют в соответствии с правилами для скалярных операторов, описанными в разделе 19.2, а вместо этого просто игнорируют все неопределенные значения своих фактических параметров (не считая выражения COUNT (*), в котором неопределенные значения рассматриваются так, как если бы они были определенными

значениями). Кроме того, если оказалось, что фактический параметр такого оператора соответствует пустому множеству, то оператор COUNT возвращает нуль, а все остальные операторы — неопределенное значение. (Как было указано в главе 8, такая организация функционирования логически не обоснована, но язык SQL определен именно так.)

- "Скалярные подзапросы". Если скалярное выражение фактически представляет собой табличное выражение, заключенное в круглые скобки, например, (SELECT S.CITY FROM S WHERE S.S# = S#('S1')), ТО В обыЧНЫХ УСЛОВИЯХ ЭТО Таб-

  личное выражение должно давать в результате таблицу, содержащую точно один столбец и точно одну строку. Поэтому значением этого скалярного выражения должно служить единственное скалярное значение, содержащееся внутри этой таблицы. Но если результатом вычисления данного табличного выражения становится таблица с одним столбцом, вообще не содержащая строк, то в языке SQL значение скалярного выражения задается как неопределенное.

Ключи

Различные варианты взаимодействия между неопределенными значениями и ключами в языке SQL можно в целом описать, как показано ниже.

- Потенциальные ключи. Допустим, что с — столбец, который является компонентом некоторого потенциального ключа к некоторой базовой таблицы. Как было отмече но раньше в этом разделе, если к — первичный ключ, то согласно спецификации SQL наличие каких-либо неопределенных значений в столбце с не допускается (иными словами, спецификация SQL требует соблюдения правила целостности сущности). А если к не является первичным ключом, то спецификация SQL до пускает наличие в столбце С любого количества неопределенных значений (но, ра зумеется, при условии, что никакие две разные строки не содержат одинакового

В связи со сказанным, автор предлагает поразмыслить над приведенной ниже немного отредактированной цитатой из [4.20].

"Допустим, что к2 — новое значение для К, которое определенный пользователь пытается ввести с помощью операции INSERT или UPDATE... Эта операция INSERT или UPDATE будет отвергнута, если к2 совпадает с некоторым другим значением К, скажем, kl, которое уже существует в таблице... Но что подразумевается под тем, что два значения, kl и к2, совпадают ? Оказывается, что никакие два из следующих трех утверждений не являются эквивалентными:

1. kl и к2 являются одинаковыми с точки зрения операции сравнения;
2. kl и к2 являются одинаковыми с точки зрения обеспечения уникальности потен циальных ключей;
3. kl и к2 являются одинаковыми с точки зрения устранения дубликатов.

Утверждение 1 определено в соответствии с правилами трехзначной логики, утверждение 2 определено в соответствии с правилами для условия UNIQUE, а утверждение 3 сформулировано в соответствии с определением дубликатов, приведенном в разделе 19.2. В частности, если оба значения, kl и к2, являются неопределенными, то проверка,

выполняемая в соответствии с утверждением 1, приводит к получению unk, *с утверждением 2* — false *и с утверждением 3* — true".

- Внешние ключи. Правила, определяющие, что подразумевается под совпадением данного значения внешнего ключа с соответствующем ему значением потенциального ключа в присутствии неопределенных значений, являются весьма сложными. В этом разделе подробные сведения об этом не рассматриваются. *Примечание.* Наличие неопределенных значений отражено также в определениях ссылочных действий (CASCADE, SET NULL и т.д.), указанных в конструкциях ON DELETE и ON UPDATE. (Кроме того, поддерживается ключевое слово SET DEFAULT, имеющее очевидную интерпретацию.) И в этом случае применяемые правила являются весьма сложными и их описание выходит за рамки данной книги; дополнительные сведения приведены в [4.20].

Внедренные операторы SQL

- *Индикаторные переменные.* Рассмотрим следующий пример внедренного выражения SQL. (Он уже рассматривался в главе 4 при описании однострочного оператора SELECT.)

```
EXEC SQL SELECT STATUS, CITY INTO
         :RANK, :TOWN FROM S WHERE
         S# = S# ( :GIVENS# ) ;
```

Предположим, что для некоторых поставщиков атрибут STATUS может содержать неопределенное значение. Тогда выполнение приведенного выше оператора SELECT завершится неудачей, если в выбранном кортеже атрибут STATUS будет содержать неопределенное значение (в переменную SQLSTATE будет помещен код ошибки 22 002). В общем случае, когда существует вероятность, что выбираемое значение может оказаться неопределенным, для данного атрибута пользователь должен указать, помимо целевой, еще и *индикаторную переменную.* Пример определения индикаторной переменной приведен ниже.

```
EXEC SQL SELECT STATUS, CITY
         INTO :RANK INDICATOR :RANKIND, :TOWN
         FROM S
         WHERE S# = S# ( :GIVENS# ) ;
IF RANKIND = -1 THEN /* Значение STATUS было неопределенным */
... ; END IF ;
```

Если значение, которое нужно выбрать, является неопределенным и для данного атрибута задана индикаторная переменная, то в эту переменную будет помещено значение -1 (значение, помещаемое в целевую переменную, зависит от реализации).

- *Упорядочение.* Для упорядочения строк, полученных в результате вычисления таблично выражения, в определении курсора используется конструкция ORDER BY. (Безусловно, она может использоваться и при вводе интерактивных запросов.) Возникает вопрос: "Каков относительный порядок двух скалярных значений A и B, если либо A является неопределенным значением, либо в является неопределенным значением, либо оба они одновременно являются неопределенными значениями?". В стандарте языка SQL на этот вопрос даются приведенные ниже ответы.

1. В процессе упорядочения все неопределенные значения считаются равными одно другому.
2. В процессе упорядочения все неопределенные значения считаются *либо* боль шими всех определенных значений, *либо* меньшими всех определенных значе ний (реальный выбор одной из этих возможностей зависит от конкретной реа лизации).

19.8. РЕЗЮМЕ

В этой главе обсуждалась проблема **отсутствующей информации,** а также выбранный в настоящее время (хотя и очень неудачный) подход к ее решению, базирующийся на использовании **неопределенных значений** и **трехзначной логики.** Было показано, что неопределенное значение на самом деле значением не является, хотя так и принято говорить (например, говорят, что некоторый атрибут кортежа содержит "значение NULL"). Результатом любых операций сравнения, в которых один из операндов содержит неопределенное значение, служит *третье* истинностное значение, *unknown* (сокращенно — *unk),* поэтому такая логика называется *трехзначной.* Кроме того, отмечалось, что, по крайней мере, концептуально может существовать много разных видов неопределенных значений, в частности, в качестве удобного (и явного) сокращения для той разновидности неопределенных значений, когда значение неизвестно, было введено сокращение UNK.

Далее исследовалось влияние использования величины UNK и трехзначной логики на вычисление **логических операторов AND, OR** и **NOT** (а также MAYBE), кванторов **EXISTS** и **FORALL,** других скалярных операторов, **реляционных операторов и операторов** обновления INSERT и UPDATE. В этой главе были представлены оператор **IS_UNK** (проверяющий наличие величины UNK), оператор **IF_UNK** (для преобразования величины UNK в значение, отличное от UNK). Дополнительно обсуждалась проблема **дубликатов кортежей** с учетом присутствия величины UNK. Кроме того, было особо подчеркнуто, что величина UNK и логическое значение *unk* — это не одно и то же.

Затем рассматривались некоторые следствия изложенных идей. Было показано, что **некоторые эквивалентности двухзначной логики не являются эквивалентностями** в трехзначной логике. В результате пользователи и программы-оптимизаторы СУБД могут совершать **ошибки в процессе преобразования выражений.** Но даже без учета названных ошибок трехзначная логика обладает одним очень серьезным недостатком — она **не соответствует реальному миру,** т.е. результаты операций в трехзначной логике не всегда являются правильными в реальном мире.

Затем были описаны последствия присутствия неопределенных значений в **первичных** и **внешних ключах** (в частности, упоминались правила поддержки целостности **сущностей** и пересмотренное правило поддержки **ссылочной** целостности). Было сделано отступление для описания **внешнего соединения.** Отмечалось, что автор не является сторонником прямой поддержки этой операции (по крайней мере, в ее обычном смысле), так как считает, что существует лучшее решение той проблемы, которую призвано решать внешнее соединение. В частности, автор предлагает использовать подход на основе *атрибутов со значениями в виде отношений.* Также кратко рассматривалась возможность существования и других "внешних" операций, в частности операции **внешнего объединения.**

В данной главе проанализирована поддержка изложенных идей в стандарте языка SQL. В трактовке проблемы отсутствия информации в стандарте языка SQL широко

используется трехзначная логика, но эта трактовка вносит множество осложнений, описание которых выходит за рамки данной книги. И действительно, в дополнение к собственным недостаткам трехзначной логики ([19.6], [19.10]) стандарт языка SQL вносит свои недостатки. Более того, эти дополнительные недостатки, по сути, выступают как факторы замедления процесса оптимизации (более подробно это обсуждается в разделе 18.8 главы 18).

В заключение попытаемся кратко сформулировать некоторые дополнительные замечания.

- Читатель, безусловно, согласится, что проблемы, связанные с использованием неопределенных значений и трехзначной логики, описаны в книге несколько сжато. Тем не менее, здесь представлен достаточно большой объем материала, позволяющего убедиться в том, что "преимущества", достигаемые при использовании трехзначной логики, более чем сомнительны.
- Следует также отметить, что даже если читатель не слишком озабочен проблемами, связанными с использованием трехзначной логики, ему все же рекомендуется избегать применения соответствующих возможностей языка SQL из-за упоминавшихся выше "дополнительных недостатков".
- Исходя из всего сказанного, автор рекомендует пользователям СУБД полностью игнорировать все средства поддержки трехзначной логики, предоставленные разработчиком выбранного им продукта. Вместо них целесообразно использовать более строгий механизм "специальных значений" (благодаря своей строгости остающийся в рамках двухзначной логики). Подобная схема подробно описана в [19.12].
- Наконец, повторим важное утверждение, сформулированное в разделе 19.3. Говоря весьма неформально, если значение данного атрибута данного кортежа данного отношения "является неопределенным", то в этой позиции атрибута вообще ничего не содержится. Это означает, что данный "атрибут" уже не является атрибутом, что этот "кортеж" уже не является кортежем, что это "отношение" уже не является отношением и для обоснования наших дальнейших действий (какими бы они ни были) использовать математически строгую реляционную теорию уже нельзя.

УПРАЖНЕНИЯ

19.1. Укажите логические результаты вычисления следующих выражений, если A = 6, B = 5, C = 4 и D содержит величину UNK.

```
а) A = B OR ( B > C AND A > D ).
б) A > B AND ( B < C OR IS_UNK ( A - D ) ).
в) A < C OR B < C OR NOT ( A = C ).
г) B < D OR B = D OR B > D.
д) MAYBE ( A > B AND B > C ) .
е) MAYBE ( IS_UNK ( D ) ).
ж) MAYBE ( IS_UNK ( A + B ) ).
з) IF_UNK ( D, A ) > B AND IF_UNK ( C, D ) < B.
```

19.2. Отношение r содержит следующие кортежи.

```
(    6,    5,    4  )
( UNK,     5,    4  )
```

```
( 6, UNK, 4 )
( UNK, UNK, 4 )
( UNK, UNK, UNK )
```

Как было сказано выше в этой главе, предположим, что три атрибута в указанном порядке слева направо называются A, в и с соответственно и каждый атрибут имеет тип INTEGER. Укажите логические результаты вычисления следующих выражений, где v — переменная диапазона, принимающая значения из отношения r.

```
а) EXISTS V ( V.B > 5 ).
б) EXISTS V ( V.B > 2 AND V.C > 5 ).
в) EXISTS V ( MAYBE ( V.C > 3 ) ).
Г) EXISTS V ( MAYBE ( IS_UNK ( V.C ) ) ).
Д) FORALL V ( V.A > 1 ).
е) FORALL V ( V.B > 1 OR IS_UNK ( V.B ) ).
Ж) FORALL V ( MAYBE ( V.A > V.B ) ).
```

19.3. Строго говоря, оператор is_UNK не нужен. Почему?

19.4. В [14.6] Кодд предложил "maybe"-версии для некоторых операторов реляционной алгебры. Например, *maybe-сокращение* отличается от обычного оператора сокращения тем, что возвращает кортежи, для которых условие сокращения принимает значение *unk,* а не *true.* Тем не менее, строго говоря, подобные операторы не нужны. Почему?

19.5. В двухзначной логике существует только два логических значения — *true* и *false.* Следствием этого является существование ровно четырех возможных *унарных* логических операторов: один преобразует оба значения в *true,* второй преобразует оба значения в *false,* третий преобразует *true* в *false* и наоборот (т.е. это оператор NOT), а четвертый сохраняет значение операнда неизменным. Кроме того, существует ровно 16 возможных бинарных логических операторов (см. таблицу истинности, приведенную ниже).

A	B															
t	t	t	t	t	t	t	t	t	t	f	f	f	f	f	f	f	f
t	f	t	t	t	t	f	f	f	f	t	t	t	t	f	f	f	f
f	t	t	t	f	f	t	t	f	f	t	t	f	f	t	t	f	f
f	f	t	f	t	f	t	f	t	f	t	f	t	f	t	f	t	f

Докажите, что все четыре унарных оператора и все шестнадцать бинарных операторов можно записать с помощью комбинаций операторов NOT, AND и OR (следовательно, нет необходимости явно поддерживать все двадцать операторов).

19.6. Сколько логических операторов может быть в трехзначной логике? А в четырехзначной? А для более общего случая — в w-значной логике?

19.7. Ниже приведена таблица истинности для оператора "НЕ-ИЛИ" двухзначной логики (который называется также штрихом Шеффера и обычно записывается в виде одного вертикального штриха, "|").

	t	f
t	f	f
f	f	t

Как показывает эта таблица истинности, выражение p | q эквивалентно выражению NOT p AND NOT q (поэтому оператор "НЕ-ИЛИ" можно рассматривать как оператор "ни ... ни ..." — он принимает истинное значение, если "ни первый операнд, ни второй операнд не являются истинными"). Покажите, что с помощью этого оператора можно сформулировать все 20 операторов двухзначной логики. *Примечание.* Таким образом, оператор "НЕ—ИЛИ" является "порождающим" оператором для всей двухзначной логики.

Можете ли вы найти оператор, который выполняет аналогичную функцию для трехзначной логики? Для четырехзначной логики? Для n-значной логики?

19.8. (Взято из [19.5].) На рис. 19.5 приведены примеры данных для немного измененной базы данных поставщиков и деталей. Этот вариант отличается от предыдущих тем, что в отношение SP включен атрибут SHIP# (номер поставки), а атрибут P# в этом отношении определен как разрешающий использование величины UNK. (Отношение p в данном примере не используется и в упражнении не описывается.)

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London

SP

SHIP#	S#	P#	QTY
SHIP1	S1	P1	300
SHIP2	S2	P2	200
SHIP3	S3	UNK	400

Рис. 19.5. Вариант базы данных поставщиков и деталей Рассмотрим

следующее выражение реляционного исчисления.

```
S WHERE NOT EXISTS SP ( SP.S# = S.S# AND
                        SP.P# = P# ('P2') )
```

Здесь s и SP — это неявные переменные области значений. Какая из следующих интерпретаций этого запроса будет правильной (если таковая вообще существует)?

а) Получить сведения о поставщиках, которые не поставляют деталь с номером P2.

б) Получить сведения о поставщиках, о которых не известно, поставляют ли они деталь с номером P2.

в) Получить сведения о поставщиках, о которых известно, что они не поставляют деталь с номером P2.

г) Получить сведения о поставщиках, о которых либо известно, либо не известно, что они поставляют деталь с номером P2.

19.9. Спроектируйте схему физического представления базовых таблиц SQL, допускающих использование неопределенных значений.

19.10. Сформулируйте определения операторов EXISTS, UNIQUE И IS DISTINCT FROM языка SQL. Являются какие-либо из этих операторов примитивными, в том смысле, что их нельзя выразить в терминах других операторов? Существует ли оператор IS

NOT DISTINCT FROM? Приведите пример запроса, в котором применяются следующие операторы и вырабатывается "неправильный" ответ:

а) EXISTS;

б) UNIQUE.

СПИСОК ЛИТЕРАТУРЫ

19.1. Codd E.F. and Date C.J. Much Ado About Nothing // C.J. Date. Relational Database Writings 1991-1994. - Reading, Mass.: Addison-Wesley, 1995.

Вероятно, именно Кодд является наиболее известным сторонником использования неопределенных значений и трехзначной логики в качестве основы для обработки отсутствующей информации (причем особенно странно то, что неопределенные значения нарушают информационный принцип, сформулированный самим Коддом!). В этой статье содержится текст дискуссии по этому поводу между Коддом и автором данной книги. В ней имеется следующее превосходное замечание: "Управление базой данных существенно упростилось бы, если бы отсутствующие значения не существовали" (Кодд).

19.2. Darwen H. Into the Unknown // C.J. Date. Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

В этой работе анализируется ряд дополнительных вопросов в отношении использования неопределенных значений и трехзначной логики, наиболее сложным из которых является следующий: "Если (как показано в разделе 6.4 главы 6) отношение TABLE_DEE соответствует значению *true,* а отношение TABLE_DUM - значению *false,* и если отношения TABLE_DEE И TABLE_DUM ЯВЛЯЮТСЯ единственными возможными отношениями степени «нуль», то что же соответствует значению *unк?".*

19.3. Darwen H. Outer Join with No Nulls and Fewer Tears // C.J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

В работе предложен простой вариант внешнего соединения, в котором не используются неопределенные значения и решается много других задач, для решения которых предназначены внешние соединения. См. также [3.3].

19.4. Date C.J. The Outer Join // C.J. Date. Relational Database: Selected Writings. — Reading, Mass.: Addison-Wesley, 1986.

Здесь подробно обсуждается проблема внешнего соединения и приводятся некоторые предложения по поддержке этой операции в таких реляционных языках, как SQL.

19.5. Date C.J. NOT is Not "Not"! (Notes on Three-Valued Logic and Related Matters) // C.J. Date. Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

Предположим, что x — это переменная типа BOOLEAN. В этом случае в системе трехзначной логики X должна иметь одно из значений *true, false* или *unк.* Тогда утверждение "X не равно *true"* означает, что значением X является либо значение *unк,* либо значение *false.* В противоположность этому, выражение X = NOT *true* означает, что значением X является *false* (см. таблицу истинности для операции NOT). Таким образом, действие операции NOT в языке трехзначной логики не соответствует частице "не" в обычном языке... Этот факт уже привел в замешательство

многих (включая разработчиков стандарта языка SQL) и, несомненно, будет продолжать удивлять.

19.6. Date C.J. EXISTS is Not "Exist"! (Some Logical Flaws in SQL) // C.J. Date. Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

19.7. Date C.J. Watch Out for Outer Join // C.J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

В разделе 19.5 настоящей главы упоминается тот факт, что внешнее соединение обладает несколькими "неприятными свойствами". В этой работе приводится перечень указанных свойств.

1. Внешнее в-соединение не является выборкой из результата декартова произве дения.
2. Выборка не распространяется на результаты операции внешнего 0-соединения.
3. В трехзначной логике выражение A < в — это не то же самое, что выражение A < B OR A = в (в контексте внешнего соединения).
4. В трехзначной логике операции 9-сравнения не транзитивны.
5. Внешнее естественное соединение не является проекцией внешнего соедине ния по эквивалентности.

В статье рассматривается также влияние, оказанное на конструкции SELECT-FROM-WHERE в результате введения в язык SQL поддержки внешних соединений. В ней показано, что из указанных выше "неприятных свойств" следуют перечисленные ниже нарушения.

1. Операция расширения в конструкции WHERE не работает.
2. Оператор AND, связывающий операции внешнего соединения и выборки, не работает.
3. Попытка применения условия соединения в конструкции WHERE оказывается неудачной.
4. Внешние соединения из более чем двух отношений не могут быть сформулиро ваны без использования вложенных выражений.
5. Операция расширения в конструкции SELECT (отдельно взятой) не работает.

В этой статье также показано, что во многих существующих программных продуктах содержатся ошибки, вызванные указанными выше нарушениями.

19.8. Date C.J. Composite Foreign Keys and Nulls // C.J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

В этой статье рассматривается возможность полного и частичного присутствия неопределенных значений в составных внешних ключах.

19.9. Date C.J. Three-Valued Logic and the Real World // C.J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

19.10. Date C.J. Oh No Not Nulls Again // C.J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

В этой статье приводится дополнительная информация о неопределенных значениях.

19.11. Date C.J. A Note on the Logical Operators Of SQL // Relational Database Writings 1991-1994. — Reading, Mass.: Addison-Wesley, 1995.

В трехзначной логике (3VL) используются три истинностных значения — *true, false* и *unknown* (обозначенные здесь как *t,, f* и *u,* соответственно). Следовательно, в трехзначной логике может быть 3 * 3 * 3 = 27 возможных унарных операторов, поскольку каждое из трех возможных входных значений *t,f* и *u* может отображаться на любое из трех возможных выходных значений *t,f* и *u*. По той же причине в трехзначной логике может быть 3^9=19 683 возможных бинарных оператора (см. таблицу, приведенную ниже).

	T	u	f
t	t/u/f	t/u/f	t/u/f
u	t/u/f	t/u/f	t/u/f
f	t/u/f	t/u/f	t/u/f

Фактически в более общем случае n-значная логика (nVL) включает *n* в степени *n* унарных операторов и и в степени и^2 бинарных операторов (см. таблицу, приведенную ниже).

Логика	Унарные операторы	Бинарные операторы
2VL	4	16
3VL	27	19 683
4VL	256	4 294 967 296
...	.....	.....
*n*VL	(*n*)**(*n*)	(*n*)**(*n*^{2})

Поэтому для любой n-значной логики, где я > 2, возникают приведенные ниже вопросы.

- Каков наиболее подходящий набор *примитивных* операторов? (Например, под ходящими примитивными наборами для двухзначной логики являются и мно жество операторов {NOT, AND}, и множество операторов {NOT, OR}.)
- Каков подходящий набор *полезных* операторов? (Например, множество {NOT, AND, OR} — подходящий полезный набор для двухзначной логики.)

В [19.11] показано, что стандарт SQL (при *очень* вольной интерпретации) прямо или косвенно поддерживает все 19 710 операторов трехзначной логики.

19.12. Date C.J. Faults and Defaults (in five parts) // C.J. Date, Hugh Darwen and David McGoveran. Relational Database Writings 1994—1997. — Reading, Mass.: Addison-Wesley, 1998.

Разработан систематический подход к проблеме отсутствия информации, основанный на использовании специальных значений и двухзначной логики вместо неопределенных значений и трехзначной логики. В статье аргументированно доказывается, что в реальном мире используются именно специальные значения и поэтому было бы желательно, чтобы применяемые системы баз данных в этом отношении действовали так же, как принято в реальном мире.

19.13. Dey D., Sarkar S. A Probabilistic Relational Model and Algebra // ACM TODS. — September 1996. — № 3.

В этой статье предлагается способ работы с "неопределенностью в значениях данных" на основе теории вероятности вместо неопределенных значений и трехзначной логики. "Вероятностная реляционная модель" является совместимым расширением обычной реляционной модели.

19.14. Galindo-Legaria C.A. Outerjoins as Disjunctions // Proc. 1994 ACM SIGMOD Int. Conf. On Management of Data, Minneapolis, Minn. — May 1994.

Внешнее соединение в общем случае не является ассоциативным оператором [19.4]. Эта статья точно характеризует те внешние соединения, которые являются ассоциативными, и те, которые не являются таковыми. В ней также предлагаются стратегии реализации для каждого случая.

19.15. Galindo-Legaria C.A., Rosenthal A. Outerjoin Simplification and Reordering for Query Optimization//ACM TODS. - March 1997. - 22, № 1.

В этой статье представлен "полный набор правил преобразования" для выражений, включающих внешние соединения.

19.16. Goel P., Iyer B. SQL Query Optimization: Reordering for a General Class of Queries // Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data, Montreal, Canada. — June 1996.

Так же, как и в [19.15], в этой статье рассматриваются выражения преобразования с внешними соединениями: "[Мы] предлагаем метод переупорядочения [произвольного] запроса SQL с соединениями, внешними соединениями и... агрегирующими функциями... [Мы] предлагаем мощную примитивную операцию [для упрощения такого переупорядочения, которую мы назвали] *обобщенной выборкой".*

19.17. Heath I.J. IBM internal memo. — April 1971.

В статье впервые вводится термин (и концепция)*внешнее соединение.*

19.18. Liu K.-C, Sunderraman R. Indefinite and Maybe Information in Relational Databases // ACM TODS. - March 1990. - 15, № 1.

Содержит набор формальных предложений по расширению реляционной модели для работы с *возможной информацией* (например, "деталь с номером Р7 может быть черного цвета") и с *неопределенной* или дизъюнктивной *информацией* (например, "деталь с номером Р8 или номером Р9 имеет красный цвет"). Представлены таблицы истинности *(I-tables)* для нормальной (определенной), возможной и неопределенной информации. Для работы с таблицами истинности расширены операторы сокращения, проекции, произведения, объединения, пересечения и разности.

19.19. McGoveran D. Nothing from Nothing (in four parts) // C.J. Date, Hugh Darwen and David McGoveran. Relational Database Writings 1994-1997. — Reading, Mass.: Addison-Wesley, 1998.

Работа состоит из четырех частей. В части I показано определяющее значение логики в системах баз данных. В части II объясняется, почему эта логика должна быть двухзначной и почему попытки использовать трехзначную логику не приветствуются. В части III рассматривается, как можно "разрешить" проблемы трехзначной логики. Наконец, в части IV приводится ряд практически применимых решений этих проблем, которые не требуют использования трехзначной логики.

19.20. RescherN. Many-Valued Logic. - New-York, N.Y.: McGraw-Hill, 1969.

Общепризнанный учебник по многозначной логике.

ГЛАВА 20

Наследование типов

20.1. ВВЕДЕНИЕ

Примечание. В настоящей главе широко используется материал, впервые представленный в главе 5. Поэтому, если читатель вначале лишь "поверхностно" ознакомился с указанной главой, то рекомендуем вернуться к ней и внимательно прочитать, прежде чем приступать к глубокому изучению данной главы.

В главе 14 рассматривались понятия подтипов и супертипов (а именно подтипов и супертипов сущностей) и было отмечено, что если (например) некоторые служащие являются программистами, а все программисты — служащими, то тип сущности PROGRAMMER может рассматриваться как подтип типа сущности EMPLOYEE, а тип сущности EMPLOYEE — как супертип типа сущности PROGRAMMER. Но в этой главе было также отмечено, что понятие *типа сущности* нельзя считать равносильным понятию *типа* в наиболее формальном смысле этого термина (отчасти потому, что сам термин *сущность* определен

недостаточно формально). В данной главе приведен глубокий анализ понятий подтипов и супертипов, но термин *тип* используется в том более теоретически обоснованном и точном смысле, который был определен в главе 5. Поэтому прежде всего необходимо тщательно определить сам этот термин, как показано ниже.

- **Тип** — это именованное множество значений (под этим подразумеваются все возможные значения рассматриваемого типа), наряду со связанным с ним множеством операторов, которые могут применяться к значениям и переменным рассматриваемого типа.

Кроме того, необходимо учитывать приведенные ниже свойства типов.

- Каждый конкретный тип может быть определен системой или пользователем.
- В состав определения любого конкретного типа входит спецификация множества всех допустимых значений этого типа (данная спецификация представляет собой соответствующее ограничение типа, как описано в главах 5 и 9).
- Подобные значения могут иметь произвольную сложность.
- Физическое представление таких значений всегда скрыто от пользователя; это означает, что следует отличать типы от (физических) представлений. Но каждый тип имеет по меньшей мере одно возможное представление, которое явно предоставляется в распоряжение пользователя с помощью соответствующих операторов THE_ (или некоторых логически эквивалентных средств).
- Операции со значениями и переменными конкретного типа могут осуществляться исключительно с помощью операторов, определенных для рассматриваемого типа.
- Кроме уже упомянутых операторов THE_, к ним относятся также следующие операторы:
  - по меньшей мере один *оператор селектора* (точнее, по одному такому оператору для каждого доступного пользователю возможного представления), который позволяет "выбирать" или задавать любое значение рассматриваемого типа с помощью соответствующего вызова селектора;
  - оператор проверки на равенство, который позволяет проверять любые два значения рассматриваемого типа для определения того, действительно ли они представляют собой одной тоже значение;
  - оператор присваивания, позволяющий присваивать значение рассматриваемого типа переменной, которая объявлена как относящаяся к рассматриваемому типу.

Кроме упомянутых свойств, необходимо отметить описанные ниже свойства.

- Некоторые типы являются **подтипами** других типов, называемых **супертипами.** Если в — подтип А, то операторы и ограничения типа, которые относятся к А, относятся также и к в (т.е., условно говоря, *наследуются* типом в), но сам тип в имеет собственные операторы и ограничения типа, которые не относятся к А. (Этим утверждениям будет дано более точное определение чуть позже.)

Например, предположим, что заданы два типа, ELLIPSE (Эллипс) и CIRCLE (Окружность), которые имеют очевидную интерпретацию. В таком случае можно утверждать, что тип CIRCLE является подтипом типа ELLIPSE (а тип ELLIPSE — супертипом типа CIRCLE). Под этим подразумевается, что указанные типы характеризуются описанными ниже свойствами.

- Каждая окружность представляет собой эллипс (т.е. множество всех окружностей — это подмножество множества всех эллипсов), но противоположное утверждение не является истинным.
- Таким образом, каждый оператор, который может применяться ко всем эллипсам в целом, применяется и к окружностям в частности (поскольку окружности — это эллипсы), но противоположное утверждение ложно. Например, оператор THE_CTR ("определение центра") может применяться к эллипсам, а поэтому и к окружностям, но оператор THE_R ("определение радиуса") может относиться только к окружностям.
- Кроме того, любое ограничение, которое относится к эллипсам в целом, распространяется и на окружности в частности (опять-таки, поскольку окружности — это эллипсы), но противоположное утверждение является ложным. Например, на эллипсы распространяется ограничение a > b (где a и b, соответственно, большая и малая полуоси), то такое же ограничение должно удовлетворяться также и для окружностей. Безусловно, у окружностей длины полуосей a и b совпадают и равняются радиусу r, поэтому задача удовлетворения указанного ограничения решается тривиально; фактически ограничение a = b может служить примером именно такого ограничения, которое относится к окружностям в частности, но не к эллипсам в целом.

  Примечание. Здесь и дальше в этой главе неуточненный термин *ограничение* используется именно для указания на ограничение типа. Кроме того, термины *радиус* и *полуоси* применяются неформально для обозначения тех понятий, которые следовало бы называть более точно как *длина радиуса* или *длины полуосей.*

Подведем итог: тип CIRCLE наследует операторы и ограничения от типа ELLIPSE, но он также имеет собственные операторы и ограничения, которые не применимы к типу ELLIPSE. Поэтому следует отметить, что подтип, по сравнению с супертипом, обладает подмножеством значений, но надмножеством свойств, а этот факт может иногда приводить к путанице! (Здесь и дальше в данной главе термин *свойства* используется в качестве удобного сокращения вместо *операторы и ограничения.)*

Область применения наследования типов

Почему эта тема заслуживает внимания? По-видимому, на этот вопрос есть по меньшей мере два ответа, приведенные ниже.

- Создается впечатление, что идеи выделения подтипов и наследования выдвигаются на передний план естественным образом в самих практических задачах, поскольку нередко возникают такие ситуации, в которых все значения определенного типа имеют некоторые свойства, тогда как определенное подмножество этих значений имеет, кроме них, особые собственные свойства. Поэтому средства выделения подтипов и наследования могут рассматриваться в качестве полезных инструментальных

средств моделирования реальности (или семантического моделирования — **под** этим названием они упоминались в главе 14).

- Кроме того, если мы сумеем распознать соответствующие закономерности (под этим подразумеваются закономерности выделения подтипов и наследования) и встроить относящиеся к ним интеллектуальные средства в прикладное и системное программное обеспечение, то получим возможность добиться определенных практических преимуществ. Например, программа, которая работает с эллипсами, сможет также работать с окружностями, даже если она была первоначально написана вообще без учета окружностей (возможно, в связи с тем, что тип CIRCLE ко времени написания этой программы еще не был определен); это преимущество известно под названием **повторного использования кода.**

Но несмотря на наличие таких потенциальных возможностей, следует отметить, что исследователи еще не пришли к общему мнению в отношении того, какой должна быть формальная, строгая и абстрактная модель наследования типов. Ниже приведена характерная цитата из [20.13].

> *Основная идея наследования типов является довольно простой ... [и все же, несмотря на его] центральную роль в современных ... системах, механизм наследования продолжает оставаться весьма спорным... [Какая-либо] всесторонняя трактовка наследования все еще отсутствует.*

Изложение данного материала в настоящей главе основано на модели, разработанной автором совместно с Хью Дарвеном (Hugh Darwen) и подробно описанной[1] в [3.3]. Поэтому необходимо учитывать, что другими авторами и в других источниках такие термины, как *подтип* и *наследование,* могут использоваться в толкованиях, отличных от применяемых в настоящей книге. Призываем читателя быть внимательным!

Некоторые предварительные сведения

Ниже приведен ряд предварительных сведений, которые необходимо рассмотреть, прежде чем приступать к подробному обсуждению тематики наследования как таковой. Эти предварительные сведения являются основной темой данного подраздела.

- **Типизация значений.**

  Как было указано в главе 5, если v представляет собой значение, то v можно рассматривать как своего рода объект, обозначенный флажком, на котором указано: "Я — целое число", "Я — номер поставщика" или "Я — окружность" (и т.д.). Итак, если не применяется наследование, то каждое значение относится к одному и только к одному типу, а при использовании наследования значение может принадлежать одновременно к нескольким типам. Например, определенное значение может одновременно относиться к типам ELLIPSE И CIRCLE.

---

[1] В указанной книге ясно подчеркнуто, что ее авторы стремятся избежать того, чтобы предложенная ими модель рассматривалась просто как еще одно теоретическое упражнение. Вместо этого они предлагают ее для обсуждения всем сообществом пользователей баз данных в целом в качестве претендента на роль модели наследования, которая позволит решить столь важную задачу "всеобъемлющего охвата всей тематики наследования".

- **Типизация переменных.**

  Каждая переменная имеет один **и** только один **объявленный** тип. Например, переменная может быть объявлена следующим образом.

  ```
  VAR E ELLIPSE ;
  ```

  Здесь объявленным типом переменной E является ELLIPSE. Итак, если не используется наследование, то все возможные значения некоторой переменной относятся к одному и только к одному типу, а именно к соответствующему объявленному типу. Если же применяется наследование, то любая конкретная переменная может иметь значение, которое одновременно относится к нескольким типам, например, текущим значением переменной E может оказаться эллипс, который фактически является окружностью и поэтому относится одновременно к типам ELLIPSE **и** CIRCLE.

- **Различия между одинарным и множественным наследованием.**

  Существуют две основные "разновидности" наследования типов — одинарное и множественное. Выражаясь неформально, **одинарное наследование** означает, что каждый подтип имеет только один супертип и наследует свойства только от одного супертипа, а **множественное наследование** означает, что любой подтип может иметь любое количество супертипов и наследовать свойства от всех них. Очевидно, что первый вид наследования является частным случаем последнего. Тем не менее, не только тема множественного наследования, но и тема одинарного наследования является достаточно сложной (что фактически на первый взгляд может показаться удивительным), поэтому в данной главе все внимание будет уделено только одинарному наследованию, **и** в ней неуточненный термин *наследование* применяется для обозначения именно одинарного наследования. Подробное описание обоих видов наследования, как множественного, так и одинарного, приведено в [3.3].

- **Наследование скаляров, кортежей и отношений.**

  Безусловно, понятие наследования распространяется не только на скалярные[2] значения, но и на нескалярные, поскольку в конечном итоге нескалярные значения состоят из скалярных. В частности, это понятие распространяется также на значения кортежей и отношений. Но даже тема наследования скаляров является достаточно сложной (и этот факт также достоин удивления), поэтому в данной главе основное внимание уделено наследованию скаляров и в ней неуточненные термины *тип, значение, переменная, оператор* и *выражение* относятся именно к скалярным типам, значениям, переменным, операторам и выражениям. Для ознакомления с подробным описанием всех видов наследования, включая не только скаляры, но и кортежи и отношения, обратитесь к [3.3].

---

[2] Напомним, что *скалярными* называются такие значения, которые не имеют компонентов, видимых пользователю. Не позволяйте сбить себя с толку в связи с тем, что некоторые скалярные типы имеют возможные представления, имеющие, в свою очередь, видимые пользователю компоненты (как описано в главе 5), поскольку эти компоненты являются компонентами возможного представления, а не компонентами типа, несмотря на тот факт, что мы иногда недопустимо трактуем их так, как если бы они действительно были компонентами типа.

Наследование структуры и функций.

Напомним, что скалярные значения могут иметь внутреннее (т.е. физическое) представление, или структуру произвольной сложности, например, как уже было сказано выше, эллипсы и окружности, которые в определенных обстоятельствах могут на законных основаниях рассматриваться как скалярные значения, безусловно, могут обладать весьма сложной внутренней структурой. Но эта внутренняя структура всегда скрыта от пользователя. На основании этого можно сделать вывод, что если речь идет о наследовании (по меньшей мере, в той степени, в которой рассматривается модель, предложенная автором), под этим не подразумевается наследование структуры, поскольку с точки зрения пользователя тип не имеет структуры, которую можно было бы наследовать! Иными словами, нас интересует то, что иногда в литературе называется **наследованием функций** (behavioral inheritance), а не **наследованием структуры** (structural inheritance), где под функциями подразумеваются операторы, но напомним, что в рассматриваемой модели, по меньшей мере, наследуются также ограничения.

Примечание. Автор не исключает возможности наследования структуры, но э та тема рассматривается как относящаяся только к вопросам реализации и не имеющая отношения к самой модели.

"Подтаблицы и супертаблицы".

Теперь читателю должно быть ясно, что рассматриваемая модель наследования касается того, что в терминах теории реляционных баз данных можно было назвать *наследованием доменов* (напомним, что *домены* и *типы* — это одно и то же). Но, столкнувшись с вопросом о том, существует ли возможность наследования в реляционном контексте, большинство специалистов сразу же предполагает, что речь идет о какой-то разновидности наследования таблиц. Например, в языке SQL предусмотрены средства поддержки определенных объектов, называемых в этом языке *подтаблицами и супертаблицами,* согласно которым определенная таблица в может унаследовать все столбцы некоторой другой таблицы A, а затем быть дополнена некоторыми собственными столбцами (см. главу 26). Но, по мнению автора, понятие *подтаблиц и супертаблиц* представляет собой нечто совсем иное, и хотя оно и может представлять определенный интерес (несмотря на то, что автор относится к нему скептически [14.13]), но не имеет ничего общего с наследованием типов как таковым.

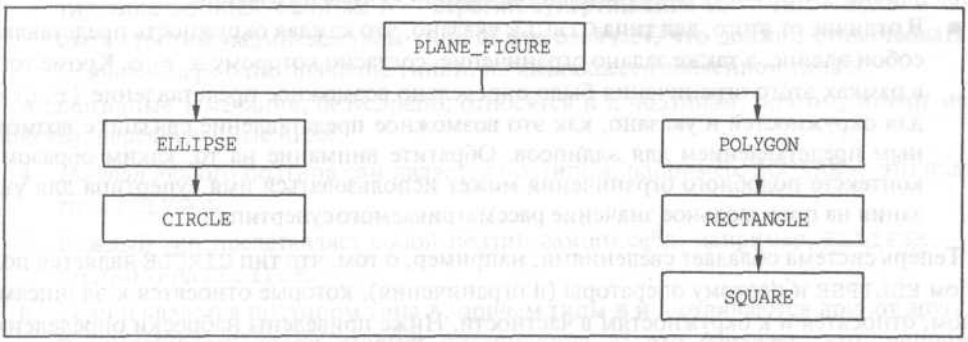
Приведем еще одно, последнее предварительное замечание — тема наследования типов фактически относится к проблематике изучения данных в целом и не ограничивается изучением лишь данных, хранимых в базе данных, в частности. Поэтому для упрощения большинство примеров этой главы выражено в терминах локальных данных (обычных переменных программ и т.д.), а не данных из базы данных.

20.2. ИЕРАРХИИ ТИПОВ

В этом разделе представлен рабочий пример, который используется на протяжении всей остальной части этой главы. Этот пример относится к множеству геометрических типов (PLANE_FIGURE, ELLIPSE, CIRCLE, POLYGON и т.д.), упорядоченных в виде так называемой **иерархии типов** (рис. 20.1). Ниже приведены наброски определений для

Рис. 20.1. Пример иерархии типов

некоторых из этих геометрических типов на языке Tutorial D (здесь заслуживают особого внимания ограничения типов).




```
TYPE PLANE_FIGURE ... ;

TYPE ELLIPSE
     IS PLANE_FIGURE
        POSSREP { A LENGTH, B LENGTH, CTR
        POINT
                  CONSTRAINT A > B } ;

TYPE CIRCLE
     IS ELLIPSE
        CONSTRAINT THE A ( ELLIPSE ) = THE B (
        ELLIPSE ) POSSREP { R = THE A ( ELLIPSE ) ,
                  CTR = THE_CTR ( ELLIPSE ) } ;
```

Рассмотрим эти определения более внимательно. Прежде всего, для упрощения в этой главе предполагается, что эллипсы всегда расположены в пространстве таким образом, что их большая ось a является горизонтальной, а малая ось b — вертикальной; кроме того, принято допущение, что большая полуось a всегда больше или равна меньшей полуоси b (иными словами, наши эллипсы — "низкие и полные", а не "высокие и худые"). Итак, для представления эллипсов достаточно знать размеры их полуосей a и b (и местонахождение их центра). В отличие от эллипсов, для представления окружностей достаточно знать их радиус r (и местонахождение их центра).

Затем необходимо сделать приведенные ниже уточнения.

- Тип PLANE_FIGURE (плоская фигура) вообще не имеет объявленного возможного представления; в ходе дальнейшего изложения в данной главе будет показано, что тип PLANE_FIGURE фактически представляет собой *объединенный тип.* (А на са мом деле он относится к особому роду объединенного типа, который называется *фиктивным типом.* Но подробное описание фиктивных типов выходит за рамки данной главы; дополнительная информация на эту тему приведена в [3.3].)
- Что касается типа ELLIPSE, то выше было указано, что каждый эллипс представ ляет собой плоскую фигуру; кроме того, в его объявлении обычным образом зада но возможное представление {a,b,ctr} с учетом действующего ограничения, согласно которому a > b.

Примечание. Для полноты определения фактически следовало бы также ввести ограничение, согласно которому значение b должно быть больше нуля. Но здесь это ограничение исключено в целях сокращения объема изложения. ■ В отличие от этого, для типа CIRCLE указано, что каждая окружность представляет собой эллипс, а также задано ограничение, согласно которому a = b. Кроме того, в рамках этого ограничения было определено возможное представление {r, ctr} для окружностей и указано, как это возможное представление связано с возможным представлением для эллипсов. Обратите внимание на то, каким образом в контексте подобного ограничения может использоваться имя супертипа для указания на произвольное значение рассматриваемогосупертипа.

Теперь система обладает сведениями, например, о том, что тип CIRCLE является подтипом ELLIPSE и поэтому операторы (и ограничения), которые относятся к эллипсам в целом, относятся и к окружностям в частности. Ниже приведены наброски определений некоторых из этих операторов.

```
OPERATOR AREA ( E ELLIPSE ) RETURNS AREA ;
   /* Оператор "вычисления площади указанного объекта" - следует
   отметить, */ /* что словом AREA обозначены и имя самого оператора,
   и тип результата  */

END OPERATOR ;

OPERATOR THE A ( E ELLIPSE ) RETURNS LENGTH ;
   /* Оператор "определения полуоси a указанного объекта" */
... ; END OPERATOR ;

OPERATOR THE B ( E ELLIPSE ) RETURNS LENGTH ;
   /* Оператор "определения полуоси Ь указанного объекта" */
... ; END OPERATOR ;

OPERATOR THE CTR ( E ELLIPSE ) RETURNS POINT ;
   /* Оператор "определения центра указанного объекта" */
... ; END OPERATOR ;

OPERATOR THE R ( C CIRCLE ) RETURNS LENGTH ;
   /* Оператор "определения радиуса указанного объекта" */
... ; END OPERATOR ;
```

Все эти операторы, кроме THE_R, относятся к значениям типа ELLIPSE, а, следовательно, в силу самого этого обстоятельства, и к значениям типа CIRCLE; но, в отличие от других операторов, действие оператора THE_R распространяется только на значения типа CIRCLE.

Терминология

К сожалению, нам придется ввести еще несколько терминов и определений, прежде чем мы сможем приступить к дальнейшему изложению. Но рассматриваемые ниже понятия в основном являются несложными.

.1. *Супертип* любого супертипа сам является супертипом, например, POLYGON (Многоугольник) — это супертип типа SQUARE (Квадрат).

2. Каждый тип представляет собой супертип по отношению к самому себе, например, ELLIPSE — ЭТО СуперТИП ТИПа ELLIPSE.

3. Если A — супертип типа B, причем типы A и B отличаются друг от друга, то A — строгий супертип (proper supertype) типа B, например, POLYGON — строгий супертип типа SQUARE. Если же A — строгий супертип типа B, то тип A должен также быть строгим надмножеством типа B; это означает, что должно существовать по меньшей мере одно значение типа A, не являющееся значением типа B.

Аналогичные замечания, безусловно, относятся и к подтипам, поэтому имеют место свойства, перечисленные ниже.

4. *Подтип* любого подтипа сам является подтипом, например, SQUARE — это подтип типа POLYGON.
5. Каждый тип представляет собой подтип самого себя, например, ELLIPSE — это подтип типа ELLIPSE.
6. Если в является подтипом типа A, причем типы в и A отличаются друг от друга, то в представляет собой строгий подтип типа A; это означает, что, например, SQUARE — это строгий подтип типа POLYGON. Если же в — строгий подтип типа A, то тип в должен быть также строгим подмножеством типаA.

Кроме того, справедливы приведенные ниже утверждения.

7. Если A — супертип типа в и не существует тип с, который был бы строгим подти пом типа A и строгим супертипом типа в, то A является непосредственным супер типом типа в, а в — непосредственным подтипом типа A, например, RECTANGLE (Прямоугольник) — непосредственный супертип типа SQUARE, а SQUARE — непо средственный подтип типа RECTANGLE. Поэтому заслуживает внимания то, что в синтаксисе языка Tutorial D, применяемом в данной главе, ключевое слово IS (Является) означает именно то, что указанный тип "является непосредственным подтипом" другого указанного типа.
8. Корневым типом (root type) называется тип, не имеющий строгого супертипа, на пример, PLANE_FIGURE является корневым типом.

   Примечание. Мы не исходим из предположения, что существует только один корневой тип. Но если имеется два или несколько таких типов, то всегда можно придумать своего рода "системный" тип, который будет служить непосредственным супертипом для них всех, поэтому без потери общности можно предположить, что существует только один корневой тип.
9. Листовым типом (leaf type) называется тип без строгого подтипа, например, в рас сматриваемой иерархии CIRCLE — листовой тип.
   Примечание. Это определение нескольку упрощено, но является подходящим для текущих целей (если речь идет о множественном наследовании, то оно требует небольшого дополнения [3.3]).
10. Каждый строгий подтип имеет один и только один непосредственныйсупертип.
    Примечание. В данном случае мы просто явно подчеркиваем принятое ранее пред положение, что в настоящей главе рассматривается только одинарное наследова ние. Как было отмечено выше, в [3.3] подробно описано, к каким последствиям приводит отказ от этого предположения.
11. Если соблюдаются такие условия, что, во-первых, имеется по меньшей мере один тип, и во-вторых, отсутствуют циклы (т.е. не существует последовательность типов

Т1, Т2, Т3, ..., Тп, такая что Т1 является непосредственным подтипом типа Т2, Т2 — непосредственным подтипом типа Т3,..., а Тп — непосредственным подтипом Т1), то хотя бы один тип обязательно должен быть корневым типом.
Примечание. **В** действительности, никаких циклов и не может быть (объясните, почему).

Предположение об отсутствии пересечения

В этой главе применяется еще одно упрощающее допущение, которое формулируется следующим образом: если Т1 и Т2 — разные корневые типы или разные непосредственные подтипы некоторого супертипа (из этого, в частности, следует, что ни один из них не является подтипом другого), то предполагается, что они являются **непересекающимися;** это означает, что ни одно значение не относится одновременно к типам Т1 и Т2. Например, ни одно значение не является одновременно эллипсом и многоугольником.

Из этого непосредственно следуют описанные ниже дополнительные свойства типов.

12. Иерархии различных типов являются непересекающимися.
13. Различные листовые типы являются непересекающимися.
14. Каждое значение относится к одному и только одному **наиболее конкретному типу.** Например, некоторое значение может представлять собой "просто эллипс", а не окружность; это означает, что для него наиболее конкретным типом служит ELLIPSE (в реальном мире некоторые эллипсы не являются окружностями). В действительности утверждение о том, что наиболее конкретным типом некоторого значения v является Т, равносильно именно такому утверждению, что множество типов, которыми обладает значение v, — это множество всех супертипов **т** (множество, которое по определению включает сам тип Т).

Одна из причин, по которым желательно принять предположение об отсутствии пересечения, состоит в том, что оно позволяет избежать некоторых неоднозначностей, которые могли бы возникнуть в ином случае. Например, предположим, что некоторое значение v может относиться к двум типам, Т1 и Т2, но ни одно из них не является подтипом другого. Кроме того, допустим, что для типа Т1 определен некоторый оператор Op, а для типа Т2 определен другой оператор с тем же именем — Op (иными словами, оператор Op является перегруженным — см. раздел 20.3). В таком случае вызов оператора Op с фактическим параметром v мог бы одновременно относиться к двум разным типам, т.е. был бы неоднозначным.

Примечание. Предположение об отсутствии пересечения является целесообразным лишь при том условии, что предметом нашего внимания является только одинарное наследование, а если речь идет о множественном наследовании, то от этого предположения необходимо отказаться. Подробное описание данной темы приведено в[3.3].

Несколько слов о физическом представлении

Несмотря на то, что нас интересует в основном модель наследования, а не вопросы реализации, ниже приведены некоторые замечания, касающиеся реализации, которые нужно в определенной степени учитывать, чтобы правильно понять всю концепцию наследования.

Из того факта, что в является подтипом типа A, не следует[3], что скрытое физическое представление значений типа в является таким же, как для значений типа A. Например, эллипсы могут быть физически представлены с указанием их центра и полуосей, а окружности могут быть физически представлены с указанием их центра и радиуса (хотя в общем не предъявляется такое требование, чтобы любое физическое представление было таким же, как любое из объявленных возможных представлений). Важность этого замечания станет очевидной после изучения некоторых из следующих разделов.

20.3. ПОЛИМОРФИЗМ И ЗАМЕНЯЕМОСТЬ

В этом разделе рассматриваются два важных понятия, *полиморфизм* и *заменяемость,* которые вместе составляют основу для достижения того преимущества повторного использования кода, которое было кратко упомянуто в разделе 20.1. Необходимо сразу отметить, что эти два понятия фактически представляют собой лишь разные способы трактовки одной и той же темы. Но так или иначе, начнем с рассмотрения понятия полиморфизма.

Полиморфизм

На основании самого определения наследования можно сделать вывод, что если T' — подтип типа т, то все операторы, которые могут применяться к значениям типа т, являются также применимыми к значениям типа T'. Например, если оператор AREA (**e**) является применимым по отношению к тому значению e, которое представляет собой эллипс, то применимым должен также быть оператор AREA (c), где c представляет собой окружность. Поэтому следует отметить, что необходимо тщательно учитывать логическое различие между **формальными параметрами,** в терминах которых определен данный оператор (вместе с их объявленными типами), и соответствующими **фактическими параметрами** в конкретном вызове этого оператора (которые обладают своими наиболее конкретными типами). Например, оператор AREA определен в терминах формальных параметров объявленного типа ELLIPSE (см. раздел 20.2), но наиболее конкретным типом фактического параметра в вызове AREA (c) является CIRCLE.

Еще раз напомним, что эллипсы и окружности (по меньшей мере, в том виде, в каком они были определены в разделе 20.2) имеют разные возможные представления, как показано ниже.

```
TYPE ELLIPSE . . .
     POSSREP { A ..., B ..., CTR ... } ;

TYPE CIRCLE . . .
     POSSREP { R . . ., CTR . . . } ;
```

[3] В действительности, не существует каких-либо обоснованных причин, по которым все значения одного и того же типа должны были бы иметь одно и то же физическое представление. Например, одни точки могут быть физически представлены с помощью декартовых координат, а другие — с помощью полярных координат; одни температуры могут быть физически представлены в градусах Цельсия, а другие — в градусах Фаренгейта; одни целые числа могут быть физически представлены в виде десятичных чисел, а другие — в виде двоичных и т.д. Безусловно, в подобных случаях система должна иметь сведения о том, как преобразуются физические представления, для того чтобы обладать способностью реализовывать операторы присваивания, сравнения и т.д. должным образом.

Поэтому допустимо применение двух разных версий оператора AREA, скрытых от пользователя, в одной из которых используется возможное представление ELLIPSE, а в другой — возможное представление CIRCLE. Еще раз отметим, что это условие является допустимым, но не необходимым. Например, код этого оператора применительно к эллипсам может выглядеть следующим образом.

```
OPERATOR AREA ( E ELLIPSE ) RETURNS AREA ;
   RETURN ( 3.14159 * THE A ( E ) * THE B ( E
) ) ; END OPERATOR ;
```

(Площадь эллипса определяется по формуле πab.) Вполне очевидно, что этот код работает правильно, если при вызове ему передается окружность вместо более общей фигуры — эллипса, поскольку применительно к окружности оба оператора THE_A и THE_B возвращают радиус r. Но программист, отвечающий за определение типа CIRCLE, может решить по многим причинам реализовать другую версию оператора AREA, которая относится только к окружностям и вызывает оператор THE_R вместо операторов THE_A и THE_B.

Примечание. Действительно, в целях повышения эффективности, так или иначе было бы желательно реализовать две версии оператора, даже если возможные представления являются одинаковыми. Рассмотрим, например, многоугольники и прямоугольники. Алгоритм вычисления площади многоугольника общей формы, безусловно, будет действовать и применительно к прямоугольнику, но для вычисления площадей прямоугольников предусмотрен более эффективный алгоритм— умножение высоты на ширину.

Но следует отметить, что код для эллипсов, безусловно, будет не применим для окружностей, если он разработан в терминах физического представления типа ELLIPSE, а не возможного представления, и физические представления типов ELLIPSE и CIRCLE различаются. Подход к реализации операторов в терминах физических представлений вообще не рекомендуется[4]. Поэтому код следует разрабатывать осмотрительно!

Так или иначе, но если оператор AREA не будет повторно реализован для типа CIRCLE, то имеет место ситуация повторного использования кода (здесь речь идет о коде реализации оператора AREA).

Примечание. Более важная разновидность ситуации повторного использования кода встретится в следующем подразделе.

Безусловно, что касается модели, то не имеет значения, сколько оудет существовать версий оператора AREA, скрытых от пользователя, поскольку с точки зрения пользователя имеется лишь один оператор AREA, применимый к эллипсам, а поэтому, по определению, также и к окружностям. Иными словами, что касается модели, то оператор AREA является **полиморфным,** а это означает, что он может принимать фактические параметры различных типов при разных вызовах. Поэтому следует отметить, что подобный полиморфизм является логическим продолжением наследования, поскольку, если предусмотрено наследование, то мы обязаны предусмотреть полиморфизм, а в противном случае не может быть и наследования!

---

[4] Наша собственная рекомендация фактически состоит в том, что доступ к физическим представлениям должен быть ограничен только тем кодом, в котором реализуется операторы, предписанные моделью (селекторы, операторы THE_ и т.д.). (Более того, многие из этих операторов на практике, скорее всего, будут иметь реализации, предусмотренные в системе.)

Итак, идею полиморфизма как таковую нельзя назвать такой уж новой, как уже мог заметить читатель (фактически эта тема уже кратко рассматривалась в главе 5). Например, в языке SQL давно применяются полиморфные операторы ("=", " + ", " | | " и многие другие) и фактически такая же ситуация обнаруживается в большинстве других языков. Некоторые языки даже позволяют пользователям определять свои собственные полиморфные операторы; например, в языке PL/I такое средство предоставляется под именем "универсальных" функций (ключевое слово GENERIC). НО ВО всех приведенных здесь примерах наследование как таковое отсутствует, поскольку все эти примеры относятся к так называемому **полиморфизму перегрузки** (overloading polymorphism). В отличие от этого, тот вид полиморфизма, который обнаруживается в операторе AREA, НОСИТ название **полиморфизма включения** (inclusion polymorphism) на том основании, что связь между (скажем) окружностями и эллипсами по сути представляет собой отношение включения множеств [20.4]. По очевидным причинам до конца этой главы неуточненный термин *полиморфизм* применяется для обозначения именно полиморфизма включения, если явно не указано иное.

Примечание. Ниже приведены наглядные определения, позволяющие лучше понять различия между полиморфизмом перегрузки и полиморфизмом включения.

- *Полиморфизм перегрузки* означает наличие нескольких различных операторов с од ним и тем же именем (при этом пользователь не обязан знать, что рассматривае мые операторы фактически являются различными и имеют разную семантику — хотя и желательно, чтобы их семантика была достаточно близкой). Например, в большинстве языков перегруженным является оператор "+", причем один опера тор "+" применяется для сложения целых чисел, другой оператор " + " предназна чен для сложения рациональных чисел и т.д.
- *Полиморфизм включения* означает наличие только одного оператора, который мо жет иметь несколько разных версий реализации, скрытых от пользователя (причем пользователь не обязан знать, действительно ли существует несколько версий реа лизации, — еще раз подчеркнем, что для пользователя это просто один оператор).

Программирование с использованием полиморфизма

Рассмотрим следующий пример. Предположим, что требуется разработать программу отображения некоторой схемы, состоящей из квадратов, окружностей, эллипсов и т.д. Без использования полиморфизма такая программа (представленная в виде псевдокода) будет выглядеть примерно следующим образом.

```
FOR EACH x ∈
    DIAGRAM CASE ;
       WHEN IS SQUARE ( X ) THEN CALL DISPLAY SQUARE ...
       ; WHEN IS CIRCLE ( X ) THEN CALL DISPLAY CIRCLE
       ... ;

    END CASE ;
```

(Здесь предполагается, что предусмотрены операторы IS_SQUARE, IS_CIRCLE и т.д., которые могут применяться для проверки того, относится ли данное значение к указанному типу; см. раздел 20.6.) В отличие от этого, при использовании полиморфизма код становится более простым и намного более выразительным, как показано ниже.

```
FOR EACH X £ DIAGRAM CALL DISPLAY ( X ) ;
```

Пояснение. В этом примере DISPLAY— полиморфный оператор. Версия реализации оператора DISPLAY, предназначенная для работы со значениями типа т, задается при определении типа Е и с этого момента становится известной системе. Поэтому на этапе прогона программы при обнаружении системой вызова оператора DISPLAY с фактическим параметром х система должна определить наиболее конкретный тип х, а затем вызвать версию оператора DISPLAY, соответствующего этому типу. Такой процесс называется **связыванием на этапе прогона**[5] (run-time binding). Другими словами, полиморфизм по сути означает, что выражения CASE и операторы CASE, которые в ином случае должны были бы появиться в исходном коде пользовательской программы, становятся скрытыми от пользователя, поскольку эти операции выбора направления дальнейшего продолжения программы с помощью операторов CASE фактически выполняет система от имени пользователя.

В частности, заслуживает внимания то, какое воздействие оказывают описанные выше средства на весь процесс сопровождения программы. Например, предположим, что в качестве еще одного непосредственного подтипа POLYGON определен новый тип TRIANGLE и поэтому на отображаемой схеме теперь будут появляться также и треугольники. Без использования полиморфизма каждую программу, которая содержит выражение CASE или другие операторы, аналогичные приведенным в данном примере, теперь придется модифицировать и включить в нее код примерно в следующей форме.

```
WHEN IS_TRIANGLE ( X ) THEN CALL DISPLAY_TRIANGLE ... ;
```

Однако при использовании полиморфизма подобные модификации исходного кода не требуются.

Под впечатлением примеров, подобных приведенному выше, полиморфизм иногда характеризуют таким метафорическим выражением, как "способ вызывать новый код с помощью старого кода"; это означает, что программа р фактически приобретает возможность вызывать некоторую версию некоторого оператора, которая даже еще не существовала (имеется в виду версия) ко времени написания программы Р. Поэтому в данном случае показан другой (и более важный) способ повторного использования кода, в котором совершенно одинаковая программа может применяться, кроме всего прочего, и для обработки данных типа т (еще раз повторяем), даже не существовавшего ко времени написания программы Р.

Заменяемость

Как уже было сказано, понятие заменяемости по сути представляет собой просто понятие полиморфизма, рассматриваемое немного с другой точки зрения. Например, было показано, что если оператор AREA (е), где е — эллипс, является допустимым, то должен быть также допустимым оператор AREA (с), где с — окружность. Иными словами, в любом месте, где система допускает наличие эллипса, можно всегда заменить его окружностью. Более общая формулировка этого принципа состоит в том, что в любом месте, где система допускает использование значения типа т, его всегда можно заменить значением типа Т', где Т' — подтип типа т. Этот принцип называется **принципом заменяемости значений.**

---

[5] Безусловно, что применение процесса связывания на этапе прогона относится к задачам реализации, а не к задачам определения модели. Это — еще одна из тех задач реализации, важность которых читатель должен в определенной степени оценить, чтобы правильно понять всю концепцию наследования.

В частности, следует отметить, что из этого принципа следует такой вывод: если некоторое отношение r имеет атрибут A с объявленным типом ELLIPSE, то некоторые значения атрибута A в отношении r могут принадлежать к типу CIRCLE, а не просто к типу ELLIPSE. Аналогичным образом, если некоторый тип t имеет возможное представление, включающее компонент c объявленного типа ELLIPSE, то для некоторых значений v типа t вызов оператора THE_C (v) может возвращать значение типа CIRCLE, а не просто типа ELLIPSE.

Наконец, следует отметить, что заменяемость также является логическим продолжением наследования, поскольку понятие заменяемости в действительности представляет собой лишь другое определение понятия полиморфизма. Это означает, что если применяется наследование, то должна обеспечиваться заменяемость; в противном случае, не может быть и наследования.

20.4. ПЕРЕМЕННЫЕ **И ОПЕРАТОРЫ ПРИСВАИВАНИЯ**

Предположим, что имеются две переменные E и c с объявленными типами, соответственно, ELLIPSE и CIRCLE, как показано ниже.

```
VAR  E  ELLIPSE  ;
VAR  C  CIRCLE  ;
```

Допустим, что происходит инициализация переменной c для определения некоторой окружности, предположим (просто, чтобы этот пример был более конкретным), окружности с радиусом три и центром в начале координат.

```
C := CIRCLE ( LENGTH ( 3.0 ), POINT ( 0.0, 0.0 ) ) ;
```

В этом примере выражение справа от оператора присваивания представляет собой вызов селектора для типа CIRCLE. Как было указано в главе 5, для каждого объявленного возможного представления предусмотрен конкретный оператор селектора с тем же именем и с формальными параметрами, соответствующими компонентам рассматриваемого возможного представления. Назначение селектора состоит в том, чтобы дать пользователю возможность определять или "выбирать" значение рассматриваемого типа, предоставляя значение для каждого компонента рассматриваемого возможного представления.

Примечание. Для упрощения здесь и дальше в этой главе предполагается, что декартово возможное представление для точек называется POINT, а не (как в главе 5) CARTESIAN. Поэтому вторым фактическим параметром в селекторе CIRCLE в данном примере является вызов этого селектора декартовых координат для точек.

Теперь рассмотрим следующий оператор присваивания.

```
E  :=  C  ;
```

В обычных обстоятельствах (т.е. в отсутствие применения подтипов и наследования) для выполнения соответствующей операции присваивания требуется, чтобы значение, представленное выражением в правой части, имело тот же тип (а именно, объявленный тип), что и переменная в левой части. Но из *принципа заменяемости значений* следует, что в любом месте, где система ожидает значение типа ELLIPSE, всегда можно подставить значение типа CIRCLE, поэтому данная операция присваивания является допустимой (фактически операция присваивания является полиморфной). Кроме того, действие этой операции сводится к тому, что значение окружности копируется из переменной C

в переменную E, поэтому значение переменной E после этого присваивания относится к типу CIRCLE, а не к типу ELLIPSE. Иными словами, справедливы приведенные ниже утверждения.

- **Значения сохраняют свой наиболее конкретный тип после их присваивания перемен ным наименее конкретного объявленного типа.** При таком присваивании преобразо вание типов не происходит[6] (в данном примере окружность не преобразуется так, чтобы она стала "просто эллипсом"). Следует также отметить, что нам и не нужно, чтобы выполнялись какие-либо подобные преобразования, поскольку они приве ли бы к потере наиболее конкретных функциональных свойств рассматриваемого значения. Например, в данном случае применение такого преобразования означало бы, что после присваивания мы теряем возможность определить радиус из значе ния окружности в переменной E. (См. подраздел "Оператор TREAT DOWN", приве денный ниже в данном разделе, где описано, какие действия необходимо выпол нить для решения указанной задачи определения радиуса.)
- Поэтому из понятия заменяемости следует, что *переменная объявленного типа* T *может иметь любое значение, наиболее конкретным типом которого является* любой **подтип** *типа* T. Поэтому необходимо учитывать, что пользователи типов и подти пов обязаны тщательно учитывать логическое различие между объявленным ти пом некоторой переменной и наиболее конкретным типом текущего значения этой переменной. Мы вернемся к этому важному вопросу **в** следующем подразделе.

Продолжая приведенный выше пример, предположим, что теперь предусмотрена еще одна переменная A с объявленным типом AREA, как показано ниже.

```
VAR A AREA ;
```

Рассмотрим следующий оператор присваивания.

```
A := AREA ( E ) :
```

Ниже описано, что происходит при выполнении данного оператора.

- Во-первых, система выполняет проверку типов на этапе компиляции выражения AREA(E) . Эта проверка завершается успешно, поскольку E относится к объявлен ному типу ELLIPSE, а единственный формальный параметр оператора AREA также относится к объявленному типу ELLIPSE, как показано в разделе 20.2.
- Во-вторых, система на этапе прогона обнаруживает, что текущим наиболее кон кретным типом переменной E (вернее, хранящегося в ней значения) является CIRCLE, и поэтому вызывает версию оператора AREA, которая относится к окруж ностям; иными словами, система выполняет процесс связывания на этапе прого на, описанный в предыдущем разделе.

Безусловно, тот факт, что вызывается версия оператора AREA, относящаяся к окружностям, а не версия для типа ELLIPSE, не должен заботить пользователя; еще раз отметим, что с точки зрения пользователя существует только один оператор AREA.

---

[6] И действительно, даже беглый анализ показывает, что сама идея подобного преобразования бессмысленна, поскольку даже если бы оно было возможно, это означало бы, что одно и то же значение может относиться к двум наиболее конкретным типам одновременно.

Переменные

Выше было показано, что текущее значение v переменной V объявленного типа т может иметь любой подтип типа т в качестве своего наиболее конкретного типа. Из этого следует, что существует возможность моделировать переменную v как упорядоченную тройку в форме <DT, MST, v>. Здесь используются обозначения, описанные ниже.

- DT. Объявленный тип (Declared Type — DT) переменной V.
- MST. Текущий наиболее конкретный тип (Most Specific Type — MST) для перемен ной V (под этим подразумевается наиболее конкретный тип значения, которое яв ляется текущим значением переменной V).
- v. Значение наиболее конкретного типа MST, а именно текущее значение перемен ной V.

Для обозначения компонентов DT, MST и v рассматриваемой модели переменной V будут использоваться, соответственно, обозначения DT (V), MST (v) и v (V). Необходимо учитывать следующее: во-первых, MST (V) всегда представляет собой подтип (причем не обязательно строгий подтип) типа DT (V); во-вторых, компоненты MST (V) и v (V) в общем могут изменяться во времени; в-третьих, компонент MST (V) фактически определяется компонентом v (V), поскольку каждое значение относится к одному и только одному наиболее конкретному типу.

Эта модель переменной может применяться для подробного анализа семантики различных операций, включая, в частности, операции присваивания. Но прежде чем приступить к этому анализу, необходимо пояснить, что понятия объявленного типа и текущего наиболее конкретного типа могут быть очевидным образом дополнены, чтобы они могли применяться не только к простым переменным, но и к произвольным выражениям. Допустим, что x— такое выражение, а v(X) — результат вычисления этого выражения. В таком случае справедливы приведенные ниже утверждения.

- Выражение x имеет объявленный тип DT (X), а именно объявленный тип операто ра Op, который вызывается на самом внешнем уровне выражения X. Тип DT(X) становится известным на этапе компиляции.
- Выражение x имеет также текущий наиболее конкретный тип MST(X), а именно тип, который является наиболее конкретным типом значения v(X). Тип MST(X) (в общем) остается неизвестным до этапа прогона.

  Теперь мы можем приступить к подробному описанию действия оператора присваивания. Рассмотрим следующий оператор присваивания.

  V := X ;

  Здесь V — переменная, а x — выражение. Тип DT (X) должен быть подтипом типа DT(V) , так как в противном случае этот оператор присваивания становится недопустимым (проверка происходит на этапе компиляции). А если оператор присваивания является допустимым, то результат его действия сводится к тому, что тип MST (V) становится равным типу MST (X), а значение v (V) — равным значению v(X).

Кстати, следует отметить, что если текущий наиболее конкретный тип переменной V равен т, то каждый строгий супертип типа т также является "текущим типом" переменной V.

Например, если переменная E (объявленного типа ELLIPSE) имеет текущее значение наиболее конкретного типа CIRCLE, то "текущими типами" E являются сразу все типы — CIRCLE, ELLIPSE и PLANE_FIGURE. Но выражение "текущий тип X" обычно применяется, по меньшей мере, неформально для обозначения именно наиболее конкретного типа MST(x).

Дополнительные сведения о заменяемости

Рассмотрим следующее определение оператора.

```
OPERATOR COPY ( E ELLIPSE )
   RETURNS ELLIPSE ; RETURN ( E ) ;
END OPERATOR ;
```

Благодаря использованию заменяемости оператор COPY можно вызвать с указанием в качестве фактического параметра наиболее конкретного типа либо типа ELLIPSE, либо типа CIRCLE, и, безусловно, к какому бы типу ни относился этот фактический параметр, оператор COPY должен возвратить результат с тем же наиболее конкретным типом. Поэтому из определения заменяемости вытекает еще одно следствие, что если оператор Op определен как имеющий результат объявленного типа т, то фактический результат вызова оператора Op может (в общем) относиться к любому подтипу типа т. Иными словами, так же как, во-первых, ссылка на переменную объявленного типа т может (в общем) фактически указывать на значение любого подтипа типа т, так и, во-вторых, вызов любого оператора с объявленным типом т может (опять-таки, в общем) фактически возвращать значение любого подтипа типа т.

Оператор TREAT DOWN

Еще раз рассмотрим пример, приведенный в начале данного раздела.

```
VAR E ELLIPSE ;
VAR C CIRCLE ;

C := CIRCLE ( LENGTH ( 3.0 ), POINT ( 0.0, 0.0 ) ) ;
E := C ;
```

Теперь компонент модели переменной MST(E) представляет собой тип CIRCLE. Поэтому предположим, что требуется определить радиус рассматриваемого окружности и присвоить его некоторой переменной L. Первая попытка выполнить такое действие может состоять в следующем.

```
VAR L LENGTH ;
L := THE_R ( E ) ; /* Ошибка несоответствия типов на этапе
компиляции ! ! ! */
```

Но, как указано в комментарии к этому коду, попытка его применения оканчивается неудачей из-за ошибки при проверке типа на этапе компиляции. А именно, такое неудачное завершение происходит из-за того, что оператор THE_R ("оператор определения радиуса"), присутствующий в правой части оператора присваивания, требует использования фактического параметра типа CIRCLE, а объявленным типом переменной E является ELLIPSE, а не CIRCLE. Следует отметить, что если бы такая проверка типов на этапе компиляции не была выполнена, то мы получили бы вместо ошибки времени компиляции ошибку из-за несоответствия типов на этапе прогона (а это гораздо хуже) в случае

обнаружения того, что значение E на этапе прогона представляет собой просто эллипс, а не окружность. Тем не менее, в рассматриваемой ситуации нам известно, что на этапе прогона данное значение будет представлять собой окружность, но вся сложность состоит в том, что мы это знаем, а компилятор— нет.

Для решения подобных проблем автор предлагает ввести новый оператор и присвоить ему неформальное название TREAT DOWN ("рассматривать как относящийся к подтипу"). В таком случае правильный способ получения радиуса в данном примере становится таким, как показано ниже.

```
L := THE_R ( TREAT_DOWN_AS_CIRCLE ( E ) ) ;
```

Выражение TREAT_DOWN_AS_CIRCLE(E) определяется как относящееся к объявленному типу CIRCLE, поэтому теперь проверка типов на этапе компиляции оканчивается успешно. Затем на этапе прогона осуществляются описанные ниже действия.

- Если текущее значение переменной E действительно относится к типу CIRCLE, то все это выражение успешно возвращает радиус данной окружности. Точнее, вызов оператора TREAT DOWN приводит к получению некоторого результата, скажем, z, который, во-первых, имеет объявленный тип DT(Z), равный CIRCLE, поскольку задана спецификация ". . ._AS_CIRCLE" ("... как окружность"); во-вторых, относится к текущему наиболее конкретному типу MST(Z), равному MST(E), который в данном примере также представляет собой CIRCLE; в-третьих, имеет текущее значение v(Z), равное v(E); наконец, в-четвертых, вычисляется выражение "THE_R(Z)", позволяющее получить требуемое значение радиуса (которое затем может быть присвоено переменной L).
- Но если текущее значение E относится только к типу ELLIPSE, а не к типу CIRCLE, то выполнение оператора TREAT DOWN на этапе прогона оканчивается неудачей из-за ошибки, связанной с несоответствием типов.

Общее назначение оператора TREAT DOWN состоит в обеспечении того, чтобы ошибки из-за несоответствия типов ко время прогона могли возникать только в контексте вызова самого оператора TREAT DOWN.

Примечание. Предположим, что тип CIRCLE, в свою очередь, имеет строгий подтип, скажем, O_CIRCLE (O-окружность; здесь под "O-окружностью" подразумевается окружность с центром в начале координат), который определен, как показано ниже.

```
TYPE O CIRCLE
     IS CIRCLE
        CONSTRAINT THE CTR ( CIRCLE ) = POINT ( 0.0,
        0.0 ) POSSREP { R = THE_R ( CIRCLE ) } ;
```

В таком случае в некоторый момент времени текущее значение переменной E может относиться к наиболее конкретному типу O_CIRCLE, а не просто CIRCLE. Если дело обстоит таким образом, то приведенный ниже вызов оператора TREAT DOWN завершится успешно.

```
TREAT_DOWN_AS_CIRCLE ( E )
```

Этот вызов приведет к получению результата, скажем, Z, такого что, во-первых, его тип DT(Z) равен CIRCLE, поскольку задана спецификация ". . ._AS_CIRCLE";

во-вторых, тип MST(Z) равен O_CIRCLE, поскольку O_CIRCLE — наиболее конкретный тип переменной E; и в-третьих, значение v(Z) равно v(E). Иными словами (выражаясь неформально), оператор TREAT DOWN всегда оставляет неизменным наиболее конкретный тип и ни при каких обстоятельствах "не продвигает его вверх по иерархии типов", чтобы он стал менее конкретным, чем перед этим.

Ниже приведено предназначенное для использования в будущем более формальное определение семантики вызова оператора TREAT_DOWN_AS_T(X) , где x— некоторое выражение. Во-первых, тип MST(X) должен быть подтипом типа T (проверка этого выполняется на этапе прогона); если предположить, что это условие удовлетворяется, то данный вызов возвращает результат Z с типом DT(Z) , равным т, типом MST (Z) , равным MST(X), и значением v (Z) , равным v(X).

Примечание. **В** [3.3] определена также обобщенная форма оператора TREAT DOWN, которая позволяет рассматривать один операнд *как относящийся* к типу другого, а не как относящийся к некоторому явно указанному типу.

20.5. УТОЧНЕНИЕ С ПОМОЩЬЮ ОГРАНИЧЕНИЯ

Рассмотрим следующий пример вызова селектора для типа ELLIPSE.

```
ELLIPSE ( LENGTH ( 5.0 ), LENGTH ( 5.0 ), POINT ( . . . ) )
```

Это выражение возвращает эллипс с равными полуосями. Но в реальном мире эллипс с равными полуосями фактически представляет собой окружность, поэтому можно ли рассчитывать на то, что данное выражение возвратит результат наиболее конкретного типа CIRCLE, а не просто наиболее конкретного типа ELLIPSE?

Вопросы, подобные этому, вызвали публикацию в литературе многочисленных откликов, зачастую выражающих противоположные мнения (а фактически эта дискуссия все еще продолжается [20.6]). После тщательных размышлений автор в определении своей собственной модели решил настаивать на том, что указанное выражение должно возвращать результат наиболее конкретного типа CIRCLE. Более общее определение этого подхода состоит в следующем: если тип T' представляет собой подтип типа т, а вызов селектора для типа T возвращает значение, которое удовлетворяет ограничению типа для типа T', то (в рассматриваемой модели) результат этого вызова селектора относится к типу T'. Теперь (ко времени написания данной книги) лишь немногие из современных коммерческих продуктов (если вообще таковые имеются) фактически действуют указанным образом, но автор рассматривает этот факт как упущение со стороны разработчиков данных продуктов. В частности, в [3.3] показано, что вследствие данного упущения подобные системы вынуждены поддерживать "некруглые окружности", "неквадратные квадраты" и аналогичные абсурдные объекты, тогда как эти критические замечание не относятся к предлагаемому автором подходу.

Примечание. См. также описание **второго серьезного заблуждения** в главе 26.

Из приведенного выше следует, что ни одно значение наиболее конкретного типа ELLIPSE не должно характеризоваться таким значением длин полуосей a и b, при котором a = b (по меньшей мере, в предложенной автором модели); иными словами, значения наиболее конкретного типа ELLIPSE ДОЛЖНЫ ТОЧНО соответствовать тем эллипсам реального мира, которые действительно не являются окружностями. В отличие от этого, в других моделях наследования значения наиболее конкретного типа ELLIPSE соответствуют таким

эллипсам реального мира, которые могут быть или не быть окружностями. Поэтому автор надеется, что предложенная им модель немного ближе к такому определению, как "достоверная модель реального мира".

Наконец, идея о том, что, например, эллипс с полуосями a = b должен относиться к типу CIRCLE, известна под названием **уточнения с помощью ограничения** (specialization by constraint) [3.3], но мы обязаны предупредить читателя, что другие авторы используют этот термин или очень близкий к нему для обозначения совсем другого понятия (см., например, [20.10], [20.14]).

Дополнительные сведения о псевдопеременных THE_

Как было описано в главе 5, псевдопеременные THE_ предоставляют способ обновления одного компонента переменной, притом что другие компоненты остаются неизменными (под "компонентами" здесь подразумеваются компоненты некоторого возможного представления, а не обязательно физического представления). Например, предположим, что переменная E относится к объявленному типу ELLIPSE, а текущее значение E представляет собой эллипс (скажем) с полуосью a, равной пяти, и полуосью Ь, равной трем. В таком случае приведенный ниже оператор присваивания обновляет значение длины полуоси b переменной E (после чего эта длина становится равной четырем), не изменяя значение полуоси или положение центра.

```
THE_B ( E ) := LENGTH ( 4.0 ) ;
```

Итак, из этого следует вывод, что псевдопеременные THE_ фактически не требуются; они в действительности представляют собой просто сокращения (что было также отмечено в главе 5). Например, приведенный выше оператор присваивания, в котором используется псевдопеременная THE_, эквивалентен следующему, в котором она не используется.

```
E := ELLIPSE ( THE_A ( E ), LENGTH ( 4.0 ), THE_CTR ( E ) ) ;
```

Поэтому рассмотрим следующий оператор присваивания.

```
THE_B ( E ) := LENGTH ( 5.0 )
```

По определению этот оператор присваивания эквивалентен приведенному ниже.

```
E := ELLIPSE ( THE_A ( E ), LENGTH ( 5.0 ), THE_CTR ( E ) ) ;
```

Таким образом, вступает в силу принцип уточнения с помощью ограничения (поскольку выражение в правой части возвращает эллипс с полуосями a = b), и общий эффект состоит в том, что после присваивания типом MST(E) становится CIRCLE, а не ELLIPSE.

Затем рассмотрим следующий оператор присваивания.

```
THE_B ( E ) := LENGTH ( 4.0 ) ;
```

Теперь переменная E содержит эллипс с полуосью a, равной пяти, и полуосью b, равной четырем, а тип MST (E) снова становится типом ELLIPSE, т.е. по существу происходит действие, которое автор называет **обобщением** с помощью ограничения (generalization by constraint).

Примечание. Предположим, что (как и в конце раздела 20.4) тип CIRCLE имеет строгий подтип O_CIRCLE (где "O-окружность" представляет собой окружность с центром в начале координат) с приведенным ниже определением.

```
TYPE O_CIRCLE IS CIRCLE
    CONSTRAINT THE_CTR ( CIRCLE ) = POINT ( 0.0, 0.0 )
    POSSREP { R = THE_R ( CIRCLE ) } ;
```

В таком случае текущее значение переменной E в некоторый заданный момент времени может относиться к наиболее конкретному типу O_CIRCLE, а не просто к типу CIRCLE. Предположим, что так оно и есть, и рассмотрим приведенную ниже последовательность операторов присваивания[7].

```
THE_A ( E ) := LENGTH (7.0) ;
THE_B ( E )  := LENGTH (7.0) ;
```

После первого из этих присваиваний E содержит "просто эллипс" благодаря обобщению с помощью ограничения. Но после второго присваивания она снова содержит окружность, но является ли последняя именно O-окружностью или "просто окружностью"? Безусловно, желательно, чтобы это была именно O-окружность. И разумеется, так оно и есть, поскольку данное значение удовлетворяет ограничению типа O_CIRCLE (включая ограничение, унаследованное этим типом от типа CIRCLE).

Побочные следствия изменения типов

Снова предположим, что E — переменная объявленного типа ELLIPSE. Выше было описано, как изменить тип E, чтобы эта переменная рассматривалась "как находящаяся на более низком уровне иерархии типов" (например, если ее текущим наиболее конкретным типом является ELLIPSE, то значение этой переменной можно обновить таким образом, чтобы ее текущим наиболее конкретным типом стал CIRCLE). Кроме того, в этой главе было показано, как изменить тип переменной E, чтобы она рассматривалась "как находящаяся на более высоком уровне иерархии типов" (например, если ее текущим наиболее конкретным типом является CIRCLE, значение этой переменной можно обновить таким образом, чтобы ее текущим наиболее конкретным типом стал ELLIPSE). А какие *побочные эффекты* вызывает такое изменение типа? Предположим, что рассматриваемый пример будет дополнен таким образом, что тип ELLIPSE имеет два непосредственных подтипа[8], CIRCLE и NONCIRCLE, смысл которых является очевидным. Не углубляясь в излишние подробности, можно сделать приведенные ниже выводы.

- Если текущее значение переменной E относится к типу CIRCLE (поэтому a = b), такое обновление E, в результате которого становится истинным условие a >b, повлечет за собой то, что MST (E) станет равным NONCIRCLE.
- Если текущее значение переменной E относится к типу NONCIRCLE (поэтому a > b), такое обновление E, в результате которого становится истинным условие a = b, по влечет за собой то, что MST(E) станет равным CIRCLE.

Таким образом, при конкретизации с помощью ограничений учитываются также *побочные эффекты* изменения типов.

---

[7] Если бы поддерживалось множественное присваивание, можно было бы выполнить эту последова тельность операций в виде одной операции.

[8] Кстати, следует отметить, что теперь тип ELLIPSE становится объединенным типом. См. раздел 20.7.

Примечание. Если читатель задумался над следующим вопросом, но не нашел на него ответ, отметим, что такое обновление E, при котором выполняется условие a < Ь, является невозможным (оно нарушает ограничение, налагаемое на тип ELLIPSE).

20.6. ОПЕРАЦИИ СРАВНЕНИЯ

Предположим, что рассматриваются два обычных примера переменных E и с с объявленными типами, соответственно, ELLIPSE и CIRCLE, и переменной E присваивается текущее значение переменной C следующим образом.

```
E := C ;
```

В таком случае, безусловно, очевидно, что теперь при выполнении показанной ниже операции сравнения на равенство должен быть получен результат TRUE, И ЭТО действительно так и происходит.

```
E = C
```

Общее правило состоит в следующем. Допустим, что x и Y — произвольные выражения. В таком случае операция сравнения x = Y является допустимой, если объявленные типы DT (X) и DT (Y) имеют общий супертип (это требование, безусловно удовлетворяется, если один из этих типов является супертипом другого). В противном случае такое сравнение является недопустимым (соответствующая проверка проводится на этапе компиляции). А если эта операция сравнения является допустимой, она возвращает TRUE, если значение v(x) равно значению v(Y), и FALSE — в ином случае. Кстати, следует отметить, что X и Y вряд ли могут "рассматриваться как равные", если их наиболее конкретные типы являются разными, поскольку если v(X) равно v(Y), то MST(X) должно быть равно MST (Y).

Применение операций сравнения в реляционной алгебре

Как было описано в главе 7, *операции сравнения на равенство* предусмотрены[9] явно или неявно во многих операциях реляционной алгебры. А если речь о супертипах и подтипах, то оказывается, что некоторые из этих операций обнаруживают такое поведение, которое может (по крайней мере, на первый взгляд) показаться не совсем понятным. Рассмотрим отношения RX и RY, показанные на рис. 20.2. Отметим, что единственный атрибут A в отношении RX принадлежит к объявленному типу ELLIPSE, а его аналог A в отношении RY принадлежит к объявленному типу CIRCLE. Примем такое соглашение, что значения в форме Ei на данном рисунке представляют собой эллипсы, которые не являются окружностями, а значения в форме Ci — это окружности. Наиболее конкретные типы обозначены строчными буквами.

Теперь рассмотрим соединение отношений RX и RY, скажем, RJ (рис. 20.3). Очевидно, что каждое значение A в соединении RJ ДОЛЖНО обязательно принадлежать к типу CIRCLE (поскольку любое значение A в отношении RX, наиболее конкретным типом которого является просто ELLIPSE, не может "рассматриваться как равное" любому значению A в RY). Поэтому можно считать, что объявленным типом атрибута A в соединении RJ

[9] Рассматриваемые операции сравнения фактически представляют собой операции сравнения кортежей, но в интересах текущего изложения мы можем считать их такими, как если бы они были просто операциями сравнения скаляров.

должен быть CIRCLE, а не ELLIPSE. Но необходимо учитывать также приведенные ниже соображения.

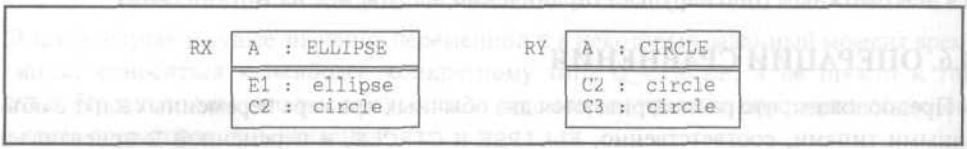


Рис. 20.2. Отношения RX и RY

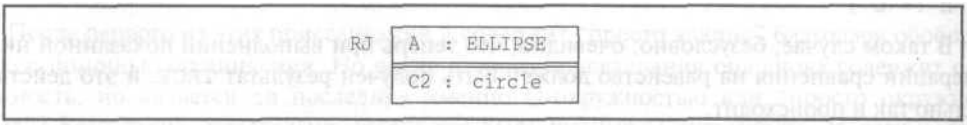


Рис. 20.3. Соединение RJ отношений RX и RY

Поскольку каждое из отношений, RX и RY, включает A в качестве своего единственного атрибута, то выражение RX JOIN RY сводится к RX INTERSECT RY. Поэтому в данном случае правило, касающееся объявленного типа атрибута результата для операции соединения JOIN, должно, очевидно, сводиться к аналогичному правилу ДЛЯ INTERSECT.

В свою очередь, выражение RX INTERSECT RY логически эквивалентно выражению RX MINUS (RX MINUS RY). Допустим, что при этом результатом выполнения выражения, которое является вторым операндом (речь идет о выражении RX MINUS RY), является отношение RZ. Тогда должно быть очевидно следующее:

а) в общем отношение RZ должно включать некоторые значения A наиболее кон кретного типа ELLIPSE и поэтому объявленным типом атрибута A в отноше нии RZ должен быть ELLIPSE;

б) таким образом, первоначальное выражение сводится к RX MINUS RZ, где объ явленным типом атрибута A и в RX, и в RZ является ELLIPSE, И поэтому приво дит к получению окончательного результата, в котором объявленным типом атрибута A снова, безусловно, должен быть ELLIPSE.

- Из этого следует, что объявленным типом атрибута результата для выражения RX INTERSECT RY, и поэтому также для выражения RX JOIN RY, должен быть ELLIPSE, а не CIRCLE, даже несмотря на то (еще раз напоминаем), что каждое значение этого атрибута должно фактически принадлежать к типу CIRCLE!

Теперь перейдем к описанию реляционной операции, которая обозначается оператором MINUS. Вначале рассмотрим выражение RX MINUS RY. Должно быть очевидно, что некоторые значения A в результатах этой операции должны принадлежать к типу ELLIPSE, а не CIRCLE, и поэтому объявленный тип атрибута A в этих результатах также должен представлять собой тип ELLIPSE. А что насчет выражения RY MINUS RX? Очевидно, что каждое значение A в результатах этой последней операции должен принадлежать к типу CIRCLE, и поэтому снова можно предположить, что объявленным типом атрибута A в этих результатах должен быть CIRCLE, а не ELLIPSE. Но следует отметить, что выражение RX INTERSECT RY логически эквивалентно не только выражению

RX MINUS (RX MINUS RY), как уже было отмечено выше, но также и выражению RY MINUS (RY MINUS RX), с учетом того факта, что объявление типа A в результатах выражения RY MINUS RX как CIRCLE приводит к противоречию. Из этого следует, что объявленным типом атрибута результата для выражения RY MINUS RX также должен быть ELLIPSE, а не CIRCLE, даже несмотря на то, что каждое значение этого атрибута должно фактически принадлежать к типу CIRCLE.

Наконец, рассмотрим выражение RX UNION RY. В данном случае должно быть очевидно, что в общем результаты будут включать некоторые значения A наиболее конкретного типа ELLIPSE, и поэтому объявленным типом атрибута A в этих результатах должен обязательно быть ELLIPSE. Таким образом, объявленным типом атрибута результата для оператора UNION также должен быть ELLIPSE (НО В данном конкретном случае, в отличие от случаев с использованием операторов JOIN, INTERSECT И MINUS, такое решение уже вряд ли можно назвать противоречащим здравому смыслу).

Поэтому может быть сформулировано общее правило, приведенное ниже.

- Допустим, что rx и ry - отношения с общим атрибутом A, а объявленными типами A в отношениях rx и ry являются, соответственно, DT (AX) И DT (Ay). Рассмотрим соединение отношений rx и ry (обязательно по атрибуту A, применяемому, по крайней мере, как один из атрибутов соединения). Типы DT (Ax) и DT (Ay) должны иметь общий супертип t, поскольку в противном случае такое соединение будет недопустимым (проверка этого условия осуществляется на этапе компиляции). Если подобное соединение является допустимым, объявленным типом A в результате становится наиболее конкретный из всех таких общих супертипов t.

Аналогичные правила распространяются на операции объединения, пересечения и разности, причем в каждом случае должны учитываться следующие условия: во-первых, соответствующие атрибуты операндов должны быть такими, чтобы их объявленные типы имели общий супертип t, и, во-вторых, объявленным типом соответствующего атрибута в результатах становится наиболее конкретный из всех таких общихсупертипов t.

Проверка типа

В разделе 20.3 приведен фрагмент кода, в котором используются операторы в форме IS_SQUARE, IS_CIRCLE и т.д. для проверки того, принадлежит ли заданное значение к некоторому указанному типу. В данном подразделе такие операторы рассматриваются более подробно. Прежде всего предполагается, что определение некоторого типа t вызывает автоматическое определение оператора с истинностным значением в следующей форме.

```
IS_T ( X )
```

Это выражение принимает значение TRUE, если выражение X принадлежит к типу t, а в противном случае принимает значение FALSE. Например, если c — переменная с объявленным типом CIRCLE, то оба приведенные ниже выражения принимают значение TRUE.

```
IS CIRCLE ( C
) IS ELLIPSE (
C )
```

Если же E — переменная с объявленным типом ELLIPSE, а текущим наиболее конкретным типом является некоторый подтип CIRCLE, то следующее выражение также принимает значение TRUE.

```
IS_CIRCLE ( E )
```

Проверка типа имеет также смысл и для реляционных операторов. Рассмотрим следующий пример. Допустим, что переменная отношения R имеет атрибут A объявленного типа ELLIPSE, и предположим, что необходимо получить те кортежи R, в которых значение A действительно представляет собой окружность, а радиус этой окружности больше двух. Для этого может быть предпринята попытка применить приведенное ниже выражение.

```
R WHERE THE_R ( A ) > LENGTH (2.0)
```

Но обработка этого выражения окончится неудачей из-за ошибки при проверке типа на этапе компиляции, поскольку для оператора THE_R требуется фактический параметр типа CIRCLE, а объявленным типом атрибута A является ELLIPSE, а не CIRCLE. (Безусловно, если бы эта проверка на этапе компиляции не была выполнена, то вместо этого была бы обнаружена ошибка при проверке типа на этапе прогона, как только встретился бы кортеж, в котором значение A было бы эллипсом, а не окружностью.) Очевидно, что для успешного решения этой задачи необходимо исключить из рассмотрения те кортежи, в которых значение A представляет собой эллипс, еще до того, как будет предпринята сама попытка проверить радиус. А именно, такая операция осуществляется, если используется приведенная ниже формулировка.

```
R : IS_CIRCLE ( A ) WHERE THE_R ( A ) > LENGTH (2.0)
```

Это выражение определено (выражаясь неформально) как средство получения таких кортежей, в которых значением A являются окружности с радиусом больше двух. Точнее, данное выражение возвращает отношение со следующими свойствами:

а) оно имеет такой же заголовок, как и отношение R, за исключением того, что объявленным типом атрибута A в этих результатах является CIRCLE, а не ELLIPSE;

б) тело этого отношения состоит только из таких кортежей R, в которых значение A принадлежит к типу CIRCLE, а радиус рассматриваемой окружности больше двух.

Иными словами, автор предлагает использовать новый реляционный оператор, имеющий следующую форму.

```
R : IS_T ( A )
```

Здесь R— реляционное выражение, а A— атрибут соответствующего отношения (скажем, r), обозначенного этим выражением. Значение всего данного выражения определено как отношение, обладающее следующими свойствами:

а) оно имеет такой же заголовок, как и отношение r, за исключением того, что объявленным типом атрибута A в этих результатах является t;

б) тело этого отношения состоит только из таких кортежей r, в которых значение A принадлежит к типу t, если не учитывать того, что объявленным типом атрибута A в каждом из этих кортежей является t.

Примечание. В [3.3] определены обобщенные формы обоих операторов, введенных в данном подразделе, например, обобщенная форма оператора IS_T, который проверяет, принадлежит ли один операнд к такому же типу, как и другой, а не просто проверяет, принадлежит ли он к некоторому явно указанному типу.

20.7. ОПЕРАТОРЫ, ВЕРСИИ И СИГНАТУРЫ

Как было отмечено в разделе 20.3, каждый конкретный оператор может иметь много разных версий реализации, которые скрыты от пользователя. Это означает, что по мере продвижения по пути от некоторого супертипа т к некоторому подтипу T' в иерархии типов необходимо иметь (по многим причинам), по меньшей мере, право на повторную реализацию операторов типа т для типа T'. В качестве примера рассмотрим следующий оператор.

```
OPERATOR MOVE ( E ELLIPSE, R RECTANGLE ) RETURNS ELLIPSE
   VERSION ER MOVE ;
   RETURN ( ELLIPSE ( THE A ( E ) , THE B ( E ) ,
                                    R CTR ( R ) ) )
; END OPERATOR ;
```

Оператор MOVE, неформально выражаясь, "передвигает" эллипс E таким образом, чтобы центр его совпадал с центром прямоугольника R или, точнее, он возвращает такой же эллипс, как и эллипс, заданный в виде фактического параметра, соответствующего формальному параметру E, за исключением того, что центр его находится в центре прямоугольника, заданного в качестве фактического параметра, соответствующего формальному параметру R. Обратите внимание на спецификацию версии VERSION во второй строке, в которой вводится различимое имя ER_MOVE для данной конкретной версии MOVE (вскоре будет определена еще одна версия этого оператора). Следует также отметить, что подразумевается наличие оператора R_CTR, который возвращает координаты центра указанного прямоугольника.

Теперь определим еще одну версию оператора MOVE, предназначенную для перемещения окружностей, а не эллипсов[10], которая приведена ниже.

```
OPERATOR MOVE ( C CIRCLE, R RECTANGLE ) RETURNS CIRCLE
   VERSION CR MOVE ;
   RETURN ( CIRCLE ( THE R ( C ), R CTR ( R ) ) )
; END OPERATOR ;
```

Аналогичным образом может быть определена версия оператора MOVE для того случая, когда фактические параметры относятся, соответственно, к наиболее конкретным типам ELLIPSE и SQUARE (скажем, ES_MOVE), и еще одну версию для того случая, когда фактические параметры, соответственно, относятся к наиболее конкретным типам CIRCLE И SQUARE (скажем, CS_MOVE).

---

[10] В данном конкретном примере определение такой версии фактически имеет мало смысла (объясните, почему).

Сигнатуры

Термин *сигнатура* (signature) неформально определяется как сочетание имени некоторого оператора и типов операндов рассматриваемого оператора. (Но, кстати, следует отметить, что разными авторами и в разных языках этому термину даются немного иные толкования. Например, иногда как часть сигнатуры рассматривается тип результата, кроме того, как принадлежащие к сигнатуре определяются имена операндов и результата.) Но еще раз подчеркнем, что необходимо очень тщательно учитывать различия между следующими понятиями:

а) фактические параметры и формальные параметры;
б) объявленный тип и наиболее конкретный тип;
в) операторы, рассматриваемые с точки зрения пользователя и с позиций системы (в последнем случае подразумеваются версии реализации этих операторов, существ вование которых скрыто от пользователя, как было описано выше).

Фактически можно провести различие между по меньшей мере тремя разновидностями сигнатур, которые связаны с любым конкретным оператором Op (хотя в литературе такое различие часто не учитывается!), — уникальная сигнатура спецификации (specification signature), множество сигнатур версий (version signature) и множество сигнатур вызовов (invocation signature). Эти три разновидности сигнатур подробно описаны ниже.

- Уникальная **сигнатура спецификации** состоит из имени оператора Op наряду с объявленными типами, взятыми в том же порядке, что и формальные параметры оператора Op, которые заданы пользователем в объявлении оператора Op. Такая сигнатура соответствует оператору Op, рассматриваемому в том виде, в каком он выглядит с точки зрения пользователя. Например, сигнатурой спецификации для приведенного выше оператора MOVE является просто MOVE (ELLIPSE, RECTANGLE).

  Примечание. **В** [3.3] изложено предложение, чтобы была предусмотрена возможность отделить определение сигнатуры спецификации для любого конкретного оператора от определений всех версий реализации этого оператора. Основная идея состоит в том, что должна быть обеспечена поддержка **объединенных типов** (иногда называемых также *абстрактными* или *неконкретизируемыми* типами, а иногда просто *интерфейсами);* под этим подразумеваются типы, которые вообще не могут стать наиболее конкретным типом любого значения. Такой тип предоставляет способ определения операторов, применяемых к нескольким разным обычным типам, притом что все они являются строгими подтипами рассматриваемого объединенного типа. В таком случае версии реализации такого оператора могут быть определены для каждого из этих обычных подтипов. Если речь идет о примере, который рассматривается на протяжении всей данной главы, то в этом смысле в качестве объединенного типа вполне может рассматриваться PLANE_FIGURE; в таком случае сигнатура спецификации оператора AREA вполне может быть определена на уровне типа PLANE_FIGURE, ЧТО дает возможность после этого определить явные версии реализации для типа ELLIPSE, типа POLYGON и т.д.

- Каждая версия реализации оператора Op имеет свою собственную **сигнатуру версии,** состоящую из имени оператора Op наряду со взятыми в том же порядке объявленными типами формальных параметров, которые определены для данной

версии. Эти сигнатуры соответствуют различным фрагментам кода реализации, в котором реализованы версии оператора Op, скрытые от пользователя. Например, сигнатурой версии для версии CR_MOVE оператора MOVE должна быть MOVE(CIRCLE, RECTANGLE).

Каждое возможное сочетание наиболее конкретных типов фактических параметров имеет свою собственную **сигнатуру вызова,** состоящую из имени оператора Op наряду со взятыми в том же порядке наиболее конкретными типами фактических параметров. Эти сигнатуры соответствуют всем возможным вызовам оператора Op (безусловно, что это соответствие относится к типу "один ко многим"; это означает, что одна сигнатура вызова может соответствовать многим фактическим вызовам). Например, предположим, что переменные E и R имеют, соответственно, наиболее конкретные типы CIRCLE и SQUARE. Тогда сигнатурой вызова для вызова MOVE (E, R) оператора MOVE является MOVE (CIRCLE , SQUARE).

Поэтому различные сигнатуры вызова, относящиеся к одному и тому же оператору, соответствуют (по меньшей мере, потенциально) различным версиям реализации оператора, скрытым от пользователя. Таким образом, если действительно существуют несколько версий одного и того же оператора, скрытых от пользователя, то принятие решения о том, какая версия должна быть вызвана в каждом конкретном случае, зависит от того, какая сигнатура версии является "в наибольшей степени соответствующей" применимой сигнатуре вызова. Процесс выработки решения об этом наилучшем соответствии (т.е. процесс выработки решения о том, какая версия должна быть вызвана) и является тем процессом связывания на этапе прогона, о котором уже было сказано в разделе 20.3.

Кстати, следует отметить, что, во-первых, сигнатуры спецификации в действительности представляют собой понятие уровня модели; во-вторых, сигнатуры версии — это понятие уровня реализации; в-третьих, сигнатуры вызова, хотя и представляющие собой в определенной степени понятие уровня модели, фактически являются прежде всего просто логическим следствием основной идеи наследования типов (как и понятие заменяемости). На самом деле нельзя оспаривать тот вывод, что возможны разные сигнатуры вызова, поскольку он в действительности просто следует из понятиязаменяемости.

Сравнение операторов, обеспечивающих только чтение, и операторов обновления

Вплоть до этого момента рассматриваемый в данной главе оператор MOVE принадлежал к типу именно такого оператора, который обеспечивает только чтение. Но предположим, что данный оператор вместо этого необходимо сделать оператором обновления, как показано ниже.

```
OPERATOR MOVE ( E ELLIPSE, R RECTANGLE ) UPDATES
   E VERSION ER MOVE ;
      THE CTR ( E )  := R CTR ( R
) ; END OPERATOR ;
```

(Следует напомнить, что операторы, обеспечивающие только чтение, и операторы обновления иногда именуются, соответственно, *наблюдателями* и *модификаторами.* Дополнительные сведения о том, в чем состоят различия между этими разновидностями операторов, приведены в главе 5.)

Теперь отметим, что вызов этой версии оператора MOVE приводит к обновлению его первого фактического параметра (неформально выражаясь, в результате этого вызова "изменяется центр" объекта, заданного с помощью данного фактического параметра). Отметим также, что эта операция обновления действует успешно независимо от того, принадлежит ли этот первый фактический параметр к наиболее конкретному типу ELLIPSE или к наиболее конкретному типу CIRCLE; иными словами, явно заданная версия реализации для окружностей больше не требуется". Поэтому одним из преимуществ операторов обновления в целом является то, что они позволяют исключить необходимость явно разрабатывать определенные версии реализации. Обратите внимание на то, какое значение это имеет, в частности, для сопровождения программ; например, что произойдет, если в дальнейшем будет введен тип O_CIRCLE как подтип CIRCLE? *(Ответ.* Вызов оператора MOVE с использованием в качестве фактического параметра переменной с объявленным типом ELLIPSE или CIRCLE, но с текущим наиболее конкретным типом O_CIRCLE, будет выполнен вполне успешно. Но в общем вызов этого оператора с использованием в качестве фактического параметра переменной с объявленным типом O_CIRCLE не будет выполнен успешно.)

Изменение семантики оператора

Тот факт, что всегда, по меньшей мере, допустимой является повторная реализация операторов по ходу продвижения вниз по иерархии типов, имеет одно очень важное следствие — он открывает возможность изменения семантики рассматриваемого оператора. Например, в случае оператора AREA может оказаться, что реализация для типа CIRCLE фактически возвращает, скажем, периметр рассматриваемой окружности вместо площади. (Тщательное проектирование типа позволяет в определенной степени смягчить остроту этой проблемы; например, если оператор AREA определен как возвращающий результат типа AREA, то, безусловно, данная реализация не может вместо этого возвращать результат типа LENGTH. Но эта реализация все равно будет способна возвращать неправильное значение площади!)

Но, хотя это на первый взгляд кажется удивительным, можно утверждать (и фактически такие заявления были сделаны), что изменение семантики указанным способом может оказаться желательным. Например, предположим, что тип TOLL_HIGHWAY (Платное шоссе)— строгий подтип типа HIGHWAY (Шоссе), а TRAVEL_TIME (Время проезда) — оператор, который вычисляет количество времени, необходимое для проезда между двумя указанными пунктами на указанном шоссе. Для платного шоссе эта формула выглядит как (d/s) + (n*t), где d — расстояние, s — скорость, n — количество пунктов сбора платы за проезд и t — время, проводимое в каждом пункте сбора. В отличие от этого, для бесплатного шоссе эта формула выглядит просто как d/s. Изменение семантики позволяет использовать один и тот же оператор TRAVEL_TIME для шоссе обоих типов.

В качестве контрпримера (т.е. примера ситуации, в которой изменение семантики, безусловно, является нежелательным) еще раз рассмотрим эллипсы и окружности. Предположим, что оператор AREA должен быть определен таким образом, чтобы каждая конкретная

---

[11] Как было отмечено в сноске 10, такая версия фактически не требовалась и в случае оператора, обеспечивающего только чтение; она была приведена исключительно для использования в рассматриваемом примере.

окружность имела одну и ту же площадь, независимо от того, рассматривается ли она именно как окружность или просто как разновидность эллипса. Иными словами, допустим, что происходит описанная ниже последовательность событий.

1. Определены тип ELLIPSE и соответствующая версия оператора AREA. Для упрощения предположим, что в коде AREA не используется физическое представление для эллипсов.
2. Определен тип CIRCLE как подтип типа ELLIPSE, но (еще) не определена отдельная версия реализации оператора AREA для окружностей.
3. Вызывается оператор AREA с указанием некоторой конкретной окружности с для получения результата, скажем, al. В этом вызове используется версия оператора AREA, предназначенная для типа ELLIPSE (поскольку это — единственная версия, которая существует в настоящее время).
4. Затем будет определена отдельная версия реализации оператора AREA для окружностей.
5. Снова вызывается оператор AREA С указанием той же конкретной окружности с для получения результата, скажем, a2 (и на этот раз вызывается именно та версия оператора AREA, которая относится к типу CIRCLE).

В данный момент, безусловно, желательно, чтобы соблюдалось условие a2 = al. Но такому необязательному пожеланию нельзя придать силу закона; это означает, что (как уже было сказано выше) всегда существует такая возможность, что версия оператора AREA, реализованная для использования с окружностями, может возвращать (скажем) периметр вместо площади или просто неправильное значение площади.

Еще раз вернемся к примеру с оператором TRAVEL_TIME. Дело в том, что автор рассматривает данный пример и другие подобные примеры как крайне неубедительные (это означает, что нет ни одного примера, который мог бы служить примером такой ситуации, в которой изменение семантики любого оператора может рассматриваться как желательное). Это следует из приведенных ниже соображений.

- Если тип TOLL_HIGHWAY действительно является подтипом HIGHWAY, это по определению означает, что каждое отдельное платное шоссе фактически представляет собой просто шоссе.
- Поэтому некоторые шоссе (т.е. некоторые значения типа HIGHWAY) действительно представляют собой платные шоссе — на них установлены пункты сбора платы за проезд. Поэтому сам тип HIGHWAY нельзя назвать типом, который описывает "шоссе без пунктов сбора платы за проезд"; он описывает "шоссе с п пунктами сбора платы за проезд" (где п может быть равно нулю).
- Поэтому оператор TRAVEL_TIME для типа HIGHWAY не "вычисляет время проезда по шоссе без пунктов сбора платы за проезд", а " вычисляет время проезда d/s по шоссе без учета пунктов сбора платы за проезд".
- В отличие от него, оператор TRAVEL_TIME ДЛЯ типа TOLL_HIGHWAY предназначен для "вычисления времени проезда (d/s) + (n*t) по шоссе с учетом пунктов сбора платы за проезд". Поэтому на самом деле два оператора TRAVEL_TIME ЯВЛЯЮТСЯ логически разными! В данном случае путаница возникла из-за того, что двум разным операторам присвоено одно и то же имя; фактически в данном случае

мы имеем дело с полиморфизмом перегрузки, а не полиморфизмом включения. (В качестве побочного замечания можно указать, что на практике может возникать еще более серьезная путаница, поскольку, к сожалению, многие авторы используют термин *перегрузка,* когда речь идет фактически о полиморфизме включения.)

Подведем итог. Автор все еще ни разу не встретил убедительных доводов в пользу того, что изменение семантики является целесообразным. Как было описано выше, требованию по поддержке единой семантики нельзя придать силу закона; но, безусловно можно определить свою собственную модель наследования (и автор решил эту задачу), позволяющую утверждать, что если произошло изменение семантики, то нарушена и реализация (т.е. такая реализация не является реализацией модели и последствия становятся непредсказуемыми). Необходимо отметить, что позиция автора по этому вопросу (которая заключается в том, что подобные изменения семантики являются недопустимыми), имеет свое преимущество в том, что восприятие пользователем каждого конкретного оператора Op остается одинаковым, независимо от того, какие бы неявные уточнения этого оператора не были определены. А именно, с точки зрения пользователя, во-первых, существует некоторый оператор (единственный оператор), называемый Op, и во-вторых, этот оператор Op применяется к значениям фактических параметров некоторого указанного типа т и поэтому, по определению, к значениям фактических параметров любого строгого подтипа типа т.

20.8. АНАЛИЗ ВЗАИМОДЕЙСТВИЯ МЕЖДУ ТИПАМИ И ПОДТИПАМИ НА ПРИМЕРЕ ОКРУЖНОСТЕЙ И ЭЛЛИПСОВ

Могут ли экземпляры подтипов рассматриваться как экземпляры типов, например, действительно ли окружности являются эллипсами? До сих пор в данной главе (на вполне резонных основаниях!) предполагалось, что ответ на этот вопрос является положительным, но теперь необходимо признать тот факт, что по этому вопросу, который внешне кажется однозначным, в литературе можно найти совершенно противоположные мнения [20.6]. Рассмотрим обычно используемые в этой главе переменные E и c, которые имеют, соответственно, объявленные типы ELLIPSE и CIRCLE. Предположим, что эти переменные были инициализированы следующим образом.

```
E := ELLIPSE ( LENGTH ( 5.0 ), LENGTH ( 3.0 ),
                                  POINT ( 0.0, 0.0 ) )
; C := CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0
) ) ;
```

Отметим, в частности, что теперь оба оператора, THE_A (C) и THE_B (C) , возвращают значение пять.

Затем рассмотрим одну операцию, которую можно, безусловно, применить к переменной E — "обновление значения полуоси a", например, как показано ниже.

```
THE_A ( E ) := LENGTH ( 6.0 ) ;
```

Но если теперь будет предпринята попытка выполнить аналогичную операцию с переменной c (как показано ниже), то будет получена ошибка!

```
THE_A ( C ) := LENGTH ( 6.0 ) ;
```

Но в чем именно заключается эта ошибка? Дело в том, что если бы было выполнено это обновление, переменная c в конечном итоге содержала бы "окружность", нарушающую

ограничение, налагаемое на окружности в той части, которая касается условия a = b; а именно: компонент a теперь имел бы значение шесть, тогда как компонент b, безусловно, был бы по-прежнему равен пяти (поскольку в него не были внесены изменения). Иными словами, переменная c после этого содержала бы "некруглую окружность", нарушая тем самым ограничение типа, наложенное на тип CIRCLE.

Поскольку "некруглые окружности" противоречат логике и здравому смыслу, то может показаться резонным такое предложение, чтобы подобные обновления прежде всего не были бы разрешены. И очевидным способом достижения этой цели является отбраковка подобных операций на этапе компиляции путем определения того, что обновление полуоси a или b окружности (т.е. присваивание ей нового значения) является синтаксически недопустимым. Иными словами, операция присваивания значений псевдопеременной THE_A или THE_B не применима к типу CIRCLE и попытка подобного обновления должна оканчиваться неудачей из-за ошибки при проверке типов на этапе компиляции.

Примечание. В действительности очевидно, что подобные присваивания и должны быть синтаксически недопустимыми. Напомним, что операция присваивания псевдопеременной THE_ фактически представляет собой просто сокращение. Поэтому, например, попытка присваивания псевдопеременной THE_A(C), если бы была допустимой, то представляла бы собой сокращенное обозначение примерно следующего оператора.

```
C := CIRCLE ( ... ) ;
```

При этом вызов селектора CIRCLE в правой части должен был бы содержать фактический параметр THE_A со значением LENGTH (6 . 0) . Но вызов селектора CIRCLE не включает фактический параметр THE_A! В нем предусмотрен только фактический параметр THE_R и фактический параметр THE_CTR. Поэтому первоначальное присваивание, безусловно, является недопустимым.

Дополнительные сведения об изменении семантики

Прежде всего, следует немедленно отвергнуть предложение, которое могло быть сделано в попытке спасти от краха идею, что в конечном итоге присваивание псевдопеременной THE_A или THE_в должно быть допустимым для окружностей. Такое предложение состоит в том, что операцию присваивания (например) псевдопеременной THE_A необходимо переопределить (иными словами, реализовать новую версию этого оператора) для окружностей таким образом, чтобы в качестве побочного эффекта происходило также присваивание псевдопеременной THE_B, т.е. чтобы окружность по-прежнему удовлетворяла бы ограничению a = b после обновления. Автор отвергает это предложение по меньшей мере по трем указанным ниже причинам.

- Во-первых, семантика присваивания псевдопеременным THE_A и THE_B является предписанной (вполне сознательно!) в разработанной автором модели и ее не следует изменять в предложенной форме.
- Во-вторых, даже если бы эта семантика не была предписана моделью, в этой книге уже было указано, что в целом не рекомендуется изменять произвольным образом семантику любого оператора, а тем более не рекомендуется изменять семантику любого оператора так, чтобы в результате возникали побочные эффекты. Обоснованный общий принцип состоит в том, что следует неуклонно стремиться к тому, чтобы операторы создавали один и только один требуемый эффект, не больше и не меньше.

- В-третьих, наиболее важным возражением является то, что возможность изменения семантики в предложенной форме иногда даже полностью отсутствует. Например, предположим, что тип ELLIPSE имеет еще один непосредственный подтип, NONCIRCLE; допустим, что к фигурам, отличным от окружностей, принадлежащим к этому подтипу, относится ограничение a > b, и рассмотрим операцию присваивания псевдопеременной THE_A одной из таких фигур, которая в случае ее успешного выполнения установила бы значение a, равное b. Каким было бы приемлемое переопределение семантики для такого присваивания? Вернее, какой побочный эффект был бы приемлемым?

Поиск применимой модели наследования

Итак, в предыдущих подразделах мы остановились на рассмотрении такой ситуации, что операция присваивания псевдопеременной THE_A или THE_B является операцией, которая может применяться к эллипсам в целом, но не к окружностям в частности. Но необходимо также учитывать приведенные ниже соображения.

а) Предполагается, что тип CIRCLE — это подтип типа ELLIPSE.

б) Из того, что тип CIRCLE — подтип типа ELLIPSE, следует, что операции, приме нимые к эллипсам в целом, являются применимыми и к окружностям в частности (иными словами, операции с эллипсами наследуются окружностями).

в) А теперь мы в конечном итоге утверждаем, что операция присваивания псевдопе ременной THE_A или THE_B не унаследована.

Разве здесь мы не сталкиваемся с противоречием? Что же в конце концов происходит?

Прежде чем попытаться найти ответы на эти вопросы, необходимо подчеркнуть серьезность рассматриваемой проблемы. Приведенный выше довод действительно выглядит как парадокс, поскольку если некоторые операторы не наследуются типом CIRCLE от типа ELLIPSE, то на основании чего именно мы можем утверждать, что окружность является эллипсом? А какой смысл имеет понятие *наследование,* если некоторые операторы в конечном итоге вообще не наследуются? Существует ли вообще применимая модель наследования? Не гоняемся ли мы за призраками, пытаясь найти такую модель?

Примечание. Некоторые авторы вполне серьезно предлагали, чтобы операция присваивания псевдопеременной THE_A была допустимой как для окружностей, так и для эллипсов (применительно к окружностям она должна была обновлять радиус), а операция присваивания псевдопеременной THE_B должна была быть применимой только для эллипсов и поэтому фактически тип ELLIPSE должен был представлять собой подтип типа CIRCLE! Иными словами, эти авторы предлагали развернуть иерархию типов в противоположном направлении. Но достаточно задуматься лишь на мгновение, чтобы понять, что указанная идея является неприменимой, поскольку, в частности, нарушается принцип заменяемости (например, что такое радиус эллипса общего вида?).

Именно такие соображения, какие были описаны выше, привели некоторых авторов к выводу, что в действительности никакой применимой модели наследования не существует (см. аннотацию к [20.2]). Другие авторы предложили модели наследования с такими средствами, которые противоречат здравому смыслу или, безусловно, являются нежелательными. Например, как показано в разделе 20.10, стандарт SQL допускает наличие "некруглых окружностей" и других подобных абсурдных объектов. (В стандарте SQL

фактически не поддерживаются ограничения типов, а именно это упущение и становится причиной, по которой этот стандарт допускает существование подобных нелепостей. Об этом также сказано в разделе 20.10.)

Решение задачи определения модели наследования

Подводя итог описанной выше ситуации, можно отметить, что мы столкнулись с описанной ниже дилеммой.

- Если окружности наследуют операторы "присваивания псевдопеременным THE_A и THE_в" от эллипсов, то могут создаваться "некруглые окружности".
- Способ предотвращения появления "некруглых окружностей" состоит в поддерж ке ограничений типа.
- Но если поддерживаются ограничения типа, то операторы не могут быть унасле дованы.
- Итак, оказывается, что наследования в конечном итоге вообще не существует!

Как же разрешить эту дилемму?

Для этого может применяться способ (который часто позволяет выйти из сложной ситуации при обсуждении вопросов теории реляционных баз данных), состоящий в том, что следует признать факт существования важного *логического различия между значениями и переменными* и действовать в соответствии с этим фактом. Под выражением "каждая окружность является эллипсом" подразумевается именно то, что каждое значение окружности является значением эллипса. Таким образом, это выражение, безусловно, не означает, что каждая переменная окружности является переменной эллипса (переменная с объявленным типом CIRCLE — это не переменная с объявленным типом ELLIPSE и не может содержать значение наиболее конкретного типа ELLIPSE). Иными словами, **наследование применяется к значениям, а не к переменным.** Например, в случае эллипсов и окружностей справедливы приведенные ниже утверждения.

- Как уже было отмечено, каждое значение окружности представляет собой значе ние эллипса.
- Поэтому все операции, применимые к значениям эллипсов, применимы также и к значениям окружностей.
- Но ни с одним значением нельзя выполнить одно действие — изменить его! Если бы мы могли изменять значение, то оно больше не было бы значением. (Безусловно, мы можем "изменить текущее значение" переменной, обновляя эту переменную, но еще раз повторяемом *не можем изменить значение как таковое.)*

Итак, все операции, применяемые к значениям эллипсов, являются именно операциями только чтения, которые определены для типа ELLIPSE, а операции, обновляющие переменные типа ELLIPSE, безусловно, являются операциями обновления, определенными для этого типа. Поэтому приведенное выше утверждение, что "наследование применяется к значениям, а не к переменным", может быть сформулировано более точно, как показано ниже.

- **Операции только чтения наследуются значениями и поэтому, в силу самого этого факта, текущими значениями переменных** (поскольку очевидно, что операции толь ко чтения могут на законных основаниях применяться к тем значениям, которые оказались текущими значениями переменных).

Эта более точная формулировка позволяет также объяснить, почему понятия полиморфизма и заменяемости относятся именно к значениям, а не к переменным. Например (просто, чтобы освежить это в памяти), отметим, что согласно определению заменяемости, всегда можно заменить **значение** типа т, ожидаемое системой, **значением** типа Т', где Т' — подтип типа т (полужирным шрифтом отмечены ключевые слова этого определения). И действительно, данный принцип при первом знакомстве с ним был представлен именно как *принцип заменяемости* **значений** (здесь также полужирным шрифтом отмечено ключевое слово).

Что в таком случае можно сказать об операциях обновления? По определению, такие операции относятся к переменным, а не к значениям. Но означает ли это, что можно утверждать, будто операции обновления, применимые к переменным типа ELLIPSE, наследуются переменными типа CIRCLE?

Скорее всего, такое утверждение будет неправильным, вернее, не совсем правильным. Например, операция присваивания псевдопеременной THE_CTR может применяться к переменным обоих объявленных типов, но (как было отмечено выше) операция присваивания псевдопеременной THE_A таковой не является. Итак, наследование операций обновления должно быть условным; в действительности, необходимо явно определять, какие именно операции обновления наследуются. В качестве примера можно указать приведенные ниже условия наследования.

- Переменные объявленного типа ELLIPSE имеют операторы обновления MOVE (это — версия, предназначенная для обновления) и операторы присваивания псевдопеременным THE_A, THE_B и THE_CTR.
- Переменные объявленного типа CIRCLE имеют операторы обновления MOVE (это — версия, предназначенная для обновления) и операторы присваивания псевдопеременным THE_CTR и THE_R, но не псевдопеременной THE_A или THE_B.

Примечание. Оператор MOVE рассматривался в предыдущем разделе.

Безусловно, если некоторая операция обновления наследуется, то мы имеем дело с той разновидностью полиморфизма и той разновидностью заменяемости, которые относятся к переменным, а не к значениям. Например, версия оператора MOVE, предназначенная для обновления, принимает фактический параметр, который представляет собой переменную объявленного типа ELLIPSE, но этот оператор можно вызвать и с фактическим параметром, который вместо этого представляет собой переменную объявленного типа CIRCLE (но не с фактическим параметром, являющимся переменной объявленного типа O_CIRCLE!). Поэтому мы можем (и должны) с полным правом обсуждать еще один принцип — *принцип заменяемости* **переменных,** но следует учитывать, что этот принцип является более ограничительным, *чем принцип заменяемости* **значений,** описанный выше.

20.9. ДОПОЛНИТЕЛЬНАЯ ИНФОРМАЦИЯ ОБ УТОЧНЕНИИ С ПОМОЩЬЮ ОГРАНИЧЕНИЙ

В данном разделе дано небольшое, но важное послесловие, которое необходимо добавить к описаниям, приведенным в предыдущих разделах. Оно касается таких примеров, как следующий: "Допустим, что тип CIRCLE имеет строгий подтип, называемый COLORED_CIRCLE" (под этим подразумевается, что определяемые этим подтипом

"цветные окружности" являются частным случаем окружностей в целом). Примеры такого общего характера часто можно встретить в литературе. Но следует отметить, что, по мнению автора, такие примеры, по некоторым важным причинам, являются крайне неубедительными и даже вносящими путаницу. А именно, в отношении, скажем, данного конкретного случая можно отметить, что фактически не имеет смысла рассматривать цветные окружности как некоторый частный случай окружностей в целом. В конце концов, "цветные окружности" должны по определению представлять собой изображения, возможно, на экране дисплея, тогда как окружности в целом являются не изображениями, а геометрическими фигурами. Поэтому представляется более разумным рассматривать тип COLORED_CIRCLE не как подтип типа CIRCLE, а скорее как полностью отдельный тип[12]. Этот отдельный тип вполне может иметь такое возможное представление, в котором один компонент относится к типу CIRCLE, а другой — к типу COLOR, но (еще раз отметим) он не должен определяться как подтип типа CIRCLE.

Наследование возможных представлений

Один из весомых доводов в поддержку изложенной выше позиции состоит в следующем. Прежде всего, вернемся на время к рассмотрению более привычного примера с эллипсами и окружностями. Ниже еще раз приведены определения типов (в виде набросков).

```
TYPE ELLIPSE ...
     POSSREP { A ..., B ..., CTR ... } ;

TYPE CIRCLE ...
     POSSREP { R ..., CTR ... } ;
```

Здесь заслуживает внимания, в частности, то, что эллипсы и окружности имеют разные объявленные возможные представления. Но возможные представления для эллипсов (обязательно, хотя и неявно) являются возможными представлениями также и для окружностей, поскольку окружности представляют собой эллипсы. Это означает, что окружности, безусловно, "могут быть заданы" с указанием их полуосей a и Ь (и центра), даже несмотря на тот факт, что в окружностях обе полуоси, a и b, являются одинаковыми. Разумеется, обратное утверждение нельзя назвать истинным; это означает, что возможное представление для окружностей не обязательно должно быть возможным представлением для эллипсов.

Из этого следует, что возможные представления, как и операторы и ограничения, могут рассматриваться в качестве дополнительных "свойств", которые наследуются окружностями от эллипсов или, вообще говоря, подтипами от супертипов[13]. Но (теперь

---

[12] Безусловно, COLORED_CIRCLE может быть подтипом типа CIRCLE точно в такой же степени, в какой он является подтипом типа COLOR (иначе говоря, он отнюдь не может быть подтипом типа CIRCLE).

[13] Автор не рассматривает их под этим углом в предложенной им формальной модели; это означает, что здесь такие унаследованные возможные представления не рассматриваются как объявленные, по скольку такое утверждение, что они были объявлены, может привести к противоречию. А именно, если бы мы утверждали, что тип CIRCLE наследует возможное представление типа ELLIPSE, то согласно [3.3] для этого потребовалось бы, чтобы операция присваивания псевдопеременной THE_A или THE_B для переменной объявленного типа CIRCLE была бы допустимой, а нам, безусловно, известно, что дело обстоит иначе. Поэтому утверждение, что тип CIRCLE наследует возможное представление типа ELLIPSE, — это просто манера высказывания, которая не несет формального смысла.

обратимся к случаю окружностей и цветных окружностей) должно быть ясно, что объявленное возможное представление для типа CIRCLE не может служить возможным представлением для типа COLORED_CIRCLE, поскольку в нем нет ничего, что было бы способно представлять цвет! Этот факт становится убедительным свидетельством в пользу того, что цветные окружности нельзя рассматривать как подтипы обычных окружностей в том же смысле, в каком, например, окружности рассматриваются как подтипы эллипсов.

Точное определение понятия подтипа

Приведенный ниже довод (в определенной степени) связан с предыдущим, но фактически является еще более весомым (т.е. весомым с точки зрения логики). Он состоит в том, что не существует способа получить цветную окружность из обычной окружности по принципу уточнения с помощью ограничения.

Для того чтобы было проще пояснить эту мысль, еще раз вернемся **к** случаю эллипсов и окружностей. Ниже снова приведены определения соответствующих типов, но на этот раз в полной форме.

```
TYPE ELLIPSE
     IS PLANE_FIGURE
        POSSREP { A LENGTH, B LENGTH, CTR
                  POINT CONSTRAINT A > B } ;

TYPE CIRCLE
     IS ELLIPSE
        CONSTRAINT THE A ( ELLIPSE ) = THE B (
        ELLIPSE ) POSSREP { R = THE A ( ELLIPSE ) ,
                  CTR = THE_CTR ( ELLIPSE ) } ;
```

Как известно, благодаря использованию уточнения с помощью ограничения спецификация CONSTRAINT для типа CIRCLE гарантирует, что эллипс со значениями a = b будет автоматически приведен к типу CIRCLE в результате уточнения. Но, вернувшись снова к примеру обычных окружностей и цветных окружностей, мы не сможем найти такую спецификацию CONSTRAINT для типа COLORED_CIRCLE, которая аналогичным образом позволила бы привести обычную окружность к типу COLORED_CIRCLE в результате уточнения; иными словами, не существует возможности применить такое ограничение типа, выполнение которого для данной конкретной окружности означало бы, что рассматриваемая окружность действительно является цветной окружностью.

Поэтому еще раз отметим, что более обоснованный подход, по-видимому, состоит в том, что типы COLORED_CIRCLE и CIRCLE должны рассматриваться как полностью различные и, в частности, тип COLORED_CIRCLE должен рассматриваться как имеющий возможное представление, в котором один компонент относится к типу CIRCLE, а другой — к типу COLOR, поэтому может применяться следующее определение.

```
TYPE COLORED_CIRCLE POSSREP { CIR CIRCLE, COL COLOR } ... ;
```

Здесь фактически мы коснулись гораздо более обширной темы. Дело обстоит таким образом, что, по мнению автора, определение подтипов должно всегда осуществляться по принципу уточнения с помощью ограничения! Это означает, что можно утверждать следующее: **если T' — подтип типа T, то всегда должно существовать такое ограничение типа, что если оно удовлетворяется для некоторого конкретного значения типа T, то это**

рассматриваемое значение фактически является значением типа T[1] (и должно автоматически приводиться к типу T' с помощью уточнения). Поэтому допустим, что T и T^1 -типы, а T' — подтип типа т (фактически мы можем предположить без потери точности, что T' является непосредственным подтипом типа т). В таком случае справедливы приведенные ниже утверждения.

- И тип т, и тип T', по сути, представляют собой множества (именованные множе ства значений), причем T' является подмножеством множества т.
- Поэтому и тип т, и тип T' имеют предикаты принадлежности к множеству; т.е. они имеют такие предикаты, что некоторое значение является элементом рассмат риваемого множества (и поэтому значением рассматриваемого типа) тогда и толь ко тогда, когда оно удовлетворяет рассматриваемому предикату. Допустим, что этими предикатами являются, соответственно, P и P'.
- Теперь отметим, что предикат P' по определению является таким предикатом, который может принимать значение TRUE только применительно к тем опреде ленным значениям, которые фактически представляют собой значения типа т. Поэтому в действительности этот предикат может быть сформулирован в терминах свойств значений типа т (а не в терминах свойств значений типа T').
- А предикат P', сформулированный в терминах свойств значений типа т, является не чем иным, как ограничением типа, которому должны отвечать значения типа T, для того чтобы стать значениями типа T'. Иными словами, любое значение типа T приводится к типу T' путем уточнения именно в том случае, если оно отвечает ог раничению P'.

Поэтому автор утверждает, что уточнение с помощью ограничения является единственным концептуально оправданным средством определения подтипов. Вследствие этого автор оспаривает возможность использования примеров, подобных приведенному выше, для доказательства того, что COLORED_CIRCLE может быть подтипом типа CIRCLE.

20.10. СРЕДСТВА ЯЗЫКА SQL

Явная поддержка наследования в языке SQL ограничивается (только) одинарным наследованием (только) для *структурированных типов;* в этом языке отсутствует явная поддержка наследования для сгенерированных типов, нет явной поддержки для множественного наследования и вообще не поддерживается наследование для встроенных типов ИЛИ ТИПОВ DISTINCT[14].

Ниже показан синтаксис определения структурированного типа (немного упро-. щенный).

[14] На основании этих замечаний можно сделать вывод, что в языке SQL имеется определенная неявная поддержка наследования сгенерированных типов и множественного наследования (и это также указано в предложениях, изложенных в [3.3], хотя последние предложения предусматривают более широкую поддержку). Но в данной главе наше внимание ограничивается только одинарным наследованием и скалярными типами.

```
CREATE TYPE <type name>
     [ UNDER <type name> ]
     [ AS <representation> ]
     [ [ NOT ] INSTANTIABLE ]
       NOT FINAL [ <method specification commalist> ] ;
```

А здесь представлены примеры возможных определений SQL для типов PLANE_FIGURE, ELLIPSE И CIRCLE.

```
CREATE TYPE
        PLANE_FIGURE NOT
        INSTANTIABLE NOT
        FINAL ;

CREATE TYPE ELLIPSE UNDER
   PLANE_FIGURE AS ( A LENGTH, B
   LENGTH, CTR POINT ) INSTANTIABLE
   NOT FINAL ;

CREATE TYPE CIRCLE UNDER
   ELLIPSE AS ( R LENGTH )
        INSTANTIABLE
        NOT FINAL ;
```

На основании приведенных определений можно сделать перечисленные ниже выводы (некоторые из них взяты из главы 5).

1. Ключевое слово NOT INSTANTIABLE означает, что рассматриваемый тип не имеет *экземпляров,* где под термином *экземпляр* (по большей части) подразумевается значение, для которого наиболее конкретным типом является рассматриваемый тип[15]. Иными словами, рассматриваемый тип представляет собой то, что в дан ной главе упоминается под названием объединенного типа. Ключевое слово INSTANTIABLE означает, что рассматриваемый тип имеет по меньшей мере один *экземпляр;* это означает, что он не представляет собой объединенный тип и поэтому существует по меньшей мере одно значение, наиболее конкретным типом которого является рассматриваемый тип. В данном примере тип PLANE_FIGURE определен как NOT INSTANTIABLE, а ТИПЫ ELLIPSE И CIRCLE — Как INSTANTIABLE (по умолчанию применяется ключевое слово INSTANTIABLE).
2. Как было отмечено в главе 5, должно быть обязательно указано ключевое слово NOT FINAL (хотя в спецификации SQL:2003, по-видимому, вместо него будет раз решено задавать альтернативное ключевое слово FINAL). Ключевое слово NOT FINAL означает, что для рассматриваемого типа разрешено определять строгие подтипы, а ключевое слово FINAL, когда оно будет поддерживаться, должно озна чать обратное.
3. Спецификация UNDER определяет непосредственный супертип данного типа (или, в терминологии SQL, прямой супертип — direct supertype), если он имеет ся. Поэтому, например, CIRCLE — это *прямой подтип* типа ELLIPSE и свойства,

[15] В [4.23] экземпляр определен как "физическое представление значения" (?!).

относящиеся к эллипсам в общем, безусловно, наследуются окружностями в частности. Но следует учитывать приведенные ниже замечания.

а) Здесь под "свойствами" (в отличие от предложенной автором модели наследования) не подразумеваются операторы и ограничения, и этим термином обозначаются операторы и структура (или представление). Иными словами, язык SQL поддерживает и наследование функций, и наследование структуры, поскольку внутренняя структура *структурированных типов* явно предоставляется в распоряжение пользователя (см. п. 5).

б) Здесь под термином *операторы* (в отличие от предложенной автором модели наследования) не подразумеваются лишь операторы, предназначенные только для чтения; этим термином обозначаются все операторы. Иными словами, в языке SQL не проводится должным образом различие между значениями и переменными, и этот язык требует безусловного наследования не только операций обновления, но и операций только чтения. Из этого следует, что в этом языке, например, окружности могут оказаться некруглыми, квадраты — неквадратными и т.д. (Рассмотрим это замечание более подробно. Модель, предложенная автором, характеризуется следующим: если некоторое значение v относится к наиболее конкретному типу ELLIPSE, то оно определенно представляет собой эллипс, а не окружность, а если оно относится к наиболее конкретному типу CIRCLE, то определенно представляет собой эллипс, являющийся окружностью, в том толковании, которое применяется в реальном мире. В отличие от этого, в языке SQL имеет место следующее: если значение v относится к наиболее конкретному типу ELLIPSE, то оно может фактически представлять собой окружность, а если оно относится к наиболее конкретному типу CIRCLE, то может фактически быть эллипсом, который не является окружностью; при этом также применяется толкование в терминах реального мира.)

в) Рассматриваемые операторы подразделяются на три категории: функции, процедуры и методы. Как было отмечено в главе 5, функции и процедуры примерно соответствуют операторам, предназначенным только для чтения, и операторам обновления; методы действуют аналогично функциям, но вызываются с использованием другого синтаксического стиля. Кроме того, функции и процедуры определяются с помощью отдельных операторов CREATE FUNCTION и CREATE PROCEDURE, а методы определяются в виде встроенных конструкций в составе соответствующего оператора CREATE TYPE, как указано в синтаксической структуре оператора CREATE TYPE (для упрощения в приведенных здесь примерах спецификации методов не применяются). Связывание на этапе компиляции (т.е. связывание на основе только объявленных типов) применяется к функциям и процедурам, а связывание на этапе прогона применяется к методам, но осуществляется на основе только одного из существующих фактических параметров (как это обычно принято в объектных системах; см. главу 25).

4. Аналогом термина *корневой тип* в языке SQL является *максимальный супертип* (maximal supertype), поэтому в данном примере PLANE_FIGURE — это максимальный супертип. (Но, как ни странно, в языке SQL для обозначения листового типа

применяется термин не *минимальный подтип,* а *листовой тип,* поэтому в рассматриваемом примере тип CIRCLE — это листовой тип.)

5. Опция <*representation*> с обозначением представления, если она задана, состоит из разделенного запятыми списка определения атрибутов <*attribute definition commalist*>, заключенного в круглые скобки, где атрибут <*attribute*> состоит из имени атрибута <*attribute name*>, за которым следует имя типа <type *name*>. Но следует отметить, что такое представление <*representation*> является действительным физическим представлением для значений рассматриваемого типа, а не *возможным представлением* (и поэтому пользователю предоставляется доступ к этим физическим представлениям, как уже было отмечено в п. 3). В частности, заслуживает внимания то, что в языке SQL невозможно определить два или больше разных представлений <*representation*> для одного и того типа.

   Примечание. Но, как было указано в главе 5, проектировщик типа может в конечном итоге скрыть тот факт, что представление <*representation*> является физическим благодаря продуманному выбору и проектированию операторов.

6. Для каждого атрибута <*attribute*> предусмотрены метод-наблюдатель (observer method) и метод-модификатор (mutator method), которые предоставляются автоматически и могут совместно использоваться для реализации функциональных возможностей, в определенной степени аналогичных возможностям операторов THE_, предусмотренным в языке Tutorial D (некоторые примеры на эту тему приведены в главе 5). Операторы селекторов не предоставляются автоматически, но для каждого типа предусмотрена предоставляемая автоматически функция конструктора, которая после ее вызова возвращает такое уникальное значение этого типа, что для любого и каждого атрибута устанавливается применимое значение по умолчанию. Как было сказано в главе 5, это значение должно быть пустым для любого атрибута, который, в свою очередь, относится к типу, определяемому пользователем. Поэтому, например, следующее выражение возвращает "эллипс", в котором и A, и B имеют "пустую длину", а CTR является "пустой точкой" (ее не следует, безусловно, путать с такой точкой, где оба компонента, X и Y, являются пустыми).

   ```
   ELLIPSE ()
   ```

   А приведенное ниже выражение возвращает эллипс, в котором длина полуоси a равна четырем, длина полуоси b — трем, а центр находится в начале координат.

   ```
   ELLIPSE () . A ( LENGTH () . L ( 4.0 ) ) . B
             ( LENGTH () . L ( 3.0 ) ) . CTR (
            POINT () . X ( 0.0 ) . Y ( 0.0 ) )
   ```

   (Здесь предполагается, что тип LENGTH имеет представление, состоящее только из одного атрибута L типа FLOAT.)

7. Следует отметить, что не существует способа определить множество допустимых значений данного типа; иными словами, не существует ограничений типа, за исключением тех априорных ограничений, которые определяются используемым

физическим представлением. На основании этого можно сделать вывод (в силу самого указанного факта), что в языке SQL практически не поддерживается наследование ограничений! Об этом уже было сказано в п. 3. 8. Каждый структурированный тип имеет связанный с ним ссылочный тип. Но мы не рассматриваем ссылочные типы в этой главе, не считая того замечания, что, наряду с предусмотренной в нем поддержкой ссылочных типов, язык SQL включает поддержку для *подтаблиц и супертаблиц.* Эти вопросы рассматриваются более ' подробно в главе 26.

Теперь перейдем к более подробному описанию поддержки наследования типов в языке SQL. Прежде всего, необходимо отметить, что язык SQL напоминает модель, предложенную автором (по меньшей мере, в том, что эта модель относится только к одинарному наследованию), поскольку этот язык опирается на предположение о непересечении, равносильное предположению, что наиболее конкретные типы являются уникальными. Кроме того, язык SQL поддерживает заменяемость (но поскольку в нем не проводится должным образом различие между значениями и переменными, с одной стороны, а также операциями только чтения и операциями обновления — с другой, язык SQL не позволяет должным образом провести различие между заменяемостью значений и заменяемостью переменных). В языке SQL не используется *термин полиморфизм.*

В языке SQL поддерживаются аналоги предложенного автором оператора TREAT DOWN и операторов проверки типов, например, как показано ниже.

- Аналог в языке SQL оператора TREAT_DOWN_AS_CIRCLE (E) — TREAT(E AS CIRCLE).
- Аналог в языке SQL оператора I S_CIRCLE (E) — TYPE (E) IS OF (CIRCLE).
- Аналогом в языке SQL оператора

R : IS_CIRCLE (E)

является следующий оператор.

```
SELECT TREAT ( E AS CIRCLE ) AS E, F, G, ..., H
FROM R
WHERE TYPE ( E ) IS OF ( CIRCLE )
```

Здесь в состав атрибутов F, G, . . ., н входят все атрибуты R, кроме E.

А поскольку в языке SQL не поддерживаются ограничения типа, то в нем, безусловно, отсутствует также поддержка уточнения или обобщения с помощью ограничений. Но следует отметить, что этот факт не означает отсутствие возможности изменения наиболее конкретного типа переменной, например, как показано ниже.

```
DECLARE E ELLIPSE ;

SET E = CX
; SET E =
EX ;
```

Здесь CX и EX— выражения, которые возвращают, соответственно, значения наиболее конкретного типа CIRCLE и ELLIPSE. Поэтому после первого присваивания переменная E (который имеет объявленный тип ELLIPSE) характеризуется наиболее конкретным типом CIRCLE, а после второго присваивания она имеет наиболее конкретный

тип ELLIPSE. Но заслуживает особого внимания то, что эти эффекты (повторяем еще раз) не достигаются благодаря уточнению или обобщению с помощью ограничений.

Наконец, напомним сказанное в главе 5, что любой конкретный структурированный тип не обязательно должен иметь связанный с ним оператор "=", и даже если для него предусмотрен такой оператор, то его семантика полностью определяется пользователем. В действительности, в языке SQL даже не предъявляется такое требование, что наиболее конкретные типы сравниваемых значений должны быть одинаковыми для того, чтобы результат операции сравнения принял значение TRUE! Из ЭТОГО следует, что если в сравнении участвует какой-либо структурированный тип, то даже нет гарантии должной поддержки в языке SQL операций соединения, объединения, пересечения и разности. Кроме того, следует отметить, что эти критические замечания остаются справедливыми независимо от того, применяется ли в этом языке наследование того или иного типа.

Сравнение принципов наследования и делегирования

На данном этапе следует признать, что описание, приведенное выше в данном разделе, может привести к неправильным выводам в одном очень важном отношении. Дело в том, что механизм наследования типов языка SQL (в отличие от предложенной автором модели наследования) почти наверняка не предназначен для поддержки той идеи, что создание подтипов происходит в результате применения ограничений к супертипам. Еще раз рассмотрим определение эллипсов и окружностей, как показано ниже.

```
CREATE TYPE ELLIPSE UNDER PLANE_FIGURE
   AS ( A LENGTH, B LENGTH, CTR POINT )  ... ;

CREATE TYPE CIRCLE UNDER
   ELLIPSE AS ( R LENGTH ) ...
   ;
```

В этих определениях тип CIRCLE имеет атрибуты A, в, CTR (унаследованные от типа ELLIPSE) и R (определенный только для типа CIRCLE). А если является справедливым утверждение о том, что заданные атрибуты образуют физическое представление, то каждая конкретная окружность должна быть физически представлена в виде коллекции из четырех значений, притом три из них в обычных условиях будут иметь одинаковое значение! По этой причине, вполне вероятно, что в определении типа CIRCLE фактически не будет вообще указываться какая-либо собственная опция представления *<representation>;* вместо этого данный тип будет просто наследовать представление, которое определено для типа ELLIPSE. С другой стороны, если представление для типа CIRCLE не имеет компонента R ("radius" — радиус), то для радиуса не будут автоматически предусмотрены такие методы, как наблюдатель и модификатор. А из этого следует еще одно замечание — если представление окружности включает компонент R и мы его "модифицируем", то в конечном итоге получим "некруглую окружность", т.е. "окружность", в которой, безусловно, не являются одинаковыми все значения A, в и R.

Итак, по этой или по другой причине, можно утверждать, что "эллипсы и окружности" не являются хорошим примером для использования в качестве основы при описании функциональных средств наследования типов языка SQL. Безусловно, справедливо и то, что язык SQL недостаточно хорошо проявляет себя в данном примере. Поэтому перейдем к анализу другого примера, как показано ниже.

```
CREATE TYPE CIRCLE
   AS ( R LENGTH, CTR POINT
        ) INSTANTIABLE NOT
        FINAL ;

CREATE TYPE COLORED CIRCLE UNDER
   CIRCLE AS ( COL COLOR )
        INSTANTIABLE
        NOT FINAL ;
```

Этот пример является именно тем, который мы признали неудачным раньше, в разделе 20.9, где было сказано, что "цветные окружности" нельзя назвать окружностями в том же смысле, например, в каком окружности являются эллипсами. Но если речь идет о наследовании и, возможно, даже дополнении представлений, то этот пример приобретает немного больше смысла. Безусловно, вполне резонно рассматривать цветные окружности как объекты, представленные с помощью таких атрибутов, как радиус, центр и цвет. Следует также отметить, что если тип COLORED_CIRCLE определен как подтип типа CIRCLE с помощью ключевого слова "UNDER", то вполне допустимо также рассматривать операторы, применимые к окружностям в целом (например, оператор определения радиуса), как применимые и к цветным окружностям в частности (цветные окружности могут быть подставлены вместо обычных окружностей). Но остается одна идея, лишенная смысла, — рассматривать цветные окружности как "ограниченную форму" окружностей в целом или, что равносильно, формировать цветные окружности из обычных по принципу уточнения с помощью ограничений. Иными словами, создается впечатление, что механизм наследования SQL разработан вообще не для использования в процессе наследования в том смысле, в каком этот термин был определен выше в данной главе, а скорее для использования в том процессе, который некоторые авторы называют *делегированием.* Делегирование означает, что ответственность за реализацию некоторых операций, связанных с данным типом, "поручается" (или делегируется) типу некоторого компонента представления данного типа (например, операция "получение радиуса" для цветной окружности реализуется с помощью вызова операции "получения радиуса" в соответствующем компоненте окружности). Поэтому было бы более понятно, если бы этот механизм SQL именовался в первую очередь механизмом делегирования, а не выдвигались претензии на то, что он имеет какое-то отношение к наследованию и подтипам.

20.11. РЕЗЮМЕ

В настоящей главе кратко описаны основные понятия **модели наследования типа.** Если тип **в** является подтипом типа А (равным образом, если тип А — супертип типа в), то каждое значение типа В является также значением типа А и поэтому ограничения и операции, применяемые к значениям типа А, применимы также к значениям типа в (но, кроме того, имеются ограничения и операции, которые применяются к значениям типа в, но не могут применяться к значениям, которые относятся только к типу А). Теперь фактически это определение можно уточнить и указать, что если **в** — строгий подтип типа А, то справедливы приведенные ниже утверждения.

- Множество значений в является строгим подмножеством множества значенийА.
- На тип В распространяется строгое надмножество тех ограничений, которые отно сятся к типу А.

Тип в имеет строгое надмножество таких операций только чтения, которые относятся к типу А.

Тип в имеет строгое подмножество таких операций обновления, которые относятся к типу А (но может также иметь собственные дополнительные операции обновления, не относящиеся к А).

Затем в этой главе было описано различие между **одинарным** и **множественным** наследованием (но приведены сведения только об одинарном наследовании), а также различие между наследованием **скаляров, кортежей** и **отношений** (но дано описание только наследования скаляров)[16], кроме того, представлено понятие **иерархии типов. К** тому же определены термины **строгий** подтип и супертип, **непосредственный** подтип и супертип, а также **корневой** тип и **листовой** тип и сформулировано **предположение о непересечении,** согласно которому типы Т1 и Т2 являются непересекающимися, если ни один из них не является подтипом другого. Из этого предположения следует, что каждое значение имеет уникальный наиболее конкретный тип (не обязательно листовой тип).

После этого в данной главе представлены понятия **полиморфизма** (включения) и **заменяемости** (значений) и указано, что оба эти понятия являются логическим следствием из основного определения понятия наследования. Кроме того, было показано различие между полиморфизмом **включения,** который связан с понятием наследования, и полиморфизмом **перегрузки,** который не связан с этим понятием. В ней также показано, что полиморфизм включения позволяет обеспечить **повторное использование кода** благодаря осуществлению **связывания на этапе прогона.**

Затем мы перешли к описанию влияния наследования на операции **присваивания.** При этом была подчеркнута основная мысль, что не происходит преобразований типов (значения сохраняют свой наиболее конкретный тип после присваивания переменным с менее конкретным объявленным типом), и поэтому переменная объявленного типа **т** может иметь значение, наиболее конкретным типом которого является любой подтип типа т. (Аналогичным образом, если оператор Op определен как имеющий результат объявленного типа т, то фактическим результатом вызова оператора Op может быть значение, наиболее конкретным типом которого является любой подтип типа т.) Поэтому была предложена модель переменной V (или в более общем смысле, произвольного выражения) как упорядоченной тройки в форме <DT, MST, v>, где DT — объявленный тип, MST — текущий наиболее конкретный тип и v — текущее значение. Затем был представлен оператор **TREAT DOWN,** ПОЗВОЛЯЮЩИЙ ВЫПОЛНЯТЬ операции таким образом, чтобы можно было предотвратить возникновение на этапе компиляции ошибок из-за несоответствия типов при обработке таких выражений, наиболее конкретным типом которых на этапе прогона становится некоторый строгий подтип их объявленного типа. (Ошибки из-за несоответствия типов на этапе прогона все еще могут возникать, но только в контексте оператора TREAT DOWN.)

После этого были более подробно описаны **селекторы** и показано, что вызов селектора для типа **т** иногда приводит к получению результата, имеющего некоторый строгий

---

[16] Но автор может, по крайней мере, утверждать, что описанные в данной главе идеи, касающиеся одинарного наследования и скалярного наследования скаляров, распространяются исключительно успешно (к тому же на удивление легко) на множественное наследование, а также на наследование кортежей и отношений, как показано в [3.3].

подтип типа т (по меньшей мере, в предложенной автором модели, хотя обычно не во всех современных коммерческих продуктах), что представляет собой **уточнение с помощью ограничения.** Затем были более подробно описаны **псевдопеременные THE_;** поскольку по существу они представляют собой просто сокращенное обозначение, то при выполнении операций присваивания псевдопеременным THE_ может происходить и **уточнение,** и **обобщение** с помощью ограничений.

Затем мы перешли к описанию того, какое влияние оказывает применение подтипов и супертипов на операции **проверки на равенство** и на некоторые реляционные операции **(соединение, объединение, пересечение** и **разность). В** этой главе представлен также ряд операторов **проверки типов** (IS_T и т.д.). После этого был рассмотрен вопрос использования операций **только чтения** и **обновления, версий** операторов и **сигнатур** операторов, а также указано на то, что возможность определять различные версии оператора создает предпосылки **изменения семантики** рассматриваемых операторов (но в модели, предложенной автором, подобные изменения запрещены).

Наконец, был исследован вопрос о том, "действительно ли окружности являются эллипсами". Это исследование позволило прийти к заключению, что **принципы наследования распространяются на значения,** а не на переменные. А именно, операции только чтения (которые применяются к значениям) могут быть унаследованы полностью без каких-либо затруднений, а операции обновления (применяемые **к** переменным) могут быть унаследованы только **условно. (В** этом предложенная автором модель расходится с большинством других подходов. Все прочие подходы обычно требуют, чтобы операции обновления наследовались безусловно, но из-за этого они характеризуются наличием многочисленных проблем, когда приходится сталкиваться с "некруглыми окружностями" и тому подобными аномалиями.) По мнению автора, единственным логически допустимым способом определения подтипов является уточнение с помощью ограничения.

Кроме того, в этой главе было кратко описано понятие **делегирования** (delegation), которое связано с наследованием, но логически отличается от него (хотя целью делегирования также является повторное использование кода). К тому же был кратко описан механизм наследования SQL и сделано заключение, что рассматриваемый механизм фактически предназначен для решения задачи делегирования, а не для обеспечения наследования. Напомним, что дополнительные сведения о наследовании в стиле SQL будут приведены в главе 26.

УПРАЖНЕНИЯ

20.1. Дайте определение следующим терминам:

делегирование
заменяемость
корневой тип
листовой тип
многократное использование кода
наследование
непосредственный подтип
обобщение с помощью ограничения
объединенный тип
полиморфизм
связывание на этапе прогона
сигнатура
строгий подтип
уточнение с помощью ограничения

20.2. Опишите назначение оператора TREAT DOWN.

20.3. В чем состоят различия между перечисленными ниже понятиями:

- фактический параметр и формальный параметр;
- объявленный тип и текущий наиболее конкретный тип;
- полиморфизм включения и полиморфизм перегрузки;
- сигнатура версии и сигнатура спецификации;
- сигнатура версии и сигнатура вызова;
- операция только чтения и операция обновления;
- значение и переменная.

(Применительно к последним двум терминам см. такжеупр. 5.2.)

20.4. Используя иерархию типов, изображенную на рис. 20.1, рассмотрите значение e, которое имеет тип ELLIPSE. Каковым является *наиболее* конкретный тип e (ELLIPSE или CIRCLE) и каковым *наименее* конкретный тип e?

20.5. Любая заданная иерархия типов включает несколько субиерархий, каждая из которых может рассматриваться как полноценная и самостоятельная иерархия типов. Например, самостоятельной может считаться иерархия, полученная путем удаления типов PLANE_FIGURE, ELLIPSE и CIRCLE (и только этих типов) из иерархии, показанной на рис. 20.1, а также иерархия, полученная путем удаления типов CIRCLE, SQUARE и RECTANGLE (и только этих типов). С другой стороны, иерархию, полученную путем удаления типа ELLIPSE (и только этого типа), нельзя назвать иерархией типов в полном смысле этого слова (по меньшей мере, такой нельзя назвать иерархию, полученную указанным способом из иерархии, показанной на рис. 20.1), поскольку тип CIRCLE во вновь созданной иерархии "теряет часть наследуемых им свойств". Какое общее количество различных полноценных и самостоятельных иерархий типа представлено на рис. 20.1?

20.6. С использованием синтаксиса, кратко описанного в данной главе, сформулируйте определения типов RECTANGLE (Прямоугольник) и SQUARE (Квадрат). Для упрощения примите предположение, что центры всех прямоугольников находятся в начале координат, но не предполагайте, что их стороны обязательно должны быть либо вертикальными, либо горизонтальными.

20.7. По результатам, полученным при выполнении упр. 20.6, определите операцию поворота указанного прямоугольника на 90° вокруг его центра. Кроме того, определите вариант реализации этой операции для квадратов.

20.8. Повторим пример из раздела 20.6 в следующей формулировке: "Допустим, что переменная отношения R имеет атрибут A объявленного типа ELLIPSE. Необходимо составить запрос, чтобы выполнить выборку из R таких кортежей, где значением A фактически является окружность, и радиус этой окружности больше двух". Для данного запроса в разделе 20.6 приведено определение:

```
R : IS_CIRCLE ( A ) WHERE THE_R ( A ) > LENGTH (2.0)
```

а) Почему нельзя просто поместить выражение проверки типа в предложение WHERE, например, как показано ниже?

```
R WHERE IS_CIRCLE ( A ) AND THE_R ( A ) > LENGTH
(2.0)
```

б) Приведем еще один пример возможной формулировки запроса.

```
R WHERE CASE
       WHEN IS CIRCLE ( A ) THEN
            THE_R ( TREAT_DOWN_AS_CIRCLE ( A ) )
                                  > LENGTH (2.0)
        WHEN NOT ( IS CIRCLE ( A ) ) THEN
      FALSE END CASE
```

Верна ли эта формулировка? Если нет, то почему?

20.9. В [3.3] предложено ввести поддержку реляционных выражений следующего вида.

```
R TREAT_DOWN_AS_T ( A )
```

Здесь R— реляционное выражение, А— атрибут отношения (скажем, r), обозначаемый этим выражением, и т — некоторый тип. Значение всего выражения определяется как отношение с описанными ниже свойствами.

а) Оно имеет такой же заголовок, как и отношение r, но объявленным типом атрибута А в этом заголовке является т.

б) Тело этого отношения состоит из тех же кортежей, что в отношении r, за исключением того, что значение атрибута А в каждом из таких кортежей рассматривается как относящееся к типу т.

Напомним, что применяемый здесь оператор TREAT DOWN является просто сокращенным обозначением. Но для чего именно?

20.10. Выражение вида R:IS_T(A) также является сокращенным обозначением. Для чего именно?

20.11. Язык SQL поддерживает функции-конструкторы вместо селекторов. В чем состоят различия между ними?

20.12. Почему, по вашему мнению, язык **SQL** не поддерживает ограничения типа? А что можно сказать по поводу уточнения с помощью ограничения?

СПИСОК ЛИТЕРАТУРЫ

В начале этого раздела отметим без дополнительных пояснений тот небольшой перечень значительных изменений, которые требуется внести в предложенную автором модель наследования, рассматриваемую в основной части данной главы, для обеспечения поддержки множественного наследования. Во-первых, необходимо сделать менее строгим допущение о непересечении, определив, чтобы непересекающимися были только корневые типы. Во-вторых, определение "наиболее конкретного типа" заменяется следующим требованием — каждое множество типов T1, T2, . . ., Tп (п > 0) должно иметь общий подтип T', такой что любое конкретное значение относится к каждому из типов T1, T2, . . ., Tп тогда и только тогда, когда оно относится к типу T'. Подробное описание этих дополнений, а также всех расширений, которые требуются для поддержки наследования кортежей и отношений, приведены в [3.3].

20.1. Alphora. Dataphor™ Product Documentation // Документация программного продукта Dataphor, предоставляемая компанией Alphora (см. также http://www.alphora. com).

Dataphor— это коммерческий программный продукт, который поддерживает значительное подмножество модели наследования, описанной в настоящей главе (а также значительную часть прочих предложений, изложенных в Третьем Манифесте [3.3]).

20.2. Atkinson M. et al. The Object-Oriented Database System Manifesto // Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases. — Kyoto, Japan, 1989. New York, N.Y.: Elsevier Science, 1990.

Относительно разногласий по вопросу, что должна собой представлять настоящая модель наследования (как уже отмечалось в разделе 20.1), авторы этой статьи высказываются так: "Имеется по крайней мере четыре типа наследования: наследование *заменяемости,* наследование *включения,* наследование *ограничений* и наследование *уточнения*... Различные уровни этих четырех типов наследования предусмотрены в существующих системах и прототипах, и мы не предписываем конкретный стиль наследования".

Далее приводится еще несколько цитат, которые дополняют ту же основную мысль.

- В [20.5] Кливленд (Cleaveland) пишет: "[Наследование может] основываться на [множестве] различных критериев, и не существует какого-либо общеприня того стандартного определения". Он также предлагает восемь возможных ин терпретаций. (Мейер в [20.11] перечисляет двенадцать.)
- Баклавски (Baclawski) и Индурхия (Indurkhya) в [20.3] пишут: "Язык програм мирования [просто] предоставляет ряд механизмов [наследования]. Хотя эти механизмы, безусловно, ограничивают возможности конкретного языка и раз новидности реализуемых моделей наследования... сами они не позволяют оце нить те или иные разновидности наследования. Классы, уточнения, обобщения и наследование — это только понятия, и... они не имеют всеобщего объективного смысла... Отсюда следует — то, как наследование будет включено в конкретную систему, зависит от проектировщиков [данной] системы; именно это составля ет стратегию разработки, которая должна быть реализована с помощью дос тупных механизмов". Другими словами, нет никакой модели!

Но автор не согласен с приведенными выше выводами, как следует изданной главы.

Примечание. Эта работа упомянута еще раз в главе 25 как [25.1], где можно найти дополнительные комментарии к ней.

20.3. Baclawski K., Indurkhya B. Technical Correspondence // CACM. — September 1994. — 37, № 9.

20.4. Cardelli L., Wegner P. On Understanding Types, Data Abstraction, and Polymorphism // ACM Сотр. Surv. - December 1985. - 17, № 4.

20.5. Cleaveland G.J. An Introduction to Data Types. — Reading, Mass.: Addison-Wesley, 1986.

20.6. Date C. J. Is a Circle an Ellipse? //http: / /www. dbdebunk. com, July 2001.

Создается впечатление, что почти все специалисты в области баз данных дают отрицательный ответ на вопрос о том, является ли окружность эллипсом, поставленный в заголовке этой статьи. В статье приведены мнения некоторых авторитетных специалистов по этому поводу и предпринята попытка понять, на чем основаны их доводы.

Примечание. После ее первой публикации эта статья стала предметом многочисленных комментариев и критических замечаний, опубликованных в интерактивных средствах массовой информации; большинство из них также можно найти по адресу http://www.dbdebunk.com.

20.7. Date C. J. What Does Substitutability Really Mean? // http: / /www. dbdebunk. com, July 2002.

Тщательный анализ и критика работы [20.9].

20.8. Fun Y.-C et al. Implementation of SQL3 Structured Types with Inheritance and Value Substitutability // Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland. — September 1999.

Приведем цитату из резюме к этой работе: "В настоящей статье описан подход, принятый в DB2... Во-первых, значения структурированных типов представлены в такой форме, что служат описанием для самих себя, а манипуляции с ними осуществляются только с помощью методов наподобие наблюдателей/модификаторов, вырабатываемых системой, что сводит к минимуму отрицательное влияние на работу диспетчера памяти низкого уровня. Во-вторых, семантика модификаторов, основанных на значениях, реализована эффективно с помощью алгоритма, позволяющего избежать копирования данных на этапе компиляции. В-третьих, значения структурированных типов сохраняются в объектах или вне объектов динамически".

Примечание. Выражение "семантика модификаторов, основанных на значениях", указывает на тот факт, что модификаторы SQL фактически осуществляют операции только чтения (необходимый эффект "модификации" достигается путем вызова модификатора с указанием желаемого целевого объекта, скажем, т, а затем присваивания результата этого вызова снова объекту т).

20.9. Liskov B., Wing J. A Behavioral Notion of Subtyping // ACM TOPLAS (Transactions on Programming Languages and Systems). — November 1994. — 16, № 6.

В основной части литературы *заменяемость* упоминается под названием *принципа замены Дисков* (Liskov Substitution Principle — LSP). Данная статья рассматривается как первая публикация с упоминанием данного принципа.

20.10. Mattos N., DeMichiel L.G. Recent Design Trade-Offs in SQL3 // ACM SIGMOD Record. — December 1994. — 23, № 4.

В статье обосновывается решение разработчиков языка SQL не поддерживать ограничения типов (это обоснование базируется на доводах, которые были приведены ранее Здоником (Zdonik) и Мейером (Maier) в [20.14]). Однако автор не согласен с таким обоснованием. Фундаментальная проблема заключается в том, что в нем не проводится должным образом различие между значениями и переменными.

20.11. Meyer B. The Many Faces of Inheritance: A Taxonomy of Taxonomy // IEEE Compu ter. — May 1996. — 29, № 5.

20.12. Rumbauch J. A Matter of Intent: How to Define Subclasses //Journal of Object-Oriented Programming. — September 1996.

Как указывалось в разделе 20.9, автор придерживается мнения, что уточнение с помощью ограничения — это единственный логически допустимый способ определения подтипов. Поэтому интересно отметить, что сообщество разработчиков объектно-ориентированных систем (или, по меньшей мере, определенная его часть) занимает прямо противоположную позицию. Процитируем автора настоящей публикации: "Класс SQUARE — это подкласс класса RECTANGLE? ... Растяжение прямоугольника по оси X вполне допустимо. Но если то же самое сделать с квадратом, он перестанет быть квадратом. Концептуально это не обязательно плохо. Когда вы растягиваете квадрат, *получается* прямоугольник... Но... большинство объектно-ориентированных языков не допускают, чтобы объекты меняли свой класс... Поэтому предлагается следующий принцип проектирования для систем классификации: *Подкласс не должен быть определен путем ограничения суперкласса"* (такое же выделение курсивом сделано и в оригинале).

Примечание. Как указано в главе 25, в сообществе разработчиков объектно-ориентированных систем часто используется термин *класс* для обозначения понятия, которое мы называем *типом.*

Нас поражает, что автор принимает указанную позицию просто потому, что объектно-ориентированные языки "не допускают, чтобы объекты меняли свой класс". Мы бы скорее побеспокоились о получении, прежде всего, правильной *модели,* а не о соответствии конкретной реализации. (Так или иначе, но мы считаем, что нам известен способ эффективной реализации уточнения с помощью ограничения, и мы изложили некоторые свои идеи в этой области в [3.3].)

20.13. Taivalsaari A. On the Notion of Inheritance //ACM Cотр. Surv. — September 1996. - 28, № 3.

20.14. Zdonik S.B., Maier D. Fundamentals of Object-Oriented Databases// [25.52].

ГЛАВА 21

Распределенные базы данных

21.1. ВВЕДЕНИЕ

В конце главы 2 уже затрагивалась тема распределенных баз данных. При этом было указано: "Полная поддержка распределенных баз данных означает, что отдельное приложение может «прозрачно» обрабатывать данные, распределенные между множеством различных баз данных, управление которыми осуществляют разные СУБД, работающие на соединенных коммуникационными сетями компьютерах разных типов с различными операционными системами. Здесь понятие «прозрачно» означает, что приложение выполняет обработку данных с логической точки зрения так, как будто управление данными полностью осуществляется одной СУБД, работающей на единственном компьютере". В этой главе мы поговорим о распределенных базах данных подробнее, а именно — более точно определим, что такое распределенная база данных, почему такие базы данных играют все более важную роль (достаточно лишь подумать о современных масштабах развития **World Wide Web** — см. главу 27) и какие технические проблемы существуют в области распределенных баз данных.

Кроме того, в главе 2 обсуждались системы "клиент/сервер", которые можно рассматривать как простой частный случай распределенных систем вообще. Системы "клиент/ сервер" будут рассмотрены в разделе 21.5.

Общий план данной главы приведен в конце следующего раздела.

21.2. ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

Начнем с рабочих определений, приведенных ниже, которые на этом этапе неизбежно будут не совсем точны.

- Система распределенных баз данных состоит из набора **узлов** (site), связанных коммуникационной сетью, в которой:
  а) каждый узел — это полноценная СУБД сама по себе, но
  б) узлы взаимодействуют между собой таким образом, что пользователь любого из них может получить доступ к любым данным в сети так, как будто они находят ся на его собственном узле.

Из этого определения следует, что так называемая *распределенная база данных* в действительности представляет собой *виртуальную* базу данных, компоненты которой физически хранятся в нескольких различных *реальных* базах данных на нескольких различных узлах (в сущности, являясь логическим объединением этих реальных баз данных). Пример подобной структуры показан на рис. 21.1.

Еще раз отметим, что **каждый узел сам по себе является системой баз данных.** Иначе говоря, на каждом узле есть собственные локальные *реальные* базы данных, собственные локальные пользователи, собственные локальные СУБД и программное обеспечение управления транзакциями (включая собственное программное обеспечение блокировки, ведения журналов, восстановления и т.д.) и собственный локальный диспетчер передачи данных. В частности, любой пользователь может выполнять операции над данными на своем локальном узле точно так же, как если бы этот узел вовсе не входил в распределенную систему (по крайней мере, так должно быть). Всю распределенную систему баз данных можно рассматривать как некоторое **партнерство** между отдельными локальными СУБД на отдельных локальных узлах. Новый программный компонент на каждом узле — логическое расширение локальной СУБД — предоставляет необходимые функциональные возможности для организации подобного партнерства. Именно этот компонент вместе с существующими СУБД составляет то, что обычно называется **распределенной системой управления базами данных** (РСУБД).

Чаще всего предполагается, что узлы разделены физически (а возможно, и территориально, как показано на рис. 21.1), хотя в действительности достаточно того, чтобы они были разделены *логически.* Два *узла* могут даже сосуществовать на одном и том же физическом компьютере (в особенности на начальном этапе тестирования). Главная цель создания распределенных систем со временем изменялась. В ранних исследованиях в основном предполагалась территориальная распределенность, но в большинстве первых коммерческих реализаций предполагалось *локальное* распределение, когда несколько *узлов* размещалось в одном здании и соединялось с помощью локальной сети (ЛВС). Однако позже стремительное распространение глобальных сетей (ГВС) снова пробудило интерес к использованию территориального распределения. В любом случае это не имеет большого значения с точки зрения системы баз данных — решать в основном требуется одни и те же технические (связанные с базами данных) проблемы. Поэтому в настоящей главе мы можем обоснованно рассматривать представленную на рис 21.1 систему в качестве типичного представителя распределенных систем.

Примечание. Для упрощения дальнейшего изложения будем предполагать, что система *однородна,* если нет иного соответствующего явного указания. Система однородна в том смысле, что на каждом узле работает некоторая копия одной и той же СУБД. Будем

называть это предположением о **строгой однородности.** Возможности и особенности неоднородных систем будут проанализированы в разделе 21.6.

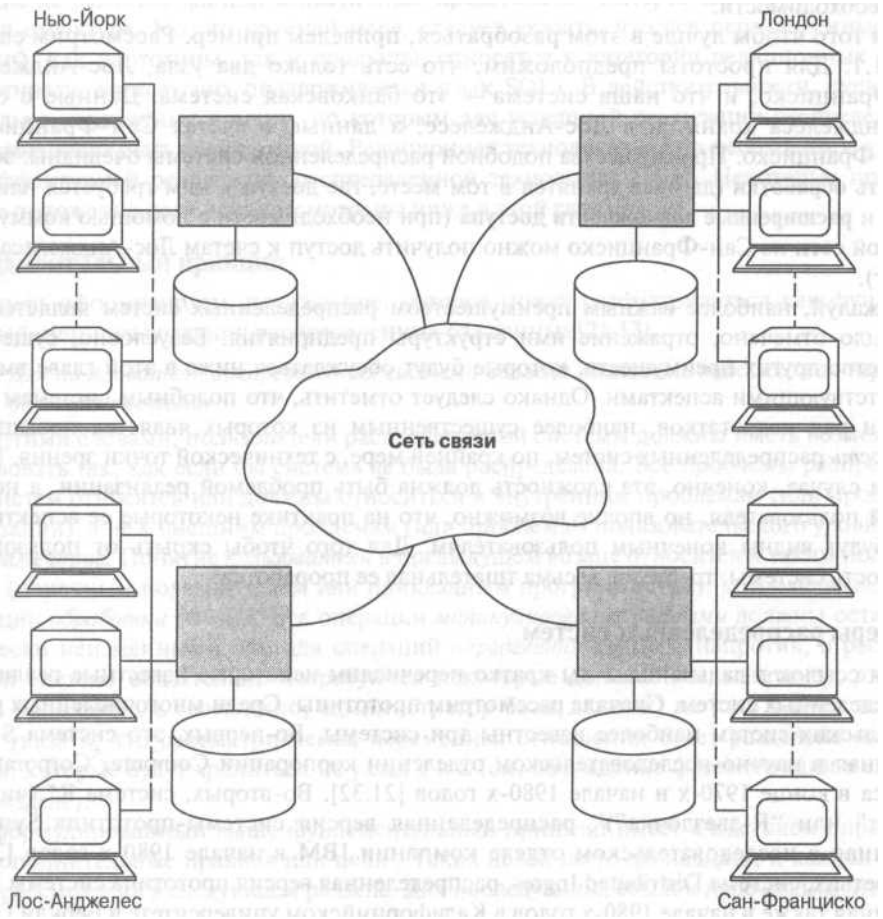


Рис. 21.1. Пример типичной системы распределенной базы данных

Преимущества

Зачем нужны распределенные базы данных? Основная причина заключается в том, что сами предприятия обычно уже распределены, по крайней мере, логически, т.е. разбиты на подразделения, отделы, рабочие группы и т.д. Очень часто они распределены и физически, т.е. разделены на отдельно расположенные заводы, фабрики, лаборатории и т.д. Из этого следует, что данные также обычно распределены, поскольку каждая организационная единица на предприятии создает и обрабатывает собственные данные, относящиеся к ее деятельности. Таким образом, информация предприятия разбивается на отдельные автономные части, которые иногда называют *островами информации.* А распределенная система обеспечивает *мосты* для их соединения в единое целое. Иначе говоря, распределенная система позволяет структуре базы данных отображать структуру

предприятия — локальные данные могут храниться локально, в соответствии с логической принадлежностью, тогда как к удаленным данным доступ может осуществляться по мере необходимости.

Для того чтобы лучше в этом разобраться, приведем пример. Рассмотрим еще раз рис. 21.1. Для простоты предположим, что есть только два узла, Лос-Анджелес и Сан-Франциско, и что наша система — это банковская система. Данные о счетах Лос-Анджелеса хранятся в Лос-Анджелесе, а данные о счетах Сан-Франциско — в Сан-Франциско. Преимущества подобной распределенной системы очевидны: **эффективность обработки** (данные хранятся в том месте, где доступ к ним требуется наиболее часто) и **расширенные возможности доступа** (при необходимости с помощью коммуникационной сети из Сан-Франциско можно получить доступ к счетам Лос-Анджелеса и наоборот).

Пожалуй, наиболее важным преимуществом распределенных систем является, как уже было отмечено, отражение ими структуры предприятия. Безусловно, существует множество других преимуществ, которые будут обсуждаться ниже в этой главе вместе с соответствующими аспектами. Однако следует отметить, что подобным системам свойствен и ряд недостатков, наиболее существенным из которых является повышенная *сложность* распределенных систем, по крайней мере, с технической точки зрения. В идеальном случае, конечно, эта сложность должна быть проблемой реализации, а не проблемой пользователя, но вполне возможно, что на практике некоторые ее аспекты все-таки будут видны конечным пользователям. Для того чтобы скрыть от пользователя сложность системы, требуется весьма тщательная ее проработка.

Примеры распределенных систем

Для ссылок в дальнейшем мы кратко перечислим некоторые известные реализации распределенных систем. Сначала рассмотрим прототипы. Среди многочисленных исследовательских систем наиболее известны три системы. Во-первых, это система SDD-1, созданная в научно-исследовательском отделении корпорации Computer Corporation of America в конце 1970-х и начале 1980-х годов [21.32]. Во-вторых, система R* (читается "R-star" или "R-звездочка")[1], распределенная версия системы-прототипа System R, созданная в исследовательском отделе компании IBM в начале 1980-х годов [21.37]. И в-третьих, система Distributed Ingres, распределенная версия прототипа системы Ingres, созданная также в начале 1980-х годов в Калифорнийском университете в Беркли [21.34].

Что касается коммерческих реализаций, то большинство современных продуктов SQL включает определенную поддержку распределенных баз данных, но, безусловно, с разными уровнями функциональных возможностей. Наиболее известные среди них следующие: Ingres/Star (распределенный компонент базы данных СУБД Ingres), версия для распределенных баз данных СУБД Oracle и средства распределения данных для СУБД DB2.

-> *Примечание.* Разработчики программных продуктов часто выпускают новые версии своих продуктов под другими названиями, поэтому автор не может гарантировать, что перечисленные выше названия (а в некоторых случаях даже продукты) в настоящее время все еще продолжают использоваться. Кроме того, этот список продуктов и прототипов

---

[1] В данном обозначении звездочка — это так называемый "оператор Клина" (Kleene), а конструкция "R*" означает "от нуля или больше экземпляров СУБД [System] R":

ни в коем случае не следует считать исчерпывающим; автор просто хотел упомянуть системы, которые по тем или иным причинам оказывали или оказывают определенное влияние на развитие данной области либо представляют интерес благодаря своей внутренней структуре. Но, по крайней мере, следует указать, что все перечисленные здесь системы, как прототипы, так и продукты, относятся к категории реляционных систем (в частности, во всех них поддерживается язык SQL). В действительности, существует несколько конкретных причин, по которым для успешной реализации распределенная система *должна* быть реляционной. Реляционная технология — это необходимое условие для эффективной реализации распределенной технологии [15.6]. Некоторые причины такого положения дел будут рассмотрены ниже в этой главе.

Фундаментальный принцип

Теперь сформулируем утверждение, которое может рассматриваться как фундаментальный принцип создания распределенных баз данных [21.13].

- *Для пользователя распределенная система должна выглядеть так же, как нераспределенная система.*

Другими словами, пользователи распределенной системы должны иметь возможность действовать так, как если бы система *не* была распределена. Все проблемы распределенных систем относятся или должны относиться к внутренним проблемам (или проблемам реализации), а не к внешним проблемам (или проблемам пользовательского уровня).

Примечание. Понятие *пользователи* в предыдущем абзаце относится к таким пользователям (конечным пользователям или прикладным программистам), которые выполняют операции *обработки данных.* Все операции *манипулирования данными* должны оставаться логически неизменными. Но для операций *определения* данных, напротив, в распределенной системе обязательно потребуются некоторые дополнения, например, для того чтобы пользователь (возможно, администратор базы данных) на узле X имел возможность указать, что рассматриваемая переменная отношения будет разделена на *фрагменты,* которые будут храниться на узлах Y и z (см. обсуждение фрагментации в следующем разделе).

Сформулированный выше фундаментальный принцип имеет следствием определенные дополнительные правила или цели[2]. Таких целей всего двенадцать, и каждая из них будет рассмотрена в следующем разделе. Для последующих ссылок перечислим эти цели.

1. Локальная независимость.
2. Отсутствие зависимости от центрального узла.
3. Непрерывное функционирование.
4. Независимость от расположения.
5. Независимость от фрагментации.
6. Независимость от репликации.
7. Обработка распределенных запросов.
8. Управление распределенными транзакциями.

---

[2] "Правила" — это *термин,* используемый в статье, в которой эти правила впервые были представлены [21.13]. Указанный "фундаментальный принцип" был назван *"Правилом ноль".* Однако на самом деле здесь более уместен термин "цели" — слово "правила" звучит слишком категорично. В этой главе будет использоваться более умеренное название — "цели".

9. Аппаратная независимость.
10. Независимость от операционной системы.
11. Независимость от сети.
12. Независимость от типа СУБД.

Обратите внимание на то, что *не* все эти цели независимы одна от другой. Кроме того, они не исчерпывающие и не все одинаково важны (разные пользователи могут придавать различное значение разным целям в различных средах, и, кроме того, некоторые из этих целей могут быть вообще неприменимы в некоторых ситуациях). Но данные цели полезны как основа для понимания самой распределенной технологии и как общая схема описания функциональных возможностей конкретных распределенных систем. Поэтому мы будем использовать список этих целей как организационный принцип для большей части данной главы. В разделе 21.3 кратко обсуждается каждая из целей. В разделе 21.4 некоторые конкретные вопросы рассматриваются более подробно, а в разделе 21.5, как уже отмечалось, приводится обсуждение систем "клиент/сервер". В разделе 21.6 мы детально обсудим некоторые конкретные проблемы, связанные с достижением независимости от СУБД. Наконец, раздел 21.7 посвящен поддержке языка SQL, а в разделе 21.8 содержится резюме и представлено несколько заключительных замечаний.

Рассмотрим последний дополнительный вопрос в этом разделе. Важно отличать истинные обобщенные системы распределенных *баз данных* от систем, которые предоставляют просто *удаленный доступ к данным* (кстати, это все, что на самом деле предоставляет пользователям система "клиент/сервер"). В системах удаленного доступа к данным конечный пользователь может оперировать данными на удаленном узле или даже данными на нескольких удаленных узлах одновременно, но ему будут видны все "швы". Пользователю, несомненно, в той или иной мере будет известно, что данные расположены не локально, и поэтому он должен действовать с учетом этого. В истинной системе распределенных баз данных, напротив, все швы скрыты. (Большая часть этой главы посвящена выяснению вопроса, что же означает выражение "все швы скрыты".) Далее термин *распределенная система* будет означать именно истинную обобщенную систему распределенной базы данных (в противоположность обычной системе удаленного доступа к данным), если только явно не будет указано иное.

21.3. ДВЕНАДЦАТЬ ОСНОВНЫХ ЦЕЛЕЙ 1.

Локальная независимость

Узлы в распределенной системе должны быть **независимы,** или **автономны.** Локальная независимость означает, что все операции на узле контролируются этим узлом. Никакой узел X не должен зависеть от некоторого узла Y, чтобы успешно функционировать (иначе, если узел Y будет отключен, узел X не сможет функционировать, даже если на самом узле X будет все в порядке; возникновение таких ситуаций, безусловно, нежелательно). Локальная независимость также означает, что локальные данные имеют локальную принадлежность, управление и учет. Все данные *реально* принадлежат одной и той же локальной базе данных, даже если доступ к ней осуществляется с других, удаленных узлов. Следовательно, такие вопросы, как безопасность, целостность, защита и представление локальных данных на физическом устройстве хранения, остаются под контролем и в пределах компетенции локального узла.

В действительности, локальная независимость не вполне достижима — есть множество ситуаций, в которых узел X *должен* отказываться в определенной степени от контроля в пользу узла y. Поэтому цель достижения локальной независимости точнее можно было бы сформулировать так: узлы должны быть независимыми в максимально возможной степени. (См. аннотацию к [21.13], где этот вопрос описан подробнее.)

2. Отсутствие зависимости от центрального узла

Локальная независимость предполагает, что **все узлы в распределенной системе должны рассматриваться как равные.** Поэтому, в частности, не должно быть никаких обращений к *центральному,* или *главному,* узлу для получения некоторой централизованной услуги. Не должно быть, например, централизованной обработки запросов, централизованного управления транзакциями или централизованной службы присваивания имен, поскольку в таких случаях система в целом будет зависимой от центрального узла. Таким образом, вторая цель на самом деле является следствием первой цели — если первая цель достигнута, то вторая цель также *заведомо* достигается. Но достижение цели "Отсутствие зависимости от центрального узла" полезно само по себе, даже если полная локальная независимость узлов не будет достигнута. Поэтому отдельная формулировка данной цели также важна.

Зависимость от центрального узла может оказаться нежелательной по крайней мере по двум следующим причинам. Во-первых, такой центральный узел, скорее всего, станет узким местом системы, и, во-вторых (что еще хуже), система станет *уязвимой* — если работа центрального узла будет нарушена, то вся система выйдет из строя (проблема *единственного источника отказа).*

3. Непрерывное функционирование

В общем случае преимущество распределенных систем состоит в том, что они должны предоставлять более высокую степень *надежности* и *доступности.*

- **Надежность** понимается как высокая степень вероятности того, что система будет работоспособна и будет функционировать в любой заданный момент. Надежность распределенных систем повышается за счет того, что они не опираются на принцип "все или ничего"; распределенные системы могут непрерывно функционировать (по меньшей мере, в сокращенном варианте) даже в случаях отказов части их компонентов, таких как отдельный узел.
- **Доступность** понимается как высокая степень вероятности того, что система окажется исправной и работоспособной и будет непрерывно функционировать в течение определенного времени. Доступность, как и надежность распределенных систем также повышается — частично по тем же причинам, а также благодаря возможности дублирования данных (подробности приводятся в комментарии к цели 6).

Предыдущие рассуждения относятся к случаям **незапланированного отключения** системы, т.е. к аварийным ситуациям, которые могут возникнуть в системе в любой момент. Незапланированные отключения системы, безусловно, нежелательны, но их трудно предупредить! **Планируемые** отключения системы, напротив, *никогда* не должны быть необходимыми, т.е. никогда не должна возникать необходимость отключить систему, чтобы выполнить какую-либо задачу, например, добавить новый узел или заменить на уже существующем узле текущую версию СУБД ее новой реализацией.

4. Независимость от расположения

! Основная идея **независимости от расположения,** или так называемой **прозрачности** расположения, проста. Пользователи не должны знать, где именно данные хранятся физически и должны поступать так (по крайней мере, с логической точки зрения), как если бы все данные хранились на их собственном локальном узле. Благодаря независимости от расположения упрощаются пользовательские программы и терминальные операции. В частности, данные могут быть перенесены с одного узла на другой, и это не должно требовать внесения каких-либо изменений в использующие их программы или действия пользователей. Такая переносимость желательна, поскольку она позволяет перемещать данные в сети в соответствии с изменяющимися требованиями к эффективности работы системы.

Примечание. Нетрудно заметить, что независимость от расположения представляет собой простое расширение обычной концепции физической *независимости от данных* применительно к распределенным системам. Забегая несколько вперед, следует сказать, что на самом деле каждую из целей, в названии которой употреблено слово "независимость", можно рассматривать как расширение обычной концепции физической независимости от данных, в чем мы скоро убедимся. Мы еще возвратимся к вопросу независимости от расположения в разделе 21.4, в котором будет обсуждаться *присваивание имен объектам* (подраздел "Управление каталогом").

5. Независимость от фрагментации

Система поддерживает **независимость данных от фрагментации,** если некоторая переменная отношения может быть разделена на части, или *фрагменты,* при организации ее физического хранения, а различные фрагменты могут храниться на разных узлах. Фрагментация желательна для повышения производительности системы. В этом случае данные могут храниться в том месте, где они чаще всего используются, что позволяет достичь локализации большинства операций и уменьшения сетевого трафика. Например, рассмотрим переменную отношения EMP с данными о служащих, пример данных которой приведен на рис. 21.2. В системе, которая поддерживает независимость от фрагментации, два фрагмента этой переменной отношения можно определить следующим образом (см. нижнюю часть рис. 21.2).

```
FRAGMENT EMP AS
   N_EMP AT SITE 'New York' WHERE DEPT#      =   DEPT#  ('Dl')
                            OR    DEPT# =     DEPT#      ('D3') ,
   L_EMP AT SITE 'London'   WHERE DEPT#      =   DEPT#  ('D2') ;
```

Примечание. Здесь подразумевается, что кортежи **с** данными о служащих переменной отношения EMP отображаются в физической памяти каким-то непосредственным способом, причем D1 и D3 — отделы, расположенные в Нью-Йорке, а D2 — отдел, расположенный в Лондоне. Таким образом, кортежи с данными о служащих из Нью-Йорка хранятся на узле в Нью-Йорке, а кортежи с данными о служащих из Лондона — на узле в Лондоне. Отметим, что внутрисистемные имена фрагментов— N_EMP И L_EMP.

Существует два основных вида фрагментации: *горизонтальная* и *вертикальная;* они соответствуют реляционным операциям сокращения и проекции (на рис. 21.2 показан пример горизонтальной фрагментации). В более общем случае фрагмент можно представить в виде результата произвольного сочетания операций сокращения и проекции. Они являются произвольными, если не учитывать описанные ниже условия.

В случае операции сокращения это должна быть *ортогональная* декомпозиция в смысле, указанном в разделе 13.6 главы 13. В случае операции проекции это должна быть декомпозиция *без потерь* в смысле, указанном в главах 12 и 13.

Благодаря этим двум правилам все фрагменты данной переменной отношения будут *независимыми,* т.е. ни один из фрагментов не может быть представлен как производный от других фрагментов, а также, вообще говоря, не может содержать результат операции сокращения или проекции, который может быть получен на основании данных, хранящихся на других узлах.

Примечание. Если действительно необходимо хранить одну и ту же часть данных в нескольких различных местах, можно воспользоваться системным механизмом *репликации* для достижения такого эффекта; см. следующий подраздел.

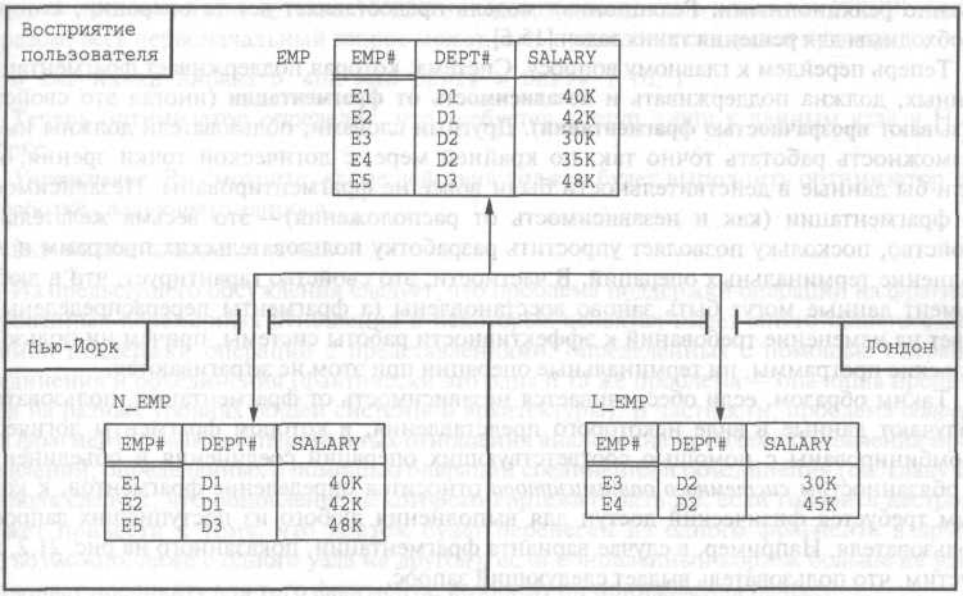


Рис. 21.2. Пример фрагментации

Восстановление исходной переменной отношения из ее фрагментов выполняется с помощью соответствующих операций соединения и объединения (соединения — для вертикальных фрагментов, а объединения — для горизонтальных). Кстати, обратите внимание на то, что благодаря первому правилу в случае объединения операцию исключения дубликатов выполнять не потребуется (т.е. объединение должно быть непересекающимся объединением).

Необходимо сказать еще несколько слов относительно вертикальной фрагментации. Как утверждалось выше, несомненно, что такая фрагментация должна выполняться без потерь. Поэтому разбиение переменной отношения EMP, показанной на рис. 21.2, на фрагменты-проекции, например, вида {EMP#,DEPT#} и {SALARY} было бы недопустимым. Однако в некоторых системах хранимые переменные отношения рассматриваются

как имеющие скрытый, предоставляемый системой атрибут *идентификатор кортежа,* или сокращенно — атрибут TID (Tuple ID). Для каждого хранимого кортежа атрибут **TID,** грубо говоря, является *адресом.* Очевидно, что он является одним из потенциальных ключей для соответствующей переменной отношения. Поэтому, например, если бы переменная отношения EMP содержала такой атрибут, то она *могла* бы быть фрагментирована на проекции {TID,EMP#,DEPT#} и {TID, SALARY}, поскольку такая фрагментация уже, безусловно, выполняется без потерь. Также обратите внимание на то, что если, например, атрибут **TID** является скрытым, то это никак не нарушает *информационный принцип,* поскольку независимость от фрагментации (которая вскоре будет рассматриваться в данной главе) означает, что пользователь не должен знать о существовании фрагментации.

Кстати, отметим, что простота осуществления и фрагментации, и восстановления — это две основные причины, по которым распределенные базы данных должны быть именно реляционными. Реляционная модель предоставляет все те операции, которые необходимы для решения таких задач [15.6].

Теперь перейдем к главному вопросу. Система, которая поддерживает фрагментацию данных, должна поддерживать и **независимость от фрагментации** (иногда это свойство называют **прозрачностью фрагментации).** Другими словами, пользователи должны иметь возможность работать точно так, по крайней мере, с логической точки зрения, как если бы данные в действительности были вовсе не фрагментированы. Независимость от фрагментации (как и независимость от расположения) — это весьма желательное свойство, поскольку позволяет упростить разработку пользовательских программ и выполнение терминальных операций. В частности, это свойство гарантирует, что в любой момент данные могут быть заново восстановлены (а фрагменты перераспределены) в ответ на изменение требований к эффективности работы системы, причем ни пользовательские программы, ни терминальные операции при этом не затрагиваются.

Таким образом, если обеспечивается независимость от фрагментации, пользователи получают данные в виде некоторого представления, в котором фрагменты логически скомбинированы с помощью соответствующих операций соединения и объединения. К обязанностям *системного оптимизатора* относится определение фрагментов, к которым требуется физический доступ для выполнения любого из поступивших запросов пользователя. Например, в случае варианта фрагментации, показанного на рис. 21.2, допустим, что пользователь выдает следующий запрос.

```
EMP WHERE SALARY > 40K AND DEPT# = DEPT# ( ' Dl ' )
```

Из определений фрагментов (которые хранятся, конечно же, в каталоге; оптимизатору должно быть известно, что весь требуемый результат может быть получен только по данным узла в Нью-Йорке, а значит, нет никакой необходимости обращаться к узлу в Лондоне.

Рассмотрим этот пример более подробно. Переменная отношения EMP с точки зрения пользователя может рассматриваться (упрощенно) как некоторое **представление,** сформированное на основе базовых фрагментовNJEMP и L_EMP, следующим образом.

```
VAR EMP "VIEW"                          /* Псевдокод */
    N_EMP UNION L_EMP ;
```

Тогда оптимизатор преобразует исходный запрос пользователя в следующее выражение.

```
{ N_EMP UNION L_EMP ) WHERE SALARY > 4OK
                    AMD DEPT# = DEPT# ('Dl')
```

В процессе дальнейшей оптимизации это **выражение** будет преобразовано в следующее выражение (поскольку операция сокращения распределяется по объединению).

```
( N EMP WHERE SALARY > 4OK AND DEPT# = DEPT# ('Dl') )
 UNION ( L EMP WHERE SALARY > 40K AND DEPT# =
DEPT# ('Dl') )
```

Из определения фрагмента L_EMP в каталоге оптимизатору будет известно, что второй из этих двух операндов объединения UNION эквивалентен следующему выражению.

```
EMP WHERE SALARY > 4OK AND DEPT# =
         DEPT# ('Dl') AND DEPT# = DEPT# ('D2')
```

Это выражение в результате вычисления даст пустое отношение, поскольку соответствующее условие в конструкции WHERE никогда не может стать *истинным* (TRUE). Таким образом, весь первоначальный запрос может быть приведен к следующему виду.

```
N_EMP WHERE SALARY > 40K AND DEPT# = DEPT# ('Dl')
```

Теперь оптимизатор определит, что требуется доступ лишь к данным узла в Нью-Йорке.

Упражнение. Рассмотрите, какие действия должен будет выполнить оптимизатор **при** обработке следующего запроса.

```
EMP WHERE SALARY > 4OK
```

Из предыдущего обсуждения следует, что проблема поддержки операций на фрагментированных переменных отношения в некоторых аспектах имеет много общего с проблемой поддержки операций с представлениями, определенных с помощью операций соединения и объединения (фактически это одна и та же проблема — она лишь проявляется на разных уровнях общей системной архитектуры). В частности, проблема *обновления* фрагментированных переменных отношения аналогична проблеме обновления представлений, определенных с помощью операций соединения и объединения (см. главу 10). Отсюда следует, что обновление некоторого кортежа (опять же, если говорить нестрого) может привести к тому, что кортеж будет перенесен из одного фрагмента в другой (и, возможно, даже с одного узла на другой), если обновленный кортеж больше не удовлетворяет предикату для того фрагмента, которому он принадлежал ранее.

6. Независимость от репликации

Система поддерживает **репликацию данных,** если данная хранимая переменная отношения (или в общем случае данный *фрагмент* данной хранимой переменной отношения) может быть представлена несколькими отдельными копиями, или *репликами,* которые хранятся на нескольких отдельных узлах. Рассмотрим конкретный пример (рис. 21.3). Обратите внимание на то, что внутри системы реплики имеют имена NL_EMP И LN_EMP.

```
REPLICATE N_EMP AS
   LN_EMP AT SITE 'London' ;

REPLICATE L_EMP AS
   NL_EMP AT SITE 'New York' ;
```

Репликация желательна по крайней мере по двум причинам. Во-первых, она способна обеспечить более высокую производительность, поскольку приложения смогут обрабатывать локальные копии вместо того, чтобы устанавливать связь с удаленными узлами. Во-вторых, наличие репликации может также обеспечивать более высокую степень доступности, поскольку любой реплицируемый объект остается доступным для обработки (по крайней мере, для выборки данных), пока хотя бы одна реплика в системе остается доступной. Главным недостатком репликации, безусловно, является то, что если реплицируемый объект обновляется, то и **все его копии** должны быть обновлены (проблема **распространения обновлений). В** разделе 21.4 мы еще скажем несколько слов относительно этой проблемы.

Кстати, отметим, что репликация в распределенных системах представляется специфическим приложением идеи *контролируемой избыточности,* которая обсуждалась в главе 1.

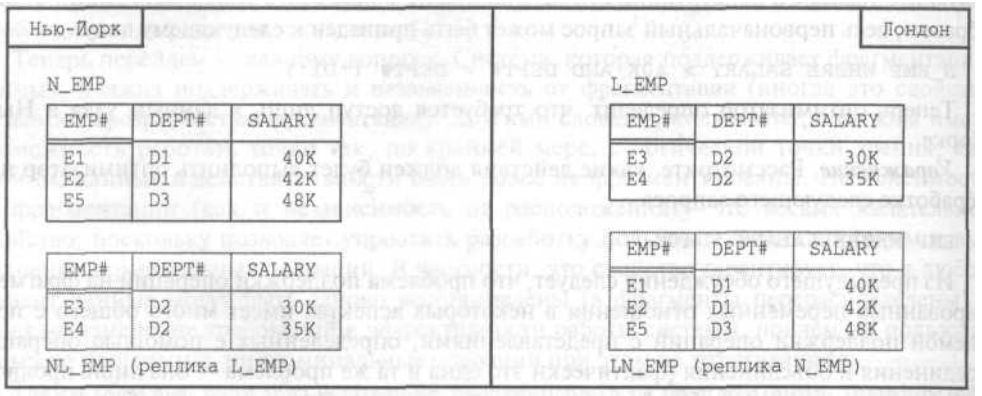


Рис. 21.3. Пример репликации данных

Очевидно, что репликация, как и фрагментация, теоретически должна быть "прозрачной для пользователя". Другими словами, система, которая поддерживает репликацию данных, должна также поддерживать **независимость от репликации** (иногда говорят **"прозрачность репликации").** Для пользователей должна быть создана такая среда, чтобы они, по крайней мере, с логической точки зрения могли считать, что в действительности данные не дублируются. Независимость от репликации (как и независимость от расположения и независимость от фрагментации) является весьма желательной, поскольку она упрощает создание пользовательских программ и выполнение терминальных операций. В частности, независимость от репликации позволяет создавать и уничтожать дубликаты в любой момент в соответствии с изменяющимися требованиями, не затрагивая при этом никакие из пользовательских программ или терминальных операций.

Из требования независимости от репликации следует, что к обязанностям системного оптимизатора также относится определение, какой именно из физических дубликатов будет применен для доступа к данным при выполнении каждого введенного пользователем запроса. Здесь не приведены подробные сведения по этому вопросу.

Завершая подраздел, отметим, что многие коммерческие продукты в настоящее время поддерживают такой вид репликации, который *не* обеспечивает полной независимости от репликации, т.е. репликация будет не полностью "прозрачна для пользователя". Некоторые

дополнительные замечания по этому вопросу будут приведены в подразделе о распространении обновления в разделе 21.4.

7. Обработка распределенных запросов

Существует две особенности, касающиеся темы этого раздела, которые необходимо предварительно прокомментировать.

- Во-первых, рассмотрим запрос: "Получить сведения о лондонских поставщиках деталей красного цвета". Предположим, что пользователь работает на узле в Нью-Йорке, а данные размещены на узле в Лондоне. Предположим также, что имеется л поставщиков, которые удовлетворяют данному запросу. Если система реляци онная, то для выполнения этого запроса, по существу, потребуется пересылка двух сообщений: одно — с запросом из Нью-Йорка в Лондон, а другое — с возвращае мым результатом, т.е. набором из л кортежей, пересылаемых из Лондона в Нью-Йорк. Если, с другой стороны, система — не реляционная и использует операции с таблицами на уровне отдельных записей, выполнение запроса будет, по существу, включать пересылку *2n* сообщений — л сообщений из Нью-Йорка в Лондон, ко торые запрашивают данные *следующего* поставщика, и л сообщений из Лондона в Нью-Йорк, которые возвращают информацию об *очередном* поставщике. Этот пример показывает, что по производительности распределенная реляционная сис тема может на несколько порядков превосходить нереляционную.
- Во-вторых, **оптимизация** в распределенных системах даже более важна, чем в цен трализованных. Суть в том, что для запроса, подобного рассмотренному выше, может потребоваться обращение к нескольким узлам. В такой системе может быть много возможных способов пересылки данных, позволяющих выполнить рассмат риваемый запрос. Поэтому крайне важно, чтобы была найдена эффективная стра тегия. Например, запрос для объединения, скажем, отношения Rx, хранимого на узле X, и отношения Ry, хранимого на узле Y, может быть выполнен посредством пересылки отношения Rx на узел Y или отношения Ry на узел X, или обоих отно шений на какой-либо узел Z и т.п. Наглядная иллюстрация важности оптимиза ции, включающая упомянутый выше запрос ("Получить сведения о лондонских поставщиках деталей красного цвета"), представлена в разделе 21.4. Подводя краткий итог этого примера, можно отметить, что для обработки данного запроса анализируется шесть различных стратегий с учетом определенного набора вероят ных допущений. В результате показано, что время ответа в каждом случае различ но и изменяется в широких пределах от минимального (от одной десятой секунды) до максимального (около *шести часов!).* Таким образом, оптимизация, несомнен но, весьма важна для распределенной системы и, кроме того, эту особенность можно считать еще одной причиной, по которой распределенные системы всегда должны быть реляционными (ответ прост: реляционные системы позволяют оп тимизировать обработку запросов, а нереляционные— нет).

8. Управление распределенными транзакциями

Как известно, существует два главных аспекта управления транзакциями, а именно: управление восстановлением и управление параллельностью обработки. Оба этих аспекта имеют расширенную трактовку в среде распределенных систем. Чтобы разъяснить

особенности этой расширенной трактовки, сначала необходимо ввести новое понятие — *агент.* В распределенной системе отдельная транзакция может потребовать выполнения кода на многих узлах, в частности, это могут быть операции обновления, выполняемые на нескольких узлах. Поэтому говорят, что каждая транзакция содержит несколько **агентов,** где под *агентом* подразумевается процесс, который выполняется для данной транзакции на отдельном узле. Система должна знать, что два агента являются элементами одной и той же транзакции, например, два агента, которые являются частями одной и той же транзакции, безусловно, не должны оказываться в состоянии взаимной блокировки.

Теперь обратимся непосредственно к управлению восстановлением. Чтобы обеспечить неразрывность транзакции (выполнение ее по принципу "все или ничего") в распределенной среде, система должна гарантировать, что все множество относящихся к данной транзакции агентов или зафиксировало свои результаты, или выполнило откат. Такого результата можно достичь с помощью протокола **двухфазной фиксации** транзакции, который уже обсуждался в главе 15, хотя это обсуждение и не было прямо связано с распределенными системами. Подробнее об использовании протокола двухфазной фиксации в распределенных системах речь пойдет в разделе 21.4.

Что касается управления параллельностью, то оно в большинстве распределенных систем базируется на механизме **блокировки,** точно так, как и в нераспределенных системах. В нескольких более новых коммерческих продуктах используется управление параллельной работой на основе одновременной поддержки многих версий [16.1]. Но на практике обычная блокировка, по-видимому, все еще остается тем методом, который лучше всего подходит для большинства систем. Подробнее этот вопрос также будет обсуждаться в разделе 21.4.

9. Аппаратная независимость

По этому вопросу фактически нечего сказать — заголовок раздела говорит сам за себя. Парк вычислительных машин современных организаций обычно включает множество разных компьютеров, допустим, компьютеры производства компаний IBM, Fujitsu, HP, персональные компьютеры, различного рода рабочие станции и т.д. Поэтому действительно существует необходимость интегрировать данные всех этих систем и предоставить пользователю "образ единой системы" [21.9]. Следовательно, желательно иметь возможность эксплуатировать одну и ту же СУБД на различных аппаратных платформах и, более того, добиться, чтобы различные компьютеры участвовали в работе распределенной системы как равноправные партнеры.

10. Независимость от операционной системы

Достижение этой цели частично зависит от достижения предыдущей и также не требует дополнительного обсуждения. Очевидно, что необходимо иметь не только возможность обеспечить функционирование одной и той же СУБД на различных аппаратных платформах, но и обеспечить ее функционирование под управлением различных операционных систем для многих платформ — включая различные операционные системы на одном и том же оборудовании (например, чтобы версия СУБД для операционной системы OS/390, версия для UNIX и версия для Windows могли совместно использоваться в одной и той же распределенной системе).

11. Независимость от сети

Здесь, опять же, нечего добавить. Если система имеет возможность поддерживать много принципиально различных узлов, отличающихся оборудованием и операционными системами, безусловно необходимо, чтобы она также поддерживала ряд типов различных коммуникационных сетей.

12. Независимость от типа СУБД

В этом разделе мы рассмотрим, с чем приходится сталкиваться при отказе от требования строгой однородности системы. Необходимость такого сильного ограничения вызывает сомнения. Действительно, кажется, все, что необходимо, — так это то, чтобы экземпляры СУБД на различных узлах **все вместе поддерживали один и тот же интерфейс,** и совсем не обязательно, чтобы это были копии одной и той же версии СУБД. Например, СУБД Ingres и Oracle обе поддерживают официальный стандарт языка SQL, а значит, можно добиться, чтобы узел с СУБД Ingres и узел с СУБД Oracle обменивались сообщениями между собой данными в рамках распределенной системы. Иными словами, распределенные системы вполне могут быть, по крайней мере, в некоторой степени,*неоднородными.*

Поддержка неоднородности весьма желательна. На практике современное программное обеспечение обычно используется не только на многих различных компьютерах и **в** среде многих различных операционных систем. Оно довольно часто используется и **с** различными СУБД, и было бы очень удобно, если бы различные СУБД можно было каким-то образом включить в распределенную систему. Иными словами, идеальная распределенная система должна **обеспечиватьнезависимость от СУБД.**

Однако эта тема слишком обширна (и важна на практике), поэтому ниже ей посвящен отдельный раздел (см. раздел 21.6).

21.4. ПРОБЛЕМЫ РАСПРЕДЕЛЕННЫХ СИСТЕМ

В этом разделе подробно рассматриваются проблемы, которые упоминались в разделе 21.3. Ключевая проблема распределенных систем состоит в том, что коммуникационные сети, по крайней мере, сети, которые охватывают большую территорию, или глобальные сети, пока остаются *медленными.* Обычная глобальная сеть чаще всего имеет среднюю скорость передачи данных от 5 до 10 тысяч байтов в секунду. Обычный же жесткий диск имеет скорость обмена данными около 5—10 *миллионов* байтов в секунду. (С другой стороны, некоторые локальные сети поддерживают скорость обмена данными того же порядка, что и диски.) Поэтому основная задача распределенных систем (по меньшей мере, в случае глобальной сети, а также до некоторой степени и в случае локальной сети) — **минимизировать использование сетей,** т.е. минимизировать количество и объем передаваемых сообщений. Решение этой задачи, в свою очередь, затрудняется из-за проблем в нескольких дополнительных областях. Ниже приведен список таких областей, хотя нельзя гарантировать, что он является полным.

- Обработка запросов.
- Управление каталогом.
- Распространение обновлений.
- Управление восстановлением.
- Управление параллельностью.

Обработка запросов

Чтобы решить задачу минимизации использования сети, процесс оптимизации запросов должен быть распределенным, как и процесс выполнения запросов. Иначе говоря, в общем случае процесс оптимизации будет включать этап **глобальной оптимизации,** за которым последуют этапы **локальной оптимизации** на каждом участвующем узле. Например, предположим, что запрос Q передан на выполнение на узел X, и предположим, что запрос Q требует соединения отношения Ry на узле Y, содержащего 10 тыс. кортежей, с отношением Rz на узле z, содержащим 10 млн. кортежей. Оптимизатор на узле x должен выбрать глобальную стратегию выполнения запроса Q. В данном случае очевидно, что было бы выгоднее переслать отношение Ry на узел Z, а не отношение Rz на узел Y (или, в зависимости от кардинальности результата соединения, может оказаться, что лучше переслать и отношение Ry, и отношение Rz на узел X). Предположим, что решено переслать данные отношения Ry на узел Z, поэтому реальная стратегия выполнения соединения на узле Z будет определяться локальным оптимизатором этого узла.

Приведенная ниже более подробная иллюстрация рассматриваемого подхода основана на примере, взятом из одной из ранних статей Ротни (Rothnie) и Гудмана (Goodman) [21.31].

Примечание. Фактически приведенные здесь данные теперь немного устарели в связи с дальнейшим развитием аппаратных средств и существенным увеличением размеров современных баз данных, но общая идея все еще остается верной.

- *База данных поставщиков и деталей* (упрощенная)

```
, .,'. S { S#, CITY } 10 000 хранимых кортежей на
    узле A P { P#, COLOR } 100 000 хранимых
    кортежей на узле B SP { S#, P# }        1 000 000
    хранимых кортежей на узле A
```

  Предположим, что каждый хранимый кортеж имеет длину 25 байт (200 бит).

- *Запрос* ("Получить номера лондонских поставщиков деталей красного цвета")

```
( ( S JOIN SP JOIN P ) WHERE CITY = 'London' AND
                          COLOR = COLOR ('Red') ) { S# }
```

- *Оценка кардинальности некоторых промежуточных результатов*

  Количество деталей красного цвета 10
  Количество поставок, выполненных поставщиками из Лондона 100 000

- *Предполагаемые параметры линий передачи данных*

  Скорость передачи данных 50 000 бит/с
  Задержка доступа 0,1 с

Теперь кратко рассмотрим шесть возможных стратегий выполнения этого запроса, и для каждой стратегии i рассчитаем общее время Ti передачи данных по следующей формуле:

```
( общая задержка доступа ) +
( общий объем данных / скорость передачи данных )
```

В этом случае она сводится к такой формуле (время в секундах):

```
( количество сообщений / 10 ) + (количество битов / 50000 )
```

Пересылка записей о деталях на узел А и обработка запроса на узле А.

```
T[1] = 0,1 + ( 100000 * 200 ) / 50000
     = 400 с (приблизительно 6,67 мин)
```

Пересылка записей о поставщиках и поставках на узел В и обработка запроса на узле В.

```
T[2] = 0,2 + ( ( 10000 + 1000000 ) * 200 ) /
     50000 = 4040 с (приблизительно 1,12 ч)
```

Соединение отношений поставщиков и поставок на узле А, сокращение результата для получения данных только о поставщиках из Лондона с последующей проверкой на узле в каждого выбранного поставщика соответствующих деталей для определения того, имеет ли деталь красный цвет. Каждая из таких проверок требует передачи двух сообщений — запроса и ответа. Время передачи этих сообщений будет мало по сравнению с задержкой доступа.

T[3] = 20000 с (приблизительно 5,56 ч) Сокращение данных о деталях на узле в для определения только тех деталей, которые имеют красный цвет, а затем для каждой из выбранных деталей проверка на узле А наличия поставок, связывающих эту деталь с поставщиком из Лондона. Каждая из таких проверок требует передачи двух сообщений. Время передачи этих сообщений также будет мало по сравнению с задержкой доступа.

T[4] = 2 с (приблизительно) Соединение отношений поставщиков и поставок на узле А, сокращение результата для получения данных только о поставщиках из Лондона, получение проекции этих результатов по атрибутам s# и P# и пересылка результата на узел в. Завершение обработки на узле в.

```
T[5] = 0,1 + ( 100000 * 200 ) / 50000
     = 400 с (приблизительно 6,67 мин)
```

Сокращение данных о деталях на узле в для получения сведений только о тех деталях, которые имеют красный цвет, и пересылка результата на узел А. Завершение обработки на узле А.

```
T[6] = 0,1 + ( 10 * 200 ) /
     50000 = 0,1 с
     (приблизительно)
```

В табл. 21.1 подведены итоги по результатам обработки различных вариантов запроса. Ниже приведены комментарии к этим результатам.

- Каждая из шести стратегий представляет возможный подход к проблеме, но откло нения по времени передачи данных огромны. Наибольшее время в *два миллиона* раз больше, чем наименьшее.
- Скорость обмена данными и задержки доступа — важные факторы в выборе стратегии.
- Для плохих стратегий время вычислений и ввода—вывода ничтожно мало по срав нению со временем передачи данных.

  Примечание. Фактически для наилучших стратегий это соотношение может зависеть от дополнительных обстоятельств [21.33]. Такое большое расхождение может также не наблюдаться в случае использования быстрых локальных сетей.,,

Кроме того, некоторые стратегии позволяют выполнять параллельную обработку на

Таблица 21.1. Возможные стратегии распределенной обработки запроса (сводка)

Стратегия	Метод	Время передачи данных
1	Пересылка отношения P на узел A	6,67 мин
2	Пересылка отношений S и SP на узел B	1,12 ч
3	Проверка деталей красного цвета для каждой поставки из Лондона	5,56 ч
4	Проверка поставщика из Лондона для каждой детали красного цвета	2,00 с
5	Пересылка данных о поставках из Лондона на узел B	6,67 мин
6	Пересылка данных о деталях красного цвета на узел A	0,10 с (наилучший вариант)

двух узлах, поэтому время ответа системы пользователю фактически может оказаться меньше, чем в централизованной системе, но следует отметить, что в приведенном выше примере не рассматривался вопрос о том, на каком узле должны быть получены эти результаты.

Управление каталогом

В распределенной системе системный каталог включает не только обычные для каталога данные, касающиеся базовых переменных отношения, представлений, ограничений целостности, полномочий и т.д., но также содержит всю необходимую управляющую информацию, которая позволит системе обеспечить независимость от размещения, фрагментации и репликации. Возникает вопрос: "Где и как должен храниться сам каталог?". Имеется несколько перечисленных ниже возможностей.

1. *Централизованное хранение.* Единственный общий каталог хранится на отдельном центральном узле.
2. *Полная репликация.* Общий каталог целиком хранится на каждом узле.
3. *Частичное секционирование.* Каждый узел поддерживает собственный каталог для объектов, которые на нем хранятся. Общий каталог представляет собой объединение всех этих непересекающихся локальных каталогов.
4. *Сочетание подходов 1 и 3.* На каждом узле поддерживается свой локальный каталог, как предусмотрено в подходе 3. Кроме того, отдельный центральный узел сопровождает объединенную копию всех этих локальных каталогов, как предусмотрено в подходе 1.

Каждый подход имеет свои недостатки. При подходе 1 очевидно нарушается принцип "отсутствия зависимости от центрального узла". При подходе 2 в значительной мере утрачивается независимость узлов, поскольку всякое обновление каталога должно распространяться на каждый узел. При подходе 3 нелокальные операции становятся слишком дорогостоящими (для поиска удаленного объекта требуется доступ в среднем к половине всех узлов). Подход 4 более эффективен по сравнению с подходом 3 (для поиска удаленного объекта требуется доступ лишь к одному удаленному каталогу), но в этом случае вновь нарушается принцип отсутствия зависимости от центрального узла. Поэтому

на практике в системах обычно не используется *ни один* из этих четырех подходов! В качестве примера рассмотрим подход, который применяется в системе R* [21.37].

Прежде чем описать структуру каталога системы R*, необходимо сказать несколько слов о механизме именования объектов в этой системе. Присваивание имен объектам — это вообще очень важный вопрос для распределенных систем. Поскольку два отдельных узла X и У могут иметь объект (скажем, базовую переменную отношения) с одним и тем же именем, допустим, R, требуется некоторый механизм (обычно — уточнение с помощью имени узла) для того, чтобы *устранить неоднозначность,* т.е. гарантировать уникальность имени в пределах всей системы. Но, если уточненные имена, такие как X.R и Y. R, будут предоставляться системой пользователю, будет нарушено требование независимости от размещения. Поэтому необходимы средства отображения имен для пользователя в соответствующие системные имена.

Теперь изложим подход к рассматриваемой проблеме, принятый в системе R*. В этой системе различаются вводимые имена, с помощью которых пользователи обычно обращаются к объектам (например, в конструкции SELECT языка SQL), и общесистемные имена, которые представляют собой глобально уникальные внутренние идентификаторы для этих же объектов. *Общесистемные имена* имеют четыре описанных ниже компонента.

- *Идентификатор создателя,* т.е. идентификатор пользователя, который впервые вызвал на выполнению операцию CREATE для создания рассматриваемого объекта.
- *Идентификатор узла создателя,* т.е. идентификатор узла, на котором была введена соответствующая операция CREATE.
- *Локальное имя,* т.е. неуточненное имя объекта.
- *Идентификатор узла происхождения,* т.е. идентификатор узла, на котором объект хранился первоначально.

Идентификаторы пользователя уникальны на узле, тогда как идентификаторы узла уникальны глобально. Рассмотрим, например, следующее общесистемное имя.

```
MARILYN @ NEWYORK . STATS @ LONDON
```

Оно обозначает объект (для определенности будем считать, что это — базовая переменная отношения) с локальным именем STATS, созданный пользователем MARILYN на узле в Нью-Йорке и первоначально сохраненный на узле в Лондоне[3]. Это имя гарантированно защищено от каких-либо изменений, даже если объект в дальнейшем будет перемещен на другой узел (см. ниже).

Как уже указывалось, пользователи обычно ссылаются на объекты с помощью вводимых имен. Вводимое имя, как показано ниже, представляет собой простое, неуточненное имя — либо компонент "локальное имя" общесистемного имени (в приведенном выше примере — STATS), либо синоним для этого общесистемного имени, который в системе R* определяется с помощью специального оператора CREATE SYNONYM.

```
CREATE SYNONYM MSTATS FOR MARILYN @ NEWYORK . STATS @ LONDON ;
```

---

[3] В системе R* базовые переменные отношения отображаются непосредственно на хранимые переменные отношения. А фактически такой подход применяется почти во всех системах, известных автору. Некоторые критические замечания по поводу указанного состояния дел приведены в приложении А.

После выполнения этого оператора пользователь сможет с одинаковым успехом ввести любой из двух приведенных ниже операторов.

```
SELECT ... FROM STATS ... ;
SELECT ... FROM MSTATS ... ;
```

В первом случае, т.е. при использовании локального имени, система будет исходить из того, что все компоненты общесистемного имени определяются по умолчанию, а именно — что объект создан данным пользователем, создан на данном узле и сохранен на данном узле. Кстати, благодаря такому допущению по умолчанию старые приложения системы System R могут выполняться без каких-либо исправлений в новой версии системы R* (т.е. после переопределения данных системы System R для системы R*).

Во втором случае, т.е. при использовании синонима, система для определения общесистемного имени обращается к соответствующей **таблице синонимов.** Таблицы синонимов могут рассматриваться как первый компонент каталога. Каждый узел поддерживает некоторый набор таких таблиц, созданных для каждого пользователя данного узла, что позволяет системе отображать применяемые пользователем синонимы в соответствующие общесистемные имена.

Кроме таблиц синонимов, на каждом узле поддерживаются описанные ниже данные.

1. Запись каталога для каждого объекта, местом создания которого является указан ный узел.
2. Запись каталога для каждого объекта, **хранимого в настоящее время** на данном узле.

Предположим теперь, что пользователь выдал запрос, содержащий ссылку на синоним MSTATS. Сначала система выполнит поиск соответствующего общесистемного имени в соответствующей таблице синонимов (исключительно локальный просмотр). После того как станет известен узел создания (в нашем примере это узел в Лондоне), можно будет опросить каталог узла в Лондоне (здесь мы для общности подразумеваем удаленный просмотр — первое удаленное обращение). Каталог узла из Лондона будет содержать запись для объекта в соответствии с п. 1. Если объект по-прежнему хранится на узле в Лондоне, он будет найден. Однако, если объект перемещен, скажем, на узел в Лос-Анджелесе, запись каталога на узле в Лондоне будет содержать сведения об этом и система должна будет опросить каталог на узле в Лос-Анджелесе (второе удаленное обращение). Каталог на узле в Лос-Анджелесе будет содержать запись для искомого объекта согласно п. 2. Таким образом, для поиска объекта будет выполнено не больше двух удаленных обращений.

Кроме того, если объект вновь потребуется переместить, скажем, на узел в Сан-Франциско, системой будут выполнены описанные ниже действия.

- Вставка записи в каталог на узле в Сан-Франциско.
- Удаление записи из каталога на узле в Лос-Анджелесе.
- Обновление записи каталога на узле в Лондоне, который теперь будет указывать на узел в Сан-Франциско вместо узла в Лос-Анджелесе.

В результате объект всегда может быть найден с помощью не больше двух удаленных обращений. И это полностью распределенная схема — нет никакого узла с основным каталогом и нет никакого единого источника отказа внутри системы.

Отметим, что схема идентификации объектов, которая используется в распределенной СУБД DB2, хотя и подобна описанной выше, совпадает с ней не полностью.

Распространение обновлений

Основная проблема репликации данных, как указывалось в разделе 21.3, заключается в том, что обновление любого заданного логического объекта должно распространяться по всем хранимым копиям этого объекта. Сложности, которые немедленно возникают **в** этом случае, состоят в том, что некоторый узел, содержащий копию данного объекта, в момент обновления может оказаться недоступным, поскольку произошел отказ узла или сети во время обновления. Таким образом, очевидная стратегия немедленного распространения обновлений по всем существующим копиям будет, вероятно, неприемлема, поскольку она подразумевает, что любая операция обновления, а значит, и вся включающая ее транзакция, закончится аварийно, если одна из требуемых копий в момент обновления окажется недоступной. В каком-то смысле при этой стратегии данные будут *менее* доступными, чем при отсутствии репликации. Таким образом, исчезает одно из главных преимуществ репликации, которое указывалось в предыдущем разделе.

Общепринятая схема решения рассматриваемой проблемы (но не единственное возможное решение) состоит в использовании так называемой схемы **первичной копии,** которая действует описанным ниже образом.

- Одна копия для каждого реплицируемого объекта устанавливается как *первичная* копия, а все оставшиеся копии — как вторичные.
- Первичные копии различных объектов находятся на различных узлах (поэтому данная схема также является распределенной).
- Операции обновления считаются логически завершенными, как только обновлена первичная копия. Узел, содержащий такую копию, будет отвечать за распростра нение обновления на вторичные копии в течение некоторого последующего вре мени.
- ***Примечание.*** Это *последующее время,* тем не менее, должно предшествовать опера ции завершения транзакции COMMIT, если должны гарантироваться свойства ACID распределенных транзакций (см. главы 15 и 16). Дополнительные замечания по этому вопросу будут приведены ниже.

Безусловно, данная схема порождает несколько дополнительных проблем, обсуждение большинства которых выходит за рамки данной книги. Отметим также, что эта схема приводит к нарушению принципа локальной автономности, поскольку транзакция может теперь завершиться аварийно из-за того, что удаленная (первичная) копия некоторого объекта недоступна (даже если доступна локальная копия).

Как уже указывалось, вследствие требования гарантии соблюдения свойств ACID транзакций, весь процесс распространения обновлений должен быть завершен прежде, чем соответствующая транзакция может быть зафиксирована *{синхронная репликация).* Однако несколько коммерческих продуктов поддерживают менее строгую форму репликации, в которой распространение обновлений откладывается на позднее время (возможно, на некоторое указываемое пользователем время), а *не* обязательно выполняется в рамках соответствующей транзакции *{асинхронная репликация).* Безусловно, что определение термина *репликация,* к сожалению, носит на себе в той или иной степени

отпечаток свойств этих продуктов, в результате чего, по крайней мере, на коммерческом рынке, под ним почти всегда подразумевается, что распространение обновлений откладывается до тех пор, пока соответствующая транзакция не завершится (например, см. [21.1], [21.16] и [21.18]). Недостаток подобного подхода с задержкой распространения обновлений заключается, безусловно, в том, что база данных не может больше гарантировать согласованности всех ее данных в любой момент[4]. В действительности, пользователь даже может не знать, согласована ли база данных.

Завершая этот подраздел, приведем несколько приведенных ниже дополнительных замечаний в отношении подхода с задержкой распространения обновлений.

1. Концепция репликации в системе с задержкой распространения обновлений мо жет рассматриваться как применение идеи *снимков,* речь о которых шла в главе 10. На самом деле, для обозначения такого вида репликации лучше было бы исполь зовать другой термин. Тогда можно было бы сохранить термин *реплика* для обо значения того, что понимается под ним в обычном смысле (а именно — точная копия). Но следует отметить, что снимки рассматриваются как предназначенные только для чтения (не считая их периодического обновления), тогда как некото рые системы позволяют пользователям обновлять такие *реплики* непосредственно (см. [21.18]). Безусловно, последний вариант представляет собой нарушение прин ципа независимости от репликации.
2. Мы не утверждаем, что задержка распространения обновлений — плохая идея. Это — очевидно самое лучшее, что можно было сделать при определенных обстоя тельствах, как мы убедимся, например, в главе 22. Суть в том, что задержка рас пространения подразумевает, что "реплики" — не настоящие реплики (поскольку возможно, что любое конкретное значение данных на логическом уровне будет представлено с помощью двух или большего количества хранимых значений на физическом уровне, более того, что эти хранимые значения будут разными!), а система — не настоящая распределенная система баз данных.
3. Одна из причин (может быть, даже главная причина), по которой репликация в коммерческих продуктах реализована с задержкой распространения, заключает ся в следующем: альтернатива, т.е. обновление всех дубликатов перед выполне нием операции COMMIT, требует поддержки двухфазной фиксации транзакции (см. следующий подраздел), что может существенно повлиять на производитель ность системы. Именно по этой причине в компьютерных журналах иногда встре чаются статьи с озадачивающими названиями, такими как "Репликация или двух фазная фиксация?" (озадачивающими, поскольку в них фактически сравниваются характеристики двух принципиально различных подходов).

---

[4] Безусловно, если все операции проверки целостности выполняются немедленно (см. главы 9 и 16), то такие ситуации становятся невозможными. И даже если такая проверка откладывается до этапа вызова оператора COMMIT (а такой подход рассматривается нами как логически неправильный, но применяется во многих системах), подобные ситуации не могут возникнуть. Поэтому, в определенном смысле, отложенное распространение можно рассматривать как "еще более логически неправильный подход", чем отложенную проверку (если только можно рассуждать в терминах "более" или "менее" неправильного).

Управление восстановлением

Как уже было описано в разделе 21.3, управление восстановлением в распределенных системах обычно базируется на протоколе **двухфазной фиксации транзакций** (или некоторых его вариантах). Двухфазная фиксация транзакций требуется в *любой* среде, где отдельная транзакция может взаимодействовать с несколькими автономными диспетчерами ресурсов. Однако в распределенных системах ее использование приобретает особую важность, поскольку рассматриваемые диспетчеры ресурсов, т.е. локальные СУБД, функционируют на отдельных узлах и поэтому в *значительной* мере автономны. Рассмотрим некоторые особенности этого процесса, описанные ниже.

1. Принцип "отсутствия зависимости от центрального узла" предписывает, что функции координатора не должны назначаться одному выделенному узлу в сети, а должны выполняться на различных узлах для различных транзакций. Обычно управление транзакцией передается на тот узел, на котором она была инициировано. Поэтому каждый узел (как правило) должен быть способен выполнять функции координатора для некоторых транзакций и выступать в качестве участника выполнения остальных транзакций.
2. Для двухфазной фиксации транзакций координатор должен взаимодействовать с каждым участвующим узлом, что подразумевает увеличение количества сообщений, а значит, дополнительную нагрузку на коммуникации.
3. Если узел Y является участником транзакции, выполняемой по протоколу двухфазной фиксации и координируемой узлом X, узел Y *должен* делать то, что предписывает ему узел X (фиксацию результатов транзакции или ее откат в зависимости от того, что именно потребуется), а это означает потерю (хотя и относительно не значительную) локальной автономности.

Рассмотрим повторно процесс двухфазной фиксации транзакций, описанный ранее, в главе 15. Обратимся к рис. 21.4, на котором показано взаимодействие между координатором и обычным участником. Для общности предположим, что координатор и участник находятся на разных узлах, кроме того, на данном рисунке ось времени направлена слева направо (но иногда временная последовательность нарушается!). К тому же для упрощения предположим, что в транзакции затребовано выполнение операции COMMIT, а не ROLLBACK. После получения запроса на операцию COMMIT координатор организует следующий описанный ниже двухфазный процесс.

- Каждому участнику координатор отдает распоряжение "приготовиться к фиксации или откату" транзакции. На рис. 21.4 показано сообщение "приготовиться", отправленное в момент *t1* и полученное участником в момент *t2*. Далее участник принудительно помещает запись в журнал локального агента из своего физического журнала, а затем выдает координатору подтверждение "OK". Безусловно, если возникнет какая-либо ошибка (в частности, если произойдет сбой в работе локального агента), будет передано сообщение "Not OK". На рисунке это сообщение, которое для простоты обозначено как "OK", отправлено в момент *t3* и получено координатором в момент *t4*. Как уже отмечалось выше, участник теперь теряет автономность: он *должен* делать то, что ему будет предписывать координатор. Кроме того, любые ресурсы, которые заблокированы локальным агентом, *должны оставаться заблокированными* до тех пор, пока участник не получит и не выполнит распоряжение координатора.

- После получения подтверждения от всех участников координатор принимает решение либо *зафиксировать* транзакцию, если все ответы были "ОК", либо выполнить *откат* транзакции в противном случае. Затем в момент t5 координатор вносит в свой физический журнал запись о принятом решении. Время t5 служит границей между первой и второй фазами фиксации транзакции.
- Будем считать, что было принято решение о *фиксации* транзакции. В этом случае координатор отдаст распоряжение всем участникам "выполнить", т.е. запустить обработку операции фиксации для локального агента. На рис. 21.4 показано сообщение "выполнить", отправленное в момент *t6* и полученное участником в момент *t7*. Участник выполняет операцию фиксации для локального агента и отправляет подтверждение "выполнено" координатору. На рисунке подтверждение отправлено координатору в момент *t8* и получено координатором в момент *t9*.
- После того как координатором будут получены все подтверждения, процесс полностью завершается.

Рис. 21.4. Двухфазная фиксация транзакции

На практике, безусловно, этот процесс в целом значительно сложнее, чем описано выше, поскольку необходимо еще позаботиться о возможных отказах узла или сети, которые могут произойти в любой момент. Предположим, например, что на узле координатора сбой произошел в некоторый момент t между отметками времени *t5* и *t6*. В процессе восстановления работы узла процедура повторного пуска обнаружит в журнале сведения о том, что в момент отказа некоторая транзакция находилась на второй фазе двухфазного процесса фиксации, после чего процесс будет продолжен, начиная с отправки участникам сообщений "выполнить". Отметим, что в период от *t3* до t7 участник находится в состоянии "отсутствия определенной информации" о состоянии транзакции. Если произойдет отказ узла координатора в момент t, как указано выше, период "отсутствия определенной информации" может оказаться достаточно длительным.

Теоретически, безусловно, было бы желательно, чтобы процесс двухфазной фиксации оказался устойчивым *к любым возможным* сбоям. К сожалению, несложно понять, что эта цель, по сути, недостижима, т.е. не существует какого-либо конечного протокола, который бы *гарантировал,* что все участники одновременно зафиксируют успешно завершившуюся транзакцию или одновременно ее отменят, столкнувшись при выполнении с некоторым отказом. Предположим обратное, что такой протокол существует. Пусть *N*— минимальное количество сообщений, которые требуются для работы подобного протокола. Теперь допустим, что последнее из этих *N* сообщений утеряно из-за сбоев. Тогда или это сообщение не было необходимым, а это противоречит предположению о том, что *N*— минимальное количество сообщений, или протокол в такой ситуации работать не будет. И в том, и в другом случае возникает противоречие, из которого следует, что такого протокола не существует.

Но несмотря на этот удручающий факт, существует ряд описанных ниже усовершенствований, которые можно внести в основной алгоритм для повышения его производительности.

- Во-первых, как было отмечено в аннотации к [15.6], если агент на некотором кон кретном узле участника выполняет операции только чтения, такой участник при завершении первой фазы процесса может ответить "игнорируйте мое присутст вие" и координатор действительно может игнорировать такого участника во вто рой фазе процесса.
- Во-вторых (и это указано в аннотации к [15.6]), если все участники в первой фазе процесса ответят "игнорируйте мое присутствие", вторая фаза может быть полно стью пропущена.
- И в-третьих, существует два важных варианта основной схемы, которые называ ются вариантами *предполагаемой фиксации* и *предполагаемого отката,* соответст венно, которые будут описаны более подробно в нескольких следующих абзацах.

В общем случае в результате применения схем предполагаемой фиксации и предполагаемого отката сокращается количество передаваемых сообщений, как в случае успешного завершения (для предполагаемой фиксации), так и в случае отказа (для предполагаемого отката). Прежде чем приступить к объяснению действия этих двух схем, отметим, что основной механизм, как описано выше, требует, чтобы координатор помнил о своем решении, пока не получит подтверждения от каждого участника. Причина, очевидно, заключается в том, что если произойдет отказ узла какого-либо из участников в период "отсутствия определенной информации о транзакции", то при перезапуске он будет вынужден вновь отослать координатору запрос, чтобы узнать, каково же было решение координатора в отношении данной транзакции. Но как только все подтверждения будут получены, координатор будет считать, что все участники выполнили то, что им было предписано, и поэтому он может "забыть" о данной транзакции.

Теперь рассмотрим схему предполагаемой фиксации. Согласно этой схеме, от участников требуется подтверждение сообщений "произвести откат" ("отменить"), а не сообщений "зафиксировать" ("выполнить"). Поэтому координатор может забыть о транзакции после того, как передаст широковещательное сообщение о своем решении, при условии, что это решение — "зафиксировать". Если в период отсутствия определенной информации произойдет отказ узла участника, то в процессе перезапуска участник, как всегда, запросит у координатора информацию о состоянии данной транзакции. Если координатор

еще помнит об этой транзакции, т.е. еще ожидает подтверждений от ее участников, его решением должно быть "произвести откат", в противном случае этим решением должно быть "зафиксировать".

Схема **предполагаемого отката** — это противоположная схема. От участников требуется подтверждение сообщений "зафиксировать", а не сообщений "произвести откат". И координатор может забыть о транзакции после того, как передаст широковещательное сообщение о своем решении, если это решение — "произвести откат". Если на узле участника в период отсутствия определенной информации произойдет сбой, то после перезапуска участник должен передать запрос координатору о принятом им решении. Если координатор еще помнит о данной транзакции, т.е. еще ожидает подтверждений от ее участников, его решением будет "зафиксировать", в противном случае — "произвести откат".

Любопытно отметить (что в определенной степени противоречит здравому смыслу), что метод предполагаемого отката кажется более предпочтительным по сравнению со схемой предполагаемой фиксации (это *противоречит здравому смыслу,* поскольку, безусловно, большинство транзакций завершаются успешно, а схемой предполагаемой фиксации ограничивается количество сообщений именно в случае успеха). Проблема, связанная со схемой предполагаемой фиксации, состоит в следующем. Предположим, что на узле координатора сбой произошел во время первой фазы, т.е. *до* принятия решения. После перезапуска узла координатора данная транзакция будет отменена, поскольку она не завершена. Следовательно, некоторые участники будут запрашивать координатора о его решении относительно данной транзакции. Координатор такой транзакции не помнит, поэтому предполагается решение "зафиксировать", что, конечно же, неверно.

Чтобы избежать такой "ложной фиксации", координатор (использующий схему предполагаемой фиксации) должен в начале первой фазы поместить в свой физический журнал запись, содержащую список всех участников данной транзакции. (Если теперь на узле координатора произойдет сбой во время первой фазы фиксации транзакции, то после его перезапуска он сможет передать всем участникам сообщение "выполнить откат".) А такая необходимость выполнения физической операции ввода—вывода при обращении к журналу координатора создает критический путь в процессе выполнения *каждой транзакции.* Поэтому схема предполагаемой фиксации не так заманчива, как может показаться на первый взгляд. В действительности, можно без преувеличения сказать, что ко времени написания данной книги схема предполагаемого отката стала *фактическим* стандартом в реализованных системах.

Управление параллельностью

Как указано в разделе 21.3, управление параллельным доступом в большинстве распределенных систем строится на использовании механизма блокировки, т.е. точно так, как и в большинстве нераспределенных систем. Однако в распределенных системах запросы на проверку, установку и отмену блокировки становятся *сообщениями* (если считать, что рассматриваемый объект расположен на удаленном узле), а сообщения создают дополнительные издержки. Рассмотрим, например, транзакцию т обновления объекта, для которого существуют дубликаты на *n* удаленных узлах. Если каждый узел отвечает за блокировку объектов, которые на нем хранятся (как предполагается в соответствии с принципом локальной независимости), то непосредственная реализация будет требовать по крайней мере 5л таких сообщений:

- n запросов на блокировку;
- n разрешений на блокировку;
- n сообщений об обновлении;
- n подтверждений;
- n запросов на снятие блокировки.

Безусловно, мы можем разумно воспользоваться, как и в предыдущем случае, *комбинированными* сообщениями. Например, можно объединить сообщение запроса на блокировку и сообщение об обновлении, а также сообщение о разрешении блокировки и сообщение о подтверждении. Но даже в этом случае общее время обновления может быть на порядок больше, чем в централизованной системе.

Для решения проблемы обычно выбирается стратегия **первичной копии,** описанная выше, в подразделе "Распространение обновлений". Для данного объекта А все операции блокировки будет обрабатывать узел, содержащий его первичную копию (напомним, что первичные копии различных объектов будут в общем случае размещаться на разных узлах). При использовании этой стратегии набор всех копий объекта с точки зрения блокировки можно рассматривать как единый объект, а общее количество сообщений будет сокращено с 5л до 2л+3 (один запрос блокировки, одно разрешение блокировки, л обновлений, л подтверждений и один запрос на снятие блокировки). Но обратите внимание на то, что это решение влечет за собой серьезную потерю независимости — транзакция может теперь не завершиться, если первичная копия окажется недоступной, даже если в транзакции выполнялось лишь чтение и локальная копия была доступна. (Отметим, что блокировка первичной копии требуется не только для операций обновления, но и для операций выборки [15.6]. Таким образом, у стратегии первичной копии есть нежелательный побочный эффект — снижение уровня производительности и доступности как для выборки, так и для обновления.)

Другая проблема, касающаяся блокировок в распределенных системах, состоит в том, что блокировка может привести к состоянию **глобальной взаимоблокировки,** охватывающей два или больше узлов. Обратимся, например, к рис. 21.5, где показана описанная ниже последовательность событий.

1. Агент транзакции Т2 на узле х ожидает, пока агент транзакции Т1 на узле х осво бодит блокировку.
2. Агент транзакции т1 на узле X ожидает, пока агент транзакции Т1 на узле Y завер шит транзакцию.
3. Агент транзакции Т1 на узле Y ожидает, пока агент транзакции Т2 на узле У осво бодит блокировку.
4. Агент транзакции Т2 на узле Y ожидает, пока агент транзакции Т2 на узле X завер шит транзакцию. Налицо явная *взаимоблокировка!*

В состоянии глобальной взаимоблокировки, подобном только что рассмотренному, *никакой узел не может обнаружить ситуацию взаимоблокировки, используя лишь собственную внутреннюю информацию.* Иными словами, в локальном графе ожиданий нет никаких циклов, и подобный цикл возникает только при объединении локальных графов в общий глобальный граф. Итак, для обнаружения состояния глобальной блокировки требуются дополнительные коммуникационные расходы, поскольку необходимо, чтобы отдельные локальные графы рассматривались вместе.

Изящная (и распределенная) схема для определения состояния глобальной блокировки описана в статьях о системе R* (например, [21.37]).

Примечание. Как указывалось в главе 16, в действительности не все системы распознают состояние взаимоблокировки — некоторые вместо этого просто используют механизм тайм-аута. По очевидным причинам данное замечание относится, в частности, и к распределенным системам.

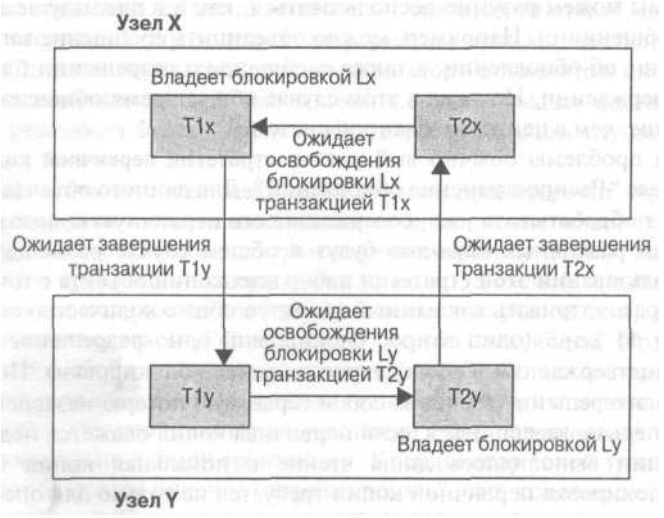


Рис. 21.5. Пример состояния глобальной взаимоблокировки

21.5. СИСТЕМЫ "КЛИЕНТ/СЕРВЕР"

Как отмечалось в разделе 21.1, системы "клиент/сервер" могут рассматриваться как частный случай распределенных систем. Точнее, система "клиент/сервер" — это распределенная система, в которой одни узлы — *клиенты,* а другие — *серверы;* все данные размещены на узлах, которые являются серверами; все приложения выполняются на узлах-клиентах и "швы видны пользователю" (полная локальная независимость не предоставляется). Обратимся к рис. 21.6 (или к рис. 2.6 из главы 2).

В конце 1980-х годов и в период от начала до середины 1990-х годов повышенный коммерческий интерес проявлялся к системам "клиент/сервер" и сравнительно небольшой интерес — к настоящим распределенным системам общего назначения. В последнее время эти тенденции немного изменились, как показано в следующем разделе, но системы "клиент/сервер" все еще играют важную роль, поэтому и потребовалось включить в книгу данный раздел.

Прежде всего, напомним, что термин "клиент/сервер" определяет *архитектуру,* или логическое разделение обязанностей. **Клиент** — это приложение, которое называют также *интерфейсной частью* (front end), а **сервер** — *прикладной частью* (back end) или СУБД. Однако именно потому, что всю систему можно так четко разделить на две части, появилась возможность эксплуатации ее частей на разных компьютерах. И эта возможность по многим причинам оказалась настолько привлекательной (см. главу 2), что под понятием *система "клиент/сервер"* на практике подразумевается исключительно тот случай, когда

клиент и сервер действительно размещаются на разных компьютерах[5]. Хотя это и несколько вольное использование термина, но оно широко распространено и поэтому мы будем его придерживаться.

Рис. 21.6. Система "клиент/сервер"

Напомним, что возможны несколько вариантов основной схемы, которые описаны ниже.

- Несколько клиентов могут совместно использовать один и тот же сервер (фактически это обычная практика).
- Отдельный клиент может иметь доступ к нескольким серверам. Этот вариант, в свою очередь, подразделяется на два представленных ниже случая.
  а) Клиент ограничен доступом за один раз лишь к одному серверу, т.е. каждый отдельный запрос к базе данных должен быть направлен только на один сервер. Невозможно в пределах одного запроса получить данные с двух или нескольких различных серверов. Более того, пользователь должен знать, на каком именно сервере хранятся те или иные части данных.
  б) Клиент может иметь одновременный доступ к нескольким серверам, т.е. отдельный запрос может объединять данные с нескольких серверов. А это означает, что несколько серверов представляются клиенту так, как будто это на самом деле один сервер. Пользователь не должен знать, какие части данных хранятся на каждом сервере.

[5] Также по очевидным соображениям используется термин "двухуровневая система", который, по сути, имеет такой же смысл.

Но в случае *б)* фактически описана система распределенной базы данных (швы оказываются скрытыми). Это не совсем то, что обычно подразумевают под термином "клиент/сервер", поэтому мы данный случай рассматривать не будем.

Стандарты для систем "клиент/сервер"

Существует несколько стандартов, имеющих отношение к системам "клиент/сервер". Эти стандарты описаны ниже.

- Прежде всего, определенные функции для поддержки систем "клиент/сервер" включены в стандарт **языка SQL.** Обсуждение этих возможностей отложим до раздела 21.7. Кроме того, имеется стандарт ISO для **удаленного** доступа **к данным** (Remote Data Access — RDA) (см. [21.23] и [21.24]). Задача спецификации RDA состоит в определении **форматов и протоколов** взаимодействия в среде "клиент/сервер". Подразумевается, что клиент формулирует запрос к базе данных в стандартной форме языка SQL (по существу, применяется подмножество стандарта SQL), а сервер поддерживает стандартный каталог (также в основном соответствующий требованиям стандарта SQL). Кроме того, определены конкретные форматы представления для сообщений, передаваемых между клиентом и сервером (запросы SQL, данные и результаты, диагностическая информация).

  Третий, и последний, стандарт, который мы здесь упоминаем, — стандарт **архитектуры распределенных реляционных** баз **данных** (Distributed Relational Database Architecture — DRDA), предложенный компанией IBM [21.22] (он является фактически признанным, а не официально утвержденным стандартом). Стандарты DRDA и RDA имеют аналогичное назначение, но стандарт DRDA отличается от стандарта RDA во многих важных отношениях. В частности, многие характеристики стандарта DRDA обусловлены его происхождением (он разработан компанией IBM). Например, в стандарте DRDA клиент не обязательно должен использовать стандартную версию языка SQL, поэтому разрешено применение любых произвольных диалектов языка SQL. Следствием этого, возможно, является повышение производительности, поскольку клиенту разрешается использовать некоторые специфические возможности сервера. Но, с другой стороны, этот подход снижает возможности переносимости, поскольку подобные специфические функции не являются скрытыми от клиента, т.е. клиенту известно, с каким типом сервера он работает. Аналогично этому, в стандарте DRDA не подразумевается наличие какой-либо конкретной структуры каталога сервера. Форматы и протоколы DRDA существенно отличаются от форматов стандарта **RDA.** По существу, стандарт DRDA базируется на собственной архитектуре и собственных стандартах IBM, в то время как стандарт RDA основывается на международных стандартах, независимых от конкретных поставщиков.

Более подробно стандарты RDA и DRDA в настоящей книге не рассматриваются. (См. [21.20] и [21.28], где приведен их сравнительный анализ.)

Программирование приложений "клиент/сервер"

Необходимо отметить, что система "клиент/сервер" — это частный случай распределенных систем в целом. Как указывалось во введении к этому разделу, она может рассматриваться как распределенная система, в которой все запросы создаются на одном

узле, а вся обработка выполняется на другом, если считать для простоты, что имеется лишь один узел клиента и один узел сервера.

Примечание. Согласно этому простому определению, безусловно, узел клиента вовсе не является *узлом системы баз данных* в полном смысле этого понятия, и такая система противоречит определению систем распределенных баз данных общего назначения, которое было дано в разделе 21.2. Разумеется, узел клиента может иметь собственную локальную базу данных, однако такие базы данных не имеют непосредственного отношения к системе "клиент/сервер" как таковой.

Но, как бы то ни было, подход "клиент/сервер" имеет определенные особенности с точки зрения программирования (как и сами распределенные системы). На одну из таких особенностей мы уже указывали при обсуждении задачи 7 (распределенная обработка запросов) в разделе 21.3, а именно — на то, что реляционные системы по определению и по происхождению являются системами, в которых данные обрабатываются на уровне множеств. В системах "клиент/сервер" (как и в самих распределенных системах) чрезвычайно важно то, что программист, пишущий приложение, *не* "использует сервер как некоторое средство доступа", создавая при этом код обработки данных на уровне записей. Вместо этого функционирование приложения в максимально возможной степени должна быть основано на использовании запросов на уровне множеств. В противном случае неизбежны существенные потери производительности системы, связанные с передачей слишком большого количества сообщений.

Примечание. В терминах языка SQL предыдущее высказывание означает, что требуется в максимально возможной степени *избегать использования курсоров,* т.е. циклов с оператором FETCH и форм CURRENT для операций UPDATE и DELETE (см. главу 4).

Количество сообщений, передаваемых между клиентом и сервером, может быть сокращено еще больше, если система предоставляет в распоряжение пользователя некоторый механизм поддержки хранимых процедур. Хранимые процедуры представляют, по существу, предварительно откомпилированные программы, которые *хранятся на узле сервера* (и известны серверу). Клиент обращается к хранимой процедуре с помощью механизма вызова удаленных процедур (Remote Procedure Call — RPC). Поэтому, в частности, потери в производительности, связанные с обработкой данных на уровне записей в системе "клиент/сервер", могут быть частично компенсированы за счет создания подходящих хранимых процедур, позволяющих выполнить обработку данных непосредственно на узле сервера.

Примечание. Хотя это и не имеет прямого отношения к теме обработки данных в системах "клиент/сервер", необходимо отметить, что более высокая производительность — не единственное преимущество, которое предоставляют хранимые процедуры. Другие преимущества хранимых процедур приведены ниже.

- Хранимые процедуры позволяют скрыть от пользователя множество специфиче ских особенностей СУБД и базы данных и благодаря этому достичь более высокой степени независимости от данных по сравнению с тем случаем, когда хранимые процедуры не используются.
- Одна хранимая процедура может совместно использоваться многими клиентами.
- Оптимизация может быть осуществлена при создании хранимой процедуры, а не во время выполнения. (Безусловно, это преимущество проявляется лишь в тех системах, в которых оптимизация обычно осуществляется во время выполнения.)

- Хранимые процедуры позволяют обеспечить более высокую степень безопасности данных. Например, некоторому пользователю может быть разрешено вызывать определенную процедуру, но не разрешено непосредственно обрабатывать данные, к которым он может иметь доступ через эту хранимую процедуру.

Недостатком хранимых процедур является то, что поставщики программного обеспечения предоставляют в этой области слишком отличающиеся между собой средства, а расширение языка SQL для поддержки хранимых процедур появилось, к сожалению, лишь в 1996 году. Как указывалось в главе 4, это средство называется SQL/PSM (Persistent Stored Module — постоянный хранимый модуль).

21.6. НЕЗАВИСИМОСТЬ ОТ СУБД

Вновь обратимся к обсуждению двенадцати общих целей систем распределенных баз данных. Последняя из перечисленных целей предусматривала обеспечение *независимости от СУБД*. Как уже указывалось в разделе 21.3, предположение о строгой однородности оказывается неоправданно строгим, поскольку в действительности необходима лишь поддержка любыми СУБД на различных узлах одного и того же интерфейса. Как указывалось в разделе 21.3, если, например, СУБД Ingres и Oracle поддерживают официальный стандарт SQL (не больше и не меньше!), можно будет добиться, чтобы они играли роли партнеров в неоднородной распределенной системе. Фактически такая возможность — один из первых доводов, который обычно приводится в пользу стандарта языка SQL. Здесь эта возможность рассматривается более подробно.

Примечание. Обсуждение будет построено конкретно на примере СУБД Ingres и Oracle, просто для того, чтобы дать более реальное представление о состоянии дел. Тем не менее, рассматриваемые концепции, безусловно, имеют общее применение.

Шлюзы

Предположим, что имеются два узла (X и Y), на которых установлены СУБД Ingres и Oracle, соответственно. Также предположим, что некоторый пользователь и на узле X желает получить доступ к единой распределенной базе данных, содержащей данные как из базы данных Ingres на узле X, так и из базы данных Oracle на узле Y. По определению пользователь и — это пользователь СУБД Ingres, и, следовательно, с точки зрения данного пользователя распределенная база данных должна быть базой данных Ingres. В результате обязанность предоставлять необходимую функциональную поддержку ложится на СУБД Ingres. В чем же заключается такая поддержка?

В принципе, все довольно просто: СУБД Ingres должна предоставить специальную программу, задача которой — "обеспечить возможность обращаться к СУБД Oracle, как к СУБД Ingres". Такие программы обычно называют **шлюзами** (а в последнее время — *оболочками)*. Рассмотрим[6] рис. 21.7. Шлюз может функционировать на узле Ingres, на узле Oracle или (как это показано на рисунке) на некотором специальном узле между двумя

---

[6] Для обозначения структуры, показанной на данном рисунке, иногда употребляется термин "трехуровневая система" (по очевидным соображениям). Он используется также по отношению к другим системным конфигурациям, которые аналогично включают три отдельных компонента (в частности, обратитесь к обсуждению промежуточного программного обеспечения, приведенному в следующем подразделе).

СУБД. Независимо от того, где именно установлен шлюз, необходимо, чтобы он обеспечивал все перечисленные ниже функции. (Обратите внимание на то, что при реализации некоторых из этих функций возникают весьма сложные проблемы. Однако в стандартах RDA и DRDA, которые рассматривались в разделе 21.5, учтено наличие некоторых из таких проблем, как и в стандарте, основанном на использовании языка XML (см. главу 27).)

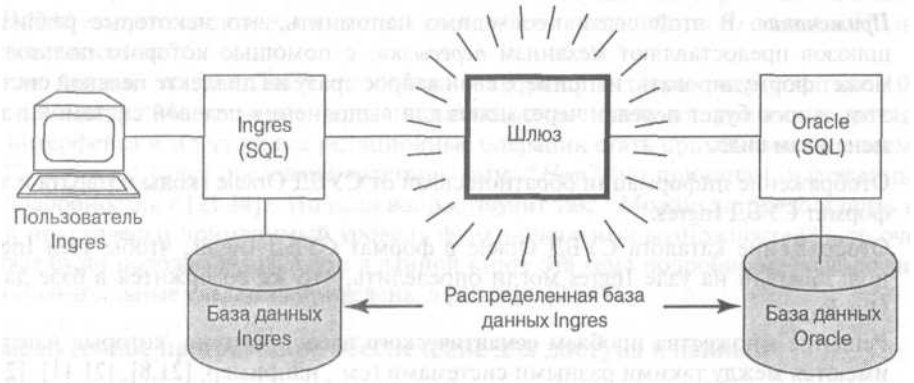


Рис. 21.7. Гипотетический шлюз к СУБД Oracle, предоставленный в СУБД Ingres

- Реализация протоколов для обмена информацией между СУБД Ingres и Oracle. Такая реализация, кроме всего прочего, должна включать средства преобразования сообщений из формата, в котором исходные предложения передаются из СУБД Ingres, в формат, подходящий для СУБД Oracle, а также средства отображения формата сообщений, в котором передаются результаты из СУБД Oracle, в формат, требуемый для СУБД Ingres.
- Предоставление возможностей "сервера SQL" для СУБД Oracle (функционально он аналогичен интерактивному серверу SQL, который в настоящее время имеется в большинстве продуктов). Другими словами, должна существовать возможность выполнять с помощью шлюза в базе данных Oracle произвольные незапланированные запросы на языке SQL. Для этого шлюз должен обеспечивать динамическую поддержку языка SQL или, скорее всего, интерфейса уровня вызовов (Call-Level Interface — CLI), такого как SQL/CLI, ODBC или JDBC на узле Oracle (см. главу 4). *Примечание.* В качестве альтернативы шлюз может обеспечить непосредственное использование интерактивного процессора SQL, предоставляемого СУБД Oracle.
- Отображение между типами данных Ingres и Oracle. Эта задача включает ряд подзадач, которые должны учитывать различия в процессорах (т.е. различные структуры машинных слов), различия в кодировке символов (иначе сравнения символьных строк и запросы с конструкциями ORDER BY могут дать непредсказуемые результаты), различия в форматах чисел с плавающей запятой (широко известная проблема), различия в поддержке дат и времени (в настоящее время автору неизвестны СУБД, которые предоставляли бы идентичную поддержку в этой области), не говоря уже о различиях в типах, определяемых пользователем. Более подробную информацию можно найти в [15.6], где приводится развернутое обсуждение данных вопросов.

- Отображение диалекта языка SQL СУБД Ingres в диалект языка SQL СУБД Oracle (поскольку фактически ни СУБД Ingres, ни СУБД Oracle не поддерживают точно стандарт языка SQL). На самом деле каждый продукт поддерживает определенные возможности, которые не поддерживает другой, а также есть и такие языковые средства, которые в обоих продуктах имеют один и тот же синтаксис, но различную семантику.

  Примечание. В этой связи необходимо напомнить, что некоторые реализации шлюзов предоставляют механизм *пересылки,* с помощью которого пользователь может формулировать, например, свой запрос сразу на диалекте целевой системы; этот запрос будет передан через шлюз для выполнения целевой системой в неизмененном виде.

- Отображение информации обратной связи от СУБД Oracle (коды возврата и т.д.) в формат СУБД Ingres.
- Отображение каталога СУБД Oracle в формат СУБД Ingres, чтобы узел Ingres и пользователи на узле Ingres могли определить, что же содержится в базе данных Oracle.
- Решение множества проблем **семантического несоответствия,** которые наверняка имеются между такими разными системами (см., например, [21.8], [21.11], [21.14], [21.36] и [21.38]). В качестве примеров таких проблем можно указать различия в способах именования (СУБД Ingres позволяет использовать имя атрибута EMP#, а в СУБД Oracle приходится заменять его на EMPNO); различия в типах данных (СУБД Ingres может использовать символьную строку для представления атрибута, который в СУБД Oracle представляется числовой величиной); различия в единицах измерения (в СУБД Ingres могут использоваться сантиметры, а в СУБД Oracle используются дюймы); различия в логических представлениях информации (в СУБД Ingres можно опускать кортежи в тех случаях, где в СУБД Oracle используются неопределенные значения) и многие другие примеры.
- Выполнение обязанностей участника (в варианте СУБД Ingres) в протоколе двухфазной фиксации (подразумевается, что транзакции Ingres могут выполнять восстановления в базе данных СУБД Oracle). Когда именно шлюз будет на самом деле готов выполнить эту функцию, зависит от возможностей, предоставляемых диспетчером транзакций на узле Oracle. Стоит подчеркнуть, что ко времени написания этой книги коммерческие диспетчеры транзакций (с некоторыми исключениями) обычно *не* предоставляли все необходимое в этом отношении, а именно — возможность для прикладных программ передавать диспетчеру транзакций команду "приготовиться к завершению транзакции" (как альтернативу безусловной команде завершения, т.е. фиксации или отката).
- Контроль блокировки на узле Oracle данных, которые требуются для узла Ingres, т.е. проверка, действительно ли данные будут заблокированы, когда это потребуется для узла Ingres. И опять-таки, будет ли шлюз на самом деле готов выполнить эту функцию, по-видимому, зависит от ответа на вопрос, соответствует ли архитектура механизма блокировки СУБД Oracle архитектуре механизма блокировки СУБД Ingres.

До сих пор мы рассматривали независимость от СУБД лишь в контексте реляционных систем. А как же быть с нереляционными системами? Существует ли возможность включения нереляционного узла в распределенную систему, в которой все остальные узлы являются реляционными? Можно ли, например, предоставить доступ к узлу IMS из узла Ingres или Oracle? Безусловно, такая возможность была бы очень полезной на практике, поскольку открывала бы доступ к огромному количеству данных[7], хранящихся в системах СУБД IMS и других системах, созданных до появления реляционного подхода. Но можно ли это осуществить?

Если данный вопрос означает: "Можно ли решить такую задачу на все 100%?" (имеется в виду "Могут ли все нереляционные данные стать доступными из реляционного интерфейса и могут ли все реляционные операции стать применимыми к этим данным?"), то ответ будет предельно категоричным: *"Нет"* (по причинам, изложенным во всех подробностях в [21.14]). Но если вопрос звучит так: "Можно ли предоставить некоторый практически применимый уровень функциональных возможностей?", то очевидно ответ будет положительным. Но в данной книге эта тема подробно не рассматривается. Дополнительные сведения приведены в [21.13] и [21.14].

Промежуточное программное обеспечение для доступа к данным

Шлюзы, которые описаны в предыдущем подразделе, иногда более конкретно называются шлюзами типа *"точка-точка"*. Такие шлюзы имеют ряд очевидных недостатков. Во-первых, они предоставляют ограниченную независимость от размещения. Во-вторых, для практически одинаковых приложений может потребоваться использовать несколько отдельных шлюзов (скажем, один — для СУБД DB2, другой — для СУБД Oracle и третий — для СУБД Informix), не имея при этом никакой поддержки, например, для операции соединения, которая включает несколько узлов разных типов, и т.д. Вследствие этого (и несмотря на технические трудности, указанные в предыдущем подразделе) за несколько последних лет через довольно короткие интервалы времени стали появляться продукты, реализующие шлюзы со все более сложными функциональными возможностями. Фактически все разработки, которые относились к так называемому *промежуточному программному обеспечению* (middleware) для доступа к данным или к *связующему программному обеспечению* (mediator), теперь выделились в важное самостоятельное направление программирования.

Очевидно, что указанные выше термины (промежуточное программное обеспечение и оболочка) определены не совсем точно. Любая часть программного обеспечения, используемая для сокрытия различий между отдельными системами, которые предназначены для совместной работы, например, монитор выполнения транзакций, может обоснованно считаться *промежуточным программным обеспечением* [21.3]. Но мы сейчас сосредоточимся на том, что можно назвать промежуточным программным обеспечением для *доступа к данным*. Примером такого программного обеспечения могут служить продукты Cohera компании Cohera, DataJoiner корпорации IBM, а также OmniConnect и InfoHub корпорации Sybase. В качестве примера рассмотрим продукт DataJoiner [21.6].

---

[7] Как показывает обычная деловая практика, в таких системах до сих пор размещено около 85% производственных данных (т.е. огромная часть данных находится в нереляционных системах баз данных и даже в файловых системах). И очень мала надежда на то, что пользователи будут когда-либо переносить эти данные в более новые системы.

Охарактеризовать этот продукт можно несколькими способами (рис. 21.8). С точки зрения отдельного клиента он выглядит, как обычный сервер базы данных (т.е. СУБД). DataJoiner сохраняет данные, поддерживает запросы SQL, предоставляет доступ к каталогу, выполняет оптимизацию запросов и т.д. (Фактически основой системы DataJoiner является версия AIX СУБД DB2 компании IBM.) Но данные хранятся в основном не на узле системы DataJoiner (хотя такая возможность также имеется), а на любом количестве других скрытых от пользователя узлов, которые контролируются несколькими другими СУБД (или даже диспетчерами файлов, подобными, например, VSAM). Таким образом, DataJoiner фактически дает пользователю доступ к виртуальной базе данных, которая представляет собой объединение всех таких скрытых от пользователя баз данных и/или файлов. Кроме того, эта система позволяет использовать запросы[8], которые охватывают все указанные базы данных и/или файлы, а также принимает решение по созданию *глобально оптимальных* планов выполнения запросов, используя заложенные в ней знания о возможностях, скрытых от пользователя систем (а также о характеристиках сети).

Примечание. В продукте DataJoiner эмулируются также некоторые возможности языка SQL СУБД DB2 для систем, которые не поддерживают такие возможности непосредственно. Примером может служить опция WITH HOLD для объявления курсора (см. главу 15).

Система, подобная описанной выше, — далеко еще не полная распределенная система баз данных, поскольку все возможные узлы, скрытые от пользователя, не имеют информации о существовании друг друга (т.е. не могут рассматриваться как равноправные партнеры в совместном деле). Однако если к скрытым от пользователя узлам будет добавлен новый узел, он сможет использовать все возможности, предоставляемые узлам клиентов и, следовательно, выдавать запросы через систему DataJoiner, которая гарантирует доступ к любому узлу (или сразу ко всем остальным узлам). В целом это означает, что система представляет собой так называемую **интегрированную** систему, которая известна и как система, состоящая из нескольких **баз данных** [21.17]. Интегрированная система — это распределенная система, обычно неоднородная, которая поддерживает почти полную локальную автономию. Локальные транзакции в ней управляются локальными СУБД; реализация же глобальных транзакций — это отдельный вопрос [21.7].

Для каждой из скрытых от пользователя систем в программном продукте DataJoiner предусмотрен такой компонент, как драйвер, фактически представляющий собой шлюз "точка-точка", или оболочку, в том смысле, который указан в предыдущем подразделе. (Для доступа к удаленной системе в таких драйверах обычно применяется интерфейс ODBC.) Продукт DataJoiner также поддерживает *глобальный каталог,* который используется, в частности, для определения действий в ситуациях, когда встречается семантическое несовпадение между системами.

Отметим, что система DataJoiner может применяться сторонними поставщиками, которые разрабатывают общие средства (например, средства для написания отчетов, статистические пакеты и т.д.), чтобы не слишком заботиться о различиях между теми продуктами СУБД, на которых предполагается их использовать. Наконец, отметим, что компания IBM

---

[8] На слове "запросы" нужно сделать ударение; возможности обновления неизбежно становятся ограниченными, особенно в связи с тем (но не только из-за этого), что система, скрытая от пользователя, представляет собой, скажем, СУБД IMS или является какой-то другой системой, не поддерживающей язык SQL (опять-таки, см. [21.14]).

недавно включила в технологию DataJoiner свой программный продукт— СУБД DB2; очевидно, это было сделано в целях усовершенствования СУБД DB2, чтобы она взяла на себя роль *единственного настоящего интерфейса* (по крайней мере, единственного настоящего интерфейса IBM) к данным, хранимым во всех формах (ко времени написания данной книги с помощью указанной технологии предоставлялся доступ к данным, хранимым, кроме всего прочего, в СУБД Informix, Oracle, SQL Server и Sybase). Иными словами, с помощью своей СУБД DB2, в которой применяется технология DataJoiner, компания IBM предприняла попытку решить проблему, известную под названием "проблемы интеграции информации" [21.9].

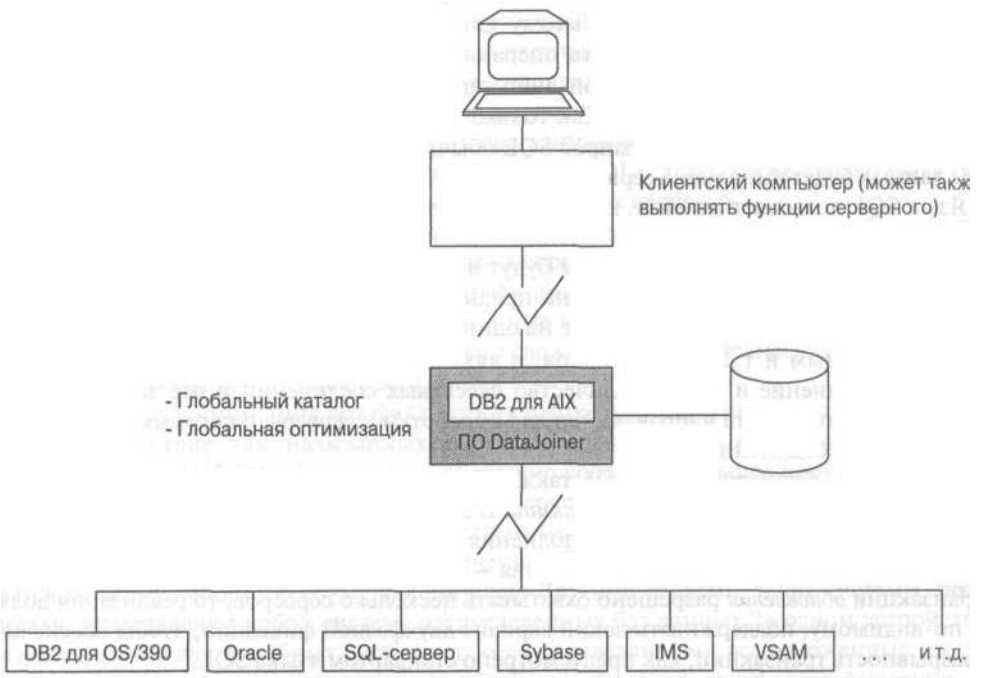


Рис. 21.8. DataJoiner — пример промежуточного программного обеспечения для доступа к данным

Заключительное слово

Очевидно, что при попытках предоставить полную независимость от СУБД возникают значительные технические сложности, даже если все участвующие СУБД являются системами SQL. Но в будущем эти усилия должны окупиться с лихвой, даже если решения не будут безукоризненными. По этой причине в настоящее время уже создано несколько промежуточных продуктов доступа к данным, и их, безусловно, будет еще больше в ближайшем будущем. Но предупреждаем, что решения в таких продуктах неизбежно будут далеки от совершенства, хотя поставщики утверждают обратное (пользователь, будь осторожен!).

21.7. СРЕДСТВА SQL

В настоящее время в языке SQL отсутствует поддержка[9] настоящих распределенных систем баз данных. Безусловно, в области обработки данных никакой поддержки и *не требуется* — основная задача распределенной базы данных, с точки зрения пользователя, состоит в том, чтобы сохранить возможности обработки данных неизменными. Тем не менее, *требуются* операции определения данных, такие как FRAGMENT, REPLICATE и т.д. [15.6]. Однако до сих пор такие операции в языке SQL отсутствуют.

С другой стороны, язык SQL поддерживает некоторые возможности создания системы "клиент/сервер", включая, в частности, операторы **CONNECT** и **DISCONNECT** для установления и разрыва соединений между клиентом и сервером. В действительности, приложения SQL *должны* выполнять операцию CONNECT для соединения с сервером, прежде чем они смогут выдать какой-либо запрос к базе данных (хотя такая операция CONNECT может быть и неявной). Как только соединение будет установлено, приложение, т.е. клиент, сможет выдать запрос SQL обычным образом, а необходимая обработка базы данных будет выполнена сервером.

Язык SQL позволяет клиенту, который подключился к одному серверу, подключиться **и** к другому серверу. После установления второго соединения первое соединение становится **пассивным.** Поэтому запросы будут выполняться вторым сервером до тех пор, пока клиент не переключится либо на предыдущий сервер с помощью другой новой операции **SET CONNECTION,** либо еще на один сервер, и тогда второе соединение также станет пассивным и т.д. Иначе говоря, в любое время у клиента имеется лишь одно **активное** соединение и любое количество **пассивных** соединений и все запросы к базе данных от этого клиента направляются для обработки к серверу, с которым поддерживается активное соединение.

Примечание. Стандарт языка SQL также позволяет (но не требует), чтобы реализация поддерживала *мультисерверные транзакции.* В этом случае клиент может переключаться с одного сервера на другой по ходу выполнения транзакции, поэтому одна часть транзакции выполняется на одном сервере, а другая — на другом. Отметим, в частности, что если в транзакции *обновления* разрешено охватывать несколько серверов, то реализация должна, по-видимому, поддерживать некий вариант двухфазной фиксации, чтобы обеспечить неразрывность транзакции, как предусмотрено стандартом языка SQL.

Наконец, каждое соединение, установленное данным клиентом (то ли активное, то ли пассивное), рано или поздно должно быть разорвано с помощью соответствующей операции DISCONNECT, хотя в простых случаях операция DISCONNECT, как и соответствующая операция CONNECT, может быть неявной.

Дополнительную информацию по данному вопросу (в частности, подробные сведения о средствах SQL, предназначенных для создания хранимых процедур) можно найти в самом стандарте языка SQL [4.23], [4.24] или в учебном пособии по этому языку [4.20].

21.8. РЕЗЮМЕ

В настоящей главе кратко рассматривались системы распределенных баз данных. В качестве стержня изложения использовались **двенадцать целей** для этих систем [21.13]. Но еще раз подчеркнем, что не все цели равнозначны во всех ситуациях. Также здесь речь шла о технических проблемах, которые возникают в областях **обработки запросов, управления каталогом, распространения обновлений, управления восстановлением** и **управления параллельностью.** Кроме того, обсуждались те вопросы, попытки решения которых должны обеспечить **независимость от СУБД;** в частности, в разделе 21.6 описывались шлюзы, промежуточное программное обеспечение для доступа к данным и интегрированные системы. Затем мы познакомились с обработкой данных в системах **"клиент/сервер",** которая может рассматриваться как важный частный случай распределенной обработки данных в целом. Наконец, в этой главе перечислялись те аспекты языка SQL, которые отвечают требованиям обработки данных в системах "клиент/сервер", а также подчеркивалось, что пользователи должны избегать **программирования на уровне записей** (т.е. операций с курсорами, как они именуются в стандарте SQL). Здесь также кратко была описана концепция **хранимых процедур** и **вызовов удаленных процедур.**

Примечание. Одной из проблем, которую мы не обсуждали вообще, является проблема физического **проектирования базы** данных для распределенных систем. На самом деле, даже если не учитывать возможность фрагментации и репликации, проблема принятия решения о том, какие должны храниться данные и на каких узлах (так называемая **проблема размещения),** считается исключительно сложной [21.31]. Фрагментация и репликация лишь еще больше усложняют этот вопрос.

Заслуживает упоминания тот факт, что на рынке программных продуктов стало заметно присутствие так называемых *массовых параллельных* компьютерных систем (см. аннотацию к [18.56]). Такие системы обычно содержат большое количество процессоров, соединенных между собой с помощью высокоскоростной шины. Каждый процессор имеет собственную основную память, собственные дисковые накопители и поддерживает собственную копию программного обеспечения СУБД, а полная база данных распределяется по всему набору дисковых накопителей. Другими словами, такие системы, по существу, представляют собой систему распределенных баз данных "в одном устройстве". Безусловно, что для подобных систем остаются справедливыми все изложенные рассуждения по таким вопросам, как стратегия обработки запросов, двухфазная фиксация, ситуация глобальной взаимоблокировки и т.д.

В заключение отметим, что двенадцать целей создания распределенных баз данных (или их подмножество, которое включает, по крайней мере, цели 4, 5, 6 и 8), рассматриваемые совместно, похоже, равносильны правилам *независимости от распределения* Кодда для реляционной СУБД [10.3]. Ниже это правило приведено для использования в дальнейшем.

- *Независимость от распределения* (Кодд). "Реляционная СУБД обладает независимостью от распределения... [которая подразумевает, что] СУБД имеет подъязык данных, который позволяет пользовательским программам и процедурам терминальной обработки оставаться логически незатронутыми в описанных ниже ситуациях.

  а) Когда распределение данных вводится впервые (если первоначально установленная СУБД оперировала только нераспределенными данными).

б) Когда данные перераспределяются (если СУБД оперирует распределенными данными)".

Наконец отметим, что (как уже упоминалось в этой главе) цели 4—6 и 9—12, т.е. все цели, в названии которых есть слово "независимость", могут рассматриваться как расширение привычного понятия независимости от данных на случай его применения для распределенной среды. При этом указанные принципы, по сути, превращаются в инструмент защиты инвестиций в приложения.

УПРАЖНЕНИЯ

21.1. Сформулируйте определения понятий независимости от расположения, независимости от фрагментации и независимости от репликации.

21.2. Почему почти все системы распределенных баз данных являются реляционными?

21.3. Какими преимуществами обладают распределенные системы? Какие им свойственны недостатки?

21.4. Объясните следующие термины:

- стратегия обновления на основе первичной копии;
- стратегия блокировки на основе первичной копии;
- ситуация глобальной взаимоблокировки;
- двухфазная фиксация;
- глобальная оптимизация.

21.5. Опишите схему присваивания имен объектам в системе R*.

21.6. Успешность реализации шлюзов типа "точка-точка" зависит (помимо многих других обстоятельств) от компенсации различий в интерфейсах между двумя используемыми СУБД. Рассмотрите любые две знакомые вам системы SQL и укажите как можно больше различий между их интерфейсами. Учитывайте как синтаксические, так и семантические различия.

21.7. Исследуйте любую доступную вам систему "клиент/сервер". Поддерживаются ли в ней явным образом операции установления соединения CONNECT и разрыва соединения DISCONNECT? Поддерживается ли в ней операция SET CONNECTION или какие-либо другие операции "с соединениями"? Поддерживаются ли в ней мультисерверные транзакции? Поддерживается ли в ней двухфазная фиксация? Какие форматы и протоколы используются для взаимодействия между клиентом и сервером? Какие разновидности сетевой среды в ней поддерживаются? Какое аппаратное обеспечение поддерживается для узлов клиентов и узлов серверов? Какие в ней поддерживаются программные платформы (операционные системы, СУБД)?

21.8. Исследуйте любую доступную вам СУБД, поддерживающую язык SQL. Поддерживаются ли в ней хранимые процедуры? Если поддерживаются, то как они создаются? Как они вызываются? На каких языках они разрабатываются? Полностью ли в них поддерживается стандарт языка SQL? Поддерживаются ли в них условное ветвление (IF-THEN-ELSE)? Поддерживаются ли циклы? Каким образом результаты возвращаются клиенту? Может ли одна хранимая процедура вызвать другую? А на другом узле? Выполняется ли хранимая процедура в составе вызова транзакции?

СПИСОК ЛИТЕРАТУРЫ

21.1. Anderson T., Breitbart Y., Korth H. F., Wool A. Replication, Consistency, and Practica lity: Are These Mutually Exclusive? // Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data. — Seattle, Wash. — June 1998.

В этой статье описаны три схемы для асинхронной репликации (называемой здесь *отложенной* — lazy), которые гарантируют неразрывность транзакций и их глобальную упорядочиваемость без использования двухфазной фиксации. Также описывается имитационное исследование их относительной производительности. Глобальная блокировка, предложенная в [21.18], используется только в первой схеме. В двух других схемах (первая из которых пессимистическая, а вторая оптимистическая) используется *граф репликации.* В статье делается вывод, что схемы с графом репликации обычно превосходят схемы с использованием блокировки "с огромным преимуществом".

21.2. Bell D., Grimson J. Distributed Database Systems. — Reading, Mass.: Addison-Wesley, 1992.

Это одно из нескольких существующих учебных пособий, посвященных теме распределенных систем (двумя другими являются [21.10] и [21.29]). Особенностью книги служит подробное изложение учебного материала на основе примера создания сети в учреждениях здравоохранения. К тому же по сравнению с двумя другими указанными публикациями, она является более практической.

21.3. Bernstein P. A. Middleware: A Model for Distributed System Services // CACM. — February 1996. - 39, № 2.

Из краткого обзора: "Классифицированы различные типы межплатформенного программного обеспечения, описаны их свойства, а также рассмотрен процесс их изменения. Предоставлена концептуальная модель для исследования в области программного обеспечения современных и будущих распределенных систем".

21.4. Bernstein P. A. et al. Query Processing in a System for Distributed Databases (SDD-1) // ACM TODS. - December 1981. - 6, № 4.

См. аннотацию к [21.32].

21.5. Bernstein P. A., Shipman D. W., Rothnie J. B. Concurrency Control in a System for Distributed Databases (SDD-1) //ACM TODS. — March 1980. - 5, № 1.

См. аннотацию к [21.32].

21.6. Bontempo C. J., Saracco C. M. Data Access Middleware: Seeking out The Middle Ground // InfoDB. - August 1995. - 9, № 4.

Полезное учебное пособие, в котором внимание акцентируется на продукте DataJoiner корпорации IBM (хотя упоминаются и другие продукты).

21.7. Breitbart Y., Garcia-Molina H., Silberschatz A. Overview of Multi-Database Transaction Management // The VLDB Journal. — October 1992. — 1, № 2.

21.8. Bright Y., Hurson A. R., Pakzad S. Automated Resolution of Semantic Heterogeneity in Multi-Databases// ACM TODS. - June 1994. - 19, № 2.

21.9. Brodie M.L. Data Management Challenges in Very Large Enterprises (выдержки из протокола совещания) // Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong. — August 2002.

Значительная часть этого совещания была посвящена вопросам информационной интеграции. Приведем цитату из протокола: "Ведущей [проблемой] является [информационная] интеграция. В проведенных недавно аналитических исследованиях сделано заключение, что свыше 40% бюджета информационных отделов выделяется на интеграцию новых и существующих систем баз данных... С такой массовой интеграцией связаны значительные сложности и затраты".

21.10. Ceri S., Pelagatti G. Distributed Databases: Principles and Systems. — New York, N.Y.: McGraw-Hill, 1984.

21.11. Cohen W. W. Integration of Heterogeneous Databases without Common Domains Using Queries Bases on Textual Similarity // Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data. — Seatle, Wash. — June 1998.

Описывается подход, который иногда называют "проблемой ненужных писем". Он позволяет определить, когда две различные текстовые строки (скажем, "AT&T Bell Labs" и "AT&T Research") ссылаются на один и тот же объект (таким образом, эта проблема относится к категории проблем определения семантических различий). Данный подход включает возможности определения *схожести* таких строк, "которые рассчитаны на использование модели векторного пространства, обычно применяемой при выборке статистических данных". По мнению авторов статьи, быстродействие этого метода значительно выше, чем быстродействие "простых методов вывода", и, кроме того, он действительно дает удивительно точные результаты.

21.12. Daniels D. et al. An Introduction to Distributed Query Compilation in R* // Distributed Data Bases (ed. H.-J. Schneider): Proc. 2nd Int. Symposium on Distributed Data Bases. - New York, N.Y.: North-Holland, 1982.

См. аннотацию к [21.37].

Date C. J. What is a Distributed Database System? // Date C. J. Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

В статье введены двенадцать целей для распределенных систем (раздел 21.3 построен в строгом соответствии с этой статьей). Как уже упоминалось в данной главе, требование *локальной автономности* не может быть достигнуто на все 100%. Поэтому определенные ситуации требуют принятия в этом отношении компромиссных решений, кратко перечисленных ниже.

- К отдельным фрагментам фрагментированной переменной отношения обычно не может быть обеспечен непосредственный доступ даже с того узла, на котором они хранятся.
- К отдельным копиям реплицируемой переменной отношения (или фрагмента) при обычных условиях не может быть обеспечен непосредственный доступ даже с того узла, на котором они хранятся.
- Пусть P является первичной копией некоторой реплицируемой переменной отношения (или фрагмента) R и пусть P хранится на узле X. Тогда каждый узел, который обращается к отношению R, зависит от узла X, даже если на этом узле хранится другая копия отношения R.
- К переменной отношения, которая фигурирует в ограничении целостности
- для многих узлов, нельзя осуществить доступ в целях обновления в локальном

контексте узла, на котором она хранится. Это возможно только в контексте распределенной базы данных, для которой задано ограничение.

- На узле, который действует как участник процесса двухфазной фиксации, должны строго выполняться решения (относительно фиксации или отката), принятые соответствующим узлом-координатором.

  См. также работу [15.6], которая является продолжением данной статьи.

21.14. Date C. J. Why Is It So Difficult to Provide a Relation Interface to IMS? // Relational Database: Selected Writings. — Reading, Mass.: Addison-Wesley, 1986.

Вопрос о том, может ли быть предоставлен реляционный интерфейс к СУБД IMS, имеет две приведенные ниже возможные интерпретации.

1. Может ли быть создана реляционная СУБД, в которой IMS используется как диспетчер памяти?
2. Может ли быть создана "оболочка" к IMS, благодаря которой существующие данные IMS будут выглядеть как реляционные данные?

Если желаемой интерпретацией является первая, то ответ, безусловно, становится положительным (несмотря на то, что имеется значительное количество средств IMS, которые, по-видимому, не могут быть использованы); а если выбрана вторая интерпретация, ответ становится отрицательным (по меньшей мере, на уровне 100%). В статье обоснован именно этот вывод.

21.15. Epstein R., Stonebraker M., Wong E. Distributed Query Processing in a Relational Database System // Proc. 1978 ACM SIGMOD Int. Conf. on Management of Data. — Austin, Tex. — May-June 1978.

См. аннотацию к [21.34].

Goldring R. A Discussion of Relational Database Replication Technology // InfoDB. — 1994.-8, № 1.

Прекрасный обзор асинхронной репликации.

21.17. Grant J., Litwin W., Roussopoulos N., Sellis T. Query Languages for Relational Multi-Databases // The VLDB Journal. — April 1993. — 2, № 2.

В статье предлагаются расширения реляционной алгебры и реляционного исчисления в отношении систем со многими базами данных. В ней обсуждаются вопросы оптимизации, а также показано, что каждое выражение реляционной алгебры для нескольких отношений имеет эквивалент в реляционном исчислении для нескольких отношений (в статье также отмечено, что "доказательство обратной теоремы представляет собой интересную научную задачу").

21.18. Gray J., Helland P., O'Neil P., Shasha D. The Dangers of Replication and a Solution // Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data. — Montreal, Canada — June 1996.

Цитата из резюме, немного перефразированная: "Процесс репликации с помощью транзакций обновления, выполняемых повсеместно, в любое время и любым способом по мере возрастания рабочей нагрузки становится все более неустойчивым... Предложен новый алгоритм, предусматривающий мобильные (отсоединенные) приложения для предварительных транзакций обновления, которые затем применяются и к главной копии".

21.19. Gupta R., Haritsa J., Ramamritham K. Revisiting Commit Processing in Distributed Database Systems // Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data. - Tucson, Ariz. — May 1997.

Предложен новый протокол распределенной фиксации, называемый OPT, который можно легко реализовать и использовать вместе с традиционными протоколами и который "обеспечивает наиболее высокую эффективность выполнения транзакции при различных рабочих нагрузках и системных конфигурациях".

21.20. Hackathorn R. D. Interoperability: DRDA or RDA? // InfoDB. — 1991. — 6, № 2.

21.21. Hammar M., Shipman D. Reliability Mechanism for SDD-1; A System for Distributed Databases // ACM TODS. - December 1980. - 5, № 4.

См. аннотацию к [21.32].

21.22. **IBM** Corporation. Distributed Relational Database Architecture Reference. — **IBM** Form№SC26-4651.

В стандарте DRDA, разработанном фирмой IBM, заданы четыре приведенных ниже уровня функциональности распределенной базы данных.

1. Удаленный запрос.
2. Удаленная единица работы.
3. Распределенная единица работы.
4. Распределенный запрос.

Поскольку эти термины фактически стали стандартами программного обеспечения, по крайней мере, некоторой его части, здесь следует объяснить их немного подробнее.

Примечание. Термины *запрос* и *единица работы,* применяемые в компании IBM, являются, соответственно, аналогами терминов *предложение* и *транзакция,* которые используются в стандарте *SQL.*

- **Удаленный запрос.** Означает, что приложение на узле x может отослать для выполнения отдельное предложение SQL некоторому удаленному узлу Y. Этот запрос полностью выполняется и *фиксируется* (или откатывается) на узле Y. Исходное приложение на узле X может впоследствии отослать другой запрос узлу Y (или, возможно, третьему узлу z) независимо от того, был ли пер вый запрос выполнен успешно или окончился неудачей.
- **Удаленная единица работы** (сокращенно RUW — Remote Unit of Work). Означает, что приложение на одном узле X может отсылать на некоторый удаленный узел Y для выполнения все запросы к базе данных в составе некоторой задан ной *единицы работы* (т.е. транзакции). Таким образом, обработка транзакции для этой базы данных выполняется целиком на удаленном узле Y. Однако ре шение о том, будет ли данная транзакция зафиксирована или отменена, при нимается на локальном узлех.

Примечание. Удаленная единица работы, по сути, представляет собой пример обработки данных по принципу "клиент/сервер" с единственным сервером.

- **Распределенная единица работы** (сокращенно DUW — Distributed Unit of Work). Означает, что приложение на одном узле X может отсылать к одному или нескольким удаленным узлам Y, Z ... для выполнения некоторые запросы или

все запросы к базе данных в составе некоторой заданной *единицы работы* (т.е. транзакции). Таким образом, обработка транзакции для этой базы данных в общем случае распределяется по нескольким узлам. Причем каждый индивидуальный запрос может выполняться полностью на отдельном узле, а разные запросы — на нескольких различных узлах. Однако локальный узел X все еще остается координатором, т.е. решение о том, будет ли данная транзакция зафиксирована или отменена, принимается на этом узле.

Примечание. Распределенная единица работы фактически представляет собой пример обработки данных по принципу "клиент/сервер" с несколькими серверами.

- **Распределенный запрос.** Это единственный из всех четырех уровней, который наиболее близок к понятию истинной поддержки распределенной базы данных. Понятие *распределенный запрос* означает то же, что *к распределенная единица работы, наряду* с разрешением на выполнение индивидуальных запросов (предложений SQL) к базе данных с охватом нескольких узлов. Например, согласно запросу, поступившему от узлах, может потребоваться выполнить соединение или объединение отношения на узле Y и отношения на узле Z. Отметим, что только на этом уровне о системе можно сказать, что она обладает истинной независимостью от расположения. Во всех трех предыдущих случаях пользователь должен был иметь сведения о физическом расположении данных.

21.23. International Organization for Standardization (ISO). Information Processing Systems, Open Systems Interconnection, Remote Data Access Part 1: Generic Model, Service, and Protocol. Документ ISO DIS 9579-1. - March 1990.

21.24. International Organization for Standardization (ISO). Information Processing Systems, Open Systems Interconnection, Remote Data Access Part 2: SQL Specialization. Доку мент ISO DIS 9579-2. - February 1990.

21.25. Kossmann D. The State of the Art in Distributed Query Processing // ACM Comp. Surv. — December 2000. — 32, No. 4.

21.26. Lindsay B. G. et al. Notes on Distributed Databases // IBM Research Report RJ2571. — July 1979.

Эта статья, написанная некоторыми членами команды разработчиков системы R*, которые работали над ней с самого начала, разделена на приведенные.ниже пять глав.

1. Реплицируемые данные.
2. Права доступа и представления.
3. Введение в управление распределенными транзакциями.
4. Средства восстановления.
5. Инициирование, миграция и завершение выполнения транзакции.

В главе 1 обсуждается проблема распространения обновлений. Глава 2, за исключением нескольких замечаний в самом ее конце, почти полностью посвящена вопросам предоставления прав доступа в нераспределенной системе (в стиле системы R). В главе 3 очень кратко рассматриваются процедуры инициирования и завершения транзакций, организации параллельной работы и восстановления. Глава 4 посвящена теме восстановления, опять же, в нераспределенной системе.

Наконец, в главе 5 более подробно обсуждается управление распределенными транзакциями, в частности, дается очень подробное описание протокола двухфазной фиксации.

21.27. Mohan C, Lindsay B. G. Efficient Commit Protocols for the Tree of Processes Model of Distributed Transaction // Proc. 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing. — 1983.
См. аннотацию к [21.37].

21.28. Newman S., Gray J. Which Way to Remote SQL? // DBP&D. — December 1991. — 4, № 12.

21.29. Oszu M. T., Valduriez P. Principles of Distributed Database Systems (2nd edition). — Englewood Cliffs, N.J.: Prentice-Hall, 1999.

21.30. Rennhackkamp M. Mobile Database Replication // DBMS. — October 1997. — 10, № 11.

Благодаря созданию недорогих и портативных компьютеров, а также распространению беспроводных коммуникаций стало возможным появление систем распределенных баз данных нового типа, которые имеют свои специфические преимущества и недостатки. В частности, данные в таких системах могут быть реплицированы буквально на тысячах *узлов,* но эти узлы мобильны, часто отключаются, их операционные характеристики очень отличаются от характеристик обычных узлов (например, в стоимость соединения входит и использование аккумуляторных батарей, и время соединения) и т.д. Исследования таких систем начаты сравнительно недавно (см. [21.1], [21.18]). В этой краткой статье освещены некоторые основные понятия и проблемы.

21.31. Rothnie J. B., Goodman N. A Survay of Research and Development in Distributed Database Management // Proc. 3rd Int. Conf. on Very Large Data Bases. — Tokyo, Japan — October 1977.

Один из наиболее ранних, но очень полезных обзоров приведенных ниже тем.

1. Синхронизация транзакций обновления.
2. Обработка распределенного запроса.
3. Организация работы в условиях отказа компонентов.
4. Управление каталогом.
5. Проектирование базы данных.

Последняя тема относится к проблеме *физического* проектирования, которая в разделе 21.8 называется проблемой *размещения.*

21.32. Rothnie J.B., Jr. et al. Introduction to a System for Distributed Databases (SDD-1) // ACM TODS. - March 1980. - 5, № 1.

Работы [21.4], [21.5], [21.21], [21.32] и [21.32] связаны с ранним вариантом распределенной системы SDD-1, которая работала на нескольких компьютерах PDP-10 фирмы DEC, подключенных к сети ARPAnet (см. раздел 27.2 главы 27). В этой системе обеспечивалась полная независимость от размещения, фрагментации и репликации. Ниже приводится несколько замечаний по поводу некоторых ее особенностей.

Обработка запросов. Оптимизатор запросов в системе SDD-1 (см. [21.4]) в значительной мере использует оператор *полусоединения,* который упоминался в разделе 7.8

главы 7. Преимущество использования полусоединений при обработке распределенных запросов заключается в том, что они могут привести к сокращению объема данных, пересылаемых по сети. Предположим, например, что переменная отношения поставщиков S хранится на узле A, переменная отношения поставок SP — на узле в, а запрос формулируется следующим образом: "Определите результат соединения отношений поставщиков и поставок". Вместо того чтобы пересылать все кортежи переменной отношения S, скажем, на узел в, можно выполнить перечисленные ниже действия.

- Вычислить проекцию TEMPI переменной отношения SP по атрибуту S# на узле в.
- Переслать проекцию TEMPI на узел A.
- Вычислить полусоединение TEMP2 проекции TEMPI И отношения S по атрибу ту S# на узле A.
- Переслать полусоединение TEMP2 на узел в.
- Вычислить полусоединение проекции TEMP2 и отношения SP по атрибуту s# на узле в. В результате будет получен ответ на сформулированный выше запрос.

Очевидно, что эта процедура приведет к общему сокращению объема пересылки данных по сети тогда и только тогда, когда выполняется следующее условие. size (TEMPI) + size (TEMP2) < size (S) Здесь функция size() возвращает кардинальность отношения, указанного в качестве параметра, умноженную на длину отдельного кортежа (скажем, в битах). Это позволяет оптимизатору точно оценить размер промежуточных результатов типа TEMP1 и TEMP2.

Распространение обновлений. Алгоритмом распространения обновления в системе SDD-1 предусматривается "немедленное распространение" (понятие первичной копии в ней не вводится). **Управление параллельностью.** Управление параллельностью основано не на механизме блокировок, а на методе синхронизации с использованием **временных отметок.** Этот метод представляет собой попытку избежать дополнительных издержек на отправку сообщений о блокировке, но за счет того, что степень распараллеливания становится не столь значительной! Дополнительные подробности в этой книге не приводятся, хотя в аннотации к [16.3] очень кратко описана основная идея. Более подробную информацию можно получить в [21.5]. **Управление восстановлением.** Восстановление основано на протоколе четырехфаз-ной фиксации, который был выбран для того, чтобы в случае выхода из строя координирующего узла обеспечить большую гибкость всего процесса, чем при использовании обычного протокола двухфазной фиксации. Но при такой организации управления, к сожалению, процесс в целом значительно усложняется. Дополнительные подробности в настоящей книге также не приводятся. Каталог. Каталог управляется так же, как и обычные пользовательские данные, т.е. он может быть произвольно фрагментирован, а сами фрагменты могут реплицироваться и распространяться произвольным образом, как и любые другие данные. Преимущества такого подхода очевидны, а недостатки, к сожалению, состоят в том, что система не обладает никакими априорными сведениями о расположении произвольной части каталога, поэтому для хранения таких сведений необходимо

поддерживать каталог более высокого уровня (определитель каталога), который реплицируется полностью, т.е. на каждом узле хранится его копия.

21.33. Selinger P. C, Adiba M E. Access Path Selection in Distributed Data Base Management Systems // S.M. Deen and P. Hammersley. Proc. Int. Conf. On Data Bases. — Aberdeen, Scotland — July 1980; London, England: Heyden and Sons Ltd., 1980.

См. аннотацию к [21.37].

21.34. Stonebraker M. R., Neuhold E. J. Distributed Data Base Version of Ingres // Proc. 2nd Berkley Conf. On Distributed Data Management and Computer Networks. — Lowrence Berkley Laboratory, May 1977.

Работы [21.15], [21.34] и [21.35] связаны с прототипом распределенной системы Distributed Ingres. Распределенная система Distributed Ingres состоит из нескольких копий программного обеспечения University Ingres, установленных и запущенных на нескольких компьютерах PDP-11 фирмы DEC. Она поддерживает независимость от расположения (как в системах SDD-1 и R*); она также поддерживает фрагментацию данных (с применением операций сокращения, но не проекции), независимую от фрагментации, и репликацию данных для этих фрагментов, независимую от репликации. В отличие от систем SDD-1 и R*, в системе Distributed Ingres не предполагается, что передача данных по сети обязательно должна быть медленной; наоборот, эта система спроектирована как для медленных сетей дальней связи, так и для сравнительно быстрых локальных сетей, а оптимизатор запросов учитывает различие между двумя этими вариантами. В алгоритме оптимизации запросов применяется расширение стратегии декомпозиции системы Ingres, описанной в главе 18 данной книги. Более подробно он рассматривается в [21.15]. В системе Distributed Ingres обеспечиваются два алгоритма распространения обновления: *эффективный* алгоритм, согласно которому обновляется первичная копия и управление возвращается данной транзакции (причем распространение обновлений выполняется параллельно с помощью набора подчиненных процессов), и *надежный* алгоритм, согласно которому все копии обновляются одновременно [21.35]. В обоих случаях управление параллельностью основано на механизме блокировки, а управление восстановлением — на протоколе двухфазной фиксации (с некоторыми усовершенствованиями).

Что же касается каталога, то в системе Distributed Ingres используется сочетание полной репликации для некоторых частей каталога, в основном тех, которые содержат логическое описание видимых для пользователя базовых переменных отношения, с описанием того, как фрагментированы эти переменные отношения. К тому же с полной репликацией комбинируются элементы локальных каталогов других частей, например, описывающих локальные индексы, локальную статистику базы данных (используемую оптимизатором), а также ограничения защиты и целостности.

21.35. Stonebraker M. R. Concurency Control and Consistency of Multiple Copies in Distributed Ingres // IEEE Transactions on Software Engineering. — May 1979. — 5, №3.

См. аннотацию к [21.34].

21.36. Wen-Syan Li and Clifton C. Semantic Integration in Heterogeneous Databases Using Neural Networks // Proc. 20th Int. Conf. on Very Large Data Bases. — Santiago, Chile. - September 1994.

21.37. Williams R. et al. R*: An Overview of the Architecture // P. Scheuermann. Improving Database Usability and Responsiveness. — New York, N.Y.: Academy Press, 1982. (Эта работа также опубликована в виде отчета: **IBM** Research RJ3325. — December 1981.)

В работах [21.12], [21.27], [21.33] и [21.37] обсуждается система R*, которая является распределенной версией первоначального прототипа системы System R. В системе R* обеспечивается независимость от расположения, но не поддерживаются фрагментация и репликация, а значит, не обеспечивается независимость от фрагментации и репликации. По той же причине для данной системы не возникает вопрос о распространении обновления. Управление параллельностью основано на механизме блокировки (обратите внимание на то, что существует только одна копия любого блокируемого объекта; при этом вопрос о первичной копии также не возникает). Управление восстановлением основано на протоколе двухфазной фиксации, но с некоторыми усовершенствованиями.

21.38. Yan L.L., Miller R.J., Haas L.M., Fagin R. Data-Driven Understanding and Refinement of Schema Mappings // Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. — May 2001.

ГЛАВА 22

Поддержка принятия решений

22.1. ВВЕДЕНИЕ

Примечание. Первый вариант этой главы был написан Дэвидом Макговереном (David McGoveren) из компании Alternative Technologies.

Системы **поддержки принятия решений** — это системы, которые служат для анализа деловой информации. Их назначение — помочь руководителям "выявить тенденции, определить проблемы и предложить ... разумное решение" [22.9]. Подобные системы создаются на основе таких теорий, как исследование операций, теория поведения и научная теория управления, а также с помощью методов статистической обработки. Первые теоретические работы в этой области появились в конце 1940-х и начале 1950-х годов, т.е. задолго до того, как компьютеры приобрели широкое распространение. Основной идеей было и по-прежнему остается накопление производственных *операционных* данных (см. главу 1) и приведение их к виду, в котором они могли бы использоваться для анализа хода деловых процессов и корректировки деловой активности в целях направления ее в разумное русло. По очевидным причинам степень преобразования данных на первых порах была почти минимальной — обычно все сводилось к составлению простых итоговых отчетов.

В конце 1960-х и начале 1970-х годов исследователи Гарвардского университета и Массачусетского технологического института начали пропагандировать идею использования компьютеров в процессе выработки решений [22.26]. Сначала такое использование ограничивалось в основном автоматизацией генерации отчетов, хотя иногда предусматривались и элементарные аналитические возможности [22.6]-[22.8]. Первые компьютерные системы сначала назывались **автоматизированными системами управления,** а позже — **системами управления информацией.** Но автор предпочитает современный термин — *системы поддержки принятия решений* (decision support system), поскольку "системами управления информацией" могут или должны считаться *все* информационные системы, включая, например, систему оперативной обработки транзакций (On-Line Transaction Processing — OLTP), поскольку в конечном счете все они используются в деловых процессах и влияют на управление ими. Поэтому в дальнейшем в этой главе будет применяться современная терминология.

В 1970-х годах велись также разработки нескольких **языков запросов,** и на их основе было создано несколько заказных (внутрифирменных) систем поддержки принятия решений. Они реализовывались с применением средств генерации отчетов, таких как язык RPG, или систем поиска данных, таких как Focus, Datatrieve и NOMAD. Эти системы были первыми из числа тех, которые позволяли соответствующим образом подготовленным конечным пользователям получать непосредственный доступ к банкам данных на компьютере. Иначе говоря, они позволяли пользователям формулировать производственные запросы к банкам данных и выполнять эти запросы, не ожидая помощи от информационно-технологического подразделения.

Естественно, то, что мы теперь называем банком данных (data store), в то время чаще всего представляло собой просто набор файлов — производственные данные хранились или в отдельных файлах, или в нереляционных базах данных (реляционные системы еще только начинали разрабатываться). И даже в последнем случае данные извлекались из базы данных и копировались в файлы, прежде чем они могли быть обработаны системой поддержки принятия решений. Так продолжалось почти до начала 1980-х годов, пока для систем поддержки принятия решений вместо простых файлов не стали использоваться реляционные базы данных. На самом деле, поддержка принятия решений, обработка произвольных *(ad hoc)* запросов и выдача отчетов были первыми практическими задачами, в которых использовалась реляционная технология.

Хотя в настоящее время продукты SQL получили широкое распространение, идея **процесса** извлечения, т.е. копирования данных из одной операционной среды в какую-либо другую среду для последующей обработки, не утратила своей значимости. Скопированные данные пользователи могут обрабатывать каким угодно способом, без вмешательства в операционную среду. И, разумеется, очень часто выполнение такой выборки данных обусловлено необходимостью поддержки принятия решений.

Из вышесказанного должно быть ясно, что поддержка принятия решений не является частью самой технологии баз данных. Это скорее область *применения* данной технологии (хотя и очень важная). Точнее, существует несколько подобных областей применения, отдельных, но взаимосвязанных: *хранилища данных* (data warehouse), *магазины данных* (data mart), *банки оперативных данных* (operational data store), *оперативная аналитическая обработка* (online analytical processing — OLAP), *многомерные базы данных* и *разработка данных.* Все эти технологии поддержки принятия решений будут рассмотрены в следующих разделах. Но сразу же отметим — единственное, что объединяет упомянутые технологии, —

это то, что они редко следуют соответствующим логическим принципам проектирования. К сожалению, на практике системы поддержки принятия решений не базируются на строгом научном фундаменте, что было бы весьма желательно, и часто становятся итогом разработки, которая продиктована исключительно текущими потребностями. В частности, наблюдается тенденция к смещению акцентов в сторону физических, а не логических соображений (в действительности границы между физическими и логическими аспектами в области систем поддержки принятия решений часто очень расплывчаты). В основном именно поэтому в примерах данной главы будет использоваться язык SQL, а не Tutorial D. Также в ней будет применяться "менее строгая" терминология языка SQL *{строки, столбцы* и *таблицы* вместо *кортежей, атрибутов, значений отношения* и *переменных отношения).* Мы будем также использовать термины *логическая схема* и *физическая схема* вместо терминов *концептуальная* схема и *внутренняя* схема, соответственно, которые были определены в главе 2.

План этой главы таков. В разделе 22.2 рассматриваются тенденции, связанные с практикой проектирования приложений поддержки принятия решений, которые мы считаем не совсем правильными. В разделе 22.3 описан наш собственный подход к трактовке тех же тенденций. Затем в разделе 22.4 обсуждаются вопросы подготовки данных (т.е. процесс получения оперативных данных в том виде, в котором они могут быть пригодными для выполнения задач поддержки принятия решений); здесь же кратко рассказывается о банках оперативных данных. В разделе 22.5 рассматриваются хранилища данных, магазины данных и *многомерные схемы.* В разделе 22.6 обсуждаются оперативная аналитическая обработка (OLAP) и многомерные базы данных. Раздел 22.7 посвящен разработке данных, а в разделе 22.8 описаны соответствующие средства SQL. Наконец, в разделе 22.9 представлено резюме.

22.2. НЕКОТОРЫЕ ОСОБЕННОСТИ ТЕХНОЛОГИИ ПОДДЕРЖКИ ПРИНЯТИЯ РЕШЕНИЙ

Базы данных поддержки принятия решений обладают особыми характеристиками, и главная из них состоит в том, что такие базы данных в основном (хотя и не всегда) предназначены только для чтения. Модификация содержимого базы данных обычно ограничивается периодическими операциями *загрузки* или *обновления.* К тому же среди этих операций преобладает операция INSERT, операция DELETE выполняется крайне редко, а операция UPDATE не выполняется почти никогда.

Примечание. Иногда операции обновления выполняются в некоторых вспомогательных рабочих таблицах, но в обычных процессах поддержки принятия решений как таковых операция обновления самой базы данных почти никогда не используется.

Приведенные ниже характеристики базы данных поддержки принятия решений также заслуживают внимания (мы еще возвратимся к этим характеристикам в разделе 22.3 и конкретизируем их). Отметим, что первые три из них — логические, а три оставшиеся — физические.

- Столбцы чаще всего используются в сочетаниях.
- Поддержкой целостности данных в общем случае никто не занимается (при этом подразумевается, что данные должны быть правильными, поскольку после загруз ки они не корректируются).

- Ключи часто содержат временной компонент (безусловно, поддержка хронологиче ски упорядоченных данных была бы чрезвычайно полезной, как описано в главе 23, но предоставляется редко).
- Базы данных обычно имеют большой объем, особенно в том случае (как это чаще всего бывает), когда данные деловых транзакций[1] накапливаются в течение доста точно продолжительного времени.
- В базе данных, как правило, широко применяется индексация.
- База данных часто отличается различного рода *контролируемой избыточностью.*

Запросы поддержки принятия решений также имеют характерные особенности и, в частности, обычно они являются довольно *сложными.* Ниже указано, в чем именно заключается сложность составляемых запросов.

- *Сложность логических выражений.* Запросы поддержки принятия решений часто включают сложные выражения в конструкции WHERE. ЭТИ выражения нелегко формулировать, в них непросто разбираться, а в системе возникают сложности при их эффективной реализации. Широко распространенная проблема связана с тем, что в запросах часто требуется учитывать данные о времени (в качестве примера можно указать запросы на получение строк с максимальным значением временной отметки в заданный период времени). Если в запросах используются какие-нибудь соединения, то они очень быстро становятся чрезвычайно сложными, особенно по той причине, что соответствующая хронологическая поддержка чаще всего не предусмотрена. И вполне естественно, что в конечном итоге во всех этих случаях наблюдается низкая производительность.

  Сложность соединений. Для запросов в системах поддержки принятия решений часто требуется доступ ко многим видам информации, вследствие чего и в правильно спроектированной, т.е. полностью нормализованной, базе данных при выполнении этих запросов обычно осуществляется множество соединений. К сожалению, в обычно применяемых технологических решениях по выполнению операций соединения никогда не учитывалась необходимость поддержки все более сложных запросов в системах поддержки принятия решений[2]. Поэтому проектировщики часто стремятся *денормализовать* базу данных с помощью *предварительного*

---

[1] Здесь и далее в данной главе мы будем различать деловые транзакции (например, по сбыту продукции) и транзакции в том смысле, который подразумевается в части IV настоящей книги. При этом для обозначения деловых транзакций всегда будет использоваться соответствующее уточнение, кроме тех случаев, когда тип этих транзакций очевиден из контекста.

Автор этой главы (Макговерен), работавший над одной из первых версий системы поддержки решений в 1981 году, заметил, что для соединения трех таблиц даже средних размеров вполне может потребоваться много часов, а соединение от четырех до шести таблиц вообще становится слишком дорогостоящим. Но при использовании современных компьютеров соединение от шести до десяти очень больших таблиц — обычное дело, и подобная возможность, как правило, предусмотрена также технологией баз данных. Однако все еще нельзя исключить вероятность такой ситуации, что будет сформирован запрос, при выполнении которого потребуется соединить такое большое количество таблиц, какое не совсем успешно поддерживается технологией (и указанные ситуации встречаются довольно часто). Попытка выполнить запрос на соединение больше двенадцати таблиц может привести к непредсказуемому результату, но необходимость выполнения подобных запросов встречается довольно часто! *Примечание.* Описание возможных решений этой проблемы приведено в приложении А.

соединения определенных таблиц. Однако, как уже отмечалось в главе 13, этот подход редко приводит к успеху, поскольку проектировщики в таких случаях сталкиваются со многими проблемами, которые требуется решать заблаговременно. И кроме того, попытки избежать явного использования соединений на этапе прогона могут привести к неэффективному применению реляционных операций, поскольку при этом происходит выборка больших объемов данных, которые, по сути, не требуются, а обработка соединений осуществляется в приложении, а не в СУБД.

- *Сложность функций.* В запросах систем поддержки принятия решений часто ис пользуются статистические и другие математические функции. Но до последнего времени их поддерживали лишь немногие программные продукты (однако ситуа ция в этой области немного улучшилась). Поэтому часто возникает необходимость в разбиении запроса на последовательность меньших запросов, которые затем вы полняются, чередуясь с пользовательскими процедурами, в которых вычисляются требуемые функции. Но, к сожалению, при использовании такого подхода может потребоваться извлечение больших объемов данных, не говоря уже о том, что сам запрос становится значительно сложнее как для написания, так и для понимания.
- *Аналитическая сложность.* Деловые запросы редко можно сформулировать в виде одного простого запроса. Чрезмерно сложные запросы представляют значитель ные трудности не только для пользователей. Ограничения, свойственные кон кретной реализации языка SQL, могут не позволить системе успешно выполнять обработку подобных запросов. Одним из путей упрощения таких запросов являет ся их разбиение на ряд меньших запросов с сохранением результатов во вспомога тельных таблицах.

Указанные выше особенности как баз данных систем поддержки принятия решений, так и выполняемых в них запросов являются причиной того, что акцент в этом случае переносится на *проектирование с точки зрения повышения производительности,* особенно — производительности процессов пакетного добавления данных и произвольного извлечения данных. Но с точки зрения автора (которая подробно излагается в следующем разделе), такое состояние дел должно учитываться на этапе не логического, а физического проектирования. К сожалению, как уже отмечалось в разделе 22.1, разработчики и пользователи систем поддержки принятия решений часто допускают ошибку, не различая логические и физические аспекты проектирования[3], а фактически они часто совсем отказываются от создания логического проекта системы. Вследствие этого, столкнувшись с перечисленными выше сложностями, они оказываются неподготовленными к ним, и это чаще всего приводит к непреодолимым трудностям при попытках достичь компромисса между правильностью, удобством сопровождения, производительностью, возможностью расширения, эффективностью и практичностью.

---

[3] Указанную ошибку особенно часто допускают специалисты по хранилищам данных и оперативной аналитической обработке (OLAP), которые утверждают, что реляционный проект просто "непригоден" для систем поддержки принятия решений, а реляционная модель неспособна обеспечить представление данных, поэтому ее необходимо обойти. Такие доводы почти всегда обусловлены неспособностью отличить реляционную модель от ее реализации.

22.3. ПРОЕКТИРОВАНИЕ БАЗЫ ДАННЫХ ДЛЯ ПОДДЕРЖКИ ПРИНЯТИЯ РЕШЕНИЙ

: Как было указано ранее, в частности, во введении к части III, по мнению автора, проект базы данных всегда должен выполняться на двух уровнях: логическом и физическом, как описано ниже.

1. Прежде всего, должен быть выполнен логический проект. На этом уровне основное внимание уделяется *правильности реляционных определений.* Таблицы должны представлять правильные отношения, гарантируя таким образом, что реляционные операции будут выполняться так, как им надлежит выполняться, и не будет получено никаких непредвиденных результатов. Затем должны быть описаны типы (домены), на их основе определены столбцы и указаны зависимости между столбцами (в том числе функциональные и другие зависимости). На основании этих данных может быть выполнена нормализация и определены дополнительные ограничения целостности.
2. После этого создается физический проект, который должен быть производным от логического проекта. На этом уровне, безусловно, основное внимание уделяется *эффективности хранения данных и производительности.* В принципе, допустимо любое физическое расположение данных, при условии, что между логической и физической схемами вводится сохраняющее информацию преобразование, которое может быть выражено с помощью реляционной алгебры [2.5]. Отметим, в частности, что из наличия такого преобразования следует, что существуют реляционные представления физической схемы, которые отображают ее подобно логической схеме, и наоборот.

Безусловно, логическая схема может быть впоследствии изменена, например, в целях включения новых видов данных, а также новых или вновь обнаруженных зависимостей. Внесенные изменения, естественно, потребуют также соответствующих изменений в физической схеме. Но этот вопрос в данной главе не рассматривается. Здесь нас больше интересует то, что физическая схема может изменяться, в то время как логическая схема будет оставаться прежней. Например, предположим, что операция соединения таблиц SP (Поставки) и P (Детали) в значительной степени применяется намного чаще по сравнению с другими операциями доступа к данным. Тогда может возникнуть стремление выполнить *предварительное соединение* таблиц SP и P на физическом уровне, чтобы сократить тем самым количество операций ввода-вывода и расходы на соединение этих таблиц. Но *логическая схема должна оставаться неизменной,* если мы хотим сохранить физическую независимость от данных. (Безусловно, оптимизатор запросов должен иметь информацию о существовании хранимого *предварительного соединения* и действовать соответствующим образом, чтобы достичь желаемого повышения производительности.) Кроме того, если модель доступа в дальнейшем изменится таким образом, что станет преобладать доступ к отдельным таблицам, а не к соединению, должна существовать возможность вновь изменить физическую схему, чтобы физически разъединить таблицы SP и P, опять-таки, без какого-либо воздействия на логический уровень.

Из сказанного выше должно быть ясно, что проблема обеспечения независимости от физических данных — это в основном проблема поддержки представлений (исключая проблему обновления фрагментов, рассматриваемую в главе 21, которая проявляется на

другом участке общей архитектуры системы). В частности, необходимо иметь возможность обновлять такие представления. А именно, если базовые таблицы рассматриваются на логическом уровне как представления, а хранимые версии этих *представлений* на физическом уровне рассматриваются как таблицы, то должна применяться такая физическая схема, чтобы СУБД позволяла осуществлять операции обновления этих *представлений* в терминах указанных *базовых таблиц.*

Итак, как показано в главе 9, теоретически все представления являются обновляемыми. Поэтому, с теоретической точки зрения, если физическая схема разработана на основе логической схемы по принципу, описанному выше, то может быть достигнута максимальная физическая независимость от данных. Любое обновление, выраженное в терминах логической схемы, будет автоматически преобразовано в обновление, выраженное в терминах физической схемы, и наоборот, а сами изменения физической схемы не будут требовать изменений в логической схеме. Но, к сожалению, современные продукты SQL не поддерживают в необходимой мере обновления представлений. Поэтому набор допустимых физических схем в этих продуктах очень ограничен (причем совершенно излишне).

Примечание. На практике указанный выше подход может быть осуществлен путем моделирования подходящего механизма обновления представления средствами хранимых процедур, триггерных процедур, промежуточного программного обеспечения или различных их сочетаний. Однако обсуждение этих методов выходит за рамки данной главы.

Логическое проектирование

Правила логического проектирования не зависят от предполагаемого использования базы данных. То же самое относится и ко всем без исключения видам приложений. Следовательно, в частности, не должно иметь значения, какими являются эти приложения — оперативными приложениями (OLTP) или приложениями поддержки принятия решений. В любом случае, должна использоваться одна и та же процедура проектирования. Поэтому еще раз проанализируем три *логические* характеристики баз данных поддержки принятия решений, определенные в начале раздела 22.2, а также рассмотрим, насколько они важны с точки зрения логического проектирования.

- *Использование сочетаний столбцов и уменьшение количества зависимостей*
  В запросах поддержки принятия решений, а также в обновлениях, если они применимы, сочетания столбцов часто рассматриваются как единое целое. Имеется в виду, что к столбцам, входящим в состав подобных сочетаний, никогда не приходится обращаться по отдельности (наглядным примером может служить адрес, ADDRESS). Условимся называть такие сочетания столбцов *составными столбцами.* Но с логической точки зрения подобные составные столбцы ведут себя так, как будто на самом деле они *не* составные. Предположим, что CC — это составной столбец, а c — какой-либо иной столбец в той же таблице. Тогда зависимости между столбцом C и компонентом (компонентами) столбца CC сводятся к зависимостям между столбцом c и составным столбцом CC как единым целым. Более того, зависимости, включающие компоненты составного столбца CC и не включающие никакие другие столбцы, не имеют смысла и могут просто игнорироваться. В результате общее число зависимостей сокращается и логический проект становится проще, с меньшим количеством столбцов и, возможно, даже с меньшим числом таблиц.

- *Общие ограничения целостности*

  Как уже было описано выше, базы данных поддержки принятия решений в основном предназначены только для чтения, и ограничения целостности проверяются только при загрузке (или обновлении) базы данных. Поэтому часто полагают, что нет никакого смысла объявлять ограничения целостности в их логической схеме. Но этот довод звучит неубедительно. Если база данных действительно предназначена только для чтения, такие ограничения на самом деле не могут быть нарушены. Но нельзя недооценивать *семантическое значение* этих ограничений. Как уже отмечалось в главе 9, ограничения служат для определения формального смысла таблиц и формального смысла всей базы данных в целом. Поэтому определение ограничений предоставляет способ передачи пользователям сведений о смысле данных, а это помогает им решать задачи формулирования запросов. Кроме того, объявление ограничений может предоставлять крайне важную информацию для оптимизатора (см. обсуждение *семантической оптимизации* в главе 18).

  Примечание. В некоторых программных продуктах объявление определенных ограничений приводит к автоматическому созданию соответствующих индексов и/или других механизмов принудительного выбора путей доступа, что может существенно увеличить стоимость операций загрузки и пополнения базы данных. В свою очередь, для проектировщиков это может послужить поводом для отказа от объяапений ограничений. И снова повторим, что эта проблема возникает из-за путаницы между логическими и физическими аспектами проектирования. Должна существовать возможность определить ограничения целостности декларативно, на *логическом* уровне, после чего отдельно определить соответствующие механизмы принудительного выбора путей доступа на *физическом* уровне. К сожалению, современные программные продукты не позволяют должным образом провести различия между логическим и физическим уровнями (более того, разработчики этих продуктов редко вообще признают семантическое значение ограничений).

- *Временные ключи*

  Оперативные базы данных обычно содержат только текущие данные. Базы данных поддержки принятия решений, наоборот, обычно содержат архивы исторически накопленных данных и поэтому большая часть данных или даже все данные включают *временной штамп,* или *временную отметку.* Ключи в таких базах данных часто содержат столбцы с временными отметками. Например, вновь обратимся к базе данных поставщиков и деталей. Предположим, что необходимо расширить ее таким образом, чтобы для каждой поставки был показан конкретный месяц (от 1 до 12), в который производилась поставка. В этом случае таблица поставок SP может выглядеть, как показано на рис. 22.1. Обратите внимание на то, что дополнительный столбец MID (идентификатор месяца) входит в состав ключа расширенной версии таблицы SP. Следует также отметить, что теперь запросы к данным из таблицы SP необходимо формулировать очень внимательно для того, чтобы доступ осуществлялся именно к тем данным, которые требуются, не больше и не меньше. Эти вопросы уже кратко рассматривались в разделе 22.2, а в главе 23 они описаны более подробно.

  Примечание. В результате добавления столбцов с временными отметками может возникнуть необходимость провести корректировку проекта базы данных. Предположим, например (хотя это и несколько надуманно), что количество деталей

в каждой поставке зависит от того месяца, когда производилась эта поставка (на рис. 22.1 данные соответствуют этому ограничению). Тогда пересмотренная версия таблицы SP будет удовлетворять функциональной зависимости MID -> QTY и поэтому не будет находиться в пятой нормальной форме и даже в третьей нормальной форме. Таким образом, возникает необходимость провести ее дальнейшую нормализацию, как показано на рис. 22.2. К сожалению, проектировщики баз данных поддержки принятия решений редко задумываются над тем, что база данных должна учитывать такие *зависимости.* Дополнительные сведения об этом также приведены в главе 23.

S#	P#	MID	QTY
S1	P1	3	300
S1	P1	5	100
S1	P2	1	200
S1	P3	7	400
S1	P4	1	200
S1	P5	5	100
S1	P6	4	100
S2	P1	3	300
S2	P2	9	400
S3	P2	6	200
S3	P2	8	200
S4	P2	1	200
S4	P4	8	200
S4	P5	7	400
S4	P5	11	400

Рис. 22.1. Пример таблицы SP, включающей поле идентификатора месяца

SP

S#	P#	MID
S1	P1	3
S1	P1	5
S1	P2	1
S1	P3	7
S1	P4	1
S1	P5	5
S1	P6	4
S2	P1	3
S2	P2	9
S3	P2	6
S3	P2	8
S4	P2	1
S4	P4	8
S4	P5	7
S4	P5	11

MONTH_QTY

MID	QTY
1	200
2	600
3	300
4	100
5	100
6	200
7	400
8	200
9	400
10	100
11	400
12	50

Рис. 22.2. Нормализованный аналог таблицы SP, представленной на рис. 22.1

Физическое проектирование

В разделе 22.2 было отмечено, что базы данных поддержки принятия решений, как правило, велики и в них широко применяются *индексы,* а также часто предусмотрены

различные виды *контролируемой избыточности.* В этом и следующих двух подразделах кратко рассматриваются эти вопросы, касающиеся физического проектирования.

Вначале рассмотрим подход к организации базы данных, называемый *секционированием* (известный также под названием **фрагментации).** Секционирование (partitioning) представляет собой возможный подход к решению проблемы *больших баз данных.* Каждая таблица делится на группу непересекающихся *секций,* или *фрагментов,* предназначенных для независимого физического хранения (см. обсуждение вопроса о фрагментации в главе 21). Такое секционирование позволяет существенно расширить возможности управления и доступа к рассматриваемой таблице. Обычно за каждой секцией закрепляются определенные выделенные (в той или иной степени) аппаратные ресурсы, например диск или процессор, вследствие чего конкуренция между секциями за такие ресурсы сводится к минимуму. Таблицы секционируются по горизонтали[4] с помощью *функции* секционирования, которая использует значения выбранных столбцов, составляющих *ключ секции,* в качестве фактических параметров, а возвращает номер или адрес секции. Такие функции обычно поддерживают *секционирование по диапазону* (range partitioning), *с помощью хэш-функции* (hash partitioning) и *циклическое секционирование* (round-robin partitioning), не считая других видов секционирования (см. аннотацию к [18.56] в главе 18).

Теперь обратимся к вопросу **индексации.** Известно, что используя подходящий индекс, можно резко сократить количество физических операций ввода—вывода. В ранних продуктах SQL предоставлялся лишь один вид индексов, а именно — индексы, представленные в виде B-деревьев. Позже, через несколько лет, стали применяться и некоторые другие виды индексов, особенно в связи с использованием баз данных поддержки принятия решений. Среди них наибольшее распространение получили *битовые индексы, хэшированные индексы, мультитабличные индексы, логические индексы, функциональные индексы,* а также уже известные нам индексы в виде B-деревьев. Рассмотрим каждый из этих пидов индексов, как описано ниже.

- *Индексы в виде B-деревьев.* Индексы в виде B-деревьев предоставляют для запросов эффективный доступ к диапазонам значений, за исключением тех случаев, когда ко личество подходящих строк становится слишком велико. Обновление B-деревьев осуществляется относительно эффективно.
- *Битовые индексы.* Предположим, что в индексированной таблице т содержится *n* строк. Тогда битовый индекс для столбца с таблицы т будет содержать вектор из л битов для каждого возможного значения С и установленный бит будет соответст вовать строке R, если в строке R содержится соответствующее значение из столбца С. Такие индексы позволяют эффективно выполнять запросы, в которых исполь зуются множества значений, но их эффективность снижается, если эти множества становятся слишком велики. Обратите внимание на то, что несколько реляцион ных операций (соединение, объединение, сокращение по равенству и т.п.) могут быть выполнены исключительно с индексами с помощью простых логических операций (AND, OR, NOT) над битовыми векторами. Доступ к реальным данным во обще не происходит до тех пор, пока не потребуется получение окончательного

[4] Хотя вертикальное секционирование также может иметь определенные преимущества, оно используется довольно редко, поскольку большинство программных продуктов его не поддерживает.

результирующего набора. Операция обновления битовых индексов выполняется относительно неэффективно.

- *Хэшированные индексы* (доступ с их использованием называется также *хэш-адресацией* или просто *хэшированием). Хэшированные* индексы эффективны для доступа к кон кретным строкам, а не к диапазонам. Вычислительная стоимость зависит от коли чества строк линейно, при том условии, что хэш-функцию не требуется дополнять с учетом дополнительных значений ключа. Хэширование может также использо ваться для эффективной реализации соединений, как описано в главе 18.
- *Мультитабличные индексы* (иногда они называются *индексами соединении).* По су ществу, элемент мультитабличного индекса содержит указатели на строки в не скольких таблицах, а не просто указатели на строки в одной таблице. Такие индек сы позволяют повысить производительность операций соединения и ускорить проверку ограничений целостности для нескольких таблиц (т.е. для базы данных), как описано в [22.33].
- *Логические индексы* (более известные как *индексы на основе логических выражений). Логический* индекс указывает, для каких строк определенной таблицы определен ное логическое выражение (включающее столбцы рассматриваемой таблицы) принимает значение TRUE. Такие индексы особенно полезны, если соответствую щее логическое выражение является общим компонентом условий операции со кращения.
- *Функциональные индексы.* Функциональные индексы индексируют строки таблицы не на основе значений в этих строках, а на основе результата применения к этим значениям некоторой функции.

В дополнение к сказанному следует отметить, что предусмотрены также различные виды *смешанных,* или гибридных, индексов, представляющих собой сочетание перечисленных выше индексов. Смысл таких сочетаний трудно изложить в общих словах. Также предлагается множество *специализированных* видов индексов, таких как *R-деревья,* которые предназначены для обработки пространственных данных. В настоящей книге не предпринимается попытка решить такую трудоемкую задачу, как описание всех видов индексов; заинтересованные читатели могут обратиться к [26.37], где содержится исчерпывающее изложение этой темы.

Наконец, рассмотрим вопрос **контролируемой избыточности.** *Контролируемая избыточность* — это важный инструмент сокращения количества операций ввода-вывода и минимизации конкурентного доступа к данным в базе. Как пояснялось в главе 1, избыточность контролируема, если она управляется СУБД и скрыта от пользователей. (Отметим, что по определению избыточность, которая должным образом контролируется на физическом уровне, остается невидимой на логическом уровне и поэтому не влияет на правильность данных на логическом уровне.) Существует два общих вида такой избыточности, которые описаны ниже.

- Первый вид включает обслуживание в базе данных точных копий *или реплик.*

  Примечание. Широко поддерживается также *управление копированием,* которое может рассматриваться как более упрощенная форма репликации (см. следующий подраздел).

- Второй вид включает обслуживание *производных данных* в дополнение к базовым данных. Наиболее часто используется в форме *итоговых таблиц* и/или *расчетных (вычисленных* или *производных) столбцов.*

Ниже каждая из этих возможностей рассматривается в отдельном подразделе.

Репликация

Основные понятия **репликации** были рассмотрены в разделах 21.3 и 21.4 главы 21 (см., в частности, подраздел "Распространение обновлений" в разделе 21.4). Здесь мы просто повторим несколько основных пунктов из этих разделов и приведем некоторые дополнительные замечания. Прежде всего, напомним, что репликация может быть *синхронной* или *асинхронной,* как описано ниже.

- В случае *синхронной* репликации, если данная реплика обновляется, все другие ре плики того же фрагмента данных также должны быть обновлены в одной и той же транзакции. Логически это означает, что существует лишь одна версия данных. В большинстве продуктов синхронная репликация реализуется с помощью триггерных процедур (возможно, скрытых и управляемых системой). Но синхронная репликация имеет тот недостаток, что она создает дополнительную нагрузку при выполнении всех транзакций, в которых обновляются какие-либо реплики (кроме того, могут возникать проблемы, связанные с доступностью данных).
- В случае *асинхронной* репликации обновление одной реплики распространяется на другие спустя некоторое время, а *не* в той же транзакции. Таким образом, при асинхронной репликации вводится *задержка,* или *время ожидания,* в течение кото рого отдельные реплики могут быть фактически неидентичными (т.е. определение *реплика* оказывается не совсем подходящим, поскольку мы не имеем дело с точными и своевременно созданными копиями). В большинстве продуктов асинхронная репликация реализуется посредством чтения журнала транзакций или постоянной очереди тех обновлений, которые подлежат распространению.

  Преимущество асинхронной репликации состоит в том, что дополнительные из держки репликации не связаны с транзакциями обновлений, которые могут иметь важное значение для функционирования всего предприятия и предъявлять высокие требования к производительности. К недостаткам этой схемы относится то, что данные могут оказаться несовместимыми (т.е. несовместимыми с точки зрения пользователя)[5]. Иными словами, избыточность может проявляться на логическом уровне, а это, строго говоря, означает, что термин *контролируемая избыточность* в таком случае не применим.

  Рассмотрим кратко проблему согласованности (или, скорее, несогласованности). Дело в том, что реплики могут становиться несовместимыми в результате таких ситуаций, которых трудно (или даже невозможно) избежать и последствия которых трудно исправить. В частности, конфликты могут возникать по поводу того, в каком порядке должны применяться обновления. Например, предположим, что в результате выполнения транзакции A происходит вставка строки в реплику x, после чего транзакция в удаляет эту строку, а также допустим, что Y — реплика X. Если обновления распространяются на Y, но вводятся в реплику Y в обратном порядке

[5] См. примечания на эту тему в предыдущей главе.

(например, из-за разных задержек при передаче), то транзакция **в** не находит в Y строку, подлежащую удалению, и не выполняет свое действие, после чего транзакция А вставляет эту строку! Суммарный эффект состоит в том, что реплика У содержит указанную строку, а реплика X — нет. В целом задачи устранения конфликтных ситуаций и обеспечения согласованности реплик являются весьма сложными. Дополнительные сведения по этой теме выходят за рамки данной книги.

Следует отметить, что, по крайней мере, в сообществе пользователей коммерческих баз данных термин *репликация* стал означать преимущественно (или даже исключительно) *асинхронную* репликацию (как уже отмечалось в главе 21).

Основное различие между репликацией (которая рассматривалась выше) и **управлением копированием** заключается в следующем. Если используется репликация, то обновление одной реплики в конечном счете распространяется на все остальные *автоматически.* В режиме управления копированием, напротив, не существует такого автоматического распространения обновлений. Копии данных создаются и управляются с помощью пакетного или фонового процесса, который отделен во времени от транзакций обновления. Управление копированием в общем более эффективно по сравнению с репликацией, поскольку за один раз могут копироваться большие объемы данных. К недостаткам можно отнести то, что большую часть времени копии данных не идентичны базовым данным, поэтому пользователи должны учитывать, когда именно были синхронизированы эти данные. Обычно управление копированием упрощается благодаря тому требованию, чтобы обновления применялись в соответствии со схемой *первичной копии* того или иного вида (см. главу 21).

Производные данные

Еще одним видом избыточности, который рассматривается в данной главе, являются производные данные, а именно **расчетные столбцы** и **итоговые таблицы.** Эти объекты особенно важны в контексте систем поддержки принятия решений. В них содержатся предварительно подсчитанные значения данных — значения, которые вычисляются на основе других данных, хранящихся в той же базе данных. Ясно, что такие объекты позволяют избежать необходимости каждый раз повторно вычислять итоговые значения, когда они понадобятся в каком-то запросе. *Расчетный столбец* — это такой столбец, значение которого в данной строке является производным от других значений в той же строке. (Еще один вариант состоит в том, что расчетное значение может быть получено на основании значений из нескольких строк в той же таблице или в какой-то другой таблице (таблицах). Но такой подход характеризуется тем, что обновление одной строки может повлечь за собой также обновление многих других строк; в частности, это может оказать очень отрицательно влияние на операции загрузки и пополнения базы данных.) *Итоговая таблица* — это таблица, в которой содержатся агрегированные значения из других таблиц (суммарные, средние значения, данные о количестве строк и т.п.). Такие агрегированные значения часто предварительно вычисляются для нескольких различных группировок одних и тех же исходных данных (см. раздел22.6).

Примечание. Итоговые таблицы часто упоминаются под разными названиями; в частности, они именуются итоговыми таблицами с автоматизированным расчетом данных (Automatic Summary Table — AST), материализованными таблицами запросов (Materialized

Query Table — MQT), *снимками* и *материализованными представлениями*. Последние два из этих терминов уже встречались раньше в этой книге (см. раздел 10.5 главы 10), где была высказана крайне резкая критика, в частности, такого термина, как *материализованное представление*. Но как бы то ни было, теперь это понятие стало темой большого раздела литературы, и в основной части этой литературы термин *представление* используется исключительно для обозначения понятия *материализованное представление* (кроме всего прочего, см. [22.3] и [22.4]).

Расчетные столбцы и итоговые таблицы чаще всего реализуются с помощью триггерных процедур, управляемых системой, хотя они могут быть реализованы и с помощью пользовательских процедур. В первом случае автоматически поддерживается согласованность между базовыми и производными данными; а при последнем подходе с проблемами несогласованности чаще всего приходится сталкиваться самому пользователю. Безусловно, если расчетные столбцы и итоговые таблицы действительно должны стать средствами контролируемой избыточности, то необходимо, чтобы они были полностью скрыты от пользователя, но при использовании описанного последнего подхода к их реализации такая цель может оказаться недостижимой.

Распространенные ошибки проектирования

В этом подразделе, как показано ниже, мы вкратце прокомментируем ошибки проектирования в среде поддержки принятия решений, которые широко распространены на практике.

- *Дублирование строк.* Проектировщики систем поддержки принятия решений часто утверждают, что применяемые ими данные просто не имеют уникальных идентификаторов, и поэтому допустимо дублирование строк. В [6.3] и [6.6] подробно объяснено, почему отсутствие средств исключения дубликатов является ошибкой. Здесь же мы просто отметим, что эта "необходимость" возникает из-за того, что физическая схема не является производной от логической схемы, которая, возможно, никогда и не создавалась. Также отметим, что в подобных проектах строки часто имеют неоднородное значение, особенно если в этих строках присутствуют неопределенные данные. Иначе говоря, не все строки являются экземплярами одного и того же предиката (см. раздел 3.4 или главу 9).

Примечание. Иногда дубликаты даже рассматриваются как полезное средство, особенно теми специалистами, которые имеют подготовку в области объектно-ориентированного программирования (см. последний абзац в разделе 25.2 главы 25).

- *Денормализация и связанные с ней действия.* Руководствуясь необоснованным стремлением исключить соединения и тем самым сократить количество операций ввода—вывода, проектировщики часто выполняют предварительные соединения таблиц, вводят различного рода производные столбцы и т.д. Такая практика может быть приемлемой на физическом уровне, но только тогда, когда результаты всех этих изменений не проявляются на логическом уровне.
- *Звездообразные схемы.* Звездообразные схемы, называемые также *многомерными* схемами, чаще всего становятся результатом попытки игнорировать правильные методы проектирования. Однако от таких попыток мало пользы. По мере увеличения количества данных в базе, снижается производительность и теряется гибкость, а разрешение возникающих трудностей с помощью физического перепроектирования требует внесения изменений и в приложения (поскольку *звездообразные*

схемы в действительности являются чисто *физическими* схемами, даже несмотря на то, что они непосредственно доступны для приложений). В общем случае проблема заключается в произвольной и необоснованной природе созданного проекта.
Примечание. Звездообразные схемы будут подробно рассматриваться в разделе 22.5.

- *Неопределенные значения.* Проектировщики часто пытаются сэкономить место, до пуская наличие в столбцах *неопределенных* значений (этот прием *может* оказаться оправданным, только если рассматриваемый столбец имеет тип данных перемен ной длины, а в используемом программном продукте *неопределенные* значения в таких столбцах на физическом уровне представляются пустыми строками). Но в общем случае подобные попытки являются ошибочными. Следует учитывать, что не только возможно, но и желательно разрабатывать проект просто таким образом, чтобы сразу же исключить необходимость использования *неопределенных* значений [19.19], так как при этом результирующие схемы часто обеспечивают более высо кую эффективность использования памяти и лучшую производительность опера ций ввода-вывода.
- *Проектирование итоговых таблиц.* Необходимость логического проектирования итоговых таблиц нередко игнорируется, вследствие чего возникают неконтроли руемая избыточность и трудности с поддержкой согласованности данных в базе. В результате пользователи сталкиваются с затруднениями при интерпретации суммарных значений и при формулировке запросов с их использованием. Чтобы избежать подобных проблем, все итоговые таблицы, относящиеся "к одному и тому же уровню агрегирования" (раздел 22.6), необходимо проектировать так, как если бы они составляли отдельную базу данных. В этом случае появляется возможность исключить определенные проблемы *циклического обновления,* запрещая обновле ния, которые охватывают несколько уровней агрегирования данных, а также син хронизируя итоговые таблицы с помощью метода, который всегда предусматрива ет агрегирование от уровня исходных данных в направлении все большего обоб- щения данных.
- *Множественные пути доступа.* Проектировщики систем поддержки принятия ре шений и их пользователи часто ошибочно указывают на "множественность путей доступа" к некоторым необходимым им данным, подразумевая, что одни и те же данные могут быть получены с помощью нескольких разных реляционных выра жений. Иногда такие выражения действительно равносильны, как в случае, на пример, A JOIN (B JOIN C) и (A JOIN B) JOIN C (см. главу 18). Иногда они равносильны благодаря действию некоторого ограничения целостности (снова см. главу 18), но чаще всего на самом деле они оказываются отнюдь не равносиль ными! Для иллюстрации последнего случая предположим, что таблицы A, в и с та ковы: A, в имеют общий столбец КАВ, в и с — общий столбец квс, а А и с — об щий столбец КАС; тогда соединение таблиц A и в по столбцу КАВ, а затем соедине ние полученного результата с таблицей с по столбцу квс, безусловно, не будет од ним и тем же, что и соединение таблицА и с по столбцу КАС.
  Ясно, что в таких ситуациях пользователи могут быть поставлены в тупик. Они не знают, какое именно выражение необходимо применять и будут ли одинаковыми полученные результаты. Безусловно, частично эта проблема может быть решена за счет дополнительного обучения пользователей. Еще одну часть проблемы можно

решить за счет обеспечения правильной работы оптимизатора. Но определенная доля ответственности возлагается и на проектировщиков, которые допускают избыточность в логической схеме и/или предоставляют пользователям непосредственный доступ к физической схеме. Следует отметить, что именно *эта* часть проблемы может быть решена только за счет правильного проектирования.

В заключение отметим, что, по мнению автора, многие затруднения при проектировании, якобы возникающие из-за специфических требований систем поддержки принятия решений, могут быть успешно преодолены в результате строгого обоснования правильного подхода. В действительности, большинство подобных проблем *вызвано* именно отказом от неукоснительного соблюдения принципов проектирования реляционных баз данных, но, откровенно говоря, эти затруднения часто *усугубляются* еще и проблемами, свойственными самому языку SQL.

22.4. ПОДГОТОВКА ДАННЫХ

Многие из вопросов, связанных с системами поддержки принятия решений, в первую очередь касаются задач получения и подготовки данных. Эти данные следует *извлечь* из разных источников, *очистить, преобразовать* и *консолидировать,* после чего *загрузить* в базу данных поддержки принятия решений. Впоследствии загруженные данные должны периодически *обновляться.* Каждая из этих операций[6] заслуживает отдельного обсуждения. Рассмотрим их поочередно, а затем завершим раздел кратким обсуждением *банков оперативных данных.*

Извлечение данных

Извлечение данных — это процесс выборки данных из оперативных баз данных и других источников. Для извлечения данных существует множество инструментов, включая утилиты, предоставляемые системой, пользовательские программы извлечения и коммерческие продукты извлечения данных (общего назначения). В процессе извлечения обычно интенсивно используются операции ввода-вывода, что может послужить помехой для выполнения других операций, важных с точки зрения деятельности предприятия. Поэтому извлечение данных часто осуществляется в параллельном режиме (т.е. как множество параллельно выполняемых подпроцессов) и на физическом уровне. Но такие "физические операции извлечения" могут вызвать проблемы при последующей обработке, поскольку они могут сопровождаться потерей информации (особенно данных о связях), которая представляется каким-либо физическим способом, например, с помощью указателей или физически смежного размещения. По этой причине программы извлечения иногда предоставляют средства защиты такой информации с помощью полей последовательных номеров записей и замены указателей значениями внешнего ключа.

Очистка данных

Удовлетворительный контроль информации обеспечивают лишь немногие источники данных. Вследствие этого, прежде чем данные будут введены в базу данных поддержки принятия решений, обычно требуется выполнить их **очистку** (как правило, эта операция

---

[6] Отметим, между прочим, что в этих операциях часто можно было бы использовать преимущества обработки на уровне множеств, свойственные реляционным системам, хотя на практике это происходит редко.

выполняется в пакетном режиме). Обычно очистка предусматривает заполнение отсутствующих значений, корректировку опечаток и других ошибок, допущенных при вводе данных, определение стандартных сокращений и форматов, замену синонимов стандартными идентификаторами и т.д. Данные, которые определяются как ошибочные и не могут быть исправлены, отбрасываются.

Примечание. Информация, полученная при выполнении очистки данных, иногда используется для выявления причин ошибок в источниках данных и поэтому способствует повышению качества содержащейся в них информации.

Преобразование и консолидация данных

После очистки данных полученная информация, скорее всего, еще не будет отвечать требованиям системы поддержки принятия решений и, следовательно, будет нуждаться в соответствующем **преобразовании.** Обычно данные подготавливаются в виде набора файлов, по одному файлу для каждой таблицы, определенной в физической схеме. Поэтому процедура преобразования данных должна предусматривать построчное разбиение или объединение исходных записей, как описано в разделе 1.5 главы 1. Кроме того, в целях повышения производительности операции преобразования часто выполняются параллельно. Они могут требовать выполнения большого количества операций ввода—вывода и больших затрат процессорного времени.

Примечание. Ошибки, которые не были исправлены во время очистки, иногда проявляются в процессе преобразования данных. Как и при очистке, любые неправильные данные в общем случае отбрасываются. Аналогичным образом, информация об ошибках, полученная в ходе преобразования данных, может использоваться для повышения качества источников данных.

Процедура преобразования приобретает особую важность, когда необходимо выполнить слияние данные, поступившие из нескольких разных источников. Этот процесс называется **консолидацией. В** таком случае любая неявная связь между данными из отдельных источников должна быть преобразована в явную путем введения явных значений данных. Кроме того, если отдельные значения даты и времени связаны и имеют определенный деловой смысл, они должны быть проконтролированы и приведены в соответствие между отдельными источниками. Этот процесс называется *синхронизацией времени.*

Следует отметить, что задача синхронизации времени может оказаться довольно сложной. Предположим, например, что необходимо найти средний доход за квартал от обслуживания заказчиков на каждого торгового работника. Предположим также, что данные о заказчиках и доходах ведутся по финансовым кварталам в базе данных бухгалтерии, а данные о торговых работниках и заказчиках ведутся по календарным кварталам в базе данных отдела сбыта. Очевидно, что необходимо выполнить слияние данных двух указанных баз. Задача консолидации данных о заказчиках менее сложна — она предусматривает простую проверку соответствия идентификаторов заказчиков. Однако задача синхронизации времени значительно сложнее. Мы можем вычислить доходы от обслуживания заказчиков за *финансовый* квартал (из базы данных бухгалтерии), но мы не можем сказать, какие торговые работники отвечали за обслуживание каждого из заказчиков в конкретное время, и поэтому вообще не можем определить доходы от обслуживания заказчиков за *календарный* квартал.

Загрузка данных

Разработчики СУБД придают большое значение повышению эффективности операций **загрузки данных. В** данном случае *операцию загрузки данных* разобьем на следующие этапы: а) пересылка преобразованных и консолидированных данных в базу данных поддержки принятия решений; б) проверка согласованности данных (т.е. проверка их целостности); в) построение всех необходимых индексов. Ниже приведены краткие комментарии относительно каждого из этих этапов.

Пересылка данных. Современные СУБД обычно предоставляют утилиты параллельной загрузки данных. Иногда, прежде чем будет выполнена настоящая загрузка, данные преобразуются во внутренний физический формат, требуемый для целевой СУБД. Альтернативный и более эффективный метод предусматривает загрузку в рабочие таблицы, состав которых отражает структуру целевой схемы. В этих рабочих таблицах (см. выше п. б) может быть выполнена необходимая проверка целостности, а затем для пересылки данных из рабочих таблиц в целевые таблицы может использоваться операция INSERT, осуществляемая на уровне множеств.

Проверка целостности. Большая часть процесса проверки целостности загружаемых данных может быть проведена еще до реальной загрузки, без обращения к данным, уже находящимся в базе. Однако некоторые ограничения все же не могут быть проверены без обращения к существующей базе данных. Например, ограничение, контролирующее уникальность значений, в общем случае должно проверяться во время реальной загрузки (или, если загрузка выполняется в пакетном режиме, после ее завершения). *Построение индексов.* Наличие индексов может резко замедлить процесс загрузки данных, поскольку большинство продуктов выполняет обновление индексов при вставке в таблицу каждой строки. Поэтому иногда имеет смысл удалять индексы перед загрузкой данных, а затем, после ее завершения, создавать их заново. Однако такой подход не будет целесообразным, если количество новых данных по отношению к уже существующим довольно мало; в этом случае затраты на создание индексов для всей таблицы будут существенно больше затрат на обновление индексов. Кроме того, создание индексов для большой таблицы может быть причиной неустранимых ошибок выделения памяти, причем вероятность их возникновения по мере увеличения объема таблицы увеличивается. ***Примечание.*** Большинство современных СУБД поддерживает режим параллельного создания индексов, позволяющий ускорить процессы загрузки данных и построения индексов.

Обновление данных

В большинстве баз данных поддержки принятия решений требуется периодическое **обновление** данных для поддержки их актуальности. Обновление обычно предусматривает частичную загрузку, хотя для некоторых систем поддержки принятия решений требуется удаление всех данных из базы и их полная перезагрузка. При обновлении возникают те же проблемы, что и при загрузке, и, кроме того, может потребоваться, чтобы обновление выполнялось в то время, когда пользователи обращаются к базе данных (а это влечет за собой появление других проблем).

Банки оперативных данных

Банки оперативных данных (Operational Data Store — ODS) представляют собой "предметно-ориентированные, интегрированные, изменчивые, т.е. обновляемые, текущие или почти текущие коллекции данных" [22.20]. Другими словами, банки оперативных данных — это специализированные базы данных. Термин *предметно-ориентированные* означает, что рассматриваемые данные представляют некоторую конкретную предметную область, например информацию о заказчиках, товарах и т.п. Банки оперативных данных могут использоваться для разных целей — как область накопления для физической реорганизации извлеченных оперативных данных, как средство формирования оперативных отчетов, а также как средство поддержки оперативных решений. Кроме того, банки оперативных данных могут служить местом консолидации данных, если оперативные данные поступают из нескольких источников. Поэтому они могут быть предназначены для многих целей.

Примечание. Поскольку банки оперативных данных не накапливают исторических данных, обычно они не разрастаются до слишком больших объемов. С другой стороны, банки оперативных данных обычно достаточно часто или даже непрерывно пополняются информацией из источников оперативных данных. Для этой цели иногда используется асинхронная репликация данных из источников оперативных данных в банки оперативных данных (благодаря этому часто удается поддерживать полную актуальность данных с задержкой лишь в несколько минут). Проблемы синхронизации времени (см. приведенный выше подраздел "Преобразование и консолидация данных") в операционных хранилищах данных могут успешно преодолеваться, если обновление выполняется достаточно часто.

22.5. ХРАНИЛИЩА ДАННЫХ И МАГАЗИНЫ ДАННЫХ

Оперативные системы с базами данных обычно характеризуются жесткими требованиями к производительности, предсказуемым уровнем общей нагрузки, сравнительно небольшими единицами работы и высоким коэффициентом использования. Системы поддержки принятия решений, напротив, обычно имеют изменяющиеся со временем требования к производительности, непредсказуемый уровень нагрузки, большие единицы работы и характеризуются нерегулярным использованием. Из-за этих различий могут возникнуть трудности при организации совместного функционирования оперативной системы обработки данных и системы поддержки принятия решений в рамках единой системы. Особенно это касается планирования объемов, управления ресурсами и настройки производительности системы. Поэтому системные администраторы оперативных систем обычно неохотно разрешают устанавливать приложения поддержки принятия решений в своих системах. И в связи с этим применяется подход, основанный на раздельном использовании двух систем.

Примечание. Отметим в качестве небольшого отступления, что так было не всегда. Раньше системы поддержки принятия решений реализовались на базе оперативных систем, но для эксплуатации с низким приоритетом или во время так называемых "перерывов на обработку в пакетном режиме". При наличии достаточных вычислительных ресурсов этот подход имеет несколько преимуществ и наиболее очевидным из них, пожалуй, является то, что он позволяет избежать всевозможных затратных процедур копирования и переформатирования данных, а также дополнительных операций передачи информации, необходимых при работе с двумя отдельными системами. На практике

преимущества от совместного функционирования оперативных систем и систем поддержки принятия решений находят все большее понимание (см. [21.9]). Однако подробное рассмотрение интеграции этих систем выходит за рамки данной главы.

Тем не менее, несмотря на сказанное в предыдущем абзаце, остается неоспоримым тот факт, что, по крайней мере, ко времени написания книги данные систем поддержки принятия решений обычно извлекаются из различных оперативных систем (часто в корне отличных по своей организации) и помещаются в собственное хранилище, реализованное на отдельной платформе. Такое отдельное хранилище на*зываютхранилищем данных.*

Хранилище данных

Подобно банку оперативных данных (а также магазину данных; см. следующий подраздел), хранилище данных — это специальная база данных. Этот термин возник, повидимому, в конце 1980-х годов [22.15], [22.18], хотя соответствующее определение данного понятия было сформулировано немного позднее. В [22.19] хранилище данных определяется как "предметно-ориентированное, интегрированное, постоянное, изменяющееся во времени хранилище информации для поддержки управленческих решений". Здесь термин *постоянное* означает, что, будучи введенными, данные впоследствии не изменяются, хотя и могут быть удалены. Хранилища данных появились по двум причинам: во-первых, для систем поддержки принятия решений необходимо было предоставить отдельный, чистый, согласованный источник данных, и, во-вторых, этой цели требовалось достичь, не оказывая влияния на функционирование оперативных систем.

По определению ожидаемая рабочая нагрузка на хранилище данных обусловлена нагрузкой в системе поддержки принятия решений. Поэтому можно предположить, что хранилище данных будет подвергаться частым обращениям с запросами, кроме того, в ней будет периодически осуществляться обработка в пакетном режиме для обновления данных. К тому же для хранилищ данных характерен весьма большой объем занимаемой памяти — часто он составляет многие терабайты, а увеличение этого объема может достигать 50% в год или даже больше. Вследствие этого бывает трудно добиться высокой производительности системы, хотя и нельзя сказать, что это невозможно. Могут также возникнуть проблемы, связанные с масштабируемостью базы данных. Причины подобных затруднений обычно включают: а) ошибки проектирования базы данных (обсуждавшихся в последнем подразделе раздела 22.3); б) неэффективное использование реляционных операций (что было кратко описано в разделе 22.2); в) наличие недостатков в реализации реляционной модели в целевой СУБД; г) недостаточные возможности масштабирования, предусмотренные в самой целевой СУБД; д) наличие ошибок в архитектуре проекта, ограничивающих объемы и препятствующих масштабированию платформы. Пункты а) и б) уже рассматривались в этой главе, п. в) подробно обсуждался в части II и в других главах книги, а описаниепп. г) и д) выходит за рамки данной книги.

Магазины данных

Хранилища данных в общем случае представляют собой единый источник информации для любой обработки, связанной с поддержкой принятия решений. Но в начале 1990-х годов, когда хранилища данных только приобретали популярность, было обнаружено, что пользователи чаще всего составляли отчеты и выполняли различные операции анализа данных на относительно небольшом подмножестве всего множества информации в хранилище данных. И действительно, пользователи повторяли те же самые операции на том же самом подмножестве данных каждый раз после их обновления. Более того, некоторые

из этих операций, например, прогностический анализ (прогнозирование), имитация, моделирование возможных сценариев дальнейшего развития событий на основе деловых данных, требовали создания новых схем и данных с последующим обновлением этих новых данных.

Неоднократное повторное выполнение таких операций на одном и том же подмножестве информации полного хранилища данных, безусловно, не очень эффективно. Поэтому возникла очевидная идея формирования некоторого ограниченного "хранилища" специального назначения, которое подходило бы для достижения рассматриваемых целей. Кроме того, в определенных случаях приходится извлекать и обрабатывать требуемые данные непосредственно из локальных источников, предоставляя более быстрый доступ к данным по сравнению с тем, который мог быть предоставлен при синхронизации со всеми остальными данными, загруженными в полное хранилище. Подобные соображения привели к появлению концепции **магазинов** данных.

На самом деле, вокруг точного определения термина *магазин данных* еще ведутся споры. Для наших целей магазин данных можно определить как "специализированное, предметно-ориентированное, интегрированное, непостоянное, изменяющееся во времени хранилище информации для поддержки конкретного подмножества управленческих решений". Как видим, ключевое различие между магазинами данных и хранилищами данных заключается в том, что магазины данных — *специализированные* и *непостоянные.* Под характеристикой *специализированные* подразумевается то, что они содержат данные для поддержки лишь некоторой конкретной области делового анализа, а под характеристикой *непостоянные* подразумевается то, что пользователи могут обновлять данные и, возможно, даже создавать в каких-то целях новые объекты данных, например новые таблицы.

Ниже описаны три основных подхода к созданию магазинов данных.

- Данные просто извлекают из хранилища данных, по существу, следуя подходу "разделяй и властвуй" применительно к процессу поддержки принятия решений в целом. Это позволяет достичь более высокого уровня производительности и мас штабируемости. Обычно извлеченные данные загружаются в базу данных с такой физической схемой, которая имеет близкое сходство с соответствующим подмно жеством хранилища данных. Часто такая схема может быть даже несколько упро щена благодаря узкой специализации магазинов данных.
- Несмотря на то, что хранилища данных предназначены для создания "единого пункта управления" информацией, магазины данных все еще могут функциони ровать независимо (т.е. без использования данных, извлеченных из хранилища). Такой подход может быть приемлемым, когда хранилище данных по каким-то причинам недоступно, например, по финансовым, оперативным или даже управ ленческим соображениям (или же хранилище данных может просто еще не суще ствовать; см. следующий пункт).
- В некоторых установках используется обратный подход: сначала создаются необ ходимые магазины данных, а полное хранилище данных формируется впоследст вии путем объединения информации из различных магазинов данных.

Для двух последних подходов обычно характерны проблемы, связанные с семантическим несоответствием. К таким проблемам особенно восприимчивы независимые магазины данных, поскольку не существует очевидных способов проверки семантического несоответствия, если базы данных спроектированы независимо. Консолидация магазинов

данных в одно хранилище данных в общем случае оканчивается неудачей, кроме тех ситуаций, когда сначала создается единая логическая схема для хранилища данных, и только затем формируются схемы для отдельных магазинов данных, производные от схемы полного хранилища данных. (Безусловно, при необходимости схема для хранилища данных может постепенно расширяться в целях включения данных о каждом новом магазине; разумеется, если она была должным образом спроектирована с самого начала.)

При проектировании базы данных поддержки принятия решений важно правильно определить уровень детализации базы данных. Под термином *уровень детализации* здесь подразумевается самый низкий уровень агрегирования данных, предназначенных для хранения в базе данных. Для большинства приложений поддержки принятия решений рано или поздно требуется доступ к исходным данным, так что с уровнем детализации для хранилищ данных определиться несложно. Но для магазинов данных это сделать несколько труднее. Если уровень детализации занижен и данные нижнего уровня используются не очень часто, то извлечение больших объемов исходных данных из хранилища данных и их сохранение в магазине данных может оказаться весьма неэффективным решением. С другой стороны, иногда трудно точно установить, какой именно нижний уровень агрегирования фактически необходим. В таких случаях доступ к исходным данным при возникновении этой необходимости может предоставляться непосредственно с помощью хранилища данных, тогда как сопровождение агрегированных данных осуществляется с помощью магазина данных. В то же время полное агрегирование данных в целом не выполняется, поскольку в результате применения множества способов агрегирования данных будут получены слишком большие объемы итоговых данных. Эта тема обсуждается более подробно в разделе 22.6.

Рассмотрим еще одно замечание. Поскольку пользователи магазинов данных часто применяют определенные аналитические инструменты, требования к физическому проекту часто определяются, по крайней мере, частично, на основании того, какие конкретные инструменты должны использоваться (см. обсуждение двух видов аналитической обработки ROLAP и MOLAP в разделе 22.6). Такое неудовлетворительное состояние дел может привести к созданию *многомерных схем* (рассматриваемых ниже), которые не отвечают правилам качественного реляционного проектирования.

Многомерные схемы

Предположим, что необходимо накапливать исторические сведения о выполнении деловых транзакций для последующего анализа. Как отмечалось в разделе 22.1, в первых версиях систем поддержки принятия решений эта история сохранялась бы в виде обычного файла, доступ к которому осуществлялся бы с помощью последовательного просмотра. Но по мере роста объема данных для просмотра файла все более выгодной (с различных точек зрения) становится поддержка прямого доступа к записям файла. Например, может оказаться желательным, чтобы существовала возможность поиска всех деловых транзакций, касающихся конкретного товара, или всех деловых транзакций, осуществленных в указанный период времени или относящихся к конкретному заказчику.

Один из методов организации данных, обеспечивающий подобный тип доступа, известен под названием *многокаталоговой*[7] базы данных. Продолжая приведенный выше

[7] Здесь слово "каталог" не имеет ничего общего с каталогами баз данных в современном смысле этого термина.

пример, можно предположить, что подобная база данных могла бы состоять из огромного централизованного файла данных, содержащего данные о деловых транзакциях наряду с тремя отдельными файлами *каталогов* для товаров, периодов времени и заказчиков, соответственно. Рассматриваемые файлы каталогов схожи с индексами, в которых содержатся указатели на записи в файле данных, но элементы каталога могут создаваться пользователем явно *{сопровождение каталога),* а каталоги могут содержать дополнительную информацию (например, адрес заказчика), которую затем можно удалить из файла данных. Следует отметить, что файлы каталогов обычно малы по сравнению с файлами данных.

При такой организации данных более эффективно используется память и затраты на ввод-вывод становятся гораздо меньше, чем при использовании первоначального проекта, предполагающего наличие простых линейных файлов данных. Отметим, в частности, что информация о товарах, периодах времени и заказчиках в основном файле данных теперь состоит просто из *идентификаторов* товаров, периодов времени и заказчиков.

Если этот подход имитировать в реляционной базе данных, то файл данных и файлы каталогов преобразуются в таблицы (образы соответствующих файлов), указатели в каталогах файлов будут заменены первичными ключами в таблицах, которые служат образами файлов каталогов, а идентификаторы в файле данных станут внешними ключами **в** таблице, соответствующей этому файлу данных. Обычно эти первичные и внешние ключи полностью индексированы. При таком соответствии образ файла данных называют **таблицей фактов,** а образы файлов каталогов — **таблицами размерностей.** Весь проект называют **многомерным** или имеющим схему **типа** "звезда". Такое название подчеркивает характерную особенность структуры этой схемы; если начертить соответствующую диаграмму "сущность/связь", то таблица фактов будет окружена таблицами размерностей и связана с ними.

Примечание. Смысл термина *размерность* разъясняется в разделе 22.6.

В целях демонстрации предположим, что снова необходимо модифицировать базу данных поставщиков и деталей, но на этот раз таким образом, чтобы показать каждую поставку за определенный период времени, когда осуществлялась эта поставка. Присвоим поставке за определенный период идентификатор TI# и введем еще одну таблицу тт., чтобы связать идентификаторы с соответствующими периодами. Теперь исправленная таблица SP и новая таблица периодов времени TI будут выглядеть так, как показано на рис. 22.3[8]. В соответствии с терминологией схем типа "звезда", таблица поставок SP представляет собой таблицу фактов, а таблица периодов времени TI — таблицу размерностей (в эту схему также входят таблица поставщиков S и таблица деталей P, как показано на рис. 22.4).

Примечание. Напомним, что общие вопросы обработки данных, представляющих периоды времени, будут подробно рассмотрены в главе 23.

При обработке запросов в схеме типа "звезда" таблицы размерностей обычно используются для поиска всех необходимых сочетаний внешних ключей, после чего найденные сочетания используются для доступа к таблице фактов. Предположим, что доступ к таблице размерностей и последующий доступ к таблице фактов связаны в единый запрос. Тогда лучшим способом реализации такого запроса, как правило, является так называемое

[8] В столбцах FROM и TO таблицы TI содержатся данные типа временной отметки. Для упрощения rfa рисунке показаны не реальные значения временных отметок, а символические обозначения.

звездообразное соединение. Звездообразное соединение представляет собой специальную стратегию реализации операции соединения, которая отличается от обычных стратегий тем, что соединение намеренно начинается с вычисления декартова произведения, а именно — декартова произведения таблиц размерностей. Как уже было показано в главе 18, оптимизаторы запросов обычно пытаются избежать вычисления декартова произведения. Но в данном случае формирование в первую очередь декартова произведения таблиц размерностей гораздо меньшего объема, а затем использование полученного результата для просмотра таблицы фактов очень больших размеров с помощью индексов почти всегда эффективнее любой другой стратегии. Поэтому для эффективной обработки запросов в схеме типа "звезда" многие оптимизаторы в коммерческих продуктах были дополнены с учетом звездообразного соединения.

SP

S#	P#	TI#	QTY
S1	P1	TI3	300
S1	P1	TI5	100
S1	P2	TI1	200
S1	P3	TI2	400
S1	P4	TI1	200
S1	P5	TI5	100
S1	P6	TI4	100
S2	P1	TI3	300
S2	P2	TI4	400
S3	P2	TI1	200
S3	P2	TI3	200
S4	P2	TI1	200
S4	P4	TI3	200
S4	P5	TI2	400
S4	P5	TI1	400

TI

TI#	FROM	TO
TI1	*ta*	*tb*
TI2	*tc*	*td*
TI3	*te*	*tf*
TI4	*tg*	*th*
TI5	*ti*	*tj*

Рис. 22.3. Пример таблицы фактов (SP) и таблицы размерностей (TI)

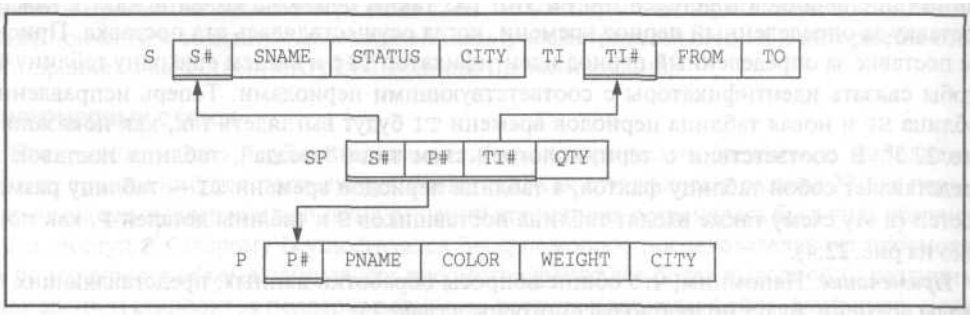


Рис. 22.4. Схема типа "звезда" для базы данных поставщиков и деталей (в которой учитываются периоды времени)

У читателя может возникнуть вопрос: в чем же отличие схемы типа "звезда" от схемы, которую можно было считать настоящим реляционным проектом? Действительно, *простая* схема типа "звезда", подобная показанной на рис. 22.4, может показаться очень похожей на настоящий реляционный проект (и даже идентичной ему). Но, к сожалению, в общем случае при использовании схем типа "звезда" возникает целый ряд описанных ниже проблем.

1. Прежде всего, эта схема — *произвольная,* поскольку она основана на интуиции, а не на теоретически обоснованном подходе. Из-за недостаточной научной обосно ванности возникают сложности, когда схему требуется надлежащим образом мо дифицировать, например, чтобы добавить в базу новые типы данных или изменить ограничения. На практике схемы типа "звезда" часто формируются с помощью простого редактирования предыдущего проекта, притом что предыдущий проект, в свою очередь, формировался методом проб и ошибок.
2. Схемы типа "звезда" в действительности представляют собой *физические,* а не логические описания данных, хотя о них и говорят как о логических. Проблема заключается в том, что при данном подходе отсутствует концепция логического проектирования, отдельного от физического.
3. Подход с использованием схем типа "звезда" не всегда позволяет получить в ре зультате правильный физический проект (т.е. проект, который сохраняет всю ин формацию правильного реляционного логического проекта). Этот недостаток ста новится все более очевидным по мере усложнения схемы.
4. Поскольку отсутствует строгий теоретический подход, проектировщики часто включают в одну таблицу фактов несколько различных типов фактов. Вследствие этого строки и столбцы таблицы фактов обычно не имеют единой интерпретации. Более того, определенные столбцы часто предназначены только для узкого набора типов фактов, а это означает, что рассматриваемые столбцы должны допускать использование неопределенных значений. По мере включения в таблицу все большего количества фактов различных типов, ее становится все сложнее сопро вождать и понимать, а доступ становится все менее эффективным. Например, до пустим, что нужно модифицировать таблицу поставок для отслеживания закупок и поставок деталей. Поэтому потребуется некоторый столбец с "флажками", указы вающими, какие строки относятся к закупкам, а какие к поставкам. В отличие от этого, в концептуально правильном проекте была бы создана отдельная таблица фактов для каждого типа фактов.
5. Опять-таки, из-за отсутствия строгого теоретического подхода таблицы размерно стей могут оказаться неоднородными. Такая ошибка обычно возникает, когда таб лица фактов используется для размещения данных, относящихся к разным уровням агрегирования. Например, в таблицу поставок могут быть (ошибочно) добавлены строки, содержащие итоговые количества деталей, поставляемых за каждый день, каждый месяц, каждый квартал, каждый год и даже сводный текущий итог на оп ределенную дату. Прежде всего, такое изменение приводит к тому, что столбцы идентификатора периода времени (TI#) и количества (QTY) в таблице SP не будут иметь в разных строках единообразный смысл. Предположим теперь, что столбцы FROM и то в таблице размерностей тт. заменены комбинациями столбцов YEAR, MONTH, DAY И Т.Д. Тогда все ЭТИ столбцы YEAR, MONTH, DAY И др. ДОЛЖНЫ будут ДО-
пускать использование неопределенных значений. Кроме того, возможно, потребуется также столбец флажка, обозначающего соответствующий период времени.

6. Таблицы размерностей часто не полностью нормализованы[9]. Желание избежать использования операций соединения часто приводит проектировщиков к решению объединить разные данные в одной таблице, хотя лучше было бы сохранять их отдельно. В крайнем случае столбцы, к которым лишь иногда осуществляется совместное обращение, также содержатся все вместе в одной и той же таблице размерностей. Должно быть ясно, что проявления таких крайностей и отсутствие реляционной строгости почти наверняка приведут к неконтролируемой избыточности (а может быть, даже к такой избыточности, которую и невозможно контролировать).

Наконец, отметим, что одним из вариантов схемы типа "звезда" является **схема типа "снежинка",** которая предусматривает нормализацию таблиц размерностей. Название этой схемы также было принято по аналогии с ее изображением в виде диаграммы "сущность-связь". Изредка можно встретить термины *схема типа "созвездие"* и *схема типа "метель "* с очевидным (?!) смыслом.

22.6. ОПЕРАТИВНАЯ АНАЛИТИЧЕСКАЯ ОБРАБОТКА

Термин **оперативная аналитическая обработка** (On-Line Analytical Processing— OLAP) впервые был упомянут в докладе, подготовленном для корпорации Arbor Software Corp. в 1993 году [22.11], хотя определение этого термина, как и в случае с хранилищами данных, было сформулировано намного позже. Понятие, обозначенное этим термином, может быть определено как "интерактивный процесс создания, сопровождения, анализа данных и выдачи отчетов". Кроме того, обычно добавляют, что рассматриваемые данные должны восприниматься и обрабатываться таким образом, как если бы они хранились в *многомерном массиве.* Но прежде чем приступить к обсуждению собственно многомерного представления, рассмотрим соответствующие идеи в терминах традиционных таблиц SQL.

Первая особенность состоит в том, что при аналитической обработке непременно требуется некоторое агрегирование *данных,* обычно выполняемое сразу с помощью нескольких различных способов или, иными словами, в соответствии с многими различными критериями группирования. В сущности, одной из основных проблем аналитической обработки является то, что количество всевозможных способов группирования очень скоро становится слишком большим. Тем не менее, пользователям необходимо рассмотреть все или почти все такие способы. Безусловно, теперь в стандарте SQL поддерживается подобное агрегирование, но любой конкретный запрос SQL вырабатывает в качестве своего результата только одну таблицу, а все строки в этой результирующей таблице имеют одинаковую форму и одну и ту же интерпретацию[10] (по крайней мере, так

---

[9] Приведем совет из книги по хранилищам данных [22.24]: "[Откажитесь] от нормализации... По пытки нормализовать любую из таблиц в многомерной базе данных исключительно ради экономии дис кового пространства [именно так!] — напрасная трата времени... Таблицы размерности не должны быть нормализованы... Нормализованные таблицы размерности исключают возможность просмотра".

[10] Если только эта таблица результатов не включает какие-либо неопределенные значения, или NULL-значения (см. главу 19, раздел 19.3, подраздел "Дополнительные сведения о предикатах"). На самом деле конструкции SQL: 1999, которые должны быть описаны в данном разделе, можно охарактеризовать как "основанные на использовании" этого весьма не рекомендуемого средства SQL (?); в действительности, они подчеркивают тот факт, что в своих различных проявлениях неопределенные значения могут иметь разный смысл, и поэтому позволяют представить в одной таблице много разных преди катов (как будет показано ниже).

было до появления стандарта SQL: 1999). Поэтому, чтобы реализовать *n* различных способов группирования, необходимо выполнить *n* отдельных запросов и создать в результате л отдельных таблиц. Например, рассмотрим приведенную ниже последовательность запросов, выполняемых в базе данных поставщиков и деталей.

1. Определить общее количество поставок.
2. Определить общее количество поставок по поставщикам.
3. Определить общее количество поставок по деталям.
4. Определить общее количество поставок по поставщикам и деталям.
   (Безусловно, "общее" количество для данного поставщика и для данной детали — это просто фактическое количество для данного поставщика и данной детали. Пример был бы более реалистичным, если бы использовалась база данных по ставщиков, деталей и проектов. Но, чтобы не усложнять этот пример, мы все же остановились на обычной базе поставщиков и деталей.)

Теперь предположим, что есть только две детали, с номерами P1 и P2, а таблица поставок выглядит следующим образом.

SP

S#	P#	QTY
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200

Тогда формулировки на языке SQL для четырех указанных выше запросов и соответствующие результаты их выполнения будут такими, как показано ниже.

Примечание. В стандарте SQL: 1999 разрешено (а в стандарте SQL: 1992— нет), во-первых, заключать операнды конструкции GROUP BY в круглые скобки и, во-вторых, применять конструкцию GROUP BY вообще без операндов (последнее эквивалентно полному исключению конструкции GROUP BY).

```
1. SELECT  SUM(QTY) AS
   TOTQTY FROM    SP GROUP
   BY () ;
```

TOTQTY
1600

```
2. SELECT  S#
          SUM(QTY) AS
   TOTQTY FROM    SP GROUP
   BY (S#) ;
```

S#	TOTQTY
S1	500
S2	700
S3	200
S4	200

Недостатки этого подхода очевидны. Составление такого большого количества про-

```
3.  SELECT  P#
          SUM(QTY)  AS
    TOTQTY FROM    SP GROUP
    BY  (P#) ;
```

P#	TOTQTY
P1	600
P2	1600

```
4.  SELECT  S#,  P#
          SUM(QTY) AS
    TOTQTY FROM    SP GROUP
    BY (S#,P#) ;
```

S#	P#	TOTQTY
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200

стых, но разных формулировок запросов утомительно для пользователя. Выполнение же всех необходимых запросов (их запуск по одним и тем же данным снова и снова), вероятно, будет слишком расточительным по времени. Поэтому стоит попытаться найти какой-то способ, чтобы в одном запросе можно было получить несколько уровней агрегирования и таким образом, во-первых, упростить работу пользователя и, во-вторых, предложить реализацию, позволяющую вычислять все эти агрегированные данные более эффективно (т.е. за один проход). Именно этими соображениями руководствовались разработчики, создавая ОПЦИИ GROUPING SETS, ROLLUP И CUBE КОНСТРУКЦИИ GROUP BY.

Примечание. Эти опции уже поддерживаются в коммерческих продуктах в течение нескольких лет, а в стандарт SQL введены в 1999 году.

Опция **GROUPING SETS** позволяет пользователю точно указать, какой именно способ группирования ему требуется. Например, приведенный ниже оператор SQL представляет собой сочетание запросов 2 и 3.

```
SELECT S#, P#, SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY GROUPING SETS  (  (S#),  (P#)  )  ;
```

Здесь с помощью конструкции GROUP BY системе фактически передается требование на выполнение двух запросов, в одном из которых группирование происходит по атрибуту s#, а в другом — по атрибуту P#.

Примечание. Внутренние скобки в этом примере на самом деле не требуются, поскольку каждое *группируемое множество* включает просто один столбец. Мы показали их для наглядности.

Идея объединения нескольких отдельных запросов в один оператор указанным способом сама по себе не вызывает особых возражений (хотя необходимо все же сказать, что

мы бы предпочли, чтобы эта очень важная проблема была бы решена с помощью более общего, систематического и обоснованного подхода). Но, к сожалению, создатели языка SQL пошли по пути объединения *результатов* этих логически отдельных запросов в одну результирующую таблицу! В рассматриваемом примере таблица результатов могла бы выглядеть следующим образом.

S#	P#	TOTQTY
S1	*null*	500
S2	*null*	700
S3	*null*	200
S4	*null*	200
null	P1	600
null	P2	1000

Этот результат действительно можно считать *таблицей* (во всяком случае, таблицей в стиле SQL), но вряд ли его можно назвать *отношением.* Обратите внимание на то, что строки с данными о поставщиках (они содержат неопределенные значения в столбце P#) и строки с данными о деталях (которые содержат неопределенные значения в столбце S#) имеют совершенно различные интерпретации. А смысл значений TOTQTY становится различным в зависимости оттого, относятся ли они к строкам с данными о поставщиках или к строкам с данными о деталях. Так каким же может быть предикат для такого "отношения"? (В действительности, результирующую таблицу в данном примере можно рассматривать, скорее как *внешнее объединение* результатов запросов 2 и 3, но весьма странное внешнее объединение. Как описано в главе 19, операция *внешнего объединения,* причем даже и не в такой странной форме, не заслуживает названия полноценной реляционной операции.)

Отметим, что неопределенные значения в этом результирующей таблице представляют собой еще и некоторый вид *отсутствующей информации.* Они определенно не означают, что "значение неизвестно" или "значение не используется", но что именно они означают, совершенно неясно.

Примечание. В языке SQL предоставляется возможность отличить эти новые неопределенные значения от других видов, но изучение подробных сведений о такой процедуре весьма утомительно; по сути, пользователь вынужден отдельно анализировать каждую строку. Определенное представление о том, что с этим связано, можно получить, рассмотрев следующий пример (который показывает, как фактически может выглядеть на практике приведенный выше пример применения конструкции GROUPING SETS).

```
SELECT CASE GROUPING (
       S# ) WHEN 1 THEN '?
       ?' ELSE S# AS S#,
       CASE GROUPING ( P#
       ) WHEN 1 THEN '!!'
       ELSE P# AS P#,
       SUM(QTY) AS TOTQTY FROM   SP
GROUP  BY GROUPING SETS ( (S#), (P#)
) ;
```

При использовании этой пересмотренной формулировки неопределенные значения в столбце s# результата заменяются парой вопросительных знаков, а пустые неопределенные значения в столбце p# заменяются парой восклицательных знаков, что приводит к получению следующих результатов.

S#	P#	TOTQTY
S1	!!	500
S2	!!	700
S3	!!	200
S4	!!	200
??	P1	600
??	P2	1000

Возвратимся к конструкции GROUP BY. Две другие опции конструкции GROUP BY, ROLLUP и CUBE, по сути, являются сокращениями для определенных сочетаний в конструкции GROUPING SETS. Сначала рассмотрим опцию ROLLUP на примере следующего запроса.

```
SELECT S#, P#, SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY ROLLUP ( S#, P# ) ;
```

В данном случае конструкция GROUP BY будет равносильно следующей.

```
GROUP  BY GROUPING SETS ( (S#,P#), (S#), () ) ;
```

Другими словами, этот запрос объединяет формулировки запросов 4, 2 и 1 на языке SQL. Результат его выполнения выглядит следующим образом.

S#	P#	TOTQTY
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S1	*null*	500
S2	*null*	700
S3	*null*	200
S4	*null*	200
null	*null*	1600

Термин ROLLUP (накопить) принят из-за того, что в данном примере количества "накапливаются" для каждого поставщика (т.е. накапливаются "по размерности поставщика"; см. приведенный ниже подраздел "Многомерные базы данных"). В общем конструкция GROUP BY ROLLUP (A, B, . . . , Z), или, проще говоря, "накопить по размерности A", означает "сгруппировать по всем следующим сочетаниям".

(A, B, . . . , *Z*)
***{A,B,* . . .)**
(A,B)
(A)

Обратите внимание на то, что в общем случае выполняется много отдельных "накоплений по размерности A" (ЭТО зависит от того, какие другие столбцы упомянуты в разделенном запятыми списке ROLLUP). Также отметим, что конструкции GROUP BY ROLLUP *(A, B)* и GROUP BY ROLLUP *(B, A)* имеют различные значения, т.е. конструкция GROUP BY ROLLUP (A, B) не симметрична относительно A и B.

А теперь рассмотрим опцию CUBE. В качестве примера возьмем следующий запрос.

```
SELECT S#, P#, SUM(QTY) AS TOTQTY
FROM   SP
GROUP  BY CUBE ( S#, P# ) ;
```

Конструкция GROUP BY здесь будет логически эквивалентна конструкции, показанной ниже.

```
GROUP  BY GROUPING SETS ( (S#,P#),  (S#),  (P#),  () ) ;
```

Другими словами, этот запрос объединяет формулировки всех четырех запросов 4, 3, 2 и 1 на языке SQL. Результат его выполнения выглядит следующим образом.

S#	P#	TOTQTY
S1	P1	300
S1	P2	200
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S1	*null*	500
S2	*null*	700
S3	*null*	200
S4	*null*	200
null	P1	600
null	P2	1000
null	*null*	1600

Маловыразительное ключевое слово CUBE (куб) было принято к использованию из-за того, что в терминологии технологии OLAP, по крайней мере, в многомерной, значения данных рассматриваются как хранящиеся в ячейках многомерного массива, или *гиперкуба.* В данном случае значения данных являются количественными; *куб* имеет лишь два измерения — измерение поставщиков и измерение деталей (такой *куб* следовало бы назвать плоскостью!); эти два измерения имеют неравные размеры (так что данный *куб даже* не является квадратом; скорее, это— прямоугольник). Но как бы то ни было, конструкция GROUP BY CUBE *(A, в, . . . Z)* означает "сгруппировать по всем возможным подмножествам множества {A, *в,* . . ., Z}".

Любая конструкция GROUP BY может включать произвольные сочетания опций GROUPING SETS,ROLLUP И CUBE.

Перекрестные таблицы

Продукты OLAP часто отображают результаты запросов не в виде таблиц SQL, а в виде перекрестных таблиц (cross tabulation, или сокращенно — crosstab). Вновь рассмотрим запрос 4 ("Определить общее количество поставок по поставщикам и деталям"). Ниже представлены результаты выполнения этого запроса в виде перекрестной таблицы. Отметим, кстати, что количество деталей с номером P1 для поставщиков с номерами S3 и S4 показано (правильно) как нулевое. В языке SQL, напротив, для этого количества было бы получено неопределенное значение (см. главу 19). В действительности, *таблица,* которая формируется в языке SQL в ответ на запрос 4, не содержит строк для сочетания атрибутов (' S3 ', ' P1') или (' S4 ', ' P1'), поэтому попытка создать из нее перекрестную таблицу является не такой уж легко осуществимой.

	P1	P2
S1	300	200
S2	300	400
S3	0	200
S4	0	200

Эта перекрестная таблица, безусловно, обеспечивает более компактный и наглядный способ представления результата выполнения запроса 4. Кроме того, она несколько похожа на реляционную таблицу. Но обратите внимание на то, что *количество столбцов в этой "таблице" зависит от фактических данных,* а точнее, для каждого вида детали имеется один столбец и поэтому структура перекрестной таблицы и смысл строк зависят от фактических данных. Следовательно, перекрестная таблица — это не отношение, а *отчет;* точнее — отчет, который отформатирован в виде простого массива. (Отношение имеет предикат, который можно вывести логически из предикатов отношений, производным от которых он является. Однако *предикат* для перекрестной таблицы (если сделать допущение, что таковой существует) не может быть производным от предикатов соответствующих отношений, поскольку, как мы убедились, структура перекрестной таблицы зависит от фактических *значений* данных.)

О перекрестных таблицах, подобных показанной выше, часто говорят, что они имеют две размерности; в данном случае — поставщиков и деталей. Эти размерности рассматриваются, как если бы они были *независимыми переменными,* тогда *ячейки* на пересечениях размерностей содержат значения соответствующих *зависимых* переменных. Дальнейшие пояснения изложены в подразделе "Многомерные базы данных".

Ниже приведен другой пример перекрестной таблицы, которая представляет собой результат примера применения опции CUBE, рассматриваемой в предыдущем подразделе.

В крайнем справа столбце содержатся итоги по строкам, т.е. итоги для указанных поставщиков по всем деталям. В нижней строке содержатся итоги по столбцам, т.е. итоги для указанных деталей по всем поставщикам. В крайней справа нижней ячейке содержится общий итог, который представляет итог по строке итогов по столбцам или итог по столбцу итогов по строкам.

	P1	P2	*Итого*
S1	300	200	500
S2	300	400	700
S3	0	200	200
S4	0	200	200
Итого	600	1000	1600

Многомерные базы данных

До сих пор предполагалось, что данные OLAP хранятся в обычной базе данных, использующей язык SQL (не считая того, что иногда мы все же касались терминологии и концепции *многомерных баз данных).* Фактически мы, не указывая явно, описывали так называемую систему *ROLAP* (Relational **OLAP—** реляционная **OLAP).** Однако многие считают, что использование системы *MOLAP* (Multi-dimensional **OLAP** — многомерная OLAP) — более перспективный путь. В этом подразделе принципы построения систем MOLAP будут рассмотрены подробнее.

Система MOLAP обеспечивает ведение **многомерных баз данных,** в которых данные концептуально хранятся в ячейках многомерного массива.

Примечание. Хотя выше **и** было сказано о *концептуальном* способе организации хранения, в действительности физическая организация данных в **MOLAP** очень похожа на их логическую организацию.

Поддерживающая СУБД называется *многомерной.* В качестве простого примера можно привести трехмерный массив, представляющий, соответственно, товары, заказчиков и периоды времени. Значение каждой отдельной ячейки может представлять общий объем указанного товара, проданного заказчику в указанный период времени. Как отмечалось выше, перекрестные таблицы из предыдущего подраздела также могут считаться такими массивами.

Если имеется достаточно четкое понимание структуры совокупности данных, то могут быть известны и все связи между данными. Более того, *переменные* такой совокупности (не в смысле обычных языков программирования), грубо говоря, могут быть разделены на **зависимые** и **независимые. В** предыдущем примере *товар, заказчик* и *период времени* можно считать независимыми переменными, а *количество* — единственной зависимой переменной. В общем случае независимые переменные — это переменные, значения которых вместе определяют значения зависимых переменных (точно так же, как, если воспользоваться реляционной терминологией, потенциальный ключ является множеством столбцов, значения которых определяют значения остальных столбцов). Следовательно, независимые переменные задают размерность массива, с помощью которого организуются данные, а также образуют *схему адресации*[11] для данного массива. Значения зависимых переменных, которые представляют фактические данные, сохраняются в ячейках массива.

Примечание. Различие между значениями независимых, или *размерных,* переменных, и значениями зависимых, или *неразмерных,* переменных, иногда характеризуют как различие между *местонахождением* и *содержанием.*

" Поэтому ячейки массива адресуются символически, а не с помощью числовых индексов, которые обычно применяются для работы с массивами.

К сожалению, приведенная выше характеристика многомерных баз данных слишком упрощена, поскольку большинство совокупностей данных изначально остаются *не* изученными в полной мере. По этой причине мы обычно стремимся, в первую очередь, проанализировать данные, чтобы лучше их понять. Часто недостаточное понимание может быть настолько существенным, что заранее невозможно определить, какие переменные являются независимыми, а какие зависимыми. Тогда независимые переменные выбираются согласно текущему представлению о них (т.е. на основании некоторой гипотезы), после чего проверяется результирующий массив для определения того, насколько удачно выбраны независимые переменные (см. раздел 22.7). Подобный подход приводит к тому, что выполняется множество итераций по принципу проб и ошибок. Поэтому система обычно допускает замену размерных и неразмерных переменных, и эту операцию называют *сменой осей координат* (pivoting). Другие поддерживаемые операции включают *транспозицию массива* и *переупорядочение размерностей.* Должен быть также предусмотрен способ добавления размерностей.

Между прочим, из предыдущего описания должно быть ясно, что ячейки массива часто оказываются пустыми (и чем больше размерностей, тем чаще наблюдается такое явление). Иными словами, массивы обычно бывают *разреженными.* Предположим, например, что товар p не продавался заказчику c в течение всего периода времени *t.* Тогда ячейка *[c,p, t]* будет пустой (или в лучшем случае содержать нуль). Многомерные СУБД поддерживают различные методы хранения разреженных массивов в более эффективном, сжатом представлении[12]. К этому следует добавить, что пустые ячейки соответствуют *отсутствующей информации* и, следовательно, системам необходимо предоставлять некоторую вычислительную поддержку для пустых ячеек. Такая поддержка действительно обычно имеется, но стиль ее, к сожалению, похож на стиль, принятый в языке SQL. Обратите внимание на тот факт, что если данная ячейка пуста, то информация или не известна, или не была введена, или не применима, или отсутствует в силу других причин (см. главу 19).

Независимые переменные часто связаны в *иерархии,* определяющие пути, по которым может происходить агрегирование зависимых данных. Например, существует временная иерархия, связывающая секунды с минутами, минуты с часами, часы с сутками, сутки с неделями, недели с месяцами, месяцы с годами. Или другой пример: возможна иерархия композиции, связывающая детали с комплектом деталей, комплекты деталей с узлом, узлы с модулем, модули с изделием. Часто одни и те же данные могут агрегироваться многими разными способами, т.е. одна и та же независимая переменная может принадлежать ко многим различным иерархиям. Система предоставляет операторы для *прохождения вверх* (drill up) и *прохождения вниз* (drill down) по такой иерархии. *Прохождение вверх* означает переход от нижнего уровня агрегирования к верхнему, а *прохождение вниз* — переход в противоположном направлении. Для работы с иерархиями имеются и другие операции, например операция для переупорядочения уровней иерархии.

Примечание. Между операциями *прохождения вверх* (drill up) и *накопления итогов* (roll up) есть одно тонкое различие: операция *накопления итогов* — это операция реализации

[12] Обратите внимание на отличие от реляционных систем. В настоящем реляционном аналоге этого примера в строке *I c , p,* t) не было бы пустой "ячейки" количества, в связи с тем, что строка *(c,p,* t) просто бы отсутствовала. Поэтому при использовании реляционной модели, в отличие от многомерных массивов, нет необходимости поддерживать "разреженные массивы", или скорее "разреженные таблицы", а значит, не требуются искусные методы сжатия для работы с такими таблицами.

требуемых способов группирования и агрегирования, а операция *прохождения вверх*— это операция *доступа* к результатам реализации этих способов. А примером операции *прохождения вниз* может служить такой запрос: "Итоговое количество поставок известно; получить итоговые данные для каждого отдельного поставщика". Безусловно, для ответа на этот запрос должны быть доступными (или вычислимыми) данные более детализированных уровней.

В продуктах многомерных баз данных предоставляется также ряд статистических и других математических функций, которые помогают формулировать и проверять гипотезы (т.е. гипотезы, касающиеся предполагаемых связей). Кроме того, предоставляются инструменты визуализации и генерации отчетов, помогающие решать подобные задачи. Но, к сожалению, для многомерных баз данных пока еще нет никакого стандартного языка запросов, хотя ведутся исследования в целях разработки исчисления, на котором мог бы базироваться такой стандарт [22.31]. Но ничего подобного реляционной теории нормализации, которая могла бы служить научной основой для проектирования многомерных баз данных, пока, к сожалению, нет.

Завершая этот раздел, отметим, что в некоторых продуктах сочетаются оба подхода — ROLAP и MOLAP. Такую *гибридную систему OLAP* называют *HOLAP*. Проводятся широкие дискуссии с целью выяснить, какой из этих трех подходов лучше, поэтому стоит и нам попытаться сказать по данному вопросу несколько слов[13]. В общем случае системы MOLAP обеспечивают более быстрое проведение расчетов, но поддерживают меньшие объемы данных по сравнению с системами ROLAP, т.е. становятся менее эффективными по мере возрастания объемов данных. А системы ROLAP предоставляют более развитые возможности масштабируемости, параллельности и управления по сравнению с аналогичными возможностями систем MOLAP. Кроме того, недавно был дополнен стандарт SQL и в него включены многие статистические и аналитические функции (см. раздел 22.8). Из этого следует, что в настоящее время продукты ROLAP способны к тому же предоставлять расширенные функциональные возможности.

22.7. РАЗРАБОТКА ДАННЫХ

Разработку данных можно охарактеризовать как *исследовательский анализ данных*. Цель такой разработки — поиск интересных зависимостей среди данных, которые впоследствии могут использоваться для выработки стратегии деловой активности или для выявления необычного поведения, например, внезапного возрастания интенсивности использования какой-то кредитной карточки (а это может означать, что она украдена). В инструментах разработки данных используются статистические методы, применяемые к большим объемам хранимых данных, что и позволяет найти интересующие пользователя закономерности.

---

[13] В частности, не следует упускать из виду следующее замечание. Часто приходится слышать, что "таблицы являются плоскими" (т.е. двумерными), а "реальные данные — многомерными", и поэтому отношения не подходят для использования в качестве основы OLAP. Но сторонники этих доводов допускают путаницу между таблицами и отношениями! Как было показано в главе 6, таблицы представляют собой просто изображения отношений, а не отношения как таковые. И хотя верно, что эти изображения двумерны, применительно к отношениям это не верно, поскольку отношения являются n-мерными, где *n* — степень отношения. Точнее, каждый кортеж в отношении с n атрибутами представляет собой точку в n-мерном пространстве, а отношение в целом является множеством таких точек.

Примечание. Слово **большие** здесь нужно выделить особо. Базы для разработки данных часто *чрезвычайно* велики, поэтому очень важно, чтобы применяемые алгоритмы обеспечивали масштабируемость.

Рассмотрим *не* очень большую таблицу с данными о сбыте SALES, показанную на **рис.** 22.5, в которой содержатся данные, касающиеся определенных деловых сделок в системе розничного сбыта[14]. На основе этих данных требуется выполнить *анализ набора потребительских товаров,* где под набором потребительских товаров понимается перечень товаров, приобретаемых во время одной сделки. Благодаря такому анализу можно определить, например, что потребитель, который покупает обувь, вероятно, покупает и носки в составе одной и той же сделки. Эта зависимость между обувью (shoes) и носками (Socks) представляет собой пример **правила связи.** Оно может быть выражено (немного неформально) таким образом.

FORALL tx (Shoes ∈ tx => Socks 6 tx)

Здесь Shoes ∈ tx — антецедент, или *условие правила,* Socks e tx — результат, или *следствие правила,* а переменная tx принимает свои значения среди всех торговых сделок.

SALES

TX#	CUST#	TIMESTAMP	PRODUCT
TX1	C1	d1	Shoes
TX1	C1	d1	Socks
TX1	C1	d1	Tie
TX2	C2	d2	Shoes
TX2	C2	d2	Socks
TX2	C2	d2	Tie
TX2	C2	d2	Belt
TX2	C2	d2	Shirt
TX3	C3	d2	Shoes
TX3	C3	d2	Tie
TX4	C2	d3	Shoes
TX4	C2	d3	Socks
TX4	C2	d3	Belt

Рис. 22.5. Таблица продаж SALES

Введем некоторые дополнительные термины. Множество всех торговых сделок в данном примере называют **совокупностью.** Любое данное правило связи имеет уровень *поддержки* и уровень *достоверности,* или доверительный уровень. Поддержка — это процентная доля совокупности, в которой удовлетворяется правило связи. **Достоверность** — это отношение объема совокупности, в которой удовлетворяется правило связи, к объему совокупности, в которой удовлетворяется условие. (Отметим, что условие и следствие не обязательно должны относиться к одному товару; они могут относиться к любому количеству различных товаров.) Рассмотрим, например, такое правило, касающееся зависимости покупки галстука (Tie) от покупки носков (Socks).

[14] Отметим, что ключом этой таблицы является {TX#, PRODUCT}, данные в таблице удовлетворяют функциональным зависимостям TX#—»CUST# и TX#—>TIMESTAMP, а значит, она не приведена к нормальной форме Бойса-Кодда (БКНФ); версия таблицы, в которой столбец PRODUCT содержал бы значения в виде отношения (с использованием столбца tx# в качестве ключа), *могла* бы находиться в БКНФ и лучше бы подходила для выполнения данных исследований (хотя, возможно, меньше подходила бы для других видов исследования).

```
FORALL  tx  (  Socks    tx => Tie ∈ tx  )
```

По условиям примера, представленного на рис. 22.5, совокупность составляет 4 сделки, поддержка равна 50%, а достоверность— 66,67%.

Более общие правила связи могут быть исследованы с помощью соответствующих результатов агрегирования рассматриваемых данных. Например, после группирования по заказчикам можно проверить допустимость такого правила: "Если заказчик покупает обувь, то, вероятно, он также покупает носки, хотя не обязательно во время той же торговой сделки".

Могут быть определены и другие виды правил. Например, правило **зависимости следствия** может использоваться для определения закономерностей совершения покупок в течение некоторого времени ("Если заказчик купил обувь сегодня, то он, вероятно, купит носки в течение пяти дней"). Правило **классификации** может использоваться для принятия решения по удовлетворению заявки на получение товара в кредит ("Если доход заказчика превышает 75 тыс. долл. в год, то, вероятно, риск неплатежа будет невелик") и т.д. Подобно правилам связей, правила зависимости следствия и правила классификации также имеют уровни поддержки и достоверности.

Разработка данных представляет собой огромную самостоятельную тему [22.2], поэтому очевидно, что рассмотреть ее достаточно подробно в этой книге невозможно. Мы ограничимся кратким описанием вероятного применения методов разработки данных к расширенной версии базы данных поставщиков и деталей. Прежде всего, при отсутствии других источников информации можно использовать логический вывод с помощью нейронной сети для классификации поставщиков по их специализации, например, по крепежным деталям и деталям двигателя, а *предсказание значений* (value prediction) — для прогнозирования того, какими поставщиками и какие детали наиболее вероятно будут поставляться. Затем можно использовать *демографическую кластеризацию,* т.е. разбивку на группы, чтобы связать расходы на поставки с географическим расположением и тем самым закрепить поставщиков за регионами поставки. После этого можно применить *исследование связей,* чтобы определить те детали, которые получены вместе, в одной поставке. С помощью *последовательного обнаружения закономерностей* можно определить, что поставки крепежных деталей обычно следуют за поставками деталей двигателя, а путем обнаружения аналогичных *временных последовательностей* открыть, что имеются сезонные изменения объемов поставок определенных деталей (некоторые из таких изменений происходят осенью, а другие— весной).

22.8. СРЕДСТВА SQL

В первоначально опубликованную версию стандарта SQL: 1999 с самого начала были включены определенные средства OLAP (по сути, дополнения GROUPING SETS, ROLLUP и CUBE к конструкции GROUP BY, которые описаны в разделе 22.7), а через год после публикации этой версии в стандарт было введено много дополнительных средств в форме "Дополнения OLAP" к этому стандарту [22.21]. Изложение исчерпывающих сведений об этом дополнении далеко выходит за рамки данной книги, поэтому остановимся на приведенном ниже кратком описании включенных в него средств.

- Новые числовые функции (например, для вычисления натурального логарифма и антилогарифма, возведения в степень, извлечения квадратного корня, округления в меньшую сторону и округления в большую сторону).

- Новые операции агрегирования (например, для вычисления дисперсии и стан дартного отклонения).
- Функции ранжирования (предоставляющие, например, возможность определить ранг деталей в гипотетическом списке, упорядоченном по весу).
- Кумулятивные функции и другие типы функций вычисления *скользящего среднего* (которые предусматривают использование новой конструкции WINDOW в обычных выражениях SELECT, FROM, WHERE, GROUP BY И HAVING языка SQL).
- Функции анализа распределения, обратного распределения, корреляции и другие статистические функции, применяемые к столбцам, обрабатываемым попарно.

22.9. РЕЗЮМЕ

В этой главе было рассмотрено использование технологии баз данных для систем **поддержки принятия решений.** Основная идея заключается в том, что нужно взять оперативные данные и привести их к виду, в котором их можно было бы применять для оказания помощи управляющему персоналу в понимании особенностей функционирования предприятия.

Сначала были определены понятия систем поддержки принятия решений, которые устанавливаются отдельно от систем оперативных баз данных. Характерная черта баз данных поддержки принятия решений заключается в том, что они предназначены преимущественно (но не полностью) **для чтения.** Как правило, такие базы данных очень велики и имеют **много индексов. В** них обычно присутствует **контролируемая избыточность,** особенно в форме *репликации* и предварительно вычисленных итоговых таблиц. Ключи обычно содержат **временной** компонент, а запросы, как правило, очень **сложны.** Исходя из этих соображений, при проектировании первостепенное внимание уделяется **обеспечению производительности** систем. Признавая важность этой задачи, автор все же считает, что способы ее достижения не должны вступать в противоречие с правильной практикой проектирования. Проблема заключается в том, что в практике проектирования систем поддержки принятия решений обычно недостаточно четко различаются вопросы **логического** и **физического** проектирования.

Затем рассматривались вопросы подготовки оперативных данных к вводу в системы поддержки принятия решений: задачи **извлечения, очистки, преобразования** и **консолидации, загрузки** и **обновления** данных. Также упоминалась концепция **банков оперативных данных,** которые, кроме всего прочего, могут использоваться и как области накопления в процессе подготовки данных. Еще одно применение банков оперативных данных — предоставление поддержки принятия решений на основе текущих данных.

Далее речь шла о **хранилищах данных** и **магазинах данных** (последние могут расцениваться как специализированные хранилища данных). Была рассмотрена основная идея построения **схем типа** "звезда", в которых данные организованы, как большая основная **таблица фактов** и несколько значительно меньших таблиц **размерностей.** В простых случаях схемы типа "звезда" неотличимы от обычных классических нормализованных схем. Но на практике они во многом отходят от принципов классического проектирования по причинам, связанным с необходимостью обеспечения достаточно высокой производительности. (Проблема, опять-таки, состоит в том, что схемы типа "звезда" на самом деле **в** большей степени имеют физическую, а не логическую природу.) Также мы коснулись стратегии реализации операции соединения, известной как *звездообразное соединение,* и разновидности схемы типа "звезда", которая называется **схемой типа "снежинка".**

Кроме того, в этой главе уделено внимание оперативной аналитической обработке данных (OLAP). Обсуждались возможности языка SQL, которые предоставляются с помощью ОПЦИЙ GROUPING SETS, ROLLUP И CUBE КОНСТРУКЦИИ GROUP BY, а именно — возможности осуществления нескольких различных видов агрегирования в одном запросе SQL. Также отмечалось, что язык SQL, к сожалению (с точки зрения автора), допускает возможность объединения результатов этих разных видов агрегирования в одной "таблице", содержащей множество неопределенных значений. В этой главе было также указано, что на практике системы OLAP могут предусматривать преобразование этих "таблиц" в перекрестные таблицы (простые массивы) для их отображения. Затем были кратко описаны многомерные базы данных, в которых данные концептуально хранятся не в таблицах, а в многомерных массивах, или *гиперкубах.* Размерности такого массива составляют независимые переменные (по крайней мере, в отношении этих переменных выдвигается гипотеза, что они являются независимыми), а в ячейках содержатся значения соответствующих зависимых переменных. Независимые переменные обычно связаны в различные иерархии, которые определяют приемлемые способы группирования и агрегирования зависимых данных.

Наконец, была рассмотрена концепция разработки данных. Основная идея состоит в том, что данные часто недостаточно хорошо изучены, поэтому необходимо использовать возможности вычислительных средств, чтобы обнаружить некоторые характерные закономерности во всей совокупности данных и поэтому достичь лучшего их понимания. Здесь кратко рассматривались различные виды *правил,* а именно — правила связи, классификации и зависимости следствия, и обсуждались связанные с ними понятия уровней поддержки и достоверности.

В конце данной главы кратко перечислены средства, которые предусмотрены в дополнении OLAP к стандарту SQL: 1999.

УПРАЖНЕНИЯ

22.1. Назовите некоторые из основных различий между базами данных поддержки принятия решений и оперативными базами данных. Почему для систем поддержки принятия решений и оперативных приложений обычно используются разные хранилища данных?

22.2. Кратко опишите этапы подготовки оперативных данных к вводу в систему поддержки принятия решений.

22.3. Опишите различия между *контролируемой* и *неконтролируемой* избыточностью. Приведите соответствующие примеры. Почему контролируемая избыточность важна в системах поддержки принятия решений? Что происходило бы при использовании неконтролируемой избыточности?

22.4. Опишите различия между *хранилищами данных* и *магазинами данных.*

22.5. Что вы понимаете под термином *схема типа "звезда"?*

22.6. Схемы типа "звезда" обычно не полностью нормализованы. Что служит оправданием такого положения дел? Объясните методологию проектирования таких схем.

22.7. Объясните различия между системами ROLAP и MOLAP.

22.8. Сколькими способами можно агрегировать данные, если они характеризуются четырьмя измерениями, каждое из которых принадлежит к трехуровневой иерархии обобщения (например, город, район, область)?

Используя базу данных поставщиков, деталей и проектов, выразите на языке SQL следующие запросы.

а) Определить количество поставок и средний объем поставок по поставщикам, деталям и проектам, рассматривая их попарно (например, для каждой пары S#-P#, для каждой пары P#-J# и каждой пары J#-s#).

б) Определить максимальный и минимальный объемы поставки по каждому про екту, каждому сочетанию "проект—деталь" и в целом.

в) Определить общие объемы поставок, формируя итоги "по всей размерности поставщиков" и "по всей размерности деталей". *Предупреждение.* Здесь есть ловушка.

г) Определить средний объем поставок по поставщикам, деталям, сочетаниям "поставщик-деталь" и в целом.

Для каждого случая покажите результаты выполнения соответствующего запроса SQL, считая, что обработке подвергаются данные, представленные на рис. 4.5 (см. стр. 163). Кроме того, представьте эти результаты в виде перекрестных таблиц.

В начале раздела 22.6 показан упрощенный вариант таблицы SP, в котором имеется всего 6 строк. Предположим, что эта таблица дополнительно включает следующую строку (которая означает, что поставщик с номером S5, возможно, существует, но в данное время деталей не поставляет).

S5	NULL	NULL

Рассмотрите, какие последствия будут этим вызваны при выполнении всех запросов SQL, приведенных в разделе 22.6.

22.11. Означает ли термин *многомерный* одно и то же, когда он используется во фразах "многомерная схема" и "многомерная база данных"? Объясните свой ответ.

22.12. Прокомментируйте проблему анализа набора потребительских товаров и услуг. Опишите в общих чертах алгоритм для определения правил связи, для которых уровни поддержки и достоверности превышают указанные предельные величины. *Подсказка.* Если некоторые сочетания товаров и услуг "не представляют интере са", поскольку они встречаются в слишком небольшом количестве торговых сде лок, то же самое верно и для всех надмножеств этих сочетаний товаров и услуг.

СПИСОК ЛИТЕРАТУРЫ

Примечание. Термин *представление* (view), который применяется в названиях в **[22.3]**—[22.5], [22.10], [22.12], [22.16], [22.25], [22.28], [22.30] и [22.35], в действительности обозначает не представления, а снимки. Поэтому в аннотациях к этим работам вместо термина *представление* применяется термин *снимки.*

22.1. Adelberg B., Garcia-Molina H., Widom J. The STRIP Rule System for Efficiently Maintaining Derived Data // Proc. 1997 ACM S1GMOD Int. Conf. on Management of Data, Tucson, Ariz. — May 1997.

Здесь STRIP — это сокращение от STanford Real-time Information Processor (станфордская система обработки информации в реальном времени). В этой системе используются *правила* (под этим подразумеваются такие программные конструкции, которые обычно принято называть *триггерами)* для обновления снимков

(называемых в этой работе *производными данными)* после каждого изменения в соответствующих базовых данных. Общим недостатком подобных систем является то, что при частых изменениях базовых данных вычислительные издержки, связанные с выполнением правил, могут стать слишком большими. В этой работе описан метод STRIP, позволяющий уменьшить указанные издержки. Adriaans P., Zantinge D. Data

22.2. Mining. — Reading, Mass.: Addison—Wesley, 1996. Хотя эта книга рекламировалась как краткий обзор для сведения руководящего звена, фактически в ней тема разработки данных раскрыта довольно подробно (и хорошо).

22.3. Afrati F.N., Li C, Ullman J.D. Generating Efficient Plans for Queries Using Views // Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. — May 2001.

22.4. Agrawal D., El Abbadi A., Singh A., Yurek T. Efficient View Maintenance at Data Warehouses // Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz.-May 1997.

Как отмечено в аннотации к [22.12], сопровождение снимков может осуществляться инкрементно, и такое инкрементное сопровождение способствует повышению производительности. Но при инкрементном сопровождении иногда возникают проблемы, если снимки создаются на основе нескольких различных баз данных, которые обновляются одновременно. В этой работе приведено решение указанной проблемы.

Agrawal S., Chaudhuri S., Narasayya V. Automated Selection of Materialized Views and Indexes for SQL Databases // Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt. — September 2000. Alter S. Decision Support Systems: Current Practice and Continuing Challenges. — Reading, Mass.: Addison-Wesley, 1980. Bennett J.L. (ed.) Bilding Decision Support Systems. — Reading, Mass.: Addison-Wesley, 1981.

Bonczek R.H., Holsapple C.W., Whinston A. Fondations of Decision Support Systems. — Orlando, Fla.: Academic Press, 1981. Одна из первых публикаций в защиту строго методического подхода к созданию систем поддержки принятия решений. Особое внимание уделено описанию роли моделирования (в общем смысле эмпирического и математического моделирования) и науки управления.

22.9 Bontempo C.J., Saracco CM. Database Management: Principles and Products. — Upper Saddle River, N.J.: Prentice-Hall, 1996.

22.10. Chirkova R., Halevy A.Y., Suciu D. A Formal Perspective on the View Selection Problem // Proc. 27th Int. Conf. on Very Large Data Bases, Rome, Italy. — September 2001. 22.11. Codd E.F., Codd S.B., Salley C.T. Providing OLAP (Online Analytical Processing) to User-Analysts: An IT Mandate, 1993. Предоставляется в Arbor Software Corp. Как указано в разделе 22.6, благодаря этой статье появился термин *OLAP,* хотя и не само понятие. *Примечание.* В начале статьи категорически утверждается, что "Существует потребность НЕ в создании еще одной технологии баз данных, а скорее в разработке надежных... инструментальных средств анализа". Далее следуют описание **и**

доводы в пользу новой технологии баз данных (?!) — с новым концептуальным представлением данных, новыми операторами (как для обновления, так и для выборки), многопользовательской поддержкой (включая возможности защиты и параллельного доступа), новыми структурами памяти и новыми возможностями оптимизации. Одним словом, новая модель данных и новая СУБД.

22.12. Colby L.S. et al. Supporting Multiple View Maintenance Policies // Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. — May 1997.

Как правило, применяются три основных подхода к *сопровождению снимков:* немедленное сопровождение (каждое обновление в любой базовой переменной отношения немедленно активизирует соответствующее обновление снимка), отложенное сопровождение (снимок обновляется только тогда, когда в систему поступает запрос к этому снимку) и периодическое сопровождение (снимок обновляется через заданные промежутки времени, скажем, один раз в сутки). В общем снимки предназначены для повышения производительности запросов за счет снижения производительности обновления, и эти три подхода к сопровождению представляют собой весь спектр компромиссных решений между этими двумя крайностями. В данной работе рассматриваются проблемы, связанные с одновременным применением различных подходов к сопровождению разных снимков в одной и той же системе.

22.13. Date C.J. We Don't Need Composite Columns // Date C.J., Darwen H., McGoveran D. Relational Database Writings 1994-1997. — Reading, Mass.: Addison-Wesley, 1998.

Понятие *составных столбцов* упоминается в разделе 22.3, а в этой короткой статье данное понятие рассматривается более подробно. Сам заголовок данной работы указывает на тот факт, что в прошлом предпринимались ошибочные попытки вве сти поддержку составных столбцов, не основываясь на поддержке типов, опреде ляемых пользователем. Если предоставлена соответствующая поддержка пользо вательских типов, то вопрос о составных столбцах полностью отпадает.

22.14. Delvin B. Data Warehouse from Architecture to Implementation. — Reading, Mass.: Addison-Wesley, 1997.

22.15. Delvin B.A., Murphy P.T. An Architecture for a Business and Information System // IBMSys.J. -1988. -27, № 1.

Первая опубликованная статья, в которой определен и использован термин *информационное хранилище.*

22.16. Goldstein J., Larson P.-E. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution // Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. — May 2001.

22.17. Gray J., Bosworth A., Layman A., Pirahesh H. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals // Proc. 12th IEEE Int. Conf. on Data Engineering. — New Orleans, La. — February 1996.

В статье впервые предлагается добавить в конструкцию языка SQL GROUP BY такие опции,как CUBE.

22.18. Inmon W.H. Data Architecture: The Information Paradigm. — Wellesley, Mass.: QED Information Sciences, 1988.

В работе обсуждается происхождение понятия *хранилище данных* и описывается, как хранилище данных могло бы выглядеть на практике. Термин *хранилище данных* впервые появился в этой книге.

22.19. Inmon W.H. Building The Data Warehouse. — New York, N.Y.: John Wiley & Sons, 1992.

Первая книга, посвященная хранилищам данных. В ней определяется этот термин и обсуждаются ключевые проблемы, которые возникают при разработке хранилищ данных. В книге в первую очередь обосновывается концепция хранилищ данных, а также рассматриваются вопросы эксплуатации и физического проектирования.

22.20. Inmon W.H., Hackathorn R.D. Using the Data Warehouse.— New York, N.Y.: John Wiley & Sons, 1994,

В книге обсуждается роль пользователей и администраторов хранилища данных. Как и другие книги по этой теме, она в основном посвящена вопросам физического проектирования. Достаточно подробно обсуждается понятие хранилищ операционных данных.

22.21. International Organization for Standardization (ISO). SQL/OLAP, Document ISO/IEC 9075-1:1999/Amd. 1:2000(E).

Учебное руководство по материалам данного документа можно найти в [26.32].

22.22. Keen P.G.W., Morton M.S.S. Decision Support Systems: Ал Organizational Perspective. — Reading, Mass.: Addison-Wesley, 1978.

Это классическое изложение — одно из самых ранних, если не *самое* раннее, которое явно посвящено поддержке принятия решений. Публикация ориентирована на анализ поведения и охватывает вопросы анализа, проектирования, реализации, оценки и разработки систем поддержки принятия решений.

Kiessling W. Foundations of Preferences in Database Systems, Preference SQL — Design, Implementation, Experiences // Proc. 28th Int. Conf. on Very Large Data Bases, Hong Kong. — August 2002.

Пожелания (preference), упомянутые в названиях этих работ, позволяют пользователю формировать *нечеткие запросы* (например, "Найдите для меня хороший китайский ресторан, не слишком дорогой, желательно за городом").

22.24. Kimball R. The Data Warehouse Toolkit. — New York, N.Y.: John Wiley & Sons, 1996.

Эта книга — руководство к действию. Как и гласит подзаголовок: "Практические методы создания многомерных хранилищ данных", в ней основное внимание уделяется практическим, а не теоретическим вопросам. По умолчанию в ней предполагается, что не существует значительных различий между логическим и физическим уровнями систем.

22.25. Kotidis Y., Roussopoulos N. A Case for Dynamic View Management // ACM TODS. — December 2001. — 26, № 4.

22.26. Morton M.S.S. Management Decision Systems: Computer-Based Support for Decision Making. — Harvard University, Division of Research, Graduate School of Business Administration, 1971.

Это классическая статья, в которой было введено понятие систем поддержки управленческих решений и поддержка принятия решений была явно обозначена как относящаяся к сфере применения компьютерных систем. Создана конкретная

система управленческих решений для координации производственного планирования поставок оборудования для прачечных. Затем она была подвергнута научно обоснованной проверке, в которой в качестве пользователей участвовали руководители маркетинговых и производственных отделов.

22.27. Parsaye K., Chignell M. Intelligent Database Tools and Applications. — New York, N.Y.: John Wiley & Sons, 1993.

Это первая книга, посвященная принципам и методам разработки данных, хотя сами авторы определяют тему своей работы как *интеллектуальные базы данных.*

22.28. Pottinger R., Levy A. A Scalable Algorithm for Answering Queries Using Views // Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt. — September 2000.

22.29. Quass D., Widom J. On-Line Warehouse View Maintenance // Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data, Tucson, Ariz. — May 1997.

В работе представлены алгоритмы сопровождения снимков, позволяющие выполнять транзакции сопровождения одновременно с запросами к сопровождаемым снимкам.

22.30. Salem K., Beyer K., Lindsay B., Cochrane R. How to Roll a Join: Asynchronous Incremental View Maintenance // Proc. 2000 ACM SIGMOD Int. Conf. on Management of Data, Dallas, Tex. — May 2000.

22.31. Thomsen E. OLAP Solutions: Building Multi-Dimensional Information Systems (2nd ed.). - New York, N.Y.: John Wiley & Sons, 2002.

Одна из первых книг по оперативной аналитической обработке данных (OLAP) и, возможно, наиболее полная. Она в основном посвящена определению концепций и методов анализа с использованием многомерных систем. Эту книгу можно считать серьезной попыткой внести определенный порядок в описание этой сложной темы.

22.32. Uthurusamy R. From Data Mining to Knowledge Discovery: Current Challenges and Future Directions // Fayyad U.M., Piatetsky-Shapiro G., Smyth P., Uthurusamy R. (eds.) Advances in Knowledge Discovery and Data Mining. — Cambridge, Mass.: AAAI Press/MIT Press, 1996.

22.33. Valduriez P. Join Indices//ACM TODS. - June 1987. - 12, № 2.

22.34. Zaharioudakis M. et al. Answering Complex SQL Queries Using Automatic Summary Tables// Proc. 2000 ACM SIGMOD Int. Conf. on Management of Data, Dallas, Tex. — May 2000.

22.35. Zhuge Y., Garcia-Molina H., Hammer J., Widom J. View Maintenance in a Warehousing Environment // Proc. 1995 ACM SIGMOD Int. Conf. on Management of Data, San Jose, Calif. — May 1995.

После получения сообщения о том, что произошло обновление некоторых базовых данных, на узле хранилища данных может возникнуть необходимость передать запрос на узел с базовыми данными и только после этого произвести необходимую операцию сопровождения снимка. В связи с этим возникает запаздывание по времени между первоначальным обновлением базовых данных и выдачей такого запроса, что иногда приводит к возникновению различных аномалий. В данной работе описан алгоритм, позволяющий устранить указанные аномалии.

ГЛАВА 23

Хронологические базы данных

23.1. ВВЕДЕНИЕ

Хронологическая база данных может быть неформально определена как база, которая содержит исторические данные[1] наряду с текущими данными или вместо них (в качестве наглядного примера такой базы данных можно указать хранилище данных; см. главу 22). Обычные, или нехронологические, базы данных содержат только текущие данные; актуальность таких баз поддерживается путем обновления данных сразу же после того, как представленные в них высказывания становятся ложными. В отличие от них, хронологические базы данных обновляются очень редко (а могут вообще не обновляться), если не считать выполнения операций INSERT, которые применяются для их первоначального заполнения. Например, рассмотрим базу данных поставщиков и деталей. Если в ней находятся значения, обычно используемые в данной книге в качестве примеров, то эта база данных, кроме всего прочего, показывает, что статус поставщика S1 (под этим подразумевается тот статус, каковым он является "в настоящее время") равен 20. Но в хронологической версии этой базы данных может быть показано не только то, что в настоящее

[1] Хронологические базы данных могут содержать данные, относящиеся не только к прошлому, но и к будущему, поэтому термин "исторический" следует понимать как охватывающий и эту возможность.

время статус поставщика S1 равен 20, но и то, что он был равен 20 с 1 июля прошлого года, а также, возможно, что он был равен 15 с 5 апреля до 30 июня прошлого года и т.д.

Исследования в области хронологических баз данных интенсивно проводились, по меньшей мере, с начала 1980-х годов и предусматривали в значительной степени изучение самой природы времени. Ниже перечислены некоторые проблемы, которые рассматривались в этих исследованиях.

- Имеет ли время начало или конец?
- Является ли время непрерывным или делится на дискретные кванты?
- Как можно лучше всего охарактеризовать важное понятие "текущего времени" (иногда определяемое как "движущаяся по временной шкале позиция текущего времени")?

Но подобные проблемы фактически не относятся к проблематике баз данных как таковой, поэтому они не будут рассматриваться в данной главе слишком подробно; вместо этого мы просто будем принимать по ходу изложения разумные предположения и сосредоточим свое внимание на задачах, которые непосредственно касаются стоящей перед нами общей цели.

Выше в данной книге было указано, что любые данные в целом можно рассматривать как представляющие истинные высказывания. Из этого следует, что хронологические данные, в частности, можно рассматривать как представляющие истинные высказывания с обозначением времени; под этим подразумеваются высказывания, которые включают по меньшей мере один фактический параметр с временной отметкой того или иного типа. Например, рассмотрим следующий кортеж.

S#	SINCE
S1	July 1, 2003

Вполне очевидно, что этот кортеж содержит атрибут с номером поставщика s# и атрибут с временной отметкой SINCE, а соответствующее ему высказывание с временной отметкой выглядит, как показано ниже.

Поставщик S1 работал по контракту от 1 июля 2003 года.

Безусловно, при этом предполагается, что данное высказывание представляет собой конкретизацию предиката[2] в следующей форме: "Поставщик s# работал по контракту от даты SINCE". Мы объясним чуть позже, почему выражение "от" в этом предикате (и в примере конкретизации) выделено полужирным шрифтом. Но вначале рассмотрим еще один пример.

S#	FROM	TO
S1	May 1, 2002	April 30, 2003

Этот кортеж включает атрибут номера поставщика s# и два атрибута с временными отметками, FROM и то; соответствующее высказывание с временными отметками приведено ниже.

---

[2] Во всей данной главе неуточненный термин "предикат" используется для обозначения того понятия, которое в главе 9 именовалось *внешним предикатом* или *предикатом, определяемым пользователем.*

Поставщик S1 работал по контракту на протяжении интервала времени от 1 мая 2002 года до 30 апреля 2003 года.

На этот раз предполагается, что данное высказывание стало конкретизацией предиката в следующей форме: "Поставщик s# работал по контракту на протяжении интервала времени от даты FROM до даты то". Еще раз отметим, что причины применения полужирного шрифта будут описаны чуть позже.

Как показывают эти два примера, такие выражения, как "от" и "на протяжении интервала времени", приобретают исключительно большую важность при описании данных, которые привязаны к временным интервалам (фактически они охватывают все возможные при этом ситуации). Но для того чтобы соответствующие понятия могли быть действительно полезными, необходимо определить их смысл очень точно (намного точнее, чем в неформальном общении!). Ниже приведены определения указанных понятий.

1. Выражение "от" рассматривается как обозначающее именно "начиная от указан ной позиции на временной шкале и не включая непосредственно предшествую щего времени". Итак, если сказано, что поставщик si работал по контракту от 1 июля 2003 года, то под этим подразумевается, что, во-первых, поставщик si ра ботал по контракту именно от 1 июля 2003 года вплоть до текущей даты (неза висимо от того, какой может быть эта текущая дата), кроме того, во-вторых, 30 июня 2003 года поставщик si еще не работал по контракту.
2. Выражение "на протяжении интервала времени" рассматривается как обозна чающее "на протяжении указанного интервала времени, не включая того времени, которое непосредственно предшествует или непосредственно следует за этим ин тервалом". Поэтому, если сказано, что поставщик S1 работал по контракту на протяжении интервала времени от 1 мая 2002 года до 30 апреля 2003 года, под этим подразумевается, во-первых, что поставщик S1 работал по контракту в течение всего интервала времени от 1 мая 2002 года до 30 апреля 2003 года включительно[3] и, кроме того, во-вторых, что 30 апреля 2002 года поставщик S1 еще не работал по контракту, а 1 мая 2003 года уже не работал по контракту.

В приведенных выше высказываниях и предикатах выражения "от" и "на протяжении интервала времени" были выделены полужирным шрифтом для обозначения того факта, что эти термины используются в указанном выше чрезвычайно строгом смысле. Но в дальнейшем изложении полужирный шрифт применяться не будет.

Некоторые фундаментальные допущения

В этой главе уже упоминались как интервалы времени, так и определенные позиции на временной шкале, а в данном разделе будут даны определения этих понятий или, по меньшей мере, описаны некоторые фундаментальные допущения, на которых основаны эти понятия. Прежде всего, предполагается, что само время может рассматриваться как временная шкала, состоящая из конечной последовательности дискретных, неделимых *квантов времени,* где квант времени, в свою очередь, является наименьшей единицей времени,

[3] Во всей данной главе принята так называемая "закрыто-закрытая" интерпретация интервалов времени, согласно которой интервал времени от Ь до е рассматривается как включающий и "начальную позицию на временной шкале" Ь, и "конечную позицию на временной шкале" е. Отметим без дополнительных комментариев, что в литературе можно найти другие интерпретации интервалов времени.

которая может быть представлена в используемой компьютерной системе. Иными словами, даже если время в реальном мире непрерывно и бесконечно, в рассматриваемой модели оно представлено как дискретное и конечное.

Примечание. Здесь обнаруживается очевидная аналогия с тем, что в обычно применяемой модели вычислений вещественные числа представлены с помощью рациональных.

Затем необходимо тщательно провести различие, с одной стороны, между квантами времени как таковыми, которые являются (согласно приведенному выше определению) минимальными единицами времени, представимыми в данной компьютерной системе, и, с другой стороны, единицами измерения времени, наиболее подходящими для некоторой конкретной цели (таковыми могут быть сутки, месяцы, миллисекунды и т.д.). В частности, в приведенных выше примерах, касающихся поставщиков, безусловно, нас интересуют только такие значения времени, которые измеряются с точностью до одних суток. В этой главе единицы измерения времени, подходящие для какой-то определенной цели, будут именоваться **позициями** на **временной шкале** (или сокращенно временными позициями), поскольку это позволяет подчеркнуть тот факт, что для рассматриваемой цели они также считаются неделимыми. Таким образом, можно неформально определить *временную позицию* как "отрезок временной шкалы"; под этим подразумевается множество квантов времени от одного "граничного" кванта до другого (например, от полуночи одних суток до полуночи других). Поэтому можно утверждать (опять-таки, неформально), что временнь/е позиции имеют определенную продолжительность (в данном примере одни сутки). Но с формальной точки зрения временнь/е позиции действительно можно рассматривать как некие дискретные позиции на временной шкале — они неделимы и к ним понятие продолжительности, строго говоря, не применимо.

Примечание. Невзирая на сказанное в предыдущем абзаце, в данной главе иногда будет использоваться термин *степень детализации*, с помощью которого неформально обозначается "размер" или продолжительность соответствующих временных позиций, или, равным образом, "размер" или продолжительность промежутка между двумя смежными временными позициями. Таким образом, можно считать, что в этом примере степень детализации составляет одни сутки, а это означает, что в данном контексте исключаются наши обычные представления о том, что сутки состоят из часов, которые состоят из минут, и т.д. Указанные понятия могут быть выражены только в терминах более низких уровней детализации.

Поэтому для некоторой определенной цели временная шкала может рассматриваться как конечная последовательность временнь/х позиций (а не квантов времени); безусловно, что такая последовательность является хронологической. А из того, что она конечна, следуют приведенные ниже выводы.

- Последовательность временнь/х позиций имеет начало и конец. Иными словами, в последовательности имеется уникальная начальная позиция, которая соответству ет началу отсчета времени, и уникальная конечная позиция, соответствующая **кон цу** отсчета времени.
- Каждая временная позиция в последовательности, отличная от временной пози ции, соответствующей началу отсчета времени, имеет уникальную **предшествую щую** временную позицию, и каждая временная позиция в последовательности, от личная от временной позиции, соответствующей концу отсчета времени, имеет уникальную последующую временную позицию.

Интервал времени определяется как непустой отрезок временной шкалы[4]. Говоря точнее, интервал времени, имеющий начальную временную позицию Ь и конечную временную позицию в, может рассматриваться как подпоследовательность временной шкалы, состоящая из всех временных позиций р, таких что b < p < e (где "<" означает "раньше чем").

Рабочий пример

Если не указано иное, примеры, рассматриваемые в данной главе, будут основаны на существенно упрощенной версии базы данных поставщиков и деталей, которая здесь именуется как "база данных поставщиков и поставок". Конкретные сведения об этой версии приведены ниже.

1. Переменная отношения деталей P полностью исключена.
2. Переменная отношения поставщиков S упрощена путем удаления всех атрибутов, кроме s#. Предикат для этой пересмотренной (и существенно упрощенной!) переменной отношения состоит в следующем утверждении.

Поставщик s# в настоящее время работает по контракту.

3. Атрибут QTY из переменной отношения поставок SP удален, и эта пересмотренная переменная отношения интерпретируется следующим образом.

Поставщик s# в настоящее время способен поставлять деталь P#.

Иными словами, вместо представления фактических поставок деталей поставщиками эта упрощенная версия переменной отношения SP представляет то, что можно было бы назвать *потенциальными поставками;* это означает, что она показывает способность определенных поставщиков поставлять определенные детали.

Примечание. Несмотря на это изменение смысла, автор все еще находит приемлемым использование в последующем изложении неуточненного термина "поставки".

На рис. 23.1 показано множество значений, применяемых в качестве примера в этой упрощенной базе данных.

Примечание. Эта база данных, безусловно, все еще является полностью обычной базой данных — в ней до сих пор вообще не учитываются хронологические аспекты.

Теперь рассмотрим некоторые ограничения и запросы для этой базы данных. В следующем разделе будет показано, что произойдет с этими ограничениями и запросами после расширения базы данных для включения различных хронологических средств.

Ограничения (первоначальная база данных). Единственные ограничения, которые желательно учесть, приведены ниже.

---

[4] Автор должен предупредить читателей, знакомых с языком SQL, что определяемое здесь понятие интервалов времени (interval) не имеет ничего общего с тем, как интервалы времени трактуются в языке SQL — последние вообще не представляют собой интервалы времени в обычном смысле этого понятия, а скорее служат для обозначения продолжительности (например, "3 суток").

- {S#} и {S#,P#} являются первичными ключами, соответственно, для переменных отношения[5] S и SP.
- {S#} является внешним ключом в переменной отношения SP, который ссылается на первичный ключ переменной отношения S.

Запросы (первоначальная база данных). Здесь рассматриваются только приведенные ниже два запроса.

- Запрос A. Получить номера поставщиков, которые в настоящее время способны поставлять по меньшей мере одну деталь.

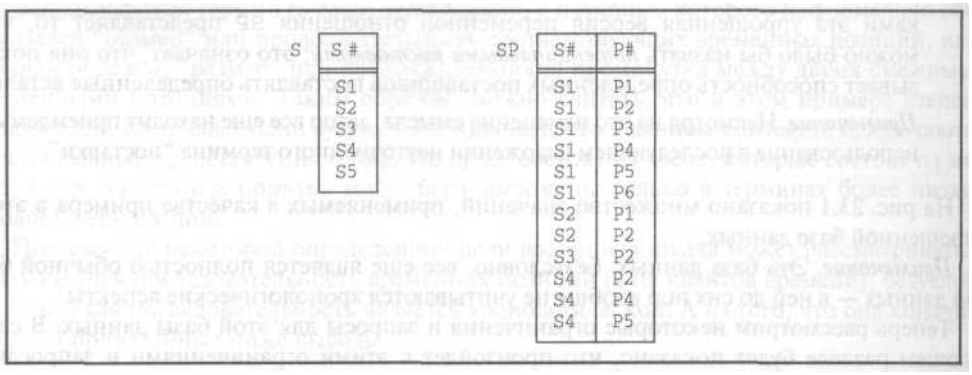
  SP { S# }

- Запрос в. Получить номера поставщиков, которые в настоящее время не способны поставлять вообще ни одной детали.

  S { S# } MINUS SP { S# }

Обратите внимание на то, что в запросе A требуется выполнить простую проекцию, а в запросе в — операцию разности над двумя такими проекциями. При рассмотрении хронологических версий этих запросов в разделе 23.5 станет очевидно, что в них необходимо применять "хронологические" (или, по меньшей мере, обобщенные) версии двух указанных операций, поэтому читатель вряд ли будет удивлен, когда узнает в том же разделе, что могут быть также определены обобщенные версии других реляционных операций.

В завершение этого довольно продолжительного вводного раздела представим план остальной части главы. Прежде всего, в разделе 23.2 показано, почему для хронологических данных чаще всего требуется особая трактовка.



S

S#
S1
S2
S3
S4
S5

SP

S#	P#
S1	P1
S1	P2
S1	P3
S1	P4
S1	P5
S1	P6
S2	P1
S2	P2
S3	P2
S4	P2
S4	P4
S4	P5

Рис. 23.1. База данных поставщиков и поставок (исходная версия) и примеры значений

Затем в разделе 23.3 описано, какие последствия связаны с применением такой трактовки, что интервалы времени рассматриваются как самостоятельные значения, а не как пары, состоящие из начальных и конечных значений; в частности, в этом разделе определен

[5] На протяжении всей данной главы для определенности предполагается, что переменные отношения имеют именно первичные ключи. Но в целом мы не стремимся проводить различия между первичными и альтернативными ключами в определениях переменных отношения на языке Tutorial D, а рассматриваем их все просто как потенциальные ключи.

целый ряд операций для работы с такими интервалами времени. В разделе 23.4 рассматриваются два чрезвычайно важных реляционных оператора, PACK и UNPACK. После этого в разделе 23.5 приведено описание обобщенных версий знакомых операций реляционной алгебры; наконец, разделы 23.6 и 23.7, соответственно, посвящены проблемам проектирования баз данных и применения ограничений целостности.

23.2. ОБЩАЯ ПОСТАНОВКА ПРОБЛЕМЫ

На первом этапе преобразования базы данных поставщиков и поставок в хронологическую форму осуществляется (так сказать) *полухронологизация* (т.е. введение информации только о начальной позиции на временной шкале) переменных отношения S и SP путем добавления атрибута с временной отметкой SINCE к каждой из этих переменных отношения и переименования их соответствующим образом (рис. 23.2).

S_SINCE

S#	SINCE
S1	d04
S2	d07
S3	d03
S4	d04
S5	d02

SP_SINCE

S#	P#	SINCE
S1	P1	d04
S1	P2	d05
S1	P3	d09
S1	P4	d05
S1	P5	d04
S1	P6	d06
S2	P1	d08
S2	P2	d09
S3	P2	d08
S4	P2	d06
S4	P4	d04
S4	P5	d05

Рис. 23.2. База данных поставщиков и поставок (полухронологическая версия) и примеры значений

Для упрощения на рис. 23.2 не показаны настоящие временные отметки; вместо этого на данном рисунке используются символические обозначение в форме dOl, dO2 и т.д., где "d" — сокращение от "day" (сутки); этого соглашения мы будем придерживаться на протяжении всей данной главы. (В большинстве рассматриваемых здесь примеров используются временнь/е позиции, представляющие собой именно сутки, поэтому соответствующая степень детализации в этих примерах составляет одни сутки.) Предполагается, что первые сутки непосредственно предшествуют вторым суткам, вторые сутки непосредственно предшествуют третьим суткам и т.д.; кроме того, незначащие ведущие нули в таких выражениях, как "первые сутки", не будут указаны (примеры таких выражений нам только что встретились; в них не применяются, допустим, слова "нуль-первые сутки").

Ниже приведены предикаты для переменных отношения S_SINCE и SPSINCE.

- S_SINCE. Поставщик s# работал по контракту от суток SINCE.
- SP_SINCE. Поставщик s# был способен поставлять деталь P# от суток SINCE.

Ограничения (полухронологическая база данных). Первичный и внешний ключи для полухронологической базы данных, показанной на рис. 23.2, являются такими же, как и

в исходной базе данных, приведенной на рис. 23.1. Поэтому определения переменных отношения могут выглядеть следующим образом.

```
VAR S SINCE BASE RELATION { S# S#, SINCE
    DATE } KEY { S# } ;

VAR SP SINCE BASE RELATION { S# S#, P# P#, SINCE
    DATE } KEY { S#, P# } FOREIGN KEY { S# }
    REFERENCES S_SINCE ;
```

Применяемый здесь тип DATE представляет грегорианские даты; под этим подразумеваются даты, определяемые с точностью до суток и ограничиваемые правилами грегорианского календаря (из этого, кроме всего прочего, следует, что, например, даты "31 апреля 2005 года" и "29 февраля 2100 года" не являются допустимыми).

Но кроме ограничения внешнего ключа, которое формирует связь между SP_SINCE и S_SINCE, необходимо дополнительное ограничение для обозначения того факта, что ни один поставщик не может поставлять какие-либо детали до тех пор, пока не начнет работать по контракту, как показано ниже.

```
    CONSTRAINT XST1 /* XST1 - сокращение от extra semitemporal
                                              constraint no. 1 */
/* (дополнительное полухронологическое ограничение № 1)     */
   IS_EMPTY  (  ( S_SINCE RENAME SINCE AS SS )
             JOIN ( SP_SINCE RENAME SINCE AS SPS ) )
             WHERE SPS < SS ) ;
```

Это ограничение можно описать словами следующим образом: "Если кортеж sp в переменной отношения SP_SINCE ссылается на кортеж s в переменной отношения S_SINCE, то значение SINCE в кортеже sp не может быть меньше, чем соответствующее значение в кортеже s". На основании этого примера можно приступить к определению важной проблемы — если приходится иметь дело с такой "полухронологической" базой данных, которая показана на рис. 23.2, то, по-видимому, придется сформулировать множество ограничений такого же общего и довольно странного вида, как ограничение XST1 и поэтому вскоре возникнет необходимость в использовании для этой цели некоторого удобного сокращения.

Запросы (полухронологическая база данных). Теперь рассмотрим приведенные ниже полухронологические аналоги запросовА и в.

- Запрос А. Получить номера поставщиков, которые в настоящее время способны поставлять по меньшей мере одну деталь, показывая в каждом случае дату, начиная от которой они получили такую возможность.

Если поставщик Sx в настоящее время способен поставлять несколько разных деталей, это означает, что поставщик Sx приобрел способность поставлять по меньшей мере одну деталь, начиная от самой ранней даты SINCE, показанной для Sx в переменной отношения SP_SINCE (например, если номер поставщика Sx равен S1, то самой ранней датой SINCE является d(4). Поэтому в формулировке этого запроса можно применить следующее выражение.

```
SUMMARIZE SP BY { S# } ADD MIN ( SINCE ) AS SINCE
```

Полученный результат выглядит следующим образом.

S#	SINCE
S1	d04
S2	d08
S3	d08
S4	d04

- Запрос в. Получить номера поставщиков, которые в настоящее время не способны поставлять ни одной детали вообще, показав в каждом случае дату, начиная от которой они стали неспособными выполнять поставку.

В рассматриваемом примере данных имеется только один поставщик (а именно поставщик S5), который в настоящее время не способен поставлять какие-либо детали вообще. Но мы не можем определить дату, начиная от которой S5 не имеет возможности поставлять какие-либо детали, поскольку в этой базе данных нет достаточной информации (еще раз повторяем, что эта база данных — лишь полухронологическая, т.е. содержит информацию только о начальных временных позициях). Например, предположим, что сегодня — десятые сутки. В таком случае может сложиться такая ситуация, что поставщик S5 был способен поставлять по меньшей мере одну деталь, начиная от вторых суток, когда с поставщиком S5 был заключен контракт, и заканчивая (самое позднее) девятыми сутками, когда этот контракт был расторгнут. Можно также рассматривать другую крайность, когда имеет место случай, что S5 вообще не имел возможности когда-либо что-либо поставлять.

Для того чтобы получить возможность формировать ответы на запрос в, необходимо завершить *хронологизацию* всей базы данных или, по меньшей мере, той части, которая касается переменной отношения SP; точнее, необходимо предусмотреть возможность хранения в базе данных исторических записей, которые показывают, какой поставщик был способен поставлять те или иные детали и когда это было (рис. 23.3).

S_FROM_TO

S#	FROM	TO
S1	d04	d10
S2	d02	d04
S2	d07	d10
S3	d03	d10
S4	d04	d10
S5	d02	d10

SP_FROM_TO

S#	P#	FROM	TO
S1	P1	d04	d10
S1	P2	d05	d10
S1	P3	d09	d10
S1	P4	d05	d10
S1	P5	d04	d10
S1	P6	d06	d10
S2	P1	d02	d04
S2	P1	d08	d10
S2	P2	d03	d03
S2	P2	d09	d10
S3	P2	d08	d10
S4	P2	d06	d09
S4	P4	d04	d08
S4	P5	d05	d10

Рис. 23.3. База данных поставщиков и поставок (первая полностью хронологическая версия, в которой используются явно заданные атрибуты

Сравнивая рис. 23.3 с рис. 23.2, можно обнаружить, что атрибуты SINCE стали атрибутами FROM, а каждая переменная отношения приобрела дополнительный атрибут с временной отметкой, называемой то (при этом суффикс _SINCE в именах переменных отношения был, соответственно, заменен суффиксом _FROM_TO). Атрибуты FROM и TO вместе выражают понятие интервала времени, в течение которого некоторое высказывание было истинным.

Примечание. Для определенности предположим, что сегодня — десятые сутки и поэтому в каждом кортеже, который отражает текущее состояние дел, в качестве значения TO было показано dlO. Но это предположение может (а по сути должно!) немедленно натолкнуть читателя на размышления о том, какой механизм мог бы обеспечить замену всех этих значений dlO значениями dll после полуночного удара часов, который произойдет по окончании десятых суток. К сожалению, пока мы вынуждены отложить обсуждение этого вопроса. Мы вернемся к нему в разделе 23.6.

Следует отметить, что теперь количество кортежей в базе данных увеличилось, поскольку в ней ведутся исторические записи; в действительности, полностью хронологическая база данных, приведенная на рис. 23.3, включает всю информацию из полухронологической базы, показанной на рис. 23.2, не считая того, что исключительно ради примера в последней версии базы данных показано значение то для двух поставок поставщика S4 по состоянию на дату, предшествующую текущей дате (т.е. эти две поставки были преобразованы из "текущей" информации в "историческую"). База данных, приведенная на рис. 23.3, включает также историческую информацию, касающуюся более раннего интервала времени, от dO2 до dO4, в течение которого поставщик S2 работал по ранее заключенному контракту и был способен поставлять некоторые детали.

Ниже описаны предикаты хронологической версии рассматриваемой базы данных.

- S_FROM_TO. Поставщик S# работал по контракту на протяжении интервала времени от суток FROM ДО суток то.
- SP_FROM_TO. Поставщик s# был способен поставлять деталь p# на протяжении интервала времени от суток FROM до суток то.

Теперь необходимо сделать еще несколько замечаний, касающихся данного рабочего примера, после того как он был полностью хронологизирован. А именно, в дальнейшем предполагается, что справедливы приведенные ниже (достаточно реалистичные) допущения (которые связаны с тем, что в этом примере базы данных отсутствуют данные о номерах контрактов).

1. Ни один поставщик не может закончить работу по одному контракту в одни сутки **и** начать работу по другому контракту на следующие же сутки.
2. Ни один поставщик не может работать одновременно по двум разным контрактам.
3. Контракты поставщиков могут не иметь обусловленной даты завершения; это оз начает, что поставщик имеет возможность в настоящее время работать по опреде ленному контракту, притом что дата завершения этого контракта в настоящее время остается неизвестной.

Ограничения (первая полностью хронологическая база данных). Прежде всего, как показывает двойная линия на рис. 23.3, теперь атрибут FROM включен в состав первичного ключа для обоих переменных отношения. Безусловно, очевидно, что первичным

ключом для переменной отношения S_FROM_TO (например) не может быть просто {S#}, поскольку, если бы это было так, то мы не могли бы иметь дело с такими поставщиками, как S2, которые работали по контракту на протяжении двух или нескольких отдельных интервалов времени. Аналогичное замечание относится и к переменной отношения SP_FROM_TO.

Примечание. Вместо атрибутов FROM в состав первичных ключей можно было ввести атрибуты TO; в действительности, обе эти переменные отношения, S_FROM_TO и SP_FROM_TO, имеют два потенциальных ключа и служат наглядными примерами переменных отношения, при использовании которых нет очевидных причин для выбора одного из таких потенциальных ключей в качестве первичного [9.14]. Такой выбор, как в этих примерах, был сделан исключительно ради определенности.

Ниже приведены определения на языке Tutorial D.

```
VAR S FROM TO
        BASE RELATION { S# S#, FROM DATE, TO
    DATE } KEY { S#, FROM } KEY { S#, TO } ;

VAR SP_FROM TO
        BASE RELATION { S# S#, P# P#, FROM DATE, TQ DATE ^
    KEY { S#, P#, FROM }
    KEY { S#, P#, TO } ;
```

Затем необходимо предусмотреть защиту от появления таких бессмысленных пар FROM-TO, в которых значение то меньше значения FROM, следующим образом.

```
CONSTRAINT S_FROM_TO_OK
   IS_EMPTY ( S_FROM_TO WHERE TO < FROM ) ;

CONSTRAINT SP_FROM_TO_OK
   IS_EMPTY ( SP_FROM_TO WHERE TO < FROM ) ;
```

Но описанные до сих пор ограничения все еще не предоставляют всех тех возможностей, которые хотелось бы получить с их помощью. Например, рассмотрим переменную отношения S_FROM_TO. Очевидно, что если в этой переменной отношения имеется кортеж с данными о поставщике Sx со значением FROM, равным f, и значением TO, равным t, то хотелось бы, чтобы в той же переменной отношения не было кортежа с данными о поставщике Sx, указывающими, что Sx работал по контракту в сутки, непосредственно предшествующие f, или в сутки, непосредственно следующие за t. В качестве примера рассмотрим поставщика S1, которого касается лишь один кортеж из переменной отношения S_FROM_TO, где FROM = dO4 и TO = cПО. Того факта, что {S#, FROM} является потенциальным ключом для этой переменной отношения, безусловно, не достаточно, чтобы можно было предотвратить появление дополнительного, "перекрывающего" кортежа S1 (скажем) с FROM = dO2 и TO = dO6, который, кроме всего прочего, указывал бы, что S1 работал по контракту в сутки, непосредственно следующие за четвертыми сутками. Безусловно, желательно, чтобы эти два кортежа S1 были объединены в один кортеж со значениями FROM = d02HTO = dlO.

Теперь читатель должен был уже понять, что идея объединения кортежей оказалась очень важной. Безусловно, если два указанных кортежа в приведенном выше примере не будут объединены, это повлечет за собой почти такие же неприятные последствия, как и

появление дубликатов! Наличие дубликатов можно рассматривать как равносильное "повторению одного и того же дважды". А в указанных двух кортежах с данными о поставщике S1, в которых интервалы времени FROM-то перекрываются, действительно "одно и то же повторяется дважды"; а именно, в обоих этих кортежах указано, что поставщик S1 работал по контракту в четвертые, пятые и шестые сутки. На самом деле, если в переменной отношения S_FROM_TO появятся оба эти кортежа, то будет иметь место нарушение того предиката, который мы сами для нее определили. Вернемся к этому вопросу и обсудим его более подробно в разделе 23.7.

Затем отметим, что того факта, что {S#, FROM} является потенциальным ключом для переменной отношения S_FROM_TO, также недостаточно для предотвращения возможности появления "примыкающего" кортежа si со значением (скажем) FROM = d02 и то = cЮЗ, который снова показывает, что поставщик S1 работал по контракту в сутки, непосредственно следующие за четвертыми сутками. Как и прежде, желательно, чтобы эти два рассматриваемых кортежа были объединены в один, поскольку в противном случае переменная отношения S_FROM_TO снова будет нарушать заданный для нее предикат. Следует опять отметить, что мы вернемся к этому вопросу и подробно его обсудим в разделе 23.7.

Ниже приведено ограничение, позволяющее исключить описанные выше ситуации перекрытия и примыкания.

```
CONSTRAINT XFT1
   IS_EMPTY
      ( ( ( S FROM TO RENAME ( FROM AS F1, TO AS T1 ) )
          JOIN ( S FROM TO RENAME ( FROM AS F2, TO AS
          T2 ) ) ) WHERE ( T1 > F2 AND T2 > F1 ) ) OR
                 ( F2 = T1+1 OR F1 = T2+1 ) ) ;
```

Итак, теперь действительно проблема стала полностью очевидной! Приведенное здесь ограничение является очень сложным, не говоря уже о том, что в нем допущена весьма произвольная форма записи (например) T1+1 для обозначения суток, непосредственно следующих за сутками, обозначенными как T1; к этому вопросу мы вернемся в следующем разделе. Кроме того, если речь идет о полностью хронологической базе данных, наподобие показанной на рис. 23.3, то в ней, вероятно, придется определить множество ограничений такого же общего характера, как и ограничение XFT1, и, безусловно, снова потребуется иметь какое-то удобное сокращение для этой цели.

Примечание. Фактически в той формулировке ограничения XFT1, которая здесь приведена, имеется еще один недостаток, а именно — она не позволяет ответить на вопрос о том, как будет вычисляться выражение T1+1, если окажется, что T1 обозначает "конечную позицию интервала времени".

Далее отметим, что сочетание атрибутов {S#,FROM} в переменной отношения SP_FROM_TO не является внешним ключом, связывающим эту переменную отношения с переменной отношения S_FROM_TO (даже несмотря на то, что в этом сочетании применяются такие же атрибуты, как и в первичном ключе переменной отношения S_FROM_TO). Но, безусловно, необходимо обеспечить гарантию того, что если некоторый поставщик представлен в переменной отношения SP_FROM_TO, TO OH должен быть также представлен и в переменной отношения S_FROM_TO, следующим образом.

```
CONSTRAINT XFT2
   SP_FROM_TO { S# } C S_FROM_TO { S# } ;
```

Это ограничение является примером зависимости включения **[11.4].** Зависимости включения могут рассматриваться как обобщение ограничений ссылочной целостности (о чем было сказано в главе 11). К тому же должно быть очевидно, что любая хронологическая база данных, подобная приведенной на рис. 23.3, скорее всего, будет содержать большое количество таких зависимостей (по крайней мере, неявно).

Но ограничение XFT2 все еще не является достаточно приемлемым, поскольку необходимо также гарантировать, что если в переменной отношения SP_FROM_TO показан некоторый поставщик как вообще способный поставлять какие-то детали на протяжении определенного интервала времени, то переменная отношения S_FROM_TO должна показывать, что тот же поставщик работает по контракту на протяжении всего того же интервала времени, следующим образом.

```
CONSTRAINT XFT3
   COUNT ( SP FROM TO { ALL BUT P# } ) =
   COUNT ( ( ( SP FROM TO RENAME ( FROM AS SPF, TO AS SPT ) )
                                              { ALL BUT P# }
           JOIN
            ( S FROM TO RENAME ( FROM AS SF, TO AS ST )
       ) ) WHERE SF < SPF AND ST > SPT ) ;
```

В данном случае интуиция подсказывает, что если переменная отношения SP_FROM_TO включает кортеж, обозначающий поставщика Sx как способного поставлять некоторую определенную деталь от суток spf до суток spt, то переменная отношения S_FROM_TO должна включать кортеж, обозначающий поставщика Sx как работающего по контракту на протяжении того же интервала времени. (При этом предполагается, что все ограничения, описанные до этого, вступили в силу!) Мы намеренно не приводим здесь дальнейший анализ указанного ограничения, а остановимся лишь на том замечании, что и это ограничение является весьма сложным и что нам снова, вероятно, придется определить множество ограничений такого же общего характера в любой реальной базе данных. Поэтому и в этом случае, безусловно, желательно было бы иметь в своем распоряжении какие-то удобные сокращения.

Запросы (первая полностью хронологическая база данных). Ниже приведены полностью хронологические аналоги запросовA и в.

- Запрос А. Получить тройки значений S#-FROM-TO, относящиеся к поставщикам, которые были способны поставлять по меньшей мере одну деталь на протяжении по меньшей мере одного интервала времени, где FROM и то вместе определяют та кой интервал времени. Обратите внимание на то, что результат этого запроса мо жет содержать несколько кортежей, относящихся к одному и тому же поставщику (но, безусловно, с разными интервалами времени; более того, эти интервалы вре мени не должны ни сопрягаться, ни перекрываться).
- Запрос в. Получить тройки значений S#-FROM-TO, относящиеся к поставщикам, которые вообще не были способны поставлять какие-либо детали на протяжении по меньшей мере одного интервала времени, где FROM и то вместе определяют та кой интервал времени. (И этот результат может содержать несколько кортежей для одного и того же поставщика.)

Итак, читателю, вероятно, не потребуется много времени, чтобы убедиться в том, что ему, как и самому автору, действительно вряд ли захочется даже попытаться применить эти запросы на практике! Но и в том случае, если такая попытка действительно будет предпринята, то даже сам тот факт, каким образом они будут представлены (с невероятными затратами усилий), в конечном итоге покажет, что весьма желательно иметь какие-то способы сокращенного обозначения компонентов этих запросов.

Поэтому по самой своей сути основная проблема обработки хронологических данных состоит в том, что попытка решения этой задачи сразу же приводит к созданию ограничений и запросов, которые становится чрезвычайно сложно выразить (не говоря уже об обновлениях, описание которых выходит за рамки данной главы). Но они являются чрезвычайно сложными, только если в системе не предусмотрены какие-то приемлемые сокращения (а такие сокращения в современных коммерческих СУБД не предусмотрены).

23.3. ИНТЕРВАЛЫ ВРЕМЕНИ

Теперь перейдем к разработке такого подходящего множества сокращений. Первым и наиболее важным шагом является переход к трактовке интервалов времени в качестве самостоятельных значений, а не в виде пар отдельных начальных и конечных значений, как было до сих пор. Например, рассмотрим интервал времени от четвертых суток до десятых суток. Для того чтобы подчеркнуть тот факт, что этот интервал времени теперь рассматривается как самостоятельное значение, он будет неформально обозначаться с помощью сокращенного выражения [d04 :dl0], а не с помощью таких словесных оборотов, как "интервал времени от четвертых суток до десятых суток". Более конкретно принятые соглашения описаны ниже.

- Значение наподобие [d04 :dl0] называется *значением интервала времени* или сокращенно просто *интервалом времени.*
- *и* Значения d04 и dl0, соответственно, являются *начальной временной позицией* и *конечной временной позицией* этого значения интервала времени.
- Значение интервала времени относится к определенному интервальному типу.
- Этот интервальный тип определен на основании некоторого позиционного типа.

Точное определение этих терминов будет дано чуть позже. А вначале рассмотрим, что произойдет с базой данных, применяемой в качестве примера, если мы будем руководствоваться указанным подходом (рис. 23.4).

Предикаты этой базы данных приведены ниже.

- S_DURING. Поставщик S# работал по контракту на протяжении интервала времени от начальной позиции интервала времени DURING до конечной позиции интервала времени DURING.
- SP_DURING. Поставщик S# был способен поставлять деталь P# на протяжении интервала времени от начальной позиции интервала времени DURING до конечной позиции интервала времени DURING.

Теперь перейдем к описанию формальных определений. Прежде всего, любой конкретный тип т может использоваться в качестве *позиционного типа* (и значения типа т могут именоваться *позициями),* если для этого типа т определены все описанные компоненты.

- **Полное упорядочение,** в соответствии с которым для любой пары значений v1 **и** v2 типа **т** определен оператор ">"; если значения v1 и v2 различны, то одно **и** только одно из выражений "v1 > v2" и "v2 > v1" является истинным, а другое — ложным.

  Примечание. Как было указано в главе 5, для типа т, безусловно, определен оператор "=". Если известно, что определен также оператор ">" (а также известно, что доступен логический оператор NOT), то можно вполне обоснованно полагать, что для всех пар значений типа **т** фактически доступны и все обычные операторы сравнения: "=", "*", ">", ">", "<", и "<".

- Нуль-арные операторы **FIRST_T** И **LAST_T,** которые возвращают, соответственно, первое и последнее значения типа т, исходя из упомянутого выше полного упоря дочения.

Унарные операторы **NEXT_T** И **PRIOR_T,** которые возвращают, соответственно, предшествующее и последующее значения для любого конкретного значения типа **т** исходя **из** упомянутого выше полного упорядочения.

Примечание. NEXT_T — это **функция определения последующего значения** для типа т; безусловно, результат выражения NEXT_T(p) не определен, если p = LAST_T(). Аналогичным образом, не определен и результат выражения PRlOR_T(p), если p = FIRST_T().

S_DURING

S#	DURING
S1	[d04 : d10]
S2	[d02 : d04]
S2	[d07 : d10]
S3	[d03 : d10]
S4	[d04 : d10]
S5	[d02 : d10]

SP_DURING

S#	P#	DURING
S1	P1	[d04 : d10]
S1	P2	[d05 : d10]
S1	P3	[d09 : d10]
S1	P4	[d05 : d10]
S1	P5	[d04 : d10]
S1	P6	[d06 : d10]
S2	P1	[d02 : d04]
S2	P1	[d08 : d10]
S2	P2	[d03 : d03]
S2	P2	[d09 : d10]
S3	P2	[d08 : d10]
S4	P2	[d06 : d09]
S4	P4	[d04 : d08]
S4	P5	[d05 : d10]

Рис. 23.4. База данных поставщиков и поставок (вторая полностью хронологическая версия, в которой используются интервалы) и примеры значений

Затем определим[6] **генератор типа INTERVAL.** Если **т** — позиционный тип, то INTERVAL_T — это **интервальный тип,** полученный путем вызова соответствующего генератора типа т.

[6] Следует отметить, что генератор типа INTERVAL является единственной конструкцией, описанной в этой главе, которая не является просто сокращенным обозначением. Поэтому применяемый автором подход к определению хронологических баз данных (в отличие от некоторых других подходов, описанных в литературе) не предусматривает введения каких-либо изменений в классическую реляционную модель (хотя и связан с необходимостью ввести некоторые обобщения, как будет показано в разделах 23.5 и 23.7).

Как и все генераторы типов, INTERVAL имеет связанные с ним множество универсальных возможных представлений, множество универсальных операторов и множество универсальных ограничений, причем все эти множества применяются к любому сгенерированному типу, который получен с помощью этого генератора. Ниже приведены конкретные определения.

- Рассматривается только одно возможное представление — любое значение типа INTERVAL_T (т.е. любой **интервал времени** этого типа) может быть представлено в виде пары значений типа т, соответствующих начальной и конечной позициям рассматриваемого интервала. Соответствующий оператор селектора имеет следующий синтаксис.

  ```
  INTERVAL_T ( [ b : e ] )
  ```

  Здесь b и **e** — выражения типа т, и общий вызов селектора возвращает значение интервала с начальной и конечной позициями, которые равны значениям, обозначенным этими выражениями.

- Соответствующие "операторы THE_", **BEGIN** и **END,** имеют следующий синтаксис.

  ```
  BEGIN ( i )
  END ( i )
  ```

  Здесь i — выражение некоторого интервального типа, и эти два вызова операторов возвращают, соответственно, начальную и конечную позиции интервала, обозначенного этим выражением.

  Примечание. Как уже было указано, операторы BEGIN and END фактически являются "операторами THE_", а в языке Tutorial D такие операторы обычно обозначаются как THE_BEGIN и THE_END. Но вместо этого здесь и дальше в данной главе для согласования с другими работами в области исследования хронологических баз данных используются обозначения BEGIN и END.

- К другим универсальным операторам относятся оператор присваивания ": =", множество логических операторов (включающих, в частности, оператор "="), известных под общим названием *операторов Аллена* (они описаны ниже в этом разделе), и целый ряд других операторов (которые также описаны ниже в этом разделе).

- Рассмотрим только одно универсальное ограничение, а именно ограничение, согласно которому, если i является интервалом, то BEGIN (i) < END (i). Одним из следствий этого ограничения является то, что интервалы никогда не бывают пустыми — они всегда содержат по меньшей мере одну позицию. Другим следствием становится то, что больше не нужны явные ограничения, "позволяющие предотвратить появление бессмысленных пар FROM-TO, в которых значение то меньше значения FROM" (как было указано в предыдущем разделе).

Теперь должно быть очевидно, что тип DATE, в частности, удовлетворяет требованиям к позиционному типу и поэтому может использоваться в качестве такого типа. Следовательно, INTERVAL_DATE — это допустимый интервальный тип, поэтому определения переменных отношения S_DURING и SP_DURING могут выглядеть примерно следующим образом.

```
VAR S DURING BASE RELATION
  { S# S#, DURING INTERVAL_DATE } KEY { S#, DURING } ;

VAR SP DURING BASE RELATION
  { S# S#, P# P#, DURING INTERVAL_DATE } KEY { S#, P#, DURING } ;
```

Следует отметить, что эти определения все еще остаются весьма неполными! Мы вернемся к этой теме и доработаем указанные определения в разделе 23.7. Но следует отметить, что эти определения позволяют устранить проблему, связанную с необходимостью применять произвольный выбор в отношении того, какой из двух потенциальных ключей должен стать первичным. А что касается других ограничений и запросов, приведенных в разделе 23.2, то должно быть ясно, что непосредственные аналоги этих ограничений и запросов могут быть легко сформулированы применительно к базе данных, показанной на рис. 23.4, благодаря существованию операторов BEGIN и END. Но здесь не показаны какие-либо из этих формулировок, поскольку в задачу автора входит именно предоставление читателю гораздо более лучшего способа выражения таких ограничений и запросов.

Еще раз отметим, что тип DATE является допустимым позиционным типом, но до сих пор в этой главе еще не было выдвинуто ни одного требования, чтобы позиционные типы относились именно к типам "даты и времени" или чтобы сами интервалы были именно интервалами "даты и времени". В действительности, хотя интервалы являются фундаментальной абстракцией, необходимой для работы с хронологическими данными, : должно быть ясно, что понятие интервала фактически имеет гораздо более широкую об-: ласть применения; это означает, что имеется также много других приложений для интер-і валов, в которых интервалы не обязательно являются хронологическими. Ниже приведено несколько примеров.

- Суммы, облагаемые налогами, разбиты на шкалы с разными ставками налогов; иными словами, на интервалы, начальными и конечными позициями которых (а также всеми промежуточными позициями) являются денежные значения. Компьютеры рассчитаны на эксплуатацию в пределах определенных диапазонов температур и напряжений, иными словами, интервалов, которые состоят из позиций, представляющих собой, соответственно, температуры и напряжения.
- Чувствительность различных животных к внешней среде характеризуется разными частотами световых и звуковых волн, которые могут восприниматься их органами зрения и слуха.
- Различные природные явления возникают и могут быть измерены с учетом интер валов глубины земли или моря, либо высоты над уровнем моря.

Несмотря на то, что данная глава в основном посвящена изучению хронологических интервалов, значительная часть представленных в ней материалов фактически относится к любым интервалам в целом. Но из-за ограничений по объему автор не может позволить себе привести дополнительные сведения по этой теме, поэтому ниже указан лишь ряд примеров нехронологических интервальных типов.

- INTERVAL_INTEGER

  Здесь позиционным типом является INTEGER, функцией определения последующей позиции является функция определения "следующего целого числа" (т.е.

функция "сложения с единицей"), а значениями этого интервального типа являются интервалы в форме [b: e], где b и e — значения типа INTEGER и b ≤ e.

- INTERVAL_MONEY
  Здесь MONEY (остановимся на таком допущении) — тип, который представляет денежные суммы, измеряемые в долларах и центах. Функцией определения последующей позиции является функция "сложения с одним центом". Значениями этого интервального типа являются интервалы в форме [b: e], где Ь и e — значения типа MONEY И b ≤ e.

Операции над позициями и интервалами

Теперь перейдем к определению ряда полезных операций над позициями и интервалами (в дополнение к тем, которые уже рассматривались выше). Подготовку примеров, иллюстрирующих функциональные возможности этих операторов, оставляем читателю в качестве упражнения.

Терминология. Допустим, что т — позиционный тип, а р, p1 и p2 — значения типа т; неформально можно предположить, что для обозначения позиций, соответственно, предшествующей и последующей по отношению к р, могут использоваться выражения p+1 и p-1. Кроме того, допустим, что i, il и i2 — интервалы типа INTERVAL_T; b, Ы и Ь2 — начальные позиции, а e, el и e2 — конечные позиции, соответственно, интервалов i, il и i2; неформально можно предположить, что для обозначения интервалов i, il и i2 используются, соответственно, выражения [b:e], [Ы:e1] и [b2:e2].

В таком случае справедливы приведенные ниже утверждения.

Выражение ls_NEXT_T(pl,p2) принимает истинное значение тогда и только тогда, когда позиция pi является непосредственно предшествующей по отношению к позиции p2. Выражение is_PRlOR_T(pl,p2) принимает истинное значение тогда и только тогда, когда является истинным выражение ls_NEXT_T(p2,pl); это означает, что is_PRiOR_T(pl,p2) = is_NEXT_T(p2,pl).

Выражение MAX (pi, p2) возвращает p2, если выражение pi < p2 является истинным, в противном случае оно возвращает pi; выражение мш(p1,p2) возвращает pi, если выражение pi < p2 является истинным, в противном случае оно возвращает p2.

- Выражение p £ i является истинным тогда и только тогда, когда выражения b < p ир < e оба являются истинными; это означает, что p e i = (b < p AND p < e). Кроме того, i зр =p e i.

  Примечание. Символы "e " и "э" читаются, соответственно, как "содержится в" и "содержит".

Выражение COUNT(i) возвращает результат подсчета количества различных позицийр, таких что p E i.

- Интервал i является единичным интервалом тогда и только тогда, когда COUNT (i) = 1. Выражение POINT FROM i возвращает единственную позицию из единичного интервала i.

Выражения PRE (i) и POST (i), соответственно, возвращают позиции b-1 и e+1.
Примечание. Выражения PRE(i) и POST(i), соответственно, являются сокращениями OT PRIOR_T(BEGIN (i)) И NEXT_T(END(i)) .

Затем может быть определен целый ряд операторов для проверки того, являются ли два интервала равными, перекрываются ли они и т.д. Рассматриваемые операторы известны под общим названием **операторов** Аллена, поскольку основная из часть была впервые предложена Алленом (Allen) в [23.1], но здесь мы не всегда следуем предложенной Алленом классификации этих операторов. Эти операторы определены наиболее сжато, в терминах эквивалентностей, но читателю рекомендуется нарисовать какие-то наглядные схемы, чтобы понять их на интуитивном уровне.

- Равняется (=): (i1 = i2) = (b1 = b2 AND el = e2).
- Включает (" ⊇ ") и включено в ("⊆"): (il ⊇ i2) = (b1 ≤ b2 AND el ≥ e2); (i2 ⊆ il) = (i1 ⊇ i2).
- Строго включает ("⊃ ") и строго включено в ("⊂"): (i1 ⊃ i2) = (il ⊇ i2 AND il ≠ i2); (i2 ⊂ il) = (i1 ⊃ i2).
- BEFORE И AFTER: (i1 BEFORE i2) = (el < b2); (i1 AFTER i2) = (i2 BEFORE i1).
- MEETS: (i1 MEETS i2) = (b2 = e1+1 OR b1 = e2+l).

OVERLAPS: (i1 OVERLAPS i2) = (b1 ≤ e2 AND b2 ≤ e1).

- MERGES: (i1 MERGES i2) = (i1 OVERLAPS i2 OR il MEETS i2).

BEGINS: (i1 BEGINS i2) = (b1 = b2 AND el ≤ e2).
ENDS: (i1 ENDS i2) = (el = e2 AND b1≥ b2).

Наконец, определим некоторые полезные бинарные операторы над интервалами, которые возвращают интервалы, а именно аналоги знакомых операторов UNION, INTERSECT и MINUS для работы с интервалами. Каждый из них принимает два интервала такого же типа, как и его операнды, и возвращает в качестве результата другой интервал такого же типа (ниже даны определения этих операторов и приведены некоторые примеры).

- Оператор UNION. Выражение il UNION i2 возвращает [MlN(bl,b2) : MAX (el, e2)], если выражение il MERGES i2 принимает истинное значение; в против ном случае оно не определено. Например, объединением интервалов [d04 : d08] и [d06:dl0] является интервал [d04:dl0]; объединение интервалов [d02:d03] и [d06:dl0] не определено.
- Оператор INTERSECT. Выражение il INTERSECT i2 возвращает [MAX (Ы, Ь2) :MIN (el, e2)], если выражение il OVERLAPS i2 принимает истинное значение; в противном случае оно не определено. Например, пересечением интер валов [d04:d08] и [d06:dl0] является интервал [d06:d08]; пересечение [d02:d03] и [d06:dl0] не определено.

Оператор MINUS. Выражение il MINUS i2 возвращает [bl:MlN(b2-l, el)], если оба выражения, Ы < b2 и el ≤ e2, принимают истинное значение и возвращает [MAX(e2 + 1,b1): el], если оба выражения,b1≥ b2 и el > e2,

принимают истинное значение, а в противном случае результат этого выражения не определен. Например, разность между интервалами [d04:d08] и [d06:dl0] (в указанном порядке) равна [d04: d05], разность между интервалами [d06: dl0] и [d04:d08] (в указанном порядке) равна [dO9 :dl0], разность между интервалами [d02:d03] и [d06:dl0] (в любом порядке) не определена.

Примеры запросов

В завершение этого раздела рассмотрим несколько примеров запросов, которые наглядно иллюстрируют использование некоторых операторов, определенных в предыдущем подразделе. Вначале рассмотрим запрос: "Получить номера поставщиков, которые были способны поставлять деталь P2 в восьмые сутки". Ниже приведена возможная формулировка этого запроса к базе данных, показанной на рис. 23.4.

```
( SP_DURING WHERE P# = P# ('P2')
            AND dO8 € DURING ) { S# }
```

Пояснение. Выражение во внешних круглых скобках сокращает множество кортежей, присутствующих в настоящее время в переменной отношения SP_DURING, до такого множества кортежей, в котором значение p# равно P2, а в интервале, представляющем собой значение DURING, содержатся восьмые сутки. Затем к этому множеству кортежей применяется операция проекции по атрибуту s# для получения желаемого результата.

Примечание. На практике применяемое здесь выражение "d08" необходимо было бы заменить соответствующим вызовом селектора DATE.

В качестве второго примера рассмотрим приведенную ниже возможную формулировку запроса: "Определить пары поставщиков, которые были способны поставлять одну и ту же деталь одновременно".

```
WITH ( SP DURING RENAME (     S# AS X#, DURING AS XD ) ) AS Tl ,
     ( SP DURING RENAME (     S# AS Y#, DURING AS YD ) ) AS T2 ,
     ( Tl JOIN T2 ) AS T3     ,  ( T3 WHERE XD OVERLAPS YD ) AS T4
     ,
     ( T4 WHERE X# < Y# )     AS T5 :
T5 { X#, Y# }
```

Пояснение. Здесь Tl — отношение, которое является текущим значением переменной отношения SP_DURING, за исключением того, что атрибуты S# и DURING переименованы, соответственно, в x# и XD; отношение T2 является таким же, за исключением того, что в нем новыми именами атрибутов являются У# и YD. Отношение Т3 является соединением отношений T1 и T2 по номерам деталей. Отношение T4 представляет собой сокращение от Т3, в котором присутствуют только такие кортежи, в которых интервалы XD и YD перекрываются (это означает, что поставщики не только были способны поставлять одну и ту же деталь, но фактически были способны поставлять одну и ту же деталь одновременно, что и требовалось определить по условиям запроса). Отношение T5 является сокращенным обозначением от T4, в котором присутствуют только те кортежи, где номер поставщика x# меньше номера поставщика Y# (сравните с примером 7.5.5 из главы 7). Заключительная операция проекции пох# и Y# вырабатывает требуемый результат.

В качестве третьего примера предположим, что необходимо определить не только пары поставщиков, которые были способны поставлять одну и ту же деталь одновременно, но узнать также рассматриваемые детали и интервалы времени. Ниже приведена одна из возможных формулировок.

```
WITH ( SP DURING RENAME ( S# AS X#, DURING AS XD ) ) AS T1 ,
     ( SP DURING RENAME ( S# AS Y#, DURING AS YD ) ) AS T2 ,
     ( T1 JOIN T2 ) AS T3 ,
     ( T3 WHERE XD OVERLAPS YD ) AS T4 ,
     ( T4 WHERE X# < Y# ) AS T5 ,
     ( EXTEND T5 ADD ( XD INTERSECT YD ) AS DURING ) AS
T6 : T6 { X#, Y#, P#, DURING }
```

Пояснение. Отношения T1, T2, T3, T4 и T5 являются точно такими же, как в предыдущем примере. После их получения с помощью оператора EXTEND вычисляются соответствующие интервалы, а завершающая операция проекции вырабатывает желаемый результат.

23.4. УПАКОВКА И РАСПАКОВКА ОТНОШЕНИЙ

В данном разделе рассматриваются два новых (и чрезвычайно важных) реляционных оператора, называемых PACK и UNPACK. Но в качестве промежуточного этапа на пути к определению этих операторов вначале необходимо сделать краткое отступление и рассмотреть два более простых их аналога, которые называются, соответственно, COLLAPSE и EXPAND. Кроме того, для упрощения изложения эти последние операторы будут описаны в обратном порядке.

Операторы EXPAND и COLLAPSE

Оба оператора, EXPAND и COLLAPSE, в том виде, в каком они описаны в этом разделе[7], принимают в качестве единственного операнда унарное отношение, кортежи которого содержат интервалы, и вырабатывают в качестве результата другое отношение такого же типа. Например, предположим, что отношение r выглядит примерно следующим образом.

DURING
[d06:d09]
[d04:d08]
[d05:d10]
[d01:d01]

В таком случае операторы EXPAND r И COLLAPSE r вырабатывают результаты, которые выглядят следующим образом.

EXPAND r

DURING
[d01:d01]
[d04:d04]
[d05:d05]
[d06:d06]
[d07:d07]
[d08:d08]
[d09:d09]
[d10:d10]

COLLAPSE r

DURING
[d01:d01]
[d04:d10]

[7] Более общие версии этих операторов описаны в [23.4].

Пояснение. Допустим, что унарное отношение r включает DURING в качестве единственного атрибута, а DURING имеет значение в виде интервала. Тогда и развернутая, и свернутая формы отношения r являются унарными отношениями такого же типа, как r, которые определены, как описано ниже.

> Развернутой формой является такое отношение rx, содержащее все кортежи (и только такие кортежи), которые содержат единичный интервал в форме [p: p], где p — позиция в некотором интервале из некоторого кортежа r, определенного, как показано ниже.
>
> Сжатой формой отношения r является отношение rc, такое что имеют место описанные ниже условия.
>
> а) Отношения r и rc имеют одну и ту же развернутую форму.
>
> б) Никакие два разных кортежа в отношении rc не содержат интервалы i1 и i2, соответственно, такие что выражение i1 MERGES i2 является истинным. Рав ным образом, никакие два разных кортежа в отношении rc не содержат, соответственно, интервалы i1 и i2, такие что определено выражение i1 UNION i2. (Из этого следует, что отношение rc может быть вычислено из отношения r путем последовательной замены пар кортежей t1 и t2 в отношении r с помощью кортежа t, содержащего объединение интервалов в кортежах t1 и t2, до тех пор, пока такие замены не будут больше возможны.)

Теперь рассмотрим эти идеи более подробно. Для упрощения в остальной части данного подраздела предполагается, что единственными рассматриваемыми здесь отношениями являются именно унарные отношения, в которых кортежи содержат интервалы. Поэтому ниже приведено определение важного понятия эквивалентности таких отношений.

- Два отношения, r1 и r2, являются **эквивалентными** тогда и только тогда, когда множество всех позиций, содержащихся в интервалах кортежей отношения r1, равно множеству всех позиций, содержащихся в интервалах кортежей r2.

После анализа этого и приведенных ранее определений развернутых и свернутых форм становятся очевидными приведенные ниже выводы.

- Для любого конкретного отношения r всегда существует соответствующая развер нутая форма этого отношения r.
- Такая развернутая форма эквивалентна отношению r. Действительно, можно утверждать, что два отношения являются эквивалентными тогда и только тогда, когда они имеют одинаковую развернутую форму.
- Такая развернутая форма является уникальной; точнее, развернутая форма пред ставляет собой такое уникальное эквивалентное отношение, в котором все интер валы имеют минимально возможную длину (единичную длину).
- Интуитивно ясно, что развернутая форма отношения r позволяет нам заниматься анализом информационного наполнения отношения r на элементарнейшем уров не, не беспокоясь о том, что может существовать множество разных способов, с помощью которых эта информация могла бы быть объединена в какие-то проме жуточные "конгломераты".

- Если отношение r является пустым, то развернутая форма r также пуста.

Аналогичным образом, можно прийти к приведенным ниже заключениям.

- Для любого данного отношения r всегда существует соответствующая свернутая форма отношения r.
- Такая свернутая форма эквивалентна отношению r. Можно фактически утверждать, что два отношения являются эквивалентными тогда и только тогда, когда они имеют одну и ту же свернутую форму.
- Такая свернутая форма является уникальной; точнее, она представляет уникальное эквивалентное отношение, которое имеет минимально возможную кардинальность.
- Интуитивно ясно, что свернутая форма отношения r позволяет нам заниматься анализом информационного наполнения отношения r в сжатой форме ("представленной в виде единого конгломерата"), не беспокоясь о том, что существует вероятность сопряжения или перекрытия различных "конгломератов".
- Если отношение r является пустым, то свернутая форма отношения r также пуста.

Кстати, предположение о том, что операторы EXPAND и COLLAPSE противоположны по отношению друг к другу, является ошибочным. Фактически, вообще говоря, ни выражение EXPAND (COLLAPSE r), ни выражение COLLAPSE (EXPAND r) не являются тождественно равными отношению r (хотя оба эти выражения, безусловно, эквивалентны отношению r). Действительно, легко показать, что имеют место приведенные ниже тождества.

```
EXPAND ( COLLAPSE r ) ≡ EXPAND r.
COLLAPSE ( EXPAND r ) ≡ COLLAPSE r.
```

Из этого следует, что в последовательности операций "свертывания, а затем развертывания" или "развертывания, а затем свертывания", применяемой к некоторому отношению r, первую операцию можно просто игнорировать; это наблюдение может оказаться полезным при решении задач оптимизации (особенно если первой операцией ЯВЛЯеТСЯ EXPAND).

В завершение этого подраздела приведем еще два замечания.

- В [23.4] показано, что и оператор EXPAND, и оператор COLLAPSE можно определить в терминах операторов, которые уже предусмотрены в реляционной алгебре. Иными словами, оба оператора являются просто сокращениями.
- Кроме того, в [23.4] показано, что весьма желательно определить такие версии операторов EXPAND И COLLAPSE, которые работают с нуль-арными, а не с унарными отношениями. Подробный анализ этого вопроса здесь не приведен, а вместо этого отметим, что если r — нуль-арное отношение, то выражения EXPAND Г И COLLAPSE r возвращают r. К тому же определим, что два нуль-арных отношения являются эквивалентными тогда и только тогда, когда они являются равными.

Операторы PACK и UNPACK

В своей наиболее общей форме и оператор PACK, и оператор UNPACK принимают в качестве единственного операнда n-арное отношение, в котором один из атрибутов имеет значение в виде интервала и в качестве своего результата вырабатывает другое отношение такого же типа. Например, предположим, что отношение r выглядит примерно следующим образом.

S#	DURING
S2	[d02:d04]
S2	[d03:d05]
S4	[d02:d05]
S4	[d04:d06]
S4	[d09:d10]

В таком случае выражения PACK r ON DURING и UNPACK r ON DURING вырабатывают, соответственно, результаты, которые выглядят следующим образом.

PACK r ON DURING

S#	DURING
S2	[d02:d05]
S4	[d02:d06]
S4	[d09:d10]

UNPACK r ON DURING

S#	DURING
S2	[d02:d02]
S2	[d03:d03]
S2	[d04:d04]
S2	[d05:d05]
S4	[d02:d02]
S4	[d03:d03]
S4	[d04:d04]
S4	[d05:d05]
S4	[d06:d06]
S4	[d09:d09]
S4	[d10:d10]

Следует отметить, что неформально каждый из этих результатов представляет такую же информацию, что и первоначальное отношение r, но характеризуется описанными ниже особенностями.

- В случае оператора PACK эта информация была переупорядочена таким образом, что никакие два интервала DURING, относящиеся к определенному поставщику, не являются смежными и не перекрываются.
- В случае оператора UNPACK эта информация была переупорядочена таким образом, что каждое значение DURING (а значит, в силу этого и каждый отдельный интервал DURING, относящийся к определенному поставщику) представляет собой

именно единичный интервал. Взаимосвязь операторов COLLAPSE и EXPAND с операторами PACK и UNPACK должна быть интуитивно ясной. Должно быть, по-видимому, также очевидно, что первоначальное отношение r и соответствующие ему упакованная и распакованная формы являются в определенном смысле эквивалентными друг другу (точный смысл этого утверждения будет определен в конце текущего раздела).

Теперь сосредоточимся именно на изучении оператора PACK. Вначале рассмотрим приведенную ниже версию запроса A, переформулированную в терминах базы данных, приведенной на рис. 23.4.

- Запрос A. Определить пары S#-DURING для поставщиков, которые были способны поставлять по меньшей мере одну деталь на протяжении по меньшей мере одного интервала времени, где DURING обозначает такой интервал.

Если в качестве примера рассматриваются данные, приведенные на рис. 23.4, то желаемый результат выглядит следующим образом.

S#	DURING
S1	[d04:d10]
S2	[d02:d04]
S2	[d08:d10]
S3	[d08:d10]
S4	[d04:d10]

Это отношение представляет собой упакованную форму некоторой проекции переменной отношения SP по атрибуту DURING, а именно проекции по S# и DURING. И мы, наконец, подошли к тому этапу, когда появилась возможность получить этот результат с помощью простого выражения в следующей форме.

```
PACK T1 ON DURING
```

Здесь T1 — указанная проекция. Но для достижения указанной цели необходимо выполнить ряд небольших шагов. На первом шаге применим следующее выражение.

```
WITH SP_DURING { S#, DURING } AS T1 :
```

Это выражение позволяет получить требуемую проекцию (в действительности, в данном выражении просто "отбрасываются с помощью операции проекции" номера деталей, которые не требуются в рассматриваемом запросе). При использовании значений данных, обычно применяемых в качестве примера, проекция T1 принимает следующий вид.

S#	DURING
S1	[d04:d10]
S1	[d05:d10]
S1	[d09:d10]
S1	[d06:d10]
S2	[d02:d04]
S2	[d08:d10]
S2	[d03:d03]
S2	[d09:d10]
S3	[d08:d10]
S4	[d06:d09]
S4	[d04:d08]
S4	[d05:d10]

Обратите внимание, что это отношение содержит избыточную информацию; например, в нем не меньше трех раз указано, что поставщик S1 был способен поставлять какие-то детали в шестые сутки (в отличие от этого, желаемый результат не должен содержать такой избыточности).

На следующем шаге должно быть получено еще одно промежуточное отношение,Т2.

```
WITH ( Tl GROUP { DURING } AS X ) AS T2 :
```

Результирующее отношение Т2 показано на рис. 23.5.

Атрибут X отношения Т2 имеет значение в виде отношения, поэтому к унарным отношениям, являющимся значениями этого атрибута, можно применить оператор COLLAPSE следующим образом.

```
WITH ( EXTEND T2 ADD COLLAPSE ( X ) AS Y )
                              { ALL BUT X } AS T3 :
```

Очередное промежуточное отношение Т3 приведено ниже (обратите внимание на то, что атрибут X был удален с помощью операции проекции, заданной в виде спецификации "{ALL BUT X}"), как показано на рис. 23.6.

Наконец, выполняется разгруппирование полученного отношения следующим образом.

```
T3  UNGROUP  У
```

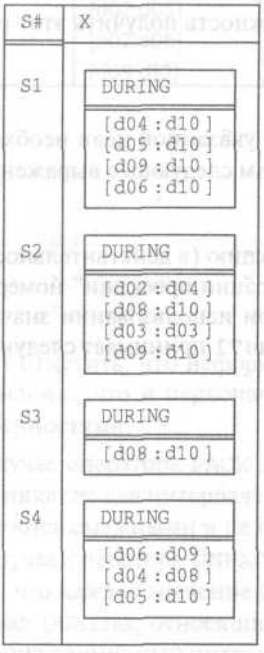


Рис. 23.5. Результирующее отношение Т2

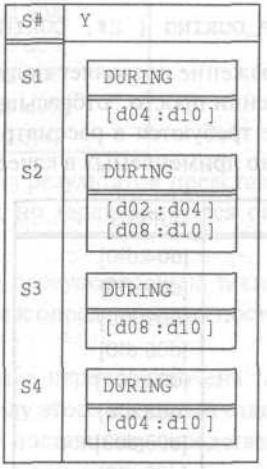


Рис. 23.6. Промежуточное отношение Т3

Данное выражение позволяет получить желаемый результат. Иными словами, после объединения всех описанных выше шагов (и небольшого упрощения) формируется следующее общее выражение, позволяющее достичь желаемого результата.

```
WITH SP DURING { S#, DURING  } AS T1 ,
   ( T1 GROUP { DURING } AS  X ) AS T2 ,
   ( EXTEND T2 ADD COLLAPSE  ( X ) AS Y ) { ALL BUT X } AS
T3 :
T3 UNGROUP Y
```

Теперь можно определить оператор PACK (который, безусловно, является сокращенным обозначением). Этот оператор имеет следующий синтаксис.

```
PACK  r  ON  A
```

Здесь r — реляционное выражение, а A — интервальный атрибут отношения, обозначенного этим выражением. Семантика данного оператора определена путем очевидного обобщения операций группирования, расширения, проекции и разгруппирования, с помощью которых был получен результат на основании выражения T1, как показано ниже.

```
PACK r ON A = WITH ( r GROUP { A } AS X ) AS R1 ,
                  ( EXTEND R1 ADD COLLAPSE ( X ) AS Y )
                                    { ALL BUT X } AS R2
              : R2 UNGROUP Y
```

Как было указано выше, теперь запрос A может быть сформулирован следующим образом.

```
PACK SP_DURING { S#, DURING } ON DURING
```

Примечание. В качестве пояснения следует явно отметить (как должно быть очевидно из этого определения оператора PACK), что для упаковки отношения по некоторому атрибуту A необходимо выполнить группирование этого отношения по всем его атрибутам, кроме атрибута A. (Как было указано в главе 7, например, выражение "T1 GROUP {DURING} ..." можно прочитать как "группировать T1 по S#", где s# является единственным атрибутом отношения T1, кроме указанного в спецификации GROUP.) Но следует отметить, что хотя и гарантируется возвращение выражением "r GROUP {A} ..." результата, содержащего точно один кортеж для каждого отдельного значения в (где в обозначает все атрибуты r, кроме A), выражение "PACK r ON A" может возвратить результат, который содержит несколько кортежей, относящихся к любому заданному значению в. В качестве иллюстрации можно указать результат операции PACK для запроса A, который содержит два кортежа, относящиеся к поставщику S4.

Теперь перейдем к изучению оператора UNPACK и запроса в.

- Запрос в. Определить пары S#-DURING с данными о поставщиках, которые не были способны поставлять какие-либо детали вообще на протяжении по меньшей мере одного интервала времени, где DURING обозначает такой интервал.

Теперь читателю, вероятно, стало ясно, что здесь фактически требуется найти все пары S#-DURING, которые присутствуют или следуют из отношения S_DURING, и вместе с тем не присутствуют и не следуют из отношения SP_DURING. Такого краткого анализа рассматриваемой задачи должно быть достаточно, чтобы сделать предположение (вполне обоснованное), что здесь, по сути, снова требуется выполнить ряд операций распаковки, применить к результатам операцию разности, затем снова упаковать полученный результат операции разности. Итак, вначале рассмотрим приведенный ниже оператор UNPACK.

```
UNPACK r ON A = WITH (        r GROUP {   A } AS X ) AS Rl ,
                    (EXTEND Rl ADD EXPAND ( X ) AS Y )
                                                    { ALL
BUT X } AS R2 : R2 UNGROUP Y
```

Это определение идентично определению оператора PACK, если не считать того, что во второй строке вместо оператора COLLAPSE появился оператор EXPAND. Назовем результат этого выражения "распакованной формой отношения r по атрибуту A".

Поэтому, что касается запроса в, то левый необходимый нам операнд (т.е. пары S#-DURING, которые присутствуют или следуют из отношения S_DURING) можно получить следующим образом.

```
UNPACK S_DURING { S#, DURING } ON DURING
```

Ниже приведена развернутая форма этого выражения.

```
WITH S DURING { S#, DURING } AS Tl ,
   ( Tl GROUP { DURING } AS X ) AS T2 ,
   ( EXTEND T2 ADD EXPAND ( X ) AS  Y ) { ALL BUT X } AS
T3 :
T3 UNGROUP Y
```

Подробную пошаговую проработку этого выражения оставляем читателю в качестве упражнения. Но если в качестве примера рассматриваются данные, приведенные на рис. 23.4, то общий результат (назовем его Ul) выглядит следующим образом.

S#	DURING
S1	[d04:d04]
S1	[d05:d05]
S1	[d06:d06]
S1	[d07:d07]
S1	[d08:d08]
S1	[d09:d09]
S1	[d10:d10]
S2	[d02:d02]
S2	[d03:d03]
S2	[d04:d04]
S2	[d07:d07]
S2	[d08:d08]
S2	[d09:d09]
S2	[d10:d10]
S3	[d03:d03]
S3	[d04:d04]
S3	[d05:d05]
S3	[d06:d06]
S3	[d07:d07]

S#	DURING
S3	[d08:d08]
S3	[d09:d09]
S3	[d10:d10]
S4	[d04:d04]
S4	[d05:d05]
S4	[d06:d06]
S4	[d07:d07]
S4	[d08:d08]
S4	[d09:d09]
S4	[d10:d10]
S5	[d02:d02]
S5	[d03:d03]
S5	[d04:d04]
S5	[d05:d05]
S5	[d06:d06]
S5	[d07:d07]
S5	[d08:d08]
S5	[d09:d09]
S5	[d10:d10]

Безусловно, правый операнд (т.е. пары S#-DURING, которые присутствуют или следуют из отношения SP_DURING) может быть получен аналогичным образом.

```
UNPACK SP_DURING { S#, DURING } ON DURING
```

Теперь можно применить оператор разности следующим образом.

Ниже приведен результат этого выражения (назовем его U2).

S#	DURING
S1	[d04:d04]
S1	[d05:d05]
S1	[d06:d06]
S1	[d07:d07]
S1	[d08:d08]
S1	[d09:d09]
S1	[d10:d10]
S2	[d02:d02]
S2	[d03:d03]
S2	[d04:d04]
S2	[d08:d08]
S2	[d09:d09]

S#	DURING
S2	[d10:d10]
S3	[d08:d08]
S3	[d09:d09]
S3	[d10:d10]
S4	[d04:d04]
S4	[d05:d05]
S4	[d06:d06]
S4	[d07:d07]
S4	[d08:d08]
S4	[d09:d09]
S4	[d10:d10]

U1 MINUS U2

Результат этого выражения (назовем его U3) имеет следующий вид.

S#	DURING
S2	[d07:d07]
S3	[d03:d03]
S3	[d04:d04]
S3	[d05:d05]
S3	[d06:d06]
S3	[d07:d07]
S5	[d02:d02]
S5	[d03:d03]

S#	DURING
S5	[d04:d04]
S5	[d05:d05]
S5	[d06:d06]
S5	[d07:d07]
S5	[d08:d08]
S5	[d09:d09]
S5	[d10:d10]

Наконец, упакуем отношение из, чтобы получить желаемый общий результат.

```
PACK U3 ON DURING
```

Окончательный результат приведен ниже.

S#	DURING
S2	[d07:d07]
S3	[d03:d07]
S5	[d02:d10]

Итак, запрос в имеет следующую формулировку в виде одного выражения.

```
PACK
   ( ( UNPACK S DURING { S#, DURING } ON DURING )
     MINUS
   ( UNPACK SP DURING { S#, DURING } ON DURING ) )
ON DURING
```

Отметим, что распаковка отношения r по атрибуту A (как и упаковка отношения r по атрибуту A) сводится к группированию отношения r по всем атрибутам отношения r, кроме A.

Как и операторы COLLAPSE и EXPAND, на которых они основаны, операторы PACK и UNPACK не являются обратными по отношению друг к другу. Поэтому ни выражение UNPACK (PACK r ON A) ON A, ни выражение PACK (UNPACK r ON A) ON A в общем случае не являются тождественно равными r (хотя оба эти выражения эквивалентны отношению r, в том смысле, который будет описан чуть позже). Безусловно, можно легко показать, что справедливы приведенные ниже тождества.

```
UNPACK r ON A ≡ UNPACK ( PACK r ON A ) ON A.
PACK r ON A ≡ PACK ( UNPACK r ON A ) ON A.
```

Из этого следует, что первая операция в последовательности операций "упаковка, затем распаковка" или "распаковка, затем упаковка" над некоторым конкретным отношением может просто игнорироваться; этот факт может оказаться полезным для целей оптимизации (особенно если первой операцией является UNPACK).

Дополнительные примеры

В этом разделе рассматриваются некоторые дополнительные примеры использования операций PACK и UNPACK при формулировке запросов. Здесь принято вполне обоснованное предположение, что в каждом случае требуется .результат в форме, упакованной должным образом.

В качестве первого примера намеренно рассмотрен запрос, не относящийся к временным интервалам. Предположим, что задана переменная отношения NHW с атрибутами NAME, HEIGHT и WEIGHT, в которой содержится информация о росте и весе отдельных лиц. Рассмотрим запрос: "Для каждого значения веса, представленного в переменной отношения NHW, получить каждый диапазон значений роста, такой что для каждого указанного диапазона r и для каждого значения роста в диапазоне r существует по меньшей мере одно лицо, представленное в NHW, которое имеет такой рост и такой вес". Ниже приведена одна из возможных формулировок требуемого запроса.

```
PACK
   ( ( EXTEND NHW { HEIGHT, WEIGHT }       ADD
  INTERVAL HEIGHT ( [ HEIGHT : HEIGHT ] ) AS HR )
     { WEIGHT, HR }
) ON HR
```

Пояснение. Начнем с формирования проекции NHW по атрибутам HEIGHT и WEIGHT, получив тем самым все пары значений роста и веса из первоначального отношения (т.е. все такие пары роста и веса, что имеется по меньшей мере одно лицо с таким ростом и весом). Затем эта проекция расширяется путем введения еще одного атрибута, HR, значением которого в любом заданном кортеже является единичный интервал в форме [h:h], где h — значение атрибута HEIGHT в том же кортеже (обратите внимание на вызов селектора интервала INTERVAL_HEIGHT). После этого с помощью операции проекции удаляется атрибут HEIGHT и результат упаковывается по атрибуту HR. Окончательным результатом становится отношение с двумя атрибутами WEIGHT и HR, которое имеет следующий предикат.

Для всех значений роста h в HR (но не для h = PRE (HR) или h = POST (HR)) существует по меньшей мере одно лицо p, такое что p имеет вес WEIGHT и рост h.

В качестве второго примера снова рассмотрим переменную отношения SP_DURING. В любой конкретный момент времени (при условии, что в это время вообще выполнялись какие-либо поставки) существует некоторый номер детали ртах, такой что ни один поставщик не способен поставлять в это время какую-либо деталь с номером больше ртах. (Безусловно, при этом предполагается, что для значений типа P# определен оператор ">".) Поэтому рассмотрим запрос: "Для каждого номера детали, который когда-либо был таким значением ртах, получить этот номер детали вместе с интервалом (интервалами), на протяжении которого он фактически был таким значением pmax". Ниже приведена одна из возможных формулировок подобного запроса.

```
WITH ( UNPACK SP DURING ON DURING ) AS
     SP UNPACKED , ( SUMMARIZE SP UNPACKED
       BY { DURING }
       ADD MAX ( P# ) AS PMAX ) AS SUMMARY
: PACK SUMMARY ON DURING
```

Заключительные замечания

По поводу операторов PACK И UNPACK может быть сказано гораздо больше, чем было отведено места в этой главе. Их подробное описание можно найти в [23.4]; ниже просто перечислены без доказательства или дополнительных комментариев некоторые наиболее важные свойства этих операторов.

- Упаковка или распаковка любого отношения r вообще без указания каких-либо атрибутов приводит просто к получению отношения r.
- Распаковка отношения r по двум или нескольким атрибутам, притом что все эти атрибуты имеют значения в виде интервала[8], осуществляется очень просто; если рас сматриваемыми атрибутами являются A1, A2, . . ., An (в некотором порядке), то желаемый результат может быть получен путем распаковки r по атрибуту A1, затем распаковки результата этой первой распаковки по атрибуту A2, . . ., наконец, распаковки результата предпоследней распаковки по атрибуту An.
- Упаковка отношения r по двум или нескольким атрибутам, притом что все эти атрибуты имеют значения в виде интервала, становится немного более сложной задачей. Но неформально можно утверждать, что если рассматриваемыми атрибу тами являются A1, A2, . . ., An (в указанном порядке), то желаемый результат можно получить, вначале распаковав r по всем указанным атрибутам, а затем упа ковав полученный результат распаковки по атрибуту A1, упаковав результат пер вой упаковки по атрибуту A2, . . ., наконец, упаковав результат предпоследней упаковки по атрибуту An.
- Предположим, что rl и r2 — отношения одного и того же типа и атрибуты A1, A2, . . ., An этих двух отношений имеют значения в виде интервала. Тогда

[8] Из этого следует, что такая операция идеально подходит для обработки отношений, имеющих два или несколько атрибутов со значениями в виде интервала.

отношения rl и r 2 являются эквивалентными (применительно к атрибутам A1, A2, . . ., An) тогда и только тогда, когда результаты операций UNPACK rl ON (Al, A2, . . . , An) И UNPACK r2 ON (Al, A2, , , . , . , An) равны. *:t·t Й*

23.5. ОБОБЩЕНИЕ РЕЛЯЦИОННЫХ ОПЕРАТОРОВ

В предыдущем разделе была приведена следующая формулировка запросав.

```
PACK
   ( ( UNPACK S DURING { S#, DURING } ON
     DURING ) MINUS
      ( UNPACK SP DURING { S#, DURING } ON
DURING ) ) ON DURING
```

Теперь необходимо отметить, что, как оказалось, выражения, подобные этому (включающие целый ряд распаковок, за которыми следует обычная реляционная операция, затем снова распаковка), так часто требуются на практике, что приобрела большую значимость идея определения сокращения для этих выражений (здесь речь идет о дальнейшем сокращении, поскольку, как следует из приведенного выше описания, сами эти выражения уже по сути являются сокращениями!). Безусловно, благодаря такому сокращению выражения должны стать намного короче. Более того, применение сокращенной формы записи открывает возможность повышения производительности следующим образом: если в выражении применяются длинные интервалы с большой степенью детализации, то выходные данные операции распаковки могут стать гораздо более объемными по сравнению с входными, а если бы системе действительно пришлось материализовать результат такой распаковки, то могло бы оказаться, что запрос "выполняется бесконечно долго" или требует слишком большого объема памяти. В отличие от этого, если все требования к обработке данных выражены в виде одной операции, то оптимизатор получает возможность выбрать наиболее эффективную реализацию, которая, в частности, не требует материализации распакованных промежуточных результатов.

Поэтому, исходя из изложенных выше пожеланий, определено следующее выражение.

USING (ACL) ◄ rl MINUS r2 ►

Это выражение является сокращенным обозначением для такого выражения.

```
PACK
    ( ( UNPACK rl ON ( ACL ) ) MINUS ( UNPACK r2 ON ( ACL
) ) ) ON ( ACL )
```

Здесь rl и r2 — реляционные выражения, обозначающие отношения одного и того же типа, а ACL — разделенный запятыми список имен атрибутов, в котором каждый указанный атрибут, во-первых, имеет некоторый интервальный тип, и, во-вторых, присутствует в обоих отношениях. Из этого следуют приведенные ниже выводы.

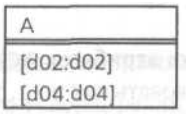
1. Если не указано иное, то определенный выше оператор будет именоваться просто как "U_разность" (U— сокращение от USING), ИЛИ ДЛЯ краткости просто U_MINUS.
2. Круглые скобки, в которые заключен разделенный запятыми список имен атрибутов в спецификации USING, могут быть опущены, если такой список содержит только одно имя атрибута.

Примечание. Это соглашение относится ко всем сокращениям "U_", которые будут определены в данной главе, поэтому мы не будем его обусловливать в каждом случае.

3. В каждом рассматриваемом в данном разделе контексте, в котором может появляться спецификация USING, затененные концы стрелок, "◄ и "►", служат для указания начала и конца выражения, к которому применяется спецификация USING.

4. В отличие от обычного оператора MINUS, оператор UMINUS может вырабатывать результат, имеющий большую кардинальность по сравнению с его левым операндом! Например, допустим, что отношения r 1 и r 2 определены следующим образом.

В таком случае применение выражения USING A ◄ rl MINUS r2 ► приводит к получению следующего результата.



На этом завершается рассмотрение оператора U_MINUS. Теперь должно быть очевидно, что можно определить версии "U_" всех обычных реляционных операторов (и в [23.4] такая задача действительно решена). Но ради экономии места мы здесь ограничимся только наиболее полезными из указанных операторов, а в качестве таковых мы рассматриваем (кроме U_MINUS) операторы U_UNION, U_INTERSECT, U_JOIN и U_проекции. Определения операторов U_UNION и U_INTERSECT формулируются по такому же общему принципу, как и U_MINUS; это означает, что выражение

USING (ACL) ◄ rl op r2 ►

является сокращенным обозначением для приведенного ниже выражения.

```
PACK
    ( ( UNPACK rl ON ( ACL ) ) op ( UNPACK r2 ON ( ACL )
) ) ON ( ACL )
```

Здесь op — операция UNION или INTERSECT, а опции ACL, rl и r2 определены как для операции U_MINUS.

Кстати, следует отметить, что в случае операции U_UNION фактически нет необходимости предварительно выполнять операции UNPACK. Это означает, что выражение, сокращенной формой которого является операция U_UNION, можно дополнительно упростить до следующей формы.

PACK (rl UNION r2) ON (ACL)

Несложно понять, почему такое упрощение возможно, но читатель может попытаться самостоятельно проработать какой-то пример, если он хочет убедиться в справедливости этого утверждения. (В действительности, аналогичные упрощения возможны

также применительно к некоторым другим операторам "U_", но подробные сведения об этом выходят за рамки данной главы.)

Кроме того (что немного противоречит здравому смыслу), так же, как операция U_MINUS может (неформально выражаясь), увеличить кардинальность результата, так и операция U_UNION может ее уменьшить; фактически операция U_UNION может вырабатывать результат с меньшей кардинальностью по сравнению с любым операндом (подготовку соответствующего примера оставляем в качестве упражнения для читателя). Аналогичным образом, операция U_INTERSECT может вырабатывать результат с большей кардинальностью, чем у любого из операндов (еще одно упражнение).

Теперь перейдем к рассмотрению оператора U_JOIN и определим следующее выражение.

```
USING ( ACL ) ◄ r1 JOIN r2 ►
```

Это выражение является сокращенным обозначением для приведенного ниже выражения.

```
PACK
   ( ( UNPACK r1 ON ( ACL ) ) JOIN ( UNPACK r2 ON ( ACL ) ) )
ON ( ACL )
```

Каждый атрибут, указанный в разделенном запятыми списке атрибутов ACL, должен относиться к некоторому интервальному типу и должен присутствовать как в rl, так и в r2 (и поэтому соединение должно выполняться по всем атрибутам, указанным в списке ACL, а также, возможно, по другим атрибутам).

Примечание. Если отношения rl и r2 имеют один и тот же тип, то операция U_JOIN вырождается в операцию U_INTERSECT.

Ниже приведен пример, иллюстрирующий использование операции U_JOIN. Предположим, что в базе данных определена еще одна переменная отношения, S_CITY_DURING, которая имеет атрибуты S#, CITY и DURING, потенциальный ключ {S#, DURING} и приведенный ниже предикат.

Поставщик s# находился в городе CITY на протяжении интервала времени от начальной позиции интервала DURING до конечной позиции интервала DURING.

Теперь рассмотрим запрос: "Получить кортежи S#-CITY-P#-DURING, такие что поставщик s# находился в городе CITY и был способен поставлять деталь P# на протяжении интервала DURING, где DURING включает четвертые сутки". Ниже приведена одна из возможных формулировок данного запроса.

```
( USING DURING ◄ S_CITY_DURING JOIN SP_DURING ► )
                                        WHERE dO4 6 DURING
```

Наконец, перейдя к операции U проекции, определим следующее выражение.

```
USING ( ACL ◄ R { BCL } ►
```

Это выражение является сокращенным обозначением для следующего выражения.

```
PACK ( ( UNPACK r ON ( ACL ) ) { BCL } ) ON ( ACL )
```

Каждый атрибут, указанный в разделенном запятыми списке атрибутов ACL, должен иметь некоторый интервальный тип и должен быть указан в списке BCL (а следовательно, в силу этого обстоятельства, должен быть атрибутом отношения r). В качестве примера снова рассмотрим запрос A, как показано ниже.

- Запрос A. Получить пары S#-DURING для поставщиков, которые были способны поставлять по меньшей мере одну деталь на протяжении по меньшей мере одного интервала времени, где DURING обозначает такой интервал.

Ниже приведена формулировка данного запроса с помощью операции"U_проекции".

```
USING DURING ◄ SP_DURING { S#, DURING } ►
```

А поскольку уже была разработана следующая формулировка запроса в с помощью операции "U_MINUS", то уже была достигнута одна из первоначальных целей данной главы, а именно, был найден (гораздо более!) лучший способ формулировки запросовA и B.

```
USING DURING ◄ S_DURING { S#, DURING }
               MINUS
               SP_DURING { S#, DURING } ►
```

Реляционные операции сравнения

Строго говоря, реляционные операции сравнения (или просто реляционные сравнения) не являются реляционными операциями как таковыми, поскольку они возвращают истинностное значение, а не отношение. Тем не менее, на них можно распространить такую же трактовку, которая применялась в данной главе к реляционным операциям, причем фактически и решить такую задачу весьма желательно. Дело в том, что если рассматриваемые отношения включают интервальные атрибуты, то часто требуется сравнить некоторые распакованные аналоги этих отношений, а не сами отношения как таковые. Для этой цели вначале определим аналог "U_" для обычного оператора "реляционного сравнения на равенство". А именно, определим следующее выражение.

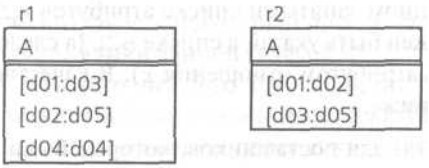
```
USING ( ACL ) ◄ rl = r2 ►
```

Это выражение является сокращенным обозначением для приведенного ниже выражения.

```
( UNPACK rl ON ( ACL ) ) = ( UNPACK r2 ON ( ACL ) )
```

Каждый атрибут, указанный в списке ACL, должен быть интервальным атрибутом и обязан присутствовать и в отношении rl, и в отношении r2. Обратите внимание на то, что вопрос о применении заключительного шага PACK не возникает, поскольку, как уже было сказано, результат операции "=" представляет собой истинностное значение, а не отношение.

В качестве примера рассмотрим отношения rl и r2, которые определены следующим образом.



В таком случае результатом сравнения rl = r2 становится FALSE, а выражение USING A ◄ rl = r2 ► принимает значение TRUE.

Указанный выше оператор будет именоваться для краткости как "U_=" (фактически он представляет собой именно оператор эквивалентности, который определен для n-арных отношений в самом конце предыдущего раздела). По такому же принципу можно определить аналогичные операторы "U_" для всех других реляционных операторов сравнения ('V","c", "c", "r>" и "з"). Например, если отношения rl и r2 остаются такими же, как и в примере применения операции "и_=", то выражение USING A ◄ rl с r2 ► равно TRUE, а выражение USING A ◄ rl с r2 ►равно FALSE.

Дополнительные сведения об обычных реляционных операциях

Еще раз рассмотрим оператор U_MINUS. Напомним, что этот оператор определен с помощью следующего выражения.

```
USING ( ACL ) ◄ r1 MINUS r2 ►
```

Это выражение является сокращенным обозначением для приведенного ниже выражения.

```
PACK
    ( ( UNPACK rl ON ( ACL ) ) MINUS ( UNPACK r2 ON ( ACL
) ) ) ON ( ACL )
```

Теперь предположим, что список ACL пуст (т.е. в нем вообще не указаны атрибуты), поэтому рассматриваемое выражение принимает следующий вид.

```
USING ( ) ◄ rl MINUS r2 ►
```

В таком случае его полная форма выглядит так, как показано ниже.

```
PACK                                                '' -
    ( ( UNPACK rl ON ( ) ) MINUS ( UNPACK r2 ON (
) ) ) ON ( )
```

Теперь напомним, что, как указано в предыдущем разделе, оба выражения, UNPACK r ON () и PACK r ON (), сокращаются просто до r. Поэтому общее выражение сокращается до следующего выражения.

```
rl MINUS r2
```

Иными словами, обычная реляционная операция MINUS фактически представляет собой просто частный случай операции UMINUS! Поэтому, если мы переопределим синтаксическую структуру обычного оператора MINUS следующим образом:

[USING (ACL)] ◄ <*relation exp*> MINUS <*relation exp*> ►

и разрешим исключать спецификацию USING (а также затененные концы стрелок, "◄" и "►", обозначающие начало и конец остального выражения) тогда и только тогда, когда список ACL пуст, то нам больше не потребуется вообще вести речь о каком-то специальном операторе "U_MINUS", поскольку все вызовы оператора MINUS фактически становятся вызовами оператора U_MINUS и появляется возможность обобщить смысл оператора MINUS соответствующим образом.

Аналогичные замечания относятся ко всем другим реляционным операторам, а также к реляционным сравнениям, поскольку в любом случае обычный реляционный оператор по сути является просто частным случаем соответствующего оператора "U_", в котором спецификация USING вообще не содержит атрибутов, и поэтому можно позволить себе исключить эту спецификацию (а также затененные концы стрелок, обозначающие начало и конец остального выражения). Иначе говоря, все операторы "U_" являются просто непосредственными обобщениями своих обычных аналогов. Поэтому больше вообще нет никакой необходимости явно упоминать операторы "U_" как таковые (и эти операторы действительно больше не будут применяться, кроме как время от времени, для более четкого обозначения понятия, о котором идет речь); вместо этого достаточно лишь признать, что обычные операторы допускают, но не требуют указания некоторого дополнительного операнда, когда они применяются к отношениям с интервальными атрибутами. Поэтому автор просит читателя обратить особое внимание на то, что до конца этой главы все ссылки на реляционные операторы и все ссылки на реляционные сравнения относятся к их обобщенным версиям (если явно не указано иное). Но в качестве пояснения иногда по мере необходимости будут явно использоваться уточнители "обычный" (или классический) и "обобщенный" при упоминании соответствующих операторов и сравнений; как уже было сказано, по той же причине иногда будет также явно использоваться уточнитель "U_".

23.6. ПРОЕКТ БАЗЫ ДАННЫХ

В процессе проектирования хронологических баз данных возникают некоторые особые проблемы. Для иллюстрации некоторых из этих проблем еще раз пересмотрим наш рабочий пример следующим образом: во-первых, полностью удалим информацию о поставках, во-вторых, переопределим информацию об имени, статусе и городе поставщика. А выбранный нами проект[9] для этой пересмотренной базы данных представим немедленно, как показано ниже.

```
VAR S SINCE BASE RELATION
    { S#     S#,      S# SINCE
    DATE, SNAME  NAME,    SNAME SINCE
    DATE, STATUS INTEGER, STATUS SINCE
    DATE, CITY   CHAR,    CITY SINCE
    DATE } KEY { S# } ;

VAR S DURING
    BASE
    RELATION
    { S#     S#,
      DURING  INTERVAL^DATE  }
```

[9] Обратите особое внимание на то, что в этом проекте переменная отношения SSINCE отличается от переменной отношения SSINCE из раздела 23.2.

```
    KEY { S#, DURING } ;

VAR S STATUS DURING
    BASE RELATION
    { S#     S#,
      STATUS INTEGER,
      DURING INTERVAL DATE }
    KEY { S#, DURING } ;

VAR
    S NAME DURING
    BASE RELATION
     { s#    s#,
    SNAME  NAME,
    DURING INTERVAL DATE }
    KEY { S#, DURING } ;

VAR
    S CITY DURIN
    G       BASE
    RELATION   {
    S#     S#,
      CITY   CHAR,
      DURING INTERVAL DATE }
      KEY { S#, DURING } ;
```

Предикаты переменных отношения этого проекта описаны ниже.

- S_SINCE. Поставщик S# работал по контракту начиная от временной позиции S#_SINCE, носил имя SNAME начиная от временной позиции SNAME_SINCE, имел статус STATUS начиная от временной позиции STATUS_SINCE и находился в городе CITY начиная от временной позиции CITY_SINCE.
- S_DURING. Поставщик s# работал по контракту на протяжении интервала DURING.
- S_NAME_DURING. Поставщик s# носил имя SNAME на протяжении интервала DURING.
- S_STATUS_DURING. Поставщик s# имел статус STATUS на протяжении интервала DURING.
- S_CITY_DURING. Поставщик S# находился в городе CITY на протяжении интервала DURING.

Кроме того, необходимо добавить, что в случае четырех предикатов, в которых применяется выражение "на протяжении", сутки, являющиеся конечной позицией интервала DURING, находятся в прошлом (см. подраздел "Определение «движущейся по временной шкале позиции текущего времени»" ниже в этом разделе).

Вполне очевидно, что в проекте, который был выбран автором, текущая информация хранится в "переменной отношения с атрибутами в виде позиции", а историческая информация — в множестве "переменных отношения с атрибутами в виде интервала". Такое разграничение переменных отношения по типам называется *горизонтальной декомпозицией*. Кроме того, историческая информация хранится в виде множества из нескольких отдельных переменных отношения (неформально говоря, по одному отношению для каждого отдельного "свойства" поставщика), и такое разделение называется *вертикальной декомпозицией*. Целесообразность применения таких декомпозиций будет обоснована в следующих подразделах.

Горизонтальная декомпозиция

Целесообразность применения горизонтальной декомпозиции обусловлена просто тем, что она обеспечивает четкое логическое разделение исторической и текущей информации, как описано ниже.

- Применительно к исторической информации, известно и время начала, и время окончания соответствующего интервала.
- В отличие от этого, для текущей информации известно время начала, но не известно время окончания.

Иными словами, здесь применяются разные предикаты, а этот факт является очень весомым доводом в пользу того, что решение, предусматривающее распределение исторической и текущей информации по разным переменным отношения, вполне оправданно.

Примечание. Фактически и то, и другое из приведенных выше утверждений в определенном смысле являются слишком упрощенными, но ими вполне можно руководствоваться при решении данной задачи.

Но следует отметить, что переменная отношения S_SINCE с "текущей" информацией имеет четыре атрибута, "определяющих временную позицию", по одному для каждого из атрибутов "не определяющих временную позицию". С другой стороны, в литературе можно встретить такие указания (под рубрикой "применение временных отметок к кортежам"), что достаточно ввести единственный атрибут, "обозначающий фиксированную позицию во времени" (SINCE), следующим образом.

```
S_SINCE { S#, SNAME, STATUS, CITY, SINCE }
```

Но если мы попытаемся сформулировать и проанализировать приведенный ниже предикат для такого проекта, то можем легко обнаружить, какая в нем допущена ошибка. Начиная от суток SINCE, все четыре следующих высказывания были истинными:

а) поставщик S# работал по контракту;
б) поставщик S# носил имя SNAME;
в) поставщик S# имел статус STATUS;
г) поставщик S# находился в городе CITY.

Например, предположим, что эта переменная отношения в данный момент включает следующий кортеж.

S#	SNAME	STATUS	CITY	SINCE
S1	Smith	20	London	d04

Допустим также, что сегодня — десятые сутки и что начиная с сегодняшнего дня статус поставщика S1 должен измениться на 30, и поэтому показанный выше кортеж необходимо заменить следующим кортежем.

S#	SNAME	STATUS	CITY	SINCE
S1	Smith	30	London	d10

В результате (кроме всего прочего) была потеряна информация, что поставщик S1 находился в Лондоне начиная от четвертых суток. Вообще говоря, должно быть очевидно,

что такой проект не позволяет представить какую-либо информацию о рассматриваемом поставщике, которая предшествует времени самого последнего обновления данных об этом поставщике (выражаясь немного неформально). Эту мысль можно выразить еще таким образом, что недостатком атрибута SINCE с временной отметкой является "слишком большая сфера действия временной отметки"; фактически эта временная отметка распространяется на комбинацию высказываний, состоящую из *четырех различных высказываний* (поставщик работает по контракту, поставщик имеет имя, поставщик имеет статус, поставщик находится в городе), а не просто на одно высказывание. В отличие от этого, в проекте, выбранном автором, каждое высказывание имеет собственную временную отметку.

Вертикальная декомпозиция

Безусловно, даже несмотря на наличие четырех отдельных атрибутов, "обозначающих начало интервала", переменная отношения S_SINCE является только полухронологической и поэтому требуются также переменные отношения, "обозначающие весь интервал", чтобы можно было представить в них историческую информацию. Но почему для этой исторической информации требуется вертикальная декомпозиция? Для того чтобы исследовать данный вопрос, предположим обратное, что определена только одна переменная отношения, "обозначающая весь интервал", которая выглядит примерно следующим образом.

```
S_DURING { S#, SNAME, STATUS, CITY, DURING }
```

Ниже приведен предикат этой переменной отношения.

На протяжении интервала DURING все следующие четыре высказывания были истинными:

. а) поставщик S# работал по контракту;

б) поставщик S# носил имя SNAME;

в) поставщик S# *имел статус STATUS;*

г) поставщик Si находился в городе CITY.

Как и при анализе версии переменной отношения S_SINCE только с одним атрибутом, "обозначающим начало интервала", которая рассматривалась в предыдущем подразделе, из краткого анализа этого предиката должно быть сразу же ясно, что проект данной переменной отношения разработан не очень удачно. Например, предположим, что эта переменная отношения в настоящее время включает следующий кортеж.

S#	SNAME	STATUS	CITY	DURING
S2	Jones	10	Paris	[d02:d04]

Допустим также, что теперь стало известно, что, во-первых, статус поставщика S2 действительно был равен 10 во вторые и третьи сутки, но стал равен 15 в четвертые сутки, и, во-вторых, поставщик S2 действительно находился в Париже в третьи и четвертые сутки, но во вторые сутки должен был находиться в Лондоне. В таком случае в эту переменную отношения придется внести довольно сложный ряд обновлений для того, чтобы она отражала изменения, которые произошли в реальном мире. А именно, необходимо заменить существующий кортеж тремя кортежами, которые выглядят следующим образом.

S#	SNAME	STATUS	CITY	DURING
S2	Jones	10	London	[d02:d02]

S#	SNAME	STATUS	CITY	DURING
S2	Jones	10	Paris	[d03:d03]

S#	SNAME	STATUS	CITY	DURING
S2	Jones	15	Paris	[d04:d04]

Теперь следует отметить, что для представления информации о том, что статус на протяжении интервала [d02:d03] был равен 10, применяются два отдельных кортежа вместо одного, а для представления информации о том, что на протяжении интервала [d03 :d04] городом поставщика был Париж, также используются два отдельных кортежа вместо одного.

Как показывает этот пример, задача такого обновления переменной отношения S_DURING, чтобы она постоянно соответствовала реальному миру, вообще говоря, становится весьма сложной. И в этом случае проблема фактически заключается в том, что атрибут с временной отметкой (теперь это атрибут DURING) "имеет слишком большую сферу действия"; в действительности эта временная отметка снова распространяется на комбинацию высказываний, состоящую из четырех различных высказываний. Решение состоит в том, что эти четыре высказывания необходимо распределить по четырем отдельным переменным отношения следующим образом.

```
S_DURING          { S#, DURING }
                KEY { S#, DURING }

S NAME DURING    { S#, SNAME, DURING }
                KEY { S#, DURING }

S STATUS DURING { S#, STATUS, DURING }
                KEY { S#, DURING }

S CITY DURING    { S#, CITY, DURING }
                KEY { S#, DURING }
```

Переменная отношения S_DURING показывает, когда тот или иной поставщик работал по контракту, переменная отношения S_NAME_DURING показывает, когда тот или иной поставщик носил то или иное имя, переменная отношения S_STATUS_DURING показывает, когда тот или иной поставщик имел тот или иной статус, а переменная отношения S_CITY_DURING показывает, когда тот или иной поставщик находился в том или ином городе.

Шестая нормальная форма

Описанная выше вертикальная декомпозиция весьма напоминает (как по смыслу, так и по назначению) классическую нормализацию, поэтому имеет смысл уделить внимание более подробному изучению указанной аналогии. Безусловно, вертикальная декомпозиция представляет собой именно то, чему всегда была посвящена классическая теория нормализации; в этой теории операцией декомпозиции служит проекция (которая по

определению является операцией вертикальной декомпозиции), а соответствующей операцией рекомпозиции является соединение. Безусловно, как было описано в главе 13, именно по этим причинам окончательную (с точки зрения классической теории нормализации) нормальную форму, т.е. пятую нормальную форму (сокращенно 5НФ) иногда называют проекционно-соединительной нормальной формой.

Примечание. Поскольку эти замечания относятся именно к классической нормализации, здесь упоминания операций проекции и соединения следует понимать как обозначающие классические версии этих операций, а не обобщенные версии, представленные в разделе 23.5.

Теперь следует отметить, что еще до начала исследований в области хронологических данных некоторые теоретики (см., например, [14.21]), выдвигали доводы в пользу максимально возможной декомпозиции переменных отношения, а не просто их декомпозиции до такой степени, которой требует классическая теория нормализации. Общая идея состояла в том, что переменные отношения должны быть приведены к неприводимым компонентам [14.21]; под этим подразумеваются такие компоненты, для которых дальнейшая декомпозиция без потерь становится невозможной. В случае нехронологических переменных отношения доводы в пользу "декомпозиции до конца" не очень убедительны, но они становятся гораздо более весомыми в случае переменной отношения наподобие S_DURING из предыдущего подраздела (первая версия, только с одним атрибутом, "обозначающим интервал времени"). Такие данные, как имя, статус и город поставщика, изменяются во времени независимо друг от друга. Более того, вполне вероятно, что одни из указанных атрибутов будут изменяться чаще, а другие — реже. Например, может оказаться, что имя поставщика вряд ли вообще когда-либо изменится, в то время как местонахождение одного и того же поставщика иногда будет изменяться, а изменения соответствующего статуса будут проходить весьма часто. Кроме того, по-видимому, история смены таких атрибутов поставщиков, как имя, статус и город, отдельно взятых, является понятием, более интересным и доступным для восприятия, чем понятие объединенной истории "имя-статус - город", поэтому и предложена такая вертикальная декомпозиция.

Напомним, что пятая нормальная форма основана на так называемых *зависимостях соединения* (Join Dependency— JD). В качестве напоминания отметим также, что переменная отношения R удовлетворяет JD * {A, B, ...,Z}(где A, B, . . . , Z — подмножества атрибутов R) тогда и только тогда, когда каждое допустимое значение R равно соединению его проекций по атрибутам A, в, . . ., Z, т.е. тогда и только тогда, когда может быть выполнена декомпозиция R без потерь по этим проекциям. А поскольку в данной главе было обобщено определение операции соединения, то можно обобщить соответствующим образом определение зависимости соединения, а затем определить новую ("шестую") нормальную форму, основанную на таком обобщенном понятии зависимости соединения. Соответствующие определения приведены ниже.

- Допустим, что R— переменная отношения, A, B, . . ., Z — подмножества атрибутов R, а ACL — разделенный запятыми список атрибутов R со значениями в виде интервала. Тогда можно утверждать, что R удовлетворяет обобщенной **зависимости соединения**

  USING (ACL) * { A, B, ..., Z }

  тогда и только тогда, когда приведенное ниже выражение является истинным для любого допустимого значения R.

USING (ACL) ◄ R = R' ►

Здесь R' — **и_соединение U_проекций** переменной отношения R по атрибутам A, B, . . ., Z, притом что все рассматриваемые U_соединения и U_проекции включают спецификацию US ING в форме US ING {ACL).

Примечание. В этом определении по умолчанию признается истинным тот факт, что операция U_**соединения,** как и операция соединения, является ассоциативной; это означает, что определение, в котором речь идет об "операции" U_**соединения** любого количества отношений, не содержит противоречия. Следует также отметить, что утверждение о том, что приведенное выше выражение является истинным, равносильно утверждению, что R и R' эквивалентны (по отношению к списку ACL); см. обсуждение понятия эквивалентности в самом конце раздела 23.4. Переменная отношения R находится в шестой нормальной форме (сокращенно 6НФ) тогда и только тогда, когда она удовлетворяет всем нетривиальным зависимостям соединения вообще, где зависимость соединения является тривиальной тогда и только тогда, когда по меньшей мере одна из рассматриваемых проекций (возможно, U_**проекций)** выполняется по всем атрибутам указанной переменной отношения.

Из этого определения непосредственно следует, что каждая переменная отношения, которая находится в 6НФ, находится также в 5НФ. Кроме того, из него также непосредственно следует, что любая переменная отношения находится в 6НФ тогда и только тогда, когда она неприводима, в том смысле, который определен выше.

Итак, отметим, что версия переменной отношения S_DURING, которая имеет атрибуты s#, SNAME, STATUS, CITY и DURING, согласно этому определению не находится в 6НФ, поскольку она обладает следующими характеристиками:

а) эта переменная отношения удовлетворяет обобщенной зависимости соединения USING DURING * {SND, STD, SCD} (где ИМЯ "SND" ОТНОСИТСЯ К множеству атрибутов {S#, SNAME,DURING}, а имена "STD" и "SCD" имеют аналогичные определения);

б) эта зависимость соединения, безусловно, является нетривиальной.

Поэтому автор рекомендует выполнить декомпозицию указанной переменной отношения на проекции 6НФ, как было описано в предыдущем подразделе.

Примечание. Читатель мог заметить, что в приведенном выше примере было бы достаточно выполнить декомпозицию только на три переменные отношения, а не на четыре переменные отношения 6НФ, поскольку, строго говоря, в этой декомпозиции переменная отношения S_DURING с атрибутами s# и DURING не нужна; это связано с тем, что она всегда равна (обобщенной) проекции по атрибутам s# и DURING любой из трех остальных переменных отношения. Тем не менее, автор предпочел включить переменную отношения S_DURING в общий проект отчасти просто для полноты, а частично из-за того, что благодаря такому включению удается в определенной степени предотвратить создание громоздкого и надуманного проекта, который появился бы в противном случае [23.4].

Определение "движущейся по временной шкале позиции текущего времени"

Снова вернемся на время к обсуждению вопроса горизонтальной декомпозиции (т.е. разделения базы данных на переменные отношения "с данными, определяющими временную позицию" и "с данными, обозначающими временной интервал"). Безусловно, что в этой базе данных недостаточно иметь только переменные отношения, "определяющие временную позицию", поскольку такие переменные отношения являются просто полухронологическими и не могут представлять историческую информацию. Тем не менее, в базе данных могут применяться лишь переменные отношения с данными, "обозначающими временной интервал", но только если мы согласимся с тем, что в базе данных будет содержаться ложная информация (как описано ниже).

Рассмотрим случай с поставщиком, контракт которого еще не закончился. Безусловно, вполне возможно, что предполагаемое время завершения этого контракта известно, но, вообще говоря, чаще всего известно лишь то, что контракт продлевается автоматически (в качестве примера можно указать типичные контракты, которые заключаются при найме на работу). Поэтому, какое бы значение не было указано в качестве атрибута END (DURING) для такого поставщика в переменной отношения с данными, "обозначающими временной интервал", оно все равно скорее всего окажется ложным. Безусловно, можно было бы и даже, по-видимому, нужно принять соглашение, что такие значения атрибута END (DURING) должны задаваться как последние сутки рассматриваемого временного интервала[10] (т.е. как значение, возвращаемое функцией LAST_DATE ()). Но следует отметить, что такая схема означает, что если значение "последние сутки" появится в результатах запроса, то пользователь, вероятно, обязан интерпретировать это значение как "до дополнительного извещения", а не собственно как "последние сутки". Иными словами, утверждать, что атрибут END (DURING) для такого поставщика имеет значение "последние сутки", означает почти наверняка — давать ложную информацию.

Именно для того, чтобы избежать необходимости включать в базу данных такую ложную информацию, некоторые авторы (см., например, [23.2]) предложили использовать специальный "маркер NOW" (сейчас) для обозначения того, что в разделе 23.1 было определено как "движущаяся по временной шкале позиция текущего времени" (иными словами, для обозначения понятия "до дополнительного извещения"). Основная идея состоит в том, что нужно разрешить включать этот специальный маркер там, где, во-первых, разрешено применение соответствующего позиционного временного типа, и, во-вторых, действительно придать ему намеченную интерпретацию как "до дополнительного извещения". Таким образом, например, переменная отношения S_DURING может включать, допустим, кортеж с данными о поставщике S1 со значением DURING, равным [d04 :NOW], а не [d04 :d99]. (Здесь предполагается, что 99-е сутки — это последние сутки и данное конкретное появление значения d99 действительно должно рассматриваться как имеющее смысл "до дополнительного извещения", а не обозначает 99-е сутки как таковые.)

Но, по мнению автора, введение маркера NOW представляет собой непродуманный отход от здравых реляционных принципов. Отметив, что NOW фактически является переменной, укажем, что данный подход связан с введением очень странного понятия —

---

[10] Безусловно, можно заменять это искусственное значение истинным значением после того, как это истинное значение становится известно.

значений (точнее, интервальных значений), которые содержат переменные". Автор считает, что с точки зрения логики этот подход не выдерживает никакой критики. Ниже приведены некоторые примеры того, какие вопросы, требующие однозначного ответа, возникают в связи с анализом концепции NOW.

- Допустим, что i— интервал [N0W:dl4], t — кортеж, содержащий интервал i, а сегодня — десятые сутки. В таком случае кортеж t может рассматриваться как своего рода сокращенное обозначение пяти отдельных кортежей, содержащих единичные интервалы, соответственно, [d10:d10], [dll:dll], [d12:d12], f dl 3 : dl 3] и [d14:d14]. Но после того, как время подойдет к полуночи в деся тые сутки, первый из этих кортежей должен быть (по сути) автоматически удален! Аналогичная ситуация возникнет в 11-е сутки, 12-е сутки, 13-е сутки, ..., и что фактически произойдет в полночь в 14-е сутки?
- Каковым является результат сравнения d9 9 = NOW?
- Каковым является значение выражения "N0W+1" или "N0W-1"?
- Если il и i2 представляют собой, соответственно, интервалы [d01:NOW] и [d06:d07], то являются ли эти интервалы смежными или перекрываются?
- Каковым является результат распаковки отношения, содержащего кортеж, в кото ром интервальный атрибут, предназначенный для распаковки, имеет значение [d04:NOW]?
- Какова кардинальность множества { [d01:NOW], [d01:d04] }?

Этот список вопросов можно продолжать (поскольку он — далеко не исчерпывающий). Автор считает, что на подобные вопросы трудно дать какие-либо обоснованные ответы; очевидно, что автор предпочитает использовать такой подход, который не опирается на какие-либо подобные подозрительные понятия, как NOW, и на значения, содержащие переменные. А именно такие рассуждения автора служат основным доводом в пользу горизонтальной декомпозиции, при которой MapKepNOW не требуется.

23.7. ОГРАНИЧЕНИЯ ЦЕЛОСТНОСТИ

В данном разделе рассматривается проблема ограничений целостности, которые применяются к хронологическим данным. В разделе 23.2 было показано, насколько сложно даже сформулировать правильно эти ограничения в отсутствие должной поддержки интервалов, а в этом разделе показано, как можно решить эту проблему с применением понятий, представленных в предыдущих разделах.

Для определенности сосредоточим наше внимание (пока не будет указано иное) на переменной отношения S_STATUS_DURING из предыдущего раздела, которое имеет следующее определение.

```
S_STATUS_DURING { S#, STATUS, DURING }
                KEY { S#, DURING }
```

" В действительности, маркер NOW напоминает значение NULL, поскольку внедрение понятия NULL также влечет за собой появление значений, содержащих нечто, не являющееся значением (см. главу 19).

Теперь перейдем к анализу в следующих трех подразделах тех трех основных проблем, которые могут возникать в хронологических переменных отношения, подобных указанной переменной отношения. Эти проблемы будут именоваться, соответственно, *проблемой избыточности, проблемой многословия* и *проблемой противоречия*.

Проблема избыточности

Ограничение KEY для переменной отношения S_STATUS_DURING, хотя и логически правильное, в определенном смысле является недостаточным. А именно — оно не в состоянии предотвратить появление в этой переменной отношения, например, обоих следующих кортежей одновременно.

S#	STATUS	DURING
S4	25	[d05:d06]

S#	STATUS	DURING
S4	25	[d06:d07]

Вполне очевидно, что эти два кортежа обнаруживают определенную избыточность, поскольку статус поставщика S4 в шестые сутки фактически был указан дважды. Безусловно, было бы лучше заменить эти два кортежа одним следующим кортежем.

S#	STATUS	DURING
S4	25	[d05:d07]

Теперь следует отметить, что если бы эти два первоначальных кортежа были единственными кортежами в некотором отношении, состоящем из двух кортежей, и была бы выполнена упаковка этого отношения по атрибуту DURING, то в результате появилась бы отношение с одним кортежем, содержащее лишь один показанный выше кортеж. Поэтому неформально можно утверждать, что показанный выше кортеж — это "упакованный" кортеж, полученный путем упаковки двух первоначальных кортежей по атрибуту DURING (здесь используется выражение "неформально", поскольку операция упаковки фактически применяется к отношениям, а не к кортежам). Поэтому в действительности нам требуется заменить эти два первоначальных кортежа таким "упакованным" кортежем. На самом деле, как было указано в разделе 23.2, отказ от выполнения такой замены (при котором допускается присутствие обоих первоначальных кортежей) является столь же плохой практикой, как и отсутствие контроля за появлением дубликатов кортежей (ведь если бы было разрешено появление в отношении дубликатов кортежей, это также стало бы причиной появления определенного вида избыточности). И действительно, если в переменной отношения присутствуют оба первоначальных кортежа, то она нарушает свой собственный предикат! Например, кортеж, показанный справа, кроме всего прочего, равносилен утверждению, что поставщик S4 не имел статуса 25 в сутки, непосредственно предшествующие шестым суткам. А кортеж, показанный слева, кроме всего прочего, равносилен утверждению, что поставщик S4 все-таки имел статус 25 в пятые сутки, а эти пятые сутки, безусловно, являются сутками, предшествующими шестым суткам.

Проблема многословия

Ограничение KEY для переменной отношения S_STATUS_DURING является также неудовлетворительным по другой причине. А именно — оно не позволяет предотвратить появление в этой переменной отношения, например, обоих следующих кортежей одновременно.

S#	STATUS	DURING
S4	25	[d05:d05]

S#	STATUS	DURING
S4	25	[d06:d07]

В данном случае избыточность как таковая отсутствует, но имеет место своего рода "многословие", поскольку используются два кортежа для представления такой информации, которую лучше было бы представить с помощью следующего одного "упакованного" кортежа (фактически такого же, как указано выше).

S#	STATUS	DURING
S4	25	[d05:d07]

В действительности, можно легко показать, что отказ от замены двух первоначальных кортежей этим "упакованным" кортежем снова означал бы, что переменная отношения нарушает свой собственный предикат.

Устранение проблем избыточности и многословия

Должно быть очевидно, что для предотвращения указанных выше нарушений, связанных с избыточностью и многословием, необходимо ввести ограничение (назовем его ограничением А), разработанное в соответствии с приведенной ниже схемой.

Ограничение А. Если в любой определенный момент времени переменная отношения S_STATUS_DURING содержит два разных кортежа, которые являются полностью идентичными, за исключением своих значений i1 и i2 атрибута DURING, то выражение il MERGES i2 должно быть ложным.

Напомним, что, неформально выражаясь, операция MERGES является результатом применения логической операции "ИЛИ" к операциям OVERLAPS и MEETS: если бы мы вместо нее включили в ограничение А операцию OVERLAPS, то получили бы ограничение, которое необходимо ввести в действие, чтобы избежать возникновения проблемы избыточности, а если бы вместо операции MERGES была включена в ограничение А операция MEETS, то было бы получено ограничение, которое необходимо ввести в действие, чтобы избежать возникновения проблемы многословия.

Должно быть также очевидно, что имеется очень простой способ поддержки ограничения А, т.е. для этого необходимо, чтобы переменная отношения постоянно оставалась упакованной по атрибуту DURING. Поэтому введем новое ограничение PACKED ON, которое может присутствовать в определении переменной отношения, как показано ниже.

```
VAR S STATUS DURING BASE RELATION
  ( S# S#, STATUS INTEGER, DURING
  INTERVAL DATE } PACKED ON DURING KEY { S#,
  DURING } ;
```

В этом определении конструкция PACKED ON DURING представляет собой ограничение для переменной отношения S_STATUS_DURING (оно действительно является ограничением для переменной отношения согласно схеме классификации, описанной в главе 9). Данное ограничение интерпретируется следующим образом: переменная отношения S_STATUS_DURING должна постоянно оставаться упакованной по атрибуту DURING. Таким образом, для решения проблем избыточности и многословия достаточно было ввести указанный выше специальный синтаксис; иными словами, этот синтаксис

позволяет решить проблему, примером проявления которой является ограничение, упоминавшееся под названием ограничения XFT1 в разделе 23.2.

Проблема противоречия

Ограничения PACKED ON и KEY все еще не позволяют решить задачу обеспечения целостности даже при их совместном использовании, т.е. они не позволяют предотвратить такую ситуацию, при которой переменная отношения будет включать одновременно, например, следующие два кортежа.

S#	STATUS	DURING
S4	10	[d04:d06]

S#	STATUS	DURING
S4	25	[d05:d07]

Здесь показано, что поставщик S4 в пятые и шестые сутки имеет одновременно и статус 10, и статус 25, — безусловно, такое состояние дел является недопустимым. Иными словами, налицо противоречие; фактически эта переменная отношения снова нарушает свой собственный предикат, поскольку предполагается, что в любые отдельно взятые сутки каждый поставщик должен иметь один и только один статус.

Решение проблемы противоречия

Должно быть очевидно, что для устранения противоречий, подобных только что описанному, необходимо ввести в действие еще одно ограничение (назовем его ограничением в), которое составлено в соответствии с приведенными ниже требованиями.

Ограничение в. Если в любой определенный момент времени переменная отношения S_STATUS_DURING содержит два кортежа S#, которые различаются по своему значению STATUS, то для них значения i1 и 12 атрибута DURING должны быть такими, что выражение i1 OVERLAPS i 2 является ложным.

Следует обратить особое внимание на то, что поддержку ограничения В (как уже было показано), безусловно, нельзя обеспечить просто за счет того, что переменная отношения будет постоянно оставаться упакованной по атрибуту DURING. Еще более очевидно то, что нельзя обеспечить выполнение этого ограничения лишь за счет того, что сочетание атрибутов {S#, DURING} будет определено как потенциальный ключ. Но предположим, что переменная отношения постоянно остается распакованной по атрибуту DURING (на время будем игнорировать тот факт, что данное предположение фактически выполнить невозможно, поскольку уже было обусловлено такое требование, что переменная отношения должна постоянно оставаться упакованной по атрибуту DURING). В таком случае будет поддерживаться описанное ниже состояние.

- Все значения DURING в этой распакованной форме будут представлять собой еди ничные интервалы и поэтому фактически будут соответствовать отдельным вре менным позициям.
- Единственным потенциальным ключом для этой распакованной формы будет продолжать оставаться множество атрибутов {S#, DURING}, поскольку в это вре мя любой отдельно взятый поставщик, работающий по контракту, будет иметь один и только статус в каждый конкретный момент времени.

Из этого следует, что если бы мы имели возможность ввести в действие такое ограничение, согласно которому множество атрибутов {S#, DURING} было бы потенциальным КЛЮЧОМ ДЛЯ распакованной формы UNPACK S_STATUS_DURING ON DURING, ТО В СИЛу
этого ввели бы в действие ограничение в. Поэтому автор предлагает создать новое ограничение, WHEN/THEN, которое может присутствовать в определении переменной отношения везде, где может присутствовать простое ограничениеКЕУ, как показано ниже.

```
VAR S STATUS DURING BASE RELATION
 { S# S#, STATUS INTEGER, DURING INTERVAL DATE }
 PACKED ON DURING WHEN UNPACKED ON DURING THEN KEY { S#, DURING }
  KEY { S#, DURING } ;
```

Здесь выражение WHEN UNPACKED ON DURING THEN KEY {s#, DURING} представляет собой ограничение для переменной отношения S_STATUS_DURING (оно опять-таки является ограничением для переменной отношения, как и описанное выше ограничение PACKED ON). Это выражение интерпретируется следующим образом: переменная отношения S_STATUS_DURING должна постоянно оставаться такой, чтобы никакие два кортежа в результатах выражения UNPACK S_STATUS_DURING ON DURING не имели одного и того же значения комбинации атрибутов {S#, DURING} (неформально можно утверждать, что " {S#, DURING} является потенциальным ключом для результатов выражения UNPACK S_STATUS_DURING ON DURING"). Таким образом, для решения проблемы противоречия также достаточно ввести указанную выше специальную синтаксическую конструкцию.

U_ключи

В отношении ограничений KEY, PACKED ON и WHEN/THEN можно было бы привести гораздо больше сведений [23.4]; но из-за ограничений по объему приведем лишь следующие выводы. Во-первых, автор предлагает предусмотреть возможность вводить в определение любой конкретной переменной отношения R сокращенные спецификации в указанной ниже форме.

```
USING ( ACL ) KEY { K }
```

Здесь и ACL, и к представляют собой разделенные запятыми списки имен атрибутов, где каждый указанный атрибут в ACL должен быть также указан в к (и, как обычно, круглые скобки разрешено исключать, если ACL содержит только одно имя атрибута). Эта спецификация определена как сокращенное обозначение комбинации из следующих трех ограничений.

```
PACKED ON ( ACL )
WHEN UNPACKED ON ( ACL ) THEN KEY { K }
KEY { K }
```

Автор предлагает сокращенно называть {К} как "U_ключ" (но см. приведенное ниже описание). С использованием этого сокращения, например, определение переменной отношения S_STATUS_DURING можно упрощенно представить следующим образом.

```
VAR S STATUS DURING BASE RELATION
  { S# S#, STATUS INTEGER, DURING
  INTERVAL DATE } USING DURING KEY { S#,
  DURING } ;
```

Теперь предположим, что в спецификации U_ключа для переменной отношения R разделенный запятыми список имен атрибутов ACL пуст, поэтому имеет место следующая конструкция.

```
USING ( ) KEY { K }
```

По определению эта спецификация является сокращенным обозначением следующей комбинации ограничений.

```
PACKED ON ( )
WHEN UNPACKED ON ( ) THEN KEY { K }
KEY { K }
```

Иными словами, в объявлении переменной отношения R заданы приведенные ниже требования.

1. Переменная отношения R должна оставаться упакованной вообще ни по одному атрибуту. Но операция упаковки отношения r вообще ни по одному атрибуту про сто возвращает r, поэтому неявно заданная спецификация PACKED ON не оказы вает никакого действия.
2. Переменная отношения R должна быть такой, что если она распакована вообще ни по одному атрибуту, то {к} является потенциальным ключом для полученного результата. Но операция распаковки отношения r вообще ни по одному атрибуту просто возвращает r, поэтому неявно заданная спецификация WHEN/THEN просто означает, что {к} является потенциальным ключом для R, и поэтому неявное ог раничение KEY становится избыточным.

Из этого следует, что вместо некоторых ограничений U_ключа можно в качестве сокращенного обозначения применять обычное ограничение KEY в форме KEY {К}, а именно сокращение в форме USING () KEY {К}. Иными словами, обычные ограничения KEY, по сути, представляют собой просто частный случай предложенного автором нового синтаксиса! Поэтому, если синтаксис обычного ограничения клиента KEY будет переопределен следующим образом:

```
[ USING ( ACL ) ] KEY { K }
```

и будет разрешено опускать спецификацию USING тогда и только тогда, когда список ACL пуст, то отпадает необходимость вообще вести речь о таком понятии, как U_ключи; все потенциальные ключи становятся фактически U_ключами и можно обобщить соответствующим образом понятие "потенциального ключа" (или просто "ключа"). Так мы и сделаем.

Не вдаваясь в дополнительные подробности, автор утверждает, что аналогичные обобщения относятся также и к понятию "внешнего ключа". Одним из следствий этого является то, что спецификация в форме

```
USING DURING FOREIGN KEY { S#, DURING
            } REFERENCES S_DURING
```

(часть определения переменной отношения S_STATUS_DURING) может использоваться для поддержки такого ограничения, что если в переменной отношения S_STATUS_DURING показан определенный поставщик как имеющий определенный статус на протяжении

определенного интервала времени, то переменная отношения S_DURING должна показывать, что тот же поставщик работает по контракту на протяжении того же интервала времени. Кроме того, аналогичный подход может использоваться для решения проблемы, примером которой является ограничение, упомянутое как ограничение XFT3 в разделе 23.2, и поэтому теперь достигнута еще одна из целей, первоначально поставленных в этой главе, а именно — обнаружен лучший способ формулировки ограничений, которые рассматривались в предыдущем разделе.

Девять требований

В заключение этого раздела следует отметить, что по вопросу поддержки ограничений в хронологической базе данных необходимо рассмотреть гораздо более широкий круг проблем, чем предложено в данной главе. В [23.4] приведен тщательный и подробный анализ всей этой темы; точнее, в указанной работе рассматривается в очень общих терминах набор из девяти требований, которым должна соответствовать типичная хронологическая база данных, подобная рассматриваемой здесь базе данных поставщиков и поставок. Список этих требований приведен ниже.

- Требование R1. Если в базе данных показан поставщик Sx как работающий по контракту в сутки d, то она должна содержать один и только один кортеж, который показывает этот факт.
- Требование R2. Если в базе данных показан поставщик Sx как работающий по контракту в сутки d и d+1, то она должна содержать один и только один кортеж, который показывает этот факт.
- Требование R3. Если в базе данных показан поставщик Sx как работающий по контракту в сутки d, то она должна также показывать поставщика Sx как имею щий определенный статус в сутки d.
- Требование R4. Если в базе данных показан поставщик Sx как имеющий опреде ленный статус в сутки d, то она должна содержать один и только один кортеж, ко торый показывает этот факт.
- Требование R5. Если в базе данных показан поставщик Sx как имеющий один и тот же статус в сутки d и d+1, то она должна содержать один и только один кортеж, который показывает этот факт.
- Требование R6. Если в базе данных показан поставщик Sx как имеющий опреде ленный статус в сутки d, то она должна также показывать поставщика Sx как ра ботающего по контракту в сутки d.
- Требование R7. Если в базе данных показан поставщик Sx как способный постав лять некоторую определенную деталь Py в сутки d, то она должна содержать один и только один кортеж, который показывает этот факт.
- Требование R8. Если в базе данных показан поставщик Sx как способный постав лять одну и ту же деталь Py в сутки d и d+1, то она должна содержать один и толь ко один кортеж, который показывает этот факт.

- Требование R9. Если в базе данных показан поставщик Sx как способный поставлять некоторую деталь Py в сутки d, то она должна также показывать поставщика Sx как работающего по контракту в сутки d.

В [23.4] глубоко проанализированы эти девять требований и показано, как можно их сформулировать в реляционно полном языке наподобие Tutorial D. В этой главе дальнейшее обсуждение указанной темы не приведено.

23.8. РЕЗЮМЕ

Потребность в создании баз данных (особенно *хранилищ данных),* содержащих хронологические данные, постоянно возрастает. *Хронологические данные* могут рассматриваться как закодированное представление высказываний с временными отметками. В число указанных высказываний могут входить высказывания с информацией о начальной позиции на временной шкале (предназначенные для представления текущих данных) и высказывания с информацией об интервале времени (предназначенные для представления исторических данных); в этой главе дано очень точное определения двух указанных терминов. А именно — выражение "от" определено как обозначающее именно "начиная от указанной позиции на временной шкале и не включая непосредственно предшествующего времени", а выражение "на протяжении интервала времени" рассматривается как обозначающее "на протяжении указанного интервала времени, не включая того времени, которое непосредственно предшествует или непосредственно следует за этим интервалом".

Затем в этой главе был представлен очень простой рабочий пример (база данных поставщиков и поставок) и были выполнены следующие действия: во-первых, полухронологизация этой базы данных путем введения атрибутов SINCE, и, во-вторых, ее полная хронологизация путем введения атрибутов FROM и TO. Кроме того, было показано, что внедрение обоих этих проектов приводит к значительному возрастанию сложности формулировок ограничений и запросов. Поэтому была предложена идея рассматривать интервалы как самостоятельные значения, т.е. было определено понятие позиционного типа и генератора типа **INTERVAL,** после чего дано описание соответствующих операторов селектора интервала, а также операторов **BEGIN** и **END.** В дальнейшем было определено много других операторов для позиций и интервалов, включая операторы Аллена и такие операторы с интервалами, как **UNION, INTERSECT** и **MINUS.**

На следующем этапе были определены два чрезвычайно важных реляционных оператора, называемых **PACK** и **UNPACK** (а в качестве промежуточного этапа на пути к их определению были введены два более простых оператора на унарных отношениях, называемые **COLLAPSE** и **EXPAND).** Операторы EXPAND и UNPACK позволяют сосредоточиться на изучении информационного наполнения их реляционных фактических параметров на уровне неразрывных компонентов, не беспокоясь о том, что существует множество различных способов, с помощью которых эта информация может быть объединена в промежуточные "конгломераты". Аналогичным образом, операторы COLLAPSE и PACK позволяют сосредоточиться на информационном наполнении их реляционных фактических параметров, представленном в сжатой ("конгломерированной") форме, не беспокоясь о том, что может существовать вероятность сопряжения или перекрытия отдельных "конгломератов". Было также показано, как можно воспользоваться операторами PACK и UNPACK для упрощения формулировок хронологических запросов. Кроме того, эти

операторы использовались в качестве основы для определения **обобщенных** версий (или версий "U_") обычных реляционных операторов (U_JOIN, U_MINUS, U_**проекция** и т.д.). Затем было показано, что все эти обычные реляционные операторы фактически являются просто частными случаями обобщенных версий.

После этого было проведено исследование некоторых проблем **проектирования баз данных** и даны рекомендации использовать, во-первых, **горизонтальную декомпозицию** для разделения текущей и исторической информации и, во-вторых, **вертикальную декомпозицию** для разделения информации, касающейся различных "свойств" одной и той же "сущности" (выражаясь очень неформально). В действительности, тем самым была определена новая нормальная форма, **6НФ** (шестая нормальная форма).

После этого были проанализированы некоторые проблемы, которые могут быть связаны с использованием хронологических данных в отсутствие приемлемых ограничений целостности (а именно проблемы **избыточности, многословия и противоречия)** и было показано, как могут применяться ограничения **PACKED ON И WHEN/THEN** для решения указанных проблем. Была определена обобщенная версия обычного ограничения KEY, получившая названия ограничений U_ключа, и затем показано, что обычное ограничение KEY фактически является просто частным случаем обобщенной версии.

Ниже приведены два заключительных замечания к данной главе.

- Автор напоминает, что все новые понятия, представленные в данной главе, кроме генератора типа INTERVAL, в конечном итоге являются просто сокращенными обозначениями для некоторых других понятий, которые вполне могут быть выражены средствами реляционной модели.
- Рекомендуемый автором подход к проектированию (в частности горизонтальная декомпозиция) влечет за собой такие следствия, что применяемые запросы (а также, в случае необходимости, обновления) часто охватывают несколько переменных отношения и поэтому могут оказаться довольно сложными. В [23.4] приведен ряд предложений по введению дополнительных сокращений, позволяющих устранить также и эти сложности.

УПРАЖНЕНИЯ

23.1. Дайте определение термина *квант времени.* Дайте определение термина *позиция на временной шкале.* Приведите свою трактовку термина *детализация.*

23.2. Дайте определение терминов *позиционный тип* и *интервальный тип.*

23.3. Перечислите все известные вам доводы в пользу замены пары атрибутов FROM-то отдельным атрибутом DURING.

23.4. Допустим, что i — значение типа INTERVAL_INTEGER. Составьте выражение, обозначающее интервал, полученный путем продления интервала i на его собственную длину в обоих направлениях (например, когда интервал [5:7] становится равным [2:10]). При каких условиях вычисление этого выражения на этапе прогона окончится неудачей?

23.5. Снова допустим, что i — значение типа INTERVAL_INTEGER. Составьте выражение, обозначающее интервал, который представляет собой среднюю треть интервала i. При этом может быть принято предположение, что значение COUNT (i) кратно трем.

23.6. Допустим, что i1, i2, и i3 — такие интервалы, что существует единственный интервал i4, состоящий из всех позиций p, таких что p ∈ i1 или p ∈ i2 или p e i3. Составьте выражение, вычисление которого позволяет получить интервал i4.

23.7. Если a и Ь — отношения (или множества), то имеет место следующее тождество:

a INTERSECT b ≡ a MINUS (a MINUS b)

Является ли истинным аналогичное утверждение, если a и b — интервалы?

23.8. Приведите примеры:

а) отношения с двумя интервальными атрибутами (хронологическими или другими);

б) отношения с тремя интервальными атрибутами;

в) отношения, состоящего только из интервальных атрибутов.

23.9. Рассмотрим отношение r с двумя различными интервальными атрибутами, A1 и A2. Докажите или опровергните следующие тождества:

UNPACK (UNPACK r ON Al) ON A2 ≡ UNPACK (UNPACK r ON A2) ON Al

PACK (PACK r ON Al) ON A2 ≡ PACK (PACK r ON A2) ON Al

23.10. Даны следующие переменные отношения.

```
FEDERAL_GOVT { PRESIDENT, PARTY,
DURING } STATE_GOVT ( GOVERNOR, STATE,
PARTY, DURING }
```

При этом подразумевается, что семантика указанных переменных отношения очевидна (предполагается, что каждый из двух атрибутов DURING имеет тип INTERVAL_DATE; мы игнорируем тот факт, что время правления президента и губернатора обычно измеряется годами, а не сутками). Теперь предположим, что требуется получить примерно следующий результат.

```
RESULT ( PRESIDENT, GOVERNOR, STATE, PARTY, DURING }
```

Определенный кортеж должен появляться в этой результирующей переменной отношения тогда и только тогда, когда и указанный президент, и указанный губернатор принадлежат к указанной партии, а периоды их правления совпадают (притом что атрибут DURING точно определяет рассматриваемое совпадение). Составьте выражение, позволяющее получить этот результат.

23.11. Приведите пример переменной отношения с интервальным атрибутом, который нежелательно было бы хранить в упакованной форме.

23.12. Приведите пример выражения с оператором U_INTERSECT, результат вычисления которого имеет кардинальность больше чем у любого из отношений, к которым применяется **"U_пересечение".**

23.13. Рассмотрим оператор U_JOIN. Для упрощения предположим, что упаковка и рас паковка должны быть осуществляться на основе единственного атрибута A. Дока жите, что имеет место следующее тождество:

```
USING A < rl JOIN r2 ►
≡ WITH ( rl RENAME A AS X ) AS Tl ,
       ( r2 RENAME A AS Y ) AS T2 ,
       ( Tl JOIN T2 ) AS T3 , ( T3 WHERE X OVERLAPS Y ) AS T4 ,
       ( EXTEND T4 ADD ( X INTERSECT Y ) AS A ) AS
  T5 ,
       T5 { ALL BUT X, Y } AS T6 :
  PACK T6 ON A
```

Докажите также, что если оба отношения, rl и r2, были первоначально упакованы по атрибуту A, то заключительный шаг, в котором применяется оператор PACK, не нужен.

Примечание. В этом выражении оператор INTERSECT в шаге EXTEND является интервальным, а не реляционным оператором INTERSECT.

23.14. Дайте определение шестой нормальной формы (6НФ). Действительно ли можно рассматривать эту нормальную форму как "шестую", в том же смысле, в каком пятая нормальная форма (5НФ) рассматривается как пятая?

23.15. Понятие "движущейся по временной шкале позиции текущего времени" определяет не значение, а переменную. Приведите свое мнение по поводу этого утверждения.

23.16. Используя проект базы данных поставок, приведенный в разделе 23.2, составьте выражения Tutorial D для приведенных ниже запросов.

а) Определить номера поставщиков, которые в настоящее время способны поставлять не меньше двух разных деталей, показав в каждом случае дату, начиная от которой они были способны это сделать.

б) Определить поставщиков, которые в настоящее время не способны поставлять не меньше двух разных деталей, показав в каждом случае дату, начиная от которой они были неспособны это сделать.

23.17. Приведите свою трактовку проблем избыточности, многословия и противоречия.

23.18. Приведите свою трактовку следующих понятий:

а) ограничения PACKED ON;

б) ограничения WHEN/THEN;

в) ограничения U_ключей.

Объясните, почему ключи, определяемые согласно классической реляционной модели, могут рассматриваться как частный случай U_**ключей.**

СПИСОК ЛИТЕРАТУРЫ

Автор не приводит здесь полный список литературы, который вполне мог бы стать весьма обширным, а рекомендует читателю обратиться к исчерпывающей библиографии в [23.4].

23.1. Allen J. F. Maintaining Knowledge About Temporal Intervals // CACM. — November 1983.- 16, №11.

23.2. Clifford J., Dyreson C, Isakowitz T., Jensen C.S., Snodgrass R.T. On the Semantics of 'Now' in Databases//ACM TODS. - June 1997. - 22, № 2.

23.3. Darwen H., Date C J. An Overview and Analysis of Proposals Based on the TSQL2 Approach (статья еще не опубликована; это — намеченный заголовок). Предварительный чер новой вариант можно найти на Web-узле http: //www. thethirdmanif esto. com.

По-видимому, наиболее широко известным из ранее опубликованных предложений по решению задачи создания хронологических баз данных является TSQL2 [23.5]; на этом предложении основано также несколько других предложений. В данной работе приведен краткий обзор и критический анализ таких предложений, а также приведено их сравнение с подходом, предложенным в настоящей главе.

23.4. Date C. J., Darwen H., Lorentzos N.A. Temporal Data and the Relational Model. San Francisco, Calif.: Morgan Kaufmann, 2003.

Данная глава в значительной степени основана на указанной книге, но в этой книге приведено гораздо больше подробных сведений и рассматриваются многие темы, даже не упомянутые в настоящей главе. Эти дополнительные темы, кроме всего прочего, включают следующее:

- дополнительные варианты операторов запросов и их сокращенные версии;
- операторы обновления и их сокращенные версии;
- сопоставление понятий *допустимого времени выполнения* и *времени выполнения транзакции;*
- *и* реализация и оптимизация;
- циклические позиционные типы;
- степень детализации и масштаб;
- непрерывные позиционные типы.
- При этом заслуживает особого внимания то, что модель наследования, опи санная в главе 20 настоящей книги, оказалась очень важной для решения про блемы степени детализации; она предоставляет ключ к решению проблемы од новременного применения двух разных функций определения последующей позиции (например, *следующие сутки* и *следующий месяц)* к одному и тому же позиционному типу.

23.5. Snodgrass R.T. (ed.). The Temporal Query Language TSQL2. Dordrecht, Netherlands: Kluwer Academic Pub. 1995. См. также Snodgrass R.T. et al. TSQL2 Language Specification//ACM SIGMOD Record. - March 1994. - 23, № 1.

См. аннотацию к [23.3].

ГЛАВА 24

Логические системы управления базами данных

24.1. ВВЕДЕНИЕ

Примерно в середине 1980-х годов в сообществе специалистов в области баз данных стала обнаруживаться заметная тенденция по переходу к исследованиям **систем баз данных, основанных на логике. В** научной литературе начали появляться такие выражения, как *логическая база данных; СУБД, основанная на логическом выводе; экспертная СУБД; дедуктивная СУБД; база знаний; система управления базами знаний* (СУБЗ), *логика как модель данных; рекурсивная обработка запросов* и т.д. Но не всегда легко связать такие термины и обозначаемые ими понятиями со знакомыми терминами и понятиями баз данных, а также описать причины, послужившие стимулом к развертыванию этих исследований, с точки зрения пользователей традиционных баз данных. Иными словами, существует явная необходимость объяснить всю эту деятельность в терминах понятий и принципов обычных баз данных. Настоящая глава представляет собой попытку решить указанную задачу. Автор поставил перед собой цель объяснить, что в конечном итоге представляют собой основанные на логике системы с точки зрения того, кто знаком с

технологией традиционных баз данных, но, возможно, не слишком увлекался логикой как таковой. Поэтому при описании в этой главе каждой новой идеи из области логики приведено ее объяснение в терминах обычных баз данных, если это возможно или приемлемо. (Безусловно, в этой книге уже рассматривались некоторые понятия логики, особенно при описании реляционного исчисления в главе 8. Реляционное исчисление непосредственно основано на логике. Но, как станет очевидно после изучения данной главы, в системах, основанных на логике, заложено нечто гораздо более значительное, чем просто реляционное исчисление.)

Эта глава имеет следующую структуру. За данным вступительным разделом находится раздел 24.2, содержащий краткий обзор по этой теме и небольшую историческую справку. В разделах 24.3 и 24.4 приведена элементарная (и в значительной степени упрощенная) трактовка, соответственно, *исчисления высказываний* и *исчисления предикатов.* Затем в разделе 24.5 содержится вводное описание так *называемого доказательно-теоретического представления* базы данных, а в разделе 24.6 на основе понятий, представленных в разделе 24.5, приведено описание того, что подразумевается под термином *дедуктивная СУБД.* После этого в разделе 24.7 обсуждаются некоторые подходы к решению проблемы *рекурсивной обработки запросов.* Наконец, в разделе 24.8 приведено резюме и дано несколько заключительных замечаний.

24.2. КРАТКИЙ ОБЗОР

Исследования связей между теорией баз данных и логикой начались еще в конце 1970-х годов или даже раньше (см., например, [24.3], [24.4] и [24.8]). Но, по-видимому, самым важным стимулом к пробуждению в последнее время значительного интереса к этой теме стала публикация в 1984 году чрезвычайно важной статьи Рейтера (Reiter) [24.10]. В этой статье Рейтер охарактеризовал традиционное восприятие систем баз данных как модельно-теоретическое. Под этим в его статье подразумевались приведенные ниже особенности (в неформальном изложении).

а) База данных в любой конкретный момент времени может рассматриваться как множество явно заданных (т.е. базовых) отношений, причем каждое из них содер жит множество явно заданных кортежей.

б) Выполнение запроса может рассматриваться как вычисление некоторой заданной формулы (т.е. логического выражения) с этими явно заданными отношениями и кортежами.

Примечание. Термин *модельно-теоретический* будет определен более точно в разделе 24.5.

Затем Рейтер перешел к обоснованию того мнения, что возможно альтернативное, **доказательно-теоретическое** представление базы данных, которое в действительности является более предпочтительным по некоторым причинам. Это альтернативное представление характеризуется указанными ниже особенностями (снова описанными неформально).

а) База данных в любой конкретный момент времени рассматривается как множество аксиом (состоящее из *основных* аксиом, соответствующих значениям в доменах[1]

[1] Для согласования с другими работами в этой области, в настоящей главе автор использует термин *домен,* а не более предпочтительный (с его точки зрения) термин *тип.*

и кортежам в базовых отношениях, а также некоторых *дедуктивных* аксиом, которые рассматриваются ниже).

б) Выполнение запроса рассматривается как доказательство того, что некоторая заданная формула является логическим следствием из этих аксиом (иными словами, доказательство того, что она является теоремой).

Примечание. Термин *доказательно-теоретический* также будет более точно определен ниже, в разделе 24.5, но может оказаться полезным отметить непосредственно здесь, что доказательно-теоретическое представление является очень близким к рассматриваемой в данной книге характеристике базы данных как коллекции истинных высказываний.

Рассмотрим один пример. Предположим, что в обычно применяемой нами базе данных поставщиков и деталей выполняется следующий запрос реляционного исчисления.

```
SPX WHERE SPX.QTY > 250
```

Здесь SPX — переменная области значений, принимающая свои значения в отношении поставок. В традиционной (т.е. модельно-теоретической) интерпретации каждый кортеж с данными о поставкам рассматривается последовательно, при этом формула "QTY > 250" вычисляется для каждого из этих кортежей по очереди, поэтому результат запроса состоит только из тех кортежей отношения поставок, для которых эта формула принимает значение TRUE. В отличие от этого, в доказательно-теоретической интерпретации кортежи поставок (наряду с некоторыми другими элементами) рассматриваются как аксиомы некоторой логической теории; после этого применяются определенные методы доказательства теорем для выяснения того, для каких возможных значений переменной области значений SPX формула "SPX. QTY > 250" является логическим следствием этих аксиом в теории. Таким образом, результат запроса состоит только из этих конкретных значений SPX.

Безусловно, этот пример чрезвычайно прост. Он фактически является настолько простым, что могут возникать трудности при попытке понять, в чем действительно состоит различие между этими двумя интерпретациями. Но суть этой проблемы в том, что механизм проведения рассуждений, используемый в ходе осуществления попыток доказательства теоремы (в доказательно-теоретической интерпретации) может быть гораздо сложнее, чем способен продемонстрировать этот простой пример. На самом деле, доказательно-теоретический подход, как будет показано ниже, позволяет справиться с некоторыми проблемами, которые выходят за рамки возможностей классических реляционных систем. Более того, доказательно-теоретическая интерпретация обладает описанным ниже привлекательным набором дополнительных возможностей [24.10].

- Единообразие представления. Это свойство позволяет определить язык базы дан ных, в котором все значения в доменах, кортежи в базовых отношениях, *дедуктив ные аксиомы,* запросы и ограничения целостности по существу представлены оди наковым, единообразным способом.
- Единообразие методов организации функционирования. Это свойство создает основу для решения с помощью единого подхода многих проблем, внешне кажущихся различными, включая оптимизацию запросов (особенно семантическую оптими зацию), поддержку ограничений целостности, проектирование базы данных (теория зависимостей), доказательство правильности программ и другие проблемы.

- **Семантическое моделирование.** Благодаря этому свойству создается прочная основа, на которой могут формироваться различные *семантические* дополнения к базовой модели.
- **Расширение области применения.** Наконец, формируется основа для решения определенных задач, которые обычно создавали значительные сложности при использовании классических подходов, например, представление дизъюнктивной информации (допустим, "Поставщик S5 поставляет либо деталь P1, либо деталь P2, но не известно, какую именно из них").

Дедуктивные аксиомы

В этом подразделе приведено краткое и предварительное определение понятия **дедуктивной аксиомы,** которое уже несколько раз встречалось в настоящей главе. По сути, *дедуктивная аксиома* (называемая также **правилом логического вывода)** представляет собой правило, с помощью которого при наличии определенных фактов можно осуществить *дедуктивный логический вывод* (называемый также просто *логическим выводом)* дополнительных фактов. Например, если известны факты "Анна — мать Бетти" и "Бетти — мать Селии", то существует очевидная дедуктивная аксиома, позволяющая сделать логический вывод, что Анна — бабушка Селии. Поэтому, чтобы немного опередить ход нашего описания, мы можем представить себе **дедуктивную СУБД,** в которой два указанных факта представлены в виде кортежей отношения следующим образом.

MOTHER_OF

MOTHER	DAUGHTER
Anne	Betty
Betty	Celia

Эти два факта представляют собой основные аксиомы для системы. Теперь предположим также, что каким-то образом для этой системы формально задана дедуктивная аксиома, например, как показано ниже (здесь применяется гипотетический и упрощенный синтаксис).

```
IF MOTHER OF ( X, y ) AND MOTHER OF ( y,
z ) THEN GRANDMOTHER_OF ( X, z ) END IF
```

Теперь система может применить правило, выраженное в виде дедуктивной аксиомы, к данным, представленным с помощью основных аксиом, тем способом, который будет описан в разделе 24.4, чтобы дедуктивным путем получить результат GRANDMOTHER_OF (Anne,Celia). Таким образом, пользователи могут задавать системе примерно такие вопросы: "Кто является бабушкой Селии?" или "Кто является внучкой Анны?" (или, более точно, "Чьей бабушкой является Анна?").

После этого попытаемся связать изложенные выше идеи с понятиями традиционных баз данных. В обычных терминах дедуктивная аксиома может рассматриваться как определение представления, например, приведенное ниже.

```
VAR GRANDMOTHER OF VIEW
  { MX.MOTHER AS GRANDMOTHER, MY.DAUGHTER AS
    GRANDDAUGHTER } WHERE MX.DAUGHTER = MY.MOTHER ;
```

Здесь намеренно использован стиль реляционного исчисления; мх и MY — переменные области значений, принимающие свои значения в отношении MOTHER_OF. Запросы, подобные приведенным выше, теперь могут оформляться в терминах указанного представления, следующим образом.

```
GX.GRANDMOTHER WHERE GX.GRANDDAUGHTER = NAME ('Celia')
GX . GRANDDAUGHTER WHERE GX.GRANDMOTHER = NAME ( ' Anne ' )
```

Здесь GX — переменная области значений, принимающая свои значения в отношении GRANDMOTHER_OF.

Поэтому до сих пор фактически были представлены лишь другой синтаксис и другая интерпретация для материала, который нам уже известен. Тем не менее, как будет показано в следующих разделах, между основанными на логике системами и более традиционными СУБД имеются более существенные различия, которые не удается продемонстрировать с помощью таких простых примеров.

24.3. ИСЧИСЛЕНИЕ ВЫСКАЗЫВАНИЙ

В этом и следующем разделах приведено очень краткое введение в некоторые из основных понятий логики. В настоящем разделе рассматривается исчисление высказываний, а в следующем — исчисление предикатов. Но следует сразу же отметить, что с точки зрения рассматриваемой темы исчисление высказываний в конечном итоге не имеет слишком большого значения; главная основная задача текущего раздела в действительности состоит в том, чтобы подготовить основу для понимания следующего. А в целом назначение этих двух разделов вместе взятых состоит в подготовке базиса, на котором построена остальная часть данной главы.

Предполагается, что читатель знаком с основными концепциями булевой алгебры. Но для использования в качестве ссылок повторим некоторые законы булевой алгебры, которые нам потребуются в дальнейшем.

- Распределительные законы.

```
f AND ( g OR h ) ≡ ( f AND g ) OR ( f AND h )
f OR ( g AND h ) ≡ ( f OR g ) AND ( f OR h )
```

- Законы де Моргана.

```
NOT ( f AND g ) ≡ ( NOT f ) OR ( NOT g )
NOT ( f OR g ) ≡ ( NOT f ) AND ( NOT g )
```

Здесь f, g и h — произвольные логические выражения.

Теперь обратимся к логике как таковой. *Логику* можно рассматривать как формальный метод проведения рассуждений. Поскольку метод проведения рассуждений формализован, он может использоваться для решения формальных задач, таких как проверка допустимости фактического параметра путем исследования лишь структуры этого фактического параметра в виде стандартной последовательности шагов (например, в которой не уделяется никакого внимания смыслу самих этих шагов). В частности, поскольку это — формальный метод, он может быть автоматизирован (иными словами, запрограммирован) и поэтому его выполнение можно возложить на некоторый механизм.

Исчисление высказываний и исчисление предикатов— это, вообще говоря, два особых направления логики (фактически первое является подмножеством последнего).

Термин *исчисление,* в свою очередь, представляет собой просто общий термин, который относится к любой системе символических вычислений; в данных рассматриваемых случаях вычисление представляет собой определение истинностного значения (TRUE или FALSE) некоторых формул или выражений.

Термы

Начнем этот раздел с предположения, что имеется некоторая коллекция объектов, называемых **константами,** применительно к которым мы можем высказывать утверждения различного рода. В терминах баз данных эти константы представляют собой значения из соответствующих доменов, а утверждениями могут быть, например, логические выражения, такие как "3 > 2". Определим **терм** как утверждение, которое включает такие константы[2] и соответствует приведенным ниже требованиям:

а) либо не включает никаких логических операторов (см. следующий подраздел), либо заключено в круглые скобки;

б) в результате вычисления недвусмысленно принимает значение либо TRUE, либо FALSE.

Например, все следующие утверждения являются термами: "Поставщик S1 находится в Лондоне", "Поставщик S2 находится в Лондоне" и "Поставщик S1 поставляет деталь P1" (при обычно используемых нами значениях данных они принимают значения, соответственно, TRUE, FALSE и TRUE). В отличие от этого, следующие утверждения не являются термами, поскольку они не принимают недвусмысленно значение TRUE или FALSE: "Поставщик S1 поставляет деталь p" (где p — переменная) и "Поставщик S5 будет поставлять деталь P1 когда-то в будущем".

Формулы

Затем определим понятие **формулы.** Формулы *исчисления высказываний* и, вообще говоря, *исчисления предикатов* используются в системах баз данных (кроме всего прочего) при формулировке запросов и строятся по правилам, показанным ниже.

```
<formula>
    ::=   <term>
        | NOT <term>
        I <terra> AND <formula>
        I <fcerm>  OR <formula>
        | <term> ⇒
<formula>
<term>
    ::=   <atomic formula>
        | (  <formula>  )
```

Формулы вычисляются в соответствии с истинностными значениями составляющих их термов, с применением обычных таблиц истинности для логических операторов (AND, OR и т.д.). Из этого следуют приведенные ниже выводы.

---

[2] Точнее, содержит имена таких констант или, иными словами, литералы. В основной части литературы различие между этими понятиями практически не учитывается.

1. Показанная в определении структуры формулы <*formula*> атомарная формула <*atomic formula*> — это логическое выражение, которое не содержит логиче ских операторов и не заключено в круглые скобки.
2. Символ "$\Rightarrow$" представляет собой логический оператор, известный под названием **логической импликации.** Выражение f $\Rightarrow$ g определено как логически эквива лентное выражению (NOT f) OR g.
   Примечание. Для обозначения этого оператора в главе 8 и в других предыдущих главах использовалась конструкция" IF. .. THEN... END IF".
3. В целях сокращения количества круглых скобок, необходимых для регламентации желаемого порядка вычисления, применяются обычные правила приоритета для логических операторов (NOT, затем AND, затем OR, затем $\Rightarrow$).
4. **Высказывание** представляет собой просто формулу <*formula*>, составленную в соответствии с приведенным выше определением (термин *формула* используется для того, чтобы в этом и следующем разделах применялась единообразная терми нология).

Правила вывода

Теперь перейдем к описанию правил вывода для исчисления высказываний. Количество подобных правил весьма велико. Каждое из них представляет собой утверждение в следующей форме.

$\models$ f $\Rightarrow$ g

Здесь f и g — формулы, а символ $\models$ может рассматриваться как обозначающий "всегда справедливо то, что"; обратите внимание на то, что некоторый подобный символ требуется для того, чтобы мы могли формировать метаутверждения (т.е. утверждения об утверждениях). Ниже приведены некоторые примеры правил логического вывода.

1. $\models$(f AND g) $\Rightarrow$ f
2. $\models$f $\Rightarrow$ (f OR g)
3. $\models$((f $\Rightarrow$ g) AND (g $\Rightarrow$ h)) $\Rightarrow$ (f $\Rightarrow$)
4. $\models$(f AND (f $\Rightarrow$ g)) $\Rightarrow$ g

Примечание. Последнее правило является особенно важным. Оно называется правилом modus ponens (или **правилом отделения).** Неформально это правило гласит, что если высказывание f является истинным и из f следует д, то д также должно быть истинным. Например, допустим, что приведенные ниже формулыа) и *б)* обе являются истинными.

а) У меня нет денег.

б) Если у меня денег, то я должен зарабатывать на жизнь мытьем посуды.

Из этого можно сделать вывод, что истинной является также формула в)..

в) Я должен зарабатывать на жизнь мытьем посуды.

Продолжим описание правил логического вывода.

5. $\models$ (f $\Rightarrow$ (g $\Rightarrow$ h)) $\Rightarrow$ ((f AND g) $\Rightarrow$ h)
6. $\models$ ((f OR g) AND (NOT g OR h)) $\Rightarrow$ (f OR h)

Примечание. Последнее из этих правил также относится к числу особенно важных. Оно называется правилом **резолюции.** Дополнительная информация о нем приведена в следующем подразделе, "Доказательства", а также в разделе 24.4.

Доказательства

Теперь у нас есть необходимый логический аппарат для проведения формальных доказательств (в контексте исчисления высказываний). Проблема доказательства представляет собой проблему определения того, является ли некоторая конкретная формула g **(заключение)** логическим следствием из некоторого заданного множества формул fl, f2, . . ., fn **(предпосылок[3]),** что можно представить в символическом виде, как показано ниже.

fl, f2, . . . , fn ├ g

Это символическое обозначение можно прочитать так: "Формула g может быть получена путем **дедуктивного логического вывода** (или, проще говоря, может быть *логически выведена)* из формул fl, f2, . . ., fn". Обратите внимание на использование еще одного металингвистического символа, "├". Основной метод проведения такого доказательства известен под названием **прямого логического вывода.** Прямой логический вывод заключается в том, что правила логического вывода неоднократно применяются к предпосылкам, к формулам, дедуктивно выведенным из этих предпосылок, к формулам, дедуктивно выведенным из этих формул и т.д., до тех пор, пока не будет дедуктивно выведено заключение. Иными словами, процесс логического вывода, образно выражаясь, "разворачивается в прямом направлении в виде сплошной цепочки" от предпосылок до заключения. Но есть также несколько описанных ниже вариантов этого основного подхода.

1. **Принятие предпосылки.** Если g имеет форму p $\Rightarrow$ q, принять p в качестве допол нительной предпосылки и показать, что q может быть дедуктивно выведена из за данных предпосылок и p.
2. **Обратный логический вывод.** Вместо попытки доказать, что p $\Rightarrow$ q, доказать про тивоположное утверждение, NOT q $\Rightarrow$ NOT p.
3. **Приведение к абсурду.** Вместо того, чтобы доказывать непосредственно, что p $\Rightarrow$ q, принять предположение, что p и NOT q одновременно являются истинными, и вы вести из этого противоречие.
4. **Резолюция.** В этом методе используется правило логического вывода на основе ре золюции (правило 6 из приведенного выше списка).

Рассмотрим метод резолюции более подробно, поскольку он имеет широкое применение (в частности, он обобщается также на случай исчисления предикатов, как будет показано в разделе 24.4).

Прежде всего, следует отметить, что правило резолюции по существу представляют собой такое правило, которое позволяет исключать подформулы. Например, если даны следующие две формулы: f OR g и NOT g OR h, то можно исключить g и NOT g, чтобы вывести приведенную ниже упрощенную формулу:

---

[3] В формальной логике вместо термина *предпосылка* применяется также термин *посылка,* или *гипотеза.*

f OR h

В частности, если дано, что f OR g и NOT g (т.е. можно принять, что h имеет значение TRUE), то можно вывести f.

Поэтому следует отметить, что правило резолюции, вообще говоря, применяется к **конъюнкции** (AND) двух формул, каждая из которых представляет собой **дизъюнкцию** (OR) двух формул. Таким образом, для того чтобы использовать правило резолюции, нужно действовать следующим образом. (Чтобы это описание было немного более конкретным, рассмотрим данный процесс на примере.) Предположим, что требуется определить, действительно ли следующее предполагаемое доказательство является допустимым:

A $\Rightarrow$ (B $\Rightarrow$ C), NOT D OR A, B $\vdash$ D $\Rightarrow$ C

Здесь A, в, с и D — формулы. Начнем с того, что примем отрицание заключения в качестве дополнительной предпосылки, а затем запишем каждую предпосылку на отдельной строке следующим образом:

A $\Rightarrow$ (B $\Rightarrow$ C)
NOT D OR A
B
NOT (D $\Rightarrow$ C)

Эти четыре строки неявно соединены друг с другом операторами "AND".

Теперь преобразуем каждую отдельную строку в **конъюнктивную нормальную форму,** т.е. в форму, состоящую из одной или нескольких формул, соединенных друг с другом операторами AND, притом что каждая отдельная формула содержит (возможно) операторы NOT и OR, но не операторы AND (см. главу 18). Безусловно, вторая и третья строки уже находятся в этой форме. Для того чтобы преобразовать в нее остальные две строки, вначале удалим все вхождения оператора "$\Rightarrow$" (используя определение этого оператора в терминах NOT и OR), затем по мере необходимости применим распределительные законы и законы де Моргана (см. начало этого раздела). Кроме того, можно удалить лишние круглые скобки и пары смежных операторов NOT (которые отменяют друг друга). Четыре приведенные выше строки принимают следующий вид.

```
NOT AOR NOT B OR C
NOT DOR A
B
D ANDNOT C
```

После этого каждая строка, которая включает какие-либо явно заданные операторы AND, заменяется множеством отдельных строк, по одной для каждой из отдельных формул, соединенных оператором AND (в ходе этого процесса операторы AND уничтожаются). В рассматриваемом примере данный шаг применяется только к четвертой строке. Итак, теперь предпосылки выглядят следующим образом.

```
NOT A OR NOT B OR C
NOT D OR A
B
D NOT C
```

После этого можно приступить к применению правила резолюции. Для этого выбирается пара строк, которые могут стать объектом действия этого правила, т.е. пара, содержащая, соответственно, некоторую конкретную формулу и отрицание этой формулы.

Для этого возьмем первые две строки, содержащие, соответственно, NOT A и A, и применим к ним правило резолюции, что приводит к получению следующего результата.

```
NOT  D  OR  NOT  B  OR
C   B D  NOT  C
```

Примечание. В общем случае может также потребоваться сохранить две первоначальные строки, но в данном конкретном примере они нам больше не понадобятся.

Теперь снова применим это правило и для этого снова выберем первые две строки (при этом происходит резолюция формул NOT в и в); полученный результат показан ниже.

```
NOT  D  OR  C
D
NOT  C
```

Снова выберем первые две строки (NOT D и D) и получим следующий результат.

```
c
NOT  C
```

Опять же, применим это правило (c и NOT c); окончательный результат представляет собой пустое множество высказываний (которое обычно отображается следующим образом: [ ]); такой результат в соответствии с общепринятым соглашением рассматривается как противоречие. Таким образом, желаемый результат доказан путем *приведения к абсурду*.

24.4. ИСЧИСЛЕНИЕ ПРЕДИКАТОВ

Теперь перейдем к описанию исчисления предикатов. Основное различие между исчислением высказываний и исчислением предикатов заключается в том, что последнее позволяет включать в формулы переменные[4] и кванторы, благодаря чему эти формулы становятся намного более мощными и находят гораздо более широкую область применения. Например, следующие утверждения не являются допустимыми формулами исчисления высказываний, но они допустимы в исчислении предикатов: "Поставщик S1 поставляет деталь p" и "Некоторый поставщик s поставляет деталь p". Поэтому исчисление предикатов предоставляет основу для составления примерно таких запросов: "Какие детали поставляет поставщик S1?", или "Определить поставщиков, которые поставляют некоторую деталь", или даже "Определить поставщиков, которые не поставляют вообще никаких деталей".

Предикаты

Как было описано в главе 3, **предикат** представляет собой логическую функцию, т.е. функцию, которая после постановки соответствующих фактических параметров вместо формальных параметров возвращает либо TRUE, либо FALSE. Например, функция "> (x, y) " (которая чаще всего записывается как "x > y") представляет собой предикат с двумя формальными параметрами, x и y; она возвращает TRUE, если ее фактический

[4] Точнее, имена переменных. Рассматриваемые переменные являются логическими переменными, а не переменными языка программирования. Их можно рассматривать как переменные области значений в том смысле, который указан в главе 8.

параметр, соответствующий x, больше фактического параметра, соответствующего y; в противном случае эта функция возвращает FALSE. Предикат, который принимает n фактических параметров (т.е. равным образом, который определен в терминах n формальных параметров) называется n-местным или n-арным предикатом. Высказывание (т.е. формула в том смысле, который определен в разделе 24.3) может рассматриваться как нульместный, или нуль-арный предикат — она не имеет формальных параметров и недвусмысленно принимает значение TRUE или FALSE.

Примем удобное предположение, что предикаты, соответствующие операторам "=", ">", ">" и т.д., являются встроенными (т.е. что они входят в состав описываемой нами формальной системы) и что выражения, в которых они используются, разрешено записывать в общепринятой форме. Но, безусловно, пользователям должна быть также предоставлена возможность определять свои собственные предикаты. И действительно, весь смысл рассматриваемого нами подхода состоит в следующем: дело в том, что в терминах баз данных определяемый пользователем предикат соответствует определяемой пользователем переменной отношения (как уже было указано в предыдущих главах). Например, переменная отношения поставщиков S может рассматриваться как предикат с четырьмя формальными параметрами, S#, SNAME, STATUS и CITY. Более того, выражения S(S1, Smith, 20, London) и S (S6, White, 45, Rome), сформулированные с использованием наглядного сокращенного обозначения, представляют собой *экземпляры, варианты конкретизации* или *вызовы* этого предиката, которые (при использовании обычно принятого в качестве примера множества значений) принимают значения, соответственно, TRUE и FALSE. Неформально такие предикаты (наряду со всеми применимыми ограничениями целостности, которые также являются предикатами) могут рассматриваться как определение того, что "означает" эта база данных, как было описано в предыдущих частях данной книги (особенно в главе 9).

Правильно построенные формулы

Следующий этап состоит в том, что можно расширить определение понятия *формулы.* Для того чтобы избежать путаницы с формулами, рассматриваемыми в предыдущем разделе (которые фактически представляют собой частный случай), перейдем к использованию термина *правильно построенная формула* (Well-Formed Formula — WFF, произносится как "вэфф"), или сокращенно *ППФ,* который был определен в главе 8. Ниже приведен упрощенный синтаксис правильно построенных формул.

```
<wff>
    ::-  <term>
       |  NOT ( <wff> )
       |  ( <wff> ) AND ( <wff> )
       j  ( <wff> ) OR ( <wff> )
       |  ( <wff> ) ⇒ ( <wff> )
       |  EXISTS <var name> ( <wff> )
       I  FORALL <var name> (
<wff> )
<term>
    ::=  [ NOT ] <pred name> [ ( <argument commalist> ) ]
```

Из этого определения следуют приведенные ниже выводы.

1. В качестве терма <*term*> применяется просто *экземпляр предиката* или, возможно, его отрицание (если предикат рассматривается как логическая функция, то экземпляр предиката представляет собой вызов этой функции). Каждый фактический параметр <*argument*> должен представлять собой константу, имя переменной или вызов функции, где каждый фактический параметр в вызове функции, в свою очередь, является константой, именем переменной или вызовом функции. Разделенный запятыми список фактических параметров <*argument commalist*> и (при желании) соответствующие круглые скобки в случае нуль-местного предиката исключаются.

   Примечание. Разрешено использование функций (под этим подразумеваются функции, отличные от логических функций, представляющих собой предикаты) для того, чтобы иметь возможность включать в правильно построенные формулы вычислительные выражения, такие как "+ (x, y) " (которые обычно принято записывать как "x + y") и т.д.

2. Как описано в разделе 24.3, применяются обычные правила приоритета для логических операторов (NOT, затем AND, затем OR, затем $\Rightarrow$) для того, чтобы можно было сократить количество круглых скобок, необходимых для регламентации желаемого порядка вычисления.

3. Предполагается, что читатель знаком с кванторами EXISTS и FORALL.

   Примечание. Здесь нас интересует только *исчисление предикатов первого порядка,* а это по существу означает, во-первых, что нет *предикативных переменных* (т.е. переменных, допустимыми значениями которых являются предикаты), и поэтому, во-вторых, сами предикаты не могут становиться объектом применения кванторов. См. упражнение 8.8 из главы 8.

4. Законы де Моргана могут быть обобщены применительно к правильно построенным формулам, содержащим кванторы, следующим образом:

```
NOT ( FORALL X ( f ) ) ≡ EXISTS X ( NOT ( f ) )
NOT ( EXISTS X ( f ) ) ≡ FORALL X ( NOT ( f ) )
```

   Эта тема также обсуждалась в главе 8.

5. Повторим еще один фрагмент из главы 8: в любой конкретной правильно построенной формуле каждая ссылка на переменную является либо свободной, либо связанной. Ссылка является **связанной** тогда и только тогда, когда она, во-первых, находится непосредственно за ключевым словом EXISTS ИЛИ FORALL (т.е. обозначает переменную, на которую распространяется действие квантора) или, во-вторых, находится в области действия квантора и обозначает соответствующую квантифицированную переменную. Ссылка на переменную является **свободной** тогда и только тогда, когда она не связана.

6. **Закрытой правильно построенной формулой** называется такая формула, которая не содержит ссылок на свободные переменные (фактически она является высказыванием). Открытой **правильно построенной** формулой называется такая формула, которая не является замкнутой.

Интерпретации и модели

Что означает правильно построенная формула? Для того чтобы составить формальный ответ на этот вопрос, введем понятие интерпретации. Интерпретация множества правильно построенных формул определена, как описано ниже.

- Во-первых, определим предметную область, в которой должны интерпретироваться эти правильно построенные формулы. Иными словами, зададим отображение между допустимыми константами формальной системы (или значениями домена в терминах базы данных), с одной стороны, и объектами "реального мира", с другой. Каждая отдельная константа соответствует одному и только одному объекту в предметной области.
- Во вторых, определим смысл каждого предиката в терминах объектов предметной области.
- В-третьих, определим также смысл каждой функции в терминах объектов предметной области.

В таком случае *интерпретация* представляет собой сочетание, в которое входят предметная область, отображение отдельных констант на объекты в этой области, а также определяемый смысл для предикатов и функций по отношению к этой области.

В качестве примера предположим, что предметная область представляет собой множество целых чисел $\{0,1,2,3,4,5\}$; допустим, что такие константы, как "2", соответствуют объектам этой области очевидным образом, а также примем допущение, что предикат "x > y" определен как имеющий обычный смысл. (При желании мы можем также определить такие функции, как "+", "-" и т.д.) Теперь появляется возможность присваивать соответствующие истинностные значения правильно построенным формулам, как показано ниже.

```
2 > 1                  :  TRUE
2 > 3                : FALSE
EXISTS x ( x> 2 ) : TRUE
FORALL X ( x> 2 ) : FALSE
```

Но следует отметить, что возможны и другие интерпретации. Например, можно определить предметную область как множество степеней классификации секретности следующим образом.

```
уничтожить перед прочтением  (уровень 5)
уничтожить после прочтения   (уровень 4)
совершенно секретно          (уровень 3)
секретно                     (уровень 2)
для служебного пользования   (уровень 1)
несекретно                   (уровень 0)
```

Теперь предикат ">" может означать "более секретный (т.е. имеющий более высокую степень секретности), чем".

После этого читатель, по-видимому, обнаружил, что приведенные выше две возможные интерпретации являются *изоморфными.* Это означает, что между ними можно установить взаимно-однозначное соответствие и поэтому на более глубоком уровне эти две интерпретации фактически представляют собой одно и то же. Но следует четко понимать, что могут существовать интерпретации, которые изначально являются различными

по своему характеру. Например, можно снова взять в качестве предметной области целые числа от 0 до 5, но определить предикат ">" как обозначающий равенство. (Безусловно, это может стать причиной значительной путаницы, но, по крайней мере, мы имеем полное право принять подобное решение.) В таком случае первая правильно построенная формула из списка ("2 > 1") будет иметь значение FALSE,,а не TRUE.

Еще одна идея, которую следует четко сформулировать, состоит в том, что две интерпретации могут быть изначально различными в указанном выше смысле и, тем не менее, приводить к получению одинаковых истинностных значений для заданного множества правильно построенных формул. Таковым был бы случай с двумя разными определениями оператора ">" в рассматриваемом примере, если бы была исключена правильно построенная формула "2 > 1".

Кстати, следует отметить, что все правильно построенные формулы, которые рассматривались до сих пор в этом подразделе, были замкнутыми ППФ. Причина этого состоит в том, что при наличии некоторой интерпретации всегда возможно присвоить замкнутой ППФ конкретное истинностное значение, но истинностное значение открытой ППФ зависит от значений, присвоенных свободным переменным. Например, вполне очевидно, что приведенная ниже открытая ППФ принимает значение TRUE, если x больше 3, и FALSE в противном случае (при любой трактовке понятий "больше чем" и "3" в данной интерпретации):

```
x > 3
```

Теперь можно определить модель множества (обязательно замкнутых) правильно построенных формул как интерпретацию, при которой все ППФ в данном множестве принимают значение TRUE. Вторая из двух интерпретаций, которые были даны для четырех приведенных ниже ППФ в терминах целых чисел от 0 до 5, не была моделью для этих ППФ, поскольку некоторые из ППФ при этой интерпретации принимали значение FALSE:

```
2 > 1
2 > 3
EXISTS x ( x > 2 )
FORALL x ( x > 2 )
```

В отличие от этого, первая интерпретация (в которой операция ">" была определена "должным образом"), должна была быть моделью для следующего множества ППФ:

```
2 > 1
3 > 2
EXISTS x ( x > 2 )
FORALL x { x > 2 OR NOT ( x > 2 ) )
```

Наконец, следует отметить, что каждое конкретное множество ППФ может принимать несколько интерпретаций, в которых все ППФ становятся истинными, и поэтому (вообще говоря) может иметь несколько моделей. Это означает, что и база данных (вообще говоря) может иметь несколько моделей, поскольку база данных в модельно-теоретическом представлении является просто множеством правильно построенных формул (см. раздел 24.5).

Клаузальная форма

Так же, как любая формула исчисления высказываний может быть преобразована в конъюнктивную нормальную форму, так и любая правильно построенная формула исчисления предикатов может быть преобразована в **клаузальную форму,** которая может рассматриваться как расширенная версия конъюнктивной нормальной формы. Один из стимулов к выполнению такого преобразования (снова) состоит в том, что оно позволяет применять правило резолюции при формировании или проверке доказательств, как будет вскоре показано.

Процесс преобразования осуществляется следующим образом (здесь он описан схематически; дополнительные сведения приведены в [24.6]). Этапы преобразования будут показаны на примере применения их к следующей ППФ: ,

```
FORALL X ( p ( X ) AND EXISTS y ( FORALL z ( q ( y, z ) ) ) )
```

Здесь p и g — предикаты, а x, y и z — переменные.

1. Удалить символы "$\Rightarrow$", как показано в разделе 24.3. В данном примере этот пер вый этап преобразования не требуется.
2. Воспользоваться законами де Моргана, а также тем фактом, что два смежных опе ратора NOT взаимно исключаются, для перемещения операторов NOT так, чтобы они применились только к термам, а не к общим правильно построенным форму лам. (В настоящем примере и этот конкретный этап преобразования не требуется.)
3. Преобразовать правильно построенную формулу в *предваренную* (или *пренексную)* нормальную форму, перемещая все кванторы вперед (и в случае необходимости систематически переименовывая переменные), как показано ниже.

   ```
   FORALL x ( EXISTS y ( FORALL z ( p ( x ) AND q ( y, z ) ) ) )
   ```

4. Здесь заслуживает внимания то, что правильно построенная формула с квантором существования, подобная приведенной ниже

   EXISTS v (r (v))

   эквивалентна следующей правильно построенной формуле для некоторой неизвестной константы a:

   r (a)

   Таким образом, первоначальная правильно построенная формула подтверждает, что некоторая подобная константа a действительно существует, даже если ее значение не известно. Аналогичным образом, правильно построенная формула, наподобие

   ```
   FORALL u ( EXISTS v ( s ( u, v ) ) )
   ```

   эквивалентна приведенной ниже правильно построенной формуле для некоторой неизвестной функции f от универсально квантифицированной переменной и.

   FORALL и (s (и, f (и)))

   Константа a и функция f в этих примерах известны, соответственно, под названием **скулемовской константы** и **скулемовской функции,** которые названы в честь логика T.A.Скулема (T.A.Skolem). *{Примечание.* Скулемовская константа в действительности представляет собой просто скулемовскую функцию без параметров.) Поэтому следующий этап состоит в устранении кванторов существования путем

замены их соответствующими квалифицированными переменными с помощью (произвольных) скулемовских функций от всех универсально квантифицированных переменных, которые предшествуют рассматриваемому квантору в этой правильно построенной формуле, следующим образом.

```
FORALL x ( FORALL z ( p ( X ) AND q ( f ( x ), 2 ) ) )
```

5. Теперь все переменные квантифицированы универсально, поэтому может быть принято соглашение, в соответствии с которым все переменные неявно считаются квантифицированными универсально, позволяющее удалить явно заданные кванторы следующим образом.

   ```
   p ( x ) AND q ( f ( X ), z )
   ```

6. Преобразовать эту правильно построенную формулу в конъюнктивную нормальную форму, т.е. во множество выражений (называемых также *клаузами* или *дизъюнктами),* соединенных операторами AND, в котором каждое выражение может включать операторы NOT и OR, но не AND. В данном примере правильно построенная формула уже находится в такой форме.

7. Записать каждое выражение на отдельной строке **и** удалить операторы AND, следующим образом.

   ```
   P ( X ) q ( f ( X ), Z )
   ```

   Это и есть клаузальная форма, эквивалентная первоначальной правильно построенной формуле.

Примечание. Из описанной выше процедуры следует, что по своему общему виду любая правильно построенной формула в клаузальной форме представляет собой множество выражений, каждое из которых находится на отдельной строке и каждая имеет следующую форму.

```
NOT A1 OR NOT A2 OR ... OR NOT Am OR B1 OR B2 OR ... OR Bn
```

Здесь все термы A и B не содержат операторов отрицания. При желании, такое выражение можно записать следующим образом.

```
A1 AND A2 AND . . . AND Am => B1 OR B2 OR ... OR Bn
```

Если количество термов в не превышает одного (n = 0 или 1), такое выражение называют **хорновским выражением** в честь логика Альфреда Хорна (Alfred Horn).

Использование правила резолюции

Теперь можно перейти к изучению того, как логическая система баз данных может обрабатывать запросы. Для этого воспользуемся примером, приведенным в конце раздела 24.2. Прежде всего, задан предикат MOTHER_OF, который включает два формальных параметра, представляющих, соответственно, мать и дочь, и даны приведенные ниже два терма (экземпляры предиката).

1. `MOTHER_OF ( Anne, Betty )`
2. `MOTHER_OF ( Betty, Celia )`

   Кроме того, дана приведенная ниже правильно построенная формула *(дедуктивная аксиома).*

3. `MOTHER_OF (X,y) AND MOTHER_OF (y,z) ⇒ GRANDMOTHER_OF (X,z)`

   Обратите внимание на то, что это — хорновское выражение). Для того чтобы упростить применение правила резолюции, перепишем это выражение для удаления символа "⇒", как показано ниже.

4. `NOT MOTHER_OF ( X, y ) OR NOT MOTHER_OF (y, z ) OR GRANDMOTHER_OF ( x, z )`

   Теперь мы можем перейти к доказательству того, что Анна — бабушка Селии. Таким образом, будет показано, как ответить на запрос: "Является ли Анна бабушкой Селии?" Начнем с отрицания того заключения, которое должно быть доказано, и принятия его в качестве дополнительной предпосылки, как показано ниже.

5. `NOT GRANDMOTHER_OF ( Anne, Celia )`

   Теперь, чтобы применить правило резолюции, необходимо систематически подставлять вместо переменных конкретные значения таким образом, чтобы можно было найти, соответственно, правильно построенную формулу и ее отрицание. Такая подстановка является допустимой, поскольку все переменные неявно квантифицированы универсально и поэтому отдельные (неотрицаемые) правильно построенные формулы должны быть истинными для всех и каждой допустимой комбинации значений их переменных.

   Примечание. Осуществляемый таким образом процесс поиска множества подстановок, позволяющих сделать два выражения разрешимыми (применить к ним правило резолюции), называется унификацией.

   Для того чтобы рассмотреть, как описанный выше процесс действует в данном конкретном случае, вначале следует отметить, что строки 4 и 5 содержат, соответственно, термы GRANDMOTHER_OF (x,z) И NOT GRANDMOTHER_OF (Anne,Celia). Поэтому выполним подстановку в строке 4 значения "Anne" вместо x и "Celia" вместо z и применим правило резолюции для получения приведенного ниже выражения.

6. `NOT MOTHER_OF ( Anne, y ) OR NOT MOTHER_OF ( y, Celia )`

   Строка 2 содержит терм MOTHER_OF (Betty, Celia), поэтому можно выполнить подстановку значения "Betty" вместо y в строке 6 и применить правило резолюции для получения приведенного ниже выражения.

7. `NOT MOTHER_OF ( Anne, Betty )`

   Применив правило резолюции к строкам 7 и 1, получаем пустое множество выражений, [ ]. Это означает, что обнаружено противоречие. Поэтому ответ на первоначальный запрос состоит в следующем: "Да, Анна — бабушка Селии".

А теперь найдем ответ на запрос: "Кто является внучкой Анны?" Вначале отметим, что системе ничего не известно о внучках; она имеет информацию только о бабушках. Поэтому, чтобы выйти из положения, можно ввести еще одну дедуктивную аксиому, свидетельствующую о том, что z является внучкой x тогда и только тогда, когда x — бабушка z (в этой базе данных нет информации о мужчинах). Еще один вариант состоит в том, что первоначальный вопрос может быть перефразирован следующим образом: "Чьей бабушкой является Анна?" Рассмотрим последнюю формулировку. Применяемые предпосылки (еще раз) показаны ниже.

1. `MOTHER_OF ( Anne, Betty ) ,`
2. `MOTHER_OF ( Betty, Celia )`
3. `NOT MOTHER_OF ( X, y ) OR NOT MOTHER_OF ( y, z ) OR GRANDMOTHER_OF ( x, z )`

Введем четвертую предпосылку, как показано ниже.

4. `NOT GRANDMOTHER_OF ( Anne, r ) OR RESULT ( r )`

На интуитивном уровне ясно, что в этой новой предпосылке содержится утверждение, что либо Анна не является бабушкой кого-либо, либо существует некоторое лицо r, данные о котором входят в состав желаемого результата (поскольку Анна является бабушкой этого лица r). Нам требуется определить имена всех таких лиц r. Дальнейшие действия выполняются следующим образом. Вначале выполним подстановку значения "Anne" вместо x и переменной r вместо z и применим правило резолюции к строкам 4 и 3 для получения приведенного ниже выражения.

5. `NOT MOTHER_OF ( Anne, y ) OR NOT MOTHER_OF (y, z ) OR RESULT ( z )`

Затем выполним подстановку значения "Betty" вместо y и применим правило резолюции к строкам 5 и 1 для получения выражения, которое показано ниже.

6. `NOT MOTHER_OF ( Betty, z ) OR RESULT ( z )`

Теперь выполним подстановку значения "Celia" вместо z и применим правило резолюции к строкам 6 и 2, получив показанное ниже выражение.;

7. RESULT (Celia)

Итак, Анна — бабушка Селии.

Примечание. Если бы был введен дополнительный терм таким образом, что

`MOTHER_OF ( Betty, Delia )`

то на последнем этапе (вместо значения "Celia") можно было бы подставить в качестве z значение "Delia" и получить следующий результат.

`RESULT ( Delia )`

В таком случае у бабушки были бы две внучки и пользователь, безусловно, рассчитывал бы на получение в результате обоих имен. Итак, система должна обладать способностью применять процесс унификации и резолюции исчерпывающим образом, для выработки результата, состоящего из всех возможных значений. Подробные сведения о таком усовершенствовании выходят за рамки настоящей главы.

24.5. ДОКАЗАТЕЛЬНО-ТЕОРЕТИЧЕСКОЕ ПРЕДСТАВЛЕНИЕ БАЗ ДАННЫХ

Как описано в разделе 24.4, выражение представляет собой терм в следующей форме.

`Al AND A2 AND . . . AND Am ⇒ Bl OR B2 OR ... OR Bn`

Здесь все компоненты A и в являются термами в такой форме.

r (xl, x2, . . ., xt)

Здесь r— предикат, а xl, x2, ..., xt — фактические параметры этого предиката.)

В соответствии с работой [24.7], могут рассматриваться два описанных ниже важных частных случая этой общей конструкции.

1. m = 0, n = 1

   В данном случае рассматриваемое выражение может быть упрощенно представлено следующим образом.

   $\Rightarrow$ B1

   Иным образом, после удаления символа импликации это выражение принимает следующий вид. r (xl, x2, ..., xt) Здесь r — некоторый предикат, имеющий определенное множество фактических параметров xl, x2, . . ., xt. Если все параметры x являются константами, то данное выражение представляет собой **основную аксиому,** т.е. утверждение, которое определенно является истинным. В терминах баз данных такое утверждение соответствует кортежу[5] некоторой переменной отношения R. Предикат r соответствует *смысловому значению* переменной отношения R, как описано в главе 6и в других главах данной книги. Например, в базе данных поставщиков и деталей имеется переменная отношения SP, смысловым значением которой является то, что указанный поставщик (s#) поставляет указанную деталь (p#) в указанном количестве (QTY). Обратите внимание на то, что это смысловое значение соответствует **открытой правильно построенной формуле,** поскольку оно включает ссылки на свободные переменные (s#, P# и QTY). В отличие от этого, кортеж (SI, P1, 300), в котором все фактические параметры являются константами, представляет собой основную аксиому, или **замкнутую правильно построенную формулу,** которая недвусмысленно показывает, что поставщик S1 поставляет деталь P1 в количестве 300.

   m > 0, п = 1

   В этом случае выражение принимает следующую форму.

   A1 AND A2 AND ... AND Am $\Rightarrow$ B

   Эта форма может рассматриваться как **дедуктивная аксиома;** она дает (возможно неполное) определение предиката справа от символа импликации в терминах тех предикатов, которые находятся слева от этого символа (в качестве примера можно привести определение предиката GRANDMOTHERJDF ИЗ предыдущего раздела).

   Еще один вариант состоит в том, что такое выражение может рассматриваться как определение **ограничения целостности** (а используя терминологию главы 9, его можно рассматривать по своему назначению как ограничение переменной отношения). Для примера предположим, что переменная отношения поставщиков S имеет только два атрибута, s# и CITY. В таком случае приведенное ниже выражение представляет ограничение, согласно которому атрибут CITY является функционально зависимым от S#. S (s, cl) AND S (s, c2) $\Rightarrow$ cl = c2 Обратите внимание на то, что в данном примере используется встроенный предикат "=".

---

[5] Или значению в некотором домене.

Как показывает приведенное выше описание, кортежи в отношениях *{основные аксиомы),* производные отношения *(дедуктивные аксиомы)* и ограничения целостности, вместе взятые, могут рассматриваться как частные случаи общей конструкции — выражения. Теперь попытаемся определить, как все эти идеи могут привести к созданию *доказательно-теоретического* представления базы данных, о котором упоминалось в разделе 24.2.

Прежде всего, отметим, что традиционное представление базы данных может рассматриваться как **модельно-теоретическое.** Под "традиционным представлением" здесь просто подразумевается такое представление, согласно которому база данных рассматривается как состоящая из коллекции явно именованных переменных отношения, каждая из которых состоит из множества явно заданных кортежей, наряду с явно заданным множеством ограничений целостности. И такое представление может быть названо модельно-теоретическим, как описано ниже.

- Основополагающие домены содержат значения, или константы, которые согласно принятому предположению обозначают некоторые объекты "реального мира" (точнее, обозначают их в некоторой **интерпретации,** в том смысле этого понятия, который определен в разделе 24.4). Таким образом, они соответствуют предметной области.
- Переменные отношения (точнее, заголовки переменных отношения) представля ют собой множество предикатов, или открытых правильно построенных формул, которые должны интерпретироваться в этой предметной области. Например, заго ловок переменной отношения SP определяет предикат "Поставщик s# поставляет деталь P# в количестве QTY".
- Каждый кортеж в заданной переменной отношения представляет собой конкрети зацию соответствующего предиката; это означает, что он представляет собой вы сказывание (замкнутую правильно построенную формулу, поскольку не содержит переменных), которое, безусловно, является истинным в данной предметной об ласти.
- Ограничения целостности являются открытыми правильно построенными форму лами, интерпретируемыми в той же предметной области. Поскольку данные в базе не нарушают (т.е. не должны нарушать!) эти ограничения, то соответствующие ог раничения обязательно принимают значение TRUE при подстановке текущих зна чений из базы данных вместо их формальных параметров.
- Кортежи и ограничения целостности могут совместно рассматриваться как мно жество аксиом, определяющих некоторую **логическую теорию** (неформально вы ражаясь, *теория* в логике представляет собой множество аксиом). А поскольку все эти аксиомы в данной интерпретации являются истинными, то по определению сама интерпретация является **моделью** указанной логической теории в том смысле, который указан в разделе 24.4. Обратите внимание на то, что, как упоминалось в том же разделе, модель может не быть уникальной; это означает, что любая кон кретная база данных может иметь несколько возможных интерпретаций, притом что все они с точки зрения логики будут в равной степени действительными.

Поэтому в модельно-теоретическом представлении *смысловым значением* базы данных является модель (при использовании описанного выше толкования термина *модель).* А поскольку может быть много допустимых моделей, то может быть также много допустимых

смысловых значений, по крайней мере, в принципе[6]. Более того, обработка запроса в модельно-теоретическом представлении по существу представляет собой процесс вычисления некоторой открытой правильно построенной формулы для определения того, подстановка каких значений вместо свободных переменных в этой **ППФ** повлечет за собой то, что ППФ примет значение TRUE в рамках этой модели.

На этом описание модельно-теоретического представления заканчивается. Но для того чтобы иметь возможность применять правила логического вывода, описанные в разделах 24.3 и 24.4, необходимо встать на иную позицию, которая заключается в том, что база данных явно рассматривается как некоторая логическая теория, т.е. как множество аксиом. В таком случае *смысловым значением* базы данных становится именно коллекция, состоящая из всех истинных утверждений, которые могут быть дедуктивно выведены из этих аксиом. Таким образом, смысловым значением базы данных становится множество **теорем,** которые могут быть доказаны на основании указанных аксиом. В этом и состоит **доказательно-теоретическое представление.** В данном представлении вычисление запроса преобразуется в процесс доказательства теоремы (во всяком случае, он является таковым с концептуальной точки зрения, а в целях повышения эффективности в системе, по всей вероятности, будут использоваться более общепринятые методы обработки запроса, как показано в разделе 24.7).

Примечание. Из сказанного в приведенном выше абзаце следует, что одно из различий между модельно-теоретическим и доказательно-теоретическим представлениями (на интуитивном уровне восприятия) состоит в том, что в модельно-теоретическом представлении база данных может иметь много *смысловых значений,* а в доказательно-теоретическом представлении она обычно имеет одно и только одно *смысловое значение.* Исключениями из этого правила являются такие ситуации: во-первых, как было указано выше, в модельно-теоретическом варианте каноническое смысловое значение действительно является единственным значением, и, во-вторых, в любом варианте такое утверждение, что в доказательно-теоретическом представлении имеется только одно смысловое значение, вообще говоря, становится неправильным, если база данных включает какие-либо аксиомы с отрицаниями [24.5], [24.6].

В неформальном изложении общие сведения об аксиомах для данной конкретной базы данных (доказательно-теоретическое представление) приведены ниже [24.10].

1. *Основные аксиомы,* соответствующие значениям в доменах и кортежам в базовых переменных отношения. Эти аксиомы составляют то, что иногда называют **экстен сиональной базой данных** (extensional database), в отличие от *интенсиональной базы данных* (intensional database), которая описана в следующем разделе.
2. *Аксиомы дополнения* для каждой переменной отношения, согласно которым отсут ствие в рассматриваемой переменной отношения кортежа, допустимого по всем прочим признакам, может интерпретироваться как свидетельство того, что выска зывание, соответствующее этому кортежу, является ложным. (В действительности,

---

[6] Но поскольку принято предположение, что база данных явно не содержит какой-либо отрицательной информации (например, высказывания в форме "NOT S# (S9) ", которое означает, что S9 — это не номер поставщика), то должно также существовать "минимальное" или каноническое смысловое значение, представляющее собой пересечение всех возможных моделей [24.6]. Более того, в данном случае такое каноническое смысловое значение будет таким же, как и смысловое значение, предписанное базе данных в соответствии с доказательно-теоретическим представлением, о чем речь пойдет чуть позже.

такие аксиомы дополнения, вместе взятые, входят в состав предположения о замкнутости **мира,** которое уже рассматривалось в главах 6 и 9.) Например, тот факт, что переменная отношения поставщиков S не включает кортеж (S6, White, 45, Rome), означает, что является ложным такое высказывание: "Существует поставщик S6, работающий по контракту, имеющий имя white, статус 45 и находящийся в Риме".

3. Аксиома *уникальности имен,* которая означает, что каждая константа отличима от всех других (т.е. имеет уникальное имя).
4. Аксиома *замыкания домена,* которая означает, что не существует других констант, отличных от тех, которые находятся в доменах базы данных.
5. Множество аксиом (по существу, стандартных) для определения встроенного предиката равенства. Эти аксиомы необходимы, поскольку все такие аксиомы, как аксиомы дополнения, уникальности и замыкания домена, используют предикат равенства.

В завершение этого раздела приведено краткое итоговое описание основных различий между этими двумя представлениями (модельно-теоретическим и доказательно-теоретическим). Прежде всего, следует отметить, что с практической точки зрения различия между ними могут оказаться не столь уже значительными! (По крайней мере, если речь идет о современных СУБД.) Тем не менее, эти различия указаны ниже.

- В результате применения аксиом для доказательно-теоретического представления (не считая основных аксиом) становятся явными некоторые допущения, которые при использовании интерпретации, основанной на модельно-теоретическом представлении, были лишь неявными [24.10]. Обычно идет на пользу явная формулировка принятых допущений; более того, такие дополнительные аксиомы необходимо задавать явно для того, чтобы иметь возможность применять общие методы доказательства, такие как метод резолюции, описанный в разделах 24.3 и 24.4.
- Заслуживает внимание то, что в списке аксиом нет упоминания об ограничениях целостности. Причина такого упущения состоит в том, что (в доказательно-теоретическом представлении) после введения таких ограничений система преобразуется в *дедуктивную* СУБД (см. раздел 24.6).
- Доказательно-теоретическое представление обладает определенным изяществом (которое отсутствует в модельно-теоретическом представлении), поскольку оно обеспечивает единообразное восприятие многих конструкций, которые обычно рассматриваются как более или менее различные: базовые данные, запросы, ограничения целостности (несмотря на сказанное в предыдущем пункте), виртуальные данные и т.д. Вследствие этого появляется возможность создания более единообразных интерфейсов и более единообразных реализаций.
- Кроме того, доказательно-теоретическое представление создает естественную основу для решения некоторых проблем, при столкновении с которыми в традиционных реляционных системах всегда возникали сложности. В частности, к ним относится обработка **дизъюнктивной информации** (например,"Поставщик S6 находится либо в Париже, либо в Риме"), извлечение из базы **отрицательной информации** (например, "Кто не является поставщиком?") и обработка рекурсивных запросов (см. следующий раздел). Тем не менее, в данном последнем случае, по крайней мере, в принципе, отсутствуют причины, по которым нельзя было бы

расширить обычные реляционные системы для обработки таких запросов (и действительно, в некоторых коммерческих программных продуктах это уже сделано[7]. Дополнительная информация по этим темам приведена в разделах 24.6 и 24.7.

- Наконец, как было указано Рейтером [24.10], доказательно-теоретическое представление "обеспечивает правильную трактовку [дополнений к] реляционной модели для включения семантики реального мира в большем объеме" (как было отмечено в разделе 24.2).

24.6. ДЕДУКТИВНЫЕ СИСТЕМЫ БАЗ ДАННЫХ

Дедуктивной СУБД называется такая СУБД, которая поддерживает доказательно-теоретическое представление базы данных и, в частности, обладает способностью осуществлять дедуктивный логический вывод (называемый также просто логическим выводом) дополнительных фактов из тех фактов, которые заданы в экстенсиональной базе данных, путем применения к этим заданным фактам определенных **дедуктивных аксиом** или **правил вывода[8].** Дедуктивные аксиомы, наряду с ограничениями целостности (которые рассматриваются ниже), образуют то, что иногда называют **интенсиональной базой данных,** а экстенсиональная база данных и интенсиональная база данных совместно образуют то, что обычно называют *дедуктивной базой данных* (это не очень удачный термин, поскольку дедуктивный логический вывод осуществляется в СУБД, а не в базе данных).

Как уже было сказано, дедуктивные аксиомы образуют одну часть интенсиональной базы данных. Другая часть состоит из дополнительных аксиом, которые представляют собой ограничения целостности (т.е. правила, основное назначение которых состоит в регламентации обновлений, по существу, с помощью таких правил, которые могут также использоваться в процессе дедуктивного вывода дополнительных фактов из заданных).

Рассмотрим, как может выглядеть обычно применяемая в данной книге база данных поставщиков и деталей в форме *дедуктивной СУБД.* Прежде всего необходимо предусмотреть множество основных аксиом, определяющих допустимые значения в домене.

Примечание. В приведенном ниже описании для удобства чтения применяются по существу такие же соглашения о представлении значений, как и на рис. 3.8 (см. стр. 119) и в других главах данной книги. Таким образом, в качестве удобного сокращения для QTY (3 0 0) используется 3 0 0 и т.д.

```
S# ( S1 ) NAME ( Smith)    INTEGER ( 5 )      CHAR ( London )
S# ( S2 ) NAME( Jones )     INTEGER ( 10 )     CHAR ( Paris )
S# ( S3 ) NAME ( Blake)     INTEGER ( 15 )     CHAR ( Rome )
S# ( S4 ) NAME( Clark )     и т.д.            CHAR ( Athens )
S# ( S5 ) NAME( Adams)                      и т.д.
S# ( S6 ) NAME( White )
S# ( S7 ) NAME( Nut )
и т.д.     NAME( Bolt )
           NAME ( Scre)
             и т.д.
```

[7] Это также предусмотрено в стандарте SQL [4.23]. См. упр. 4.6 из главы 4.

[8] В этой связи следует отметить, что Кодд еще в 1974 году заявил, что одна из задач внедрения реля ционной модели состоит именно в том, "чтобы добиться интеграции средств выборки фактов и управле ния файлами для подготовки к введению в дальнейшем средств логического вывода, применимых в программных продуктах производственного назначения" [12.2], [26.12].

Затем должны быть введены основные аксиомы для кортежей в базовых отношениях следующим образом.

```
S ( SI, Smith, 20,
London ) S ( S2, Jones,
10, Paris )
и т.д.

P ( Pi, Nut, Red, 12, London )
и т.д.                    :

SP ( SI, PI, 300 )
и т.д.
```

Примечание. Безусловно, на основании этого примера не следует делать вывод о том, что экстенсиональная база данных создается путем явного перечисления всех основных аксиом, как показано в этом примере; скорее, для этого должны применяться традиционные методы определения данных и ввода данных. Иными словами, дедуктивные СУБД, как правило, должны применять свои дедуктивные средства к обычным базам данных, которые уже существуют, и были сформированы обычным образом. Но следует отметить, что теперь становится еще более важным требование, чтобы в экстенсиональной базе данных не нарушалось ни одно из объявленных ограничений целостности! Это связано с тем, что база данных, в которой нарушается хотя бы одно из таких ограничений, представляет собой (с точки зрения логики) несовместимое множество аксиом, но широко известно, что, исходя из такой начальной позиции, можно доказать, что *истинным,* вообще говоря, является абсолютно любое высказывание (иными словами, как описано в аннотации [9.16], из несовместимого множества аксиом могут быть выведены взаимно противоречивые утверждения). Именно по той же причине важно также, чтобы было совместимым заданное множество ограничений целостности.

Теперь перейдем к определению интенсиональной базы данных. Ниже приведены ограничения атрибутов.

```
S ( S, sn, St, SC ) ⇒ S# ( S ) AND
                      NAME ( sn ) AND
                      INTEGER { st ) AND
                      CHAR ( sc )

P ( p, pn, p1, pw, pc ) ⇒ P# ( p ) AND
                          NAME ( pn ) AND
                          COLOR ( p1 ) AND
                          WEIGHT ( pw )
                          AND CHAR ( pc )
```

Рассмотрим ограничения потенциального ключа.

```
S ( s, sn1, st1, sc1 ) AND S ( s, sn2, st2, sc2 )
                       ⇒ snl = sn2 AND
                         stl = St2 AND
                         scl =
sc2 и т.д.
```

Ограничения внешнего ключа представлены следующим образом.

```
SP ( s, p, q ) ⇒ S ( s, sn, st, sc ) AND P ( p, pn, pi, pw, pc )
```

Остальная часть базы данных определяется аналогично.

Примечание. Для более полной демонстрации рассматриваемого подхода предположим, что переменные, появляющиеся справа, а не слева от символа импликации (в данном примере sn, st и т.д.), квантифицированы экзистенциально. (Как было описано в разделе 24.4, все остальные переменные квантифицированы универсально.) Поэтому, говоря формально, потребуется применение определенных скулемовских функций; например, переменная sn фактически должна быть заменена (скажем) термом SN(s), где SN — скулемовская функция.

Кстати, следует отметить, что большинство ограничений, показанных в этом примере, не являются в чистом виде такими выражениями, определение которых приведено в разделе 24.5, поскольку их правая часть не представляет собой исключительно дизъюнкцию простых термов.

Теперь введем еще несколько дедуктивных аксиом, как показано ниже.

```
S ( s, sn, st, sc ) AND st > 15
                    ⇒ GOOD_SUPPLIER ( s, st, SC )
```

Рекомендуем сравнить эту аксиому с определением представления GOOD_SUPPLIER (Добросовестный поставщик), которое приведено в разделе 10.1 главы 10).

```
S ( sx, sxn, sxt, sc ) AND S ( sy, syn, syt, sc )
                        ⇒ SS COLOCATED ( SX, sy
) S ( s, sn, st, c ) AND P ( p, pn, p1, pw,
c )
                      ⇒ SP_COLOCATED ( s, p )
```

Могут быть также дополнительно введены другие дедуктивные аксиомы.

Для того чтобы этот пример стал немного более интересным, теперь дополним эту базу данных для включения в нее *переменной отношения* с описанием *структуры детали,* которая показывает, какие детали px включают другие детали py в качестве непосредственных компонентов (т.е. компонентов первого уровня). Вначале введем следующее ограничение, которое показывает, что и переменная px, и переменная py должны обозначать существующие детали.

```
PART_STRUCTURE ( px, py ) ⇒ P ( px, xn, xl, xw, xc ) AND
                             P ( py, yn, yl, yw, yc )
```

Ниже приведены некоторые значения данных.

```
PART STRUCTURE      ( P1,       P2      )
PART STRUCTURE      (P1,        P3      )
PART STRUCTURE      ( P2,       P3      )
 PART STRUCTURE     ( P2,       P4      )
(и т.д.)
```

(На практике переменная отношения PART_STRUCTURE должна была бы также включать параметр *количества,* указывающий, сколько деталей py требуется, чтобы собрать деталь px, но мы для упрощения не используем такое уточнение.)

Теперь введем следующую пару дедуктивных аксиом, позволяющих описать, что подразумевается под утверждением, что деталь px содержит деталь py в качестве компонента (на любом уровне).

```
PART STRUCTURE ( px, py ) ⇒ COMPONENT OF ( px,
py ) PART STRUCTURE ( px, pz ) AND COMPONENT OF (
pz, py )
                          ⇒ COMPONENT_OF ( px, py )
```

Иными словами, деталь py является компонентом детали px (на некотором уровне), если она является либо непосредственным компонентом детали px, либо непосредственным компонентом некоторой детали pz, которая, в свою очередь, является компонентом (на некотором уровне) детали px. Заслуживает особого внимания то, что здесь вторая аксиома рекурсивна — она определяет предикат COMPONENT_OF в терминах самого этого предиката. Отметим, что первоначально в реляционных системах не допускалось использование определений таких представлений (или запросов, или ограничений целостности, или других конструкций), которые были бы заданы как рекурсивные в указанной форме. Поэтому рассматриваемая здесь способность поддерживать рекурсию является одним из наиболее очевидных непосредственных различий между дедуктивными СУБД и их классическими реляционными аналогами. Хотя, как было указано в разделе 24.5 (и описано в главе 7, при обсуждении оператора TCLOSE), фактически не существует таких причин, по которым нельзя было бы расширить классические реляционные системы для поддержки такой рекурсии, и в некоторых системах указанная задача уже была решена. Дополнительные сведения, касающиеся рекурсии, приведены в разделе 24.7.

Язык Datalog

Из приведенного выше описания должно быть очевидно, что одной из частей дедуктивной СУБД, в наибольшей степени непосредственно доступных пользователю, можно считать язык, который позволяет формулировать дедуктивные аксиомы (обычно называемые **правилами).** Широко известным примером такого языка является **Datalog** [24.5], получивший такое название по аналогии с языком Prolog. В данном подразделе приведено краткое описание языка Datalog.

Примечание. Основное внимание при разработке языка Datalog было уделено расширению его описательных возможностей, а не вычислительных средств (в действительности, дело обстояло так же и при реализации первоначальной реляционной модели [6.1]). Цель его создания заключалась в том, что этот язык в конечном итоге должен был обладать более широкими выразительными возможностями, чем обычные реляционные языки [24.5]. В результате этого язык Datalog (а фактически все логические системы баз данных в целом) приобрел четко выраженную направленность на поддержку запросов, а не обновлений. Тем не менее, возможно и желательно расширить этот язык, чтобы он поддерживал также обновления (как описано ниже).

В своей простейшей форме язык Datalog обеспечивает формулировку правил в виде простых хорновских выражений, без функций. В разделе 24.4 хорновское выражение определено как правильно построенная формула, имеющая одну из следующих двух форм.

```
A1 AND A2 AND . . . AND An
A1 AND A2 AND ... AND An ⇒ B
```

Здесь все термы A и терм в представляют собой неотрицаемые экземпляры предикатов, которые включают только константы и переменные. Но в соответствии со стилем, принятым в языке Prolog, в Datalog второе из этих выражений фактически записывается с обратным размещением правой и левой частей, следующим образом.

B ⇐ A1 AND A2 AND . . . AND An

Поэтому для согласования с другими публикациями в этой области в последующем изложении будет применяться такая же форма записи.

В приведенном выше выражении в представляет собой голову правила (или заключение), а A — тело правила (называемое также *предпосылками* или целью; каждый отдельный терм A представляет собой подцель). Для сокращения операторы AND часто заменяются запятыми. Программа Datalog представляет собой множество таких выражений, отделенных друг от друга с применением некоторого общепринятого способа, например, с помощью символов точки с запятой (но в этой главе символы точки с запятой не используются, а вместо этого просто каждое новое выражение записывается на отдельной строке). В такой программе никакого смыслового значения последовательности расположения выражений не предписывается.

Обратите внимание на то, что вся *дедуктивная база данных* может рассматриваться как программа Datalog в указанном выше смысле. Например, можно было бы взять все показанные здесь аксиомы, касающиеся поставщиков и деталей (основные аксиомы, ограничения целостности и дедуктивные аксиомы), записать их все в стиле Datalog (отделив их друг от друга символами точки с запятой или записав на отдельных строках) и получить в результате этого программу Datalog. Но, как было описано выше, экстенсиональная часть базы данных обычно задается не в такой форме, а, скорее, с применением некоторого более привычного способа. Поэтому основное назначение языка Datalog состоит в обеспечении именно формирования дедуктивных аксиом. Как уже отмечалась, такая задача может рассматриваться как расширение механизма определения представлений, который в наши дни предусмотрен в обычных реляционных СУБД.

Datalog может также использоваться как язык запросов (опять-таки, во многом аналогично языку Prolog). Например, предположим, что дано следующее олределение предиката GOOD_SUPPLIER на языке Datalog.

GOOD_SUPPLIER (s, st, sc) ⇐ S (s, sn, st, sc)
AND st > 15

Ниже приведено несколько типичных запросов к предикату GOOD_SUPPLlER.

1. Определить всех добросовестных поставщиков.

   ? ⇐GOOD_SUPPLIER (s, st, sc)

2. Определить всех добросовестных поставщиков из Парижа.

   ? ⇐ GOOD_SUPPLIER (s, st, Paris)

3. Является ли поставщик si добросовестным?

   ? ⇐ GOOD_SUPPLIER (S1, st, sc)

Иными словами, запрос на языке Datalog представляет собой специальное правило с головой, состоящей из вопросительного знака "?", и телом, состоящим из одного терма, который обозначает результат запроса; голова "?" означает (в соответствии с принятым соглашением) "Показать".

Следует отметить, что язык Datalog, в том виде, в котором он был определен первоначально, не поддерживает весьма многих средств обычных реляционных языков: простые вычислительные операторы ("+", "*" и т.д.), операторы агрегирования (COUNT, SUM и т.д.), операцию разности множеств (поскольку нельзя применять операцию отрицания к выражениям), группирования и разгруппирования и т.д. (хотя этот язык поддерживает рекурсию). Кроме того, в языке Datalog не предусмотрена поддержка именования атрибутов (назначение фактического параметра предиката зависит от его порядковой позиции), а также не предусмотрена полная поддержка доменов (т.е. определяемых пользователем типов, в том смысле, который указан в главе 5). К тому же как было отмечено выше в данном разделе, в этом языке не предусмотрены какие-либо операции обновления, а также (как следствие из данного факта) в нем не поддерживаются декларативные спецификации правил удаления и обновления внешних ключей (ON DELETE CASCADE и т.д.).

В целях устранения некоторых из указанных выше ограничений был предложен целый ряд дополнений к базовому языку Datalog. Эти дополнения предназначены для поддержки, кроме всего прочего, перечисленных ниже средств.

- *Отрицаемые предпосылки.* Соответствующий пример приведен ниже.

```
SS_COLOCATED ( sx, sy ) ⇐ S ( sx, sxn, sxt, sc ) AND
                          S ( sy, syn, syt, sc )
                          AND NOT ( sx = sy )
```

- *Вычислительные операторы* (как встроенные, так и оппелеляемые пользователем). Соответствующий пример приведен ниже.

```
P_WT_IN_GRAMS ( p, pn, pl, pg, pc ) <=
                    P ( p, pn, pl, pw, pc ) AND pд = pw * 454
```

  В этом примере предполагается, что знак встроенной функции "*" можно записывать с использованием обычной инфиксной системы обозначений. Более строгим логическим представлением терма, следующего за оператором AND, была бы строка "=(pд,*(pw,454))".

- *Операторы группирования и агрегирования* (во многом соответствующие требовани ям к обычному реляционному оператору SUMMARIZE, который описан в главе 7). Такие операторы необходимы для решения (например) задач, иногда называемых задачами определения необходимого количества (gross requirements problem). Для решения этих задач требуется не только узнать, какие детали ру являются ком понентом некоторой детали рх на любом уровне, но также определить, какое количество различных деталей ру (на всех уровнях) потребуется для изготовле ния детали рх. (Безусловно, при этом предполагается, что переменная отноше ния PART_STRUCTURE включает атрибут QTY.)

- *Операция обновления.* Один из подходов (но не единственный) к реализации этого очевидного требования основан на том наблюдении, что в базовом языке Datalog, во-первых, любой предикат в голове правила должен быть неотрицаемым, и, во-вторых, каждый кортеж, вырабатываемый правилом, может рассматриваться как

"вставляемый" в результат. Поэтому возможное расширение языка может состоять в том, чтобы было разрешено использовать отрицаемые предикаты в голове правила и рассматривать отрицание как запрос на удаление (соответствующих кортежей).

Отличные от хорновских выражения в теле правил. Иными словами, в определении правил может быть разрешено применение правильно построенных формул, имеющих полностью общую форму.

Краткий обзор описанных выше дополнений с примерами можно найти в книге Гардарена (Gardarin) и Валдуриса (Valduriez) [24.6], где также рассматривается целый ряд методов реализации языка Datalog.

24.7. РЕКУРСИВНАЯ ОБРАБОТКА ЗАПРОСОВ

Как было указано в предыдущем разделе, одна из наиболее выдающихся особенностей дедуктивных систем баз данных состоит в том, что в них поддерживается *рекурсия* (рекурсивные определения правил и поэтому также рекурсивные запросы). На основании этого в последние несколько лет проводились интенсивные исследования методов реализации такой рекурсии. В этом разделе приведены некоторые результаты подобных исследований.

В качестве примера еще раз рассмотрим приведенное в разделе 24.6 рекурсивное определение предиката COMPONENT_OF в терминах отношения PART_STRUCTURE (но для краткости здесь вместо PART_STRUCTURE применяется имя PS, а вместо COMPONENT_OF — имя COMP). Соответствующее определение на языке Datalog выглядит следующим образом.

```
COMP ( px, py ) ⇐ PS ( px, py )
COMP ( px, py ) ⇐ PS ( px, pz ) AND COMP ( pz, py )
```

Ниже приведен типичный рекурсивный запрос к этой базе данных ("Показать компоненты детали P1").

```
? ⇐ COMP ( P1, py )
```

Еще раз кратко рассмотрим указанное определение. В этом определении второе правило (т.е. рекурсивное правило) называется **линейно** рекурсивным, поскольку предикат, находящийся в голове правила, появляется в теле правила только один раз. Ниже для сравнения приведено такое определение предиката COMP, в котором второе (рекурсивное) правило не является линейно рекурсивным в указанном выше смысле.

```
COMP ( px, py ) ⇐ PS ( px, py )
COMP ( px, py ) ⇐ COMP ( px, pz ) AND COMP { pz, py )
```

Но в целом в литературе чаще всего высказывается мнение о том, что линейная рекурсия представляет собой "наиболее интересный случай". Под этим подразумевается, что основная часть рекурсивных алгоритмов, применяемых на практике, по своему характеру являются линейными. Кроме того, известны эффективные методы, предназначенные для использования именно в случае линейной рекурсии [24.11]. Поэтому до конца этого раздела ограничимся изучением линейной рекурсии.

Примечание. Для полноты изложения следует отметить, что необходимо обобщить определение *рекурсивного правила* (и линейной рекурсии), чтобы иметь возможность рассматривать более сложные случаи наподобие следующих..

P (x, y) ⇐ Q (x, z) AND R (z, y)

Q (x, y) ⇐ P (x, z) AND S (z, y)

Но для сокращения объема изложения здесь такие уточнения игнорируются. Более подробные сведения на эту тему приведены в [24.11].

Как и в случае обработки классических (т.е. нерекурсивных) запросов, общая проблема реализации какого-то конкретного рекурсивного запроса может быть разделена на две подпроблемы: во-первых, преобразование первоначального запроса в некоторую эквивалентную, но более эффективную форму, а затем, во-вторых, фактическое выполнение программной конструкции, полученной в результате этого преобразования. В литературе можно встретить описание всевозможных подходов к решению обеих указанных проблем. А в этом разделе кратко рассматриваются некоторые из наиболее простых методов на примере их применения для выполнения запроса: "Показать компоненты детали P1" применительно к приведенным ниже данным.

PS

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

Унификация и резолюция

Один из возможных подходов состоит в использовании стандартных методов **унификации** и **резолюции** языка Prolog, как описано в разделе 24.4. В данном примере этот подход осуществляется следующим образом. Первыми предпосылками являются дедуктивные аксиомы, которые выглядят, как показано ниже (в конъюнктивной нормальной форме).

1. `NOT PS ( px, py ) OR COMP ( px, py )`
2. `NOT PS ( px, pz ) OR NOT COMP ( pz, py ) OR COMP ( px, py )`

С использованием требуемого заключения формируется еще одна предпосылка, показанная ниже.

3. NOT COMP (P1, py) OR RESULT (py)

Основные аксиомы образуют оставшиеся предпосылки. Например, рассмотрим приведенную ниже основную аксиому.

4. PS (P1, P2)

После подстановки P1 вместо px и P2 вместо py в строке 1 можно применить правило резолюции к строкам 1 и 4, получив следующий результат.

5. COMP (PI, P2)

Теперь после подстановки P2 вместо ру в строке 3 и применения правила резолюции к строкам 3 и 5 будет получен следующий результат.

6. RESULT (P2)

Итак, деталь P2 является компонентом детали P1. Точно такой же ход доказательства показывает, что деталь P3 также является компонентом детали P1. Таким образом, получены две дополнительные аксиомы (или, скорее, теоремы), COMP (P1, P2) и COMP (P1, P3); теперь можно рекурсивно применить описанный выше процесс для определения всего состава компонентов. Выполнение этих действий оставляем читателю в качестве упражнения.

Но на практике использование процедур унификации и резолюции может привести к весьма значительному снижению производительности. Поэтому обычно требуется найти некоторую более эффективную стратегию. В остальных подразделах данного раздела рассматриваются некоторые возможные подходы к решению этой проблемы.

Примитивный алгоритм вычисления

По-видимому, наиболее простой подход из всех возможных состоит в использовании **примитивного алгоритма вычисления** (naive evaluation) [24.20]. Как показывает его название, этот алгоритм является весьма упрощенным; легче всего его можно описать (применительно к запросу, рассматриваемому в качестве примера) с помощью следующего псевдокода.

```
COMP := PS ;
do until COMP <не достигнет "установившегося состояния "> ;
   COMP := COMP UNION ( COMP * PS ) ;                    ''
end ; DISPLAY := COMP WHERE PX =
P# ('P1') ;
```

Каждая из переменных отношения COMP и DISPLAY (как и переменная отношения PS) имеют два атрибута, PX и PY. Неформально выражаясь, этот алгоритм действует путем повторного формирования промежуточного результата, состоящего из результатов соединения переменной отношения PS с предыдущим промежуточным результатом, до тех пор, пока этот промежуточный результат не достигнет **установившегося состояния** f ixpoint (т.е. пока он не перестанет расти).

Примечание. Выражение "COMP *a* PS" является сокращением от выражения, которое можно описать следующим образом: "получить соединение COMP и PS no COMP.PY и PS.PX, а затем проекцию результата по COMP.PX и PS.PY"; для сокращения объема изложения здесь игнорируются операции переименования атрибутов, которые требуются в используемом диалекте алгебры для обеспечения возможности применения такой операции (см. главу 7).

Теперь поэтапно выполним этот алгоритм с использованием данных, рассматриваемых в качестве примера. После первой итерации цикла выражение COMP * PS принимает значение, показанное ниже (слева), а переменная отношения COMP принимает результирующее значение, показанное справа (кортежи, добавленные в этой итерации, обозначены звездочками).

Следует обратить особое внимание на то, что при вычислении выражения COMP

COMP ⋇ PS

PX	PY
P1	P3
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	*
P1	P5	*
P2	P5	*
P3	P6	*
P4	P6	*

После второй итерации указанные результаты выглядят следующим образом.

COMP ⋇ PS

PX	PY
P1	P3
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6
P1	P6
P2	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	
P1	P5	
P2	P5	
P3	P6	
P4	P6	
P1	P6	*
P2	P6	*

⋇ PS на втором этапе повторяется весь процесс вычисления выражения COMP » PS на первом этапе, а также дополнительно вычисляются некоторые добавочные кортежи. В рассматриваемом случае фактически формируются еще два кортежа, (P1,P6) и (P2,P6). В этом состоит одна из причин, по которой данный примитивный алгоритм вычисления является не очень эффективным.

После третьей итерации (при которой еще раз повторно выполняются те же вычисления) обнаруживается, что значение выражения COMP ⋇ PS остается таким же, как и после предыдущей итерации. Таким образом, переменная отношения COMP достигла установившегося состояния и происходит выход из цикла. Затем вычисляется следующий окончательный результат как сокращение переменной отношениясOMP.

COMP

PX	PY
P1	P2
P1	P3
P1	P4
P1	P5
P1	P6

Теперь становится очевидной другая важная причина низкой эффективности: фактически при использовании данного алгоритма вычисляется состав компонентов каждой детали (в действительности, в нем вычисляется все транзитивное замыкание отношения PS), а затем снова отбрасываются все результаты, кроме фактически требуемых кортежей. Иными словами, еще раз отметим, что при использовании этого алгоритма выполняется большой объем ненужной работы.

В завершении этого подраздела следует отметить, что описанный здесь метод, основанный на применении примитивного алгоритма, может рассматриваться как реализация процедуры прямого логического вывода: он начинает свою работу с исходной экстенсиональной базы данных (т.е. исходя из фактических значений данных), после этого повторно применяются предпосылки данного определения (т.е. тело правила) до тех пор, пока не будет получен требуемый результат. В действительности, этот алгоритм вычисляет фактически минимальную модель для программы Datalog (см. разделы 24.5 и 24.6).

Полупримитивный алгоритм вычисления

Первое очевидное усовершенствование примитивного алгоритма вычисления состава компонентов состоит в предотвращении повторного выполнения на следующем этапе тех вычислений, которые были выполнены на предыдущем этапе, поэтому он называется полупримитивным алгоритмом вычисления (seminaive evaluation) [24.23]. Иными словами, теперь на каждом этапе вычисляются только новые кортежи, которые должны быть добавлены в данной конкретной итерации. Основную идею этого алгоритма снова рассмотрим на примере запроса "Определить компоненты детали P1". Соответствующий псевдокод показан ниже.

```
NEW := PS ;
COMP := NEW ;
do until NEW <не пуста> ;
  NEW := ( NEW * PS ) MINUS COMP ;
  COMP := COMP UNION NEW ; end ;
DISPLAY := COMP WHERE PX = P#
('P1') ;
```

Снова рассмотрим поэтапно процесс выполнения этого алгоритма. При первоначальном вхождении в цикл обе переменные отношения, NEW и COMP, идентичны PS, как показано ниже.

Переменная отношения COMP является такой же, какой она была на аналогичном

NEW

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

COMP

PX	PY
P1	P2
P1	P3
P2	P3
P2	P4
P3	P5
P4	P5
P5	P6

В конце первой итерации они выглядят следующим образом.

NEW

PX	PY
P1	P4
P1	P5
P2	P5
P3	P6
P4	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	*
P1	P5	*
P2	P5	*
P3	P6	*
P4	P6	*

этапе реализации примитивного алгоритма, а NEW содержит только новые кортежи, которые были добавлены к COMP в этой итерации; здесь заслуживает внимания то, что NEW не включает кортеж (P1,P3) (в отличие от аналога этой переменной отношения, применяемого в примитивном алгоритме). В конце следующей итерации будут получены приведенные ниже результаты.

NEW

PX	PY
P1	P6
P2	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P2	P3	
P2	P4	
P3	P5	
P4	P5	
P5	P6	
P1	P4	
P1	P5	
P2	P5	
P3	P6	
P4	P6	
P1	P6	*
P2	P6	*

После следующей итерации NEW остается пустой, поэтому осуществляется выход из цикла.

Алгоритм статической фильтрации

Алгоритм **статической фильтрации** представляет собой усовершенствование основной идеи классической теории оптимизации, согласно которой операции сокращения должны применяться как можно раньше. Он может рассматриваться как практическое применение подхода, основанного на обратном логическом выводе, поскольку в этом алгоритме, по существу, используется информация из запроса (заключение) для модификации правил (предпосылок). Этот алгоритм называют также алгоритмом сокращения множества релевантных (относящихся к делу) фактов, поскольку в нем (еще раз подчеркнем) используется информация из запроса для удаления с самого начала ненужных кортежей из экстенсиональной базы данных [24.24]. Соответствующие действия можно описать на рассматриваемом примере с помощью следующего псевдокода.

```
NEW := PS WHERE PX = P# ('P1') ;
COMP := NEW ;
do until NEW <не пуста> ;
  NEW := ( NEW PS ) MINUS COMP ;
  COMP := COMP UNION NEW
; end ; DISPLAY := COMP ;
```

Еще раз выполним этот алгоритм поэтапно. При первоначальном вхождении в цикл обе переменные отношения, NEW и COMP, выглядят следующим образом.

NEW

PX	PY
P1	P2
P1	P3

COMP

PX	PY
P1	P2
P1	P3

В конце первой итерации они представлены следующим образом.

NEW

PX	PY
P1	P4
P1	P5

COMP

PX	PY	
P1	P2	
P1	P3	
P1	P4	*
P1	P5	*

В конце следующей итерации будет получен следующий результат.

NEW

PX	PY
P1	P6

COMP

PX	PY	
P1	P2	
P1	P3	
P1	P4	
P1	P5	
P1	P6	*

После следующей итерации переменная отношения NEW остается пустой, поэтому происходит выход из цикла.

На этом краткое вводное описание стратегий обработки рекурсивных запросов завершается. Безусловно, в литературе было предложено много других подходов, причем большинство из них являются намного более развитыми по сравнению с теми простыми

алгоритмами, которые были описаны в данном разделе. Но объем книги такого характера, как эта, не позволяет привести весь методический материал, необходимый для полного понимания этих подходов. Дополнительные сведения по этой теме можно найти, например, в [24.11] — [24.25].

24.8. РЕЗЮМЕ

На этом завершается краткое введение в проблематику баз данных, основанных на логике. Хотя изложенные здесь идеи еще не привлекли значительного интереса в сообществе исследователей в области баз данных, некоторые из них уже начали постепенно внедряться в коммерческие реляционные продукты (это замечание особенно справедливо в отношении определенных методов оптимизации). В целом концепция баз данных на основе логики выглядит весьма перспективной; в предыдущих разделах указано несколько потенциальных преимуществ, связанных с их использованием. Еще одно преимущество, которое не было явно указано в данной главе, состоит в том, что логика может служить основой обеспечения более тесной интеграции между языками программирования общего назначения и базами данных. Иными словами, вместо подхода со *встроенными подъязыками данных,* поддерживаемого современными программными продуктами SQL, который нельзя назвать особенно изящным (если не сказать большего), в системе можно предусмотреть единый язык, основанный на логике, в котором "данные являются данными", независимо от того, хранятся ли они в разделяемой базе данных или используются локально в самом приложении. (Безусловно, прежде чем удастся достичь этой цели, придется преодолеть множество препятствий. Не последнее из них состоит в том, что нужно прежде всего наглядно продемонстрировать всему сообществу пользователей информационными технологиями в целом, что логика— подходящая основа для языка программирования общего назначения.)

Теперь кратко повторим основные темы приведенного в этой главе материала. Вначале были представлены краткие учебные разделы по **исчислению высказываний** и **исчислению предикатов,** а также введены, кроме всего прочего, описанные ниже понятия.

- **Интерпретация** множества правильно построенных формул представляет собой сочетание, во-первых, предметной области, во-вторых, преобразования, которое связывает отдельные константы, применяемые в правильно построенных формулах этого множества с объектами указанной предметной области, и, в-третьих, множества определенных смысловых значений для предикатов и функций, применяемых в этих правильно построенных формулах.
- **Моделью** для множества правильно построенных формул является интерпретация, при которой все правильно построенные формулы этого множества принимают истинные значения. Каждое конкретное множество правильно построенных формул (вообще говоря) может иметь любое количество моделей.
- **Доказательство** — это процесс демонстрации того, что некоторая конкретная правильно построенная формула g **(заключение)** является логическим следствием из
- некоторого конкретного множества правильно построенных формул f1, f2, . . ., f n **(предпосылок). В** этой главе достаточно подробно был описан один из методов доказательства, который состоит из процедур, называемых **резолюцией** и **унификацией.**

Затем было описано **доказательно-теоретическое представление** баз данных. В таком представлении база данных рассматривается как состоящая из комбинации **экстенсиональной базы данных** и **интенсиональной базы данных.** Экстенсиональная база данных содержит **основные аксиомы** (т.е., говоря неформально, базовые данные); интенсиональная база данных содержит ограничения целостности и **дедуктивные аксиомы** (т.е., снова неформально говоря, представления). В таком случае *смысловое значение* базы данных состоит из множества теорем, которые можно дедуктивно вывести из аксиом; процесс выполнения запроса становится (по крайней мере, концептуально) процессом **доказательства теоремы. Дедуктивная СУБД** — это такая СУБД, которая поддерживает доказательно-теоретическое представление. Кроме того, кратко описан **Datalog** — пользовательский язык для подобной СУБД.

Одним из непосредственно очевидных различий между языком Datalog и классическими реляционными языками является то, что Datalog поддерживает **рекурсивные аксиомы** и поэтому — рекурсивные запросы, хотя нет никаких причин, по которым нельзя было бы расширить классическую реляционную алгебру и исчисление для выполнения такой же задачи[9] (см. описание оператора TCLOSE в главе 7). В данной главе описаны некоторые простые методы выполнения таких запросов.

В заключение необходимо отметить, что эта глава началась **с** упоминания целого ряда терминов, которые часто встречаются в научной литературе (и даже в определенной степени в рекламных объявлениях поставщиков). К ним относятся такие термины, как *логическая база данных; СУБД, основанная на логическом выводе; дедуктивная СУБД* и т.д. Поэтому в конце данного резюме приведены определения некоторых из этих терминов. Но следует предупредить читателя, что по этим вопросам не всегда достигнуто согласие и поэтому в литературе вполне можно также найти другие определения. А приведенные ниже определения автор настоящей книги считает наиболее приемлемыми.

- *Рекурсивная обработка запросов.* По поводу этого термина разногласия обычно не возникают. Рекурсивной обработкой запросов называется анализ и, в частности, оптимизация запросов, которые по определению являются рекурсивными (см. раздел 24.7).
- *База знаний.* Этот термин иногда используется для обозначения того понятия, которое в разделе 24.6 было определено как *интенсиональная база данных.* Иными словами, база знаний состоит из правил (ограничений целостности и дедуктивных аксиомы), в отличие от собственно базы данных, которая составляет экстенсиональную базу данных. Однако многие авторы используют термин *база знаний* для обозначения комбинации экстенсиональной и интенсиональной баз данных, если не считать того, что некоторые специалисты утверждают (как, например в [24.6]), что "база знаний часто содержит сложные объекты, [а также] классические отношения" (описание *сложных объектов* приведено в части VI настоящей книги). Наконец, в системах обработки естественных языков этот термин имеет другое, более специализированное значение, поэтому, видимо, лучше вообще избегать его использования.

---

[9] В этой связи любопытно отметить, что реляционные СУБД так или иначе должны быть способны ми (незаметно для пользователя) выполнять рекурсивную обработку, поскольку каталог содержит неко торую информацию, структурированную рекурсивно (дело в том, что определения представлений часто бывают выражены в терминах других определений представлений).

- *Знания.* Еще один термин, практически не вызывающий разногласий! Знаниями называют то, что содержится в базе знаний... К сожалению, при использовании этой формулировки проблема определения понятия *знания* сводится к описанной выше нерешенной проблеме окончательного определения понятиябаза *знаний.*
- *Система управления базой знаний* (СУБЗ). Программное обеспечение, которое управляет базой знаний. Этот термин обычно используется в качестве синонима для дедуктивной СУБД (см. следующий абзац).
- *Дедуктивная СУБД.* СУБД, которая поддерживает доказательно-теоретическое представление баз данных и, в частности, обладает способностью дедуктивно вы водить дополнительную информацию из экстенсиональной базы данных, приме няя правила вывода (т.е. дедуктивные правила), которые хранятся в интенсио нальной базе данных. Фактически обязательным требованием к дедуктивной СУБД является поддержка рекурсивных правил и поэтому осуществление обра ботки рекурсивных запросов.
- *Дедуктивная база данных* (не рекомендуемый термин). База данных, управляемая дедуктивной СУБД.
- *Экспертная СУБД.* Синоним *дедуктивной СУБД.*
- *Экспертная база данных* (не рекомендуемый термин). База данных, управляемая экспертной СУБД.
- *СУБД, поддерживающая логический вывод.* Синоним для *дедуктивной СУБД.*
- *Система, основанная на логике.* Синоним для *дедуктивной СУБД.*
- *Логическая база данных* (не рекомендуемый термин). Синоним для *дедуктивной базы данных.*
- *Логика как модель данных.* Любая модель данных состоит (по меньшей мере) из объектов, правил поддержки целостности и операторов. В дедуктивной СУБД все эти компоненты (объекты, правила поддержки целостности и операторы) представлены в некоторой единообразной форме (а именно, как аксиомы в таком языке, основанном на логике, как Datalog); в действительности, как было описано в разделе 24.6, база данных в такой системе может рассматриваться именно как логическая программа, состоящая из аксиом этих трех типов. Поэтому вполне обоснованным является утверждение, что абстрактной моделью данных для такой системы служит сама логика.

УПРАЖНЕНИЯ

24.1. Используя метод резолюции, определите, являются ли приведенные ниже метаутверждения правильными доказательствами в исчислении высказываний.

а) `A ⇒ B, C ⇒ B, D ⇒ (A OR C), D ├ B`

б)
```
( A ⇒ B ) AND ( C ⇒ D ) , ( B ⇒ E AND D ⇒ F ) ,
    NOT ( E AND F ), A ⇒ C ├ NOT A
```

в)
```
(A OR B ) ⇒ D, D ⇒ NOT ( E OR F ) ,
    NOT ( B AND C AND E ) ├ NOT ( G ⇒ NOT ( C AND H ) )
```

24.2. Преобразуйте следующие правильно построенные формулы в клаузальную форму.

а) `FORALL x ( FORALL y ( p ( x, y ) ⇒ EXISTS z ( q ( x, z ) ) ) )`

б) `EXISTS x ( EXISTS y ( p ( x, y ) ⇒ FORALL z ( q ( x, z ) ) ) )`

в) `EXISTS x ( EXISTS y ( p ( x, y ) ⇒ EXISTS z ( q ( X, z ) ) ) )`

24.3. Ниже приведен довольно типичный пример логической базы данных.

```
MAN ( Adam )                    /* Мужчина */
WOMAN ( Eve )                   /* Женщина */
MAN ( Cain )
MAN ( Abel )
MAN ( Enoch )
PARENT ( Adam, Cain ) /* Родитель */
PARENT ( Adam, Abel )
PARENT ( Eve, Cain )
PARENT ( Eve, Abel )
PARENT ( Cain, Enoch )
FATHER ( x, y ) ⇐ PARENT ( x, y ) AND MAN ( X )   /* Отец */
MOTHER ( x, y ) ⇐ PARENT ( x, y ) AND WOMAN ( x ) /* Мать */
SIBLING ( x, y ) ⇐ PARENT ( z, x ) AND PARENT ( z, y )   /*
Брат или сестра */
BROTHER ( x, y ) ⇐ SIBLING ( x, y ) AND MAN ( x ) /* Брат */

SISTER ( x, y ) ⇐ SIBLING ( x, y ) AND WOMAN ( x )/*Сестра */

ANCESTOR ( x, y ) ⇐ PARENT ( x, y )                /* Предок */
ANCESTOR ( x, y ) ⇐ PARENT ( x, z ) AND ANCESTOR ( z, y )
```

Примените метод резолюции для получения ответов на следующие вопросы.

а) Кто является матерью Каина (Cain)?

б) Кто является братьями и сестрами Каина?

в) Кто является братьями Каина?

г) Кто является сестрами Каина?

д) Кто является предками Еноха (Enoch)?

24.4. Дайте определение терминов *интерпретация* и *модель*.

24.5. Сформируйте множество аксиом языка Datalog, относящихся (только) к определениям базы данных поставщиков, деталей и проектов.

24.6. Приведите решения упр. 7.13-7.50 на языке Datalog там, где это возможно.

24.7. Приведите решения упр. 9.3 на языке Datalog там, где это возможно.

24.8. Завершите самостоятельно приведенное в разделе 24.7 описание процесса реализации процедур унификации и резолюции, связанного с получением ответа на запрос: "Найти все компоненты детали с номером P1".

СПИСОК ЛИТЕРАТУРЫ

В последние годы количество публикаций в области логических СУБД быстро возрастает, и приведенный ниже список представляет собой лишь малую часть всей имеющейся на сегодня литературы. Список включает следующие тематические группы.

- Книги [24.1]-[24.5] или посвящены логике в целом (отчасти в контексте вычисле ний, отчасти в контексте баз данных), или представляют собой сборники статей именно по базам данных, основанным на логике.
- Работы [24.6] и [24.7] являются учебными пособиями.
- Работы [24.9], [24.12]-[24.15], [24.25], [24.27] и [24.28] посвящены операции тран зитивного замыкания и ее реализации.
- В работах [24.16]—[24.**19]** описан важный метод обработки рекурсивных запро сов — "магические" множества, а также различные его варианты.

Примечание. С этим вопросом также связаны публикации [18.22]—[18.24].

Остальные публикации демонстрируют масштабность исследований, которые ведутся в этой области. В них рассматриваются дополнительные аспекты данной темы и они представлены в данном списке литературы в основном без комментариев.

24.1. Stoll R.R. Sets, Logic and Axiomatic Theories. — San Francisco, Calif: W.H. Freeman and Company, 1961.

Представляет собой весьма неплохое введение в логику.

24.2. Gray P.M.D. Logic, Algebra and Databases. — Chichester, England: Ellis Horwood Ltd., 1984.

Книга является прекрасным введением в исчисление высказываний и исчисление предикатов с точки зрения пользователя базы данных. В ней также освещены другие темы, имеющие непосредственное отношение к данной теме.

24.3. Gallaire H., Minker J. Logic and Data Bases.— New York, N.Y.: Plenum Publishing Corp., 1978.

Один из первых сборников статей по данной теме (если *несомый* первый).

24.4. Minker J. (ed.). Foundations of Deductive Databases and Logic Programming. — San Mateo, Calif: Morgan Kaufmann, 1988.

24.5. Ullman J.D. Database and Knowledge-Base Systems. В 2-х томах. — Rockville, Md: Computer Science Press, 1988, 1989.

Одна (самая большая) из десяти глав этой двухтомной книги целиком посвящена логическим методам. В этой большой по объему главе (в которой впервые был представлен язык Datalog) обсуждается связь между логической и реляционной алгеброй, а также реляционное исчисление в виде особого случая логического подхода — версии как для доменов, так и для кортежей. Второй том состоит из семи глав, пять из которых посвящены различным аспектам логических СУБД.

24.6. Gardarin G., Valduriez P. Relational Databases and Knowledge Bases. — Reading, Mass.: Addison-Wesley, 1989.

В книге имеется глава, посвященная дедуктивным системам, в которой несмотря на вводный характер изложения содержатся сведения о теории таких систем, об алгоритмах оптимизации и т.д., причем в гораздо большем объеме, чем в данной главе.

24.7. Gallaire H., Minker J., Nicolas J.-M. Logic and Databases: A Deductive Approach // ACM Сотр. Surv. — June 1984. — 16, № 2.

24.8. Dahl V. On Database Systems Development through Logic // ACM TODS. — March 1982.-7, №1.

Хорошее и ясное описание идей, лежащих в основе логических СУБД, с примерами, созданными автором на базе языка Prolog в 1977 году.

24.9. Agrawal R. Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries // IEEE Transaction on Software Engineering. — July 1988. — 14, № 7.

Здесь предлагается новый оператор *alpha,* с помощью которого в рамках традиционной реляционной алгебры поддерживается формулировка "большого класса рекурсивных запросов" (в действительности — надмножество линейных рекурсивных запросов). Существует мнение, что оператор *alpha* является достаточно мощным для решения многих практических проблем, касающихся рекурсии, и более простым в применении по сравнению с другими общими механизмами рекурсии. В статье приводится несколько примеров его использования. В частности, показано, как могут быть решены задачи вычисления транзитивного замыкания и определения необходимого количества (gross requirements) (см., соответственно, публикацию [24.12] и раздел 24.6).

В [24.14] описаны некоторые работы по реализации этого оператора. К данной теме относится также [24.13].

24.10. Reiter R. Towards a Logical Reconstruction of Relational Database Theory // M.L. Brodie, J. Mylopoulos, J.W. Schmidt (eds.). On Conceptual Modelling: Perspectives from Artificial Intelligence, Databases, and Programming Languages. — New York, N.Y.: Springer-Verlag, 1984.

Как уже отмечалось в разделе 24.2, хотя работа Рейтера и не является первоисточником в этой области, многие исследователи до него изучали связь между логикой и базами данных (см., например, [24.3], [24.4] и [24.8]). Однако, по всей вероятности, именно "логическая реконструкция реляционной теории", выполненная Рейтером, значительно повысила интерес к предмету и побудила к активным исследованиям в этой области.

24.11. Bancilhon F., Ramakrishnan R. An Amateur's Introduction to Recursive Query Processing Strategies // Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data. — Washington, D.C., 1986. (В пересмотренном виде опубликована в сборнике M. Stonebraker (ed.). Readings in Database Systems. — San Mateo, Calif: Morgan Kaufmann, 1988.)

Прекрасный обзор положительных и отрицательных сторон существующих методов решения проблемы реализации рекурсивного запроса. Положительно оценивается наличие множества технологий, разработанных для решения этой проблемы, а отрицательно — трудности выбора технологии, наиболее подходящей для конкретной ситуации (в частности, многие технологии представлены в литературе без описания характеристик производительности). После описания основных идей логических баз данных в статье предлагается множество алгоритмов — примитивный и полупримитивный алгоритм, итерационные запросы и подзапросы, рекурсивные запросы и подзапросы, система APEX, язык Prolog, алгоритмы Хеншена-Нэкви, Ахо-Ульмана, Кифера-Лозинского, вычислительный алгоритм, "магические" множества и обобщенные "магические" множества. Сравниваются различные подходы на основе прикладной области (т.е. класса проблем, для которых успешно применяется данный алгоритм), производительности и простоты их реализации. В статье также приводятся показатели производительности (со сравнительным анализом), полученные в результате проверки различных алгоритмов на основе простого эталонного теста.

24.12. Ioannidis Y.E. On the Computation of the Transitive Closure of Relational Operators // Proc. 12th Int. Conf. on Very Large Data Bases. — Kyoto, Japan. — August 1986.

Предложен алгоритм реализации транзитивного замыкания, действующий по принципу "разделяй и властвуй". См. также [24.9], [24.13]—[24.15], [24.27] и [24.28].

24.13. Jagadish H.V., Agrawal R., Ness L. A Study of Transitive Closure as a Recursion Mechanism // Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data. — San Francisco, Calif. — May 1987.

Немного переработанная цитата из аннотации: "В этой статье показано, что каждый линейный рекурсивный запрос может быть выражен в виде операции транзитивного замыкания, которой могут предшествовать и за которой могут следовать операции, уже существующие в реляционной алгебре". В статье также делается предположение, что для эффективной реализации линейной рекурсии в общем случае, а значит, и для создания эффективных дедуктивных СУБД для большого класса рекурсивных проблем достаточно обеспечить эффективную реализацию транзитивного замыкания.

24.14. Agrawal R., Jagadish H. Direct Algorithms for Computing the Transitive Closure of Database Relations // Proc 13th Int. Conf. on Very Large Data Bases.— Brighton, UK. - September 1987.

Предлагается набор алгоритмов для транзитивного замыкания, которые "не представляют задачу как вычисление рекурсии, а получают замыкание, опираясь на исходные принципы" (отсюда и определение *direct*— "непосредственный" — в заголовке). В статье также содержится полезное резюме предыдущих работ по другим непосредственным алгоритмам.

24.15. Lu H. New Strategies for Computing the Transitive Closure of a Database Relation // Proc 13th Int. Conf. on Very Large Data Bases. — Brighton, UK. — September 1987.

Приводится описание других алгоритмов транзитивного замыкания. Так же, как и в [24.14], в этой статье содержится полезный обзор разработанных ранее подходов к рассматриваемой проблеме.

24.16. Bancilhon F., Maier D., Sagiv Y., Ullman J.D. Magic Sets and Other Strange Ways to Implement Logic Programs // Proc. 5th ACM SIGMOD-SIGFACT Symposium on Principles of Database Systems, 1986.

Основная идея "магических" множеств состоит в том, что новые правила преобразования ("магические" правила) вводятся в процесс оптимизации динамически. Эти правила используются для замены первоначального запроса модифицированной версией, которая является более эффективной, поскольку сокращает множество "относящихся к делу фактов" (см. раздел 24.7). Изложение подробностей несколько сложно для восприятия и выходит за рамки данного комментария. Читателю предлагается прочесть эту статью или обратиться к подготовленному авторами данной статьи Бансильоном и Рамакришнаном обзору [24.11] и книгам Ульмана [24.5], Гардарена и Вальдуриса [24.6], чтобы получить более подробные сведения. Однако следует отметить, что существуют также многочисленные варианты этой основной идеи [24.17]—[24.19]. Кроме того, следует учесть работы [18.22]—[18.24].

24.17. Beeri C, Ramakrishnan R. On the Power of Magic // Proc. 6th ACM SIGMOD-SIGFACT Symposium on Principles of Database Systems, 1987.

24.18. Sacca D., Zaniolo C Magic Counting Methods // Proc. 1987 ACM SIGMOD Int. Conf. on Management of Data. — San Francisco, Calif. — May 1987.

24.19. Gardarin G. Magic Functions: A Technique to Optimize Extended Datalog Recursive Programs // Proc. 13th Int. Conf. on Very Large Data Bases.— Brighton, UK.- September 1987.

24.20. Aho A., Ullman J.D. Universality of Data Retrieval Languages // Proc. 6th ACM Symposium on Principles of Programming Languages. — San Antonio, Texas, 1979.

Для данной последовательности отношений r, *f(r), f(f(r)),...* (где *f*— некоторая постоянная функция) минимальным установившимся состоянием последовательности называется отношение r*, выведенное в соответствии со следующим алгоритмом примитивного вычисления (см. раздел 24.7).

```
r* := r ;
do until r* <не перестает расти> ;
   r* := r* UNION f(r*) ;
end ;
```

В этой статье предложено ввести в реляционную алгебру оператор вычисления "минимального установившегося состояния" (least fixpoint).

24.21. Ullman J.D. Implementation of Logical Query Languages for Databases // ACM TODS. - September 1985. - 10, № 3.

Описан важный класс технологий реализации возможных рекурсивных запросов. Методы определены в терминах "правил перехвата" (capture rule), на "деревьях правил/целей" (rule/goal tree), которые являются графами, представляющими

некоторую стратегию запроса в терминах предложений и предикатов. В статье определено несколько таких правил: одно из них соответствует приложению операторов реляционной алгебры, еще два правила соответствуют прямому и обратному логическому выводу, а "косвенное" правило позволяет передавать результаты от одной подцели к другой. Косвенная передача информации затем становится основой для так называемых методов *"магических"множеств* [24.16]—[24.19].

24.22. Tsur S., Zaniolo C. LDL: A Logic-Based Data-Language // Proc. 12th Int. Conf. on Very Large Data Bases. — Kyoto, Japan. — August 1986.

Язык LDL включает генератор типа "множества", отрицание (основанное на разности множеств), операции определения данных и операции обновления. Он представляет собой язык чисто логического типа (нет никаких зависимостей упорядочения между утверждениями), причем компилируемый, а не интерпретируемый.

24.23. Bancilhon F. Naive Evaluation of Recursive Defined Relations // M. Brodie and J. Mylopoulos (eds). On Knowledge Base Management Systems: Integrating Database and AI Systems. — New York, N.Y.: Springer-Verlag, 1986.

24.24. Lozinskii E.L. A Problem-Oriented Inferential Database System // ACM TODS. — September 1986.- 11,№3.

В этой статье впервые введено понятие "релевантных (относящихся к делу) фактов". В ней описан прототип системы, в которой применяется экстенсиональная база данных для компенсации очень быстрого расширения пространства поиска, обычно происходящего при использовании методов логического вывода.

24.25. Rosenthal A. et al. Traversal Recursion: A Practical Approach to Supporting Recursive Applications // Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data.— Washington, D.C. — June 1986.

24.26. Kifer M., Lozinskii E. On Compile-Time Query Optimization in Deductive Databases by Means of Static Filtering//ACM TODS. — September 1990. — 15, № 3.

24.27. Agrawal R., Dar S., and Jagadish H.V. Direct Transitive Closure Algorithms: Design and Performance Evaluation//ACM TODS. — September 1990. — 15, № 3.

24.28. Jagadish H.V. A Compression Method to Materialize Transitive Closure // ACM TODS. - December 1990. - 15, № 4.

Предлагается метод индексации, который позволяет хранить транзитивное замыкание данного отношения в сжатой форме, таким образом, чтобы можно было выполнить проверку и узнать с помощью одной справочной таблицы с индексом, будет ли данный кортеж находиться в замыкании.

ЧАСТЬ VI

ОБЪЕКТЫ, ОТНОШЕНИЯ И ЯЗЫК XML

Примечание. Как и в главе 20, в главах этой части книги широко используется материал, впервые представленный в главе 5. Поэтому, если читатель вначале лишь поверхностно ознакомился с указанной главой, то рекомендуем вернуться к ней и внимательно прочитать, прежде чем приступать к глубокому изучению глав данной части.

Объектная технология является важной областью в сфере разработки программного обеспечения в целом. Поэтому возникает резонный вопрос — насколько важна эта технология, в частности, в сфере управления базами данных, и если важна, то каково ее значение для этой сферы. Но единого мнения по этому вопросу пока нет! Однако создается впечатление, что появляется основа для определенного консенсуса. Когда системы объектных баз данных впервые появились на рынке, некоторые ведущие специалисты в области объектного программирования заявили, что такие системы вскоре завоюют мир и полностью вытеснят реляционные системы. А другие авторитетные исследователи высказали мнение, что объектные системы подходят лишь для определенного, очень ограниченного круга задач и никогда не займут сколько-нибудь значительную часть рынка баз данных. В разгар этих дискуссий стали появляться системы, олицетворяющие собой "компромиссный путь развития", — в них объединены объектные и реляционные технологии в целях воплощения наилучших свойств обеих систем. А в последнее время складывается впечатление, что "другие авторитетные исследователи" были правы — чисто объектные системы могут иметь определенную область применения, но эта область занимает лишь небольшую нишу на всем рынке баз данных, а реляционные системы в обозримой перспективе будут по-прежнему доминировать на этом рынке. Не последнюю роль в этом играет то, что объектно-реляционные системы в конечном итоге являются просто реляционными системами (что и показано в главе 26).

В последнее время значительное внимание привлекает определенный тип объектов — документы XML; проблема хранения таких документов в базе данных, выполнения запросов к этим документам и их обновления быстро приобретает важное практическое значение. Возможно создание так называемых *баз XML,* т.е. баз данных, которые содержат только документы XML и не что-либо иное; но, безусловно, было бы предпочтительней, по возможности, интегрировать документы XML с другими данными объектной или реляционной (или объектно-реляционной) базы данных.

В главах этой части книги приведенный выше вопрос рассматривается достаточно глубоко. В главе 25 обсуждаются чисто объектные системы, в главе 26 рассматриваются объектно-реляционные системы, а в главе 27 рассматривается язык XML.

ГЛАВА 25

Объектные базы данных

25.1. ВВЕДЕНИЕ

В период с конца 1980-х годов до середины 1990-х годов системы *объектно-ориентированных* баз данных (или сокращенно — **объектные системы)** вызывали значительный интерес. В тот период некоторые исследователи рассматривали объектные системы как серьезного конкурента реляционных систем (или, во всяком случае, конкурента систем SQL). В наши дни с такой позицией соглашаются лишь немногие; большинство специалистов в области информационных технологий теперь считают, что объектные системы, возможно, имеют определенную область применения, но эта область является довольно ограниченной [25.33]. Тем не менее, такие системы все еще заслуживают внимательного изучения. Поэтому в данной главе подробно рассматриваются объектные системы; здесь представлены и описаны основные объектные понятия; там, где это уместно, эти понятия подвергнуты анализу и критике, а также приведены некоторые выводы относительно перспектив применения этих понятий в системах баз данных в будущем.

Почему же возник такой большой интерес к объектным системам? Общеизвестно, что уже ставшие классическими системы SQL были (и, безусловно, остаются) несовершенными во многих отношениях. К тому же в литературе можно встретить даже такие утверждения, что лежащая в их основе теория (т.е. реляционная модель) также не отвечает современным требованиям. Как бы то ни было, некоторые из новых возможностей, которые считаются необходимыми в современных СУБД, уже много лет существуют в

объектно-ориентированных языках программирования, например в C++ и Smalltalk. И, вполне естественно, возникла идея реализовать эти возможности в системах баз данных, что и было сделано многими исследователями и несколькими производителями СУБД.

Таким образом, объектные системы берут свое начало от объектно-ориентированных языков программирования. Основной замысел, объединяющий эти две области, состоит в том, что нужно избавить пользователя от необходимости иметь дело с такими машинно-ориентированными конструкциями, как биты и байты (или даже записи и поля). Вместо этого пользователю должен предоставляться доступ к объектам и **операциям** над этими объектами, которые в большей степени соответствуют своим аналогам в реальном мире. Например, используя традиционные термины, можно представлять отдел, как "кортеж DEPT" с набором соответствующих "кортежей EMP", т.е. сотрудников, которые имеют "значения внешних ключей", "ссылающихся" на "значение первичного ключа" в "кортеже DEPT". А в новой технологии пользователь будет иметь дело с *объектом отдела,* который содержит соответствующее множество *объектов сотрудников.* И вместо выполнения операции "вставки" нового "кортежа" в "переменную отношения EMP" с соответствующим "значением внешнего ключа", "указывающего" на "значение первичного ключа" некоторого "кортежа" в "переменной отношения DEPT", пользователь должен иметь возможность непосредственно *"принять"* сотрудника (представленного объектом) на работу в отдел (также представленный соответствующим объектом). Иначе говоря, фундаментальная идея объектного подхода — **повышение уровня абстракции.**

Безусловно, повышение уровня абстракции — цель чрезвычайно привлекательная, и объектные понятия успешно использовались для ее достижения в сфере языков программирования. Поэтому возникает резонный вопрос: можно ли те же понятия применить в области баз данных? Действительно, с точки зрения пользователя работа со *сложными объектами,* например, представляющими отделы, которые "знают, как" принять нового сотрудника, сменить руководителя отдела или урезать бюджет отдела, выглядит более привлекательной (по крайней мере, на первый взгляд), чем необходимость оперировать понятиями *переменная отношения, вставка кортежа, внешний ключ* и т.п.

Однако здесь необходимо сделать следующее предостережение. Несмотря на то, что между языками программирования и теорией управления базами данных, бесспорно, имеется много общего, в некоторых весьма важных аспектах они все же различаются. В частности, ниже перечислены наиболее важные различия.

- Прикладная программа по определению предназначена для решения некоторых конкретных задач.
- В отличие от этого, база данных (опять же, по определению) предназначена для решения целого ряда различных задач, формулировка которых может быть даже не известна в момент создания базы данных.

Поэтому в области создания средств прикладного программирования, предназначенных для разработки отдельных приложений, включение в сложные объекты дополнительных *интеллектуальных возможностей,* очевидно, является разумным решением. За счет этого сокращается объем кода, который должен быть написан программистом для использования этих объектов. В результате повышается производительность труда программиста, упрощается сопровождение готового приложения и т.д. Для среды баз данных, напротив, дополнительные интеллектуальные возможности в одних ситуациях

могут оказаться полезными, а в других — нет. Они позволяют упростить решение одних задач, но в то же время усложнить или даже сделать невозможным решение других задач.

Кстати, точно такой же довод может быть приведен в отношении дореляционных СУБД наподобие системы IMS (разработка которой началась еще в 1970-х годах). Объект отдела, содержащий набор объектов сотрудников, концептуально очень похож на иерархию системы IMS, в которой *родительские сегменты* отделов содержали подчиненные *дочерние сегменты* сотрудников. Такая иерархия весьма удобна для выполнения запросов наподобие: "Найти сотрудников, которые работают в бухгалтерии", но не очень удобна для выполнения запросов типа: "Найти отделы, в которых принимают на работу сотрудников, окончивших бизнес-колледж" (интуиция подсказывает, что описанная иерархия действительно плохо подходит для задачи последнего типа). Таким образом, многие доводы, высказанные против иерархического подхода в 1970-х годах, применимы и теперь, в контексте объектно-ориентированного подхода.

Несмотря на приведенные выше замечания, по-прежнему бытует мнение о том, что объектные системы являются большим шагом вперед на пути развития технологий управления базами данных. В частности, многие утверждают, что объектные технологии весьма перспективны для *сложных* приложений в следующих областях:

- системы автоматизированного проектирования и автоматизированного управления производством (САПР/АСУП);
- системы комплексного автоматизированного управления технологическими процессами;
- системы автоматизированной разработки программного обеспечения;
- геоинформационные системы;
- наука и медицина;
- системы хранения и выборки документов и т.д.

(Отметим, что выше перечислены те области, в которых применение традиционных продуктов SQL сопряжено со значительными трудностями.) В последние годы на эту тему опубликовано огромное количество технических статей. Кроме того, выпущено несколько соответствующих коммерческих продуктов.

В данной главе основное внимание уделяется объектной технологии в целом. Поэтому в ней необходимо представить наиболее важные концепции объектного подхода и, в частности, рассмотреть эти концепции с точки зрения *управления базами данных* (заметим, что большая часть имеющихся публикаций посвящена анализу этого вопроса в основном с точки зрения *программирования).* Данная глава имеет определенную структуру. В следующем подразделе представлен специальный пример, наглядно демонстрирующий неспособность современных реляционных продуктов удовлетворительно решать некоторые задачи, в результате чего у объектной технологии появляется шанс предоставить лучший вариант решения этих задач. Затем, в разделе 25.2, предлагается обзор основных понятий, таких как *объекты, классы, сообщения* и *методы.* В разделе 25.3 определенное внимание уделено описанию некоторых особенностей этих понятий, а также подробно обсуждается их содержание. В разделе 25.4 представлен всеобъемлющий пример применения объектной базы данных, а в разделе 25.5 обсуждаются некоторые дополнительные вопросы. Наконец, в разделе 25.6 приведено резюме.

В заключение необходимо сделать приведенные ниже замечания.

■ Вопреки тому, что объектные системы изначально предназначались для создания *сложных* приложений, таких как САПР/АСУП, в дальнейшем для краткости и простоты изложения будут рассматриваться только очень простые примеры (данные об отделах, сотрудниках и т.п.). При этом такой упрощенный подход нисколько не принижает значимости объектной технологии; во всяком случае, если объектные базы данных действительно успешно работают, они должны легко справляться и с *простыми* задачами. •:■ Следует особо подчеркнуть, что в данной главе речь идет именно о системах объектных баз данных, поэтому здесь не уделено внимание объектному программированию или объектным языкам программирования, объектному анализу и проектированию, "объектному моделированию", графическим объектным интерфейсам и т.д. А важнее всего — мы не утверждаем, что любые критические замечания, высказанные в этой главе в отношении использования объектов в контексте баз данных, остаются в силе применительно к любым другим аспектам использования объектного подхода.

Специальный пример

В данном разделе рассматривается простой пример, первоначально предложенный Стоунбрейкером (Stonebraker), доработанный автором этой книги и описанный в [25.15]. Данный пример иллюстрирует некоторые проблемы, присущие классическим продуктам SQL. База данных, которая может рассматриваться как существенно упрощенная версия базы данных САПР/АСУП, содержит сведения о *прямоугольниках,* заданных в такой системе координат, оси *X и Y* которой параллельны сторонам этих прямоугольников, т.е. все стороны прямоугольников являются либо вертикальными, либо горизонтальными. Следовательно, каждый прямоугольник может быть уникальным образом представлен с помощью координат *(x1,y1)* и *(x2,y2)* нижнего левого и верхнего правого углов, соответственно (рис. 25.1). На языке SQL это определение можно записать с помощью следующего оператора.

```
CREATE TABLE RECTANGLES
     ( X1 . . . , Y1 . . . , X2 . . . , Y2 . .
       .,..., PRIMARY KEY ( X1, Y1, X2, Y2
       ) ) ;
```

Теперь рассмотрим запрос: "Найти все прямоугольники, которые перекрывают единичный квадрат (0, 0, 1, 1) " (рис. 25.2).

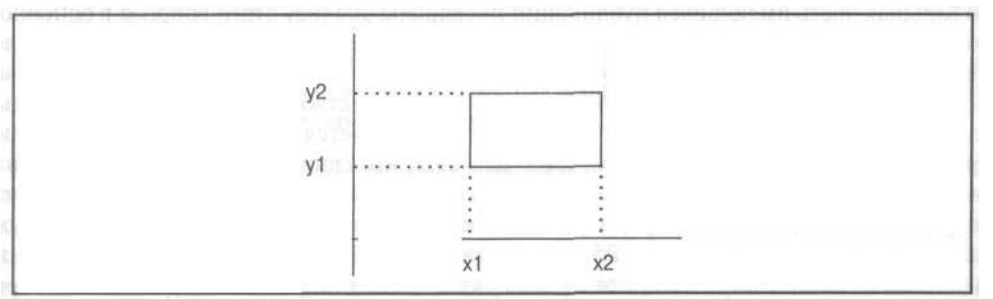


Рис. 25.1. Прямоугольник с координатами (xl, yl,x2, y2)

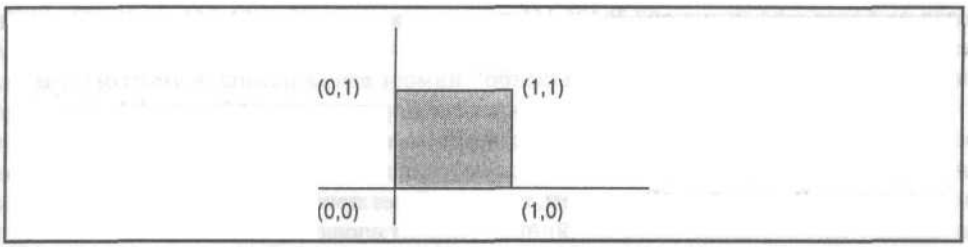


Рис. 25.2. Единичный квадрате координатами (0, 0,1,1)

"Очевидная" формулировка этого запроса на языке SQL может быть представлена следующим образом.

```
SELECT ...
FROM RECTANGLES
WHERE ( X1 >= 0 AND X1 <= 1 AND Yl >= 0 AND Yl <= 1 )
     /* нижний левый угол внутри единичного квадрата */
     OR ( X2 >= 0 AND X2 <= 1 AND Y2 >= O AND Y2 <= 1 )
     /* верхний правый угол - внутри единичного квадрата */
OR ( X1 >= 0 AND X1 <= 1 AND Y2 >= O AND Y2 <= 1 )
     /* верхний левый угол - внутри единичного квадрата */
OR ( X2 >= O AND X2 <= 1 AND Yl >= 0 AND Yl <= 1 )
     /* нижний правый угол - внутри единичного квадрата */
OR ( X1 <= O AND X2 >= 1 AND Yl <= O AND Y2 >= 1 )
     /* прямоугольник полностью включает единичный квадрат
*/ OR ( X1 <= O AND X2 >= 1 AND Yl >= 0 AND Yl <= 1 )
     /* нижний край пересекает единичный квадрат */
     OR ( X1 >= O AND X1 <= 1 AND Yl <= 0 AND Y2 >= 1 )
     /* левый край пересекает единичный квадрат */
     OR ( X2 >= O AND X2 <= 1 AND Yl <= 0 AND Y2 >= 1 )
     /* правый край пересекает единичный квадрат */
     OR ( X1 <= O AND X2 >= 1 AND Y2 >= 0 AND Y2 <= 1 ) ;
     /* верхний край пересекает единичный квадрат */
```

(Упражнение. Убедитесь в том, что эта формулировка действительно правильна.) Однако нетрудно догадаться, что данный запрос можно сформулировать и в следующей, более простой форме.

```
SELECT ...
FROM RECTANGLES
WHERE ( X1 <= 1 AND Yl <= 1
       /* нижний левый угол находится ниже и левее точки
              (1,1) */ AND
X2 >= O AND Y2 >= 0 ) ;
       /* верхний правый угол находится выше и правее
              точки (0.0) */
```

(В упр. 25.3 в конце главы предлагается убедиться, что эта формулировка также является правильной.)

Возникает вопрос, может ли системный оптимизатор преобразовать исходную длинную форму запроса в соответствующую ему краткую форму. Иначе говоря, предположим, что пользователь выражает запрос в "очевидной" (и, безусловно, неэффективной) длинной форме. Может ли оптимизатор перед выполнением запроса сократить его формулировку,

сделав ее более эффективной? В [25.15] приведено доказательство того, что это почти всегда исключено, по крайней мере, оптимизаторы современных коммерческих продуктов такой возможностью не обладают.

В любом случае, несмотря на то, что краткая формулировка "более эффективна", ее производительность может оказаться неприемлемо низкой в большинстве современных реляционных продуктов, в которых используются обычные структуры памяти, например, индексы в виде B-деревьев. (В среднем, система будет проверять 50% элементов индекса для каждой из координат X1, Y1, X2, Y2.) Иными словами, проблема заключается не только в том, что отсутствует достаточно хорошая оптимизация.

Таким образом, можно утверждать, что классические продукты SQL действительно несовершенны в некоторых отношениях. Точнее говоря, проблемы, подобные описанной выше, иллюстрируют, что в этих продуктах некоторые "простые" запросы пользователя неоправданно сложно формулируются, а выполняются с неприемлемо низкой производительностью. Именно эти соображения послужили основным побудительным мотивом для развития объектных систем.

Примечание. В следующей главе (раздел 26.1) будет приведено "эффективное" решение задачи о прямоугольниках[1].

25.2. ОБЪЕКТЫ, КЛАССЫ, МЕТОДЫ И СООБЩЕНИЯ

Ниже представлены некоторые основные термины и концепции объектного подхода, а именно — сами *объекты* (естественно), *объектные классы, методы* и *сообщения.* Там, где это возможно и уместно, данные понятия сравниваются с более знакомыми понятиями. Фактически в весьма приблизительной форме объектную терминологию вполне можно соотнести с терминологией традиционного программирования (рис. 25.3).

Объектный термин	Традиционный термин
неизменяемый объект	значение
изменяемый объект	переменная
объектный класс	тип
метод	оператор
сообщение	вызов оператора

Рис. 25.3. Объектная терминология (сводка)

Предостережение. Прежде чем перейти к подробному освещению данной темы, необходимо предупредить читателя, что впредь не следует ожидать скрупулезной точности изложения, присущей реляционной теории. Действительно, многие объектные концепции (или их опубликованные определения) выражены не очень точно, относительно формулировки их определений единства мнений не достигнуто, и разногласия возникают даже по фундаментальным вопросам (в этом можно убедиться, прочитав работы

[1] Для этого решения требуется применение типов, определяемых пользователем. Язык SQL ко времени появления на рынке первых объектных систем не поддерживал типы, определяемые пользователем; теперь такая поддержка в них предусмотрена. В действительности, язык SQL в настоящее время поддерживает несколько средств, благодаря которым он становится в определенной степени "объектно-ориентированным". Но мы намеренно отложим обсуждение этих средств до следующей главы.

[25.10], [25.39] и [25.42]). Например, не существует абстрактной, формально определенной "объектной модели данных", нет согласия даже применительно к неформальной модели. (Поэтому в данной книге термин "объектная модель" приводится в кавычках.) Также, как это ни странно, нет четкого разграничения между уровнями абстракции, **в** частности (ключевого!) разграничения между самой **моделью и ее реализацией.**

Читателя следует также предупредить о том, что, исходя из сказанного выше, определения и объяснения, предложенные в этой главе, *не* являются общепризнанными, а также *не* обязательно полностью соответствуют принципам работы всех реальных объектных систем. Действительно, почти все предлагаемые здесь определения и толкования могут быть (и, вероятно, будут) подвергнуты критике со стороны других специалистов в этой области.

Обзор объектной технологии

Вопрос: Что такое объект? *Ответ:* Все что угодно!

Основной догмат объектного подхода— "объектом является все что угодно" (или иногда говорят "все что угодно — объект **первого класса").** Одни объекты являются **неизменяемыми;** в качестве примера можно привести целые числа (например, 3, 42) и символьные строки (например, "Mozart", "Hayduke Lives!"). Другие объекты — **изменяемые;** примерами могут служить объекты, представляющие отделы и служащих, которые упоминались в начале раздела 25.1. Согласно традиционной терминологии, неизменяемые объекты соответствуют *значениям,* а изменяемые — *переменным*[2], где рассматриваемые значения и переменные могут иметь произвольную сложность (т.е. такие объекты могут содержать любое количество типов данных, имеющихся в обычных языках программирования, и конструкторов этих типов — чисел, строк, списков, массивов, стеков и т.д.).

Примечание. В некоторых системах термин *объект* употребляется только применительно к изменяемому объекту, а термин *значение,* или иногда *литерал,* — применительно к неизменяемому объекту. Даже в тех системах, в которых термин *объект* используется как в первом, так и во втором случаях, следует помнить о том, что неформально, за исключением особо оговоренных ситуаций, под ним подразумевается изменяемый объект, если явно не указано обратное.

Каждый объект имеет *тип* (в объектной терминологии — **класс).** Отдельные объекты иногда называются **экземплярами** объектов (или просто **экземплярами),** чтобы их можно было отличить от соответствующего объектного типа или класса. Обратите внимание на то, что термин *тип* здесь используется в смысле, принятом в традиционном программировании (как в главе 5), в частности, этот термин охватывает и набор *операторов* (в объектной терминологии — методов), которые могут применяться к объектам рассматриваемого типа. Операторы только чтения и операторы обновления называются, соответственно, **наблюдателями** (observer) и **модификаторами** (mutator).

Примечание. На самом деле, в некоторых объектных системах понятия типов и классов различаются. Такие системы кратко будут рассмотрены в разделе 25.3, но пока мы будем использовать эти понятия как взаимозаменяемые.

---

[2] Заметим, однако, что термин *переменная* без дополнительного уточнения в объектном контексте часто используется для обозначения, в очень узком смысле этого слова, некоторой переменной (либо локальной переменной, либо переменной экземпляра), которая содержит *идентификатор объекта* (о чем пойдет речь далее в этом разделе).

Объекты **инкапсулированы.** Это означает, что физическое представление, т.е. внутренняя структура объекта, например, объекта DEPT (Отдел), остается скрытой от пользователей. В действительности, пользователю известно только то, что объект способен выполнять некоторые операции *(методы).* Например, к объектам DEPT могут применяться методы HIRE_EMP (Нанять сотрудника), FIRE_EMP (Уволить сотрудника), CUT_BUDGET (Урезать бюджет) и т.д. *Также обратите внимание на то, что к данным объектам могут применяться ТОЛЬКО те операции, которые упомянуты среди этих методов.* Доступ к внутреннему представлению таких объектов разрешен только коду, с помощью которого эти методы реализованы, т.е., употребляя жаргон, можно сказать, что эти и только эти методы могут "взломать инкапсулированный объект"[3], но, безусловно, данный код также невидим для пользователей.

Следует отметить, что в литературе, посвященной объектному программированию, понятие *инкапсуляции* фактически трактуется во многом по-разному. Один из подходов, который представляется наиболее обоснованным, принятый в этой книге, состоит в том, что объект считается инкапсулированным тогда и только тогда, когда он является **скалярным** в том смысле, который был указан в главе 5 (это означает, что он не имеет компонентов, доступных пользователю). В связи с этим термины *инкапсулированный* и *скалярный* обозначают полностью одно и то же понятие. Поэтому следует отметить, что, согласно этому определению, некоторые объекты коллекций (см. раздел 25.3), безусловно, нельзя назвать скалярными, и поэтому инкапсулированными. В отличие от этого, некоторые авторы категорически заявляют, что все объекты инкапсулированы, а такая позиция неизбежно ведет к существенным противоречиям. Другие авторы считают, что понятие *инкапсуляции* означает не только сокрытие внутренней структуры объекта, но и физическую увязку соответствующих методов с объектом или классом рассматриваемого объекта (т.е. физическое включение методов в состав объекта). По мнению автора, в последней интерпретации не учитываются различия между моделью и реализацией; безусловно, именно эта путаница является еще одной из причин, по которой (как указано в главе 5) автор предпочитает вообще не использовать термин *инкапсуляция,* но в настоящей главе нам, безусловно, придется время от времени сталкиваться с этим термином.

Преимущество инкапсуляции состоит в том, что она позволяет изменять внутреннее представление объектов, исключая необходимость переделывать приложения, в которых используются эти объекты. Безусловно, это возможно при условии, что любое такое изменение внутреннего представления сопровождается соответствующим изменением кода, с помощью которого реализуются применяемые к объекту методы. Иначе говоря, инкапсуляция подразумевает **физическую независимость** от **данных.**

Теперь опишем инкапсуляцию в терминах независимости от данных, что имеет смысл с точки зрения баз данных, хотя в литературе по объектной технологии это понятие обычно описывается иначе. Мы определим инкапсулированные объекты как имеющие *закрытую область памяти* и *открытый интерфейс.*

---

[3] Как было указано в главах 5 и 20, автор рекомендует придерживаться более строгих требований. Это означает, что нарушать правила инкапсуляции должно быть разрешено только тем операторам, которые предписаны моделью (в частности, селекторам и операторам "THE_"); все остальные операторы должны быть реализованы в терминах этих операторов. При разработке кода необходимо соблюдать все предосторожности! Но поскольку нет согласия в части определения объектной модели, специалисты по объектным системам не пришли к единому мнению и в отношении того, какими должны быть операторы, предписанные "моделью".

- **Закрытая область памяти** состоит из **переменных экземпляра,** которые также иногда называются *членами* или *атрибутами.* Их значения представляют внутреннее состояние данного объекта. В истинно объектной системе переменные экземпляра являются полностью защищенными и скрытыми от пользователей, однако, как было сказано выше, они доступны для кода, с помощью которого реализованы методы. Здесь следует добавить, что многие объектные системы *не* являются "чистыми" в этом смысле, поскольку разрешают пользователям доступ к некоторым переменным экземпляра. К этому вопросу мы еще вернемся в следующем подразделе.
- **Открытый интерфейс** состоит из определений интерфейсов для методов данного объекта. Эти определения соответствуют тому, что в главе 20 было названо *сигнатурами спецификации.* Но, как будет показано ниже, в объектных системах обычно требуется, чтобы такие сигнатуры были связаны лишь с одним конкретным *целевым* типом или классом. А в главе 20 ни о чем подобном не было и речи, но мы не считаем, что такое требование необходимо или обязательно [3.3]. Как уже отмечалось, код, который реализует методы объекта, как и переменные экземпляра, скрыт от пользователя.

  Примечание. Точнее говоря, открытый интерфейс представляет собой часть **определения класса** рассматриваемого объекта, а не часть самого этого объекта. (Как бы то ни было, открытый интерфейс является общим для всех объектов данного класса, а не конкретным кодом в некотором отдельном объекте.) И для определения класса данный объект является его отдельным экземпляром (как элемент каталога в обычной реляционной системе). Очевидно, что открытый интерфейс для данного объекта является общим для всех объектов класса, экземпляром которого является данный объект, и поэтому имеет смысл ввести открытый интерфейс в состав класса, определяющего объект.

Методы вызываются с помощью **сообщений,** которые, по сути, являются *вызовами функций* и имеют единственный синтаксически выделенный формальный параметр — **получатель,** или **цель.** Например, ниже приводится сообщение, записанное с использованием гипотетического синтаксиса и предназначенное для передачи указания о приеме сотрудника E на работу в отдел D.

```
D HIRE_EMP  ( E )
```

Параметры D и E будут рассмотрены в подразделе "Сравнение классов экземпляров и коллекций" раздела 25.3. Получателем в этом примере является объект отдела, указанный как параметр D. Согласно синтаксису традиционных языков программирования (вернее, тех языков, в которых все фактические параметры обрабатываются одинаковым образом), это же сообщение могло быть сформулировано следующим образом[4].

```
HIRE_EMP  ( D,  E )
```

---

[4] Выделение одного фактического параметра в качестве особого может упростить выполнение системой *процесса связывания на этапе прогона,* который был описан в главе 20. Однако такой подход имеет и целый ряд недостатков [3.3]; не последним из них можно назвать то, что для программиста, разрабатывающего конкретный метод, усложняется работа по написанию кода реализации. Еще один недостаток состоит в том, что выбор целевого фактического параметра (разумеется, в том случае, если есть из чего выбирать, т.е. если имеется несколько фактических параметров) осуществляется произвольно.

Для удобства в любой объектной системе обычно заранее предусмотрен некоторый набор *встроенных* классов и методов. В частности, в ней почти всегда присутствуют такие классы, как INTEGER (с методами "=", "<", "+", "-" **и** т.д.), CHAR (с методами "=", "<", " | |", SUBSTR и т.д.), а также другие классы. Безусловно, в системе предоставляются возможности и для опытных пользователей, которые могут создать собственные классы и методы.

Переменные экземпляра

Рассмотрим более подробно концепцию переменных экземпляра, в отношении которой у специалистов все еще нет единства мнений. Как утверждалось ранее, в истинно объектной системе переменные экземпляра должны быть полностью скрыты от пользователя, но, к сожалению, большинство систем не являются действительно объектными. Поэтому на практике необходимо различать **открытые** и **закрытые** переменные экземпляра; последние действительно скрыты для внешнего мира, тогда как первые— нет.

Предположим, что имеется объектный класс *отрезков прямых* (line segment — Is). Будем считать, что каждый отрезок имеет начало (точка BEGIN) и конец (точка END) (напомним, что аналогичный пример использовался в главе 5). Чтобы "получить" значения координат этих точек для заданного сегмента *Is,* обычно используются выражения наподобие *Is.* BEGIN и *Is.*END. Таким образом, BEGIN и END — *открытые* переменные экземпляра (отметим, что по определению доступ к таким переменным должен осуществляться с помощью специального синтаксиса — обычно используется точка после имени объекта, как в нашем примере). Если физическое представление отрезков *прямых* заменить, например, комбинацией координат средней точки (MIDPOINT), длины отрезка (LENGTH) и его наклона (SLOPE), любая программа, которая содержит такие выражения, как *1s.*BEGIN и *1s.*END, перестанет работать. Иными словами, будет утрачена независимость от данных.

Обратите внимание на то, что *открытые* переменные экземпляра не являются логически необходимыми. Предположим, что для получения значений переменных экземпляра отрезка *прямой* определены методы GET_BEGIN, GET_END, GET_MIDPOINT, GET_LENGTH и GET_SLOPE. Тогда пользователь сможет "получить" значения координат начала, конца, средней точки отрезка *Is,* его длины и наклона, вызвав методы GET_BEGIN(*1s*), GET_END(*1*s*)*, GET_MIDPOINT(*1*s*)*, GET_LENGTH(*1*s) И GET_SLOPE(*1*s*)*, соответственно. Но в этом случае уже не имеет значения, каково действительное физическое представление отрезка — вполне достаточно, чтобы при каждом его изменении соответствующим образом были изменены и методы GET. Более того, не было бы ошибкой, если бы пользователю разрешалось применять выражения наподобие 1*s.*BEGIN в качестве **сокращения** для выражения вызова метода GET_BEGIN(1*s).* Обратите внимание на то, что для использования такого сокращения объект вовсе *не* обязательно должен содержать открытую переменную экземпляра BEGIN. К сожалению, в реальных системах обычно не придерживаются такого подхода. Как правило, все открытые переменные экземпляра фактически отражают физическое представление объекта (по крайней мере, частично, хотя могут существовать и некоторые дополнительные переменные экземпляра, которые являются действительно закрытыми и скрытыми). Поэтому в соответствии со сложившейся практикой будем считать, что, если не указано иное, объекты предоставляют определенные открытые переменные экземпляра, хотя это понятие логически излишне.

В связи со сказанным выше необходимо рассмотреть еще один практический случай. Предположим, что определенные фактические параметры (которые пользователь в первом приближении может считать параметрами *переменной экземпляра)* требуются для создания объектов некоторого конкретного класса[5]. Однако из этого вовсе *не* следует, что подобные *переменные экземпляра* могут применяться для любых целей. Пусть, например, для создания объекта отрезка прямой необходимы значения координат точек BEGIN и END. Но это не означает, что всегда можно будет составить запрос, например, для получения сведений обо всех отрезках прямых, которые имеют одни и те же заданные координаты точки BEGIN. Возможность выполнения такого запроса зависит от того, был ли определен подходящий для этого случая метод.

Наконец, следует отметить, что некоторые системы поддерживают особый вид закрытых переменных экземпляра, называемых **защищенными.** Если объекты класса с имеют защищенную переменную экземпляра р, то эта переменная доступна только для методов, определенных для класса с, *и* для методов, определенных для *любого подкласса* (на любом уровне) класса С. Подклассы кратко описаны в подразделе 25.3.

Идентификатор объекта

Каждый объект обладает уникальным **идентификатором объекта** (Object ID — OID). Такие примитивные *(неизменяемые)* объекты, как целое число 42, являются *самоидентифицирующимися,* т.е. они сами являются собственными идентификаторами объекта. ►Изменяемые объекты, напротив, имеют в качестве идентификаторов *адреса* (концептуальные). Эти адреса можно использовать во всей базе данных как *указатели* (концептуальные) на соответствующие объекты. Адреса объектов пользователю непосредственно не предоставляются, но они могут быть присвоены, например, переменным программы и переменным экземпляров в других объектах. Обсуждение этого вопроса будет продолжено в разделах 25.3 и 25.4.

Кстати, отметим, что иногда можно встретить утверждения о том, что с точки зрения пользователя преимущество объектных систем заключается в абсолютной идентичности двух разных объектов, т.е. они могут быть дубликатами по отношению друг к другу, но отличаться своими идентификаторами. Однако с точки зрения автора этой книги подобное утверждение обманчиво. Ибо каким же образом *пользователь* действительно сможет внешне различить оба объекта? (Более подробно этот вопрос описан в [6.3], [6.6] и особенно-в [25.17].)

25.3. ЕЩЕ РАЗ ОБ ОБЪЕКТАХ И ОБЪЕКТНЫХ КЛАССАХ

В этой главе определенные понятия, представленные в предыдущем разделе, описаны более подробно. Рассмотрим более сложный пример с двумя объектными классами: DEPT (Отдел) и EMP (Сотрудник). Предположим, что в системе уже были описаны определяемые пользователем классы MONEY (Деньги) и JOB (Работа), а класс CHAR (Символьная переменная) является встроенным. Тогда операции, необходимые для создания классов DEPT и EMP, могут выглядеть следующим образом (с использованием некоторого гипотетического синтаксиса).

---

[5] Рассматриваемые объекты обязательно должны быть изменяемыми (объясните, почему).

```
CLASS DEPT
   PUBLIC  ( DEPT#                        CHAR,
             DNAME                        CHAR,
             BUDGET                      MONEY,
             MGR                 OID ( EMP ) ,
             EMPSOID ( SET ( OID ( EMP ) ) ) )
   METHODS ( HIRE EMP( OID ( EMP ) ) <код> ,
           FIRE_EMP( OID ( EMP ) ) <код> , ... ) ... ;

CLASS EMP
   PUBLIC  ( EMP#                         CHAR,
             ENAME                        CHAR,
             SALARY                      MONEY,
             POSITION          OID ( JOB ) )
   METHODS ( ... )  ...                      ;
```

Необходимо отметить несколько приведенных ниже важных особенностей.

1. В этом примере описание отделов и сотрудников построено на основе иерархии вложения, в которой объекты EMP концептуально содержатся внутри объектов DEPT. Таким образом, объект класса DEPT содержит открытую переменную эк земпляра MGR, представляющую руководителя отдела, а также переменную EMPS, представляющую сотрудников отдела. Точнее, объекты класса DEPT содержат от крытую переменную экземпляра MGR, значение которой является указателем (т.е. идентификатором) объекта сотрудника, и переменную EMPS, значение которой является указателем на множество указателей на сотрудников. Понятие иерархии вложения в более широкой форме будет представлено ниже.
2. В данном примере в объекты класса EMP *не* была включена некоторая переменная экземпляра, содержащая идентификатор объекта отдела DEPT, или же значение номера отдела DEPT# (переменная экземпляра для *внешнего ключа).* Это решение согласуется с выбранным нами методом представления связи между отделами и сотрудниками с помощью иерархии вложения. Но это также означает, что не су ществует возможности прямого перехода от заданного объекта класса EMP к соот ветствующему ему объекту класса DEPT. Подробнее данный вопрос обсуждается в подразделе "Связи" раздела 25.5.
3. Обратите внимание на то, что определение каждого класса содержит объявления методов, которые применяются к объектам этого класса (без включения про граммного кода). Целевыми классами для подобных методов являются, безуслов но, классы, определения которых включают определение данного метода[6].

На рис. 25.4 приведено несколько примеров экземпляров объектов для определенных ранее классов DEPT и EMP. Рассмотрим объект EMP, показанный в верхней части рисунка (с идентификатором (OID) eee), который содержит перечисленные ниже компоненты.

---

[6] Отметим, что в нашем гипотетическом синтаксисе смешиваются понятия модели и реализации (хотя такая ситуация и нежелательна, но весьма типична). Кроме того, автор в другой работе ([14.12]) доказал, что пример с отделами и сотрудниками все равно плохо подходит для изучения объектных клас сов! Однако данный вопрос мы здесь обсуждать не будем, поскольку пришлось бы слишком далеко отойти от основной темы.

- Неизменяемый объект 'E001' встроенного класса CHAR в открытой переменной экземпляра EMP#.
- Неизменяемый объект ' Smith' встроенного класса CHAR в открытой переменной экземпляра ENAME.
- Неизменяемый объект '$50 000' определенного пользователем класса MONEY в открытой переменной экземпляра SALARY.
- *Идентификатор* (OID) изменяемого объекта[7] определенного пользователем класса JOB в открытой переменной экземпляра POSITION.

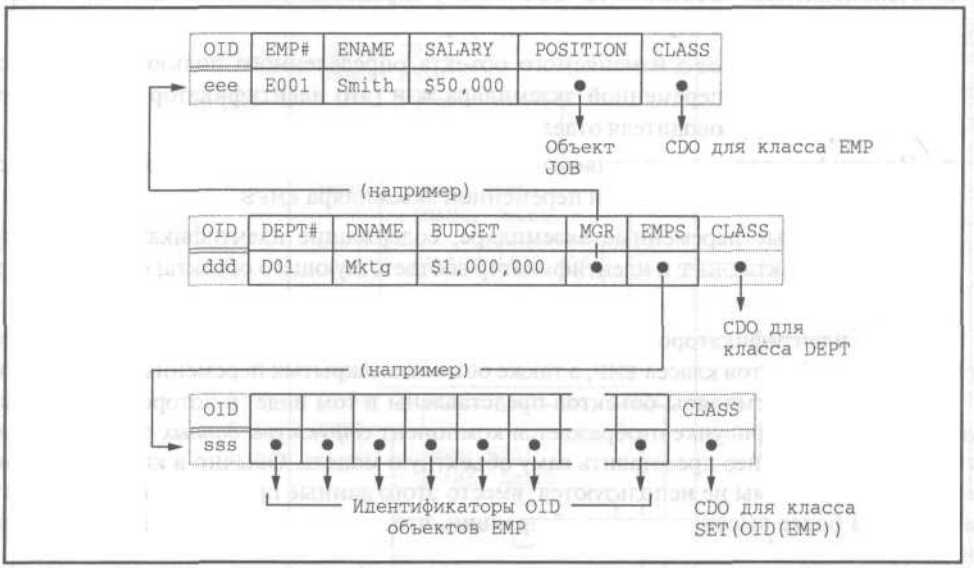


Рис. 25.4. Пример экземпляров объектов классов DEPT и EMP

Объект EMP также включает по крайней мере две дополнительные закрытые переменные экземпляра, одна из которых (OID) содержит идентификатор eee самого объекта EMP, а другая (CLASS) — идентификатор, определяющий класс объекта (Class-Defining Object — CDO) для объектов сотрудников EMP, что позволяет найти код методов данного объекта.

Примечание. Эти два идентификатора физически могут храниться как вместе с объектом, так и отдельно от него. Например, значение eee не обязательно должно храниться как часть соответствующего объекта EMP; необходимо только, чтобы в приложении был задан некоторый способ обнаружения объекта EMP по данному значению eee (т.е., чтобы было задано некоторое отображение величины eee на физический адрес объекта EMP).

Однако концептуально пользователь всегда может считать идентификатор объекта частью этого объекта.

Теперь рассмотрим объект DEPT, расположенный в центре рисунка, с идентификатором (OID) *ddd,* который содержит перечисленные ниже объекты.

- Неизменяемый объект 'D01' встроенного класса CHAR в открытой переменной экземпляра DEPT#
- Неизменяемый объект 'Mktg' встроенного класса CHAR в открытой переменной экземпляра DNAME.
- Неизменяемый объект '$1 000 000' определенного пользователем класса MONEY в открытой переменной экземпляра BUDGET.
- Идентификатор eee изменяемого объекта определенного пользователем класса EMP в открытой переменной экземпляра MGR (это идентификатор объекта, представляющего руководителя отдела).
- Идентификатор *sss* изменяемого объекта определенного пользователем класса SET (OID (EMP)) в открытой переменной экземпляра EMPS.
- Две закрытые переменные экземпляра, содержащие идентификатор (OID) *ddd* самого объекта DEPT и идентификатор соответствующего объекта, определяющего класс.

Объект с идентификатором *sss* состоит из набора идентификаторов индивидуальных (изменяемых) объектов класса EMP, а также обычных закрытых переменных экземпляра.

На рис. 25.4 экземпляры объектов представлены в том виде, в котором они реально существуют, т.е. на рисунке отображается компонент *структуры данных объектной модели,* что позволяет яснее представить саму объектную модель. Обычно в книгах или документации такие схемы не используются; вместо этого данные представляются так, как на рис. 25.5, на более высоком уровне абстракции, что, как принято считать, облегчает понимание объектной модели.

Приведенная на рис. 25.5 схема в большей степени согласуется с интерпретацией на основе иерархии вложения. Однако в ней остается скрытым тот важный факт (как уже было сказано), что в объектах часто содержатся не сами другие объекты как таковые, а скорее идентификаторы этих других объектов (т.е. указатели на эти объекты). Например, согласно рис. 25.5 можно предположить, что объект класса DEPT для отдела с номером D01 содержит *два* экземпляра объекта класса EMP для сотрудника с номером E001 (при этом, помимо всего прочего, может произойти так, что сотрудник с номером E001 получает две различные зарплаты). Такая ситуация может привести к путанице, поэтому предпочтение следует отдавать схемам, подобным приведенной на рис. 25.4.

В дополнение к этому следует отметить, что реальные определения объектных классов способствуют возникновению запутанных ситуаций, поскольку переменные экземпляра в них обычно определяются не как идентификаторы OID (как в приведенных выше выражениях с использованием гипотетического синтаксиса), а на основе непосредственного отображения иерархии вложения. Поэтому, например, переменные экземпляра EMPS в объектном классе DEPT обычно определяются не как OID (SET (OID (EMP))), а в более краткой форме: SET (EMP). Несмотря на некоторую громоздкость полной записи, мы все же предпочитаем наш стиль, как более ясный и точный.

Следует отметить, что вся прежняя критика иерархического подхода (например, иерархии вложения, воплощенной в СУБД IMS) в основном относилась именно к иерархиям вложения. Подробное рассмотрение данного вопроса заняло бы слишком много места, поэтому достаточно сказать, что основным доводом такой критики было **отсутствие симметрии.** В частности, иерархии не совсем удобны для представления отношений типа "многие ко многим". Например, для рассматриваемого ранее отношения поставщиков и деталей может возникнуть вопрос, содержатся ли объекты-поставщики в объектах-деталях или *наоборот.* А может, верно и то и другое? А что можно сказать об отношениях поставщиков, деталей и проектов?

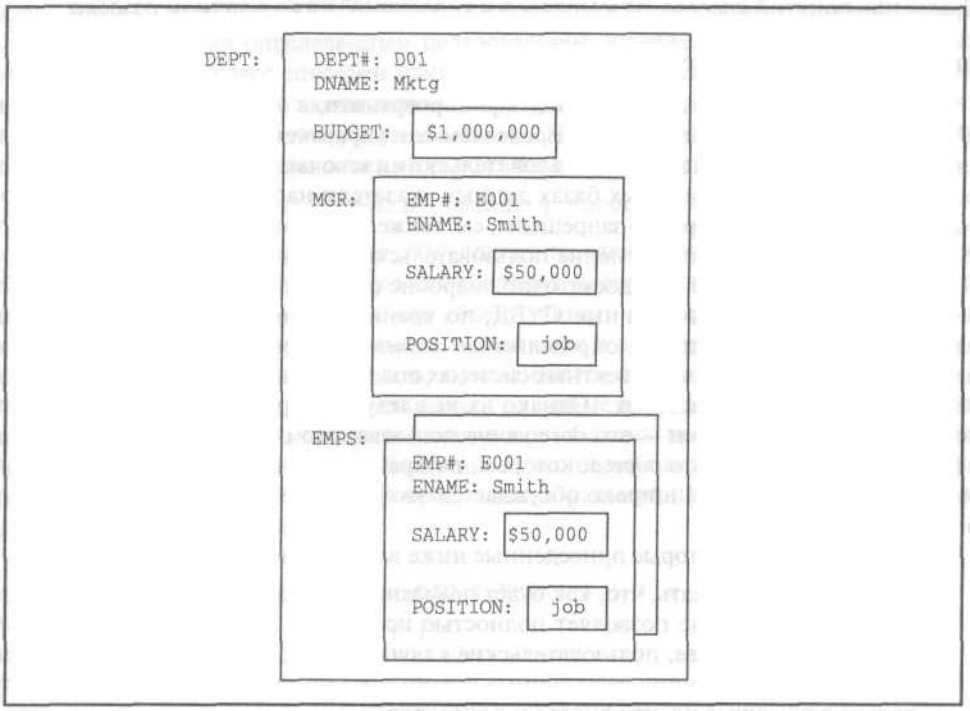


Рис. 25.5. Пример представления экземпляров объектов DEPT и EMP в соответствии с иерархией их вложения

Действительно, подобных вопросов больше, чем можно было бы предположить. С одной стороны, как мы убедились, объекты представляют собой *иерархию,* и к ним также относятся привычные критические замечания относительно иерархий. А с другой стороны, как явствует из рис. 25.4, такие объекты на самом деле образуют не иерархию, а **кортежи** с перечисленными ниже компонентами.

1. Неизменяемые *подобъекты,* т.е. самоидентифицируемые значения, такие как целые числа и денежные суммы.
2. Идентификаторы изменяемых *подобъектов,* т.е. указатели на другие (возможно, совместно используемые) изменяемые объекты.

3. Множества, списки, массивы и т.д. объектов, перечисленных в пп. 1, 2 и в этом пункте.

К этому списку следует также добавить определенные скрытые компоненты. Особого внимания заслуживает п. 3, поскольку обычно объектные системы поддерживают несколько **генераторов типа** коллекций (SET, LIST, ARRAY и др.), но обычно не поддерживают тип RELATION (отношение). Такие генераторы могут использоваться в сколь угодно сложных сочетаниях. Например, в определенных обстоятельствах как один объект может рассматриваться массив списков мультимножеств массивов указателей на целочисленные переменные. Дополнительные сведения по этой теме приведены в подразделе "Сравнение понятий классов, экземпляров и коллекций" ниже в данном разделе.

Еще раз об идентификаторе объекта

Чтобы сослаться на объект или идентифицировать его, в современных реляционных СУБД обычно используются ключи, определяемые и управляемые пользователями (далее для краткости будем называть их пользовательскими ключами). (Хотя как было описано в главах 1 и 3, в реляционных базах данных указатели наподобие идентификаторов объектов фактически намеренно запрещены; см. также продолжение этой темы в главе 26.) Но хорошо известно, что применение пользовательских ключей сопряжено с некоторыми проблемами. Эти проблемы достаточно подробно рассматриваются в [14.11] и [14.21], где сделан вывод, что реляционные СУБД, по крайней мере, в качестве альтернативы должны поддерживать и ключи, определяемые *системой {суррогатные).* Доводы в пользу идентификаторов объектов в объектных системах подобны доводам в пользу суррогатных ключей в реляционных системах. (Однако их не следует приравнивать друг к другу, поскольку суррогатные ключи — это *доступные* пользователю обычные значения, а идентификаторы объектов — это *адреса,* которые, по крайней мере, концептуально, от пользователя скрыты. В [25.17] широко обсуждается это различие и другие связанные с ним вопросы.)

Из этого следуют некоторые приведенные ниже важные выводы.

1. Необходимо понимать, что, как будет показано в разделе 25.4, наличие идентифи каторов объектов не позволяет полностью исключить применение пользователь ских ключей. Точнее, пользовательские ключи необходимы для общения с внеш ним миром, хотя внутри базы данных все объекты могут ссылаться один на другой только с помощью идентификаторов объектов.
2. Что является идентификатором *производного* объекта, например *соединения* неко торого объекта класса EMP И соответствующего объекта класса DEPT или *проекции* объекта класса DEPT ПО атрибутам BUDGET И MGR? ЭТО очень важный вопрос для *производных объектов,* рассмотрение которого мы отложим до раздела 25.5.
3. Идентификаторы объектов часто служат предметом критических замечаний, вы званных тем, что объектные системы выглядят, как "модифицированный стан дарт CODASYL". Как было описано в главе 1, стандарт CODASYL использовал ся для сетевых систем управления базами данных (например для СУБД IDMS), которые были созданы до появления реляционного подхода. Использование идентификаторов объектов приводит к низкоуровневому стилю программиро вания (см. раздел 25.4), что очень напоминает устаревший стиль программирова ния, основанный на использовании указателей, согласно стандарту CODASYL.

Кроме того, поскольку идентификаторы объектов являются указателями, часто можно встретить следующие утверждения.

- Во-первых, системы типа CODASYL ближе к объектным системам, чем реляционные.
- Во-вторых, реляционные системы основаны на *значениях,* в то время как объектные системы — на *идентификаторах.*

Сравнение понятий классов, экземпляров и коллекций

В области объектных систем четко разделяются концепции *класса, экземпляра* и *коллекции.* Как уже отмечалось, **класс** является по существу типом данных, причем он может быть встроенным или определенным пользователем, а также может быть сколь угодно сложным[8]. Каждый класс способен принять сообщение **NEW,** которое приводит к созданию нового (изменяемого) **экземпляра** объекта данного класса; вызываемый этим сообщением метод называют *конструктором класса.* Ниже приводится пример такого сообщения, записанного с помощью некоторого гипотетического синтаксиса.

```
E := EMP NEW ( 'E001', 'Smith', MONEY ( 50000 ), POS ) ;
```

Здесь программная переменная POS содержит идентификатор некоторого объекта класса JOB, а метод NEW вызывается для создания нового экземпляра класса EMP; в результате такого вызова создается новый объект данного класса, происходит инициализации переменных экземпляра заданными значениями и возвращается идентификатор нового объекта. Затем этот идентификатор присваивается программной переменной E.

Поскольку одни объекты могут ссылаться на любые другие объекты с помощью идентификаторов, один объект может *совместно* использоваться несколькими другими объектами. В частности, один и тот же объект может одновременно относиться к нескольким различным **коллекциям** объектов. В продолжение рассматриваемого примера приведем еще одно следующее выражение.

```
CLASS EMP_COLL
   PUBLIC ( EMPS OID ( SET ( OID ( EMP ) ) ) ) ...;
ALL_EMPS := EMP_COLL NEW (
) ; ALL_EMPS ADD ( E ) ;
```

Пояснения

- Объект класса EMP_COLL содержит одну открытую переменную экземпляра EMPS, значением которой является указатель (идентификатор) на изменяемый объект, значение которого представляет собой множество указателей (идентификаторов) на отдельные объекты класса EMP.
- ALL_EMPS — это программная переменная, значением которой является идентификатор объекта класса EMP_COLL. После выполнения операции присваивания

[8] Как уже было сказано в разделе 25.2, в некоторых системах используются оба понятия — и "тип", и "класс". В этих системах "тип" означает *тип* понятия, т.е. его *содержание* или *сущность,* а "класс" означает *применение* этого понятия, т.е. определенную *коллекцию,* или иногда *реализацию* (рассматриваемого типа). В других системах данные термины используются иначе... Мы же по-прежнему будем употреблять для обозначения типа термин "класс", в том же смысле, что и в главе 5.

она будет содержать идентификатор объекта, значением которого, в свою очередь, будет идентификатор *пустого* множества идентификаторов объектов EMP.

- ADD — это метод объектов класса EMP_COLL. В рассматриваемом примере данный метод применяется к объекту класса, идентификатор которого содержится в программной переменной ALL_EMPS, и предназначается для добавления идентификатора объекта EMP, который содержится в программной переменной E, к множеству идентификаторов (изначально пустому). Причем идентификатор этого объекта содержится в объекте EMP_COLL, идентификатор которого находится в программной переменной ALL_EMPS.

Рассмотрев приведенную выше последовательность операций, можно заметить, что переменная ALL_EMPS обозначает коллекцию объектов EMP, которая в настоящее время содержит только один объект, а именно — объект EMP, описывающий сотрудника с номером E001. Кроме всего прочего, обратите внимание на то, что необходимо упомянуть значение пользовательского ключа в последнем предложении!

Безусловно, в некоторый заданный момент может существовать любое количество разных и, возможно, перекрывающихся *множеств объектов сотрудников,* например, как показано ниже.

```
PROGRAMMERS := EMP COLL NEW ( )
; PROGRAMMERS ADD ( E ) ;

HIGHLY PAID := EMP COLL NEW ( )
; HIGHLY_PAID ADD ( E ) ;
```

Приведенный ниже пример выражения SQL показывает, что в реляционных системах все организовано совсем по-другому.

```
CREATE TABLE EMP
     ( EMP#      ...     NOT NULL ,
       ENAME     ...     NOT NULL ,
       SALARY    ...     NOT NULL ,
       POSITION  ...     NOT NULL ) ... ;
```

В этом- случае с помощью оператора SQL одновременно создаются *и* тип, *и* коллекция, причем сложный тип соответствует заголовку таблицы, а исходно пустая коллекция соответствует телу таблицы. Точно так же приведенное ниже выражение SQL позволяет одновременно *и* создать отдельную строку EMP, *и* добавить ее к коллекции EMP (для упрощения предполагается, что этот оператор INSERT действительно вставляет только единственную строку).

```
INSERT INTO EMP (...) VALUES ( ... ) ;
```

Можно сделать вывод, что организация этих действий с помощью языка SQL характеризуется приведенными ниже особенностями.

1. Отсутствует возможность обеспечить существование отдельного *объекта* EMP без включения его в состав некоторой *коллекции;* при этом фактически имеется только одна и только одна такая *коллекция* (но необходимо также учитывать замечания, приведенные ниже, и обратить внимание на то, что строку EMP не в полной мере можно считать *объектом,* как показано в главе 26).

2. Не существует непосредственного способа создания двух различных *коллекций* объектов EMP одного и того же *класса* (подробности приведены ниже).
3. Не существует непосредственного способа совместного использования одного **и** того же *объекта* в нескольких *коллекциях объектов* EMP (подробности **приведены** ниже).

Иногда можно услышать именно такие замечания, однако они не выдерживают серьезной критики. Во-первых, для достижения аналогичного эффекта в каждом случае может применяться реляционный метод использования внешнего ключа. Например, можно определить две базовые таблицы, PROGRAMMERS (Программисты) и HIGHLY_PAID (Высокооплачиваемые), каждая из которых состоит из номеров соответствующих сотрудников. Во-вторых, что еще важнее, для достижения того же эффекта может быть применен реляционный метод с использованием **представлений.** Например, таблицы PROGRAMMERS и HIGHLY_PAID можно определить как следующие представления, созданные на основе базовой таблицы EMP.

```
CREATE VIEW PROGRAMMERS
    AS SELECT EMP#, ENAME, SALARY,
       POSITION FROM   EMP WHERE
       POSITION = 'Programmer' ;

CREATE VIEW HIGHLY PAID
    AS SELECT EMP#, ENAME, SALARY, POSITION FROM
       EMP WHERE SALARY > некоторое предельное
       значение ;
```

Теперь один и тот **же** *объект* EMP **можно** вполне разместить одновременно в двух или нескольких *коллекциях.* Кроме того, принадлежность к этим коллекциям, которые являются представлениями, контролируется системой автоматически, а не программистом вручную.

Завершая эту тему, следует упомянуть об одной интересной параллели, существующей между изменяемыми объектами объектных систем и **явно заданными динамическими переменными** некоторых языков программирования (например, переменными типа BASED языка PL/I). Как и в случае изменяемых объектов данного класса, может существовать любое количество отдельных явных динамических переменных данного типа, память для которых выделяется во время выполнения в результате явно определяемых в программе действий. Более того, эти отдельные переменные, как и отдельные изменяемые объекты, *не имеют имен,* поэтому на них необходимо ссылаться с помощью указателей. Например, в языке **PL/I** можно записать такую последовательность выражений.

```
DCL XYZ  INTEGER BASED ;   /* XYZ - переменная типа BASED          */
DCL P POINTER ,-           /*  P — переменная-указатель            */
ALLOCATE XYZ SET ( P );    /* создать новый экземпляр XYZ,         */
                           /* и присвоить переменной P значение, которое
                           */
                           /* указывает на этот экземпляр          */
P -> XYZ = 3  ;            /* присвоить значение 3 экземпляру      */
                           /* XYZ, на который указывает P          */
```

Этот записанный на языке **PL/I** код очень похож на рассмотренный ранее объектный код. В частности, объявление переменной типа BASED подобно созданию класса объектов, а применение операции ALLOCATE — созданию нового экземпляра объекта этого

класса с помощью сообщения NEW. Таким образом, основная причина, по которой в объектной модели необходимы идентификаторы, заключается в том, что те объекты, которые они идентифицируют, не обладают уникальными именами (точно так же, как экземпляры переменных типа BASED в языке PL/I).

Иерархии классов

Описание основных концепций объектного подхода было бы неполным без рассмотрения **иерархий классов** (их не следует путать с иерархиями вложения). Однако объектная концепция *иерархии классов,* в сущности, является аналогичной концепции иерархии типов, которая подробно рассматривалась в главе 20. Следовательно, здесь можно ограничиться несколькими краткими определениями (большей частью это — перефразированные определения из главы 20) и соответствующими выводами.

Примечание. Напомним, что в объектном мире, как и в остальных случаях, все еще нет полной согласованности относительно концепции абстрактной *модели наследования,* поэтому различные системы наследования существенно отличаются одна от другой на более низком уровне детализации.

Начнем с того, что объектный класс Y называется **подклассом** класса X, а объектный класс X — **суперклассом** объектного класса y тогда и только тогда, когда каждый объект класса Y обязательно является также объектом класса *X* (т.е. "y **ISA** X"). В этом случае объект класса y наследует[9] переменные экземпляра и методы класса x. Наследование переменных экземпляра обычно называют наследованием **структуры,** а наследование методов — наследованием **поведения. В** истинно объектных системах не может быть наследования структуры, а возможно лишь наследование поведения (по крайней мере, это касается скаляров или полностью инкапсулированных объектов), поскольку отсутствует структура, которая может быть унаследована (под этим подразумевается структура, доступная пользователю). Но на практике объектные системы обычно не являются такими "чистыми" и в какой-то мере поддерживают наследование структуры, которое, подчеркиваем, означает наследование открытых переменных экземпляра.

Примечание. Если подкласс содержит дополнительные открытые переменные экземпляра, кроме тех, которые он наследует от (непосредственно предшествующему ему) суперкласса, то принято применять выражение, что этот *подкласс расширяет свой суперкласс.*

Если класс У является подклассом класса X, то пользователю предоставляется возможность применять объект класса У вместо объекта класса X везде, где это допустимо, т.е. в качестве фактического параметра различных методов. Этот принцип называется **принципом заменяемости** и весьма полезен для **повторного использования кода.** Однако, поскольку в объектных системах не всегда четко прослеживается разница между значениями и переменными (т.е. между неизменяемыми и изменяемыми объектами), возникают сложности при проведении различий между заменяемостью значений и переменных (этот вопрос дополнительно обсуждается в главе 20). Средство, позволяющее применять один и тот же метод для объектов класса **x** и класса y, называется **полиморфизмом.**

---

[9] Он, по-видимому, унаследует также закрытые переменные экземпляра, но автор рассматривает такое наследование как проблему реализации, а не как составную часть модели.

Обычно в объектных системах заранее предусмотрены определенные встроенные иерархии классов. Например, в системе OPAL (см. раздел 25.4) каждый класс рассматривается как подкласс некоторого уровня встроенного класса OBJECT (поскольку "все является объектами"). Встроенные подклассы класса OBJECT включают классы BOOLEAN, CHAR, INTEGER, COLLECTION и др. Класс COLLECTION **в** свою очередь включает подкласс BAG, а класс BAG содержит класс SET И Т.Д. И Т.П. (НО, по-видимому, COLLECTION, BAG и SET не являются классами как таковыми, а "генераторами классов" наподобие RELATION в языке Tutorial D? Создается впечатление, что в этом вопросе есть некоторая путаница.)

Еще одно важное замечание состоит в том, что объектные системы обычно не допускают изменения класса объекта (см. аннотацию к [20.12]). Вследствие этого объектные системы не поддерживают уточнение или обобщение с помощью ограничений, поэтому такие системы не способны поддерживать то, что, по мнению автора, можно считать "качественной" моделью наследования. Эта тема раскрыта более подробно в следующей главе (в разделе 26.3).

Наконец, в некоторых системах, кроме одиночного наследования, в той или иной форме поддерживается **множественное** наследование. Но автору не известно ни одной системы, которая поддерживала бы наследование кортежей или отношений (как одиночное, так и множественное) **в** том смысле, который указан в [3.3].

25.4. ВСЕОБЪЕМЛЮЩИЙ ПРИМЕР

В предыдущей главе были представлены базовые концепции объектного подхода. В данной главе на исчерпывающем примере последовательно демонстрируется применение этих идей на практике, а именно: здесь показано, как определяется объектная база данных, как она пополняется данными и как в ней выполняются операции выборки и обновления данных. В рассматриваемом примере используются объектная СУБД GemStone (разработка корпорации GemStone Systems) и ее язык запросов OPAL [25.13]. Язык OPAL, в свою очередь, основан на языке Smalltalk [25.23].

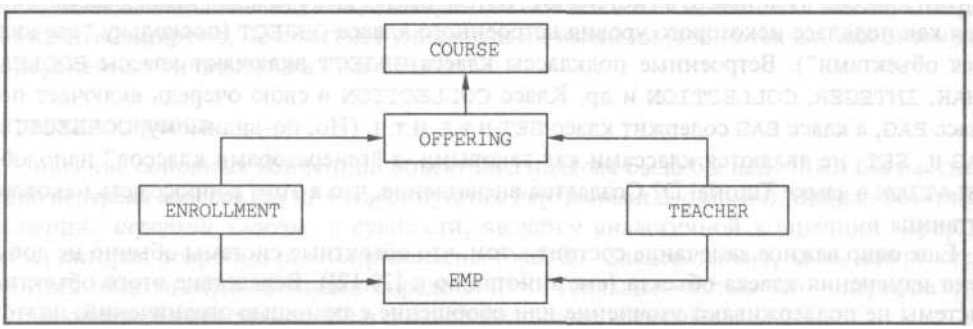
Примечание. Язык Smalltalk относится к числу "наиболее ранних и чистых воплощений" объектного подхода, поэтому он используется здесь в качестве примера, но ради справедливости следует отметить, что в программных продуктах и приложениях он все чаще заменяется языком C++, а в последнее время— языком Java.

В качестве примера воспользуемся упрощенной версией базы данных профессиональной подготовки из упр. 9.7 главы 9. В этой базе данных содержится информация о схеме подготовки и обучения специалистов внутри некоторой компании. Для каждого курса обучения (COURSE) в базе данных содержится описание отдельных потоков, организованных для его изучения (OFFERING). Для каждого потока хранятся данные обо всех его слушателях (ENROLLMENT) и преподавателях (TEACHER). Кроме того, в базе содержатся сведения о сотрудниках компании EMP. Реляционную версию базы данных можно описать следующим образом.

```
COURSE     { COURSE*, TITLE }
OFFERING   { COURSES, OFF#, OFFDATE, LOCATION }
TEACHER    { COURSES, OFF#, EMP# }
ENROLLMENT { COURSE*, OFF#, EMP#, GRADE }
EMP        { EMP#, ENAME, SALARY, POSITION }
```

Рис. 25.6. Схема связей для образовательной базы данных

На рис. 25.6 показана схема связей для рассматриваемой базы данных.



Определение данных

Теперь перейдем к определению данных на языке OPAL для образовательной базы данных. Ниже приводится первое определение объектного класса EMP, описывающего сотрудников (для удобства строки этого определения пронумерованы).

```
1   OBJECT SUBCLASS : 'EMP'
2      INSTVARNAMES : #[ 'EMP#', 'ENAME', 'POSITION' ]
3      CONSTRAINTS  : #[ #[ #EMP#, STRING ] ,

4                        [ #ENAME, STRING ],
5                        [ #POSITION, STRING ] ] .
```

Пояснение. В строке 1 определен объектный класс EMP как подкласс встроенного класса OBJECT. (Согласно терминологии языка OPAL, в строке 1 *передается сообщение* объекту OBJECT с запросом вызвать метод SUBCLASS; в этом вызове метода заданы фактические параметры INSTVARNAMES и CONSTRAINTS. Для определения нового класса, как и всего прочего, в языке OPAL необходимо отправить сообщение объекту.) В строке 2 указано, что объекты класса EMP имеют, соответственно, три закрытые переменные экземпляра — EMP#, ENAME и POSITION, а в строках 3-5 на эти переменные экземпляра накладываются ограничения, указывающие, что они должны содержать объекты класса STRING.

Примечание. В настоящей главе игнорируются чисто синтаксические подробности, несущественные для преследуемых нами целей (в частности, то, что вездесущие знаки "#", применяемые в данном примере, не допускаются по условиям синтаксиса).

Подчеркнем, что переменные экземпляра EMP#, ENAME и POSITION— *закрытые* переменные для класса EMP, поэтому доступ к ним по именам допустим только в коде реализации методов этого класса. В качестве примера ниже даны определения методов "получить и установить", т.е. методов, позволяющих выбрать и обновить номера служащих (здесь символ " ^" можно читать как "возвратить").

```
METHOD  :  EMP
      GET_EMP#
      ^EMP#
  %
  METHOD  :  EMP
```

```
   SET EMP# : EMP# PARM
       EMP# :=
EMP#_PARM %
```

В следующем подразделе о методах мы поговорим подробнее. А сейчас рассмотрим определение класса COURSE.

```
1   OBJECT SUBCLASS : 'COURSE'
2      INSTVARNAMES : #[ 'COURSE*', 'TITLE', 'OFFERINGS' ]
3      CONSTRAINTS  : #[ #[ #COURSE#, STRING ] ,

4                        [ #TITLE, STRING ] ,
5                        [ SUFFERINGS, OSET ] ] .
```

Пояснение. **В** строке 5 определена закрытая переменная экземпляра OFFERINGS, которая содержит **идентификатор** объекта класса OSET (этот класс будет определен несколько позже). Выражаясь неформально, переменная OFFERINGS обозначает множество всех потоков для данного курса. Иначе говоря, связь "курс-поток" моделируется с помощью иерархии вложения, в которой потоки концептуально содержатся внутри соответствующего курса. Определение класса потока OFFERING может быть записано следующим образом.

```
1   OBJECT SUBCLASS : 'OFFERING'
2      INSTVARNAMES : #[ 'OFF*', 'ODATE', 'LOCATION',
3                              ' ENROLLMENTS ' , ' TEACHERS ' ]
4      CONSTRAINTS  : #[ #[ #OFF#, STRING ] ,
5                        [#ODATE, DATETIME ],
6                        [#LOCATION, STRING ],
7                        [«ENROLLMENTS, NSET ],
8                        [#TEACHERS, TSET ] ] .
```

Пояснение. В строке 7 определяется закрытая переменная экземпляра ENROLLMENTS, содержащая идентификатор объекта класса NSET. Говоря неформально, переменная ENROLLMENTS обозначает множество всех слушателей в данном потоке. Аналогичным образом, переменная TEACHERS обозначает множество всех преподавателей рассматриваемого потока. Поэтому здесь вновь используется представление иерархии вложения. Определения классов NSET и TSET будут даны ниже. Определение класса слушателей ENROLLMENT может быть записано следующим образом.

```
1   OBJECT SUBCLASS : 'ENROLLMENT'
2      INSTVARNAMES : # [ ' EMP' , ' GRADE' ]
3      CONSTRAINTS  : #[ #[ #EMP, EMP ] ,
4                        [ #GRADE, STRING ] ].
```

Пояснение. В строке 3 определяется закрытая переменная экземпляра EMP, содержащая идентификатор объекта класса EMP, представляющий отдельного сотрудника, который является слушателем курса.

Примечание. Чтобы продолжить создание иерархии вложения, объект EMP помещается "знутрь" соответствующего объекта ENROLLMENT. Однако здесь можно заметить асимметрию: зачисление сотрудников на обучение в разные потоки описывается как отношение типа "многие ко многим", но участники этой связи, сотрудники и потоки, трактуются совершенно по-разному.

Наконец, рассмотрим объекты, представляющие преподавателей. Здесь мы немного отойдем от оригинальной реляционной версии базы данных и будем рассматривать преподавателей (TEACHER) как подкласс класса сотрудников (EMP).

```
1   EMP  SUBCLASS : 'TEACHER'
2        INSTVARNAMES   : #[ 'COURSES' ]
3        CONSTRAINTS     : #[ #[ #COURSES, CSET ] ] .
```

Пояснение. В строке 1 определен объектный класс TEACHER, который является подклассом ранее определенного класса EMP (иными словами, класс TEACHER соединен связью ISA с классом EMP). Таким образом, каждый отдельный объект TEACHER имеет закрытые переменные экземпляра EMP#, ENAME и POSITION (которые унаследованы[10] от класса EMP), а также переменную COURSES, которая содержит идентификатор объекта класса CSET. Объект CSET обозначает множество всех курсов, которые может вести данный преподаватель. Каждый объект TEACHER также наследует все методы класса EMP.

Как уже отмечалось, в приведенных выше определениях классов предполагалось существование нескольких классов коллекций ESET, CSET, OSET, NSET и TSET. Ниже даются определения всех этих классов, начиная с класса ESET.

```
1   SET SUBCLASS      : 'ESET'
2       CONSTRAINTS   : EMP .
```

Пояснение. В строке 1 дается определение объектного класса ESET, который является подклассом встроенного класса SET. В строке 2 на объекты класса ESET накладывается ограничение: они должны быть множествами идентификаторов объектов класса EMP. В общем случае может существовать произвольное количество объектов класса ESET, но в данной ситуации будет создан только один объект (подробности приводятся в следующем подразделе), который будет представлять собой множество идентификаторов *всех* объектов класса EMP, которые в настоящее время существуют в базе данных. Выражаясь неформально, этот единственный объект ESET можно рассматривать как объектный аналог базовой переменной отношения EMP в реляционной версии базы данных.

Определения классов CSET, OSET, NSET и TSET аналогичны (они приводятся ниже). Однако для каждого из них придется создать не один, а несколько объектов соответствующей коллекции классов. Например, в нашем случае будет существовать столько коллекций объектов OSET, сколько существует отдельных объектов COURSE.

```
SET SUBCLASS    :      'CSET'
    CONSTRAINTS :      COURSE .

SET SUBCLASS    :      'OSET'
    CONSTRAINTS :      OFFERING .

SET SUBCLASS    :      'NSET'
    CONSTRAINTS :      ENROLLMENT .

SET SUBCLASS    :      'TSET'
    CONSTRAINTS :TEACHER .
```

[10] Заметим, что здесь наследуется *закрытое* представление (т.е. физическая реализация).

Заполнение базы данных

Теперь опишем, как можно поместить в рассматриваемую базу данных требуемую информацию. При этом остановимся на пяти основных классах объектов (EMP, COURSE и т.д.). Начнем с сотрудников. Напомним о нашем намерении собрать вместе идентификаторы всех объектов EMP в единственном объекте ESET. Таким образом, прежде всего необходимо создать объект ESET, как показано ниже.

```
OID_OF_SET_OF_ALL_EMPS  :=  ESET NEW  .
```

Выражение в правой части этого оператора присваивания возвращает объектный идентификатор (OID) нового пустого экземпляра объекта класса ESET (т.е. пустое множество идентификаторов объектов класса EMP), а затем идентификатор этого нового экземпляра присваивается программной переменной OID_OF_SET_OF_ALL_EMPS. Говоря очень *неформально,* эта переменная обозначает "множество всех сотрудников".

Теперь каждый раз при создании нового объекта класса EMP идентификатор этого объекта следует помещать в объект ESET, идентификатор которого хранится в переменной OID_OF_SET_OF_ALL_EMPS. Поэтому для создания объекта класса EMP и вставки его идентификатора в объект класса ESET необходимо определить специальный *метод.* (Еще один вариант состоит в том, что для выполнения такой же задачи может быть написана прикладная программа.) Код указанного метода приведен ниже.

```
1 METHOD : ESET                          " анонимный! "
2   ADD EMP#  : EMP# PARM                " формальные параметры "
3   ADD ENAME : ENAME PARM
4   ADD POS   : POS PARM
5    | EMP OID |                          " локальная переменная "
6   EMP OID := EMP NEW .                 " новый сотрудник "
7   EMP_OID SET_EMP#   : EMP#_PARM ;     " инициализация "

8           SET ENAME  : ENAME PARM ;
9           SET POS    : POS PARM .
10  SELF ADD: EMP OID .                  " вставка "
11 %
```

Пояснение

и В строке 1 начинается запись кода данного метода (который завершается символом "%" в строке 11), который применяется к объектам класса ESET. (На самом деле, в системе во время выполнения программы будет существовать *один и только один* объект класса ESET.)

- В строках 2-4 определены три параметра с внешними именами ADD_EMP#, ADD_ENAME и ADD_POS. Эти имена будут использованы в сообщениях, вызывающих данный метод. Соответствующие внутренние имена EMP#_PARM, ENAME_PARM и POS_PARM будут применяться только внутри кода реализации данного метода.
- В строке 5 определена локальная переменная EMP_OID, а в строке 6 ей присвоен идентификатор нового неинициализированного экземпляра объекта класса EMP.
- В строках 7—9 передается сообщение новому объекту класса EMP с указанием трех вызываемых методов (SET_EMP#, SET_ENAME и SET_POS) и передачей одного фактического параметра каждому из них (EMP#_PARM для SET_EMP#, ENAME_PARM для SET_ENAME И POS_PARM для SET_POS).

Примечание. Здесь предполагается, что методы SET_ENAME и SET_POS (как и метод SET_EMP#, показанный выше) также должны быть определены заранее.

- В строке 10 передается сообщение объекту SELF. Это имя объекта является специальным обозначением, представляющим во время выполнения тот текущий объект, в котором определен указанный метод (т.е. сообщение передается самому текущему *целевому* объекту). Сообщение вызывает применение встроенного метода ADD к этому объекту (метод ADD предусмотрен во всех классах, определяющих *коллекции)*. В результате идентификатор объекта, который содержится в локальной переменной EMP_OID, будет вставлен в объект, идентифицируемый значением переменной SELF (в данном случае это будет объект ESET, содержащий идентификаторы всех существующих на текущий момент объектов класса EMP).
  Примечание. Переменная SELF необходима потому, что параметр, соответствующий объекту-получателю, не имеет собственного имени (см.строку 1).
- Обратите внимание на то, что, как отмечено в комментарии к строке 1, определенный здесь метод остается анонимным. В общем случае в языке OPAL методы не имеют имен; вместо этого для них используется *сигнатура* (в языке OPAL *сигнатура* определяется как комбинация имени класса, в котором он применяется, и внешних имен их формальных параметров). Вполне очевидно, что применение такого соглашения может привести к созданию неуклюжих и огромных конструкций. Можно отметить и еще один недостаток: если два метода применяются к одному и тому классу и имеют одинаковые формальные параметры, то в этих двух методах указанным параметрам должны быть присвоены произвольные, но разные внешние имена.

Теперь создан метод для вставки в базу данных новых объектов EMP, но до сих пор такая операция еще не была выполнена. Поэтому ниже приведен пример вставки в базу данных сведений о сотрудниках.

```
OID_OF_SET_OF_ALL_EMPS ADD EMP#  : 'E009'
                       ADD ENAME : 'Helms'
                       ADD POS   : 'Janitor'
                       .
```

При использовании приведенного выражения будет создан объект класса EMP для сотрудника с номером E009, а идентификатор этого объекта будет добавлен к множеству уже существующих идентификаторов объектов класса EMP.

Обратите внимание на то, что встроенный метод NEW никогда не должен использоваться для класса EMP, кроме тех случаев, когда он является частью только что определенного метода. Иначе могут быть созданы некоторые объекты класса EMP с оборванными связями, которые не будут представлены в объекте ESET, содержащем идентификаторы всех существующих объектов EMP.

Примечание. Следует принести извинения за слишком частое повторение таких громоздких выражений, как "только что определенный метод" и "объект ESET, содержащий идентификаторы всех существующих объектов EMP", но иначе нелегко точно описать предметы, не имеющие названий.

Объекты для сотрудников представляют собой наиболее простой случай, поскольку они соответствуют *обычным сущностям* (согласно терминологии модели типа "сущность/связь") и не содержат никаких других внедренных в них объектов (не считая неизменяемых).

Далее следует рассмотреть более сложный случай объектов для *курсов,* которые (несмотря на то, что они все еще остаются *обычными сущностями)* концептуально содержат другие включенные в них изменяемые объекты. В целом, чтобы создать объекты для курсов, необходимо выполнить описанные ниже действия.

1. Применить метод NEW к классу CSET в целях создания исходно пустого *множества всех курсов* (на самом деле — множества идентификаторов объектов класса COURSE).
2. Определить метод для создания нового объекта класса COURSE и вставить его идентификатор в *множество всех курсов.* Этот метод принимает в качестве факти ческих параметров указанные значения COURSE# и TITLE И создает новый объект COURSE с этими заданными значениями. Кроме того, с помощью этого метода к классу OSET применяется метод NEW для создания первоначально пустого множе ства идентификаторов потоков OFFERING, а затем идентификатор этого пустого множества идентификаторов потоков помещается в переменную OFERINGS внут ри нового объекта класса COURSE.
3. Вызвать только что определенный метод для каждого отдельного курса.

Теперь создадим объекты потоков. Для этого необходимо выполнить перечисленные ниже действия.

1. Определить метод для создания нового объекта класса OFFERING. Этот метод должен принимать в качестве фактических параметров значения переменных OFF#, ODATE и LOCATION и приводить к созданию нового объекта класса OFFERING с указанными параметрами. Кроме того, потребуется выполнить неко торые дополнительные действия.
   - Для создания исходно пустого множества идентификаторов слушателей ENROLLMENT следует применить метод NEW к классу NSET, а затем поместить идентификатор этого пустого множества слушателей в переменную ENROLLMENTS внутри нового объекта класса OFFERING.
   - Для создания исходно пустого множества идентификаторов преподавателей TEACHER следует применить метод NEW к классу NSET, а затем поместить иден тификатор этого пустого множества преподавателей в переменную TEACHERS внутри нового объекта класса OFFERING.
2. Данный метод принимает в качестве фактического параметра значение COURSE*, после чего значение COURSE# используется в нем для выполнения описанных ни же действий.
   - Выполнить поиск[11] соответствующего объекта COURSE для нового объекта OFFERING (рекомендации по выполнению этого действия приведены в сле дующем подразделе).

" Безусловно, данный метод должен предотвращать попытки создать новый объект потока, если не может быть найден соответствующий курс. Здесь и в дальнейшем описании такие исключительные ситуации подробно не рассматриваются.

- Найти таким образом *множество всех потоков* для данного объекта класса COURSE.
- Добавить идентификатор нового объекта класса OFFERING К соответствующе му *множеству всех потоков.*

Отметим, что (как уже упоминалось в этой главе) здесь идентификаторы не позволяют избежать необходимости применять таких пользовательские ключи, как COURSE*. Действительно, такие ключи необходимы не только для идентификации объектов во внешнем мире, но и для использования в качестве основы при выполнении определенных операций поиска внутри самой базы данных.

Наконец, следует вызвать только что определенный метод для каждого отдельного потока.

Обратите внимание на то, что (в соответствии с используемым представлением иерархии вложения) здесь не было создано *множество всех потоков.* Одним из следствий этого упущения становится то, что в любом запросе, областью действия которого становится это множество (например: "Определить все потоки, организованные в Нью-Йорке"), придется использовать определенный объем процедурного кода, позволяющего исправить эту ситуацию (см. следующий подраздел).

Теперь рассмотрим процедуру создания объектов слушателей. Эти объекты (ENROLLMENT) отличаются от объектов потоков тем, что содержат переменную экземпляра EMP, значение которой является идентификатором соответствующего объекта EMP. Поэтому последовательность действий должна быть такой, как описано ниже.

1. Определить метод для создания нового объекта класса ENROLLMENT. Этот метод в качестве фактических параметров принимает указанные значения COURSE*, OFF*, EMP# и GRADE и создает новый объект ENROLLMENT с заданным значением GRADE. Кроме того, требуется выполнить некоторые дополнительные действия.
   - Использовать значения COURSE# и OFF# для поиска соответствующего объекта OFFERING ДЛЯ НОВОГО объекта ENROLLMENT.
   - Найти *множество всех слушателей* для данного объекта класса OFFERING.
   - Добавить идентификатор нового объекта класса ENROLLMENT к соответствую щему *множеству всех слушателей.*
   - Кроме того, потребуется выполнить описанные ниже действия
   - Использовать значение EMP*#для поиска соответствующего объекта EMP.
   - Поместить идентификатор объекта EMP в переменную EMP внутри нового объ екта класса ENROLLMENT.
2. Вызывать только что определенный метод для каждого отдельного слушателя по очереди.

Наконец, перейдем к созданию объектов преподавателей. Различие между способами создания объектов для преподавателей и потоков заключается в том, что класс TEACHER является подклассом класса EMP. Ниже приведена последовательность действий, которые необходимо выполнить в данном случае.

1. Определить метод создания нового объекта класса TEACHER. Этот метод принима ет в качестве фактических параметров заданные значения COURSE#, OFF# и EMP#. Потребуются также некоторые дополнительные действия.
   - Использовать значение EMP# для поиска соответствующего объекта класса EMP.
   - Заменить наиболее определенный класс этого объекта EMP классом TEACHER, поскольку данный сотрудник теперь является также преподавателем (но способ указанного изменения класса в значительной степени зависит от конкретной рассматриваемой системы, поэтому дополнительные сведения на эту тему здесь не приведены). Кроме того, должны быть выполнены описанные ниже действия.
   - Использовать значения COURSE# и OFF# для поиска соответствующего объекта OFFERING ДЛЯ НОВОГО объекта TEACHER.
   - Найти *множество всех преподавателей* для этого объекта OFFERING.
   - Добавить идентификатор нового объекта класса TEACHER к соответствующему *множеству всех преподавателей.*

2. Определить множество всех курсов, которые может вести данный преподаватель, а также задать соответствующим образом значение переменной COURSES в новом объекте класса TEACHER. Но здесь эти подробности не приведены.
3. Вызывать только что определенный метод для каждого объекта, описывающего отдельного преподавателя.

Операции выборки

Прежде чем приступить к подробному описанию операций выборки, следует отметить (хотя это и вполне очевидно), что язык OPAL, как и другие объектные языки в целом, функционирует по принципу последовательной обработки отдельных записей (или, по меньшей мере, отдельных объектов), а не их множеств. Поэтому для решения большинства проблем программист вынужден применять процедурный код. Рассмотрим лишь один пример — запрос: "Определить все потоки для курса с номером C001, которые организованы в городе New York". Для простоты предположим, что есть переменная OOSOAC, значение которой является идентификатором *множества всех курсов.* Ниже приведен код для такого запроса.

```
1 | COURSE C001 , C001 OFFS , C001 NY OFFS |
2   COURSE C001
3       := OOSOAC DETECT : [ :CX | ( CX GET COURSE# ) = 'C001' ] .
4   C001 OFFS
5       := COURSE C001 GET OFFERINGS .
6   C001 NY OFFS
7       := C001 OFFS SELECT :
8                  [ :OX | ( OX GET_LOCATION ) = 'New York' ] .
9 ^ C001_NY_OFFS .
```

Пояснение

1. В строке 1 объявлены три локальные переменные: COURSE_C001 (которая будет использована для хранения идентификатора объекта курса с номером C001), C001_OFFS (которая будет использована для хранения идентификатора объекта

множества всех потоков для курса с номером C001) и C001_NY_OFFS (которая будет использована для хранения идентификатора множества идентификаторов для требуемых потоков, т.е. потоков, организованных в Нью-Йорке).

2. В строках 2 и 3 передается сообщение объекту (коллекции), обозначенному переменной OOSOAC. Это сообщение вызывает применение встроенного метода **DETECT** к этой коллекции. Формальным параметром метода **DETECT** является выражение в следующей форме.

   ```
   [ :x | p(x) ]
   ```

   Здесь *p(x)* — логическое выражение, включающее переменную *x,* а *x,* по сути, представляет собой переменную области значений, которая принимает свои значения среди элементов коллекции, к которым применяется метод DETECT (в рассматриваемом примере этот метод применяется к множеству объектов COURSE). В результате выполнения метода DETECT возвращается идентификатор первого найденного объекта *x* этого множества, для которого выражение p *(x)* принимает значение TRUE (в рассматриваемом примере это — экземпляр объекта класса COURSE для курса с номером C001)[12]. Затем идентификатор этого объекта класса COURSE присваивается переменной COURSE_C001.

   Примечание. Фактически также возможно передать методу DETECT *замаскированный* формальный параметр, чтобы можно было учесть тот случая, в котором выражение p *(x)* никогда не принимает значение TRUE. Но подробные сведения об этом здесь не приведены.

3. В строках 4 и 5 переменной C001_OFFS присваивается идентификатор *множества всех потоков для* курса с номером C001.
4. Строки 6-8 подобны строкам 2 и 3, поскольку встроенный метод **SELECT** подобен методу DETECT, за исключением того, что он возвращает идентификатор множества идентификаторов *всех* объектов *x* (а не просто первый обнаруженный из таких объектов), для которых выражение p *(x)* принимает значение TRUE. В рассматриваемом примере в результате выполнения этого метода переменной C001_NY_OFFS присваивается идентификатор множества идентификаторов для тех потоков курса с номером C001, которые организованы в городе New York.
5. В строке 9 этот идентификатор возвращается в точку вызова.

Следует обратить внимание на некоторые перечисленные ниже особенности.

- Логическое выражение p *(x)* в методах SELECT и DETECT может содержать (в самом сложном случае) лишь некоторое количество простых скалярных операторов сравнения, которые соединяются с помощью операторов AND, т.е. сложность условия поиска ограничена.
- Круглые скобки, окружающие выражение фактического параметра методов SELECT и DETECT, можно заменить фигурными скобками. При использовании фигурных скобок в языке OPAL будет предпринята попытка использовать индекс

[12] В этом примере подразумевается, что такие методы, как GET_COURCE# (аналог метода GET_EMP#, описанного выше в данном разделе), уже определены.

(если соответствующий индекс существует) в ходе применения данного метода, а при использовании круглых скобок индекс использоваться не будет.

- Условие, согласно которому метод DETECT возвращает идентификатор *первого найденного* объекта, для которого значением выражения p *(x)* является TRUE, оз начает, что найденный объект будет первым найденным при использовании про извольной последовательности поиска, которая будет выбрана в языке OPAL для просмотра множества (поскольку множества не имеет присущих им свойств упо рядочения). В рассматриваемом примере результат не зависит от организации по иска, поскольку *первый* объект, для которого логическое выражение принимает значение TRUE, фактически является *единственным* таким объектом.
- Внимательный читатель непременно заметит широко применяемые выражения наподобие "метод DETECT", хотя, как отмечалось выше, методы в языке OPAL не имеют имен. Действительно, DETECT и SELECT не являются именами методов (а потому выражения наподобие "метод DETECT", строго говоря, является непра вильным). Они, скорее всего, служат именами внешних параметров для некоторых встроенных (и анонимных) методов. Но для краткости и простоты далее в качестве имен методов по-прежнему будут использоваться названия DETECT и SELECT (а также другие подобные им названия для других методов).
- Кроме того, внимательный читатель заметит, что довольно часто используется вы ражение "метод NEW". В данном случае его не следует считать неправильным — методы, не принимающие иных формальных параметров, кроме обязательных це левых формальных параметров, являются исключением из общего правила в язы ке OPAL, согласно которому методы должны быть анонимными.

Операции обновления

Объектный аналог операции вставки INSERT уже обсуждался в предыдущем подразделе, а объектные аналоги операций обновления UPDATE и удаления DELETE рассматриваются ниже.

- *Обновление.* Операции обновления выполняются так же, как операции выборки, но вместо методов GET_ используются методы SET_.
- *Удаление.* Для удаления объектов используется встроенный метод REMOVE. Точнее, он используется для удаления идентификатора указанного объекта из указанной коллекции. Если на данный объект больше не имеется никаких ссылок, т.е. к нему вовсе не может быть осуществлен доступ, то в языке OPAL он автоматически уда ляется системным процессом сбора мусора. Ниже приводится пример реализации операции "удаления сотрудника с номером E001 из множества всех сотрудников".

```
E001    := OID OF SET OF ALL EMPS
             DETECT :  [ : EX |  (EX GET EMP# ) =
'E001' ] . OID_OF_SET_OF_ALL_EMPS REMOVE : E001 .
```

Но как в такой ситуации реализовать правило каскадного удаления ON DELETE CASCADE? Например, как при удалении некоторого сотрудника одновременно удалить и сведения обо всех потоках, в которых он проходит обучение? Для этого, естественно, придется создать соответствующую процедуру.

Можно было бы прийти к заключению, что механизм реализации удаления с помощью процесса сбора мусора является всего лишь некоторой разновидностью реализации правила ограничения удаления ON DELETE RESTRICT, поскольку объект не удаляется до тех пор, пока на него существует хотя бы одна ссылка. Но в данном случае это не совсем так. Например, объекты потоков (OFFERING) не содержат идентификаторов соответствующего объекта курса (COURSE), поэтому потоки не накладывают *ограничений* на удаление объектов курсов. (В иерархиях вложения неявно подразумевается некоторая разновидность правила каскадного удаления ON DELETE CASCADE, кроме случаев, когда пользователь выполняет следующее: либо включает идентификатор родительского объекта в дочерний объект; либо включает идентификатор дочернего объекта в какой-то другой объект, хранящийся в базе данных. А в таких ситуациях интерпретация на основе *иерархии вложения* больше не имеет никакого смысла. В следующем разделе этот вопрос будет рассмотрен при обсуждении *обратных переменных.)*

В заключение отметим, что операция удаления REMOVE может быть использована для эмуляции реляционной операции DROP, например, для удаления всего класса ENROLLMENT. Подробности оставляем читателю в качестве упражнения.

25.5. ДОПОЛНИТЕЛЬНЫЕ АСПЕКТЫ

В этом разделе обсуждаются некоторые перечисленные ниже традиционные аспекты управления базами данных, но в объектном контексте.

- Произвольные запросы и связанные с этим проблемы.
- Целостность базы данных.
- Реализация связей.
- Языки программирования баз данных.
- Повышение производительности.
- Возможность рассматривать объектную СУБД как обычную СУБД.

Произвольные запросы и связанные с этим проблемы

До сих пор мы намеренно не подчеркивали, что если для манипулирования объектами заданы только заранее определенные методы, то *произвольные* запросы (не предусмотренные этими методами) просто невозможны, если только классы и методы не разработаны в соответствии со специальными правилами. Например, если для объекта класса DEPT определены только методы HIRE_EMP (нанять сотрудника), FIRE_EMP (уволить сотрудника) и CUT_BUDGET (урезать бюджет) (как описано в разделе 25.2), то невозможно будет выполнить даже такой простой запрос: "Кто является руководителем отдела программирования?".

По существу, по той же причине невозможно определить представления и декларативные ограничения целостности для объектов, опять же, если не следовать некоторым конкретным правилам.

По мнению автора, решением этих проблем (т.е. определением *специальных правил,* о которых речь шла выше) могут служить описанные ниже рекомендации.

1. Определить множество операторов ("операторов THE_"), с помощью которых можно было бы получить доступ к *некоторым возможным представлениям* рассмат риваемых объектов, как описано в главе 5.
2. Надлежащим образом внедрить объекты в реляционную структуру. Эта часть ре шения подробно обсуждается в следующей главе.

Но разработчики объектных систем обычно *не* придерживаются таких рекомендаций (вернее, не совсем их придерживаются). Вместо этого они поступают, как описано ниже[13].

1. Обычно определяются операторы, которые предоставляют доступ не к некоторым *возможным* представлениям, а скорее к физическим представлениям (см. описа ние открытых переменных экземпляра в разделе 25.2). Приведем цитату из [25.31]: "В настоящее время для всех продуктов объектных СУБД требуется, чтобы [переменные экземпляра], которые упоминаются в... запросах, были открытыми".
2. Обычно поддерживается не реляционная структура, а множество других структур, которые основаны на мультимножествах, массивах и других структурах. В связи с этим напомним приведенное выше утверждение, что классы, т.е. типы в сочетании отношениями, необходимы и достаточны на логическом уровне (см. главу 3). А поскольку речь идет фактически об основной модели, то, по мнению автора, массивы и прочие структуры являются ненужными и нежелательными. На его взгляд, причиной того, что в объектных системах отдано предпочтение коллекци ям, а не отношениям (фактически отношения почти полностью отвергаются), снова является путаница между понятиями модели и реализации.

В связи с выполнением произвольных запросов возникает еще один важный вопрос, а именно: какой класс является результатом? Предположим, например, что необходимо выполнить запрос: "Определить имена и заработную плату всех служащих из отдела программирования" по базе данных отделов и служащих из раздела 25.3. Предположительно результат будет содержать (открытые) переменные экземпляра ENAME и SALARY. Однако в базе данных нет класса, который имеет такую структуру. Нужно ли предварительно определять такой класс, прежде чем выполнять запрос? (Обратите внимание на последствия: если бы было необходимо определить такой класс, то для класса с n переменными экземпляра, потребовалось бы иметь по крайней мере 2^n таких заранее определенных классов только для поддержки операций выборки!) А если есть какой-либо класс результатов, то какие методы к нему применимы?

Аналогичные вопросы возникают в связи с операциями соединения (эти операции еще более наглядно иллюстрируют приведенное выше замечание, но ими все проблемы не исчерпываются). Например, если выполняется соединение объектов отдела и служащих, то какой класс получится в результате? Какие методы нужно использовать?

Возможно, из-за того, что на такие вопросы нелегко ответить, опираясь на чисто объектную структуру, в некоторых объектных системах поддерживаются операции *отслеживания пути* (см. [25.38]), а не просто операции соединения как таковые. Для базы данных OPAL из раздела 25.4, например, допустимо следующее выражение пути.

---

[13] Здесь подразумевается, что рассматриваемая объектная система действительно поддерживает *произвольные* запросы, как и большинство современных объектных систем. Однако более ранние объектные системы иногда не поддерживали такие запросы, частично в силу причин, которые будут рассмотрены в этом разделе.

```
ENROLLMENT . OFFERING . COURSE
```

Оно означает[14] следующее: "Получить доступ к уникальному объекту COURSE, на который указывает уникальный объект OFFERING, указанный с помощью заданного объекта ENROLLMENT". Реляционный аналог этого выражения обычно включает две операции соединения и одну операцию проекции. Иными словами, в результате отслеживания пути предоставляется доступ только по предварительно определенным путям (фактически, как в дореляционных системах) и только к объектам предварительно определенных классов (опять же, как в дореляционных системах).

Целостность базы данных

В главе 9 утверждалось, что целостность данных в базе имеет абсолютно фундаментальное значение. Тем не менее, даже те объектные системы, которые поддерживают произвольные запросы, обычно не поддерживают декларативные ограничения целостности. Выполнение подобных ограничений достигается с помощью процедурного кода, т.е. методов или, возможно, прикладных программ. Рассмотрим, например, следующее ограничение (или "бизнес-правило") из раздела 9.1: "Ни один поставщик со статусом меньше 20 не может поставлять какие-либо детали в количестве больше 500". Для того чтобы обеспечить выполнение этого ограничения, процедурный код его поддержки должен быть включен, по меньшей мере, во все перечисленные ниже методы:

- метод для создан ия объекта поставки;
- метод для изменения количества поставляемых деталей;
- метод для изменения статуса поставщика;
- метод для переназначения некоторой поставки другому поставщику.

При внимательном изучении этого примера можно отметить приведенные ниже важные особенности.

1. При такой организации явно утрачивается возможность использовать саму систему для определения того, когда должна выполняться проверка целостности.
2. Как убедиться в том, что все необходимый методы содержат необходимый для поддержки целостности базы данных код?
3. Как исключить для пользователя возможность (например) обойти метод "создать поставку" и непосредственно воспользоваться встроенным методом NEW класса объектов поставки, тем самым исключив проверку целостности?
4. Как при изменении ограничений целостности найти и внести соответствующие изменения во все методы, в которых они были определены?
5. Как гарантировать правильность кода, с помощью которого поддерживаются ог раничения целостности?
6. Как поступить, если ограничения необходимо ввести лишь на время, а затем от менить?

---

[14] На самом деле это выражение *не* является допустимым путем для базы данных, в том виде, как мы ее определили, поскольку указатели определяют неверное направление! Например, объекты класса OFFERING не ссылаются на объекты класса COURSE; наоборот, объекты класса COURSE ссылаются на них.

7. Как передать системе запрос для поиска всех ограничений, которые применяются к указанному объекту или к комбинации объектов?
8. Будут ли ограничения целостности приводиться в действие во время загрузки и при выполнении других служебных функций?
9. Можно ли осуществить семантическую оптимизацию (т.е. упростить запросы с помощью ограничений целостности, как описано в главе 18)?
10. Как отражаются все перечисленные выше недостатки на производительность работы во время создания приложения и при последующем его сопровождении?

Реализация связей

Термин *связи* используется в объектно-ориентированных продуктах и в соответствующей литературе в основном для связей, представленных в реляционной системе *внешними* ключами. Обычно предоставляется также особая поддержка для ограничений целостности специального вида. В качестве примера снова рассмотрим базу данных отделов и сотрудников. В обычной реляционной системе сотрудники имели бы внешний ключ, который ссылался бы на отделы, и этим можно было бы ограничиться. В объектных же системах, напротив, имеются по крайней мере три перечисленные ниже возможности.

1. Каждый объект сотрудников может содержать указатель (идентификатор), кото рый ссылается на соответствующие объекты отделов. Такая возможность анало гична реляционному подходу, но *не* идентична ему, поскольку внешние ключи и идентификаторы — это не одно и то же.
2. Каждый объект отдела может включать множество указателей на соответствующие объекты сотрудников. Эта возможность аналогична подходу, основанному на ие рархии вложения и описанному в разделе 25.3.
3. Кроме того, указанные выше подходы могут объединяться следующим образом.

```
CLASS EMP ...
    ( ... DEPT OID ( DEPT ) INVERSE DEPT.EMPS )
... CLASS DEPT ...
    ( ... EMPS
         OID( SET ( OID ( EMP ) ) ) INVERSE EMP.DEPT ) ... ;
```

Отметим, что ключевое слово INVERSE ОТНОСИТСЯ к двум переменным экземпляра: EMP.DEPT и DEPT.EMPS. Говорят, что эти две переменные экземпляра являются обратными по отношению друг к другу. Переменная EMP . DEPT — это *ссылочная* переменная экземпляра, а DEPT. EMPS — переменная *набора ссылок* экземпляра. (Если бы обе переменные были переменными набора ссылок, то связь имела бы тип "многие ко многим", а не "один ко многим".)

Безусловно, для каждой из указанных выше возможностей требуется определенного вида поддержка ссылочной целостности. Вопросы ссылочной целостности обсуждаются немного позже. Но прежде всего необходимо ответить на вполне очевидный вопрос: "Как в объектных системах обрабатываются связи, в которых участвуют больше двух классов, например, поставщиков, деталей и проектов?". Может показаться, что наилучшим (т.е. наиболее симметричным) ответом на этот вопрос будет создание объектного

класса SPJ. Причем каждый объект класса будет обладать связями, реализованными с помощью *обратных переменных,* с соответствующим поставщиком, соответствующей деталью и соответствующим проектом. Но в таком случае, если создание нового объектного класса на основе трех объектов является наилучшим подходом к описанию *связей* со степенью больше двух, то возникает очевидный вопрос: почему таким же образом не реализуются связи со степенью два?

Кроме того, что касается именно обратных переменных, то почему необходимо вводить асимметрию, направленность и два разных имени для одной и той же конструкции? Например, рассмотрим два следующих реляционных выражения.

```
SP.P# WHERE SP.S# = S#('S1')
SP.S# WHERE SP.P# = P#('P1')
```

Их объектные аналоги выглядят следующим образом.

```
S.PARTS.P# WHERE S.S# =
S#('S1') P.SUPPS.S# WHERE
P.P# = P#('P1')
```

Здесь использован некий гипотетический синтаксис, выбранный специально таким образом, чтобы избежать несущественных в данном случае различий (например, использования двух разных имен связей PARTS и SUPPS) и предотвратить создание двусмысленных выражений.

Ссылочная целостность

Рассмотрим уже упомянутую ранее объектную поддержку ссылочной целостности, которая, кстати, как часто утверждают, является сильной стороной объектных систем. Уровни такой поддержки могут быть самыми разными; например, ниже приводится классификация таких уровней, предложенная в работе Каттелла (Cattell) [25.10].

- *Отсутствие системной поддержки.* Поддержка ссылочной целостности возлагается на код, написанный пользователем (кстати, как это было определено в первоначальной версии стандарта языка SQL).
- *Проверка указателей.* В системе выполняется проверка того, относятся ли все указатели к существующим объектам правильного типа. Но удаление объектов может быть не разрешено (вместо этого, как в языке OPAL, может быть предусмотрено уничтожение объектов, на которые нет больше ссылок, с помощью *сбора мусора).* Как уже объяснялось в разделе 25.4, этот уровень поддержки приблизительно эквивалентен (но лишь весьма приблизительно) правилу каскадного удаления ON DELETE CASCADE в иерархии для подчиненных объектов без возможности сиимеместного доступа и контролируемого удаления (ON DELETE RESTRICT) для других объектов (но только если "указатели указывают на правильный объект").
- *Системная поддержка.* На этом уровне отслеживание и обновление ссылок происходит в системе автоматически (например, с помощью установки значения nil для ссылок на удаленные объекты). Этот уровень в некоторой степени подобен правилу удаления ON DELETE SET NULL.
- *"Определяемая пользователем семантика".* Правило каскадного удаления ON DELETE CASCADE (применяемое за пределами иерархии вложения) может рассматриваться как пример "определяемой *пользователем* семантики". Такие возможности обычно

не поддерживаются объектными системами непосредственно; скорее, они должны быть реализованы **в** коде, написанном пользователем.

Языки программирования баз данных

Приведенные в разделе 25.4 примеры на языке OPAL демонстрируют, что, в отличие от большинства современных реляционных программных продуктов (на основе языка SQL), в объектных системах обычно не используется *встроенный подъязык данных.* Вместо этого для операций, выполняемых как с базами данных, так и с другими объектами, используется один и тот же **интегрированный язык.** Согласно принятой в главе 2 терминологии базовый язык и язык, ориентированный на работу с базами данных, в объектных системах *тесно связаны* (фактически эти два языка представляют собой один и тот же язык).

Такой подход, безусловно, обладает определенными преимуществами (именно поэтому он принят в языке Tutorial D). Одно из самых существенных — возможность осуществления усовершенствованной проверки типов [25.2]. В приведенной ниже цитате из [25.38] отмечено еще одно важное достоинство.

> *"В едином унифицированном языке нет несоответствия типов данных между процедурным языком программирования и встроенным языком манипулирования данными с декларативной семантикой ".*

Термин *несоответствие типов данных* (impedance mismatch) используется для описания различий между типичными современными языками программирования, функционирующими на основе последовательной обработки записей, и реляционными языками наподобие SQL, функционирующими на основе последовательной обработки множеств. Очевидно, что такие различия на практике приводят к возникновению разнообразных проблем при использовании программных продуктов SQL. Но для их решения *не* нужно переводить язык, ориентированный на работу с базами данных, на уровень последовательной обработки отдельных записей (как это принято в объектных системах!). Необходимо вместо этого в языках программирования ввести инструменты для перехода к последовательной обработке множеств записей. Применение последовательной обработки отдельных записей в объектных языках (т.е. процедурный подход) отбрасывает нас к временам, когда использовались такие дореляционные системы, как IMS и IDMS.

В отношении последнего замечания следует отметить, что в действительности большинство существующих объектных языков является либо процедурными, либо языками третьего поколения (3CL). В результате утрачиваются все преимущества реляционного подхода на основе последовательной обработки множеств. В частности, заметно уменьшается способность системы к оптимизации запросов пользователя, а значит, как и в дореляционных системах, производительность в значительной степени зависит от подхода, выбранного самим пользователем (прикладным программистом и/или администратором базы данных). Об этом более подробно сказано в следующем подразделе.

Повышение производительности

Низкая производительность — один из самых существенных недостатков всех объектных систем. Снова приведем цитату из [25.10]: "Различие производительности на порядок реально может привести к возникновению *функциональных* различий, поскольку для решения некоторых задач будет невозможно использовать данную систему, если ее

производительность гораздо ниже требуемого уровня" (автор согласен с этим замечанием, которое он лишь немного перефразировал).

Среди многочисленных факторов, влияющих на общую производительность системы, можно отметить перечисленные ниже[15].

- *Кластеризация.* Как указано в главе 18, физическая кластеризация логически взаимосвязанных данных, размещенных на жестком диске, является одним из наиболее важных факторов повышения производительности системы. В объект ных системах логическая информация из определений базы данных (относительно иерархии классов, иерархии вложения или других явно заданных связей между объектами) обычно основана на модели системы и используется в качестве при близительного плана для физической кластеризации данных. Кроме того, часто рекомендуется, чтобы администратор базы данных сам осуществлял явное и непо средственное управление отображением концептуального уровня на внутренний (по терминологии главы 2).
- *Кэширование.* Объектные системы обычно предназначены для использования в среде "клиент/сервер", в которой пользователи *копируют* на свои рабочие стан ции информацию из базы данных на сервере и хранят ее на этих рабочих станциях в течение некоторого времени. Очевидно, что в такой системе было бы полезно кэшировать логически связанные данные на рабочем месте клиента.
- *Подстановка.* Термин *подстановка указателей* (swizzling) означает процесс замены указателей, организованный по принципу замены идентификаторов объектов (которые обычно представляют собой логические адреса на диске), адресами опе ративной памяти при считывании объектов в оперативную память (и, безусловно, выполнение обратной операции, когда объекты записываются в базу данных). Преимущества такого метода для приложений, в которых обрабатываются доста точно *сложные объекты,* очевидны, и поэтому необходимо часто осуществлять по иск указателей.
- *Выполнение методов на сервере.* В качестве примера рассмотрим запрос: "Найти все книги, в которых содержится больше 20 глав". В традиционной системе SQL книги могут быть представлены в виде объектов типа CLOB или BLOB (см. гла ву 15), и в клиентском приложении может потребоваться выполнить выборку каж дой книги по очереди и просмотреть ее для определения того, имеется ли в ней больше 20 глав. В отличие от этого, в объектной системе оператор "определения количества глав" мог быть выполнен на сервере и клиенту переданы только те книги, которые фактически требуются. Поэтому выполнение методов на сервере указанным образом способствует снижению издержек, связанных с передачей данных[16].

---

[5] Кроме перечисленных факторов, можно отметить, что в объектных системах повышение произво дительности (в той степени, в которой это повышение действительно происходит), достигается за счет "приближения пользователя к физическому уровню", т.е. предоставления ему доступа к указателям и другим средствам, которые должны быть скрыты в реализации.

[6] На самом деле в этом утверждении допущено чрезмерное упрощение. Выполнение методов, для которых требуется интенсивный обмен данными (наподобие метода "подсчета количества глав"), целе сообразнее организовать на сервере; однако другие методы (например, которые выводят большой объем данных на дисплей), возможно, лучше выполнять на компьютере клиента.

Примечание. Указанное выше на самом деле служит доводом не в пользу объектов, а в пользу *хранимых процедур* (см. главу 21). Традиционная система SQL с хранимыми процедурами будет обладать в этом случае такими же преимуществами с точки зрения производительности, как и объектная система с методами.

В [25.12] обсуждается эталонный тест 001 для измерения производительности системы на основе базы данных, содержащей информацию о счетах-фактурах. Такой эталонный тест предусматривает выполнение описанных ниже действий.

1. Случайная выборка 1000 деталей с применением заданного пользователем метода для каждой детали.
2. Случайная вставка 1000 деталей с присоединением каждой к трем другим.
3. Случайное разузлование деталей (вплоть до седьмого уровня) с применением за данного пользователем метода для каждой детали, подсчитывающего соответст вующие вхождения в узлы.

Согласно данным, опубликованным в [25.12], производительность некоторой (не сказано, какой) объектной системы на два порядка выше производительности некоторой (не сказано, какой) современной системы SQL, особенно при условии, что кэш уже заполнен данными (так называемый *теплый* доступ). Однако в той же работе [25.12] приведено следующее справедливое утверждение.

> *"Это различие...* не *следует приписывать различию между собственно реляционной и объектной моделями... В основном эти различия обусловлены [особенностями реализации]".*

Это утверждение может быть подкреплено тем фактом, что различия в производительности становились заметно меньше по мере увеличения размера базы данных (когда в кэш нельзя было поместить все данные из базы).

Аналогичный, но более полный эталонный тест, OO7, описан в [25.9].

Действительно ли можно рассматривать объектную СУБД как обычную СУБД

Примечание. В данном подразделе излагаются суждения и наблюдения, которые в основном заимствованы из [25.16]. В этой работе, помимо всего прочего, сказано, что различия между объектными и реляционными системами гораздо существеннее, чем это обычно представляется. Ниже приведена цитата из этой работы.

> *Объектные базы данных возникли благодаря желанию отдельных программистов объектных приложений (продиктованному самыми разными причинами) хранить созданные ими специализированные объекты в постоянной памяти. При этом под такой* ***постоянной памятью*** *могла подразумеваться и база данных, но, что особо следует подчеркнуть, база данных,* ***предназначенная для конкретных приложений;*** *она не была разделяемой и не являлась базой данных общего назначения, предназначенной для приложений, появление которых во время определения базы данных еще нельзя было предвидеть в полном объеме. Поэтому многие возможности, которые профессионалами по базам данных расцениваются как существенные, просто не считались необходимыми в объектном мире, по крайней мере, изначально.*

На первых этапах развития объектных баз данных не было достаточного понимания необходимости в том, чтобы базы данных предоставляли описанные ниже возможности.

1. *Совместный доступ к данным со стороны нескольких приложений.*
2. *Физическая независимость от данных.*
3. *Возможность выполнения произвольных запросов.*
4. *Поддержка представлений и логической независимости от данных.*
5. *Поддержка декларативных ограничений целостности, независимых от прило жений.*
6. *Поддержка прав владения данными и гибкий механизм обеспечения их защиты.*
7. *Управление параллельной работой.*
8. *Каталог общего назначения.*
9. *Возможность проектирования базы данных независимо от приложений.*

Впоследствии, после представления основной идеи хранения объектов в базе данных, эти возможности были рассмотрены и реализованы как усовершенствования и дополнения к исходной объектной модели... Один из важных выводов... заключается в том, что объектная СУБД и реляционная СУБД — системы, которые различны по сути. На самом деле, можно доказать, что объектная СУБД фактически вовсе не является СУБД, по крайней мере, в том смысле, в котором это применимо к реляционной СУБД.

Для сравнения рассмотрим приведенные ниже замечания.

- *Реляционные СУБД поступают от изготовителя готовыми к использованию. Иными словами, как только система установлена, пользовател... могут начать строить базы данных, разрабатывать приложения, запускать запросы и т.д.*
- *Объектную же СУБД можно считать лишь некоторого рода набором средств построения СУБД. После исходной установки объектная СУБД не готова к немедленному использованию... Сначала она должна быть приспособлена к определенной области применения опытными специалистами, которые определят необходимые классы, методы и т.п. Для этого в системе предоставляется набор строительных блоков — средства для сопровождения библиотеки классов, компиляторы методов и т.д. Только после завершения такой подготовки систему можно передать в эксплуатацию прикладным программистам и пользователям. Иными словами, результат такой подготовки уже будет больше напоминать СУБД в привычном смысле этого термина.*

Кроме того, отметим, что результат подготовки такой базы будет зависеть от *конкретного приложения.* Эта система может подходить, например, для приложений САПР/АСУП, но быть совершенно непригодной, например, для медицинских приложений. Иначе говоря, она все еще не стала СУБД общего назначения в том смысле, в котором реляционная СУБД является СУБД общего назначения.

В той же статье [25.16] оспаривается идея (часто называемая **перманентностью, ортогональной типу** [25.2]), в соответствии **с** которой в базу данных можно включать (изменяемые) объекты произвольной сложности[17]. Ниже приведена еще одна цитата, взятая из [25.16].

Для объектной модели требуется поддержка [большого количества] ***генераторов типа...*** *В качестве примера можно указать такие типы, как структура (STRUCT), кортеж (TUPLE), список (LIST), массив (ARRAY), множество (SET), мультимножество (BAG) и т.д.... Вместе с идентификаторами объектов наличие таких генераторов типов фактически означает, что* ***любая структура данных, которая может быть создана в прикладной программе, может быть также создана как объект в объектной базе данных,*** *и, кроме того, такая структура объектов доступна пользователю. Например, рассмотрим объект (допустим, EX), который является коллекцией служащих в отделе (или, скорее, обозначает такую коллекцию). Тогда объект EX может быть реализован как связанный список или как массив, и пользователи должны будут знать, как именно реализован этот объект, поскольку при этом используются разные операторы доступа.*

Такая ***вседозволенность*** *по отношению к типам данных, сохраняемых в базе данных, — главное отличие объектной модели от реляционной. По сути, подходы в двух моделях можно сформулировать, как показано ниже.*

- *В объектной модели можно сохранять в базе данных все, что заблагорассудится, т.е. любые структуры данных, которые можно создать с помощью механизмов обычного языка программирования.*
- *В реляционной модели, по существу, также можно сохранять все что угодно, однако требуется, чтобы то, что хранилось в базе данных, было представлено для пользователя в строго реляционной форме.*

Точнее, в реляционной модели (вполне обоснованно) почти совсем ***ничего*** *не говорится о том, как могут физически храниться данные... Следовательно, реляционная модель не накладывает никаких ограничений на структуры данных, которые допустимы на физическом уровне. Единственное требование заключается в том, что если какая-либо структура реально физически* ***хранится в базе данных,*** *она должна отображаться в отношения на логическом уровне и поэтому должна быть скрытой от пользователя. Таким образом, реляционные системы позволяют провести четкое различие между логическим и физическим представлениями, т.е. моделью данных и их реализацией, а объектные системы этого не позволяют. И, как следствие, вопреки обычному здравому смыслу, объектные системы могут обеспечить лишь значительно меньшую независимость от данных, чем реляционные системы. Предположим, например, что в некоторой объектной базе данных для реализации упомянутого объекта EX, обозначающего коллекцию служащих в данном отделе, вместо массива стал применяться связанный список. Какими будут последствия этого для существующего кода, с помощью которого осуществлялся доступ к объекту EX?*

[17] См. также [25.19].

25.6. РЕЗЮМЕ

В завершение этой главы перечислим представленные в ней важнейшие понятия и возможности объектной модели, давая им свою субъективную оценку: какие из них важны, какие "хороши", но несущественны, какие "неприемлемы", а какие фактически не относятся к вопросу о том, является ли система объектной или какой-то другой, и т.д. Эти выводы будут служить основой для *обсужденпяобъектно-реляционных* систем в главе 26.

- **Классы объектов** (т.е. *типы).* Классы объектов соответствуют *типам данных* и, без условно, очень важны. По существу, это наиболее фундаментальная концепция из всех.
- **Объекты.** Сами объекты, *изменяемые* и *неизменяемые,* очевидно, составляют основу объектных систем, хотя мы предпочли бы их называть просто *переменными* и *зна чениями,* соответственно.
- **Идентификаторы объектов.** Не нужны и даже вредны (на уровне модели), посколь ку по существу это указатели. Этот вопрос подробно рассматривается в [25.19].
- **Инкапсуляция.** Как уже отмечалось в разделе 25.2, *инкапсулированный* означает просто *скалярный,* и мы бы предпочли именно такой термин, всегда помня при этом, что некоторые *объекты,* тем не менее, не являются скалярами.
- **Переменные экземпляра.** Во-первых, *закрытые* (и *защищенные)* переменные экзем пляра по определению относятся к реализации, поэтому они не соответствуют оп ределению абстрактной модели, которую мы здесь рассматриваем. Во-вторых, *от крытых* переменных экземпляра в чисто объектной системе не существует, поэто му они также здесь неуместны. Таким образом, приходим к заключению, что пе ременные экземпляра можно игнорировать, а *объекты* должны обрабатываться исключительно *методами.*
- **Иерархия вложения.** Как уже отмечалось в разделе 25.3, мы считаем, что понятие *иерархий вложения* вводит в заблуждение и на самом деле является неверным, по скольку такие иерархии включают *идентификаторы,* а не *объекты.*
  Примечание. Возможным, хотя и не совсем удачным решением может оказаться применение иерархий (неинкапсулированных), которые действительно включали бы объекты *как таковые* и были бы примерно аналогичными переменным отно шения, имеющим атрибуты со значением в виде отношения (см. части II и **III** этой книги).
- **Методы.** Это, конечно, важное понятие, хотя мы предпочли бы более привычный термин *операторы.* Но объединение методов с операторами *не* является обязатель ным и приводит к некоторым проблемам [3.3]. Раздельное определение *классов* (типов) и *методов* (операторов), как было показано в главе 6, позволяет обойтись без использования понятия объекта-получателя.
  Существуют некоторые операторы, на включении которых мы бы настаивали: операторы выборки, которые, кроме всего прочего, предоставляют возможность записи литеральных значений соответствующего типа, операторы THE_, операто ры присвоения и сравнения на эквивалентность, а также операторы проверки типа (см. главу 20).
  Примечание. Однако мы бы отказались от функций-конструкторов. Конструкто ры создают *переменные.* А поскольку для нас единственными необходимыми

переменными в базе данных являются переменные отношения, единственный *конструктор,* который нам нужен, — это оператор создания переменной отношения, т.е. CREATE TABLE или CREATE VIEW (по терминологии языка SQL). Операторы выборки, наоборот, выбирают *значения.* Конечно, есть и дополнительное различие в том, что конструкторы возвращают *указатели* на созданные переменные, в то время как операторы выборки возвращают сами выбранные значения.

- **Сообщения.** Это также одно из основных понятий, хотя мы предпочли бы более привычный термин *вызов.* Опять же, можно обойтись без использования понятия объекта-получателя, если отказаться от требования непосредственного обращения к некоторому объекту-получателю, а считать все фактические параметры равно правными.
- **Иерархия классов.** С иерархией классов связаны такие понятия, как наследование, полиморфизм, перегрузка, переопределение и т.д. Считаем, что понятие иерархии классов — нужное, но производное. Мы рассматриваем поддержку иерархии клас сов как составляющую поддержки самих классов.
- **Классы, экземпляры, коллекции.** Различия между этими понятиями, конечно, су щественны, но они касаются не только объектного подхода (здесь мы ограничимся констатацией, что данные *понятия* различаются).
- **Связи. В** главе 14 (раздел 14.6) уже оспаривалась идея трактовки *связей* как фор мально отдельной конструкции (особенно если это лишь бинарная связь, которая и заслужила такую специальную трактовку). Мы также не считаем удачной трак товку связанных ограничений ссылочной целостности, которая расходится с трак товкой ограничений целостности вообще (см. ниже).
- **Интегрированные языки программирования баз данных.** Весьма полезны, но не от носятся исключительно к объектной технологии. Тем не менее, следует отметить, что языки, которые поддерживаются современными объектными системами, обычно являются *процедурными* (языки третьего поколения), поэтому можно дока зать, что они неудачны (фактически это огромный шаг назад).

Теперь перечислим возможности, которые в "объектных моделях" обычно *не* поддерживаются или поддерживаются не в полной мере.

- **Произвольные запросы.** В ранних версиях объектных моделей поддержка произ вольных запросов обычно не предусматривалась, поскольку в той среде, в которой возникли объектные системы, в них не было особой необходимости. В более позд них версиях появилась поддержка произвольных запросов, но для их выполнения обычно требуется либо отказываться от инкапсуляции, либо вводить ограничения на виды запросов, которые могут быть выполнены (думается, что в последнем слу чае такие запросы вряд ли заслуживают названия произвольных).
- **Представления.** Обычно не поддерживаются (в основном по тем же причинам, что и обработка произвольных запросов).
  Примечание. В некоторых объектных системах поддерживаются *производные* или *виртуальные* переменные экземпляра (обязательно открытые). Например, переменная экземпляра AGE (Возраст) может наследоваться с помощью вычитания значения переменной экземпляра BIRTHDATE (Дата рождения) из текущей даты. Однако такая возможность еще очень далека от возможностей, которые

предоставляются с помощью механизма представлений; **и** кроме того, мы уже о тказались от понятия *открытой* переменной экземпляра.

■ **Декларативные ограничения целостности.** Обычно не поддерживаются (в основном по тем же причинам, что и представления и обработка произвольных запросов). Более того, они обычно не поддерживаются даже теми системами, которые поддерживают произвольные запросы.

■ **Внешние ключи.** В "объектной модели" предусмотрено несколько разных методов поддержки целостности на уровне ссылок, ни один из которых не похож на более универсальный метод внешних ключей, используемый в реляционной модели.
Такие понятия, как ограниченное (ON DELETE RESTRICT) и каскадное (ON DELETE CASCADE) удаления, обычно реализуются с помощью процедурного кода
(размещенного либо в методах, либо в коде приложений).

■ **Замкнутость.** Сложно найти объектный аналог реляционному свойству замкнутости.

■ **Каталог.** Где же каталог в объектной системе? Как он выглядит? Есть ли какие-либо стандарты?

Примечание. Конечно, это вопросы риторические. Каков будет реальный результат, если такой каталог будет создан специалистом-профессионалом, которому будет поручено выполнить подгонку объектной СУБД, устанавливаемой для какого-либо приложения, как обсуждалось в разделе 25.5. (Такой каталог будет специфическим для данного приложения, как, впрочем, и вся настроенная СУБД.)

Подытожив сказанное выше, можно отметить приведенные в табл. 25.1 полезные (существенные, фундаментальные) понятия **и** концепции "объектной модели" (такие, которые желательно поддерживать).

Таблица 25.1. Полезные понятия и концепции "объектной модели"

Понятие	*Предпочтительный термин*	*Замечания*
Класс объекта	Тип	Скаляр и нескаляр; может определяться пользователем
Неизменяемый объект	Значение	Скаляр и нескаляр
Изменяемый объект	Переменная	Скаляр и нескаляр
Метод	Оператор	Включает операторы выборки, операторы "THE_", ":=", "=" и оператор проверки типа
Сообщение	Вызов оператора	Никаких целевых операндов

Более кратко можно сказать, что единственная хорошая идея объектных систем в целом — это **надлежащая поддержка типов данных**[18] (все остальное, включая понятия операторов, определяемых пользователем, следует из этой идеи). Но данную идею вряд ли можно назвать новой!

[18] Кое-кто может возразить, что наследование типа — также хорошая идея. Автор против этого не возражает, а лишь настаивает на том, что поддержка наследования не связана с поддержкой объектов как таковых.

УПРАЖНЕНИЯ

25.1. Дайте определения следующим терминам:

закрытая переменная экземпляра
защищенная переменная экземпляра
идентификатор объекта
иерархия вложения
иерархия классов
инкапсуляция
класс
метод
обратная переменная
объект
объект, определяющий класс
открытая переменная экземпляра
сообщение
функция-конструктор
экземпляр
экземпляр объекта

25.2. В чем заключаются преимущества использования идентификаторов объектов? В чем состоят их недостатки? Каким образом можно реализовать идентификаторы?

25.3. В разделе 25.2 были представлены две формулировки на языке SQL следующего запроса: "Найти все прямоугольники, которые перекрывают какую-нибудь об ласть единичного квадрата". Докажите, что эти формулировки эквивалентны.

25.4. Исследуйте любую доступную вам объектную систему. Какой язык (языки) про граммирования поддерживается в этой системе? Поддерживается ли в ней язык запросов? Если поддерживается, то какой? Является ли он, по вашему мнению, более мощным, чем язык SQL? Как организован каталог системы? Как пользова тель опрашивает каталог? Предусмотрена ли в этой системе поддержка представ лений? Если предусмотрена, то в какой степени (например, поддерживается ли в ней обновление представлений)? Как обрабатывается *отсутствующая информация?*

25.5. Спроектируйте объектную версию базы данных поставщиков и деталей, исполь зуемой в этой книге.

Примечание. Этот макет будет использоваться как основа для приведенных ниже упр. 25.6-25.8.

25.6. Составьте подходящий набор предложений определения данных на языке OPAL для объектной версии базы данных поставщиков и деталей.

25.7. Составьте схематические определения методов *заполнения базы данных* для объект ной версии базы данных поставщиков и деталей.

25.8. Разработайте код на языке OPAL для реализации приведенных ниже запросов в объектной версии базы данных поставщиков и деталей.

а) Получить сведения обо всех поставщиках, находящихся в городе ' London'.

б) Получить сведения обо всех деталях красного цвета.

25.9. Еще раз рассмотрите образовательную базу данных. Покажите, какие действия должны быть выполнены для осуществления следующих операций:

а) удаление слушателя;

б) удаление служащего;

в) удаление курса;

г) удаление класса слушателей;

д) удаление класса служащих.

Подразумевается, что в каждом случае используется механизм автоматической сборки мусора языка OPAL. Приведите любые допущения, которые необходимо сделать относительно таких средств, как каскадное удаление и др.

25.10. Допустим, что база данных поставщиков, деталей и проектов организована с ис пользованием простой объектной иерархии вложения. Сколько таких иерархий может быть реализовано? Какая из них является наилучшей?

25.11. Рассмотрите вариант базы данных поставщиков, деталей и проектов, в котором вместо записей о том, что некоторые поставщики поставляют некоторые детали для некоторых проектов, содержатся записи о том, что некоторые поставщики поставляют некоторые детали, некоторые детали поставляются для некоторых проектов и некоторые проекты обеспечиваются деталями от некоторых поставщи ков. Сколько может быть реализовано таких объектных проектов (с учетом или без учета иерархии вложения)?

25.12. Рассмотрите основные факторы, оказывающие влияние на производительность, которые кратко обсуждались в разделе 25.5. Справедливо ли утверждение, что они характерны только для объектных систем? Обоснуйте свой ответ.

25.13. **В** объектных системах ограничения целостности обычно поддерживаются *проце дурно* (т.е. с помощью методов), основным исключением из этого правила являют ся ограничения ссылочной целостности, которые обычно поддерживаются *декла ративно* (по крайней мере, частично). В чем состоят преимущества процедурного способа поддержки ограничений целостности? Почему, по вашему мнению, огра ничения ссылочной целостности поддерживаются иначе?

25.14. Дайте определение понятию *обратных переменных.*

СПИСОК ЛИТЕРАТУРЫ

Публикации [25.5], [25.10], [25.23] и [25.31] представляют собой книги по объектной тематике и связанным с ней вопросам. Публикации [25.29], [25.30] и [25.42] — это сборники научно-исследовательских работ, а [25.24], [25.35] и [25.38] — учебные пособия. В работах [25.4], [25.8] и [25.21] описываются конкретные системы.

25.1. Atkinson M. et al. The Object-Oriented Database System Manifesto // Proc. 1st Int. Conf. on Deductive and Object-Oriented Databases. — Kyoto, Japan, 1989. New York, N.Y.: Elsevier Science, 1990.

Одна из первых попыток достичь согласия по вопросу, что же должна включать в себя "объектная модель". Предлагаются перечисленные ниже обязательные средства (т.е. средства, которые должны поддерживаться для того, чтобы рассматриваемая СУБД заслуживала названия *объектно-ориентированной).*

1. Коллекции.
2. Идентификаторы объектов.
3. Инкапсуляция.
4. Типы или классы (?!).

5. Наследование.
6. Позднее связывание.
7. Вычислительная полнота.
8. Типы, определяемые пользователем.
9. Перманентность.
10. Поддержка больших баз данных.
11. Организация параллельной работы.
12. Восстановление.
13. Поддержка произвольных запросов.

Также предложены некоторые необязательные средства, включая множественное наследование и проверку типов на этапе компиляции; некоторые *открытые* средства, включая "выбор принципа программирования" (например, функциональное или декларативное); вопросы, по которым авторы не смогли достичь согласия, включая (как это ни странно, учитывая их важность) представления и ограничения целостности.

Примечание. В [3.3] и [25.8] приведены комментарии к этой статье. Относительно комментариев в [3.3] необходимо отметить, что они основываются на предпосылке, что назначение статьи — определить свойства лучших, настоящих СУБД общего назначения. Автор не отрицает, что перечисленные выше средства могут быть полезны для специализированных СУБД, которые связаны с конкретным приложением, например, САПР/АСУП, для которого не требуется, скажем, поддержка ограничений целостности. Но тогда возникает вопрос, является ли такая система системой управления базами данных в полном смысле этого понятия.

Ссылка на эту работу приведена также как [20.2] в главе 20, где можно найти дополнительные комментарии.

25.2. Atkinson M.P., Buneman O.P. Types and Persistence in Database Programming Languages//ACM Cотр. Surv. — June 1987. — 19, №2.

Одна из первых статей, если не самая первая, в которой формулируется точка зрения, согласно которой перманентность в языках программирования баз данных должна быть ортогональной по отношению к типу. Эта статья рекомендуется в качестве вводного пособия по изучению языков программирования баз данных в целом ("языки программирования баз данных" многими воспринимались как *необходимое условие* объектных систем; см., например, [25.10], [25.11]).

25.3. Bancilhon F. A Logic-Programming/Object-Oriented Cocktail // ACM SIGMOD. — September 1986.- 15, № 3.

Цитата из введения: "Объектно-ориентированный подход... кажется наиболее приемлемым для управления такими новыми типами приложений, как САПР/АСУП, для разработки программного обеспечения и для решения задач искусственного интеллекта. Однако естественным расширением технологии реляционных баз данных является... подход, основанный на логическом программировании, а не объектно-ориентированный подход. В данной статье рассматривается

возможность совмещения этих двух принципов программирования". И в этой работе сделан осторожный вывод, что эти два подхода совместимы.

Примечание. В [25.40] излагается противоположная точка зрения.

25.4. Banerjee J. et al. Data Model Issues for Object-Oriented Applications // ACM TOOIS (Transaction on Office Information Systems). — March 1987. — 5, № 1. (Эта работа также опубликована в M. Stonebraker (ed.). Readings in Database Systems. — San Mateo, Calif.: Morgan Kaufmann, 1994, а также в [25.42].)

25.5. Barry D.K. et al. The Object-Oriented Database Handbook: How to Select, Implement, And Use Object-Oriented Databases. — New York, N.Y.: John Wiley and Sons, 1996.

Основная мысль этой книги заключается в том, что если мы имеем дело со *сложными данными,* необходима объектная система, а не реляционная. Сложные данные характеризуются как а) вездесущие, б) зачастую не имеющие уникальной идентификации (!), в) характеризующиеся использованием многочисленных связей типа "многие ко многим" и г) часто требующие использования кодов типов "в реляционной схеме" (поскольку в современных продуктах SQL не предусмотрена достаточная непосредственная поддержка для подтипов и супертипов).

Примечание. Автор является исполнительным директором группы Object Data Management Group (ODMG) [25.11].

25.6. Beech D. A Foundation for Evolution from Relational to Object Databases // J. W. Schmidt, S. Ceri, and M. Missikoff (eds.). Extending Database Technology. — New York, N.Y.: Springer Verlag, 1988.

Это одна из нескольких статей, в которых обсуждались возможности расширения языка SQL для преобразования его в некоторую разновидность "объектного SQL", или "OSQL" (необходимо предупредить читателя, что такие "объектные языки SQL" часто не похожи на язык SQL как таковой). Более подробные сведения о предложениях данной конкретной статьи рассматриваются в [25.32].

25.7. Bjornerstedt A., Hulten C. Version Control in an Object-Oriented Architecture. (Опубликована в [25.30].)

Для многих приложений необходима концепция отдельных **версий** данного объекта. Примерами таких приложений могут служить средства разработки программного обеспечения, проектирования аппаратных средств, создания документов и т.д. И некоторые объектные системы прямо поддерживают эту концепцию (хотя на самом деле данная концепция непосредственно не относится к вопросу о том, идет ли речь об объектной системе или о какой-то другой). Такая поддержка обычно включает перечисленные ниже возможности.

■ Создание новой версии данного объекта, обычно с помощью **оформления выдачи** копии объекта и ее перемещения из базы данных на собственную рабочую станцию пользователя, где эта копия может корректироваться в течение продолжительного времени (например, несколько часов или суток). ■ Получение версии данного объекта как версии текущей базы данных, обычно с помощью **ее регистрации** (сдачи на хранение) и пересылки с рабочей станции пользователя обратно в базу данных, для которой, в свою очередь, может потребоваться некоторый механизм **слияния** отдельных версий.

- Удаление и, возможно, **архивирование** устаревших версий.
- Запрос **истории версий** данного объекта.

  Отметим, что, как показано на рис. 25.7, истории версий не обязательно связаны линейно (из версии V. 2 образуются две разные версии, V.3a и V.3b, которые затем сливаются в версию V. 4).

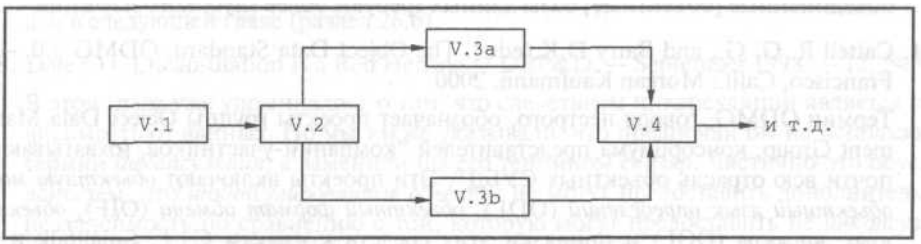


Рис. 25.7. Типичная история версий объекта

Поскольку объекты обычно взаимосвязаны различными способами, концепция версий приводит к концепции *конфигураций.* **Конфигурацией** называется коллекция взаимно согласованных версий взаимосвязанных объектов. Сопровождение конфигурации требует выполнения некоторых описанных ниже основных операций.

- **Копирование** версии объекта из одной конфигурации в другую (например, из "старой" конфигурации в "новую").
- **Перемещение** версии объекта из одной конфигурации в другую (вставка в "новую" конфигурацию и удаление из "старой").

  Для реализации таких возможностей требуется выполнить довольно много операций с указателями, что оказывает значительное влияние на синтаксис и семантику языка вообще и на средства выполнения произвольных запросов в частности.

25.8. Butterworth P., Otis A., Stein J. The GemStone Object Database Management System // CACM. - October 1991.-34, № 10.

25.9. Carey M.J., DeWitt D.J., Naughton J.F. The OO7 Object-Oriented Databases Benchmark // Proc. 1993 ACM SIGMOD Int. Conf. on Management of Data.— Washington, DC. - May 1993.

25.10. Cattell R.G.G. Object Data Management (пересмотренное издание). — Reading, Mass.: Addison-Wesley, 1994.

Первое подробное учебное пособие по применению объектной технологии специально для систем управления базами данных. Цитируемый ниже текст дает представление о том, что в 1994 году единство мнений в этой области еще не была достигнуто (и, возможно, ситуация продолжает оставаться таковой): "Для языков программирования может потребоваться новый синтаксис... подстановка указателей, репликация и новые методы доступа требуют дальнейшего изучения... требуются новые инструменты пользователя и средства разработки приложений... необходимо разработать более мощные языки запросов... необходимо новые исследования

в области управления параллельной работой... семантика средств применения временных отметок и организации параллельной работы на основе объектов также нуждается в дополнительном исследовании... необходимы модели оценки производительности... новые средства управления знаниями следует интегрировать с инструментами управления объектами и данными... это приведет к сложным проблемам оптимизации, и лишь некоторые исследователи обладают необходимым опытом... объединенные [объектные] базы данных требуют более глубокого изучения".

25.11. Cattell R. G. G., and Barry D.K.(eds.). The Object Data Standard: ODMG 3.0. — San Francisco, Calif.: Morgan Kaufmann, 2000.

Термин ODMG, говоря нестрого, обозначает проекты группы Object Data Management Group, консорциума представителей "компаний-участников, [охватывающих] почти всю отрасль объектных СУБД". Эти проекты включают *объектную модель, объектный язык определений* (ODL), *объектный формат обмена* (OIF), *объектный язык запросов* (OQL) и привязки этих средств к языкам C++, Smalltalk и Java. (Компонент *язык обработки объектов* отсутствует, а вместо него предоставляются средства манипулирования объектами с помощью любого языка, для которого ODMG предоставляет привязку.)

Детальный анализ и критику объектной модели ODMG версии 2.0 (которая фактически почти не отличается от версии 3.0) можно найти в приложении I к работе [3.3]; см также [25.28].

25.12. Cattell R.G.G., Skeen J. Object Operations Benchmark // ACM TODS. — March 1992.-17, №1.

25.13. Copeland G., Maier D. Making Smalltalk a Database System // Proc. 1984 ACM SIGMOD Int. Conf. on Management of Data. — Boston, Mass., — June, 1984. (Переиздано: M. Stonebraker. Readings in Database Systems (2-е изд.). — San Mateo, Calif.: Morgan Kaufmann, 1994.)

В работе описаны некоторые усовершенствования и изменения, внесенные в язык Smalltalk [25.23] при создании СУБД GemStone и языка OPAL.

25.14. Dahl O.J., Myhrhaug B., Nygaard K. The SIMULA 67 Common Base Language. Pub. S-22. — Oslo, Norway: Norwegian Computing Center, 1970.

Язык SIMULA 67 спроектирован специально для создания имитационных приложений. Именно на основе таких языков программирования и была создана объектная технология. Фактически язык SIMULA67 был первым объектным языком.

25.15. Date C.J. An Optimization Problem// C.J. Date and Hugh Darwen. Relational Database Writings 1989-1991. — Reading, Mass.: Addison-Wesley, 1992.

25.16. Date C.J. Why the Object Model' Is Not a Data Model // Date C.J., Darwen H. and McGoveran D. Relational Database Writings 1994—1997. — Reading, Mass.: Addison-Wesley, 1998.

Приведены доводы в пользу того утверждения, что "объектная модель", из чего бы она не состояла, "ближе к физическим структурам" (менее абстрактна и поэтому более зависима отданных), чем реляционная модель; в действительности, объектная модель представляет собой модель хранения, а не модель данных как таковую.

25.17. Date C.J. Object Identifiers vs. Relational Keys // Date C.J., Darwen H. and McGoveran D. Relational Database Writings 1994-1997. — Reading, Mass.: Addison-Wesley, 1998.

Приведены доводы в пользу того утверждения, что идентификаторы объектов не должны присутствовать в той модели данных, к которой имеет доступ пользователь.

Примечание. Один из важных доводов в пользу этой позиции, который не был включен в рассматриваемую работу, касается проблемы наследования; он приведен в следующей главе (раздел 26.6).

25.18. Date C.J. Encapsulation Is a Red Herring // DBP&D. — September 1998. — 12, № 9.

В этой главе уже упоминалось о том, что следствием инкапсуляции является независимость от данных. Но мы также указывали, что предпочли бы не использовать термин *инкапсуляция,* а заменили бы его термином *скаляр.* Частично это обусловлено тем, что *инкапсулированные объекты* не могут предоставить дополнительную независимость по сравнению с той, которую могут предоставить не *инкапсулированные* отношения (по крайней мере, в принципе). Например, нет абсолютно никаких причин для того, чтобы базовое отношение, которое представляет точку в декартовой системе координат X и Y, нельзя было хранить, используя полярные координаты R и 0.

25.19. Date C.J. Persistence Not Orthogonal to Type // DBP&D website www.dbpd.com. — October 1998.

Приведены обоснованные возражения против того утверждения специалистов в области объектно-ориентированного подхода, что "перманентность [должна быть] ортогональной по отношению к типу" [25.2], и поэтому (вполне справедливо) в пользу информационного принципа.

25.20. Date C.J. Decent Exposure//www. dbpd. com. — November 1998.

25.21. Deux O. et al. The O2 System // CACM. - October 1991. - 34, № 1.

25.22. Ferrandina F., Meyer T., Zicari R., Ferran G., Madec J. Schema and Database Evolution in the 02 Object Database System // Proc. 21st Int. Conf. on Very Large Data Bases. — Zurich, Switzerland. — September 1995.

См. аннотацию к [25.34].

25.23. Goldberg A., Robson D. Smalltalk-80: The Language and its Implementation. — Reading, Mass.: Addison-Wesley, 1983.

Перечень передовых исследований специалистов из исследовательского центра фирмы Xerox в Пало-Альто, посвященных проектированию и созданию системы Smalltalk-80. В первой из четырех частей этой книги подробно описывается язык программирования Smalltalk-80, на котором основаны язык OPAL и система GemStone.

25.24. Goodman N. Object Oriented Database Systems // InfoDB. - 1989. - 4, № 3.

Приведем цитату из этой статьи: "На данном этапе нет смысла сравнивать реляционный и объектный подходы. Следует сравнивать лишь подобные понятия, например, яблоки с яблоками, мечты с мечтами, теорию с теорией и зрелые продукты со зрелыми продуктами... Некоторое время реляционный подход использовался

потому, что имел строгий теоретический базис и лежал в основе большого количества добротных программных продуктов. Объектный подход, наоборот, является новым (по крайней мере, в области создания баз данных). Он не обладает той теоретической основой, которая сравнилась бы с реляционной моделью, и лишь немногие программные продукты, созданные на его основе, могут быть охарактеризованы как зрелые. Таким образом, прежде чем заявить об объектном подходе как о жизнеспособной альтернативе реляционному подходу, придется выполнить очень большой объем работы".

Несмотря на то, что большая часть высказанных здесь замечаний еще остается в силе, все же с 1989 года многое прояснилось. Стало очевидно, что сравнение реляционных и объектных систем является столь же "корректным", как сравнительный анализ преимуществ и недостатков яблок и апельсинов (что будет показано в главе 26).

25.25. Goodman N. Object Oriented DBMS War Story: Developing a Genome Mapping Database in C++ (опубликовано в [25.29]).

В статье поддерживаются многие критические замечания, которые были высказаны в этой главе. Далее приводится цитата из резюме статьи: "Вопреки распространенному мнению, наш опыт подсказывает, что было бы ошибкой создавать слишком интеллектуальные базы данных с помощью сложных программ в виде методов внутри объектов базы данных. Также наш опыт свидетельствует о том, что язык C++ плохо подходит для реализации баз данных, из-за проблем, связанных с механизмами определения атрибутов, механизмами установления ссылок на объекты систематическим путем, недостатками при *сборе мусора* и малозаметными дефектами в модели наследования. Мы также считаем, что современным СУБД, основанным на C++, не достает некоторых важных функций баз данных, и чтобы это компенсировать, нам пришлось предоставлять собственные простые реализации стандартных функций СУБД: средства ведения журнала транзакций для прямого восстановления и отслеживания многопотоковых транзакций, язык запросов и процессор запросов, а также структуры памяти. В результате мы использовали СУБД, основанную на C++, как объектно-ориентированный диспетчер памяти и, кроме того, встроили систему управления данными, предназначенную для крупномасштабного представления генома на ее основе".

25.26. Jagadish H.V., Xiaolei Q. Integrity Maintenance in an Object-Oriented Database // Proc 18th Int. Conf. on Very Large Data Bases. — Vancouver, Canada. — August 1992.

В работе предлагается декларативный метод задания ограничений целостности для объектных систем. При этом описывается, как компилятор ограничений помещает в методы соответствующих объектных классов необходимый код проверки целостности.

25.27. Kifer M., Kim W., Sagiv Y. Querying Object-Oriented Databases // Proc. ACM SIGMOD Int. Conf. on Management of Data. — San Diego, Calif. — June 1992.

В работе предлагается еще один "объектный вариант языка SQL" под названием XSQL.

25.28. Kim W. Observations on the ODMG-93 Proposal for an Object-Oriented Database Language //ACM SIGMOD Record. — March 1994. — 23, № 1.

25.29. Kim W (ed.). Modern Database Systems: The Object Model, Interoperability, and Beyond. — New York, N.Y.: ACM Press/Reading, Mass.: Addison-Wesley, 1995.

25.30. Kim W., Lochovsky F.H. Object-Oriented Concepts, Databases and Applications.— Reading, Mass.: ACM Press/Addison-Wesley, 1989.

25.31. Loomis M.E.S. Object Databases: The Essentials. — Reading, Mass.: Addison-Wesley, 1995.

25.32. Lyngbaek P. et al. OSQL: Language for Object Databases. — Technical Report HPL-DTD-91-4. — Hewlett-Packard Company. — January 1991.

См. аннотацию к [25.6].

25.33. Meyer B. The Future of Object Technology // IEEE Computer. — January 1998. — 31, № 1.

Приведем цитату из этой работы: "Будущее [объектных] баз данных — это интересная тема для размышлений... Изготовители реляционных баз данных с 1986 года старались подавить рост [объектных] баз данных с помощью упреждающих извещений... Десять лет спустя эксперты [объектных баз данных] будут говорить вам, что предложения от основных производителей реляционных СУБД... еще далеки от реальности... Рынок для объектных баз данных будет продолжать расти, но по-прежнему будет составлять лишь часть рынка традиционных баз данных".

25.34. Roddick J.F. Schema Evolution in Database Systems — An Annotated Bibliography // ACM SIGMOD Record. - December 1992. - 21, № 4.

В традиционных СУБД обычно поддерживаются только очень простые изменения в существующей схеме (например, добавление нового столбца в существующую базовую таблицу в программных продуктах SQL). Но в отдельных приложениях требуется поддержка более сложных изменений схемы, и в некоторых объектных прототипах эта проблема исследована достаточно глубоко. Причем в случае объектной системы она становится более сложной, поскольку сама по себе такая система имеет более сложную схему.

Ниже приведена классификация возможных изменений схемы, основанная на той классификации, которая приведена в статье по прототипу объектной системы ORION [25.4]. В статье отмечено, что некоторые из них (но сказано, какие) затрудняют решение проблемы разграничения между моделью и реализацией. .

- Изменения объектного класса

  1. Изменения переменной экземпляра:
     - добавление переменной экземпляра;
     - удаление переменной экземпляра;
     - переименование переменной экземпляра;
     - изменение принятого по умолчанию значения переменной экземпляра;

- изменение типа данных переменной экземпляра;
- изменение источника наследования переменной экземпляра.

2. Изменения метода:
   - добавление метода;
   - удаление метода;
   - переименование метода;
   - изменение внутреннего кода метода;
   - изменение источника наследования метода.

- Изменения в иерархии классов (предполагается множественное наследование):
  - добавление класса A к списку суперклассов класса B;
  - удаление класса A из списка суперклассов класса B.

- Изменения общей схемы:
  - добавление класса (в произвольном месте);
  - удаление класса (в произвольном месте);
  - переименование класса;
  - разбиение класса;
  - слияние классов.

Поскольку поддержка представлений обычно не включается в "объектную модель", не совсем ясно, насколько сильным окажется влияние таких изменений на прозрачность системы. Возможность "эволюции схемы" системы представляет собой достаточно сложную проблему именно из-за того, что объектные системы обладают свойствами языков третьего поколения. Как отмечается в [25.28]: "Если... [происходит добавление] или удаление индексов или данные перераспределяются по кластерам иным образом, то нельзя найти способ автоматически учесть такие изменения в [методах]".

Более того, задача обеспечения эволюции схемы чаще возникает в объектных системах, поскольку многие решения, которые в реляционной системе должны приниматься на уровне администратора базы данных (или даже на уровне самой СУБД!), в объектной системе принимаются на уровне прикладного программиста (см. [25.35]). В частности, к перепроектированию схемы может привести даже настройка производительности системы (также см. [25.35]).

25.35. Saracco CM. Writing an Object DBMS Application (в двух частях) // InfoDB. — 1993-1994. - 7, № 4; InfoDB. - 1994. - 8, № 1.

Приводятся несложные, но достаточно полные и информативные примеры программ.

25.36. Shaw CM., Zdonik S.B. A Query Algebra for Object-Oriented Databases // Proc. 6th IEEE Int. Conf. on Data Engineering. — February 1990.

Эта статья подтверждает мнение автора данной книги о том, что "любая объектная алгебра" является неизбежно является сложной из-за сложной организации самих объектов. В частности, операция проверки на равенство иерархических объектов с произвольной вложенностью требует очень тщательной организации всего этого процесса. Основная идея статьи заключается в том, что каждый оператор алгебры запросов приводит к созданию *отношения,* каждый кортеж которого содержит идентификаторы некоторых объектов базы данных. В случае операции соединения, например, каждый кортеж будет содержать идентификаторы объектов, которые соответствуют друг другу согласно заданному условию соединения. Но эти кортежи не наследуют никаких методов объектов-компонентов.

25.37. Shipman D.W. The Functional Data Model and the Data Language DAPLEX // ACM TODS.— March 1981.— 6, № 1. (Переиздано: M. Stonebraker (ed.). Readings in Database Systems, 2nd edition. — San Mateo, Calif.: Morgan Kaufmann, 1994.)

В течение многих лет было предпринято несколько попыток создания систем, в основе которых вместо отношений лежат *функции,* и среди них наиболее известной является система DAPLEX. Автор упоминает здесь о функциональных подходах, поскольку они обладают некоторыми чертами, характерными для объектных систем, включая, в частности, навигационный стиль адресации объектов (адресация с помощью указания пути), которые функционально связаны с другими объектами (а они, в свою очередь, также функционально связаны с другими объектами и т.д.). Но заслуживает внимания то, что термин *функция* в этих так называемых функциональных моделях обычно не имеет никакого отношения к математической функции. Например, такая функция вполне может быть многозначной. Фактически традиционное понятие функции должно претерпеть значительные изменения для того, чтобы удовлетворить всем требованиям, которые предъявляются к нему в контексте "функциональной модели данных".

25.38. Stein J., Maier D. Concepts in Object-Oriented Data Management // DBP&D. — April 1988.- 1,№4.

Прекрасный учебный материал по объектным концепциям, представленный двумя разработчиками системы GemStone.

25.39. Tsichritzis D.C., Nierstrasz O.M. Directions in **00** Research (опубликовано в [25.30]).

В этой статье также подтверждается высказанная автором настоящей книги мысль об отсутствии единства мнений в области объектных подходов: "Разногласия существуют по самым основным понятиям, например: «Что такое объект?»... Но нет причин для беспокойства, поскольку нестрогие определения неизбежны, а порой они даже крайне желательны во время динамичного развития научных исследований. Они должны стать и неизбежно станут строгими спустя некоторое время". Однако объектные концепции известны уже более 40 лет! — фактически они предшествовали реляционной модели.

25.40. Ullman J.D. A Comparison between Deductive and Object-Oriented Database Systems // Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases. — Munich, Germany. — December 1991; DeloberC, Kifer M., Masunaga Y. (eds). Lecture Notes in Computer Science 566. — New York, N.Y.: Springer Verlag, 1992.

Хотя автор настоящей книги и возражает против некоторых положений, высказанных в данной статье, он согласен с общим выводом, что *дедуктивные* (т.е. основанные на логике) системы баз данных имеют большие долговременные перспективы по сравнению с объектными системами. В статье также затронут важный вопрос относительно *возможностей оптимизации* систем:

"Предположим, что мы создали какой-то объектный класс, поведение которого аналогично поведению бинарных отношений, и... метод соединения для этого класса; тогда можно записать, например, выражение R JOIN S JOIN т. Его можно вычислять как (R JOIN S) JOIN т или как R JOIN (S JOIN T). Но сможем ли мы это сделать? Еще никто не определил, что фактически означает *метод соединения.* Является ли он, например, ассоциативным?... На основании этого можно сделать вывод, что если мы желаем программировать, используя объектную технологию и оставаясь на уровне отношений или более высоком уровне, то необходимо предоставить системе информацию в виде законов реляционной алгебры. Такая информация не может быть выведена системой, но может быть встроена в нее. Таким образом, ...единственной частью языка запросов, которая будет поддаваться оптимизации, является фиксированное множество методов, для которых... соответствующая семантика предоставлена системе".

25.41. Zaniolo C. The Database Language GEM // Proc. 1983 ACM SIGMOD Int. Conf. on Management of Data. — San Jose, Calif. — May 1983. (Переиздано: M. Stonebraker. Readings in Database Systems 2nd edition. — San Mateo, Calif.: Morgan Kaufmann, 1994.)

Аббревиатура GEM (General Entity Manipulator) обозначает язык, который фактически является расширением языка QUEL В языке GEM поддерживаются отношения с атрибутами в виде множеств и отношения с альтернативными атрибутами (например, постоянные работники EMP могут иметь атрибут SALARY, а временные работники EMP — атрибут почасовой оплаты HOURLY_WAGE и атрибут сверхурочной оплаты OVERTIME). Кроме того, в нем поддерживается объектная идея о том, что одни объекты концептуально содержат другие объекты (они используются вместо внешних ключей, которые ссылаются на другие объекты). Причем привычная запись с использованием точки расширена для ссылок на атрибуты таких объектов (хотя на самом деле при этом неявно просматриваются некоторые предпочтительные пути соединения). Например, выражение EMP. DEPT . BUDGET может быть использовано для ссылки на бюджет отдела, в котором работает определенный сотрудник. Многие другие системы либо адаптированы к восприятию этой идеи, либо построены на ее основе.

25.42. Zdonik S.B., Maier D (eds.). Readings in Object-Oriented Database Systems. — San Francisco, Calif.: Morgan Kaufmann, 1990.

ГЛАВА 26

Объектно-реляционные базы данных

26.1. ВВЕДЕНИЕ

В конце 1990-х годов некоторые поставщики выпустили программные продукты *объектно-реляционных* СУБД, известные также под названием универсальных серверов. К примерам таких продуктов относятся версия Universal Database СУБД DB2, опция Universal Data Option сервера Informix Dynamic Server и программный продукт Oracle Universal Server (для этих продуктов используются и другие названия). Выпуская все эти программные продукты, поставщики руководствовались тем основным замыслом, что в них должна обеспечиваться поддержка и объектных, и реляционных возможностей; иными словами, рассматриваемые продукты представляли собой попытку добиться сближения этих двух технологий.

Но, по глубокому убеждению автора, любое такое сближение должно быть полностью основано на реляционной модели (которая в конечном итоге является фундаментом всей современной технологии баз данных в целом, как было описано в части II этой книги).

Итак, мы стремимся к тому, чтобы реляционные системы развивались[1] и включали в себя средства (или, по крайней мере, положительные средства) объектных систем (безусловно, мы не допускаем возможности полного отказа от реляционных систем, а также не хотим того, чтобы нам пришлось иметь дело с двумя отдельными системами, реляционной и объектной, существующими бок о бок). И такое мнение разделяют многие другие специалисты, включая, в частности, авторов "Манифеста систем баз данных третьего поколения" [26.44], которые категорически утверждают, что СУБД третьего поколения должны быть созданы на основе СУБД второго поколения. (Кратко поясним, что подразумевается под поколениями СУБД: название *СУБД первого поколения* относится к дореляционным системам, таким как IMS, *СУБД второго поколения* — это системы SQL, а *СУБД третьего поколения* — те, что придут за ними.) Но очевидно, что такое мнение не разделяют некоторые поставщики объектных систем, а также определенные специалисты в области объектной технологии. Ниже приведена типичная цитата, подтверждающая эту мысль [26.7].

> *Компьютерная наука видела много поколений средств управления данными, начиная с индексированных файлов, затем сетевых и иерархических СУБД, ... [а] в последнее время появились реляционные СУБД... Теперь мы становимся свидетелями появления очередного поколения систем баз данных, ... [которые] обеспечивают управление объектами, [поддерживая] гораздо более сложные виды данных.*

Автор этой цитаты недвусмысленно намекает: так же, как реляционные системы заменили более старые иерархические и сетевые системы, так и объектные системы, в свою очередь, заменят реляционные системы.

Причина, по которой мы не согласны с этой позицией, состоит в том, что **реляционные системы действительно не похожи на** другие средства доступа **к данным** [26.17]. Их отличительная особенность состоит в том, что они созданы не по случаю. Более старые дореляционные системы создавались под влиянием конкретной потребности; они могли обеспечивать решение некоторых важных проблем того времени, но не были основаны на каком-либо солидном теоретическом фундаменте. К сожалению, сторонники реляционного подхода (в том числе и автор данной книги) в те далекие дни оказали сами себе плохую услугу, когда вступили в дискуссию по поводу относительных достоинств реляционных и дореляционных систем; в то время доводы в пользу реляционных были нужны, но имели непредвиденный эффект, поскольку заложили идею, будто реляционные и дореляционные СУБД по сути представляют собой очень близкие понятия. А эта ошибочная идея, в свою очередь, легла в основу взглядов, что объекты являются для отношений тем, чем отношения были для иерархий и сетей.

А что можно сказать об объектных системах? Созданы ли они по случаю? В этот вопрос позволяет внести ясность следующая цитата из "Манифеста объектно-ориентированных систем баз данных" [20.2], [25.1]: "Что касается спецификации данной системы,

---

[1] Следует отметить, что мы, безусловно, заинтересованы в том, чтобы технология баз данных развивалась эволюционным, а не революционным путем. В отличие от этого, заслуживает внимания следующая цитата из [25.11]: "[Объектные СУБД] являются проявлением *революционного, а не эволюционного направления развития"* (курсив автора). Автор не считает, что весь рынок баз данных в целом готов к революции, а также не думает, что она нужна или должна произойти; именно поэтому написанный с его участием Третий Манифест [3.3] по своему характеру специально подготовлен как эволюционный, а не революционный.

то мы являемся сторонниками дарвиновского подхода — мы надеемся, что после создания целого ряда экспериментальных прототипов [произойдет естественный отбор и] появится подходящая [объектная] модель". Иными словами, здесь явно прослеживается мысль, что необходимо разрабатывать код и создавать системы без какой-либо заранее определенной модели, а затем смотреть, что из этого получится!

Поэтому на будущее мы принимаем в качестве аксиомы (фактически, как и большинство крупных поставщиков СУБД), что наша задача — продолжать совершенствовать реляционные системы для включения в них положительных средств объектной технологии. Еще раз повторяем, что мы не хотим отказываться от реляционной технологии; было бы очень жалко перечеркнуть почти 35 лет плодотворных реляционных исследований и разработок.

Итак, в главе 25 было показано (см. также аннотацию к [26.31]), что объектная ориентация просто наталкивает нас на одну хорошую идею — а именно, на то, что необходимо обеспечить надлежащую поддержку типов данных (или две хорошие идеи, если рассматривать наследование типов отдельно). Поэтому вопрос переходит в другую плоскость: "Как включить надлежащую поддержку типов данных в реляционную модель?". А ответ, безусловно, состоит в том (как уже, несомненно, понял читатель), что эта поддержка уже существует, в форме доменов (которые автор, во всяком случае, предпочитает называть *типами)*. Иными словами, в реляционную модель не нужно ничего вносить, чтобы добиться появления объектных функциональных возможностей в реляционных системах, вернее, ничего, кроме полной и должной реализации того, чего так явно недостает в современных системах SQL[2].

Итак, автор считает, что реляционная система, способная поддерживать домены должным образом, будет способна справляться и со всеми теми *проблемными* разновидностями данных, которые (как иногда приходится слышать) объектные системы способны поддерживать, а реляционные системы — нет: мультимедийные данные, данные временных рядов, биологические данные, финансовые данные, данные инженерного проектирования, данные автоматизации делопроизводства и т.д. В соответствии с этим, автор считает также, что настоящая объектно-реляционная система должна представлять собой не больше и не меньше чем реляционную систему в полном смысле этого слова; иначе говоря, систему, которая поддерживает реляционную модель, включая все то, что следует из этой поддержки. Поэтому необходимо стимулировать усилия поставщиков СУБД в том направлении, которое они действительно пытаются развивать, а именно, необходимо помочь им расширить свои системы для включения полной поддержки типов. Безусловно, можно утверждать, что основная причина, по которой объектные системы казались такими привлекательными сразу после своего появления, состоит как раз в том, что поставщики систем SQL не поддерживают реляционную модель должным образом. Но этот факт не следует рассматриваться как довод в пользу полного отказа от реляционных систем (или вообще какого-либо, даже частичного отказа!).

---

[2] В частности, современные системы стали причиной появления широко распространенных ошибочных взглядов, что реляционные системы способны поддерживать лишь ограниченное количество очень простых типов. Весьма типичными являются следующие цитаты: "Реляционные... системы поддерживают небольшую, фиксированную коллекцию типов данных (например, целые числа, даты, строки)" [26.34], "реляционные СУБД способны поддерживать только свои встроенные типы [по существу, лишь целые числа, строки, значения даты и времени]" [25.31], "объектно-реляционные модели данных расширяют реляционную модель данных, предоставляя более развитую систему типов" [16.21] и т.д.

В качестве примера снова вернемся к некоторым незаконченным темам из главы 25 и рассмотрим качественное реляционное решение задачи с прямоугольниками. (Это решение дано на языке Tutorial D; подготовку его аналога на языке SQL оставляем читателю в качестве упражнения.) Вначале определим тип прямоугольника следующим образом.

```
TYPE RECTANGLE POSSREP ( X1 RATIONAL, Y1 RATIONAL,
                         X2 RATIONAL, Y2 RATIONAL ) ... ;
```

Предполагается, что прямоугольники представлены физически с помощью одной из тех структур данных, которые специально предназначены для эффективной поддержки пространственных данных: деревьев квадрантов *(квадрадеревьев),* R-деревьев и т.д. [26.37].

Кроме того, определим оператор для проверки того, перекрываются ли два заданных прямоугольника, как показано ниже.

```
OPERATOR OVERLAP ( R1 RECTANGLE, R2 RECTANGLE )
                                    RETURNS BOOLEAN ;
RETURN ( THE X1 ( Rl ) < THE X2 ( R2 )
AND THE Y1 ( Rl ) < THE Y2 ( R2 ) AND
THE X2 ( Rl ) > THE X1 ( R2 ) AND THE Y2
( Rl ) > THE_Y1 ( R2 ) ) ; END OPERATOR ;
```

Этот оператор реализует эффективную *сокращенную* форму проверки того, перекрываются ли прямоугольники (чтобы вспомнить, что подразумевается под этой сокращенной формой, вернитесь к главе 25) применительно к эффективной структуре хранения данных (в виде R-дерева или какой-то другой).

После этого пользователь может создать базовую переменную отношения с атрибутом типа RECTANGLE следующим образом.

```
VAR RECTANGLES BASE RELATION { R RECTANGLE, ... } KEY { R } ;
```

А запрос "Определить все прямоугольники, которые перекрывают данный единичный квадрат" теперь выглядит следующим образом.

```
RECTANGLES
WHERE OVERLAP ( R, RECTANGLE ( 0.0, 0.0, 1.0, 1.0 ) )
```

В данном решении преодолены все недостатки, обсуждавшиеся в связи с этим запросом в главе 25.

Теперь отметим, что изложенные выше идеи концептуально являются очень простыми и однозначными (еще раз повторяем, что нужно лишь реализовать реляционную модель, в частности, дать возможность пользователям определять свои собственные типы) и все равно (по-прежнему) продолжается путаница ... В действительности, в современных коммерческих программных продуктах мы обнаруживаем по меньшей мере две колоссальные ошибки, которые мы называем двумя серьезными заблуждениями[3]. Безусловно,

---

[3] Один из рецензентов возразил против использования в данном контексте слова "заблуждение", резонно заметив, что такой термин обычно не встречается в научной литературе. Да, автор согласен, что выбрал это слово отчасти из-за того, что оно подчеркивает серьезность ошибки. Но если некоторая система X рассматривается как реализация реляционной модели, а затем (примерно через 25 лет после того, как впервые была определена реляционная модель) некто вводит в систему X такое "средство", которое полностью нарушает предписания этой модели, по мнению автора, вполне допустимо охарактеризовать введение подобного "средства" как заблуждение.

необходимо обсудить эти ошибки и, в частности, показать, почему мы рассматриваем их как заблуждения, поэтому план остальной части данной главы состоит в следующем. В разделах 26.2 и 26.3 рассматриваются два серьезных заблуждения. Затем в разделе 26.4 обсуждаются некоторые аспекты реализации объектно-реляционной системы. В разделе 26.5 демонстрируются преимущества подлинной объектно-реляционной системы (т.е. системы, в которой нет "следов" ни одного из этих двух заблуждений). В разделе 26.6 описаны соответствующие средства SQL. В разделе 26.7 представлено резюме.

26.2. ПЕРВОЕ СЕРЬЕЗНОЕ ЗАБЛУЖДЕНИЕ

Начнем с приведенной ниже цитаты из Третьего Манифеста [3.3].

"[Прежде чем] мы сможем рассмотреть вопрос о [сближении между] объектами и отношениями более подробно, необходимо проанализировать чрезвычайно важный предварительный вопрос, который приведен ниже".

Какое реляционное понятие является аналогом такого объектного понятия, как класс объекта?

Причина столь значительной важности этого вопроса состоит в том, что класс объекта в действительности представляет собой наиболее фундаментальное из всех объектных понятий, поскольку все другие объектные понятия в большей или меньшей степени зависят именно от него. А в качестве ответа на этот вопрос могут быть и были предложены следующие два уравнения:

- *домен = класс объекта;*
- переменная отношения = класс объекта.

Теперь перейдем к строгому обоснованию того, что первое из этих уравнений является правильным, а второе — неправильным.

В действительности то, что первое уравнение является правильным, вполне очевидно, поскольку и классы объектов, и домены представляют собой просто типы. Безусловно, если исходить из того, что переменные отношения — это переменные, а классы — это типы, должно быть также сразу же ясно, что второе уравнение является неправильным (переменные и типы — **не** одно и то же); именно по этой причине в [3.3] приведено категорическое утверждение, **что** переменные отношения — это **не домены.** Тем не менее, многие специалисты, а также разработчики некоторых программных продуктов, фактически руководствуются вторым уравнением, т.е. совершают ошибку, которую мы называем **первым серьезным заблуждением** (The First Great Blunder). Поэтому было бы очень поучительным внимательное изучение второго уравнения, что и будет сделано в этой главе.

Примечание. Остальная часть данного раздела в основном представляет собой более или менее буквальную выдержку из [3.3].

Почему происходит так, что люди оказываются в плену такого заблуждения? Итак, рассмотрим следующее простое определение класса, выраженное на гипотетическом объектном языке, который намеренно сделан аналогичным, но не полностью идентичным тому языку, который применяется в разделе 25.3.

```
CREATE OBJECT CLASS EMP ( EMP#
     CHAR(5), ENAME
     CHAR(20), SAL
     NUMERIC, HOBBY
     CHAR(20), WORKS FOR
     CHAR(20) ) ... ;
```

Здесь EMP#, ENAME и т.д. — открытые переменные экземпляра. (Автор специально определил их как относящиеся к простым встроенным типам, а не к определяемым пользователем типам; к тому же аналогичного соглашения автор будет для упрощения придерживаться во всех остальных примерах данной главы.) Теперь рассмотрим следующее определение *базовой переменной отношения* на языке SQL.

```
CREATE TABLE EMP
     ( EMP#       CHAR(5)     NOT      NULL,
       ENAME      CHAR(20)    NOT      NULL,
       SAL        NUMERICNOT NULL,
       HOBBY      CHAR(20)    NOT      NULL,
       WORKS_FOR CHAR(20)    NOT      NULL ) ...;
```

Эти два определения, безусловно, кажутся очень похожими и поэтому идея их приравнивания друг другу выглядит очень соблазнительной. Поэтому (как уже было сказано) в некоторых системах, включая определенные коммерческие программные продукты, фактически сделано именно это. Таким образом, рассмотрим эту проблему более внимательно. Точнее, примем за основу только что приведенное предложение CREATE TABLE и рассмотрим ряд дополнений к нему (против чего кое-кто мог бы возразить), предназначенных для того, чтобы оно стало "более похожим на объектное".

Примечание. Приведенное ниже описание основано на конкретном коммерческом программном продукте; фактически оно основано на примере из собственной документации этого программного продукта. Но здесь мы не называем сам рассматриваемый программный продукт, поскольку в задачу настоящей книги не входит осуждение недостатков или восхваление достоинств конкретных продуктов. Вместо этого отметим, что критические замечания, которые будут приведены ниже в данном разделе, с соответствующими оговорками, относятся к любой системе, которая основана на уравнении "переменная отношения = класс объекта".

Первое необходимое дополнение состоит в том, что должны быть разрешены составные атрибуты (т.е. атрибуты со значением в виде кортежа); иными словами, необходимо разрешить, чтобы значениями атрибута были кортежи из некоторой другой переменной отношения или, возможно, из той же самой переменной отношения (?!). В данном примере можно заменить первоначальное предложение CREATE TABLE следующей коллекцией предложений (рис. 26.1).

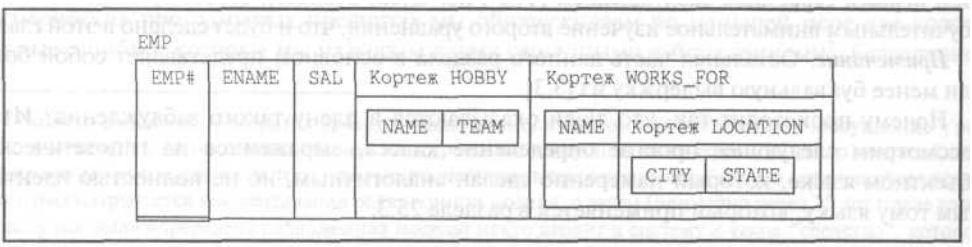


Рис. 26.1. Атрибуты, содержащие кортежи (указатели на кортежи), —

```
CREATE TABLE EMP
     ( EMP#      CHAR(5) NOT NULL,
       ENAME     CHAR(20)NOT NULL,
       SAL        NUMERIC NOT NULL,
       HOBBY     ACTIVITY     NOT NULL,
       WORKS_FOR COMPANY NOT NULL ) ;

CREATE TABLE ACTIVITY
     ( NAME      CHAR(20)     NOT NULL,
       TEAM      INTEGER NOT NULL ) ;

CREATE TABLE COMPANY
     ( NAME      CHAR(20)     NOT NULL,
       LOCATION  CITYSTATE    NOT NULL ) ;

CREATE TABLE CITYSTATE
     ( CITY      CHAR(20)     NOT NULL,
       STATE     CHAR(2)  NOT NULL ) ;
```

Пояснение. Атрибут HOBBY (Увлечение) в переменной отношения EMP объявлен как относящийся к типу ACTIVITY (Вид спорта), а сам тип ACTIVITY, в свою очередь, представляет собой переменную отношения с двумя атрибутами, NAME И TEAM, где TEAM задает количество игроков в команде, соответствующей названию NAME; например, возможным *видом спорта* может быть тот вид футбола, в котором команда состоит из 11 игроков, (Soccer, 11). Поэтому каждое значение HOBBY представляет собой фактически пару значений — значение NAME И значение TEAM (точнее, это — пара значений, которая в настоящее время присутствует в качестве кортежа в переменной отношения ACTIVITY).

Примечание. Обратите внимание на то, что мы уже нарушили предписание Третьего Манифеста, согласно которому переменные отношения не являются доменами, поскольку *домен* для атрибута HOBBY определен как переменная отношения ACTIVITY. Дополнительные сведения по этой теме приведены ниже в данном разделе.

Аналогичным образом, атрибут WORKS_FOR в переменной отношения EMP объявлен как относящийся к типу COMPANY, а COMPANY также представляет собой переменную отношения с двумя атрибутами, один из которых определен как относящийся к типу CITYSTATE, представляющему собой еще одну переменную отношения с двумя атрибутами, и т.д. Иными словами, все переменные отношения ACTIVITY, COMPANY и CITYSTATE рассматриваются как типы (домены) и вместе с тем как переменные отношения. Такое же утверждение, безусловно, является справедливым и применительно к самой переменной отношения EMP.

Таким образом, это первое дополнение примерно аналогично тому, согласно которому допускается включение одних объектов в другие и тем самым поддерживается понятие иерархии вложения (см. главу 25).

Кстати, следует отметить, что это первое дополнение было охарактеризовано как возможность применения атрибутов, содержащих кортежи, поскольку именно так его характеризуют те, кто считают правильным уравнение "переменная отношения = класс объекта". Но было бы точнее охарактеризовать это дополнение как "разрешение использовать атрибуты, содержащие указатели на кортежи"; эта тема будет более подробно рассматриваться чуть позже, а еще более подробно — в следующем разделе. (Поэтому на рис. 26.1 фактически следует заменить три упоминания термина *кортеж* терминами *указатель на кортеж.)*

Замечания, аналогичные приведенным в предыдущем абзаце, относятся также ко второму дополнению, которое состоит в том, что должны быть разрешены *атрибуты со значением в виде отношения;* из этого следует, что в качестве значений атрибутов должно быть разрешено использовать множества кортежей из некоторой другой переменной отношения или, возможно, даже из той же самой переменной отношения (?!). Например, предположим, что служащие могут иметь произвольное количество увлечений, а не только одно (рис. 26.2), как показано в следующем объявлении.

```
CREATE TABLE EMP
  ( EMP#     CHAR(5)                NOT NULL,
             ENAME     CHAR(20)          NOT
                                        NULL,
      SAL      NUMERIC               NOT NULL,
      HOBBIES SET OF ( ACTIVITY ) NOT NULL,
       WORKS_FOR COMPANY            NOT NULL ) ;
```

Пояснение. Теперь значение HOBBIES в каждом конкретном кортеже переменной отношения EMP (концептуально) представляет собой множество пар, т.е кортежей (NAME, TEAM) из переменной отношения ACTIVITY в количестве от нуля и больше. Поэтому данное второе дополнение приблизительно аналогично тому, что разрешается использовать объекты, которые включают объекты *коллекций* — это более сложная версия иерархии вложения.

Примечание. Кстати, следует отметить, что в том конкретном программном продукте, на котором основан данный пример, такими коллекциями могут быть последовательности или мультимножества, а также множества как таковые.

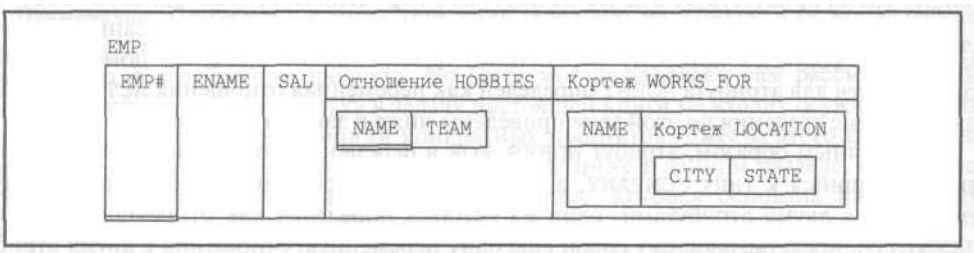


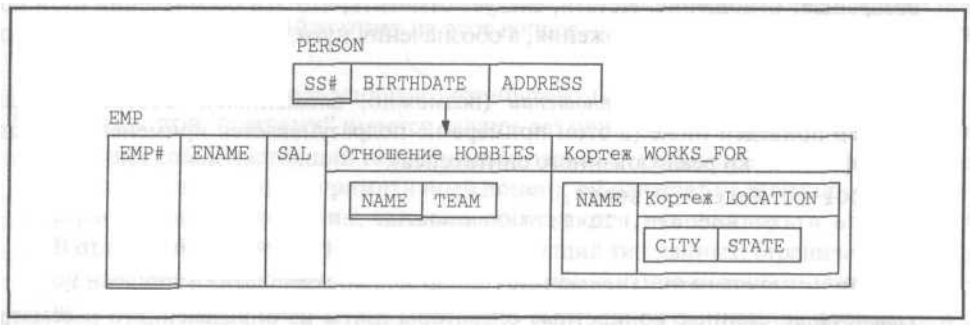
Рис. 26.2. Атрибуты, содержащие множества кортежей (указателей на кортежи), — нерекомендуемый вариант

Третье дополнение состоит в том, что в переменных отношения должно быть разрешено предусматривать связанные с ними методы (т.е. операторы), например, следующим образом.

```
CREATE TABLE EMP
      ( EMP#     CHAR(5) ,          NOT NULL,
        ENAME    CHAR (20)          NOT NULL,
        SAL      NUMERIC            NOT NULL,
        HOBBIES  SET OF ( ACTIVITY ) NOT NULL,
        WORKS FOR                     COMPANY      NOT NULL )
METHOD RETIREMENT_BENEFITS  ( ) : NUMERIC ;
```

Пояснение. Здесь RETIREMENT_BENEFITS — это метод, который принимает в качестве фактического параметра заданный кортеж EMP и вырабатывает результат типа

Рис. 26.3. Переменные отношения, которые определены как суперклассы и подклассы, — нерекомендуемый вариант



NUMERIC. Последнее дополнение в области объявления состоит в том, что должны быть разрешены подклассы, например, следующим образом (рис. 26.3):

```
CREATE TABLE PERSON
     ( SS#        CHAR(9)          NOT NULL,
       BIRTHDATE  DATE             NOT NULL,
       ADDRESS    CHAR(50)         NOT NULL ) ;

CREATE TABLE EMP AS SUBCLASS OF PERSON
     ( EMP#       CHAR(5)          NOT NULL,
       ENAME      CHAR (20)        NOT NULL,
       SAL        NUMERIC          NOT NULL,
       HOBBIES    SET OF ( ACTIVITY ) NOT NULL,
       WORKS FOR COMPANY           NOT NULL )
METHOD RETIREMENT_BENEFITS ( )  : NUMERIC ;
```

Пояснение. Теперь переменная отношения EMP имеет три дополнительных атрибута (SS#, BIRTHDATE и ADDRESS), которые унаследованы от переменной отношения PERSON (поскольку каждый экземпляр EMP "is a", т.е. представляет собой также экземпляр[4] PERSON, выражаясь неформально). Если бы таблица PERSON имела какие-либо методы, то они также могли быть унаследованы.

Примечание. Здесь PERSON и EMP представляют собой примеры того, что иногда, соответственно, именуется супертаблицами и подтаблицами. Дополнительное обсуждение (и критика) этих понятий приведено в [14.13], а также в разделе 26.6.

В соответствии с кратко описанными выше дополнениями, к определениям потребуются также некоторые дополнения, касающиеся обработки данных, например, как описано ниже.

- *Обозначения пути* (например, EMP. WORKS_FOR. LOCATION. STATE). Следует отметить, что такие выражения, вообще говоря, могут возвращать скаляры, кортежи или отношения. Заслуживает также внимания то, что в последних двух случаях

[4] Автор приносит свои извинения за то, что использует здесь такой нечеткий термин, как "экземпляр", но если бы он попытался воспользоваться точной терминологией значений и переменных, то схема, которую он пытается описать (настолько объективно, насколько это возможно), безусловно, потеряет весь сысл.

компоненты этих кортежей или отношений сами могут, в свою очередь, быть кортежами или отношениями (и т.д.); например, выражение EMP.HOBBIES.NAME возвращает отношение. Кстати, следует отметить, что эти обозначения пути проходят вниз по иерархии вложения, а обозначения пути, описанные в главе 25, проходят вверх.

- *Литералы кортежей и отношений* (возможно, вложенные). Соответствующий пример приведен ниже (в этом примере не подразумевается применение какого-либо фактически реализованного синтаксиса).

```
( 'E001', 'Smith', $50000,
        ( ( 'Soccer', 11 ), ( 'Baseball', 9 ) ),
                          ( 'IBM', ( 'San Jose', 'CA' ) ) )
```

- Реляционные операторы сравнения (например, SUBSET, SUBSETEQ и т.д.). *Примечание.* Данные конкретные операторы взяты из определенного рассматриваемого программного продукта. В этом программном продукте под оператором SUBSET фактически подразумевается оператор выделения *строгого подмножества,* а под оператором SUBSETEQ — оператор выделения *подмножества* (!?).
- Операторы для прохождения по иерархии классов (например, для одновременной выборки информации EMP и PERSON).

Примечание. Здесь также необходимо соблюдать осторожность, поскольку вполне может оказаться, что запрос на выборку информации PERSON наряду с получением соответствующей информации EMP приведет к получению результата, не являющегося отношением. Это означает, что нарушено крайне важное реляционное свойство замкнутости, а это вполне может привести с разрушительным последствиям. В этой связи в [26.41], где такой результат называется *непредсказуемым возвратом* ("jagged return"), просто приведено легкомысленное замечание, что "клиентская программа должна обладать способностью справляться со сложностями непредсказуемого возврата".

- Способность вызывать методы, например, в конструкциях SELECT и WHERE (в терминологии SQL).
- Способность обращаться к отдельным компонентам значений атрибутов, которые могут представлять собой кортежи или отношения.

На этом можно завершить краткий обзор того, как уравнение "переменная отношения = класс объекта" реализуется на практике. Так что же здесь неправильно? Прежде всего следует отметить (как было указано выше), что переменная отношения — это прежде всего переменная, а класс — это тип, поэтому в чем они могут быть аналогичными? Даже этого первого замечания должно быть с точки зрения элементарной логики достаточно для того, чтобы понять, что сама идея равенства "переменная отношения = класс объекта" является мертворожденной. Но по данному вопросу можно привести гораздо более полезные сведения, поэтому согласимся еще немного продлить это заблуждение... Ниже рассматриваются некоторые дополнительные вопросы, также заслуживающие внимания.

- Из уравнения "переменная отношения = класс объекта" следует другие уравнения: "кортеж = объект" и "атрибут = (открытая) переменная экземпляра". Итак, (как было показано в главе 25), настоящий класс объекта (по крайней мере, скалярный или *инкапсулированный* класс объекта) имеет методы и не имеет открытых

переменных экземпляра, поэтому *класс объекта* переменной отношения имеет открытые переменные экземпляра и включает методы только в качестве необязательного средства (т.е. определенно не является *инкапсулированным).* Таким образом, снова нельзя найти ответ на этот вопрос — как могут быть одинаковыми эти два понятия?

- Между двумя определениями атрибутов (например) "SAL NUMERIC" и "WORKS_FOR COMPANY" имеется важное различие. Дело в том, что NUMERIC представляет собой настоящий тип данных (в равной степени его можно рассматривать как настоящий, хотя и примитивный домен); он налагает не зависящее от времени ограничение на значения, которые допускаются использовать в атрибуте SAL. В отличие от него, COMPANY — это не настоящий тип данных; ограничение, которое он налагает на значения, разрешенные для применения в атрибуте WORKS_FOR, зависят от времени (безусловно, они зависят от текущего значения переменной отношения COMPANY). Фактически, как было указано выше, здесь не учитывается различие между переменной отношения и доменом или, если угодно, различие между коллекцией и классом.
- Выше было показано, что на первый взгляд разрешено включать в *объекты* корте жей другие такие *объекты;* например, может показаться, что *объекты* EMP включа ют *объекты* COMPANY. Но фактически в данном случае дело обстоит не так, что одни объекты включают другие объекты; вместо этого они содержат указатели на *содержащиеся в них объекты,* и пользователи должны иметь абсолютно четкое по нимание этого вопроса. Например, предположим, что пользователь каким-то об разом обновляет некоторый конкретный кортеж COMPANY (см. рис. 26.1). Тогда это обновление станет сразу же видимым во всех кортежах EMP, которые *содержат* данный кортеж COMPANY.

  Примечание. Автор не утверждает, что такой эффект нежелателен, он хочет лишь подчеркнуть, что об этом необходимо сообщить пользователю. Но сообщить об этом пользователю — значит рассказать ему о том, что "модель", показанная на рис. 26.1, является неправильной, поскольку кортежи EMP не содержат кортежи COMPANY, а содержат вместо этого указатели на кортежи COMPANY (как уже было сказано).

  Ниже приведены некоторые дополнительные выводы и вопросы, вытекающие из этого замечания.

  а) Можно ли вставить кортеж EMP и задать значение для "содержащегося" в нем кортежа COMPANY, который в настоящее время не существует в переменной отношения COMPANY? Если ответ является положительным, тот факт, что атрибут WORKS_FOR определен как принадлежащий к типу COMPANY, мало что означает, поскольку он не позволяет так или иначе в какой-то заметной степени налагать ограничения на операцию INSERT. А если ответ отрицателен, то операция INSERT становится слишком сложной, поскольку пользователь будет вынужден задавать не только имя существующей компании (т.е. значение внешнего ключа), как потребовалось бы в аналогичной ситуации с использованием реляционных объектов, но и весь существующий кортеж COMPANY. Более того, задавая весь существующий кортеж COMPANY, пользователь в лучшем случае

передает системе сведения, которые в ней уже имеются, а в худшем случае (если пользователь допускает ошибку в данных) оканчивается аварийно такая операция INSERT, которая в ином случае завершилась бы полностью успешно.

б) Предположим, что нужно ввести для данных о компаниях правило ON DELETE RESTRICT (для того чтобы попытки удалить данные о компании оканчивались неудачей, если в базе данных остается информация о сотрудниках этой компа нии). Вполне можно допустить, что это правило должно быть предписано с по мощью процедурного кода, скажем, с помощью некоторого метода M (обратите внимание на то, что переменная отношения EMP не имеет внешнего ключа, за которым можно было бы закрепить декларативную версию этого правила). Кроме того, обычные операции DELETE языка SQL теперь не должны выпол няться над переменной отношения COMPANY, кроме как с помощью кода, ко торый реализует данный метод M. Каким образом можно было бы предписать выполнение этого требования? Аналогичные замечания и вопросы, безуслов но, относятся и к другим правилам работы **с** внешними ключами, таким как ON DELETE CASCADE**.**

в) Следует также отметить, что удаление кортежа EMP может не повлечь за собой выполнение каскадной операции удаления соответствующего кортежа COMPANY, несмотря на видимое впечатление, что кортеж EMP содержит такой кортеж COMPANY.

Из всего сказанного выше следует, что, строго говоря, речь больше не идет о реляционной модели. Фундаментальный объект данных больше не является отношением, содержащим значения; его, скорее, можно назвать "отношением" в кавычках (фактически вообще не настоящим отношением в том смысле, который подразумевается реляционной моделью), содержащим значения и указатели. Иными словами, здесь **подорвана концептуальная целостность реляционной модели.** ***Примечание.*** Термин *концептуальная целостность* (conceptual integrity) был введен Фредом Бруксом (Fred Brooks), который дал ему следующее определение [26.3]: "[Концептуальная] целостность — это *наиболее* важное требование в проектировании системы. Лучше всего исключить из системы некоторые аномальные средства, чтобы воплотить в ней единый набор проектных требований, чем создавать систему, которая реализует много важных, но независимых и нескоординированных требований" (курсив оригинала). А в другой своей работе, которая вышла через 20 лет, он добавляет: "Качественный, изящный программный продукт должен представлять собой реализацию... целостной теоретической модели... Для удобства эксплуатации... наиболее важным фактором является... [концептуальная] целостность... **Теперь я в этом убежден еще больше, чем когда-либо.** Концептуальная целостность является *основой* обеспечения качества продукта" (полужирный шрифт и курсив оригинала).

■ Предположим, что определено представление V как проекция переменной отношения EMP только (скажем) по атрибуту HOBBIES. Разумеется, что V — это также переменная отношения, но производная, а не базовая. Поэтому, если уравнение "переменная отношения = класс объекта" является правильным, то V — также класс. Но какой это класс? К тому же классы имеют методы; какие методы применяются к представлению V?

Очевидно, что "класс" EMP имеет только один метод, RETIREMENT_BENEFITS, и этот метод, безусловно, не применим к "классу" V. Фактически, вряд ли можно назвать разумным такое предположение, что какие-либо методы, применимые к "классу" EMP, будут относиться и к "классу" V; а то же самое, безусловно, касается других представлений. Поэтому (в целом) создается впечатление, что к результатам проекции не могут применяться вообще какие-либо методы; это означает, что результат проекции, каким бы он ни был, в действительности вообще не является классом. (Мы могли бы, разумеется, просто "назвать" его классом, но от этого он не стал бы таковым! Дело в том, что это был бы класс, имеющий открытые переменные экземпляра и не имеющий методов, притом что уже было отмечено, что настоящий *инкапсулированный* класс имеет методы и не имеет открытых переменных экземпляра.) Вполне очевидно, что фактически те специалисты, которые ставят знак равенства между переменными отношения и классами, имеют в виду именно базовые переменные отношения, но забывают о производных переменных отношения. (Безусловно, указатели, описанные выше, представляют собой указатели на кортежи в базовых, а не в производных переменных отношения.) Но проведение различия между базовыми и производными переменными отношения указанным образом является ошибкой высшего порядка, поскольку задача определения того, какая переменная отношения является базовой, а какая производной (и это очень важно), решается полностью произвольно, в зависимости от потребностей конкретного приложения (как было сказано в описании *принципа взаимозаменяемости* в главе 10). ■ Наконец, какие домены поддерживаются? По-видимому, те, кто считает правильным уравнение "переменная отношения = класс объекта", стараются в основном избегать обсуждения вопроса о доменах, скорее всего, по той причине, что не могут определить, каким образом домены как таковые вписываются в разработанную ими общую схему. Тем не менее, как известно, понятие доменов является одним из существенных.

Общий итог сказанного выше можно подвести следующим образом. Очевидно, что системы, основанные на неправильном уравнении "переменная отношения = класс объекта", могут быть созданы, и действительно некоторые подобные системы уже существуют. Несомненно также, что эти системы в течение какого-то времени могут даже предоставлять полезные услуги (как и автомобиль без масла в двигателе или дом, построенный на песке), но они обречены на бесконечные аварии и отказы.

Анализ источников возникновения первого серьезного заблуждения

Любопытно поразмыслить над тем, с чем связано возникновение первого серьезного заблуждения. По мнению автора, его корни лежат в том, что в сообществе специалистов в области объектно-ориентированных систем до сих пор нет согласия по поводу толкования некоторых терминов (как было отмечено в главе 25). В частности, сам термин *объект* не имеет общепризнанного и согласованного толкования; именно поэтому автор данной книги предпочитает его не использовать. •■ Несмотря на приведенные выше замечания, по крайней мере, вполне очевидно, что в кругах пользователей объектными языками программирования термином *объект* во всяком случае обозначается то понятие, которое с использованием более традиционной

терминологии следовало бы называть либо значением, либо переменной (а возможно, и оба эти понятия). Но, к сожалению, этот термин применяется также специалистами в некоторых других областях, в частности, он встречается в определенных областях семантического моделирования в составе всевозможных методов и методологий *объектного анализа и проектирования* или *объектного моделирования* (см., например, [14.3]). А в этих областях, по-видимому, данный термин трактуется не как значение или переменная, а скорее как то, что в сообществе пользователей баз данных чаще всего принято называть сущностью (кстати, под этим, кроме всего прочего, подразумевается, что, в отличие от объектов из языков программирования, объекты, называемые *сущностями,* определенно не инкапсулированы). Иными словами, так называемое *объектное моделирование* в действительности представляет собой просто *моделирование сущностей/связей* под другим названием; фактически в [14.3] в той или иной степени признается справедливость этого утверждения. Вследствие этого, информационные структуры, выявленные с помощью этих методологий как *объекты,* затем (вполне обоснованно) отображаются на кортежи в переменных отношения, а не на значения в доменах. При этом возникает невероятная путаница!

26.3. ВТОРОЕ СЕРЬЕЗНОЕ ЗАБЛУЖДЕНИЕ

В этом разделе рассматривается **второе серьезное заблуждение;** как будет видно из дальнейшего изложения, это второе заблуждение является логическим следствием из первого, но оно также является значимым само по себе. В действительности, даже если удается избежать **первого серьезного заблуждения,** то можно стать жертвой второго, отдельно взятого. И следствия этого заблуждения фактически обнаруживаются почти во всех объектно-реляционных программных продуктах, имеющихся на рынке, а также в стандарте SQL (см. раздел 26.6). Это заблуждение заключается в том, что **смешиваются указатели и отношения.**

Начнем с повторного рассмотрения основных особенностей подхода, основанного на уравнении "переменная отношения = класс объекта", который описан в предыдущем разделе. Нельзя отрицать, что некоторые читатели могли посчитать этот раздел немного запутанным, поскольку на первый взгляд в нем приведены возражения против тех средств, в пользу которых автор высказывался в других главах этой книги (таковыми можно назвать атрибуты со значениями в виде кортежа или в виде отношения). Поэтому автор должен разъяснить свою позицию, как описано ниже.

- *Атрибуты со значениями в виде кортежа или в виде отношения.* Безусловно, автор не возражает против таких атрибутов (и на каких основаниях он стал бы против них возражать?). В действительности, он возражает, во-первых, против той идеи, что такие атрибуты должны иметь именно те значения, которые в настоящее время присутствуют в некоторой другой (базовой) переменной отношения, и, во-вторых, против идеи, чтобы такие атрибуты по существу имели значения, которые являются не кортежами или отношениями как таковыми, а скорее указателями на кортежи или отношения (а это означает, что на самом деле речь вообще не идет об атрибутах со значениями в виде кортежей или в виде отношений). ***Примечание.*** На самом деле идея использования указателей, которые ссылаются на *кортежи или отношения* (а под этим подразумеваются именно значения кортежей или отношений) вообще не имеет смысла; эта тема обсуждается более подробно немного позже.

- *Связывание операторов ("методов") с переменными отношения.* Автор не возражает также и против этой идеи; она практически просто определяет понятие триггеров (или хранимых процедур) в другой формулировке. Но автор фактически возражает против идеи, что такие операторы должны быть связаны с переменными отношения (и только переменными отношения), но не с доменами или типами. Он также возражает против идеи, что они должны быть связаны лишь с одной определенной переменной отношения (поскольку тем самым вводится понятие целевых фактических параметров, но в другой формулировке).
- *Подклассы и суперклассы.* Против этого автор, безусловно, возражает. В той системе, в которой проводится знак равенства между переменными отношения и классами, подклассы и суперклассы становятся подтаблицами и супертаблицами, а к этим понятием автор относится чрезвычайно скептически (см. [14.13], а также раздел 26.6). Безусловно, должно обеспечиваться правильное наследование типов, как описано в главе 20.

  Обозначения пути. Автор не возражает против таких обозначений пути, которые являются просто синтаксическими сокращениями для операций прохождения по некоторым ассоциативным ссылкам, например, от внешнего ключа к соответствующему потенциальному ключу, как было предложено в [26.15]. Но обозначения пути, которые рассматривались в разделе 26.2, скорее, представляют собой сокращения для операций прохождения по некоторым цепочкам указателей, и автор против этого возражает (поскольку он прежде всего возражает против применения указателей на уровне модели).
- *Литералы кортежей и отношений.* Эти средства являются важными, но их необходимо обобщить до уровня селекторов кортежей и отношений [3.3].
- *Реляционные операторы сравнения.* Эти средства также важны (хотя должны быть реализованы правильно).
- *Операторы для прохождения по иерархии классов.* Если под *иерархией классов* в действительности подразумевается *иерархия переменных отношения,* то (как отмечено в предыдущем разделе) автор имеет серьезные возражения, поскольку возникает вероятность нарушения реляционного свойства замкнутости (см. например [26.41]). Если бы понятие *иерархия классов* означало *иерархию типов* в том смысле, который указан в главе 20, то автор не имел бы возражений (но, к сожалению, дело обстоит иначе).
- *Вызов методов,* например, в конструкциях SELECT и WHERE. Никаких возражений.
- *Доступ к отдельным компонентам в значениях атрибутов, которые оказались кортежами или отношениями.* Никаких возражений.

Итак, теперь сосредоточимся на проблеме смешивания указателей и отношений. В основе доводов автора лежит очень простая мысль. По определению, указатели указывают на переменные, а не на значения (поскольку переменные имеют адреса, а значения — нет). Поэтому, опять-таки по определению, если допускается, чтобы переменная отношения R1 имела атрибут, значениями которого являются указатели, *направленные в* переменную отношения R2, то эти указатели указывают на переменные кортежей, а не на значения кортежей. Но в реляционной модели не определено понятие переменной кортежа.

Реляционная модель распространяется на значения отношений, которые представляют собой (выражаясь неформально) множества значений кортежей, которые, в свою очередь (снова выражаясь неформально), являются множествами скалярных значений. Кроме того, реляционная модель распространяется на переменные отношения, представляющие собой переменные, значениями которых являются отношения. Но реляционная модель не распространяется на переменные кортежей (представляющие собой переменные, значениями которых являются кортежи) или на скалярные переменные (представляющие собой переменные, значениями которых являются скаляры). Единственным видом переменной, включенной в реляционную модель (и единственным видом переменной, разрешенной для использования в реляционной базе данных), является именно переменная отношения. Из этого следует, что *идея смешивания указателей и отношений представляет собой серьезнейшее отклонение от реляционной модели, поскольку она требует введения переменной полностью нового вида* (и тем самым фактически нарушения информационного принципа). Как отмечено в предыдущем разделе, в действительности можно смело утверждать, что при этом происходит также серьезное нарушение концептуальной целостности реляционной модели.

Поскольку невозможно отрицать справедливость сказанного выше, очень грустно наблюдать за тем, что в большинстве (а возможно во всех) современных образцах объектно-реляционных программных продуктов (даже в тех, в которых удалось избежать **первого серьезного заблуждения),** тем не менее, допускается смешивание указателей и отношений именно в той форме, которая была описана и против которой были приведены возражения в предыдущем разделе. Когда Кодд впервые определил реляционную модель, он вполне сознательно исключил указатели. Ниже приведена цитата из [6.2].

> *Вполне допустимо принять предположение, что пользователи всех категорий [включая, в частности, конечных пользователей] понимают суть операции сравнения значений, но лишь относительно немногие понимают все сложности, связанные с применением указателей. Реляционная модель основана на фундаментальном принципе сравнения значений... [А] операции манипулирования с указателями в большей степени могут служить источником ошибок, чем операции сравнения значений, даже если пользователь учитывает все нюансы работы с указателями.*

А именно, применение указателей приводит к тому, что из одного указателя рождаются цепочки указателей, а использование цепочек указателей является, по общему признанию, источником неизбежных ошибок. Как отмечалось в главе 25, именно эта характеристика объектных систем стала причиной возникающих время от времени критических замечаний, которые сводятся к тому, что подобные системы "напоминают слегка обновленную модель CODASYL".

Многочисленные аргументированные доводы в поддержку изложенной выше позиции автора можно найти в [25.19] и [26.15]. См. также **[26.12]—[26.14]** и [26.17], где рассматривается связанное с этим и очень важное *понятнесущественности* (essentiality).

Несовместимость указателей и качественной модели наследования

В данном разделе фактически приведен еще один обоснованный довод против поддержки указателей, о котором Кодд, возможно, не знал, когда готовил работу [6.2]. Проиллюстрируем этот довод на очень простом примере. (Данный пример сформулирован в терминах обычных программных переменных, а не отношений базы данных, чтобы можно

было сосредоточиться на реальной проблеме, а не отвлекаться на ненужные подробности.) Еще раз рассмотрим типы ELLIPSE и CIRCLE. Допустим, что PTR_TO_ELLIPSE и PTR_TO_CIRCLE — это соответствующие типы указателей; иными словами, допустим, что значения типов PTR_TO_ELLIPSE и PTR_TO_CIRCLE (выражаясь неформально) являются, соответственно, *указателями на эллипсы* и *указателями на окружности.* Наконец, допустим, что ELLIPSE — строгий супертип типа CIRCLE и поэтому примем предположение, что PTR_TO_ELLIPSE — строгий супертип типа PTR_TO_CIRCLE. Теперь рассмотрим следующий фрагмент кода.

```
VAR E ELLIPSE ;

VAR XC PTR_TO_CIRCLE ;

E := CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) ;

XC := TREAT_DOWN_AS_PTR_TO_CIRCLE ( PTR_TO ( E ) ) ;

THE_A ( E ) := LENGTH ( 6.0 ) ;
```

Пояснение

1. Два объявления переменных говорят сами за себя.
2. После первого присваивания (переменной E) эта переменная содержит окружность с радиусом пять[5]. Обратите внимание на то, что в этом операторе присваивания используется свойство заменяемости; следует также отметить, что теперь наиболее конкретным типом переменной E является CIRCLE.
3. Оператор PTR_TO, который присутствует в выражении справа от второго оператора присваивания, представляет собой то, что обычно называется оператором **адресации:** если дана переменная V, он возвращает адрес переменной V (т.е. указатель на эту переменную). Поэтому в рассматриваемом примере данный оператор возвращает значение указателя типа PTR_TO_ELLIPSE. Но поскольку переменная, на которую указывает значение данного указателя, имеет наиболее конкретный тип CIRCLE, то значение указателя фактически имеет тип PTR_TO_CIRCLE, а не просто тип PTR_TO_ELLIPSE. Таким образом, операция приведения к подклассу TREAT DOWN завершается успешно и поэтому во второй операции присваивания в переменную xc помещается указатель на переменную E (иными словами, адрес этой переменной).
4. Третья операция присваивания значения (псевдопеременной THE_A (E)) является наиболее важной. Что происходит при ее выполнении? Скорее всего, здесь могут быть три возможности, которые описаны ниже, и все они не совсем приемлемы.
   а) Возникает ошибка на этапе прогона из-за несоответствия типов, поскольку присваивание псевдопеременной THE_A не поддерживается для переменных наиболее конкретного типа CIRCLE. Иными словами, не поддерживается обобщение с помощью ограничения, поэтому не поддерживается также уточнение с помощью ограничения, а это означает, что модель наследования неприемлема

[5] Как и в главе 20, в данной главе предполагается, что декартово возможное представление для точек называется POINT, а не CARTESIAN. Поэтому в данном примере вторым фактическим параметром селектора CIRCLE является вызов соответствующего селектора POINT.

(безусловно, ее нельзя назвать *достоверной моделью реальности,* как было описано в главе 20). Так или иначе, возникающие на этапе прогона ошибки из-за несоответствия типов всегда нежелательны.

б) Операция присваивания вроде бы завершается "успешно", но обобщение с помощью ограничения не происходит. Результатом этого становится то, что переменная E теперь содержит "некруглую окружность" (ее наиболее конкрет ным типом все еще остается CIRCLE, но эта "окружность" имеет *полуоси* раз ной длины). Поэтому, опять-таки, модель наследования является неприемлемой (и не может рассматриваться как достоверная модель реальности), поскольку обобщение с помощью ограничения и уточнение с помощью ограничения не поддерживаются. Более того, не только переменная E теперь содержит "некруг лую окружность", но и переменная xc указывает также на "некруглую окруж ность". Мало того, не могут поддерживаться ограничения типа! Ведь если бы они поддерживались, то не могли бы возникнуть "некруглые окружности". Иными словами, можно смело утверждать, что исключена возможность под держки ограничения целостности наиболее фундаментального вида: определяя тип, мы не можем указать, какие значения этого типа являются допустимыми.

Кстати, следует отметить, что эта ошибка допущена и в спецификации SQL: 1999! См. главу 20 и раздел 26.6.

в) Операция присваивания завершается успешно и происходит обобщение с по мощью ограничения; это означает, что переменная E теперь содержит "просто эллипс" и ее наиболее конкретным типом служит ELLIPSE. Но и тогда модель наследования является неприемлемой, поскольку переменная xc, объявлен ным типом которой служит PTR_TO_CIRCLE, теперь содержит значение наи более конкретного типа PTR_TO_ELLIPSE! Ид этого снова следует, что ограни чения типа не могут поддерживаться.

Примечание. Даже сама идея о том, что переменной некоторого объявленного типа T может быть разрешено содержать значение наиболее конкретного типа некоторого строгого супертипа t, является логической бессмыслицей, поэтому более вероятно, что либо первая операция присваивания завершится неудачей, как в п. а), либо внешне завершится "успешно" без обобщения с помощью ограничения, как в п. б). Иными словами, вариант, описанный в этом пункте, возможно, вообще является неосуществимым.

Из этого примера следует, что если поддерживаются указатели, то модель наследования обязательно становится неприемлемой; иными словами, указатели и качественная модель наследования несовместимы. По-видимому, эти соображения могут служить еще одним неоспоримым доводом в пользу отказа от указателей.

Анализ причин возникновения второго серьезного заблуждения

В литературе трудно обнаружить причины, по которым возникло второе серьезное заблуждение (вернее, отсутствуют какие-либо теоретические обоснования поддержки указателей, но есть свидетельства, что решение об их применении было принято вообще не по теоретическим соображениям, а скорее на основании административного решения). Учитывая тот факт, что все объектные системы и объектные языки предусматривают

использование указателей в форме идентификаторов объектов, идея смешивания указателей и отношений почти наверняка стала результатом стремления сделать реляционные системы более "похожими на объектные". Но такое "обоснование" просто переводит указанную проблему на другой уровень; автор уже привел исчерпывающие доводы в пользу сложившегося у него мнения о том, что в объектных системах пользователю предоставляется доступ к указателям именно потому, что в них не проводятся должным образом различия между моделью и реализацией.

Поэтому автор может сделать единственно допустимое предположение, что причиной, по которой идея смешивания указателей и отношений получила столь широкое распространение, является именно то, что лишь немногие понимают, почему из реляционной модели были с самого начала исключены указатели. Джон Сантаяна (John Santayana) сформулировал это так: "Те, кто не способен помнить прошлое, обречены его повторять" (обычно цитируется в форме: "Для тех, кто не знает истории, она повторяется"). По этому поводу автор выражает полное согласие с Морисом Уилксом (Maurice Wilkes), которому принадлежит приведенная ниже цитата [26.46].

> *"Я хотел бы, чтобы обучение компьютерным наукам специально проводилось в историческом контексте... Студенты должен понимать, как сложилась современная ситуация, какие предпринимались попытки, что оказалось удачным, а что — нет, и какие усовершенствования в аппаратных средствах стали предпосылкой прогресса. Отсутствие в обучении такой составляющей заставляет людей подходить к решению любой задачи исходя из самых простейших представлений. Поэтому они становятся склонными предлагать решения, от которых уже давно отказались в прошлом. Вместо того чтобы встать на плечи своих предшественников, они пытаются самостоятельно подняться над горизонтом".*

26.4. ПРОБЛЕМЫ РЕАЛИЗАЦИИ

Одним важным следствием обеспечения правильной поддержки типов данных является то, что она позволяет независимым поставщикам (а также поставщикам самих СУБД) создавать и поставлять отдельные *пакеты типов,* которые могут эффективно встраиваться в СУБД. В качестве примеров можно назвать пакеты, поддерживающие сложные алгоритмы обработки текста, обработки финансовых временных рядов, анализа геопространственных (картографических) данных и т.д. Такие пакеты называются по-разному: *пластинами данных*— data blade (Informix), *картриджами данных*— data cartridge (Oracle), *реляционными расширителями* — relational extender[6] (IBM), *прикладными пакетами* — application package (это термин используется в стандарте SQL/MM [26.25]) и т.д. В данном разделе остановимся на термине *пакеты типов.*

Но введение нового пакета типов в систему — нетривиальное мероприятие, и для предоставления такой возможности необходимо внести существенные изменения в проект и структуру самой СУБД. Для того чтобы всесторонне разобраться в этом вопросе, рассмотрим, что произойдет, если, например, некоторый запрос включает ссылки на данные какого-то определяемого пользователем типа или вызовы какого-то определяемого пользователем оператора (или то и другое).

---

[6] По мнению автора, этот термин является исключительно неудачным.

и Во-первых, компилятор языка запросов должен обладать способностью проводить синтаксический анализ и проверку типов в соответствующем запросе, поэтому должен иметь определенную информацию об этих определяемых пользователем типах и операторах.

- Во-вторых, оптимизатор должен обладать способностью принять подходящий план обработки такого запроса, поэтому он должен также иметь сведения о конкретных свойствах таких определяемых пользователем типов и операторов. В частности, он должен иметь сведения о том, как организовано физическое хранение данных (см. следующий пункт).
- В-третьих, компонент, который управляет физической памятью, должен иметь поддержку соответствующих более новых структур хранения (квадрадеревьев, R-деревьев и т.д.), которые упоминались при обсуждении задачи с прямоугольни ками в разделе 26.1. Может даже потребоваться, чтобы этот компонент позволял пользователям, обладающим соответствующими навыками, самостоятельно вво дить новые структуры хранения и методы доступа [26.29], [26.43].

Одним важным следствием из всего сказанного выше является то, что система обязательно должна быть расширяемой, а фактически способность к расширению должна обеспечиваться на нескольких уровнях. Ниже кратко описан каждый из уровней.

Синтаксический анализ и проверка типов

Поскольку в обычной системе все единственно доступные типы и операторы являются встроенными, относящуюся к ним информацию можно "жестко закодировать" в компиляторе (что обычно и делается). В отличие от этого, в системе, в которой пользователи могут определять свои собственные типы и операторы, такой подход с "жестким кодированием", безусловно, неприемлем. Таким образом, вместо этого необходимо применять решения, подобные приведенным ниже.

- Информация, касающаяся определяемых пользователем типов и операторов (и возможно также встроенных типов и операторов) хранится в системном каталоге. Из этого следует, что сам каталог требует перепроектирования (или, по крайней мере, расширения); из этого также следует, что введение каждого нового пакета типов влечет за собой существенное обновление каталога. (В терминах языка Tutorial D такое обновление незаметно для пользователя осуществляется в составе процесса выпелнения соответствующих определительных предложений TYPE и OPERATOR.)
- Необходимо переработать сам компилятор для того, чтобы он мог обращаться к каталогу для получения необходимой информации о типах и операторах. Затем он может использовать эту информацию для проведения всей той проверки типов на этапе компиляции, которая была описана в главах 5 и 20.

Оптимизация

Решение этой задачи связано с преодолением чрезвычайно многочисленных проблем, и в данной книге мы можем лишь привести их краткое описание. Но, по крайней мере, необходимо остановиться на некоторых перечисленных ниже вопросах.

- *Преобразование выражения (перезапись запроса).* Как было показано в главе 18, для перезаписи запроса в обычном оптимизаторе применяются некоторые законы преобразования. Но по традиции такие законы преобразования всегда были "жестко закодированы" в оптимизаторе (поскольку, опять-таки, все доступные типы данных и операторы были встроенными). В отличие от этого, в объектно-реляционной системе соответствующие сведения (по крайней мере, та их часть, которая относится именно к определяемым пользователем типам и операторам), должны храниться в каталоге; из этого следует, что при расширении каталога не обходимо учитывать и этот факт, кроме того, налицо такое следствие, что требует доработки и сам оптимизатор. Ниже приведены некоторые иллюстрации к ска занному.

  а) Если дано такое выражение, как NOT (STATUS > 20), то качественный обычный оптимизатор преобразует его в STATUS > 20 (поскольку во второй версии можно воспользоваться индексом на столбце STATUS, а в первой — нельзя). По аналогичным причинам нужен способ, с помощью которого можно было бы сообщить оптимизатору, что один определяемый пользователем опе ратор является отрицанием другого.

  б) Качественный обычный оптимизатор имеет также информацию о том, что, на пример, выражения STATUS > 20 и 20 < STATUS логически эквивалентны. Поэтому должен быть предусмотрен способ передачи оптимизатору информа ции о том, что два определяемых пользователем оператора являются противо положными друг другу в указанном смысле.

  в) Кроме того, качественный обычный оптимизатор имеет информацию о том, что, например, операторы "+" и "-" взаимно компенсируют друг друга (т.е. являются обратными по отношению друг к другу); например, выражение STATUS + 20 - 20 можно сократить просто до STATUS. Должен быть преду смотрен способ передачи оптимизатору сведений о том, что два определяемых пользователем оператора являются обратными по отношению друг к другу, в указанном смысле.

- *Избирательность.* Если дано логическое выражение, такое как STATUS > 20, то оптимизаторы обычно принимают предположение об избирательности этого вы ражения (т.е. о том, какова процентная доля кортежей, при обработке которых это выражение примет значение TRUE). В отношении встроенных типов данных и операторов эта информация об избирательности также может быть "жестко зако дирована" в оптимизаторе; в отличие от этого, применительно к определяемым пользователем типам и операторам должен быть предусмотрен способ предостав ления оптимизатору некоторого определяемого пользователем кода, который он мог бы вызвать, чтобы выработать предположение об избирательности.

- *Определение стоимости.* Оптимизатор должен иметь информацию о том, сколько будет стоить выполнение конкретного определяемого пользователем оператора. Например, если дано такое выражение, как p AND q, где p — (скажем) вызов опе ратора AREA для определения площади какого-то сложного многоугольника, а q — простое сравнение наподобие STATUS > 20, то, вероятно, следовало бы предпо честь, чтобы система вначале выполнила выражение q так, чтобы выражение p

выполнилось только на кортежах, для который q принимает значение TRUE. Безусловно, некоторые эвристические методы преобразования выражений, ставшие классическими (в частности, такой метод, который всегда предусматривает выполнение операторов сокращения перед операциями соединения), не обязательно будут действительными для определяемых пользователем типов и операторов (см., например, [26.10] и [26.24]).

- *Структуры хранения и методы доступа.* Оптимизатор, безусловно, должен иметь информацию о применяемых структурах хранения и методах доступа (см. следующий подраздел).

Структуры хранения данных

Вполне очевидно, что для объектно-реляционных систем требуется больше способов хранения и доступа к данным на физическом уровне (а по всей видимости, даже намного больше таких способов), чем обычно было предусмотрено (например) в системах SQL. Ниже приведены некоторые соображения поданному вопросу.

- *Новые структуры хранения.* Как уже отмечалось, для системы, по-видимому, потребуется поддержка новых "жестко закодированных" структур хранения (R-деревья и т.д.) и может даже потребоваться способ, который позволял бы пользователям, обладающим соответствующей квалификацией, самостоятельно определять дополнительные структуры хранения и методы доступа.
- *Индексы на данных определяемого пользователем типа.* Традиционные индексы основаны на данных определенного встроенного типа, а также на встроенной информации о том, что означает оператор "<". В объектно-реляционных системах должна быть предусмотрена возможность создавать индексы на данных определяемого пользователем типа исходя из семантики применимого к ним и определяемого пользователем оператора "<" (в первую очередь, безусловно, для этого предполагается, что подобный оператор уже определен).
- *Индексы на результатах выполнения операций.* По-видимому, не имеет смысла создавать индексы непосредственно на множестве значений данных, например, типа POLYGON, поскольку более вероятно то, что такие индексы будут служить для упорядочения соответствующих многоугольников по их внутренним закодированным представлениям в виде строки байтов. Но индекс, основанный на значениях площадей этих многоугольников, мог бы оказаться весьма полезным.

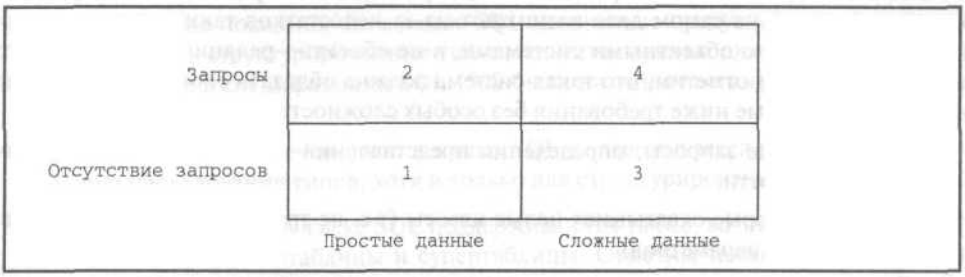
Примечание. В главе 22 такие индексы упоминались под названием *функциональных индексов.*

26.5. ПРЕИМУЩЕСТВА ПОДЛИННОГО СБЛИЖЕНИЯ ТЕХНОЛОГИЙ

В [26.41] Стоунбрейкер (Stonebraker) представил *матрицу классификации* для СУБД (рис. 26.4). Квадрант 1 этой матрицы представляет приложения, в которых применяются только относительно простые данные и не предъявляются требования по выполнению произвольных запросов (хорошим примером подобного приложения может служить обычный текстовой процессор). Такие приложения в действительности вообще нельзя назвать приложениями базы данных в обычном смысле этого термина; так сказать,

Рис. 26.4. Матрица классификации СУБД, предложенная Стоунбрейкером

"СУБД", которая наилучшим образом обслуживает их потребности, представляет собой просто встроенный диспетчер файлов, предоставляемый в составе базовой операционной системы.



В квадранте 2 представлены приложения, в которых предъявляются требования по выполнению произвольных запросов, но используются все еще относительно простые данные. В этот квадрант попадает большинство современных производственных приложений, и эти приложения вполне успешно поддерживаются традиционными реляционными СУБД (или, по крайней мере, системами SQL).

В квадранте 3 представлены приложения, в которых требуются сложные данные и алгоритмы обработки, но нет необходимости в использовании произвольных запросов. Например, к этому квадранту могут относиться приложения систем автоматизированного проектирования и производства. Современные объектные СУБД в основном предназначены для использования в этом сегменте рынка (традиционные программные продукты SQL, как правило, плохо подходят для приложений, которые относятся к квадранту 3).

Наконец, в квадранте 4 представлены приложения, для которых требуются одновременно и сложные данные, и произвольные запросы к этим данным. Стоунбрейкер приводит пример базы данных, содержащей оцифрованные 35-миллиметровые слайды, **в** которой типичный запрос выглядит так: "Найти снимки солнечных закатов, полученные в 20 милях от Сакраменто, штат Калифорния". Затем он приводит доводы в поддержку своей позиции, согласно которой, во-первых, для приложений, относящихся к этому квадранту, требуется объектно-реляционная СУБД, и, во-вторых, в течение ближайших нескольких лет большинство новых приложений будет относиться к этому квадранту, а большинство старых приложений в него перейдет. Например, может потребоваться доработать даже простые приложения для отдела кадров, чтобы предусмотреть использование в них фотографий служащих, звукозаписей (речевых сообщений) и тому подобных данных.

В конечном итоге Стоунбрейкер утверждает (с чем автор полностью согласен), что "объектно-реляционные системы в будущем затронут буквально каждого"; их нельзя рассматривать просто как модное увлечение, которое вскоре сменится какой-то другой темой, овладевшей на время вниманием публики. Но автор должен напомнить, что если дело касается специалистов в области баз данных, то настоящая объектно-реляционная система — это не больше и не меньше, чем настоящая *реляционная* система. В частности, это — такая система, в которой нет следов ни одного из **двух серьезных заблуждений!** По-видимому, Стоунбрейкер не полностью соглашается с изложенной здесь позицией автора; по крайней мере, в [26.41] не выражена явно ее поддержка и фактически из

сказанного в этой работе следует, что смешивание указателей и отношений не только приемлемо, но и желательно (и, в действительности, якобы даже требуется).

Как бы то ни было, автор настоящей книги утверждает, что подлинная объектно-реляционная система позволит решить все те проблемы, которые (как было указано в предыдущей главе) на самом деле возникают из-за недостатков таких систем, которые являются лишь чисто объектными системами, а не объектно-реляционными. Для уточнения этой позиции отметим, что такая система должна обладать способностью поддерживать все описанные ниже требования без особых сложностей.

- Произвольные запросы, определения представлений **и** декларативные ограниче ния целостности.
- Методы, которые охватывают целые классы (т.е. не требуют различимых целевых фактических параметров).
- Динамически определяемые классы (для результатов произвольных запросов).
- Двухрежимный доступ (см. главу 4; эта мысль в главе 25 не подчеркивалась, но объектные системы обычно не поддерживают принцип двухрежимности — вместо этого в них используются разные языки для программируемого и интерактивного доступа к базе данных).
- Переходные ограничения.
- Семантическая оптимизация.
- Связи со степенью больше двух.
- Правила внешних ключей (ON DELETE CASCADE И Т.Д.).
- Оптимизируемость.

•*I*

Можно также назвать и другие требования. Кроме того, можно привести изложенные ниже соображения в пользу позиции автора.

- Идентификаторы объектов и цепочки указателей теперь полностью "включены в реализацию" и скрыты от пользователя.
- Устраняются "сложные" объектные проблемы (например, не приходится искать ответ на вопрос о том, что подразумевается под соединением двух объектов).
- Преимущества инкапсуляции как таковые все еще сохраняются, но распростра-
] няются на скалярные значения в отношениях, а не на сами отношения.
- Как было указано выше, реляционные системы теперь способны поддерживать такие "сложные" прикладные области, как системы автоматизированного проек тирования и производства.

Кроме того, предлагаемый подход полностью обоснован теоретически.

26.6. СРЕДСТВА SQL

Наиболее очевидные и существенные различия между спецификацией **SQL:** 1999 и предшествующей ей спецификацией SQL: 1992 заключаются в том, что в новой спецификации рассматриваются объектно-реляционные средства. Многие из этих средств, которые кратко перечислены ниже, были уже описаны и проанализированы в предыдущих главах.

- В главе 5 было показано, что язык SQL поддерживает две разные категории опре деляемых пользователем типов, типы DISTINCT и структурированные типы. Обе эти разновидности могут использоваться (кроме всего прочего) в качестве основы для определения столбцов в базовых таблицах.
- В главе 6 было показано, что в языке SQL специально предусмотрена возможность использовать структурированные типы в качестве основы для определения струк тур, которые в спецификации SQL: 1999 были названы *типизированными таблица ми* (typed table).
- В главе 20 было отмечено, что в языке SQL поддерживается также определенная форма наследования типов, хотя и только для структурированных типов.

Но кроме этих средств, в языке SQL поддерживаются также, во-первых, генератор типа REF[7], и, во-вторых, подтаблицы и супертаблицы. Отметим также, что эти дополнительные конструкции в очень значительной степени связаны с уже указанными (особенно со структурированными типами). Хотя и не совсем ясно, почему они должны были быть настолько связанными (в принципе, понятия, лежащие в их основе, полностью ортогональны), но, возможно, все равно это не имеет большого значения, поскольку, по мнению автора, указанные в начале этого абзаца дополнительные конструкции вообще не очень-то нужны, как будет показано ниже.

Типы REF

Начнем с простого примера (ненужные подробности здесь не показаны).

```
CREATE TYPE
  DEPT TYPE AS (
  DEPT#  CHAR(3),
       DNAME  CHAR(25),
       BUDGET MONEY ) ...
  REF IS SYSTEM GENERATED ;

CREATE TABLE DEPT OF DEPT TYPE
     ( REF IS DEPT ID SYSTEM GENERATED,
       PRIMARY KEY ( DEPT# ) ) ... ;
```

Пояснение (частично здесь повторяется материал, приведенный в главе 6).

1. Вначале напомним, что (как было сказано в главе 5) при создании каждого структурированного типа ST система автоматически формирует соответствующий ссылочный тип ("тип REF"), называемый REF(ST), поэтому в данном примере автоматически формируется ссылочный тип REF(DEPT_TYPE) . Типы REF могут использоваться везде, где допустимо применение типа данных любого рода; но они могут формироваться только неявно в качестве побочного эффекта создания структурированного типа.

---

[7] Ключевое слово REF является сокращением от reference (ссылка), но типы REF не имеют ничего общего ни с привилегиями REFERENCES (см. главу 17), ни с теми ссылками, под которыми подразумеваются внешние ключи. Кстати, следует отметить, что типы REF являются скалярными, поэтому генератор типа REF представляет собой пример генератора скалярного типа.

2. Значения типа REF (ST) представляют собой *ссылки* на строки (иными словами, указатели на строки или адреса строк) в некоторой базовой таблице[8], которая была определена (с помощью ключевого слова "OF") как относящаяся к типу ST (см. п. 4). Поэтому в данном примере значения типа REF (DEPT_TYPE) представляют собой указатели на строки в базовой таблице DEPT. (Здесь предполагается, что DEPT — это единственная таблица, которая определена с помощью ключевого слова "OF" как относящаяся к типу DEPT_TYPE, хотя это предположение не всегда будет действительным.)

   Примечание. Структурированный тип ST, разумеется, может использоваться и в других контекстах (например, в качестве объявленного типа для некоторого столбца или некоторой локальной переменной), но с этими другими областями применения не ассоциируется значение REF (ST).

3. Спецификация REF IS SYSTEM GENERATED в предложении CREATE TYPE указывает, что значения соответствующего типа REF предоставляются системой. (Есть и другие опции, например, REF IS USER GENERATED, но сведения о них выходят за рамки данной книги.)

   Примечание. В действительности, опция REF IS SYSTEM GENERATED применяется по умолчанию; поэтому в данном примере можно было бы полностью исключить эту спецификацию из определения типаOEPT_TYPE.

4. Базовая таблица может быть определена (в предложении CREATE TABLE) как относящаяся к некоторому структурированному типу с помощью ключевого слова "OF"; такая таблица называется *типизированной таблицей* (typed table) или *указываемой в ссылках таблицей* (referenceable table). Но в действительности ключевое слово OF в данном случае не совсем подходит, поскольку (как описано в главе 6) данная таблица фактически не относится к рассматриваемому типу (что обычно выражает предлог "of"); к этому типу не относятся и ее строки. На самом деле в этой таблице предусмотрено по одному столбцу для каждого атрибута рассматриваемого структурированного типа, а также один дополнительный столбец (а именно, столбец применимого типа REF), хотя синтаксис для определения данного дополнительного столбца не напоминает обычный синтаксис определения столбца, а вместо этого выглядит примерно следующим образом.

   REF IS *<column name>* SYSTEM GENERATED

   Этот лишний *ссылающийся на самого себя* столбец с именем *<column name>,* который является первым в упорядочении столбцов таблицы слева направо, используется для хранения уникальных идентификаторов *(ссылок)* на строки рассматриваемой базовой таблицы (из этого следует также, что он имеет спецификации UNIQUE и NOT NULL). Идентификатор для данной конкретной строки

[8] Или, возможно, в некотором представлении. Описание того случая, когда значения типа REF ссылаются на строки представления, выходит за рамки данной книги.

присваивается во время вставки этой строки и остается связанным со строкой[9] до тех пор, пока она не будет удалена.

5. Структурированный тип, при его использовании в качестве основы для определе ния базовой таблицы, не рассматривается как инкапсулированный (хотя он в большей или меньшей степени и рассматривается как таковой в других контек стах). Поэтому в данном примере базовая таблица DEPT имеет четыре столбца, DEPT_ID, DEPT#, DNAME и BUDGET (в указанном порядке), а не просто два, как было бы в том случае, если бы тип DEPT_TYPE был инкапсулированным.
6. Значением, применяемым по умолчанию для столбца DEPT_ID, является NULL (как фактически и предусмотрено для всех столбцов, которые определены как от носящиеся к некоторому типу REF, хотя это заданное по умолчанию значение в основном теряет смысл, если для рассматриваемого столбца дополнительно задана спецификация NOT NULL).

Теперь дополним рассматриваемый пример, чтобы ввести в него базовую таблицу EMP, следующим образом[10].

```
CREATE TABLE EMP
     ( EMP#    CHAR(5)  NOT NOLL, ENAME
      CHAR(25) NOT NOLL, SALARY  MONEY
      NOT NOLL, DEPT ID REF ( DEPT TYPE )
      SCOPE DEPT REFERENCES ARE CHECKED
      ON DELETE CASCADE
                          NOT
    NOLL, PRIMARY KEY ( EMP# )
    ) ;
```

При обычных условиях базовая таблица EMP включала бы столбец внешнего ключа DEPT#, который ссылался бы на отделы с помощью номеров отделов. Но здесь имеется *справочный* столбец DEPT_ID (причем заслуживает внимания то, что он не обозначен явно именно как столбец внешнего ключа), который вместо этого указывает на отделы с помощью *ссылок* на строки с данными об этих отделах. В опции SCOPE DEPT задана соответствующая упоминаемая в ссылках таблица. Спецификация REFERENCES ARE CHECKED означает, что должна поддерживаться ссылочная целостность (при использовании опции REFERENCES ARE NOT CHECKED появилась бы возможность создания *зависших ссылок,* поэтому непонятно, зачем вообще могло бы потребоваться задавать опцию NOT CHECKED)[11] Конструкция ON DELETE. . . указывает правило удаления,

---

[9] На первый взгляд может даже показаться, что здесь возникает какая-то цикличность: выражение "эта строка" может означать только "строку, которая имеет данный конкретный рассматриваемый идентификатор". В частности, обратите внимание на путаницу между значением и переменной! Если "эта строка" должна иметь адрес, то "эта строка" должна и быть переменной строки (см. раздел 26.3).

[10] Обратите внимание на спецификации NOT NOLL в столбцах таблицы EMP. Указать, что в таблице DEPT также не разрешается иметь столбцы, содержащие неопределенные значения, не так уж просто! Анализ дополнительных сведений по этому вопросу оставляем читателю в качестве упражнения.

[11] И тем не менее, велика вероятность того, что в окончательной редакции стандарта SQL:2003 спе цификация REFERENCES. . . будет изъята; из этого следует, что (по умолчанию) всегда будет задана ОПЦИЯ REFERENCES ARE NOT CHECKED.

аналогичное обычным правилам удаления внешнего ключа (поддерживаются такие же опции). Аналогичная спецификация ON UPDATE. . . не предусмотрена.

Использование ссылок

Теперь рассмотрим несколько примеров запросов и обновлений применительно к только что объявленной базе данных отделов и служащих. Вначале рассмотрим приведенную ниже формулировку на языке SQL: "Определить номер отдела служащегоЕ1".

```
SELECT DEPT ID -> DEPT# AS DEPT#
FROM EMP
WHERE EMP# = 'El' ;
```

Обратите внимание на оператор разадресации[12] "->" в конструкции SELECT (выражение DEPT_ID -> DEPT# извлекает значение DEPT# из строки DEPT, на которую указывает рассматриваемое значение DEPT_ID). Обратите также внимание на необходимость задавать конструкцию AS; если бы эта конструкция была опущена, то соответствующий столбец результата по существу остался бы безымянным. Наконец, следует отметить противоречащий здравому смыслу характер конструкции FROM — значение DEPT#, подлежащее выборке, поступает из таблицы DEPT, а не EMP, а значения DEPT_ID поступают из таблицы EMP, а не DEPT.

Кстати, трудно преодолеть искушение и не отметить, что производительность этого первого запроса, скорее всего, будет ниже по сравнению с его аналогом, сформулированным с помощью обычных средств SQL (который обращался бы только к одной таблице, а не к двум). Автор привел это замечание потому, что обычно в пользу *ссылок* приводят такой довод, будто бы они способствуют повышению производительности (что "следуя по указателю, можно быстрее получить данные, чем выполняя соединение"). Безусловно, недостатком подобных доводов является и то, что в них перепутаны логические и физические проблемы.

В качестве второго примера предположим, что первоначальный запрос был сформулирован следующим образом: "Определить отдел (а не просто номер отдела) служащего Е1". Теперь операция разадресации выглядит совсем иначе, как показано ниже.

```
SELECT DEREF ( DEPT ID ) AS DEPT
FROM EMP
WHERE EMP# = 'El' ;
```

Более того, в данном случае вызов оператора DEREF приводит к получению не значения строки DEPT (как можно было бы ожидать), а, вместо этого, к получению *инкапсулированного* (скалярного) значения. Это значение относится к типу DEPT_TYPE И поэтому

[12] В большинстве языков, в которых поддерживается разадресация (dereferencing), поддерживается также оператор адресации (см., например, описание оператора PTR_TO а разделе 26.3), но в языке SQL такая поддержка не предусмотрена. Более того, обычно в результате разадресации возвращается переменная, но в языке SQL соответствующая операция вместо этого возвращает значение.

имеет только три атрибута, DEPT#, DNAME И BUDGET (ОНО не включает атрибут DEPT_ID)[13].

Примечание. Повторяя замечание, сделанное в главе 6, отметим, что если объявленным типом некоторого формального параметра P некоторого оператора Op является DEPT_TYPE, то нельзя передавать строку из таблицы DEPT в качестве соответствующего фактического параметра в вызове этого оператора Op. Но теперь очевидно, что вместо этого можно передавать результат выражения DEREF(DEPT_ID) , если DEPT_ID содержит ссылку на строку таблицы DEPT.

Ниже приведен еще один пример запроса: "Определить номера служащих отдела D1".

```
SELECT EMP#
FROM EMP
WHERE DEPT_ID -> DEPT# = 'D1' ;
```

Обратите внимание на использование в данном примере разадресации в конструкции WHERE. Теперь рассмотрим следующий пример предложения INSERT (вставка данных о служащем).

```
INSERT INTO EMP ( EMP#, DEPT ID )
       VALUES ( 'E5',  ( SELECT DEPT ID FROM
                         DEPT WHERE  DEPT# = ' D2 '
                         ) ) ;
```

Отметим, что сторонники рассматриваемых конструкций (типов REF и т.д.) подчеркивают такую мысль, будто не о чем беспокоиться, поскольку "все эти конструкции фактически являются просто сокращениями". Например, они утверждают, что выражение SQL.

```
SELECT DEPT ID -> DEPT# AS DEPT#
FROM EMP
WHERE EMP# = 'E1' ;
```

("Определить номер отдела служащего E1"; первый из примеров, рассматриваемых в данном подразделе) является сокращенным обозначением для приведенного ниже предложения.

```
SELECT ( SELECT DEPT#
         FROM DEPT
         WHERE DEPT.DEPT ID = EMP.DEPT ID ) AS
DEPT# FROM EMP WHERE EMP# = 'E1' ;
```

Но в действительности утверждение о том, что это — сокращение, не выдерживает никакой критики, поскольку применяемый здесь новый синтаксис (в частности, для разадресации) может использоваться только в сочетании с данными, которые были определены с

---

[13] Из семантики оператора DEREF следует, что в системе необходимо также хранить информацию о том, что столбцы DEPT#, DNAME и BUDGET таблицы DEPT происходят от структурированного типа DEPT_TYPE (хотя для большинства назначений они могут рассматриваться просто как обычные столбцы). Иными словами, используя терминологию типов и заголовков из главы 6, можно утверждать, что таблица DEPT имеет заголовок (и поэтому тип) с четырьмя компонентами в одних контекстах, но принимает заголовок (и поэтому тип) только с двумя компонентами в других. Возможно, "типизированные таблицы" лучше было бы назвать таблицами "с раздвоением личности"?

помощью специального нового способа, с использованием якобы "принципиально нового" генератора типа (REF). Кроме того, в этих определениях данных широко применяется также новый синтаксис. К тому же, рассматриваемые функциональные возможности предоставляются в высшей степени в неортогональной форме (кроме всего прочего, они распространяются только на те таблицы, которые определены с помощью данного специального нового способа, а не на все таблицы).

Более того, даже если считать правильным утверждение о том, что это — сокращение, остается без ответа вопрос, для чего вообще нужны такие сокращения. Какая проблема намечена для решения с их помощью? Почему соответствующая поддержка является такой неортогональной? При каких условиях мы должны использовать "старомодный" способ, а при каких — этот странный новый способ? И это — далеко не исчерпывающий список вопросов.

Примечание. В этой связи см. [26.15] и аннотацию к [26.21].

Подтаблицы и супертаблицы

В языке SQL разрешено определять базовую таблицу в как *подтаблицу* базовой таблицы A, только если в и A обе являются *типизированными таблицами,* а структурированный тип STB, с помощью которого определена таблица в, является подтипом структурированного типа STA, с помощью которого определена таблица A. В качестве примера рассмотрим следующие определения структурированных типов.

```
CREATE TYPE EMP TYPE /* служащие
   */ AS ( EMP# ..., DEPT# ... )
   REF IS SYSTEM GENERATED ;

CREATE TYPE PGMR TYPE UNDER EMP TYPE /*
   программисты */ AS ( LANG ... ) ;
```

Обратите внимание на то, что в предложении PGMR_TYPE нет конструкции REF IS. ..; вместо этого данный тип фактически наследует указанную конструкцию от своего непосредственного *(прямого)* супертипа EMP_TYPE. Иными словами, значение типа REF (EMP_TYPE) теперь может ссылаться на строку в таблице, которая определена как относящаяся к типу PGMR_TYPE, а не К типу EMP_TYPE.

Теперь рассмотрим приведенные ниже определения базовой таблицы.

```
CREATE TABLE EMP OF EMP TYPE
     ( REF IS EMP ID SYSTEM GENERATED,
       PRIMARY KEY ( EMP# ) ) ... ;
CREATE TABLE PGMR OF PGMR TYPE UNDER
EMP ;
```

Здесь заслуживает внимания спецификация UNDER EMP после определения базовой таблицы PGMR (следует отметить также отсутствие спецификаций REF is и PRIMARY KEY для этой базовой таблицы). Базовые таблицы PGMR и EMP называются, соответственно, подтаблицей и относящейся к ней непосредственной *(прямой)* супертаблицей; в таблице PGMR унаследованы столбцы (и т.д.) от EMP и введен один собственный дополнительный столбец LANG. На интуитивном уровне этот пример можно понять таким образом, что для служащих, не являющихся программистами, предусмотрена строка только в таблице EMP, а для программистов имеются строки в обеих таблицах, поэтому каждая строка

в таблице PGMR имеет свой аналог в таблице EMP, но обратное утверждение не является истинным.

Операции манипулирования данными в этих таблицах осуществляются, как описано ниже.

- Операция SELECT. Выборка данных с помощью операции SELECT из таблицы EMP осуществляется обычным образом. Операция SELECT на таблице PGMR выполня ется так, как если бы PGMR фактически содержала столбцы таблицы EMP, а также столбец LANG.
- Операция INSERT. Операция вставки данных с помощью оператора INSERT в таб лицу EMP осуществляется обычным образом. Выполнение операции INSERT с таб лицей PGMR фактически приводит к появлению новых строк и в таблице EMP, и в таблице PGMR.
- Операция DELETE. Выполнение операции DELETE с таблицей EMP влечет за собой исчезновение строк из таблицы EMP, а также (если оказалось, что рассматривае мые строки относятся к программистам) из таблицы PGMR. Удаление данных с по мощью оператора DELETE из таблицы PGMR вызывает исчезновение строк и из таблицы EMP, и из таблицы PGMR.
- Операция UPDATE. Обновление атрибута LANG, которое должно осуществляться обязательно через таблицу PGMR, приводит к обновлению только таблицы PGMR; обновление других столбцов (в принципе) приводит к обновлению обоих таблиц, независимо от того, осуществляется ли оно с помощью либо таблицы EMP, либо таблицы PGMR.

В частности, следует отметить, что из этого анализа операций манипулирования данными следуют приведенные ниже выводы.

- Предположим, что некоторый существующий служащий Джо стал программи стом. Если мы просто попытаемся вставить строку с данными о Джо в таблицу PGMR, то система попытается вставить строку, касающуюся Джо, также и в табли цу EMP, но такая попытка, безусловно, окончится неудачей. Вместо этого необхо димо удалить строку Джо из таблицы EMP, а затем вставить соответствующую строку в таблицу PGMR.
- Напротив, предположим, что некоторый существующий служащий Джо перестал быть программистом. На этот раз необходимо удалить строку Джо либо из EMP, либо из PGMR (какая бы таблица ни была задана, в конечном итоге происходит уда ление из обеих таблиц), а затем вставить соответствующую строку в таблицу EMP.

Кстати, следует отметить, что в языке SQL предусмотрено средство ONLY, которое действительно позволяет проводить манипуляции только с такими строками данной конкретной таблицы, которые не имеют аналогов в какой-либо подтаблице этой таблицы. Например, выражение SELECT * FROM ONLY (EMP) позволяет осуществить выборку только тех строк таблицы EMP, которые не имеют аналогов в таблице PGMR; аналогичным образом, с помощью выражения DELETE FROM ONLY (EMP) можно удалить только те строки EMP, которые не имеют аналогов в PGMR, и подобная конструкция может применяться в сочетании с оператором UPDATE (версия оператора INSERT с ключевым

словом ONLY не предусмотрена). Но заслуживает внимание то, что это средство ONLY не может помочь в решении проблем, описанных выше, когда приходится обновлять базу данных с учетом того, что некоторый существующий служащий Джо стал программистом или перестал им быть.

Итак, что общего имеют только что описанные подтаблицы и супертаблицы с истинным наследованием типов? Насколько мы можем судить, ответом является — НИЧЕГО. Таблицы — это не типы! Безусловно, здесь и речи не может быть о какой-либо заменяемости, а в главе 20 было показано, что если заменяемость не предусмотрена, то нет и подлинного наследования типов. Поэтому, если даже предположить, что подтаблицы и супертаблицы способны предоставить какие-либо полезные средства, все равно не совсем понятно, почему язык SQL должен требовать, чтобы для получения доступа к этим средствам рассматриваемые подтаблицы и их непосредственные *(прямые)* супертаблицы были объявлены в терминах типов, которые являются, соответственно, подтипом и его непосредственным *(прямым)* супертипом.

Более фундаментальный вопрос состоит в том, каковы же здесь возможные положительные средства. Чем могут нас подкупить подтаблицы и супертаблицы? По-видимому, ответ состоит в том, что "пользы от них очень мало", по крайней мере, на уровне модели[14]. Верно то, что может быть достигнута некоторая экономия в реализации, если подтаблица и ее супертаблица физически хранятся на диске в виде одной таблицы, но, безусловно, нельзя допускать, чтобы подобные соображения оказывали хоть какое-либо влияние на саму модель как таковую. Иными словами, как было отмечено в предыдущем абзаце, не только не ясно, почему "под-" и "супер-" таблицы должны создаваться на основе структурированных "под-" и "супер-" типов, но также совершенно не понятно, для чего вообще должно поддерживаться это средство.

Язык SQL и два серьезных заблуждения

Итак, позволяют ли описанные выше функциональные средства SQL достичь важной цели — предоставление качественной объектно-реляционной поддержки? Прежде всего, если даже в языке SQL не в полной мере проявляются два серьезных заблуждения, безусловно, в нем обнаруживаются весьма заметные их признаки. И как уже было сказано, причины такого положения дел не совсем понятны, по крайней мере, автору данной книги; по-видимому, в основном они следуют лишь из смутного представления, что указанные средства, являющиеся следствием непреодоленных серьезных заблуждений, должны в какой-то степени позволить языку SQL стать более "похожим на объектный" (да, по-видимому, эти средства как раз и способствуют достижению именно этой цели).

Что касается первого серьезного заблуждения, то создается впечатление, что идея привязки типизированных таблиц к структурированным типам имеет определенное касательство к идее приравнивания таблиц и классов. Точнее, представляется вполне вероятным, что если типизированная таблица тт определена с помощью ключевого слова "OF" как относящаяся к структурированному типу ST, то предполагается, что тт должна содержать то, что иногда называют *экстентом* типа ST (под этим подразумевается множество

---

[14] Можно было бы предположить, что ответ на этот вопрос имеет нечто общее с проблемой подтипов и супертипов сущностей, которые рассматривались в главе 14. Но в этой связи автор хотел бы отметить, что для решения и этой проблемы он предпочитает использовать подход, основанный на применении представлений. См. пример в самом конце раздела 14.5.

всех существующих в настоящее время *экземпляров* типа ST)[15]. Если дело обстоит иначе, то для чего нужна тесная связь между TT и ST?

Если это действительно так, то опять возникают некоторые проблемы. Одна из них состоит в том, что не исключена возможность иметь две или несколько типизированных таблиц одного и того же структурированного типа; какие последствия повлечет за собой такое решение, не совсем ясно (если не считать того, что в число этих последствий почти наверняка будет входить нарушение принципа ортогонального проектирования). Другие проблемы описаны в разделе 6.6.

Что касается второго серьезного заблуждения, то должно быть очевидно, что язык SQL страдает от этого (очень важного!) дефекта, даже если допустить, что применяемые в нем *ссылки* и связанные с ними средства являются, как утверждают некоторые, фактически просто сокращениями. Как было отмечено выше, если строки могут иметь *ссылки* (адреса), то эти строки по определению являются переменными строк. В частности, язык SQL страдает от проблемы, описанной в подразделе "Несовместимость указателей и качественной модели наследования" раздела 26.3. Здесь подробные сведения по этой теме не приведены, поскольку они являются довольно сложными; достаточно сказать, что значение REF, которое предположительно должно ссылаться на строку, описывающую окружность, может фактически ссылаться вместо этого на строку, содержащую эллипс, отличный от окружности.

26.7. РЕЗЮМЕ

В настоящей главе кратко рассматривалась проблемная область объектно-реляционных систем. Такие системы являются (или должны быть), по сути, просто реляционными системами, которые должным образом поддерживают реляционное понятие доменов (т.е. типов). Иными словами, объектно-реляционные системы — это подлинно реляционные системы, отличительной особенностью которых является то, что они позволяют пользователям определять их собственные типы. Реляционная модель не требует никаких дополнений для достижения объектно-реляционных функциональных возможностей; достаточно лишь полностью ее реализовать.

Затем в этой главе рассматривались два серьезных заблуждения. Первое из них состоит в том, что проводится знак равенства между классами объектов и переменными отношения (к сожалению, это уравнение все же слишком привлекательно, по крайней мере, если не проводится его глубокий анализ). Автор сделал предположение, что источником этого заблуждения становится путаница между двумя совершенно различными интерпретациями термина *объект*. Было подробно описано, как может выглядеть система, в которой обнаруживаются следы первого серьезного заблуждения, и показаны некоторые последствия этой ошибки. Одним из важных последствий, по-видимому, является то, что первое заблуждение приводит непосредственно к возникновению также второго серьезного заблуждения! Оно заключается именно в смешивании указателей и отношений (хотя фактически второе заблуждение может проявляться и без первого, и, к сожалению, его "следы" можно обнаружить почти в любой системе, представленной на рынке). По мнению автора, второе серьезное заблуждение во многом подрывает концептуальную

[15] Но этот *экстент* не сопровождается автоматически; вместо этого *экземпляры* типа ST появляются в таблице TT и исчезают из нее только в результате выполнения явных операций обновления на этой таблице.

целостность реляционной модели (фактически так же, как и первое). В частности, в результате него нарушаются и информационный принцип, и принцип взаимозаменяемости (базовых и производных отношений).

После этого были кратко рассмотрены некоторые проблемы реализации. При этом наиболее важная мысль состояла в том, что введение нового *пакета типов* оказывает воздействие, по меньшей мере, на такие компоненты системы, как компилятор, оптимизатор и диспетчер памяти. Вследствие этого объектно-реляционная система не может быть реализована (по крайней мере, достаточно качественно) просто путем наложения нового уровня кода *{оболочки)* на существующую реляционную систему; вместо этого система должна быть перестроена, начиная от самого основания, для того, чтобы каждый **ее** компонент стал расширяемым индивидуально по мере необходимости.

В дальнейшем в этой главе рассматривалась матрица классификации СУБД Стоунбрейкера и были кратко описаны преимущества, которые могут быть достигнуты благодаря истинному сближению между объектной и реляционной технологиями (здесь под словом "истинное", кроме всего прочего, подразумевается, что рассматриваемая система не должна носить следы ни одного из двух серьезных заблуждений). Наконец, проанализирована предусмотренная в языке SQL поддержка **типов REF,** а также **подтаблиц** и **супертаблиц.**

УПРАЖНЕНИЯ

26.1. Дайте определение термина *объектно-реляционный.* Что такое *объектно-реляционная модель"!*

26.2. Ниже приведена версия кода, который использовался в разделе 26.3 для демонст рации того, что указатели и качественная модель наследования не совместимы.

```
VAR E ELLIPSE ;
VAR XE PTR TO ELLIPSE ;
VAR XC PTR_TO_CIRCLE ;

E  := CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) ;
XE := PTR TO ( E ) ;
XC := TREAT DOWN AS PTR TO CIRCLE ( XE ) ;
THE A ( DEREF ( XE ) )  := LENGTH ( 6.0 ) ;
```

Что произойдет при выполнении этого кода?
Примечание. Здесь DEREF — обычный оператор разадресации, а не оператор SQL с тем же именем (получив адрес переменной, он возвращает указанную переменную).

26.3. (Продолжение предыдущего упражнения.) Почему аналогичная проблема не воз никает при использовании внешних ключей вместо указателей? Или она все же возникает?

26.4. Как вы считаете, являются ли структурированные типы SQL инкапсулированны ми? Обоснуйте свой ответ.

26.5. Имеет ли смысл в языке SQL объявлять локальную переменную как принадлежа щую к некоторому типу REF? Если да, то какие последствия будут с этим связаны?

26.6. Приведите версию кода, показанного в упр. 26.2, на языке SQL.

Примечание. Для выполнения этого упражнения может потребоваться доступ к стандарту SQL или к документации одного из программных продуктов SQL.

26.7. Исследуйте любую доступную вам объектно-реляционную СУБД. Имеются ли в этой системе проявления любого из двух серьезных заблуждений? Если да, то ка кие доводы приведены в ее документации в качестве обоснования необходимости использования соответствующих средств?

26.8. Дайте определения понятий подтаблиц и супертаблиц:

а) в общих терминах;

б) исключительно в терминах SQL.

СПИСОК ЛИТЕРАТУРЫ

В 1980-х и 1990-х годах было создано несколько объектно-реляционных прототипов. Среди них наиболее известными и признанными являются система **Postgres,** разработанная в университете штата Калифорния в Беркли ([26.36], [26.40], [26.42], [26.43]), и система **Starburst,** разработанная в лаборатории IBM Research ([26.19], [26.23], [26.29], [26.30]). Обратите внимание на то, что ни одна из них не может быть отнесена, по крайней мере, в оригинальной версии, к системам, в которых соблюдается уравнение *домен = класс,* указанное в этой главе как "безусловно правильное".

26.1. Adler D. W. IBM DB2 Spatial Extender— Spatial Data Within the RDBMS // Proc. 27th Int. Conf. on Very Large Data Bases, Rome, Italy. — September 2001.

26.2. Bohm C, Berchtold S., Keim D.A. Searching in High-Dimensional Spaces— Index Structures for Improving the Performance of Multimedia Databases // ACM Сотр. Surv. — September 2001. — 33, № 3.

26.3. Brooks F.P., Jr. The Mythical Man-Month (это издание посвящено 20-летнему юби лею знаменитой статьи Брукса "Мифический человеко-месяц"). — Reading, Mass.: Addison- Wesley, 1995.

26.4. Carey M.J., Mattos N.M., Nori A.K. Object/Relational Database Systems: Principles, Products, and Challenges // Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data. — Tucson, Ariz. — May 1997.

Цитата из работы: "Абстрактные типы данных, определяемые пользователем функции, типы строк, ссылки, наследование, подтаблицы, коллекции, триггеры... Что же все-таки это такое?". Хороший вопрос! В списке есть восемь средств и по умолчанию предполагается, что все они представляют собой средства SQL. Автор утверждает, что четыре из этих восьми средств нежелательны, еще два относятся к одной и той же категории, а последние два не имеют касательства к вопросу о том, относится ли система к объектно-реляционным системам или к системам других типов.

26.5. Carey M.J. et al. The BUCKY Object/Relational Benchmark // Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data. — Tucson, Ariz. — May 1997.

Цитата из резюме: "BUCKY (Benchmark of Universal or Complex Afwery interfaces [именно так!]) — эталонный тест, ориентированный на запросы, с помощью которого проверяются многие из ключевых возможностей объектно-реляционной системы, включая типы строк и наследование, ссылки и обозначения пути, множества атомарных значений и ссылок, методы и позднее связывание, а также определяемые пользователем абстрактные типы данных и их методы".

Примечание. Что касается этого списка "ключевых возможностей", см аннотацию к [26.4].

26.6. Carey M. J. et al. O-O, What Have They Done to DB2? // Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland. — September 1999.

Обзор объектно-реляционных средств СУБД DB2. См. также [26.1], [26.4] и [26.11].

26.7. Cattell R.G.G. What Are Next-Generation DB Systems? // CACM. - October 1991. - 34, № 10.

26.8. Chamberlin D.D. Relations and References — Another Point of View // InfoDB. — April 1997.-10, №6.

См. аннотацию к [26.15].

26.9. Chaudhuri S., Gravano L. Optimizing Queries over Multi-Media Repositories // Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data. — Montreal, Canada. — June 1996.

Выражение "хранилища мультимедийных данных", приведенное в заголовке этой статьи, обозначает одно возможное направление использования объектно-реляционных баз данных. Результаты запросов к мультимедийным данным обычно содержат не только множество результирующих объектов, но также показывают *степень соответствия* для каждого такого объекта, с помощью которой можно определить, в какой мере он удовлетворяет условию поиска (например, если требуется изображение, содержащее красный цвет, то степень соответствия показывает *степень красноты).* В таких запросах можно задавать пороговое значение степени соответствия, а также указывать *квоту* [7.5]. В этой статье рассматривается оптимизация подобных запросов. См. также [26.2].

26.10. Chaudhuri S., Shim K. Optimization of Queries with User-Defined Predicates // ACM TODS. - June 1999. - 24, № 2.

См. также [26.26] и [26.35].

26.11. Chen W. et al. High Level Indexing of User-Defined Types // Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland. — September 1999.

Описаны некоторые методы реализации, применяемые в DB2.

26.12. Codd E.F. and Date C.J. Interactive Support for Nonprogrammers: The Relational and Network Approaches // Date C.J. Relational Database: Selected Writings. — Reading, Mass.: Addison-Wesley, 1986.

В статье вводится понятие *существенности* — концепции, которая очень важна для правильного понимания моделей данных (в обоих смыслах этого термина! См. главу 1, раздел 1.3). Реляционная модель в своей основе имеет лишь одну существенную конструкцию данных, а именно — отношение. Объектная модель, напротив, имеет много конструкций: множества, мультимножества, списки, массивы и т.д. (не говоря уже об идентификаторах объектов). См. [26.13], [26.14] и [26.17] для ознакомления с дополнительными пояснениями.

26.13. Date C.J. Support the Conceptual Schema: The Relational and Network Approaches // Relational Database Writings 1985-1989. — Reading, Mass.: Addison-Wesley, 1990.

Один из доводов против смешивания указателей и отношений [26.15] состоит в том, что указатели становятся причиной значительного усложнения. В данной статье приводится пример, очень ясно иллюстрирующий эту мысль (рис. 26.5 и 26.6).

26.14. Date C.J. Essentiality // Relational Database Writings 1991-1994.— Reading, Mass.: Addison-Wesley, 1995.

26.15. Date C.J. Don't mix Pointers and Relations! и Don't mix Pointers and Relations! — Please // Date C.J., Darwin H. and McGoveran D. Relational Database Writings 1994—1997. — Reading, Mass.: Addison-Wesley, 1998.

В первой из этих двух статей приводятся доводы против второго серьезного заблуждения. В [26.8] Чемберлен (Chamberlin) опровергает некоторые доводы первой статьи. Вторая статья является прямым ответом на опровержение Чемберлена.

26.16. Date C.J. Objects and Relations: Forty-Seven Points of Light // Date C.J., Darwin H. and McGoveran D. Relational Database Writings 1994-1997. — Reading, Mass.: Addison-Wesley, 1998.

Подробный ответ на [26.27].

26.17. Date C.J. Relational Really Is Different. // Глава 10 из [6.9].

MAJOR_P#	MINOR_P#	QTY
P1	P2	2
P1	P4	4
P5	P3	1
P3	P6	3
P6	P1	9
P5	P6	8
P2	P4	3

Рис. 26.5. Отношение спецификации материалов

26.18. Date C. J. What Do You Mean, 'Post-Relational'? // http: / /www. dbdebunk. com. — June 2000.

В литературе иногда встречается термин *постреляционный.* Он может означать то же, что и *объектно-реляционный,* или нечто иное.

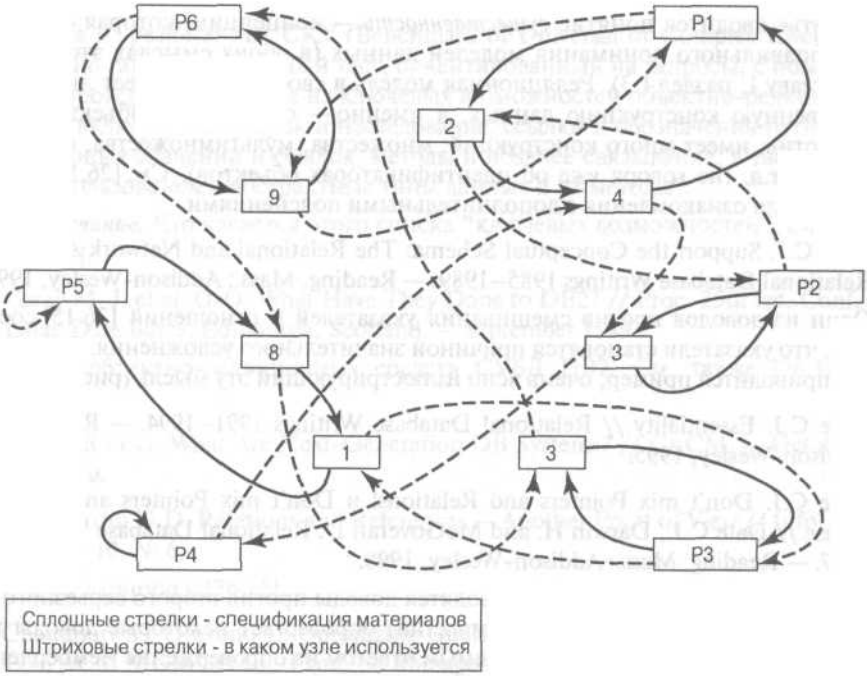


Рис. 26.6. Основанный на указателях аналог отношения, показанного на рис. 26.5

26.19. DeMichael L.G., Chamberlin D.D., Lindsay B.G., Agrawal R., Arya M. Polyglot: Extentions to Relational Databases for Sharable Types and Functions in a Multi-Language Environment // IBM Research Report RJ8888. - 1992.

Цитата из резюме: "Polyglot — расширяемая система типов реляционной базы данных, поддерживающая наследование, инкапсуляцию и динамическое планирование методов". (Динамическое планирование методов, dynamic method dispatch, — другое название для связывания на этапе прогона. Продолжаем цитату.) "[Polyglot] позволяет использовать объекты из многих прикладных языков, а также предоставляет объектам возможность сохранять в неизменном виде свои методы после пересечения ими границы между базой данных и прикладной программой. В статье описан проект системы Polyglot, расширения языка SQL для поддержки используемых системой Polyglot типов и методов, а также реализация системы Polyglot в реляционном [прототипе] Starburst".

Система Polyglot имеет непосредственное отношение к темам этой главы (а также глав 5, 20 и 25). Но здесь следует сделать несколько замечаний. Во-первых, в статье ни разу не упоминается реляционный термин **домен** (что очень удивительно). Во-вторых, в системе Polyglot имеются встроенные генераторы типов (в терминах этой системы — *метатипы):* **тип-база, тип-кортеж, тип-переименование, тип-массив** и **тип-язык,** но (что также удивительно) нет генератора **типа-отношения.** Однако в системе имеется возможность вводить новые генераторытипов.

26.20. DeWitt D.J., Karba N., Luo J., Patel J.M., Yu J.-B. Client-Server Paradise // Proc. 20th Int. Conf. on Vary Large Data Bases. — Santiago, Chile. — September 1994.

Система Paradise (Parallel Data Information System — информационная система с параллельной обработкой данных) — это разработанный в Висконсинском университете объектно-реляционный прототип системы (который первоначально именовался "расширенным реляционным"), "созданный для ГИС-приложений" (ГИС — геоинформационная система). В статье описаны архитектура и реализация системы Paradise.

26.21. Eisenberg A., Melton J. SQL: 1999, Formerly Known as SQL3 // ACM SIGMOD Record. — March 1999. - 28, № 1.

Обзор различий между спецификациями SQL: 1992 и SQL: 1999. Кстати, сразу после появления этой статьи Хью Дарвен (Hugh Darwen) и автор данной книги написали редактору выпуска SIGMOD Record следующее: "Что касается [рассматриваемой статьи], в частности, ее разделов «Наконец... объекты» и «Применение типов REF», мы хотим задать вопрос — какое полезное назначение имеют средства, описанные в этих разделах? Вернее, какие ими предоставляются полезные функциональные возможности, которые не могут быть получены с помощью средств, уже предусмотренных в спецификации SQL: 1992?" Наше письмо не было опубликовано.

26.22. Godfrey M., Mayr T., Seshadri P., von Eichen T. Secure and Portable Database Extensibility // Proc. 1998 ACM SIGMOD Int. Conf. on Management of Data. — Seattle, Wash. — June 1998.

Приведем цитату из этой статьи (немного перефразированную): "Поскольку операторы, определяемые пользователем, поставляются неизвестными или ненадежными клиентами, в СУБД должны быть предусмотрены меры предосторожности по отношению к операторам, которые могут вызвать аварию системы, непосредственно модифицировать ее файлы или содержимое памяти (обойдя механизмы авторизации), монополизировать ресурсы процессора, памяти или диска". Очевидно, что необходим дополнительный контроль. В этой статье рассказывается об исследованиях данного вопроса с использованием языка Java и объектно-реляционного прототипа PREDATOR [26.33]. В обнадеживающем заключении говорится, что система базы данных "может поддерживать безопасные и переносимые расширения на основе языка Java без значительных потерь впроизводительности".

26.23. Haas L.M., Freytag J.C., Lohman G.M., Pirahesh H. Extensible Query Processing in Starburst // Proc. ACM SIGMOD Int. Conf. on Management of Data. — Portland, Ore.-June 1989.

После выхода первоначальной статьи [26.29] цели проекта Starburst были немного расширены: "Система Starburst обеспечивает поддержку для добавления новых методов хранения таблиц, новых методов доступа и ограничений целостности, новых типов данных, функций и новых операций с таблицами". При этом система подразделяется на два основных компонента, Core и Corona, которые соответствуют компонентам RSS и RDS первоначального прототипа System R (описание этих двух компонентов System R приведено в [4.2] и [4.3]. В компоненте Core поддерживаются функции расширения, описанные в [26.29], а в компоненте Corona —

язык запросов Hydrogen системы Starburst, который является диалектом языка SQL. В этом диалекте исключено большинство ограничений реализации языка SQL, принятого в системе System R, он более ортогонален, чем язык SQL прототипа System R, поддерживает рекурсивные запросы и может быть расширен пользователем. В статье содержится интересное обсуждение проблемы *перезаписи запросов,* т.е. правил преобразования выражений (см. главу 18). Об этом можно также прочесть в [18.48].

26.24. Hellerstein J.M., Naughton J.F. Query Execution Techniques for Caching Expensive Methods // Proc. 1996 ACM SIGMOD Int. Conf. on Management of Data.— Montreal, Canada. — June 1996.

26.25. International Organization for Standardization (ISO). Information Technology — Database Languages — SQL Multimedia and Application Packages // Document ISO/IEC 13249:2000.

Официальное определение стандарта SQL/MM. Как и стандарт SQL, на котором он основан [4.23], этот стандарт состоит из открытой серии отдельных так называемых "частей" (ISO/IEC 13249-1, -2 и т.д.). Ко времени написания данной книги были определены перечисленные ниже части.

- Часть 1. Инфраструктура.
- Часть 2. Полнотекстовый доступ.
- Часть 3. Пространственный доступ.
- Часть 4. Отсутствует.
- Часть 5. Обработка неподвижных изображений.
- Часть 6. Информационная проходка.

26.26. Jaedicke M., Mitschang B. User-Defined Table Operators: Enhancing Extensibility for ORDBMS // Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland. — September 1999.

26.27. Kim W. On Marrying Relations and Objects: Relation-Centric and Object-Centric Perspectives// Data Base Newsletter. — November/December 1994. — 22, № 6.

В этой статье приводятся доводы против той позиции, что приравнивание переменных отношения и классов — это серьезная ошибка *{первое серьезное заблуждение).* Статья [26.16] — ответ на данную статью.

26.28. Kim W. Bringing Object/Relational Down to Earth // DBP&D. — July 1997. — 10, № 7.

В этой статье автор утверждает, что на рынке объектно-реляционных систем "путаница наверняка будет продолжаться", поскольку, во-первых, "на средства расширения типов данных была взвалена непомерная ноша", и, во-вторых, "степень полноты объектно-реляционных продуктов... вызывает серьезные опасения". Предлагается "практическая метрика объектно-реляционной полноты, которая может быть использована как руководство для определения, является ли продукт действительно объектно-реляционным". В схему автора включены перечисленные ниже критерии.

1. Модель данных.
2. Язык запросов.
3. Жизненно важные службы.
4. Вычислительная модель.
5. Производительность и масштабируемость.
6. Инструменты базы данных.
7. Полнота использования вычислительной мощности.

Что касается первого из этих критериев (самого важного), Ким занимает позицию (весьма отличную от изложенной в Третьем манифесте [3.3]), что моделью данных должна быть "основная объектная модель, определенная группой Object Management Group (OMG)", которая "включает реляционную модель данных, а также основные концепции объектно-ориентированного моделирования объектно-ориентированных языков программирования". По мнению Кима, сюда входят следующие понятия: *класс* (Ким добавляет "или тип"), *экземпляр, атрибут, ограничения целостности, идентификаторы объектов, инкапсуляция, {множественное) наследование классов, {множественное) наследование абстрактных типов данных, данные типа ссылок, атрибуты со значениями в виде множества, атрибуты классов, методы классов* и т.п. Отметим, что отношения, которые автор данной книги рассматривает как наиболее важные и фундаментальные, нигде явно не упоминаются. Ким утверждает, что основная объектная модель группы OMG в дополнение ко всему перечисленному в списке полностью включает реляционную модель, хотя на самом деле это не так.

26.29. Lindsay B., McPherson J., Pirahesh H. A Data Management Extension Architecture // Proc. ACM SIGMOD Int. Conf. on Management of Data. — San Francisco, Calif. — May 1987.

В работе описана архитектура прототипа системы Starburst, в котором "реализованы расширения управления данными для реляционных систем баз данных". Описаны два типа таких расширений: определяемые пользователем структуры хранения и методы доступа, а также определяемые пользователем триггеры и ограничения целостности (но должны ли *все* ограничения целостности непременно быть определены пользователем?). Однако, как указано в этой статье, "помимо этих, существуют также другие направления расширения СУБД, включая определяемые пользователем типы данных и технологии выполнения запросов".

26.30. Lohman G.M. et al. Extensions to Starburst: Objects, Types, Functions and Rules // CACM. - October 1991. - 34, № 10.

26.31. Maier D. Comments on the Third-Generation Database Sytem Manifesto // Tech. Report No. CS/E 91-012. — Oregon Gradulate Center, Ore. — April 1991.

Майер весьма критичен буквально ко всему материалу статьи [26.44]. Автор согласен с некоторыми его критическими замечаниями, но не согласен с остальными. Однако автора заинтересовали следующие высказывания (в них поддерживается его точка зрения, что концепция объектов содержит лишь одну хорошую идею, а именно — *надлежащую поддержку типов данных):* "Многие из нас, работающих в

области объектно-ориентированных данных, боролись за то, чтобы выделить суть «объектной ориентированности» для использования в системах баз данных... Мои собственные взгляды на то, что является наиболее важным [средством] объектно-ориентированных баз данных, со временем изменились. Сначала я думал, что это — модель наследования и модель обмена сообщениями. Позже я пришел к выводу, что более важны идентичность объектов, поддержка сложного состояния и инкапсуляция поведения. Но в последнее время, когда я услышал мнение пользователей объектно-ориентированных СУБД о том, чему они придают наибольшее значение в таких системах, я понял, что *решающее свойство — возможность расширения типов.* Идентичность, сложное состояние и инкапсуляция по-прежнему важны, но лишь настолько, насколько они поддерживают создание новых типов данных".

Melton J. Advanced SQL: 1999 — Understanding Object-Relational and Other Advanced Features. San Francisco, Calif.: Morgan Kaufmann, 2003.

Учебное руководство по средствам языка SQL, описанным в разделе 26.6, и другим "усовершенствованным" средствам SQL: 1999, включая SQL/MED (см. главу 21), SQL/OLAP (см. главу 22) и отдельный стандарт SQL/MM [26.25].

26.33. Patel J. et al. Bilding a Scalable Geo-Spacial DBMS: Technology, Implementation, and Evaluation // Proc. ACM SIGMOD Int. Conf. on Management of Data. — Tucson, Ariz. — May 1997.

Цитата из резюме: "В этой статье представлен ряд новых методов организации параллельной работы геопространственных систем баз данных и обсуждается их реализация в объектно-реляционной системе баз данных Paradise" (см. [26.20]).

26.34. Ramakrishnan R., Gehrke J. Database Management Systems (3d ed.) // Boston, Mass.: McGraw-Hill, 2003.

26.35. Ramasamy K., Patel J. M., Naughton J.F., Kaushik R. Set Containment Joins: The Good, the Bad, and the Ugly // Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt. — September 2000.

26.36. Rowe L.A., Stonebraker M.R. The Postgres Data Model // Proc. 13th Int. Conf. on Vary Large Data Bases. — Brighton, UK. — September 1987.

26.37. Samet H. The Design and Analysis of Spatial Data Structures. — Reading, Mass.: Addison-Wesley, 1990.

26.38. Saracco CM. Universal Database Management: Guide to Object/Relation Technolody. — San Francisco, Calif.: Morgan Kaufmann, 1999.

Интересный обзор, написанный на высоком уровне. Однако отметим, что его автор не отвергает (как, кстати, и Стоунбрейкер в [26.41]), очень сомнительную форму наследования, включая некоторый вариант идеи подтаблиц и супертаблиц (в отношении которой автор настоящей книги высказывал критические замечания, начиная со статьи [14.13]). Данный вариант существенно отличается от версии, включенной в язык SQL. Поясним это на примере. Предположим, что таблица PGMR (программисты) определена как подтаблица таблицы EMP (служащие).

В таком случае, по мнению Саракко и Стоунбрейкера, таблица EMP содержит строки лишь для служащих, которые не являются программистами, в то время как в языке SQL эта таблица рассматривается как содержащая строки с данными обо *всех* служащих, как было описано в разделе 26.6.

Seshadri P., Paskin M. PREDATOR: An OR-DBMS with Enhanced Data Types // Proc. 1997 ACM SIGMOD Int. Conf. on Management of Data. — Tucson, Ariz. — May 1997.

Приведем цитату из этой статьи: "Основной замысел системы PREDATOR — предоставить механизм для каждого типа данных, чтобы можно было задавать семантику его методов". Эта семантика затем используется для оптимизации запросов.

26.40. Stonebraker M. The Design of the POSTGRES Storage System // Proc. 13th Int. Conf. on Very Large Data Bases. — Brighton, UK. — September 1987.

26.41. Stonebraker M., Brown P. (совместное Moore D.) Object/Relational DBMSs: Tracking the Next Great Wave (2nd edition). — San Francisco, Calif.: Morgan Kaufmann, 1999.

Эта книга представляет собой руководство по объектно-реляционным системам. В большой степени — фактически почти исключительно — она базируется на продукте Universal Data Option для СУБД Informix Dynamic Server. Этот продукт основывается на более ранней системе, которая называлась Illustra (коммерческий продукт, в разработке которого лично принимал участие сам Стоунбрейкер). В [3.3] можно найти дополнительный анализ и критику этой книги. См. также аннотацию к [26.38].

26.42. Stonebraker M., Kemnitz G. The Postgres Next Generation Database Management System//CACM. - October 1991. - 34, № 10.

26.43. Stonebraker M., Rowe L.A. The Design of Postgres // Proc. ACM SIGMOD Int. Conf. on Management of Data. — Washington, D.C. — June 1986.

В работе приведены основные описанные ниже цели создания системы Postgres. .

1. Обеспечение лучшей поддержки сложных объектов.
2. Предоставление пользователю средств расширения типов данных, операторов и методов доступа.
3. Обеспечение активных инструментов базы данных (программ предупреждения об аварийных ситуациях и триггеров), а также поддержка логического вывода.
4. Упрощение кода СУБД для восстановления после аварии системы.
5. Проектирование СУБД с учетом преимуществ оптических дисков, многопро цессорных рабочих станций и специальных чипов на основе сверхбольших ин тегральных схем.
6. Минимальное количество изменений реляционной модели (а может быть, даже отсутствие изменений).

26.44. Stonebraker M. et al. Third Generation Database System Manifesto // ACM SIGMOD Record.— September 1990. — 19, № 3.

Частично эта работа является ответом (или даже возражением) на "Манифест объектно-ориентированных баз данных" [20.2], [25.1], в котором (кроме всего прочего), по сути, полностью игнорируется реляционная модель (?!). Цитата из этой работы: "Системы второго поколения внесли большой вклад в развитие двух областей — непроцедурного доступа к данным и независимости от данных, и этим вкладом нельзя пренебрегать при разработке систем третьего поколения". В статье утверждается, что важнейшим требованием к *СУБД третьего поколения* является поддержка перечисленных ниже средств (автор настоящей книги немного перефразировал перечень, приведенный в оригинале).

1. Поддержка традиционных служб базы данных, а также более развитых объектных структур и правил
   - Система с более широким набором типов.
   - Наследование.
   - Функции и инкапсуляция.
   - Необязательные идентификаторы кортежей, присваиваемые системой.
   - Правила (например, правила целостности), не связанные с конкретными объектами.
2. Преемственность по отношению к СУБД второго поколения
   - Использование навигации только в крайнем случае.
   - Определения интенсиональных и экстенсиональных множеств (под этим подразумеваются коллекции, автоматически сопровождаемые системой, и коллекции, которые сопровождаются пользователем вручную).
   - Обновляемые представления.
   - Кластеризация, индексирование и т.д., скрытые от пользователя.
3. Поддержка открытых систем
   - Поддержка нескольких языков.
   - Перманентность, ортогональная к типам.
   - Поддержка языка SQL (который в этой работе назван "межгалактическим" языком данных).
   - Запросы и результаты должны находиться на самом низком уровне взаимодействия между клиентом и сервером.

В [3.3] содержится подробный анализ и критика этой статьи. См. также [26.31].

26.45. Wang H., Zaniolo C. Using SQL to Build New Aggregates and Extenders for Object-Relational Systems // Proc. 26th Int. Conf. on Very Large Data Bases, Cairo, Egypt. — September 2000.

26.46. Wilkes M.V. Software and the Programmer// CACM. — May 1991. — 34, № 5.

ГЛАВА 27

World Wide Web и XML

27.1. ВВЕДЕНИЕ

Примечание. Первый вариант этой главы был написан Ником Тиндоллом (Nick Tindall) из компании IBM.

World Wide Web и XML — это горячие темы; им уже посвящено много книг, а в дальнейшем, безусловно, будет написано еще больше. А поскольку в данной книге рассматривается именно проблематика баз данных, автор стремился обойтись без изложения в ней подробных сведений о Web или XML, по крайней мере, в той части, которая не относится к ее основной теме. Поэтому, что касается Web, то здесь приведена лишь такая основная информация (в разделе 27.2), которая позволяет подготовить определенный контекст для обсуждения материала, изложенного в следующих разделах. А что касается XML, то по этому языку необходимо привести гораздо больше информации, поэтому ему посвящено три раздела: в разделе 27.3 приведен общий обзор, а в разделах 27.4 и 27.5, соответственно, рассматриваются вопросы определения данных XML и манипулирования данными XML. Затем в разделе 27.6 рассматривается связь между XML и базами данных. Безусловно, эта последняя тема и была основной причиной, по которой в данную книгу вообще была включена эта глава, но автор намеренно игнорирует ее (по большей части) до раздела 27.6. Наконец, в разделе 27.7 описаны соответствующие средства SQL, а в разделе 27.8 представлено резюме.

27.2. WEB И INTERNET

Термины *Web* и *Internet* часто используются так, как если бы они были взаимозаменяемыми, но, строго говоря, они обозначают разные понятия. Различия между ними можно охарактеризовать следующим образом: Web — это гигантская база данных (хотя она и не спроектирована в соответствии с общепринятыми принципами создания баз данных), а Internet — это столь же гигантская сеть, по которой распределена эта база данных.

Примечание. Как должно быть известно читателю, доступ к Web — это не единственная услуга, предоставляемая в Internet; в этой сети можно пользоваться программами чтения новостей, средствами мгновенной передачи сообщений, электронной почтой, протоколами ftp, telnet и т.д., но в данной главе нас интересует именно Web. В задачу настоящей книги не входит изложение подробных сведений о программах чтения новостей, средствах мгновенной передачи сообщений и других службах.

Сеть Internet развилась из сети Arpanet, которая была создана по проекту, разработанному в конце 1960-х годов под руководством агентства DARPA (Department of Defense Advanced Research Projects Agency— Управление перспективных исследовательских программ) Министерства обороны США для соединения всех разнообразных существовавших в то время правительственных и академических сетей США в единую "суперсеть" с общим протоколом связи, называемым TCP/IP (Transmission Control Protocol/Internet Protocol — протокол управления передачей/межсетевой протокол). Но сеть Internet как таковая (т.е. до развертывания Web) еще не была интегрирована в той степени, в какой могла бы быть; пользователям все еще приходилось использовать для доступа к информации целый ряд различных механизмов: протоколы ftp, gopher, archie, всевозможные разновидности электронной почты и т.д. Например, если требовалось перейти по ссылке, обнаруженной в каком-то документе, то для этого, как правило, приходилось выполнять следующие действия: отыскивать имя соответствующего файла с помощью электронной почты или системы досок объявлений, регистрироваться на сервере archie по протоколу telnet для определения местонахождения этого файла, регистрироваться в той системе, где хранился файл, с помощью протокола ftp, переходить в соответствующий каталог в этой системе, копировать файл в свою собственную систему и, наконец, выбирать в своей системе подходящую программу для отображения этого файла.

Система Web была изобретена Тимом Бернерсом-Ли (Tim Berners-Lee) в 1989—1990 годах в качестве основы для определения упрощенных методов решения всех сложных задач дистанционного доступа к информации [27.2]. Основным понятием Web стал формат гипертекста, предложенный за несколько лет перед этим Тэдом Нельсоном (Ted Nelson) [27.19]. Гипертекст— это способ структуризации информации, который позволяет включать в текстовые документы информацию о местонахождении других документов и файлов (или компонентов других документов и файлов) с помощью встроенных ссылок. Важный вклад Бернерса-Ли в решение этой задачи состоял в том, что он сумел реализовать операцию перехода по ссылкам в графическом браузере, который мог теперь применяться для интеграции разных видов информации в одном окне; конечный результат состоял в том, что пользователи получили возможность обращаться к любой необходимой им информации и просматривать ее в браузере с помощью одного щелчка мыши, вместо применения всех отдельных команд и процедур, которые они были вынуждены использовать раньше. Бернерс-Ли сумел добиться такого замечательного упрощения, определив описанные ниже средства.

- Механизм идентификации и формирования ссылок на документы и другие ресурсы, получивший название **URL** (Uniform Resource Locator— унифицированный локатор информационного ресурса). В дальнейшем на его основе было введено обобщенное понятие унифицированного идентификатора информационного ресурса (Uniform Resource Identifier — URI).
- Язык **HTML** (Hypertext Markup Language — язык разметки гипертекста), предназначенный для создания документов и включения в них инструкций, определяющих способ отображения этих документов.
- Протокол **HTTP** (Hypertext Transfer Protocol — протокол передачи гипертекста), с помощью которого может осуществляться передача таких документов по Internet.

Примечание. Дополнительные сведения о языках разметки и HTML приведены в следующем разделе.

Итак, как уже было сказано, Web — это гигантская база данных. Пользователи с помощью Web-браузера обращаются к этой базе данных, распределенной по многочисленным узлам (называемым *Web-узлами),* каждый из которых имеет собственный Web-сервер и обозначен своим собственным URL. Каждый узел содержит множество Web-страниц, а каждая страница имеет связанный с ней корневой документ, который обозначает, кроме всего прочего, каким должен быть способ отображения этой страницы. Как и все документы, корневой документ обычно включает ссылки[1] URL на всевозможную дополнительную информацию различных типов (текст, изображения, звуки, видео-информация и т.д.), находящуюся на разных узлах, которая должна быть представлена на странице, но пользователь воспринимает всю эту информацию как единое целое, поскольку чаще всего пользователя интересует только URL первоначальной страницы (и он не обязан знать, откуда берется вся остальная информация). Но после вывода страницы в окно браузера отображаются также содержащиеся в ней ссылки, а после щелчка пользователя на такой ссылке браузер представляет соответствующую информацию в том же окне (или в новом окне).

Примечание. Некоторые Web-страницы позволяют пользователю запрашивать дополнительные сведения, заполняя формы. Одним из важных частных случаев такого рода являются машины поиска. Как правило, машина поиска принимает заданный поисковый запрос (например, строку "Camelot") и возвращает список Web-узлов, содержащих соответствующую информацию. Для того чтобы подобный поиск можно было выполнять за достаточно короткое время, в машине поиска используется всеобъемлющие индексы ключевых слов, которые присутствуют в миллионах документов, хранящихся в Web. Такие индексы создаются и поддерживаются так называемыми "Web-навигаторами" (Web crawler), которые непрерывно работают в Web, осуществляя выборку Web-страниц и регистрируя в своей базе данных возможные параметры поиска для дальнейшего использования.

На каждом конкретном узле информация может храниться в файлах операционной системы, но все чаще применяется способ хранения такой информации в базах данных (в базах данных SQL и прочих), поэтому Web-серверы должны обладать способностью взаимодействовать с системами управления базами данных. В разделах 27.6 и 27.7 приведены определенные сведения о том, как может быть организовано такое взаимодействие.

---

[1] Такая дополнительная информация может быть также встроена в саму страницу.

27.3. КРАТКИЙ ОБЗОР ЯЗЫКА XML

Название **XML** является сокращением от "Extensible (не extensible!) Markup Language" (расширяемый язык разметки). Неформально документ **XML** можно определить как документ, созданный с использованием средств XML. Ниже приведен простой пример. Обратите внимание на то, как широко используются в нем угловые скобки, "<" и ">" (их не следует путать с угловыми скобками, которые применяются в этой книге в грамматических определениях, которые представлены в форме Бэкуса-Наура).

```
<?xml version="1.0"?>
<greeting kind="succinct">Hello, world.</greeting>
```

Первая строка в этом документе представляет собой **объявление XML** (в котором не показаны некоторые вспомогательные конструкции); такое объявление обычно можно найти в каждом документе XML, хотя оно и не является обязательным. Во второй строке представлен **элемент XML,** состоящий из **начального дескриптора,** некоторых **символьных данных** и **конечного дескриптора.** (Вообще говоря, любой элемент может содержать символьные данные или другие элементы, или сочетание тех и других.) Символьными данными является строка "Hello, world. ", начальный дескриптор — это **разметка,** предшествующая этой строке, а конечный дескриптор — это **разметка,** которая следует за ней. (Неформально неуточненный термин *дескриптор* используется также для обозначения начального дескриптора и соответствующего ему конечного дескриптора, вместе взятых.) Дескрипторы обозначаются тем именем, которое присвоено им разработчиком документа; кроме того, считается, что это имя обозначает **тип элемента,** поэтому в данном примере дескриптор является дескриптором типа "greeting", а элемент — элементом "greeting". Показанная ниже спецификация в начальном дескрипторе представляет собой **атрибут XML** (он не имеет ничего общего с атрибутами, которые рассматриваются в реляционном контексте).

```
kind="succinct"
```

Здесь именем атрибута является "kind", а его значением — строка "succinct".

Как показывает этот весьма несложный пример, документ XML вполне может рассматриваться просто как символьная строка. Эта символьная строка состоит изданных и разметки, где разметка представляет собой метаданные, предназначенные для описания этих данных[2]. Как правило, документы XML предназначены для чтения и интерпретации как людьми, так и компьютерами; в частности, они предназначены для упрощения обработки данных прикладными программами.

Примечание. В связи с этим последним замечанием следует отметить, что благодаря разметке прикладные программы приобретают способность приспосабливаться к существенным изменениям формата данных, например, как описано ниже.

- Поскольку элементы включают разграничительные дескрипторы, различные "экземпляры" внешне "одинакового" элемента вовсе не обязательно должны иметь постоянный размер, а могут изменяться от одного документа к другому или даже в одном документе.

[2] Данные, которые непосредственно содержатся в документе, должны быть именно символьными данными, но благодаря использованию встроенных ссылок документ фактически может содержать, например, изображения, видеозаписи и другие типы несимвольных данных. Такие ссылки рассматриваются как часть разметки.

- (Еще более важное замечание!) Новые элементы (т.е. элементы каких-то новых типов) могут вводиться в существующий документ в любое время, не оказывая отрицательного воздействия на работу пользователей (в частности, прикладных программ).

Иными словами, XML позволяет решать некоторые проблемы обмена данными, при условии, что производители и потребители данных согласуют друг с другом способы интерпретации разметки. В отличие от этого, при использовании традиционных протоколов с фиксированным форматом, таких как протокол электронного обмена данными (Electronic Data Interchange — EDI), для внесения изменений в формат данных необходимо внести соответствующие изменения в программное обеспечение, применяемое не отдельными, а всеми производителями и потребителями рассматриваемых данных.

Языки разметки

Для того чтобы оценить важность некоторых причин создания языка XML, необходимо ознакомиться с тем, каким образом он создавался на основе более ранних языков разметки. Языки разметки никогда не были предназначены для использования в качестве обычных языков программирования; их назначение (по меньшей мере, как оно рассматривалось с самого начала) состояло лишь в том, чтобы с их помощью можно было создавать текстовые файлы, включающие команды форматирования (разметку), которые могли восприниматься соответствующим текстовым процессором. В одних случаях разметка обозначалась с помощью двоичного кода, а в других — с помощью обычного текста, но применяемые при этом языки разметки всегда оставались собственностью отдельных компаний. Например, в компании IBM для форматирования руководств пользователей и других подобных документов использовался собственный язык текстовой разметки, Script. Ниже в качестве примера приведена выдержка из типичного файла Script.

```
.sp 2
.il 3m;You should specify the ;.us on;first;.us off; parameter
as PRIVATE. This specification will allow the processor to
complete the conversion without further input.
.br
```

Применяемая здесь разметка передает программе форматирования указания отступить вниз на две строки (". sp 2"), сделать отступ в первой строке текста на три пробела с шириной em (" .il 3m"), подчеркнуть слово "first" (".us on" и ".us off"), а затем перейти на следующую строку (". br").

Отметим, что один из недостатков языка Script и аналогичных языков состоял в том, что разметка носила весьма процедурный характер (т.е. была основана на использовании не описаний, а команд), не говоря уже о том, что эта разметка управляла только форматированием документов и к тому же была предназначена для устройств определенного типа (как правило, монохромных построчно-печатающих устройств). Именно в целях устранения этих недостатков три исследователя[3] из компании IBM предложили обобщенный язык разметки (Generalized Markup Language — GML). Основное различие между

[3] Тот факт, что первые буквы в сокращенном обозначении этого языка совпадают с первыми буквами фамилий его создателей, Чарльза Голдфарба (Charles Goldfarb), Эдварда Мошера (Edward Mosher) и Раймонда Лори (Raymond Lorie), — не случайное совпадение.

GML и языками наподобие Script состоит в том, что разметка в GML является в значительно большей степени описательной (или декларативной), чем разметка, применявшаяся в этих ранних языках, которая в основном носит процедурный характер. Ниже еще раз показан приведенный выше пример на языке Script, но в данном случае он представлен на языке GML.

```
<p>You should specify the <empl>first</empl> parameter
as PRIVATE. This specification allows the processor
to complete the conversion without further input.
```

Теперь эта разметка просто сообщает программе форматирования о том, что текст представляет собой абзац ("<p>" — сокращение от "paragraph"), а не просто подробно описывает компоновку такого абзаца ("отступить вниз на две строки" и т.д.). Кроме того, она сообщает программе форматирования, что слово "first" должно иметь первый уровень выделения ("<empl>" и "</empl>" — сокращение от "emphasis 1"), но не указывает, что в качестве такого выделения должно быть выполнено подчеркивание.

Примечание. В этом примере автор намеренно допустил немного вольное обращение с языком GML; в частности, использовал для обозначения дескрипторов угловые скобки вместо обычно применяемых в этом языке символов двоеточия (": "). В данном изложении такие отклонения от обычной практики не имеют особого значения.

Таким образом, разметка GML оказалась более описательной, но она все еще нацелена в основном на представление, или отображение текста (хотя и позволяет проще решать некоторые другие задачи, такие как подсчет количества абзацев). В частности, пользователи вынуждены ограничиваться только теми дескрипторами, которые встроены в этот язык. В отличие от него, стандартный язык GML (Standard GML — SGML), который представляет собой расширенную форму GML, позволяет пользователям определять свои собственные дескрипторы и придавать им любой желаемый смысл[4]. Используя это средство, можно расширить приведенный выше пример для определения структуры данных до мельчайших подробностей, например, следующим образом.

```
<paragraph>
  <sentence> <subj
  ect>You</subject>
  <verb> should specify</verb>
  <object> the
          <adjective><empl>first</emp1></adject
          ive> parameter</obj ect>

  </sentence>
  <sentence>

  </sentence>
</paragraph>
```

Примечание. Обратите внимание на тот факт, что в этом примере элемент "object" содержит и данные в виде символьной строки, и другой элемент ('adjective").

Но, как показывает этот пример, SGML в действительности не является языком разметки как таковым (и поэтому фактически название "SGML" не полностью соответствует

[4] Приведем в этой связи интересную цитату: "Даже в самой небольшой организации причиной большинства конфликтов становится отсутствие четких определений и единого понимания смысла используемых слов" [27.4].

назначению этого языка). Его, скорее, можно назвать *метаязыкам,* поскольку в нем предусмотрены правила, позволяющие пользователям определять собственные наиболее подходящие для них языки разметки, в частности, имеющие собственные дескрипторы, определяемые пользователями[5]. Одним из таких языков является HTML; это означает, что HTML определен в терминах SGML и может рассматриваться как конкретное приложение SGML (здесь термин *приложение SGML* обозначает язык, который определен с применением правил метаязыка SGML, а не прикладную программу, в которой используется SGML). Но, к сожалению, в HTML не удалось сохранить чисто описательный характер GML, и вместо этого в дополнение к структурной и семантической разметке снова была введена конкретная форматирующая разметка. Фактически некоторые дескрипторы HTML относятся ко всем трем типам разметки одновременно; например, такой дескриптор HTML, как "<H1>", определяет одновременно и заголовок уровня 1 (структура) и название страницы (семантика), а также может содержать необязательный атрибут с обозначением шрифта (форматирование).

Разработка языка XML

Язык XML [27.25] был первоначально разработан в 1996 году наблюдательным советом по пересмотру языка SGML под руководством консорциума World Wide Web Consortium, W3C (основанным Бернерсом-Ли в 1994 году) в целях устранения некоторых недостатков языков SGML и HTML. Недостатком SGML было лишь то, что этот язык оказался слишком большим и сложным для того, чтобы его можно было легко поддерживать в Web. А что касается HTML, то он обладает в основном двумя описанными ниже недостатками.

- Как уже было сказано, этот язык не позволяет должным образом отделить друг от друга метаданные, которые описывают структуру, семантику и форматирование.
- Более того, этот язык не исключает возможности создавать документы, которые нарушают принципы "формальной правильности", т.е. фактически этот язык не препятствует созданию и использованию документов, нарушающих[6] его собственные синтаксические правила! Такая проблема возникла в связи с тем, что на рынке одновременно представлено несколько браузеров, а все они (вполне естественно) конкурируют друг с другом; и если неправильно сформированный документ D может быть успешно отображен браузером A, но не браузером B, то этот факт воспринимается как недостаток не документа D, а скорее браузера B.

XML, как и его предшественник SGML, в действительности является не языком, а метаязыком (и поэтому его название "XML" также не полностью ему соответствует); фактически язык XML является строгим подмножеством языка SGML. Эта его особенность подчеркнута в приведенной ниже выдержке из спецификации XML [27.25].

Расширяемый язык разметки (Extensible Markup Language — XML) — это подмножество SGML... Его назначение состоит в обеспечении возможности использовать универсальные средства SGML для передачи, приема и обработки документов в Web с

[5] По сути, такое же замечание остается справедливым и по отношению к GML.

[6] К сожалению, ко времени написания данной книги аналогичная тенденция стала обнаруживаться и в развитии языка XML.

помощью таких же средств, которые в настоящее время предоставляются языком HTML. Язык XML спроектирован с учетом требований удобства реализации и предназначен для обеспечения функциональной совместимости и с языком SGML, и с языком HTML.

XML позволяет снова восстановить описательный характер разметки (т.е. языки разметки, определяемые с помощью XML, включают только описательную разметку). Но следует обратить особое внимание на то, что XML как таковой не предписывает какой-либо конкретный смысл применяемой разметке.

Несмотря на то, что такая цель была определена при его создании, XML еше не смог в значительной степени заменить HTML в качестве наиболее предпочтительного средства определения Web-страниц (браузер, который не поддерживает XML, все еще способен при обычных обстоятельствах отображать Web-страницы должным образом). С другой стороны, применение XML в других областях растет и вглубь, и вширь, например, теперь этот язык используется для таких различных целей, как формирование файлов конфигурации, разработка форматов обмена данными для инструментальных средств анализа, создание новых протоколов обмена сообщениями между прикладными программами в корпоративных сетях (а также обмена данными между этими программами). В основном вследствие этого факта постоянно возрастает необходимость в создании средств, позволяющих хранить данные XML в базах данных. Ниже в качестве иллюстрации приведен ряд примеров таких ситуаций, в которых может потребоваться хранить данные XML в базе данных.

- Рассмотрим обычно применяемую в этой книге переменную отношения деталей P. Допустим, что потребовалось расширить эту переменную отношения для включе ния дополнительного атрибута DRAWING (рис. 27.1), значением которого в любом конкретном кортеже является чертеж рассматриваемой детали, представленный с помощью какого-то конкретного языка, производного от XML (см. следующий подраздел) и называемого *языком масштабируемой векторной графики* (Scalable Vector Graphics — SVG). Следует отметить, что каждое такое значение представля ет собой целый документ SVG, и поэтому вся база данных может рассматриваться как "объектно-реляционная" база данных в том смысле этого термина, который был определен в предыдущей главе.

  Примечание. Фактически утверждение, что значения DRAWING являются чертежами как таковыми, является не совсем точным; эти значения, скорее, можно назвать документами XML, которые интерпретируются определенной прикладной программой для создания таких чертежей (см. рис. 27.1).

- Аналогичным образом, в переменную отношения P можно ввести атрибут DESCRIPTION, значением которого в любом конкретном кортеже является доку мент XML с описанием рассматриваемой детали и пояснением правильных спосо бов ее использования.

Безусловно, XML может также применяться для представления тех разновидностей данных, которые можно найти в более традиционных базах данных. Например, в XML можно представить торговые заказы, каталоги деталей и складские журналы. Таким образом, база данных (разумеется, нереляционная) может состоять исключительно из документов XML. Такой вариант кратко рассматривается в разделе 27.6.

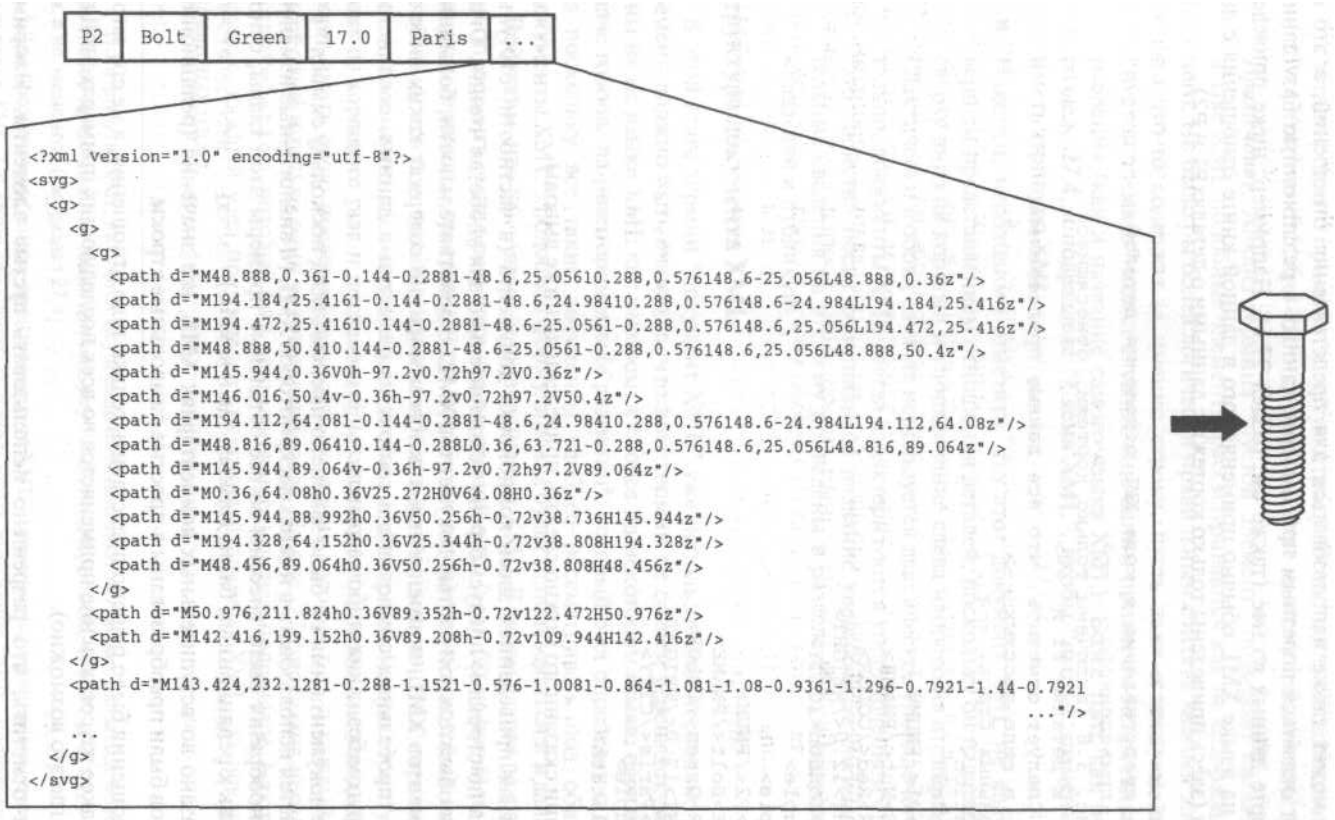


Рис. 27.1. *Добавление значения атрибута* DRAWING *в переменную отношения деталей (пример)*

Язык XML может также использоваться для представления отношений, а это его свойство может оказаться полезным при импорте данных в реляционную базу данных или при экспорте данных из нее (также см. раздел 27.6). Например, ниже приведено представление на языке XML обычно применяемого в данной книге отношения с данными о деталях (здесь приведены только кортежи с данными о деталяхP1 и P2).

```
<?xml version="1.0"?>
<!-- Это - представление в коде XML отношения деталей,
          приведенного    --> <!-- на рис. 3.8
(включены только кортежи, относящиеся
          к деталям P1 и  --> <!-- P2). Следует
отметить, что все данные представлены
        в виде простых -->
<!-- символьных строк. -->
<PartsRelation> <PartTuple>
  <PNUM>P1</PNUM>
  <PNAME>Nut</PNAME>
  <COLOR>Red</COLOR>
  <WEIGHT>12.0</WEIGHT>
  <CITY>London</CITY>
  </PartTuple>
  <Part Tuple>
<PNUM>P2</PNUM>
<PNAME>Bolt</PNAME>
<COLOR>Green</COLOR>
<WEIGHT>17.0</WEIGHT>
<CITY>Paris</CITY>
</PartTuple>
</PartsRelation>
```

На основании сказанного можно сделать приведенные ниже выводы.

- Конкретная приведенная здесь компоновка документа (в частности, отступы и разбивка по строкам) предусмотрена исключительно для удобства чтения. Она не является обязательной частью документов XML. В действительности, большинст во документов XML, применяемых в деловом мире, не содержат каких-либо по добных "пробельных символов", а состоят из разметки и данных, соединенных друг с другом без каких-либо разрывов.
- В тексте документа имя P# было заменено именем PNUM, поскольку символ "#" не разрешается использовать в именах дескрипторов XML (и аналогичные изменения для единообразия были внесены во все остальные примеры этой главы, относя щиеся как к реляционным базам данных, так и к XML).
- Как указано во вступительных **комментариях XML,** все значения (реляционных) атрибутов были преобразованы в простые символьные строки.
- Для упрощения было решено представить только тело отношения, а не его заголо вок (и такое же соглашение применяется во всех последующих примерах этой гла вы там, где это возможно).
- Следует отметить, что разрешено использовать **пустые** элементы. Например, предположим, что для некоторых деталей не предусмотрено использование ат рибута с использованием цвета, и для представления значения COLOR для такой

детали решено применять пустую строку. В таком случае достаточно ввести в спецификацию <COLOR></COLOR>, или (более кратко) просто <COLOR/>.

Примечание. Анализ термина "пустой элемент" позволяет сделать вывод, что в языке XML подобные элементы рассматриваются как не содержащие вообще ничего. Но было бы более логически правильным считать, что эти элементы все равно что-то содержат (а именно строку, пусть даже и пустую), поэтому термин "пустой элемент" может служить еще одним примером неправильного словоупотребления (как и название самого языка XML). Более того, как будет показано в разделе 27.4, в подразделе "Схема XML", якобы "пустые элементы" также могут иметь атрибуты XML.

- Наконец, необходимо отметить, что этот документ XML не является таким уж верным представлением отношения деталей, поскольку он налагает упорядочение сверху вниз на кортежи и упорядочение слева направо на атрибуты этих кортежей (фактически в обоих случаях используется лексическое упорядочение). И действительно, документы XML всегда характеризуются упорядочением своих элементов[7] (это свойство носит официальное название *упорядочение документа)*. Дополнительные сведения на эту тему приведены в разделе 27.6, в подразделе "Принцип «Разделяй и публикуй»".

Структура документа XML

В этой главе термин "документ XML" уже использовался несколько раз, но к этому времени должно быть очевидно, что этот термин не совсем верен: документы представлены не на языке XML как таковом, а скорее на некотором языке, производном от XML, иначе говоря, на некотором языке разметки[8], который был определен с помощью XML. Но поскольку все такие документы удобнее рассматривать под общим названием "документы XML", подобная формулировка будет использоваться и в дальнейшем.

Допустим, что XD — язык, производный от XML. В таком случае разработчик XD должен определить смысл разметки, применяемой в XD (хотя бы неформально), ввести операторы и написать программы для обработки документов, в которых используется XD. Допустим, что D — такой документ. Поэтому этот документ на определенном уровне абстракции может рассматриваться как имеющий иерархическую структуру, которая состоит из корневого узла и последовательности дочерних узлов (где каждый дочерний узел может иметь свою последовательность дочерних узлов и т.д., а каждый дочерний узел имеет один и только один родительский узел). На рис. 27.2 показана иерархия документа PartsRelation, который был приведен в предыдущем подразделе. Как показывает этот рисунок, каждый узел представляет некоторый элемент XML, за исключением узлов, описанных ниже.

---

[7] Но атрибуты XML являются неупорядоченными, поэтому может оказаться более предпочтитель ным способ представления таких данных, как PNUM, PNAME, COLOR, WEIGHT и CITY, с помощью атрибу тов, а не элементов (см. раздел 27.4).

[8] Для обозначения "языка, производного от XML" применяется официально утвержденный термин *приложение XML*. Но автор предпочитает использовать выражение "язык, производный от XML", по скольку при этом исключается вероятность того, что термин *приложение XML* будет по ошибке рассмат риваться как "прикладная программа, в которой используется XML".

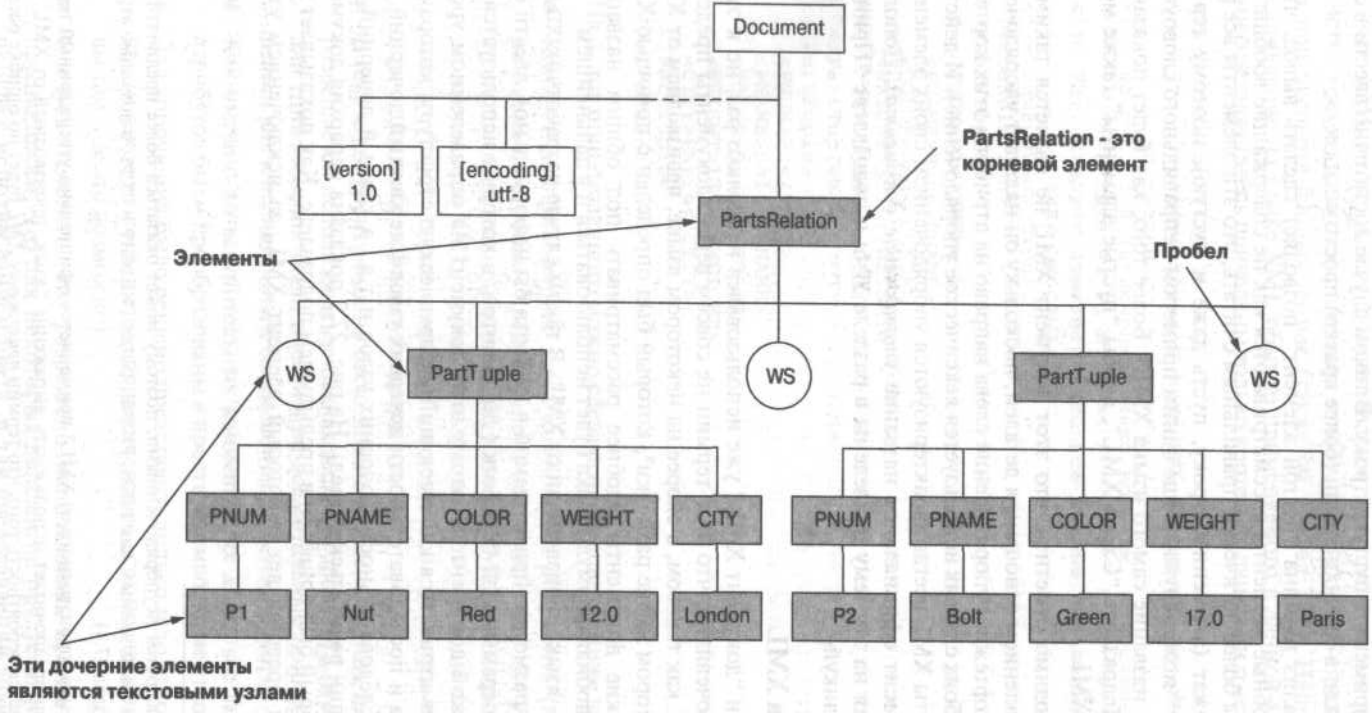


Рис. 27.2. Инфонабор документа `PartsRelation` (немного упрощенный)

- Корневой узел, или узел документа, который представляет весь документ (обратите внимание на то, что он не соответствует корневому элементу документа!).
- Листовые узлы, которые представляют символьные данные.

  Примечание. Листовые узлы используются также для представления, во-первых, атрибутов XML (но в данном примере атрибуты отсутствуют) и, во-вторых, различных элементов, таких как комментарии XML, пробельные символы и т.д. (на данном рисунке показано несколько таких узлов).

Вся эта иерархия называется информационным набором или **инфонабором** ("infoset" — сокращение от information set) для рассматриваемого документа, а интерфейс прикладного программирования (Application Programming Interface — API) для этой иерархии предусмотрен в виде средства XML, называемого **объектной моделью документа** (Document Object Model — DOM) [27.24]. С использованием такого API-интерфейса (кроме всего прочего) в прикладной программе можно осуществлять выборку, вставку, удаление и модификацию узлов[9].

Очевидно, что при использовании многих разных документов XML все они могут соответствовать одной и той же общей иерархической структуре (т.е. соответствующие инфонаборы могут иметь одну и ту же универсальную структуру). Например, может быть принято соглашение (по меньшей мере, в интересах текущего описания), что каждая книга состоит из названия, предисловия (необязательного), последовательности глав, последовательности приложений (необязательных) и предметного указателя (необязательного), каждая глава состоит из названия и непустой последовательности разделов, каждый раздел состоит из названия и непустой последовательности абзацев и т.д. Вместе с тем, безусловно, нельзя отрицать и того, что книги различаются по своей конкретной структуре, поскольку в одних отсутствует предисловие, в других — приложения, в третьих — предметный указатель, разные книги имеют различное количество глав и т.д. Поэтому, если каждая книга будет представлена с помощью документа XML, то эти документы будут характеризоваться существенными отличиями на уровне реализации конкретной структуры; вполне очевидно, что количество различных вариантов структуры будет значительно больше, чем обычно можно встретить среди кортежей отношения в реляционной модели. А поскольку отношения, как правило, рассматриваются как более жестко структурированные, а документы XML считаются обладающими гораздо более свободной структурой[10], иногда можно встретить такие утверждения, что отношения содержат структурированные данные, а документы XML — слабоструктурированные данные (иначе говоря, основаны на **слабоструктурированной модели данных).**

Но приведенные выше доводы фактически не выдерживают никакой критики. Истина заключается в том, что отношения не являются более или менее "структурированными", чем документы XML (безусловно, они имеют другую структуру, но нельзя отрицать и того факта, что все данные, представляемые в виде документа XML, могут быть столь же

---

[9] Может оказаться полезным такое замечание, что инфонабор для данного конкретного документа очень близок к той структуре, которая является "возможным представлением" (possrep) для данного документа, в том смысле, который определен в главе 5.

[10] Но, разумеется, нельзя утверждать, что они вообще не имеют структуры, поскольку "данные", полностью лишенные структуры, по определению представляют собой чистый шум, иными словами, термин "неструктурированные данные" содержит противоречие в своих компонентах.

успешно представлены в реляционной базе данных, — возможно, в виде кортежа, в виде множества кортежей или иным образом.) Но термин *слабоструктурированный* так широко используется в сфере обработки данных, что автор также вынужден его упомянуть.

Примечание. Справедливости ради следует лишь добавить, что в литературе можно также найти некоторые другие доводы в пользу применения этого термина, которые приведены ниже.

- В [26.35] указано, что данный термин возник в связи с тем, что часть данных имеет постоянную структуру, а другая часть изменяется (например, каждая книга имеет название, но не в каждой книге есть предметный указатель).
- Согласно [27.13], этот термин применяется по той причине, что полуструктурирбванные данные не имеют обычной схемы, а являются "лишенными схемы" и "описывающими сами себя". В [27.17] сказано то же самое, но дополнительно указано, что они "подобны объектам".
- В [27.23] в качестве причины применения этого термина указан тот факт, что дан ные "могут быть нерегулярными или неполными и [иметь] структуру, [которая] может изменяться быстро и непредсказуемо".
- Безусловно, структура, которой должен обладать документ XML, в значительной части зависит от проектировщика документа[11]; в определенном смысле структура налагается на данные проектировщиком, а разные проектировщики, разумеется, вправе применять различные структуры, подразделяя данные разными способами и выбирая разные дескрипторы. Например, поэму можно представить в виде еди ного сплошного фрагмента текста (<poem> ... </poem>),как последовательность стихов (<poem> <verse> ... </verse> ... </poem>) или многими другими спосо бами. Таким образом, можно утверждать, что термин *слабоструктурированный* возник с учетом этих соображений.

Но, по мнению автора, ни один из этих доводов не выдерживает тщательного критического анализа. В действительности, невозможно обнаружить существенное различие между "слабоструктурированной моделью" и структурными аспектами старомодной **иерархической** модели, которая была описана (и подвергнута критике) в главе 13 работы [1.5]. Но внимания, безусловно, заслуживает еще один нюанс — по определению предполагается, что модели данных вообще и иерархическая модель в частности, должны включать операции, но создается впечатление, что "слабоструктурированная модель" в основном (или даже исключительно) посвящена лишь описанию структуры данных.

Приведем еще одно, последнее замечание на эту тему — в литературе часто встречается мнение, что либо инфонабор (рассматриваемый с точки зрения универсальной интерпретации этого термина в [27.26]), либо, вообще говоря, "слабоструктурированная модель" — это "модель данных XML". Но это идеальное представление в значительной степени становится расплывчатым в связи с тем фактом, что в каждой из работ [27.24], [27.28], и (совместно) [27.27], и [27.29], а также в работе [27.26] как таковой, определена собственная "модель данных XML" (причем в одних из таких моделей предусмотрены операции, а в других — нет). Поэтому не совсем ясно, какая из этих моделей (если вообще таковая имеется) заслуживает право называться "моделью данных XML".

---

[11] А также от проектировщика типа документа, если он участвует в этой работе (см. раздел 27.4).

Языки, производные от XML, и стандарты XML

Язык XML как таковой был стандартизирован[12] только лишь относительно недавно (в феврале 1998 года), поэтому та степень признания, которой он сумел достичь за столь короткое время, является просто поразительной. Ко времени подготовки настоящего издания книги (начало 2003 года) имелось по меньшей мере 200 различных языков, производных от XML, причем некоторые из них оказались совершенно новыми, а другие стали результатом переработки более ранних спецификаций, основанных на языке HTML. Ниже в качестве примера приведена небольшая выборка из перечня этих производных языков.

- *CML.* Язык химической разметки (Chemical Markup Language):
- *ebXML.* Рекомендуемая замена для стандарта EDI, предназначенная для использо вания при обмене данными между предприятиями (Business-Two-Business — B2B).
- *MathML.* Язык математической разметки (Mathematical Markup Language).
- *PMML.* Стандарт информационной проходки.
- *WML.* Язык разметки для беспроводных систем (Wireless Markup Language).
- *XMLife.* Промышленный стандарт страхования.

Разработано также несколько спецификаций, применяемых в области прикладного программирования, и общепромышленных спецификаций, основанных на языке XML, часть которых перечислена ниже.

- *SOAP.* Простой протокол доступа к объектам (Simple Object Access Protocol) — один из компонентов новой технологии программирования Web Services, в кото рой приложения создаются по принципу объединения программных модулей, а обнаружение и доступ к этим модулям осуществляется в среде Web.
- *XML* Стандарт обмена данными между инструментальными средствами (XML Metadata Interchange).

Кроме того, принят целый ряд дополнительных стандартов (или документов, способных в дальнейшем стать стандартами), которые либо дополняют, либо зависят от стандарта XML (рис. 27.3). Ниже перечислены некоторые из наиболее важных таких документов.

Пространства имен в XML. Схема уточнения имен, которая позволяет совместно использовать различные словари XML[13], предотвращая при этом конфликты имен.

Информационный набор XML. Абстрактная модель структуры документов XML (см. предыдущий подраздел).

---

[12] Поскольку W3C — это консорциум организаций, а не формально утвержденная организация по стандартизации, то спецификация XML не является стандартом в полном смысле этого слова, а только "рекомендацией". Но это различие на практике не имеет значения. Кстати, следует отметить, что сама спецификация XML определена как производная от стандарта Unicode и в ней применяются понятия, которые определены в этом стандарте ([27.22]).

[13] Описание этого термина приведено в подразделе "Определения типа документа" раздела 27.4.

- *XML Schema.* Язык создания схем, которые описывают документы XML (см. раздел 27.4).
- *DOM.* Объектная модель документа (Document Object Model) — объектно-ориентированный API-интерфейс, позволяющий манипулировать инфонаборами XML (см. предыдущий подраздел).
- *XPath.* Язык XML Path, который предоставляет средства адресации *{обозначения пути)* и поэтому обеспечивает доступ к частям документа XML(см. раздел 27.5).
- *XPointer.* Язык XML Pointer, который создан на основе языка XPath, но предоставляет более развитые средства адресации.
- *XQuery.* Язык XML Query, или язык запросов XML (см. раздел 27.5).
- *XLink.* Язык связывания документов XML, позволяющий вставлять в документы XML такие элементы, которые создают и описывают связи между информационными ресурсами.
- *XSL и XSLT.* Язык таблиц стилей XML и язык преобразования таблиц стилей, которые при совместном использовании обеспечивают (а фактически поощряют) хранение информации форматирования отдельно от описательной разметки.

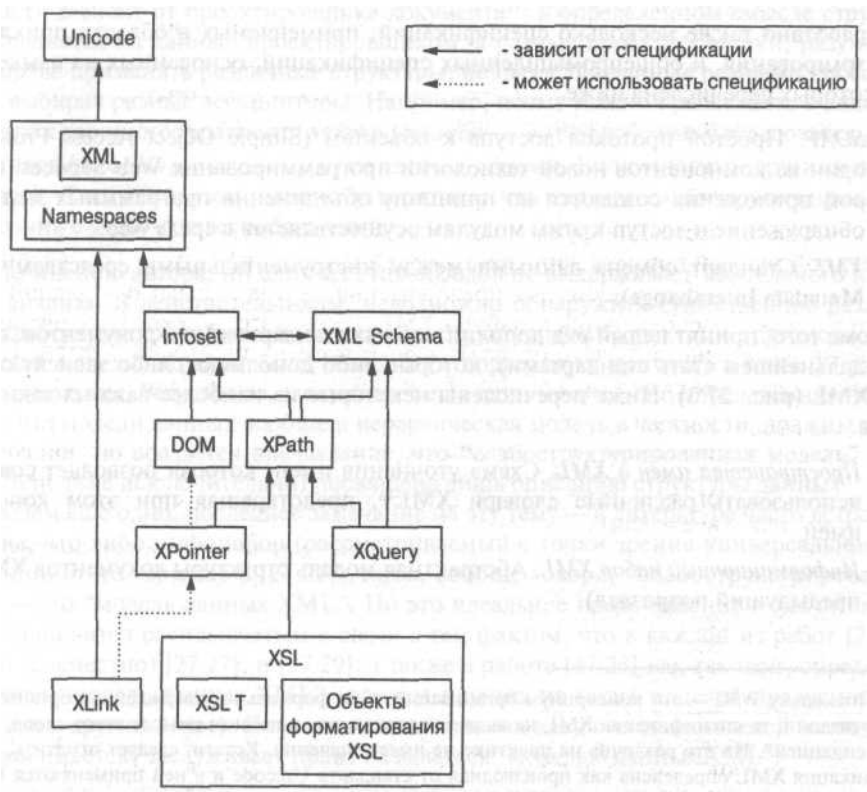


Рис. 27.3. Стандарты и спецификации, происходящие от спецификации XML или связанные с

27.4. ОПРЕДЕЛЕНИЕ ДАННЫХ XML

Как и с обычными данными базы данных, с любым документом XML, как правило, связана определенная описательная информация. Такую информацию можно задать **с** помощью либо определения типа документа (Document Type Definition — DTD), формируемого с использованием языка, который в данной книге именуется[14] **языком определения DTD** [27.25], либо с помощью схемы XML, которая формируется на основе языка **XML Schema** (имеющего название, которое вносит определенную путаницу [27.28]). Оба этих языка рассматриваются в настоящем разделе.

Определения типа документа

Язык определения DTD описан в документе [27.25], который представляет собой спецификацию XML (иными словами, определения типа документа являются частью самого стандарта XML). Кроме всего прочего, в [27.25] приведены перечисленные ниже сведения.

- Основные правила определения и использования языков разметки, основанных на XML (т.е. языков, производных от XML). Эти правила указывают, как различать разметку и символьные данные, как должным образом размещать попарно на чальные и конечные дескрипторы, как задавать комментарии и т.д.
- Правила проектирования программ, предназначенных для обработки документов XML (которые принято называть **синтаксическими анализаторами** XML). Такие программы предназначены для предоставления информации другим программам.

Кроме того (как уже было сказано), в [27.25] приведены правила определения документов DTD.

В качестве иллюстрации функциональных возможностей DTD воспользуемся пересмотренной версией примера документа PartsRelation из раздела 27.3. Основные изменения состоят в следующем: для представления цветов деталей и названий городов вместо элементов используются атрибуты XML; кроме того, в некоторых местах внутри документа разрешено включать необязательный элемент примечания NOTE. Итак, типичный документ PartsRelation может выглядеть так, как показано ниже.

```
<?xml version="1.0"?>
<!-- Это - представление в коде XML отношения деталей, приведенного
-->
<!-- на рис. 3.8 (включены только кортежи, относящиеся к деталям P1-
P3;  -->
<!-- значения COLOR и CITY теперь представлены в виде атрибутов
XML,   -->
<!-- а не элементов XML).                                          -->
<!DOCTYPE ... ><!-- См. приведенное ниже пояснение -->
<PartsRelation>
  <NOTE>Revised version</NOTE>
  <PartTuple
    CITY="London">
    <PNUM>P1</PNUM>
    <PNAME>Nut</PNAME>
```

[14] Термин "язык определения DTD" полностью расшифровывается как "язык определения определений типа документа", поэтому на первый взгляд может показаться, что в этом термине допущена некоторая избыточность. Но дело обстоит иначе, поскольку эти два "определения" обозначают разные понятия! Дополнительная информация на эту тему приведена в подразделе "Дополнительные сведения о языках, производных от XML" в самом конце текущего раздела.

```
    <WEIGHT>12.0</WEIGHT>
    <NOTE>Part color is Red by
  default</NOTE> </PartTuple>
  <PartTuple COLOR="Green"
CITY="Paris">     <PNUM>P2</PNUM>
<PNAME>Bolt</PNAME>
    <WEIGHT>17.0</WEIGHT>
  </PartTuple>
  <PartTuple CITY="Oslo"
    COLOR="Blue"> <PNUM>P3</PNUM>
    <PNAME>Screw</PNAME>
  <WEIGHT>17 . 0</WEIGHT> </PartTuple>
</PartsRelation>
```

Документы, имеющие показанную выше общую форму, могут иметь определение DTD, которое выглядит следующим образом (строки в нем пронумерованы для использования в дальнейших ссылках).

```
1. <!ELEMENT PartsRelation (NOTE?, PartTuple*)>
2. < ! ELEMENT NOTE (#PCDATA) >
3. <!ELEMENT PartTuple (PNUM, PNAME, WEIGHT,NOTE?)>
4. <!ATTLIST PartTuple
5. CITY (London | Oslo | Paris) «REQUIRED
6. COLOR (Red | Green | Blue) "Red">
7. <!ELEMENT PNUM («PCDATA)>
8. <!ELEMENT PNAME (#PCDATA)>
9. <!ELEMENT WEIGHT («PCDATA)>
```

Пояснение

- Строка 1. Каждый документ, который соответствует этому определению DTD, имеет один и только один корневой элемент, называемый PartsRelation. Этот корневой элемент содержит последовательность элементов PartTuple в количестве от нуля и больше, которым может предшествовать необязательный элемент

Примечание. Необязательные элементы обозначаются вопросительным знаком, а количество повторений "от нуля и больше" — звездочкой[15]. Если бы в этом документе вместо звездочки стоял знак плюса (т.е. если бы в него была включена строка PartTuple+, а не PartTuple*), то это означало бы "от единицы и больше", а не "от нуля и больше". Если кратность повторения элемента не указана, это означает, что кратность повторения составляет "один и только один".

Вообще говоря, определения DTD могут быть либо внутренними (т.е. непосредст венно включенными в описываемый ими документ), либо, внешними (т.е. вклю ченными в какой-то другой файл), как описано ниже.

- При использовании внутреннего определения текст DTD предшествует корневому элементу и должен быть заключен в парные разграничители. Открывающий разграничитель принимает следующую форму.

  <!DOCTYPE *document type name* [

[15] Звездочка — это оператор Клина (Kleene), который уже встречался в главе 21.

Здесь имя типа документа document *type name* (в данном примере PartsRe-lation) должно быть таким же, как и имя корневого элемента. Закрывающий разграничитель имеет следующую форму.

```
]>
```

- При использовании внешнего определения эти разфаничительные строки в доку менте отсутствуют, а перед корневым элементом каждого документа, в котором используется определение DTD, находится ссылка на рассматриваемый документ DTD (поэтому в случае внешнего определения проще обеспечить совместное ис пользование одного и того же определения DTD в нескольких документах). Такая ссылка может выглядеть следующим образом.

```
<!DOCTYPE PartsRelation SYSTEM "file:///c:/parts.dtd">
```

Здесь указано, что данное определение DTD находится в файле "parts.dtd". Поэтому даже в случае использования внешнего определения документ все еще имеет строку < ! DOCTYPE . . . >, в которой, кроме всего прочего, задано имя типа документа.

- Строка 2. Каждый элемент NOTE содержит "синтаксически проанализированные символьные данные" (parsed character data — #PCDATA), а это, неформально гово ря, означает обычный текст без какой-либо разметки. Строки 7, 8 и 9 являются аналогичными.
- Строка 3. Каждый элемент PartTuple содержит один и только один элемент PNUM, один элемент PNAME И ОДИН элемент WEIGHT (В указанном порядке), за ко торыми следует необязательный элемент NOTE.
- Строки 4-6. Каждый начальный дескриптор PartTuple включает атрибут CITY и необязательный атрибут COLOR (если заданы оба атрибута, их последовательность не играет роли). Значением атрибута CITY должны быть строки London, Oslo или Paris. Значением атрибута COLOR должны быть строки Red, Green или Blue, и по умолчанию предполагается, что этот атрибут имеет значение Red, если он не задан.

Ниже приведен ряд дополнительных замечаний.

- Во-первых, множество всех имен типов элементов и имен атрибутов, которые оп ределены в данном конкретном документе DTD, называется соответствующим ему словарем. Такое требование, чтобы эти имена были уникальными в рамках только одного словаря, не предъявляется (для разрешения конфликтов имен в случае необходимости может использоваться механизм пространств имен, упомя нутый в предыдущем разделе).
- Во вторых, в спецификации XML определены два уровня соответствия для доку ментов XML— *формальная правильность* и *допустимость*. Применительно к неко торому конкретному "текстовому объекту" (иными словами, к символьной строке), синтаксический анализатор XML должен определить, отвечает ли данный объект тому или иному требованию относительно соответствия. Эта тема рассматривается в следующих двух подразделах.

Формальная правильность

Любой конкретный текстовый объект X является **формально правильным** (well-formed) тогда и только тогда, когда он соответствует приведенным ниже требованиям.

- Объект X не нарушает общих грамматических правил, которые определены в [27.25], и удовлетворяет всему набору из двенадцати правил, которые также определены в [27.25] (подробные сведения об этом выходят за рамки настоящей главы).
- Каждый текстовый объект Y, на который явно или неявно указывает ссылка, при веденная в X, также должен быть формально правильным.

  Примечание. Здесь выражение "явно или неявно" обозначает следующее: либо объект X содержит ссылку непосредственно на Y, либо объект X содержит ссылку на некоторый другой текстовый объект Z, который явно или неявно содержит ссылку на Y.

Из этих правил, кроме всего прочего, следует, что объект X должен иметь один и только один корневой элемент, который может содержать и обычно содержит другие элементы; начальный дескриптор каждого элемента должен иметь соответствующий конечный дескриптор точно с таким же именем (причем при проверке соответствия этих имен учитывается регистр); элементы должны быть вложены правильно и т.д. В этом смысле все приведенные выше примеры документов XML были формально правильными (фактически, если текстовый объект не является формально правильным, то по определению его нельзя считать документом XML). Ниже для сравнения показан текстовый объект, который не является формально правильным по причинам, указанным в комментариях.

```
<!-- Предостережение! Этот текстовый объект не является формально
                                                                    -->
<!-- правильным (поэтому, по определению, вообще не является    -->
<!-- документом XML).                                           -->
<PartsRelation>
<PartTuple>
    <PNUM>P1</pnum> <!-- Конечный дескриптор не соответствует
    начальному -->
                    <!-- дескриптору                             -->
    <PNAME>Nut      <!-- Пропущен конечный дескриптор            -->
  </PartTuple>
  </PartTuple>      <!-- Отсутствует начальный дескриптор        -->
</PartsRelation>
<PartsRelation>     <!-- Больше одного корневого элемента        -->
```

Допустимость

Любой конкретный текстовый объект X является **допустимым** (valid) тогда и только тогда, когда он является формально правильным, а также соответствует некоторому заданному документу DTD. Ниже приведен пример документа XML, который является формально правильным (по определению), но по причинам, указанным в комментариях, не может, **тем не** менее, считаться допустимым.

```
<!DOCTYPE PartsRelation SYSTEM "file:///c:/parts.dtd">
<!-- Предостережение! Этот документ является формально правильным
(и                                                              -->
<!-- поэтому представляет собой документ XML). Но это - не
допустимый                                                      -->
<!-- документ PartsRelation, поскольку он не соответствует
определению                                                     -->
```

```
<!-- DTD для такого документа, которое приведено в файле
"parts.dtd".  -->
<partsRelation>          <!-- Не определенный элемент          -->
  <PartTuple CITY="London">
    <PNAME>Nut</PNAME>      <!-- Элементы PNUM и PNAME ...      -->
  <PNUM>P1</PNUM>          <!-- ... приведены в неправильном
  порядке  --> <WEIGHT>12 . 0</WEIGHT> </PartTuple>
  <PartTuple>             <!-- Пропущен атрибут CITY            -->
    < PNUM> P2 </PNUM>
    <PNAME>Bolt</PNAME>
    <remarks>Best quality</remarks>      <!-- Не определенный
  элемент -->
  </PartTuple>             <!-- Пропущен элемент WEIGHT -->
</partsRelation>
```

Атрибуты типа ID и IDREF

Вполне очевидно, что определения DTD не обеспечивают поддержки некоторых типов ограничений целостности[16] (не задают допустимые значения атрибутов и т.д.). Но по большей части эти ограничения по своему характеру являются довольно слабыми (особенно применительно к элементам, в отличие от атрибутов, непосредственно содержащим фактические данные, для которых, по сути, вообще не могут быть заданы ограничения). Но определения DTD обеспечивают также поддержку некоторых ограничений **уникальности** и **ссылочной целостности** благодаря наличию специальных типов атрибутов ID и IDREF. Например, предположим, что требуется задать определение DTD для документа XML, соответствующего не только отношению с данными о деталях, но и всей базе данных поставщиков и деталей. В таком случае соответствующее определение DTD может включать объявления, приведенные ниже.

```
<!ATTLIST SupplierTuple SNUM ID #REQUIRED>
<!ATTLIST PartTuple PNUM ID #REQUIRED>
<!ATTLIST ShipmentTuple SNUM IDREF #REQUIRED>
<!ATTLIST ShipmentTuple PNUM IDREF #REQUIRED>
```

Здесь заслуживает особого внимания то, что теперь элементы PartTuple имеют атрибут PNUM, а не элемент PNUM. Если документ D является допустимым согласно этому определению DTD, то он отвечает приведенным ниже требованиям.

- Каждый элемент SupplierTuple в документе D должен иметь уникальное значение SNUM, а каждый элемент PartTuple в D должен иметь уникальное значение PNUM.
- Каждый элемент ShipmentTuple в документе D должен иметь значение SNUM, которое присутствует в качестве значения некоторого атрибута типа ID в каком-либо месте документа D, и значение PNUM, которое присутствует в качестве значения некоторого атрибута типа ID в каком-либо месте документа D.

Иными словами, атрибуты типа ID действуют в определенной степени аналогично первичным ключам, а атрибуты типа IDREF — в определенной степени аналогично внешним ключам. Но такие аналогии не являются достаточно полными по перечисленным ниже причинам.

---

[16] В целом проблема применения ограничений целостности в контексте XML рассматривается в [27.8].

- Не предусмотрен способ задания такого значения "ключа", которое представляло бы собой нечто иное, чем простая символьная строка.
- Не предусмотрен способ задания значения "ключа", который охватывает больше одного атрибута. (В данном примере следует отметить, что не было определено требование, согласно которому элементы ShipmentTuple должны иметь уникальное значение SNUM/PNUM.)
- В этом примере значения SNUM В элементах SupplierTuple являются уникаль ными не просто по отношению ко всем другим элементам; они уникальны по от ношению ко всем атрибутам типа ID всего документа. Аналогичное замечание от носится и к значениям PNUM В элементах PartTuple. (Поэтому, в частности, ни одно значение SNUM не равно какому-либо из значений PNUM.)
- Кроме того, не гарантируется, что значения SNUM в элементах ShipmentTuple будут равны значению SNUM в некотором элементе SupplierTuple. Примени тельно к ним просто гарантируется, что они будут равны значению некоторого атрибута типа ID, находящегося в каком-либо месте документа.
- Еще более важно то, что проверка ограничения ссылочной целостности (в том виде, в каком она здесь предусмотрена) осуществляется только в пределах одного документа. Перекрестная проверка документов отсутствует.

Недостатки определений DTD

Как было описано выше, предусмотренная в определении DTD поддержка ограничений целостности является довольно слабой. В действительности, сразу после того, как
были впервые введены определения DTD, стали очевидными также многие другие их недостатки. Основные примеры таких недостатков перечислены ниже.

- В этих определениях не используется синтаксис XML (т.е. сами эти определения не являются документами XML), а это означает, что невозможно обеспечить их обработку с помощью обычного синтаксического анализатора XML. Например, рассмотрим следующее объявление элемента.

<!ELEMENT PNUM (#PCDATA)>

Это объявление немного напоминает начальный дескриптор XML, но не является таковым, поскольку "! ELEMENT" нельзя назвать допустимым именем типа элемента XML, а "PNUM" и " (#PCDATA) " — допустимыми атрибутами XML. И на самом деле, если язык XML действительно является таким гибким и мощным, как часто приходится слышать, то он должен предоставлять возможность описать[17] самого себя!

---

[17] Это заявление, безусловно, является чрезвычайно неформальным. Точнее, должна была быть пре- дусмотрена возможность определить такой язык XD, производный от XML, чтобы документы, допустимые в соответствии с XD, были "определениями типа документа" (безусловно, не определениями DTD как таковыми, поскольку уже было показано, что сами определения DTD являются не документами XML, а "определениями типа документа"), которые предоставляют функциональные возможности, аналогичные DTD, и в идеальном случае также многие дополнительные возможности. В действительности, именно таким языком XD является XML Schema (см. следующий подраздел).

- Эти определения, по существу, не обеспечивают поддержки типов данных (пос кольку в них любые данные представляют собой просто символьную строку).
- Они обычно требуют, чтобы элементы разных типов появлялись в некоторой за данной последовательности, даже если эта последовательность не имеет какого-то присущего ей смысла. Например, предположим, что документы PartsRelation имеют определение DTD, которое включает следующую спецификацию:

  ```
  <!ELEMENT PartTuple (PNUM, PNAME, COLOR, WEIGHT, CITY)>
  ```

  В таком случае элементы PNUM, PNAME, WEIGHT, COLOR И CITY В любом конкрет ном элементе PartTuple должны присутствовать точно в указанном порядке, да же несмотря на то, что этот порядок не имеет значения в реляционных терминах.

  Примечание. В принципе, можно сформулировать такое определение DTD, кото рое позволит задавать эти пять элементов в произвольном порядке, но только пу тем записи всех 120 разных возможных упорядочений (именно так!) в виде явных альтернатив с указанием того, что все эти 120 упорядочений равным образом при емлемы,

Этот список недостатков является далеко не исчерпывающим.

Но несмотря на перечисленные выше недостатки, нельзя отрицать, что в основе опре делений DTD лежит полностью сложившийся и важный стандарт, а сами они широко используются на практике. Более того, применяемые в реальных приложениях определе ния DTD обычно являются гораздо более сложными и разносторонними, чем можно предположить на основании приведенных здесь простых примеров. В частности, сам язык XML Schema, который рассматривается в следующем разделе, определен с помо щью документа DTD, имеющего объем больше 400 строк (см. http: //www.w3 . org/ 2001/XMLSchema.dtd).

Язык XML Schema

Сам язык XML Schema [27.28] является производным от XML (но он не сформулиро ван как часть спецификации XML как таковой, в отличие от языка определения DTD). Поэтому схема XML, соответствующая любому конкретному документу D на языке XML, сама является документом XML, скажем, SD. Оформление явных, формально заданных связей между документами D и SD не предусмотрено, но в D может применяться специ альный атрибут schemaLocation, который рассматривается как "подсказка", касаю щаяся местонахождения SD.

Как правило, в схеме XML предоставляется более обширный набор ограничений, распространяющихся на документ (документы) XML, который в ней описан, чем можно было бы представить с помощью определений DTD. В качестве примера ниже приведен аналог на языке XML Schema определения DTD для документа PartsRelation, показанного в подразделе "Определения типа документа" выше в этом разделе (того определения, в котором данные COLOR и CITY представлены с помощью атрибутов, а не элементов XML).

```
<?xml version="1.0"?>
<!-- Схема XML Schema для документов PartsRelation -->
<!DOCTYPE xsd:schema SYSTEM
"http://www.w3.org/2001/XMLSchema.dtd">
```

```
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"> <xsd:

    element name="NOTE" type="xsd: string"/>

   <xsd:element name="PartsRelation">
      <xsd:complexType>
<xsd: sequence>
          <xsd:element ref="NOTE" minOccurs="0"/>
 <xsd: element name="PartTuple" type="PartTupleType"
                    minOccurs="0" maxOccurs="unbounded"/>
 </xsd: sequence>
      </xsd:complexType>
 </xsd:element>

    <xsd:complexType name="PartTupleType">
 <xsd: sequence>
       <xsd:element name="PNUM" type="PartNum"/> <xsd:element
       name="PNAME" type="xsd:string"/> <xsd:element
       name="WEIGHT"> <xsd:simpleType>
          <xsd:restriction base="xsd:decimal"> <xsd: totalDigits
         value="5"/> <xsd:fractionDigits value="l"
       fixed="true"/> <xsd:minlnclusive value="0.1"/>
       </xsd:restriction> </xsd: simpleType> </xsd:element>
       <xsd:element ref="NOTE" minOccurs="0"/> </xsd:
     sequence>
     <xsd:attribute name="CITY" type="City"/>
     <xsd:attribute name="COLOR" type="Color" default="Red"/>
   </xsd:complexType>

   <xsd: simpleType name="PartNum">
     <xsd: restriction base= "xsd: string">
  <xsd:pattern value=" P [0-9] {1,3} "/>
     </xsd:restriction> </xsd:
   simpleType>

   <xsd:simpleType name="Color">
     <xsd:restriction base="xsd:string">
       <xsd:enumeration value="Red"/>
<xsd: enumeration value= "Green"/>
     <xsd:enumeration value="Blue"/>
 </xsd: restriction> </xsd:simpleType>

   <xsd:simpleType name="City">       <xsd:
restriction base="xsd: string"> <xsd:enumeration
value="London"/> <xsd:enumeration value="Oslo"/>
<xsd: enumeration value=" Paris "/>
     </xsd:restriction>
   </xsd:simpleType>

  </xsd:schema>
```

Вполне очевидно, что приведенная выше схема имеет гораздо больший объем и является более сложной, чем аналогичное ей определение DTD, даже несмотря на то, что по большей части в ней заданы такие же ограничения, как и в DTD. Основное различие между схемой и определением состоит в том, что схема налагает некоторые дополнительные ограничения типов на элементы и атрибуты. В языке XML Schema предусмотрено множество встроенных примитивных типов — boolean (логический), decimal (десятичный), string (строковый) и несколько других, а также некоторые встроенные производные типы — integer (целое число), positivelnteger (положительное целое число), negativelnteger (отрицательное целое число и т.д.), которые определены в терминах примитивных типов. Этот язык позволяет также пользователям определять их собственные типы в терминах встроенных типов. Типы могут быть простыми или сложными; основное различие между ними состоит в том, что элементы сложного типа могут содержать вложенные в них другие элементы, а элементы, которые относятся к простому типу, такими свойствами не обладают. Ниже приведены пояснения к этому описанию на основе схемы PartsRelation, которая налагает ограничения на документы PartsRelat ion.

1. Корневой элемент (PartsRelation) определен как относящийся к анонимному сложному типу, значения которого определены в составе того же объявления как состоящие из необязательного элемента NOTE, за которым находится последова тельность элементов PartTuple в количестве от нуля и больше, а каждый из этих элементов относится к типу PartTupleType.
2. **Элементы,** принадлежащие к типу PartTupleType, определены как состоящие из элементов PNUM, PNAME И WEIGHT В указанном порядке, наряду с атрибутами CITY и COLOR, последний из которых является необязательным (если он опущен, **то** по умолчанию применяется значение Red). В составе этих элементов и атрибу тов PNAME определен как относящийся к типу string, тогда как PNUM, CITY и COLOR, соответственно, определены как относящиеся к типам PartNum, city и Color, (см. пп. 4 и 5), а WEIGHT определен как относящийся к анонимному типу, объявление которого включено в данное объявление (см. п. 3).
3. Тип элементов WEIGHT определен как *сокращение* типа decimal, которое имеет точность 5, коэффициент масштабирования 1 и минимальное значение 0.1, иными словами, допустимыми значениями элементов типа WEIGHT ЯВЛЯЮТСЯ именно такие значения, которые входят в состав перечня чисел 0.1, 0.2, ..., 9999.9.
4. Тип PartNum (сокращение типа string) определен с помощью регулярного вы ражения P[0-9] {1,3}, которое интерпретируется как указание на то, что допус тимые значения типа PartNum состоят из прописной буквы P, за которой следуют одна, две или три десятичные цифры.

   Примечание. Такая конструкция, как *регулярное выражение,* заимствована из языка программирования Perl.
5. Типы Color и city (которые также являются сокращениями типа string) опре делены с **помощью** перечислений.

Примечание. Несмотря на сказанное выше, важно учитывать, что "типы" языка XML Schema в действительности не являются типами в том смысле, который указан в главе 5. В частности, для них почти не определены соответствующие операции, что было бы обязательным для настоящих типов. "Определения типов" языка XML Schema фактически гораздо больше напоминают спецификации PICTURE, предусмотренные в таких языках, как COBOL и PL/I; это означает, что в действительности все, что они определяют, сводится к некоторым представлениям рассматриваемых "типов" в виде символьных строк. Отчасти по этим причинам автор счел себя вправе отказаться от использования обычных определяемых пользователем типов (например таких, как показано в главе 3) в предыдущем примере. Ниже перечислены некоторые преимущества использования языка XML Schema как средства описания документов XML по сравнению с языком определения DTD.

- Схемы XML сами являются документами XML.
- Язык XML Schema поддерживает гораздо более развитый механизм определения типов.
- Язык XML Schema предоставляет более простые средства (не показанные в дан ном примере) для указания на то, что элементы различных типов, непосредствен но включенные в один и тот же элемент, могут располагаться в любом порядке.
- Язык XML Schema поддерживает элементы ключа key и ссылки на ключ keyref (также не показанные в данном примере), которые в большей степени напоминают такие конструкции, как реляционный ключ и внешний ключ. Указанные ключи могут состоять из комбинаций значений, взятых на различных уровнях вложенно сти данного конкретного элемента, поэтому обеспечивают, кроме всего прочего, поддержку такого ограничения, которое обычно именуется "уникальностью в пре делах родительского элемента" (см. главу 13 работы [1.5]).

Безусловно, одним из отрицательных следствий из описанного выше является то, что схемы XML становятся гораздо более сложными по сравнению с простыми определениями DTD. В завершение данного описания языка XML Schema, необходимо привести несколько дополнительных замечаний.

- Контроль некоторого конкретного документа XML с помощью схемы XML называется проверкой по схеме (schema validation), в отличие от простого неуточненного выражения "проверка" (validation), которое означает контроль документа с помощью определения DTD.
- Поскольку схема сама является документом XML, она обычно содержит многочисленные элементы XML. Но заслуживает внимание то, что многие из этих элементов являются пустыми; как правило, пустые элементы фактически используются везде, где нужно указать элемент, имеющий атрибуты, но не имеющий информационного наполнения, как показано ниже. `<xsd:element name="NOTE" type="xsd:string"/>`

Такие же элементы можно найти в примере PartsRelation. Схемы также имеют свои собственные определения DTD, которые заданы с помощью следующей спе-

```
<!DOCTYPE xsd:schema [ . . . ] >
```

И эту спецификацию можно найти в примере PartsRelation.

В заключение этого подраздела следует подчеркнуть тот факт, что здесь лишь бегло упомянуты возможности (и сложности) применения языка XML Schema. Дополнительная информация приведена в [27.14], а исчерпывающие сведения можно получить в спецификации языка XML Schema [27.28].

Дополнительные сведения о языках, производных от XML

Как было описано выше в этой главе, XML — это метаязык; это означает, что язык XML позволяет пользователям определять собственные специализированные языки и, в частности, предусматривать в них собственные определяемые пользователем дескрипторы. В связи с этим следует отметить, что каждое конкретное определение DTD и каждая схема XML представляют собой именно *определение специализированного языка,* т.е. такого "языка, производного отXML", каким он был назван ранее. Иными словами, любое конкретное определение DTD и любая схема XML представляют собой именно определение синтаксических правил, которым должен подчиняться соответствующий им документ XM L.

Из сказанного выше следует, что XML — не просто метаязык, а скорее язык, который заслуживает названия "метаметаязыка". XML как таковой определяет (кроме всего про чего) правила формирования определений DTD, а любое определение DTD, в свою оче редь, является метаязыком, который определяет правила формирования соответствую щих ему документов. Следует также отметить, что все указанные правила являются прежде всего синтаксическими правилами; ни язык XML в общем, ни любое конкретное опреде ление DTD в частности, не регламентируют какого-либо смыслового значения докумен тов, создаваемых в соответствии с этими правилами.

27.5. МАНИПУЛИРОВАНИЕ ДАННЫМИ XML

Теперь перейдем к вопросу о языках манипулирования данными XML. Было предложено много таких языков, но стандартным, по-видимому, должен стать XQuery [27.29]. Как будет вскоре показано, язык XQuery (работа над которым ко времени написания данной книги еще не была закончена) основан на нескольких более ранних языках, включая, в частности, XPath [27.27]; в действительности, язык XQuery полностью включает в себя XPath.

Язык XQuery обеспечивает только чтение. Обновление в случае необходимости должно выполняться либо с применением модели DOM [27.24], либо с помощью некоторых специализированных средств (предоставляемых отдельными поставщиками), но, безусловно, оба этих подхода характеризуются определенными недостатками, как указано ниже.

- Недостатком модели DOM является то, что она (как указано в разделе 27.3) пред назначена для программистов, а не для конечных пользователей.
- Недостатком специализированных средств является именно то, что они специали зированные, и каждый поставщик реализует их не так, как другие. (Тем не менее, в разделе 27.7 описано, какие функциональные возможности они обычно предос тавляют.)

В настоящее время ведется разработка языка, не зависящего от конкретных поставщиков, который называется XUpdate [27.30], но ко времени написания данной книги эти работы еще находились на предварительном этапе, поэтому рассматривать здесь этот

язык было бы еще слишком рано. В связи с этим ограничимся описанием в данном разделе только языка XQuery (и XPath).

Язык XQuery произошел от ранее созданного языка Quilt [27.9], на который в свою очередь оказали влияние SQL, OQL и некоторые более старые языки из семейства языков XML, в том числе XQL, XML-QL и Lorel (описание языка OQL приведено в [25.11], а информация, касающаяся XQL, XML-QL и Lorel, приведена в [27.5] и [25.18]). Отметим, что в целом язык XQuery является весьма развитым и сложным, поэтому в книге такого характера, как данная, нет смысла пытаться привести его исчерпывающее описание. В связи с этим здесь просто показан ряд примеров с комментариями, которых, по мнению автора, должно быть достаточно, чтобы получить общее представление о возможностях, области применения и характере этого языка. Но прежде чем приступить к изучению этих примеров, необходимо отметить, что язык XQuery в действительности вообще не оперирует с документами XML как таковыми! Причины этого перечислены ниже.

- По сути, документы XML, согласно определению, представляют собой символьные строки, которые (кроме всего прочего) предназначены для чтения людьми.
- В силу этого они включают множество таких особенностей, как дескрипторы, обо значения конца строки, отступы и т.д., которые позволяют достичь такой цели, как повышение удобства чтения, но не имеют никакого отношения к реальному информационному наполнению рассматриваемого документа.
- Сам язык XQuery в действительности предназначен для обеспечения доступа именно к информационному наполнению.

Поэтому язык XQuery определен как язык, предназначенный для функционирования не в терминах документов XML как таковых, а скорее в терминах документов XML, которые преобразованы в некоторую абстрактную форму (полученную в результате обработки документа синтаксическим анализатором). Абстрактная форма любого конкретного документа XML называется "экземпляром модели данных XQuery" [27.29]; ее можно рассматривать как инфонабор[18] (т.е. как структурную иерархию), аналогичный приведенному на рис. 27.2 (в подразделе "Структура документа XML" раздела 27.3). Поэтому заслуживает особого внимания то, что результат запроса, выраженного на языке XQuery, представляет собой инфонабор, а не документ XML. В [27.29] об этом сказано так: "В настоящее время вопрос о том, как обеспечить [обратное] преобразование экземпляра модели данных в документ XML, остается открытым". В действительности, результат вычисления выражения XQuery может даже оказаться не строго определенным инфонабором, поскольку (как будет показано ниже) он может даже не быть формально правильным.

Язык XPath

В языке XQuery широко используются характерные выражения — обозначения пути (path expression) языка XPath, поэтому начнем с краткого описания таких выражений. Концептуально эти выражения сохраняют строгое подобие обозначениям пути, которые описаны в главах 25 и 26. А именно, обозначением пути в языке Xpath является такое выражение, которое позволяет, начиная от некоторого указанного исходного узла (или узлов)

[18] Точнее, эта абстрактная форма состоит из дополненного инфонабора, называемого "инфонабором, полученным после проверки по схеме" (Post Schema Validation Infoset — PSVI), который преобразован в форму модели данных XQuery.

в некотором заданном инфонаборе пройти по заданному пути (или путям) в этом инфонаборе, чтобы найти некоторый желаемый целевой узел (или узлы).

Примечание. Термины *исходный узел* и *целевой узел* не являются официально принятыми терминами языка XPath.

Поэтому синтаксическая конструкция обозначения пути XPath представляет собой последовательность шагов, в которой каждый шаг, за исключением первого, отделен от предыдущего символом косой черты ("/"), а перед первым шагом могут быть дополнительно введены один или два символа косой черты, как показано ниже.

```
[/\/ /] step/step/.../step
```

Начальная одинарная косая черта означает, что перемещение начинается от корневого узла (т.е., исходным узлом является корневой узел); начальная двойная косая черта означает, что поиск должен начинаться с каждого узла по очереди (по сути, применение двойной косой черты вызывает просмотр всего инфонабора в режиме поиска в глубину, слева направо, в котором каждый узел по очереди выполняет роль исходного узла). Если же в обозначении пути вообще нет начальной косой черты, то в качестве исходного узла применяется текущий узел (т.е. узел, доступ к которому был выполнен последним по времени).

Ниже приведено несколько простых примеров, основанных на документе PartsRelation из подраздела "Определения типа документа" раздела 27.4 (напомним, что этот документ содержит элементы PartTuple с данными о деталях P1, P2 и P3):

- Приведенное ниже выражение возвращает последовательность узлов, соответствующих этим трем элементам PartTuple.

  ```
  /PartsRelation/PartTuple
  ```

- Приведенное ниже выражение возвращает пустую последовательность узлов, поскольку корневой узел не имеет дочерних узловParTup1e. /PartTuple

  Примечание. В данном случае корневым узлом является не узел PartsRelation, а скорее узел всего документа (см. описание рис. 27.2 в разделе 27.3). Такое же замечание, безусловно, относится как к предыдущему примеру, так и к следующему.

- Приведенное ниже выражение возвращает тот же результат, что и выражение из первого примера.

  ```
  //PartTuple
  ```

Вообще говоря, каждый шаг в заданном обозначении пути выполняется в контексте результатов предыдущего шага (точнее, если результат предыдущего шага представляет собой последовательность узлов SN, то каждый узел из SN в свою очередь становится контекстным узлом для текущего шага). Каждый шаг состоит из трех перечисленных ниже частей.

1. Ось, определяющая направление, в котором должно продолжаться перемещение. Чаще всего перемещение продолжается в направлении вверх, по оси родительского узла (parent) или узла-предка (ancestor), вниз, по оси дочернего узла (child) или узла-потомка (descendant), влево, по оси предшествующего узла (preceding) или предшествующего сестринского узла (preceding-sibling), и

вправо, по оси следующего узла (following) или следующего сестринского узла (following-sibling).

2. Выражение для проверки узла, которое определяет тип (типы) узла (узлов), представляющего интерес для поиска.
3. В качестве необязательного компонента применяются один или несколько предикатов, которые служат для исключения ненужных узлов.

Примечание. Конкретные "родительские" (parent), "дочерние" (child) и другие оси, перечисленные в первом пункте, не исчерпывают всех возможностей — они указаны просто в качестве иллюстрации. Если не определена ни одна ось, то по умолчанию применяется ось child (т.е. в процессе перемещения происходит переход к дочерним узлам контекстного узла).

Для того чтобы изучить, как происходит обработка обозначений пути, рассмотрим следующий немного более сложный пример.

```
/PartsRelation/PartTuple[WEIGHT="17.0"]
```

Пояснение

1. Первоначальный символ "/" указывает корневой узел (т.е. узел документа) в качестве контекстного узла для шага, непосредственно следующего за этим символом.
2. Обозначения пути могут записываться (и обычно записываются) в сокращенной форме. Поэтому в данном примере выражение "PartsRelation" является сокращением от выражения "child: : PartsRelation" (где "child" —ось, а "PartsRelation" — выражение для проверки узла). Таким образом, результатом этого шага является узел PartsRelation — единственный дочерний узел корневого узла.
3. Аналогичным образом, "PartTuple" — это сокращение от "child: : PartTuple"; выполнение этого шага приводит к получению последовательности из трех узлов PartTuple, которые являются дочерними по отношению к узлу PartsRelation.
4. Наконец, предикат [WEIGHT="17 .0"] исключает все узлы PartTuple, кроме узла, в котором дочерний элемент WEIGHT имеет значение 17.0.

   Примечание. Выражение "WEIGHT" само является сокращением; этот предикат в полной форме имеет следующий вид.

   ```
   [child::WEIGHT="17.0"]
   ```

   Поэтому окончательным результатом становится последовательность из двух узлов PartTuple, соответствующих деталям Р2 и Р3 (в указанном порядке).

В завершение этого краткого описания языка XPath, необходимо отметить, какую важную роль во всех представленных выше процедурах играет понятие "текущего положения". Было показано, что каждый шаг в любом конкретном обозначении пути выполняется по отношению к некоторому контекстному узлу, который служит в качестве "текущего узла". Отметим, что весьма похожее понятие преобладало в языках доступа в тех ранних системах "с управляемой вручную навигацией" (особенно иерархических системах), которые доминировали на рынке СУБД до появления на сцене систем SQL. Именно в этом понятии заключалась непосредственная причина многих сложностей

(не говоря уже об ошибках кодирования), от которых страдали подобные Системы; безусловно, одним из многих значительных достижений, связанных с внедрением реляционной модели, явилось то, что она позволила полностью исключить понятие "текущего положения". Поэтому, безусловно, следует поставить под сомнение всю "мудрость" повторного введения указанного понятия (и, более того, применения его в качестве ключевого средства).

Язык XQuery

Одним из недостатков языка XPath является то, что он в своей основе представляет собой просто механизм адресации; применяемые в нем обозначения пути позволяют переходить по существующим узлам в иерархии, но не дают возможности формировать узлы, которые еще не существуют. Иными словами, язык XPath немного напоминает "реляционный" язык (здесь слово "реляционный" заключено в кавычки, поскольку настоящие реляционные языки, безусловно, не являются навигационными) в том смысле, что он поддерживает операции сокращения и проекции, но не операции соединения[19]. Именно эта причина отчасти послужила стимулом к созданию языка XQuery; по сравнению с языком Xpath одним из основных дополнений, предусмотренных в языке XQuery, как раз и является способность формировать новые узлы.

Такая возможность иллюстрируется в приведенном ниже первом примере. И в данном случае предполагается, что существует документ XML, называемый PartsRelation, точно такой же, как и в предыдущем подразделе. Предположим также, что имеются структурированные аналогичным образом документы SuppliersRelation и ShipmentsRelation. Таким образом, ниже приведена формулировка на языке XQuery следующего запроса: "Для каждой поставки определить имя поставщика, имя детали и объем поставки".

```
<Result>
{ for $spx in document("ShipmentsRelation.xml")
             //ShipmentTuple,
      $sx in document("SuppliersRelation.xml")
             //SupplierTuple[SNUM = $spx/SNUM] ,
      $px  in document("PartsRelation.xml")
             //PartTuple[PNUM = $spx/PNUM]
  order by SNAME,
  PNAME return
    <ResultTuple>
      { $SX/SNAME, $px/PNAME, $spx/QTY }
    </ResultTuple> }
</Result>
```

Пояснение

1. В целом это выражение вычисляет *(формирует)* единственный элемент Result, содержащий последовательность элементов ResultTuple. В качестве дополнительного замечания отметим, что если бы были удалены охватывающие это выражение

---

[19] Это утверждение в определенной степени является слишком упрощенным; фактически язык XPath поддерживает своего рода операцию декартова произведения (которая, безусловно, представляет собой вырожденный случай операции соединения). Но этот язык не поддерживает операции соединения в какой-либо более общей форме.

дескрипторы Result, то оставшееся выражение по-прежнему представляло бы собой допустимый запрос XQuery, но возвращало результат, не являющийся формально правильным.

2. Удобный способ пояснения семантики всего этого выражения (за исключением охватывающих дескрипторов Result) состоит в использовании следующего его аналога — выражения реляционного исчисления.

```
{ SX.SNAME, PX.PNAME, SPX.QTY } WHERE SX.SNUM = SPX.SNUM
                                  AND PX.PNUM = SPX.PNUM
```

   Это выражение фактически требует выполнения указанного соединения отношений поставщиков, деталей и поставок, а затем получения проекции результата этого соединения по атрибутам SNAME, PNAME И QTY.

Примечание. Здесь не показан аналог шага "order by" в запросе XQuery, поскольку операция упорядочения "order by" не является реляционной. Кроме того, применяются обычно принятые в этой книге соглашения, касающиеся имен переменных области значений (SX, PX и SPX — переменные области значений, пробегающие, соответственно, по отношениям поставщиков, деталей и поставок). Наконец, игнорируется тот факт, что в реляционных выражениях дубликаты кортежей устраняются автоматически, а в выражениях XQuery — нет.

3. На основании изложенного в предыдущем абзаце можно сделать вывод, что переменные $sx, $px и $spx языка XQuery действуют в определенной степени аналогично переменным области значений в реляционном исчислении. Но такая аналогия является не совсем точной и способна сбить с толку; формулировка запроса XQuery в действительности не так уж похожа на реляционную, поскольку вполне очевидно, что она по своему характеру является полностью процедурной (в этой связи рекомендуем ознакомиться с аннотацией к [27.3]). На самом деле формулировка запроса XQuery весьма напоминает приведенную ниже формулировку запроса в виде вложенных циклов (здесь она представлена на псевдокоде, а в качестве разделителя используется ".", а не "/").

```
do for each shipment $spx ;
 do for each supplier $sx where Ssx.snum = $spx.snum
;       do for each part $px where $px.pnum =
$spx.pnum ;          emit { $sx. SNAME, $px. PNAME,
$spx.QTY } ;
   end do ;          end do ;
```

   Из этого следует, что фактически переменные $sx, $px и $spx гораздо больше напоминают переменные управления циклом (определяемые в том же смысле, как и в обычном языке программирования), чем переменные области значений. Более того, следует отметить, что итерацию по кортежам отношения поставок пришлось выполнять именно во внешнем цикле; это означает, что вначале пришлось ввести в действие переменную управления циклом $spx, поскольку две остальные переменные определены в терминах этой переменной. Такой факт может повлечь за собой некоторые интересные следствия с точки зрения организации работы оптимизатора. В отличие от нее, две другие переменные, $sx и $px, могут быть введены в любом порядке. Тем не менее, вопрос о том, оказывает ли порядок введения

этих переменных какое-то влияние на работу оптимизатора, также может оказаться весьма интересным. По крайней мере, следует отметить, что порядок, в котором вводятся эти переменные, оказывает влияние на то, в каком порядке вырабатываются результирующие элементы (см. описание конструкции return в п. 7); таким образом, вообще говоря, аналоги выражений A JOIN Ви B JOIN A на языке XQuery не являются эквивалентными.

На основании приведенных выше соображений можно смело утверждать, что XQuery фактически в большей степени напоминает язык программирования, чем язык запросов, предназначенный для конечного пользователя.

4. Теперь рассмотрим, в частности, конструкцию for, а именно ту часть конструкции, которая предшествует первой запятой. Приведенное ниже выражение возвращает абстрактную ("синтаксически проанализированную") версию документа XML, который содержится в файле ShipmentsRelation.xml, с использованием в качестве контекстного всего узла документа.

```
document("ShipmentsRelation.xml")
```

Спецификация "//ShipmentTuple" свидетельствует о том, что нас интересует элементы ShipmentTuple из данного документа, а спецификация "for $spx in" указывает, что переменная $spx должна принять в качестве своего значения каждый из этих кортежей поставок по очереди, в том порядке, в котором они присутствуют в данном документе.

5. Следующая часть конструкции for является аналогичной, за исключением того, что переменная $sx принимает свои значения только среди тех поставщиков, для которых значение SNUM равно текущему значению $ spx.

```
$sx in document ("SuppliersRelation.xml")
      //SupplierTuple[SNUM = $spx/SNUM]
```

6. Последняя часть конструкции for аналогична описанной выше части. "
7. Конструкция return выполняется для каждой комбинации значений переменных $sx, $px и $spx, в том порядке, в котором эти значения вырабатываются конструкцией for. Поэтому в данном примере конструкция return вырабатывает элементы ResultTuple в последовательности, которая определяется правилом, что значения $px изменяются чаще всего, значения $sx изменяются менее часто, а значения $spx изменяются наименее часто.
8. Конструкция order by в основном не требует пояснений. Но заслуживает внимания то, что она появляется перед соответствующей конструкцией return. Такое расположение указанных конструкций позволяет упорядочивать результаты на основании значений, которые еще фактически не появляются в результатах (как, например, в запросе SELECT CITY FROM P ORDER BY WEIGHT на языке SQL). Но концептуально и в этом случае конструкция return должна быть выполнена в первую очередь, поскольку упорядочение невозможно выполнить до тех пор, пока не появится нечто, предназначенное для упорядочения.

Ниже приведен второй пример: "Определить номера деталей и общий объем поставки для деталей, поставляемых двумя или несколькими поставщиками".

```
for $pnum in
    distinct-
values(document("ShipmentsRelation.xml")//PNUM) let $spx
:= document("ShipmentsRelation.xml")
          //ShipmentTuple[PNUM =
$prium] where count ( $spx ) > 1 order
by PNUM return
  <Result>
    { $pnum,
     <totqty> { sum ( $spx/qty ) } </totqty> }
  </Result>
```

Пояснение

1. В данном примере показано полное *выражение FLWOR* (FLWOR — сокращение от for + let + where + order by + return; оно произносится как слово "flower" — флауэ).
2. Обратите внимание на использование ключевого слова "distinct-values" в конструкции for для устранения дубликатов номеров деталей; переменная $pnum принимает значения только среди разных номеров деталей в поставках.

3. Конструкция let отличается от конструкции for тем, что заданная в ней переменная не используется для итераций по указанной последовательности значений; вместо этого такой переменной полностью присваивается соответствующая последовательность значений. Кроме того, в данном примере конструкция let вычисляется для каждого отдельного номера детали в поставке по очереди, поскольку она фактически вложена в конструкцию for.

4. Затем конструкция where вызывает вычисление оставшейся части этого выражения тогда и только тогда, когда текущая последовательность поставок (т.е. последовательность поставок с текущим номером детали) имеет длину больше единицы. Язык XQuery поддерживает обычные агрегирующие операторы count, sum, avg, max и min.
5. Следует опять отметить, что все это выражение в целом весьма напоминает процедурное. Аналог этого выражения на псевдокоде приведен ниже.

```
do for each distinct shipment part number $pnum ;
    do for all shipments $spx where $spx.pnum = $pnum ;
      if ( count of such shipments $spx ) > 1 then
          emit { $pnum, sum ( $spx.qty ) } ;
           end if ;
    end do ;
end do ;
```

Реляционный аналог данного выражения выглядит следующим образом.

```
{ SPX.PNUM, SUM ( SPY WHERE SPY.PNUM = SPX.PNUM, QTY ) }
          WHERE COUNT ( SPY WHERE SPY.PNUM = SPX.PNUM ) > 1
```

Теперь следует отметить, что приведенные выше примеры, возможно, являются не совсем реальными. Это связано с именно с тем, что они по своему характеру в определенной степени напоминают реляционные; в частности, в этих примерах почти не учитывается специфика документов XML, которые обычно имеют иерархическую структуру.

Поэтому рассмотрим следующий вариант применяемого проекта. Вначале предположим, что документ PartsRelation является точно таким же, как и прежде. Но допустим, что вместо документов SuppliersRelation И ShipmentsRelation теперь применяется документ SuppliersOverShipments, который характеризуется описанными ниже особенностями.

- Корневой элемент содержит последовательность элементов Supplier.
- Каждый элемент Supplier включает элементы SNUM, SNAME, STATUS и CITY, за которыми расположена последовательность элементов shipment.
- Каждый элемент Shipment содержит элемент PNUM и элемент QTY.

Структура этого документа показана на рис. 27.4.

Теперь рассмотрим запросы: "Определить поставщиков, которые поставляют деталь P2" и "Определить детали, поставляемые поставщиком S2". Ниже приведены реляционные формулировки этих запросов.

```
SX WHERE EXISTS SPX ( SPX.SNUM = SX. SNUM AND SPX.PNUM = ' P2 ' )
PX WHERE EXISTS SPX ( SPX.PNUM = PX.PNUM AND SPX.SNUM = 'S2' )
```

Примечание. Для упрощения предполагается, что значения SNUM и PNUM представляют собой простые символьные строки, а не значения какого-то определяемого пользователем типа. Итак, ниже приведена формулировка первого запроса на языкеXO_иегу.

```
for $sx in document("SuppliersOverShipments.xml")//Supplier
where $sx//PNUM = "P2"
return
  <Result>
    { $SX//SNUM, $sx//SNAME, $sx//STATUS, $Sx//CITY }
  </Result>
```

А формулировка второго запроса на языке XQuery выглядит следующим образом.

```
let $sx := document("SuppliersOverShipments.xml")
         //Supplier[SNUM = "S2"]
return <Result>
    { document("PartsRelation.xml")
      //PartTuple[PNUM = $sx//PNUM]
  } </Result>
```

Вполне очевидно, что формулировки XQuery оказались менее симметричными по сравнению с их реляционными аналогами, но этого и следовало ожидать из-за асимметричного (иерархического) характера проекта XML.

В завершении данного раздела приведем еще несколько замечаний.

- Как и в первоначально сформулированной версии языка SQL [4.9]—[4.11], в языке XQuery отсутствует явная поддержка операции соединения. И действительно, в [27.29] конкретно указано, что "для вычисления соединений могут применяться [выражения FLWOR]"; под этим в сущности подразумевается, что при вычислении каждого соединения пользователь должен указывать необходимую для этого последовательность шагов. Но язык XQuery включает явную поддержку операций объединения, пересечения и разности (конструкция except) с устранением дубликатов.

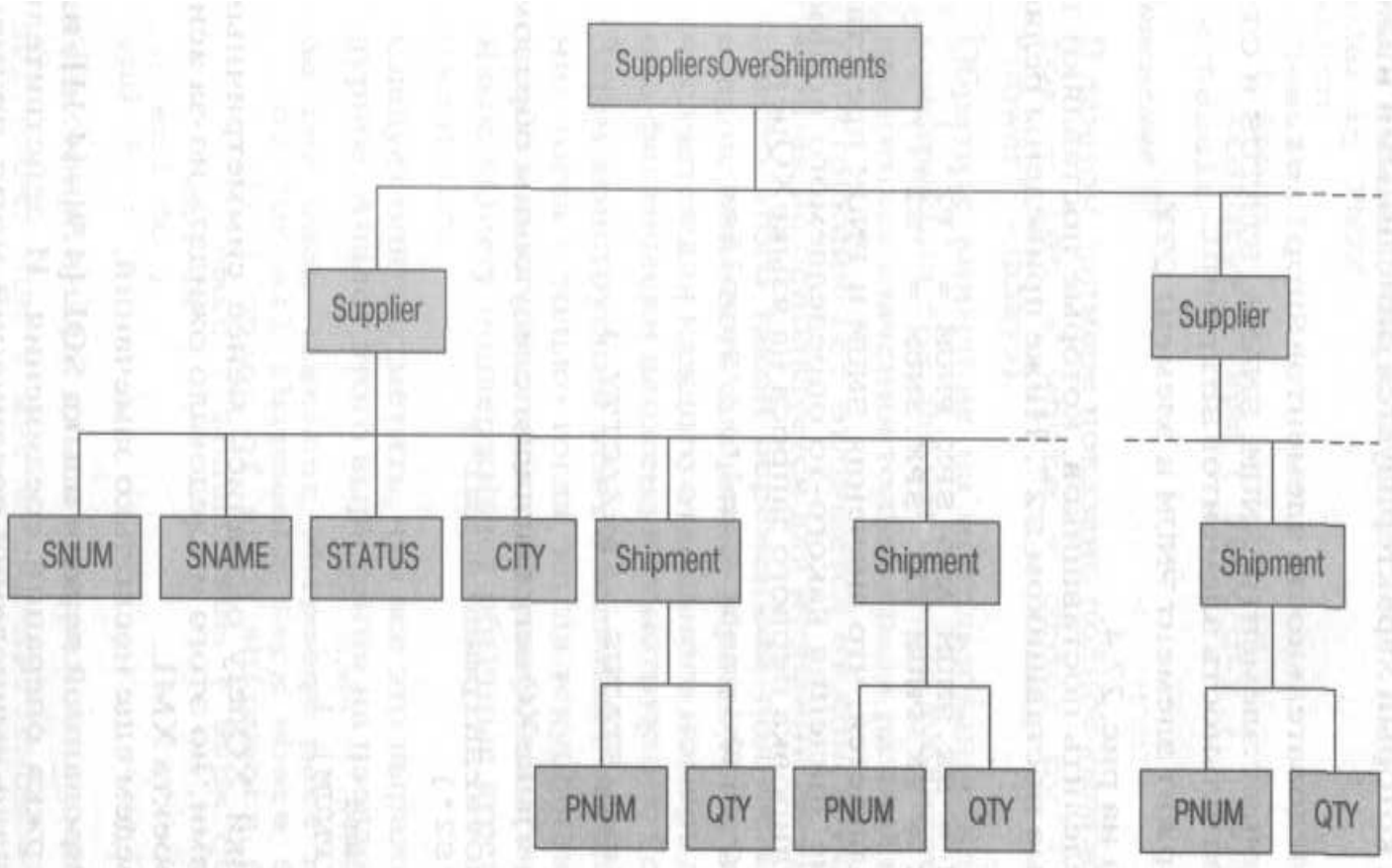


Рис. 27.4. Данные о поставщиках и поставках, представленные в виде иерархии

- Язык XQuery включает также явную поддержку кванторов существования и всеобщности. Ниже приведены соответствующие примеры.

```
some $x in ( 2, 4, 8 ) satisfies $x <
7 every $x in ( 2, 4, 8 ) satisfies $x
< 7
```

Первое из этих выражений принимает истинное значение, а второе— ложное.

- Как описано в главах 7 и 8, с реляционной моделью связано (очень важное) поня тие полноты. По-видимому, аналогичного понятия не существует ни в языке XQuery, ни в языке XML в целом[20].

27.6. ПРИМЕНЕНИЕ ЯЗЫКА XML В БАЗАХ ДАННЫХ

Теперь (наконец-то!) мы можем приступить к изучению вопроса о том, какое значение имеют описанные выше особенности языка XML, в частности, для организации работы баз данных. Безусловно, что есть потребность хранить документы XML (возможно, точнее следовало сказать, данные XML) в базах данных, а также иметь возможность извлекать и обновлять эти данные по требованию. В действительности, существует также прямо противоположная потребность, а именно получать "обычные" данные или данные, отличные от XML (в частности, результаты некоторого запроса) и преобразовывать их в форму XML, так, чтобы их можно было (например) передавать в виде документа XML некоторому потребителю. Начнем с подробного изучения первого из этих требований.

Должно быть очевидно, что могут применяться три основных способа, позволяющих сохранить документ XML в базе данных, которые перечислены ниже.

1. Весь документ может быть сохранен в виде значения некоторого атрибута в некотором кортеже.
2. Документ может быть *разделен* (это— технический термин!), и разные его фраг менты представлены в виде значений различных атрибутов различных кортежей различных отношений.
3. Документ может быть сохранен не в обычной базе данных, а скорее "собственно в базе данных XML" (т.е. в такой базе данных, которая вместо отношений содержит документы XML как таковые).

Рассмотрим каждый из этих вариантов по очереди.

Хранение документов в виде значений атрибута

Такой подход кратко упоминался в подразделе "Разработка языка XML" раздела 27.3, где была указана возможность дополнения переменной отношения деталей P для включения атрибутов DRAWING и DESCRIPTION. По сути, такую идею можно реализовать, как описано ниже.

- Прежде всего, необходимо определить новый тип данных, скажем, XMLDOC, значе ниями которого являются документы XML; после этого можно определить, что кон кретные атрибуты конкретной переменной отношения принадлежат к этому типу.

---

[20] По-видимому, исключением следует считать "flower power" ("всемогущее выражение FLWOR")? (Автор приносит свои извинения, поскольку он не мог удержаться от соблазна привести этот каламбур.)

Примечание. Как было показано выше, язык XML как таковой является довольно многословным; любой документ XML вполне может в пять или десять раз превышать по объему представленные в нем ^отформатированные данные, поэтому обработка таких документов в исходной форме иногда становится чрезвычайно неэффективной. Таким образом, может оказаться целесообразной организация хранения подобных документов в некоторой сжатой форме (возможно, в виде данных, подвергнутых синтаксическому анализу). Но, безусловно, такие соображения не имеют никакого отношения к рассматриваемой модели.

- Кортежи, содержащие значения XMLDOC, можно вставлять и удалять с использованием обычных реляционных операторов INSERT И DELETE, а значения XMLDOC, содержащиеся в таких кортежах, полностью заменять с применением обычного реляционного оператора UPDATE. Кроме того, безусловно, значения XMLDOC могут участвовать обычным образом в операциях только чтения.
- Как и все типы, тип XMLDOC должен иметь множество связанных с ним операторов. Рассматриваемые операторы предположительно должны предоставлять возможность выборки и обновления атрибутов со значениями в виде XMLDOC на более низком уровне детализации, обеспечивая (например) доступ к отдельным элементам или атрибутам XML. В случае выборки такие операторы, по-видимому, должны быть очень похожими на операторы, применяемые в языке XQuery; они могут даже вызываться с помощью "замаскированных" вызовов выражений XQuery, хотя встроенная (непосредственная) поддержка должна быть более дружественной по отношению к пользователю. Но должны также поддерживаться операторы обновления.
- Необходимо также предусмотреть операторы для проверки того, что данное конкретное значение XMLDOC соответствует некоторому заданному определению DTD или некоторой заданной схеме XML (т.е. является допустимым). (Безусловно, значения XMLDOC должны быть формально правильными по определению, но они всё еще могут оказаться недопустимыми.)

Примечание. Этот первый подход, в котором предусмотрено хранение целых документов в виде значений атрибутов, в литературе и текущей практике иногда неформально называют "применением столбца XML". Ниже перечислены некоторые факторы, благодаря которым этот подход может оказаться наиболее приемлемым.

- Документы уже существуют.
- Операции обычно выполняются со всем документом в целом, а не с отдельными его фрагментами.
- Документы редко обновляются.
- При выполнении поиска обычно за основу берется небольшое, известное множество элементов или атрибутов.
- Документы должны храниться в неизменном виде для целей аудита.

По существу, этот подход является приемлемым для приложений, которые иногда называют "приложениями, ориентированными на работу с документами" [27.7]. Таковыми являются приложения, в которых в конечном итоге рассматриваемые документы в основном предназначены для использования людьми и состоят главным образом из текста на естественном языке.

Принцип "разделяй и публикуй"

Во втором подходе какие-либо новые типы данных не используются. Вместо этого документы XML разделяются на фрагменты (например, на отдельные элементы и атрибуты XML), после чего эти фрагменты сохраняются в виде значений различных реляционных атрибутов в разных частях базы данных[21]. Поэтому следует отметить, что в этом случае база данных не содержит документы XML как таковые. Это означает, что СУБД не имеет информации о существовании таких документов, а тот факт, что некоторые значения в базе данных могут быть скомбинированы определенным образом для создания какого-то документа XML, известен разработчику некоторой прикладной программы (возможно, Web-сервера), но не отражен в СУБД.

Но следует отметить, что при этом в прикладной программе предусмотрена возможность создать документ XML из обычных данных базы, поэтому фактически достигается вторая из первоначально заданных целей! А именно, предоставляется возможность получить результат запроса к обычным данным (отличным от XML) и преобразовать их в форму XML. Такое преобразование называется *публикацией* (рассматриваемых данных); таким образом, термин *публикация,* в том смысле, в каком он применяется в данном контексте, является противоположным термину *разделение,* также определенному в этом разделе. Кстати, следует отметить, что правила преобразования, которыми руководствуются в своих действиях программы при выполнении таких операций разделения и публикации, сами обычно хранятся в форме документов XML.

Между прочим, способность публиковать отличные от XML данные в форме XML может также рассматриваться как способность поддерживать представления XML данных, отличные от XML. (Точнее, такая публикация может рассматриваться как поддержка представлений XML, предназначенных для выборки. Если же должно быть разрешено обновление таких представлений, то для этого потребуется поддержка со стороны соответствующей функции разделения.) В конечном итоге, нет каких-либо причин, по которым (используя терминологию главы 2) внешний и концептуальный уровни системы должны были быть обязательно основаны на одной той же модели данных; фактически в главе 3 упоминалась возможность создания системы, в которой концептуальное представление является реляционным, а заданное внешнее представление — иерархическим. Единственное непреложное требование состоит в том, что должно быть предусмотрено обратимое преобразование между этими двумя представлениями.

Тем не менее, при использовании указанного подхода возникают определенные проблемы, вызванные теми причинами, которые можно считать разновидностью несоответствия типов данных (см. главу 25) между реляционной моделью и так называемой "моделью XML", которая, как уже было показано в данной главе, фактически представляет собой старую иерархическую модель, но в несколько ином виде. При этом одна из проблем состоит в том, что дочерние узлы любого конкретного родительского узла в иерархической модели XML образуют не множество, а последовательность (т.е. они упорядочены), тогда как

---

[21] Вырожденным частным случаем этого подхода является хранение всего документа в виде символьной строки в позиции одного атрибута отдельного кортежа. Кроме того, необходимо учитывать, что данный и предыдущий подходы не следует считать взаимоисключающими — может оказаться приемлемым любой из них, в зависимости от того, какие действия в дальнейшем потребуется выполнять с данными. Могут даже использоваться одновременно оба подхода — возможно, даже применительно к одному и тому же документу.

кортежи в отношении не упорядочены. Поэтому, если задан определенный документ D на языке XML, то возможна ситуация, что некоторые свойства документа D (информационные или другие) будут потеряны после разделения и сохранения D в реляционной базе данных. А если это происходит, то нельзя дать гарантии, что при публикации данных в форме XML будет выполнена реконструкция документа D в его точной первоначальной форме. В частности, вполне вероятно, что не будет одинаковой расстановка пробельных символов в оригинальной и (повторно) опубликованной версии D.

Примечание. Подход по принципу "разделяй и публикуй" иногда неформально называют как "применение коллекции XML". Ниже перечислены некоторые факторы, под влиянием которых такой подход может оказаться наиболее приемлемым.

- В реляционной базе данных уже имеются необходимые данные и требуется обеспечить их взаимодействие с соответствующими данными в документах XML.
- В неизменном виде требуется хранить только те части документов, которые содер жат символьные данные (поэтому дескрипторы могут быть поставлены в соответствие именам реляционных атрибутов).
- Операции часто выполняются с отдельными элементами или атрибутами.
- Операция обновления применяются часто и важно обеспечить высокую произво дительность обновления.
- В обрабатывающих программах используются существующие реляционные ин терфейсы.

Вообще говоря, подход по принципу "разделяй и публикуй" является наиболее подходящим для таких приложений, которые иногда называют "приложениями, ориентированными на обработку данных" [27.7]. Таковыми являются приложения, в которых документы содержат (как правило) оперативную информацию, или информацию, предназначенную для поддержки принятия решений, а не текст на естественном языке.

Базы данных XML

Основная причина, по которой в данной главе упоминается этот третий подход, состоит просто в стремлении всесторонне рассмотреть данную тему. В конечном итоге, как показано в главе 3, реляционная модель является необходимой и достаточной для представления вообще любых данных. Невозможно также оспаривать то, что сделаны огромные вложения в исследования и разработку, а также в создание коммерческих продуктов в той области, которую можно в целом назвать "реляционной инфраструктурой" (т.е. сделаны огромные усилия по обеспечению поддержки восстановления, организации параллельной работы, защиты и оптимизации, не говоря уже о поддержке целостности, а также по всем другим направлениям, которые рассматривались в этой книге!). Поэтому, по мнению автора, было бы неразумно полностью переключаться на разработку принципиально новой технологии баз данных, притом что нельзя найти каких-либо достаточно убедительных причин для принятия подобного решения, не говоря уже о том, что любая такая технология будет, безусловно, страдать от недостатков, аналогичных тем, которые уже обнаруживаются в технологии иерархических баз данных (см., например, главу 13 работы [1.5] или аннотации к [27.3] и [27.6]).

27.7. СРЕДСТВА ЯЗЫКА SQL

Ко времени написания данной книги в стандарте SQL не была предусмотрена поддержка языка XML, но предполагается, что соответствующие средства поддержки будут включены в этот стандарт под общим названием SQL/XML [27.15] и определены в части 14 следующей версии стандарта (рабочий вариант которой был опубликован в 2003 году). В настоящем разделе даны предварительные сведения об указанных средствах поддержки, но следует учитывать, что весь представленный здесь материал после формального утверждения спецификации средств SQL/XML может потребовать пересмотра.

Применение "коллекции XML"

Как было показано в предыдущем разделе, существует два основных способа, с помощью которых может быть организовано хранение данных XML в базе данных SQL. В этих двух подходах используются "коллекция XML" и "столбец XML"; в спецификации SQL/XML поддерживаются оба указанных способа. (По очевидным причинам в этой спецификации не поддерживаются собственно базы данных XML как таковые.) В данном подразделе рассматривается так называемая "поддержка с использованием коллекции XML".

Прежде всего, необходимо отметить, что само требование, согласно которому "поддержка коллекции XML" вообще должна быть включена в стандарт SQL, является довольно странным! Дело в том, что (как показано в разделе 27.6) такая поддержка не имеет никакого отношения к СУБД (она, скорее, предназначена для определенных прикладных программ, таких как Web-серверы, которые применяются в качестве надстройки над СУБД). Но, так или иначе, рассматриваемые средства поддержки фактически состоят из описанных ниже компонентов.

- Правила преобразования наборов символов, идентификаторов, типов данных[22] и значений SQL в наборы символов, имена, типы данных и значения XML.
- Правила преобразования таблицы или набора таблиц SQL в два документа XML, один из которых содержит данные как таковые, а другой — соответствующую схему на языке XML Schema.

Эти правила, вместе взятые, обеспечивают публикацию данных SQL в форме XML; равным образом они обеспечивают поддержку тех информационных структур, которые в предыдущем разделе были названы "представлениями данных SQL в виде документов XML" (но предназначенных только для выборки). Поэтому, в частности, эти правила предоставляют основу для применения запросов XQuery к таким данным. Но следует отметить, что в спецификации SQL/XML не определены какие-либо правила выполнения обратного процесса, а именно для разделения данных XML на фрагменты и преобразования в форму SQL (если не считать того небольшого исключения, что в этой спецификации предусмотрены правила преобразования наборов символов и имен XML в наборы символов и идентификаторы SQL).

В качестве примера рассмотрим аналог на языке SQL обычно применяемой в этой книге переменной отношения деталей, т.е. таблицу p. Ниже приведено упрощенное определение[23] этой таблицы на языке SQL.

---

[22] Ко времени написания этой книги структурированные типы не поддерживались.

```
 CREATE TABLE P
( PNUM CHAR(6),
  PNAME CHAR(20),
  COLOR CHAR(6),
  WEIGHT
  NUMERIC(5,1), CITY
  CHAR(20) ) ;
```

Предположим, что таблица содержит только обычные строки с данными о деталях P1 и P2. В таком случае в результате преобразования этих строк в формат XML может быть получен документ с данными, который выглядит примерно следующим образом.

```
<p>
  <row>
    <PNUM>P1< / PNUM>
    <PNAME>Nut</PNAME>
    <COLOR>Red</COLOR>
    <WEIGHT>12.0</WEIGHT>
    <CITY>London</CITY>
  </row>
  <row>
    <PNUM>P2</PNUM>
    <PNAME>Bolt</PNAME>
    <COLOR>Green</COLOR>
    <WEIGHT>17.0</WEIGHT>
    <CITY>Paris</CITY>
  </row>
</p>
```

Кроме того, вырабатывается документ схемы, который имеет следующий вид.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2 001/XMLSchema">

  <xsd:simpleType name="CHAR 6">
    <xsd:restriction base="xsd:string">
      <xsd:length value="6"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="CHAR 20">
 <xsd:restriction base="xsd:string">
 <xsd: length value="20"/>
 </xsd:restriction>
  </xsd:simpleType>

 <xsd:simpleType name="DECIMAL 5 1">
  <xsd:restriction
  base="xsd:decimal">
  <xsd:totalDigits value="5"/>
  <xsd:fractionDigits value="1"/>
  </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType
    name="RowType.P">
    <xsd:sequence>
```

[23] Эти упрощения состоят в следующем — опущена конструкция PRIMARY KEY, поскольку в [27.15] не приведены сведения о преобразовании ключей в формат XML, исключены случаи применения определяемых пользователем типов и опущены спецификации NOT NULL.

```
<xsd:element name="PNUM" type="CHAR 6"/>
<xsd:element name="PNAME" type="CHAR 20"/>
<xsd:element name="COLOR" type="CHAR 6"/>
<xsd:element name="WEIGHT" type="DECIMAL 5 1"/>
<xsd: element name="CITY" type="CHAR 2 0 "/>
</xsd:sequence> </xsd:complexType>

<xsd:complexType name="TableType.P">
  <xsd: sequence>
<xsd:element name="row"
type="RowType.P" minOccurs="0"
maxOccurs="unbounded"/> </xsd:seguence>
</xsd:complexType>

<xsd:element name="P"

type="TableType.P"/> </xsd:schema>
```

Применение "столбца XML"

Теперь перейдем к изучению средств поддержки "столбца XML" в языке SQL. В спецификации SQL/XML введен новый тип данных, называемый просто XML (а не XMLDOC, как было описано в разделе 27.6), значениями которого фактически являются документы XML или их фрагменты (здесь под термином "фрагмент" подразумевается, например, отдельный элемент XML или последовательность таких элементов). Для формирования или "выработки" значений типа XML из обычных данных SQL предусмотрены различные операторы. Ниже приведен простой пример.

```
INSERT INTO RESULT ( XMLCOL)
      SELECT XMLGEN ( '<Result>
                       <SNAME>{SX.SNAME}</SNAME>
                       < PNAME>{PX.PNAME}</PNAME>
                       <QTY>{SPX.QTY}</QTY>
                       </Result>',
                       SX.SNAME, PX.PNAME, SPX.QTY ) AS
      Result FROM  S AS SX, P AS PX, SP AS SPX WHERE
      SX.SNUM = SPX.SNUM AND   PX.PNUM = SPX.PNUM ;
```

Здесь предполагается, что столбец XMLCOL таблицы RESULT относится к типу XML. Аналогия между этим примером и первым примером запроса XQuery из раздела 27.5 не является простым совпадением. Она фактически означает, что определение оператора XMLGEN почти наверняка изменится еще до того, как произойдет утверждение спецификации SQL/XML, поскольку определение этого оператора основано еще на одном предложении по дополнению языка XQuery, которое само ко времени написания этой книги все еще находилось "в процессе разработки".

Поэтому следует еще раз отметить, что в спецификации SQL/XML введен новый тип XML, но в ней не определено почти ни одной операции[24] со значениями этого типа — не предусмотрена даже операция проверки на равенство! В [27.15] фактически сказано следующее: "Если оба значения, V1 и V2, относятся к типу XML, то проверка того, являются ли V1 и V2 идентичными или нет..., зависит от реализации". Но такая ситуация, по-видимому, будет исправлена ко времени формального утверждения спецификации SQL/XML. А в [27.11] указано, что ко времени ее утверждения (или, возможно, спустя некоторое время) могут быть также дополнительно введены перечисленные ниже средства.

- Поддержка "маскирующих" выражений для конструкций XPath или XQuery.
- Возможность проверки того, является ли указанное значение типа XML формально правильным элементом XML или допустимым документом XML, соответствует ли оно некоторой заданной схеме XML и т.д.

Но даже после введения всех этих усовершенствований предусмотренный в спецификации SQL/XML доступ к данным XML все еще будет оставаться в основном предназначенным для обеспечения только чтения, а полноценный доступ будет по-прежнему от-

Специализированные средства поддержки

Как было отмечено в разделе 27.5, в некоторых программных продуктах SQL (например, в таких СУБД, как DB2 и Oracle) уже предусмотрены специализированные средства поддержки для выборки и обновления данных XML. Автор не ставил перед собой задачу излагать в данном разделе подробные сведения о конкретных программных продуктах, но здесь приведены некоторые гипотетические примеры, позволяющие показать, какого рода функциональные средства обычно предоставляют эти программные продукты. Неформально можно отметить, что эти примеры основаны на программном продукте "XML Extender" компании IBM для СУБД DB2, но в них внесены упрощения для устранения таких особенностей, которые являются несущественными с точки зрения изложения темы данного раздела.

В программном продукте "XML Extender" используется предусмотренная в языке SQL поддержка функций, определяемых пользователем (см. главу 5), для предоставления множества функций, которые (с точки зрения пользователя) по существу являются встроенными функциями. Эти функции могут вызываться из кода SQL и предоставляют целый ряд возможностей по выборке и обновлению данных XML. В этом разделе будут показаны функции XMLFILETOCLOB, XMLCONTENT, XMLEXTRACTREAL и XMLUPDATE (это — не настоящие, а вымышленные имена функций). Возможности, предоставляемые этими функциями, описаны ниже.

- Сохранение документа XML в виде значения столбца SQL.
  Пример. В приведенном ниже предложении с оператором UPDATE, во-первых, используется функция XMLFILETOCLOB для преобразования документа XML,

[24] В спецификации определен оператор XMLSERIALIZE, который преобразует значение типа XML в форму символьной строки. Кроме того, в данной спецификации определен оператор XMLPARSE для "выполнения обратного действия", т.е. для преобразования документа XML или фрагмента документа, представленного в виде символьной строки, в тип XML. Кроме всего прочего, эти два оператора предоставляют определенную поддержку для разделения и публикации.

хранящегося во внешнем файле BoltDrawing.svg в тип CLOB, а затем, во-вторых, происходит сохранение полученной строки типа CLOB в виде значения столбца DRAWING в строке таблицы P, относящейся к детали части P2, следующим образом.

```
UPDATE P
SET    DRAWING = XMLFILETOCLOB ( 'BoltDrawing.svg' )
WHERE  PNUM = 'P2' ;
```

Безусловно, здесь предполагается, что таблица p действительно включает столбец DRAWING и что этот столбец имеет тип CLOB.

- Выборка подобного значения столбца **SQL.**

  Пример. **В** приведенном ниже предложении с оператором SELECT выполняется выборка значения CLOB, записанного в предыдущем примере, и его публикация в виде документа XML во внешнем файле RetrievedBoltDrawing. svg.

```
SELECT XMLCONTENT ( DRAWING, 'RetrievedBoltDrawing.svg' )
FROM   P
WHERE  PNUM = 'P2' ;
```

- Выборка указанного компонента документа XML.

  Пример. Снова предположим, что документ PartsRelation хранится в файле PartsRelation.xml. В таком случае приведенное ниже предложение с оператором UPDATE позволяет, во-первых, извлечь из этого документа значение WEIGHT, относящееся к детали P3, и преобразовать его в тип REAL, а затем, во-вторых, сохранить это значение в качестве значения столбца WEIGHT в строке таблицы p с данными о детали P3.

```
UPDATE P
SET    WEIGHT = XMLEXTRACTREAL
              ( 'PartsRelation.xml',
                '//PartTuple[PNUM = "P3"]/WEIGHT' )
WHERE PNUM = 'P3' ;
```

  Примечание. Здесь принято фиктивное предположение, что столбец WEIGHT таблицы P имеет тип REAL, а не NUMERIC (5,1), поскольку программный продукт "XML Extender" в настоящее время не поддерживает функцию "извлечения в виде значения NUMERIC". Еще более важным является такое замечание, что XMLEXTRACTREAL и другие функции "извлечения" могут применяться к документам XML, хранящимся в столбцах SQL, а не только к документам XML, хранящимся во внешних файлах.

- Обновление указанного компонента документа XML.

  Пример. Примем немного нереальное предположение, что таблица SP включает столбец PARTDETAIL типа CLOB, значением которого в любой заданной строке является документ XML с описанием рассматриваемой детали. В таком случае приведенное ниже предложение с оператором UPDATE устанавливает значение компонента COLOR этого документа XML, равное Green для любой детали, поставляемой поставщиком S4.

```
UPDATE SP
SET    PARTDETAIL = XMLUPDATE
                  ( PARTDETAIL,
                    '//PartTuple/COLOR', 'Green'
) WHERE SNUM = 'S4' ;
```

27.8. РЕЗЮМЕ

В этой главе рассматривается связь между языком XML и базами данных. Но для того чтобы подготовить почву для такого изложения, необходимо было рассмотреть целый ряд дополнительных тем, поэтому в данной главе вначале кратко описана система World Wide Web, а затем приведены гораздо более подробные сведения о языке XM L как таковом.

Язык XML был определен на основе раннее созданных языков SGML и HTML; имя "XML" является сокращением от "Extensible Markup Language" (расширяемый язык разметки), но (как и SGML) XML фактически является **метаязыком** или даже "метаметаязыком". Конкретным приложением XML (т.е. **языком, производным от XML,** согласно определению, приведенному в начале этой главы) является язык, предназначенный для определения документов XML некоторого особого рода (например, документов PartsRelation). Первоначально язык XML не имел ничего общего с базами данных; вместо этого он был предназначен просто для обеспечения "передачи, приема и обработки универсальных документов SGML в системе Web с применением таких способов, которые в настоящее время возможны благодаря использованию языка HTML" [27.25]. Безусловно, нельзя оспаривать необходимость хранить данные XML в базах данных и манипулировать с ними с помощью СУБД. Но на основании этого факта некоторые специалисты пришли к выводу, что язык XML должен стать основой технологии баз данных как таковой.

Любой **документ XML** состоит главным образом из правильно вложенных иерархических конфигураций элементов, каждый из которых включает пару разграничительных **дескрипторов.** Любой конкретный элемент может содержать **символьные данные,** вложенные элементы или сочетание того и другого. Поддерживаются **пустые** элементы. Начальный дескриптор может дополнительно включать непустое множество **атрибутов.** Было показано, что документ XML может использоваться для представления отношения, но он обязательно налагает на кортежи упорядочение сверху вниз и при этом, возможно, также упорядочение атрибутов слева направо.

В основе любого конкретного документа XML лежит абстрактная структура, называемая **инфонабором,** с которой могут проводиться манипуляции с помощью API-интерфейса, называемого **объектной моделью документа,** и на базе которой могут выполняться запросы с использованием языка **XQuery.** В отношении документов XML иногда приходится также встречать утверждение, что они соответствуют **слабоструктурированной модели данных,** но фактически документы XML структурированы не в меньшей и не в большей степени, чем отношения; в действительности, автор не видит существенного различия между слабоструктурированной моделью и старой иерархической моделью (по крайней мере, с точки зрения структурных особенностей этих моделей).

Любой конкретный документ XML является **формально правильным** по определению. Он может быть также **допустимым;** под этим подразумевается, что он соответствует некоторому заданному определению типа документа (Document Type Definition — DTD). Правила написания определений DTD являются составной частью стандарта XML (фактически любое определение DTD представляет собой определение некоторого языка, производного от

XML). Но определения DTD характеризуются целым рядом недостатков, в частности, они лишь в очень небольшой степени обеспечивают поддержку ограничений целостности. Язык **XML Schema—** это метаязык, поддерживающий формирование схем XML, которые могут использоваться для создания более строгих и более подробных описаний документов XML (в частности, применительно к типам данных, хотя рассматриваемые типы едва ли можно считать настоящими типами в том смысле, какой был указан в главе 5). Процесс контроля того, соответствует ли конкретный документ XML заданной схеме XML, называется **проверкой по схеме.**

Далее, в данной главе рассматривались языки **XQuery** и **XPath** (второй из них является строгим подмножеством первого), которые предоставляют доступ только для чтения к данным XML, или, что более точно, к абстрактной или синтаксически проанализированной форме таких данных (т.е., по существу, к инфонабору). Здесь не была предпринята попытка подробно описать какой-либо из этих языков, но приведено несколько примеров, позволяющих получить определенное представление о том, какие действия могут осуществляться с их помощью. В этой главе были описаны **обозначения пути,** которые фактически позволяют пользователю **переходить** вдоль некоторого указанного пути в инфонаборе к некоторой намеченной цели. Кроме того, показано, какую важную роль в таких выражениях выполняет понятие текущего положения, и поставлена под вопрос необходимость применения такого средства (безусловно, им определяется общий процедурный "внешний вид" и языка Xpath, и языка XQuery, а именно к процедурному подходу, применяемому в этих языках, автор относится довольно критически). Затем было отмечено, что язык XPath фактически представляет собой просто **схему адресации,** поскольку он может использоваться для перемещения по существующим узлам в иерархии, но не позволяет создавать новые узлы (для этого требуется языкXQuery).

После этого был представлен ряд примеров применения языка XQuery, в которых, в частности, иллюстрировалось использование **выражений FLWOR** (FLWOR — сокращение от for + let + where + order by + return). В связи с этим была проведена аналогия между такими выражениями, а также, во-первых, выражениями реляционного исчисления и, во-вторых, вложенными циклами обычных языков программирования; на основании этого был сделан вывод, что вторая из указанных аналогий является более подходящей, чем первая. Кроме того, было затронуто несколько вопросов, касающихся оптимизации. Наряду с этим, было отмечено отсутствие в языке XQuery какой-либо явной поддержки операций соединения (как было и в первоначальной версии языка SQL).

На следующем этапе были описаны три перечисленных ниже способа, которые могут применяться для хранения документа XML в базе данных.

1. Применение "столбца XML". Может быть предусмотрено хранение всего документа в виде значения некоторого атрибута определенного кортежа. Для реализации этого подхода требуется создание **нового типа данных,** скажем, XMLDOC (безусловно, ос нащенного операциями, предназначенными для работы со значениями и пере менными этого типа).
2. Применение "коллекции XML". Может быть предусмотрено **разделение** документа с последующим представлением различных его фрагментов в виде различных зна чений атрибутов различных кортежей в различных отношениях.

   Примечание. Процедура, обратная разделению (т.е. преобразование данных, отличных от XML, в форму XML), называется **публикацией.** Процедуры разделения и

публикации могут рассматриваться как способ создания **представлений XML** для данных, отличных от XML (процедура публикации обеспечивает поддержку операции выборки, а процедура разделения — поддержку операции обновления).

3. Применение специализированной базы данных. Может быть предусмотрено хранение документа **собственно** в базе **данных XML** (т.е. в базе данных, которая содержит документы XML как таковые, а не отношения).

В данной главе описаны преимущества и недостатки этих подходов.

Наконец, была кратко описана спецификация **SQL/XML** (которая, по-видимому, будет окончательно включена в стандарт SQL, рабочая версия которого выпушена в 2003 году). В спецификации SQL/XML предусмотрена поддержка публикации данных SQL в форме XML (хотя, по мнению автора, такая поддержка не совсем оправдана). Кроме того, в этой спецификации введен новый тип данных, называемый XML, значениями которого являются документы или фрагменты XML; тем самым предусмотрена возможность хранить данные XML в столбцах SQL, но в этой спецификации предусмотрено лишь немного операций для работы с такими данными. Глава завершается кратким описанием **поддержки** XML в программных продуктах специального назначения на примере поддержки, предусмотренной для СУБД **DB2.**

УПРАЖНЕНИЯ

27.1. Дайте определения следующим терминам:

атрибут	URL	язык XML Schema
язык SGML	язык XML	разметка
Web-узел	язык HTTP	Web-страница
элемент	Web-браузер	язык XPath
дескриптор	язык, производный отXML	машина поиска
система World Wide Web	сеть Internet	Web-сервер
язык HTML	Web-навигатор	язык XQuery

27.2. Как связаны друг с другом языки XML, HTML и SGML?

27.3. Рассмотрите содержание, приведенное в начале данной книги. Покажите, каким образом можно представить это содержание в виде документа XML. При подго товке ответа составьте внутреннее определение DTD.

27.4. Пересмотрите ответ на упр. 27.3 и преобразуйте определение DTD во внешнее. В чем состоят преимущества внешнего определения DTD?

27.5. Что означают термины:

а) формально правильный документ XML;

б) допустимый документ XML.

27.6. Что такое пустой элемент?

27.7. Относятся ли иерархические структуры, применяемые в документах XML, к ие рархиям вложения, втом смысле, который указан в главе 25?

27.8. В подразделе "Недостатки определений DTD" раздела 27.4 определения DTD бы ли подвергнуты критике на том основании, что сами они не представлены на язы ке XML. ("И на самом деле, если язык XML действительно является таким гибким и мощным, как часто приходится слышать, то он должен предоставлять возмож ность описать самого себя!") Распространяются ли аналогичные критические за мечания и на определения данных в языке SQL? А на определения данных в реля ционной модели? Обоснуйте свой ответ.

27.9. Представьте отношение проектов, приведенное на рис. 4.5 (см. стр. 154) в виде документа XML. Для представления значений данных используйте элементы XML, а не атрибуты. В какой степени в этом документе могут быть предписаны ограничения уникальности?

27.10. Повторите упр. 27.9, но для представления значений данных примените атрибуты XML. В чем состоят преимущества использования атрибутов? Каковы недостатки?

27.11. Предположим, что ответы на упр. 27.9 и 27.10 расширены для включения отноше ний поставщиков, деталей и отгрузок. В какой степени могут быть предписаны ог раничения ссылочной целостности?

Для выполнения упр. 27.12-27.14 читателю, по-видимому, придется воспользова-ться официальной документацией по языку XML Schema [27.28] или некоторым аналогичным справочным источником (в данной главе отсутствует достаточно подробная информация, которая позволила бы дать полные ответы на эти упраж-нения).

27.12. Создайте схему XML для подготовленного вами ответа наупр. 27.3.

27.13. Рассмотрите документ PartsRelation из раздела 27.3. Создайте схему XML для документов в этой форме, которая не налагает упорядочение на элементы PartTuple.

27.14. В разделе 27.4 было приведено утверждение, что типы данных XML Schema в дей ствительности не соответствуют обычно принятому определению такого понятия, как тип данных. Согласны ли вы с этим утверждением? Обоснуйте свой ответ.

27.15. Дайте определение термина "инфонабор"?

27.16. Что такое "обозначение пути"?

27.17. Что такое "выражение FLWOR"? В чем состоит существенное различие между конструкцией for и конструкцией let? Когда следует использовать предикат, а не конструкцию where (и наоборот)?

В упр. 27.18-27.21 применяется документ PartsRelation из раздела 27.4. Все ре-зультаты должны быть формально правильными.

27.18. Составьте выражение XQuery для формирования перечня всех элементов PartTuple, которые содержат элемент NOTE.

27.19. Составьте выражение XQuery для формирования перечня всех деталей зеленого цвета, в котором каждый результирующий элемент PartTuple включен в элемент GreenPart.

27.20. Если вычисление приведенного ниже выражения XQuery будет выполнено с использованием версии документа PartsRelation, в которой представлены все шесть деталей P1-P6, то к чему это приведет?

```
<Parts>
  {
count(document("PartsRelation.xml")//PartTuple) }
</Parts>
```

27.21. Предположим, что имеется документ SuppliersOverShipments (см. рис. 27.4), а также документ PartsRelation. Составьте выражение XQuery для формирования перечня поставщиков, которые поставляют не меньше одной детали синего цвета.

27.22. Если бы документ SuppliersOverShipments, рассматриваемый в упр. 27.21, представлял все данные о поставщиках и отгрузках, приведенные на рис. 3.8 (см. стр. 119), то к чему привело бы выполнение следующего выражения?

```
for $sx in document("SuppliersOverShipments.xml")/
          Supplier[CITY = 'London']
return
  <Result>
    { $SX/SNUM, $SX/SNAME, $Sx/STATUS,
  $Sx/CITY } </Result>
```

27.23. В чем состоит семантическое различие между следующими двумя конструкциями return (если таковое существует)?

```
return <Result> { $a, $b }
</Result> return <Result> { $a } {
$b } </Result>
```

27.24. Каким образом может быть предусмотрено хранение данных XML в базе данных? В чем состоят преимущества и недостатки каждого подхода?

27.25. Рассмотрите функции, (кратко) описанные в подразделе "Специализированные средства поддержки" раздела 27.6. Выскажите свое мнение по поводу проектирования этих функций.

27.26. Иногда приходится сталкиваться с таким утверждением, что документ XML напоминает кортеж, в том толковании этого понятия, которое принято в сообществе специалистов по реляционным базам данных. Обсудите это утверждение.

27.27. Иногда встречаются утверждения, что данные XML являются "лишенными схемы" (schemaless). Каково ваше мнение по поводу выполнения запросов к данным, которые вообще не имеют схемы? Как бы вы спроектировали язык запросов для таких данных?

27.28. В разделе 27.3 было указано, что структура, которой обладает документ XML, в значительной степени зависит от того, какая структура накладывается на данные проектировщиком документа. Относится ли аналогичное замечание к реляционным данным? Если нет, то почему? Обоснуйте свой ответ.

27.29. Если вы знакомы с иерархической моделью данных, то укажите все различия (которые сможете выявить) между ней и "слабоструктурированной моделью", кратко описанной в данной главе.

27.30. Обсудите следующую Цитату из [27.4]: "[При использовании языка] XML приходится избегать фундаментального вопроса о том, *что* мы должны сделать, полностью сосредотачиваясь на том, *как* мы должны это сделать".

СПИСОК ЛИТЕРАТУРЫ

27.1. Abiteboul S., Buneman P., Suciu D. Data on the Web: From Relations to Semistructured Data and XML. San Francisco, Calif.: Morgan Kaufmann, 1999.

В рекламном проспекте издательства к этой книге сказано: "Современное исследование быстро развивающихся стратегий выборки и обработки реляционных и слабоструктурированных данных".

27.2. Berners-Lee T., Fischetti M. Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor. San Francisco, Calif.: Harper San Francisco, 1999. См. также Berners-Lee T. Information Management: A Proposal // http://www.w3.org/History/1989/Proposal.html— оригинальный документ с описанием проекта Web. В институте CERN, где работал Бернерс-Ли, было распространено в качестве предложений для обсуждения несколько версий этого документа, но они так и не были опубликованы какой-либо официальной организацией.

27.3. Bernstein P. et al. The Asilomar Report on Database Research, ACM SIGMOD Record. — December 1998. — 27, №4.

В этой статье приведены следующие довольно пессимистические замечания по поводу применения языка XML для баз данных: "К сожалению, существует вероятность того, что внедрение XML породит хаос в системах баз данных. Разрабатываемый язык запросов XML напоминает процедурные языки обработки запросов, которые доминировали лет 25 тому назад. Применение XML требует также развертывания в клиентской части приложения кэшей данных, предназначенных для поддержки операций обновления, а это вынуждает разработчиков XML заниматься решением огромного количества проблем, связанных с распределенными транзакциями. К сожалению, основная часть работ в области XML осуществляется почти без влияния со стороны сообщества специалистов по системам баз данных".

27.4. Boiko B. Understanding XML // http://metatorial.com/papers/xml.asp, 2000.

27.5. Bonifati A., Ceri . Comparative Analysis of Five XML Query Languages // ACM SIGMOD Record. - March 2000. -29, №1.

Пятью языками запросов XML, упомянутыми в заголовке этой статьи, являются XSL, XQL, Lorel, XML-QL и XML-GL (как описано в разделе 27.3, XSL фактически является языком таблиц стилей, или таблиц форматирования, но может использоваться для формирования простых запросов, так же, как и язык XSLT). Языки XSL и XQL поддерживают запросы, касающиеся только одного документа XML, а остальные языки поддерживают запросы, охватывающие несколько документов. В языке XML-GL предоставляется примерно такой же графический интерфейс, как в языке QBE. Языки Lorel и XML-GL включают средства обновления. См. также [27.16].

27.6. Bosak J., Bray T. XML and the Second-Generation Web // http: //www.sciam.com. — May 1999.

Эта статья содержит превосходное доказательство того, что язык XML не должен использоваться в качестве основы для новой технологии баз данных (хотя, возможно, сформулированное вопреки намерению авторов). Приведем одну цитату: "[Документы] XML имеют структуру, которую в информатике принято называть древовидной... Деревья не позволяют представить информацию любого вида, но способны отобразить большинство видов информации, для обработки которой требуется применение компьютеров. Более того, древовидные структуры являются чрезвычайно удобными для программистов (?!). Если предназначенный для обработки банковский документ преобразован в форму дерева, то совсем несложно написать небольшую программу, которая переупорядочит описанные в нем транзакции или покажет только оплаченные чеки". Вполне возможно, что авторы правы; сделанные ими выводы точны в той части, которой они касаются, но рассмотрена ли данная проблема достаточно полно? Изучение истории применения древовидных (иными словами, иерархических) структур в контексте базы данных наглядно показывает, что ответ на этот вопрос является отрицательным. При этом фундаментальная идея состоит в том, что даже если данные по своей природе имеют иерархическую структуру (что можно показать, допустим, в примере с отделами и служащими), из этого не следует, что они обязательно должны быть представлены в иерархической форме. Дело в том, что иерархическое представление не подходит для всех видов операций обработки, которые может потребоваться применить к этим данным. А как быть с данными, которые не являются "иерархическими по своей природе"? Например, является ли древовидное представление наиболее подходящим для высказываний в форме: "Поставщик s поставляет деталь p для проекта j "?

Примечание. Автор выдвинул аналогичные возражения в разделе 25.1 в связи с объектами, которые также являются иерархическими, как и документы XML.

27.7. Bourret R. XML and Databases // http: //rpbourret .com/xml/XMLAndDatabases.htm. — November 2002.

Содержательное введение и обзор. Приведем одну цитату: "В данной статье приведено общее описание вопросов применения XML в базах данных. В ней рассматриваются... различия между [приложениями], ориентированными на обработку данных и ориентированными на обработку документов, показано, как обычно используется язык XML в реляционных базах данных, описано, что такое собственно базы данных XML и при каких условиях они должны применяться".

27.8. Buneman P., Fan W., Simeon J., Weinstein S. Constraints for Semistructured Data and XML//ACM SIGMOD Record. - March 2001. - 30, № 1.

27.9. Chamberlin D.D., Robie J., Florescu D. QUILT: An XML Query Language for Heterogeneous Data Sources // Dan Suciu and Gottfried Vossen (eds.). Lecture Notes in Computer Science 1997. New York, N.Y.: Springer-Verlag, 2000.

27.10. Chamberlin D.D. XQuery: An XML Query Language // IBM Sys. J. - 2002. - 41, № 4.

Автор этой статьи, Чемберлен, был одним из первых разработчиков языка SQL [4.9]—[4.11], а теперь является членом рабочей группы консорциума W3C, на которую возложена ответственность за определение языка XQuery. Данная статья является полезным учебным руководством, но в ней содержатся некоторые спорные замечания, включая, в частности, следующее: "Итерация — это важная часть любого языка запросов". Такое заявление прямо противоречит утверждению Кодда, что "для извлечения любой информации [из] базы данных ни прикладной программист, ни рядовой пользователь (не являющийся программистом) [не должны нуждаться] в разработке каких-либо итерационных или рекурсивных циклов" (девятый из "Фундаментальных законов управления базой данных" Кодда [6.2]).

27.11. Eisenberg A., Melton J. SQL/XML and the SQLX Informal Group of Companies // ACM SIGMOD Record. - September 2001. - 30, № 3; SQL/XML Is Making Good Progress//ACM SIGMOD Record- June 2002. -31, №2.

27.12. Florescu D., Levy A., Mendelzon A. Database Techniques for the World-Wide Web: A Survey//ACM SIGMOD Record - September 1998. - 27, № 3.

Эта статья скорее касается применяемых в Web данных в общем, а не данных XML в частности. В ней приведен обзор проектов, прототипов и языков, имеющих отношение к "обеспечению применимости концепций баз данных для решения проблем управления и выборки" данных Web. В данной статье приведена обширная библиография.

27.13. Garcia-Molina H., Ullman J.D., Widom J. Database Systems: The Complete Book. Upper Saddle River, N.J.: Prentice Hall, 2002.

27.14. Harold E.R. XML Bible (2d ed.). New York, N.Y.: Hungry Minds, Inc., 2001 ;>

27.15. International Organization for Standardization (ISO). XML-Related Specifications (SQL/XML) Working Draft,. Document ISO/IEC JTCl/SC32/WG3:DRS-020.- August 2002.

Учебник, основанный на одной из предыдущих версий этого документа, можно найти в [26.32]. См. также [27.11].

27.16. Lee **D.,** Chu **W.W.** Comparative Analysis of Six XML Schema Languages // ACM SIGMOD Record.-September 2000-29, №3.

Шестью языками схем XML, о которых идет речь в названии статьи, являются XML Schema, XDR, SOX, Schematron, DSD и тот язык, который был в данной главе назван языком определения DTD. Последний из этих шести языков, очевидно, является самым слабым, а язык XML Schema принадлежит к числу наиболее сильных. См. также [27.5].

27.17. Lewis P.M., Bernstein A., Kifer M. Databases and Transaction Processing: An Application-Oriented Approach. Boston, Mass.: Addison-Wesley, 2002.

27.18. McHugh J., Widom J. Query Optimization for XML // Proc. 25th Int. Conf. on Very Large Data Bases, Edinburgh, Scotland. — September 1999.

Отчет по результатам экспериментов с оптимизирующим компонентом Lore — "СУБД для данных, основанных на XML, поддерживающая выразительный язык запросов [называемый Lorel]".

27.19. Nelson Th.H. A File Structure of the Complex, the Changing, and the Indeterminate // Proc. 20th Nat. ACM Conf., Cleveland, Ohio. — August 24-26, 1965. См. также Nelson Th.H. Literary Machines. Sausalito, Calif.: Mindful Press, 1993. (Первое издание опубликовано в 1982 году.)

Эта статья Нельсона, опубликованная в 1965 году (в которой впервые появился термин "гипертекст"), основана на результатах исследовательских работ Венневара Буша (Vannevar Bush) (Memex, 1945) и Дугласа Энгелбарта (Douglas Engelbart) (NLS: oNLine System, 1963).

27.20. Pascal F. Managing Data with XML: Forward to the Past? // http: //searchdatabase. techtarget. com. — January 2001.

В [27.6] Босак и Брэй фактически предлагают отделить часть функциональных средств, которые должны входить в состав СУБД, и вместо этого переместить их в прикладные программы. Автор данной статьи, Паскаль, доказывает, что такое предложение — это возврат к прошлому.

27.21. Tatarinov I., Ives Z.G., Halevy A.Y., Weld D.S. Updating XML // Proc. 2001 ACM SIGMOD Int. Conf. on Management of Data, Santa Barbara, Calif. — May 2001.

В статье предложен набор расширений для языка XQuery, поддерживающий операции обновления, и описана реализация, которая преобразует эти обновления через "представления XML" в основополагающие данные SQL.

27.22. The Unicode Consortium: The Unicode Standard, Version 4.0. Reading, Mass: Addison-Wesley, 2003.

27.23. Widom J. Data Management for XML // http://www-db.stanford.edu/~widom/xml-whitepaper .html. —June 17, 1999.

27.24. W3C. Document Object Model (DOM) Level 3 Core Specification Version 1.0 Working Draft//http://www.w3.org/TR/DOM-Level-3-Core — February 26, 2003.

27.25. W3C. Extensible Markup Language (XML) 1.0 (2d ed.) // http://www.w3.org/TR/REC-xml, October 6, 2000. См. также Bray T. The Annotated XML 1.0 Specification //http: //www. xml. com (перейдите по ссылке "Annotated XML").

27.26. W3C. XML Information Set//http://www. w3 .org/TR/xml-infoset. — October 24,2001.

27.27. W3C. XML Path Language (XPath) Version 2.0 Working Draft // http: //www. w3. org/TR/xpath20. - May 2, 2003.

27.28. W3C. XML Schema Part 0: Primer; Part 1: Structures; Part 2: Datatypes // http://www.w3.org/TR/xmlschema-0/, -1/, -2/, May 2, 2001.

27.29. W3C. XQuery 1.0: An XML Query Language Working Draft // http://www.w3.org/TR/xquery. — May 2, 2003.

Примечание. К языку XQuery относятся также перечисленные ниже документы W3C.

- XML Query Requirements Working Draft// http://www.w3.org/TR/xquery-requirements, May 2, 2003.
- XQuery 1.0 and XPath 2.0 Data Model Working Draft// http: //www. w3 .org/TR/xpathdatamodel. — May 2, 2003.
- XQuery 1.0 and XPath 2.0 Formal Semantics Working Draft // http: //www.w3 .org/TR/xquery-semantics. — May 2, 2003.
- XQuery 1.0 and XPath 2.0 Functions and Operators Working Draft// http://www.w3.org/TR /xpath-functions. — May 2, 2003.
- XML Query Use Cases Working Draft // http: //www. w3 . org/TR/xquery-use-cases. — May 2, 2003.

27.30. XML:DB. XML Update Language Working Draft // http://www.xmldb.org/xupdate/xupdatewd.html. — September 14, 2000.

XML:DB — это открытый промышленный консорциум (а не организация по стандартизации), который "в своем уставе взял на себя обязательства по разработке конкретных спецификаций баз данных XML" (?!). Этот консорциум был основан в 2000 году, поскольку, как указано в данной работе, "базы данных XML... имеют гораздо более широкую область применения, чем даже сама система World Wide Web". Описанный в этой статье язык XUpdate предназначен для использования в качестве языка обновления для данных XML.

ЧАСТЬ VII

ПРИЛОЖЕНИЯ

В этой книге — четыре приложения. В приложении А приведено вводное описание новой технологии реализации баз данных, получившей название модели TransRelational™. В приложении Б приведены для справок дополнительные сведения о синтаксисе и семантике выражений SQL. В приложении В содержится список наиболее важных сокращений, аббревиатур и символов, введенных в тексте этой книги. Наконец, в приложении Г дано методическое описание наиболее широко применяемых структур хранения и методов доступа, а в приложении Д приведены ответы к отдельным упражнениям.

ПРИЛОЖЕНИЕ А

Модель TransRelational™

A1. ВВЕДЕНИЕ

В различных областях научных исследований время от времени рождаются идеи, которые оказываются настолько новаторскими и превосходящими по значимости все созданное ранее, что их можно смело назвать открытиями. Одним из наглядных примеров такого открытия в мире баз данных явилось создание реляционной модели; почти все, что описано в этой книге, может служить доказательством революционного характера и огромного влияния одной этой блестящей идеи. А теперь мы становимся свидетелями рождения того, что может оказаться еще одним важным открытием, — **модели TransRelational™.** По мнению автора настоящей книги, модель TransRelational, разработанная Стивом Тареном (Steve Tarin) и в дальнейшем называемая сокращенно моделью TR, вполне может оказаться наиболее значительным достижением в этой области с тех пор, как Кодд ознакомил нас с реляционной моделью примерно 35 лет тому назад.

Необходимо сразу же отметить, что модель TR не предназначена для замены реляционной модели; приставка "trans" в слове "transrelational" не означает "оставляющий за собой" (как, например, в слове "трансатлантический"). В данном случае она является сокращением от слова transformation (преобразование). Верно, что и модель TR, и реляционная модель являются абстрактными моделями данных, но модель TR находится на более низком уровне абстракции (т.е. она ближе к структурам физической памяти); в действительности, модель TR, кроме всего прочего, предназначена для использования в качестве средства реализации реляционной модели. Напомним, что в конце главы 18 есть такие слова: "Но в последнее время появились реализации принципиально нового подхода к организации работы СУБД, а этот подход, по сути, ставит под сомнение многие предположения, лежащие в основе указанных эмпирических подходов". Под этим новым подходом подразумевалась модель TR.

Прежде чем перейти к описанию технических подробностей, необходимо привести необходимую подготовительную информацию. В действительности, модель TR как таковая представляет собой конкретное приложение более общей технологии, получившей название **преобразования Тарена** в честь ее разработчика. Метод преобразования Тарена, на который распространяется патент США [A.2], предназначен для использования в качестве технологии реализации для систем хранения и выборки

данных многих типов (а не только для СУБД), включая, например, системы хранилищ данных, инструментальные средства разработки данных, системы SQL, машины поиска Web, системы, основанные на использовании языка XML и т.д. В отличие от этого, тема данного приложения (т.е. модель TransRelational как таковая) просто представляет собой пример применения этой более общей технологии для реализации, в частности, реляционных систем. Но, как вскоре станет очевидно, эта общая технология особенно хорошо приспособлена для реализации именно реляционных систем; в действительности, она разрабатывалась в основном с учетом данной конкретной цели.

Теперь приступим к техническому описанию этой модели. Модель TR очень удобно рассматривать с точки зрения ее использования для решения все той же задачи — обеспечения независимости от данных (точнее, независимости от физических данных). Обеспечение независимости от данных означает проведение четкого различия между логическим и физическим уровнями системы, а проведение такого четкого различия, в свою очередь, требует наличия средств **преобразования** между этими двумя уровнями, с помощью которых логический уровень отображается на физический, и наоборот. Но в большинстве современных СУБД применяется такой способ преобразования, который может рассматриваться почти как взаимно однозначный, предусматривающий отображение базовых переменных отношения на хранимые файлы, а кортежей таких переменных отношения — на хранимые записи в таких файлах. В подобной системе все, что хранится на физическом уровне, можно рассматривать (по крайней мере, в первом приближении) в качестве **непосредственного отображения** того, что пользователь видит на логическом уровне (рис. А. 1). Одним из следствий из указанного факта является то, что подобные системы в действительности не обеспечивают настолько уж значительную независимость от данных. Еще одним следствием является то, что обязательно приходится располагать данные в памяти только в одной физической последовательности, а в результате этого возникает необходимость в использовании индексов и других избыточных структур для поддержки доступа к данным, расположенным в той последовательности, которая отличается от требуемой. Кроме того, возникает и такое следствие, что для достижения приемлемой производительности требуется сложная оптимизация. Наконец, важным следствием является то, что задача администрирования базы данных становится намного более сложной, чем должна была быть.

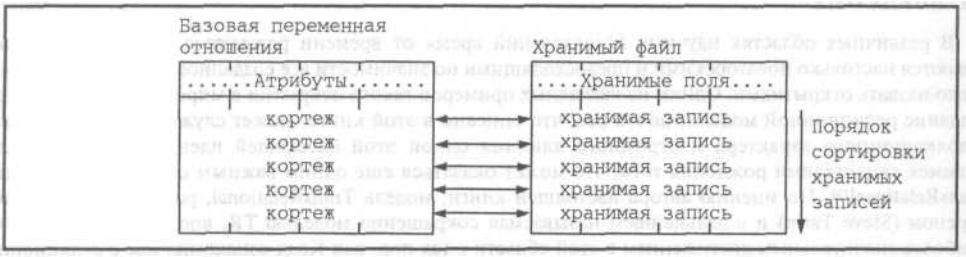


Рис. АЛ. Реализация с прямым отображением

В отличие от этого, в модели TR используются намного более совершённые средства преобразования. Ниже описаны некоторые непосредственные следствия из этого факта.

- Модель TR обеспечивает гораздо большую независимость от данных по сравнению с той, которая достигнута или может быть достигнута в системах с непосредственным отображением.
- В модели TR данные по существу хранятся во многих разных физических последовательностях одновременно, поэтому исключается необходимость в использовании индексов и тому подобных структур.
- При использовании модели TR оптимизация осуществляется значительно проще, чем при использовании систем с непосредственным отображением; зачастую существует только один очевидный и при этом наилучший способ реализации любой конкретной реляционной операции.
- Производительность модели TR на несколько порядков выше по сравнению с системами непосредственного отображения. В частности, производительность операции соединения является линейной! Это фактически означает, что количество времени, которое требуется для

соединения 20 отношений (говоря неформально) только в два раза превышает количество времени, необходимое для соединения 10 отношений. Кроме того, из этого следует такой важный вывод, что система вообще становится способной поддерживать соединение 20 отношений (что само по себе очень сложно), иными словами, система приобретает способность к масштабированию.

- Администрирование системы в значительной степени упрощается, поскольку гораздо реже
  приходится принимать субъективные решения.
- На физическом уровне системы вообще отсутствует такое понятие, как "хранимая переменная
  отношения", или "хранимый кортеж"!

В данном приложении приведено краткое описание принципов работы модели TR. Безусловно, что здесь недостаточно места для описания всех аспектов ее функционирования. Поэтому для того чтобы иметь возможность описать ее, не выходя за установленные рамки, автор решил не рассматривать, во-первых, операции обновления и, во-вторых, операции с вторичной памятью. Иными словами, в этом описании приняты предположения, что база данных предназначена только для чтения и находится в оперативной памяти. Но из этого не следует делать вывод, что модель TR предназначена для использования лишь в базах данных, обеспечивающих только чтение и находящихся в оперативной памяти; дело обстоит совсем иначе. Подробное описание всех областей применения модели TR, включая операции обновления и базы данных, хранимые на диске, приведено в учебном руководстве, которое написано автором настоящей книги [АЛ].

А.2. ТРИ УРОВНЯ АБСТРАКЦИИ

Реляционная система, реализованная с использованием модели TR, может рассматриваться как охватывающая три уровня абстракции: реляционный (или пользовательский) уровень, файловый уровень и уровень модели TR (рис. А.2), которые описаны ниже.

Рис. А.2. Три уровня абстракции

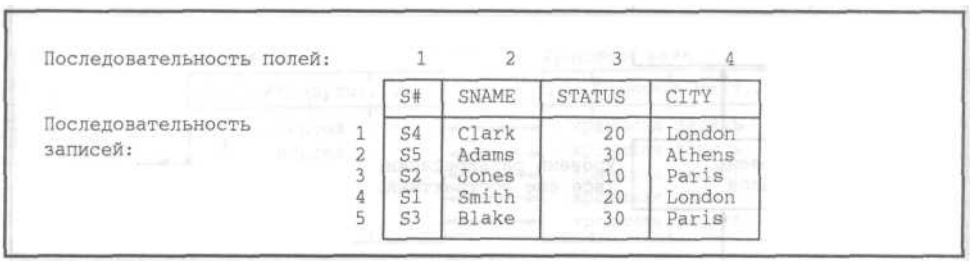
- На верхнем уровне данные представлены в виде отношений, которые обычным образом составлены из кортежей и атрибутов.
- На нижнем уровне данные представлены с помощью различных внутренних структур модели TR, называемых *таблицами,* а сами эти таблицы состоят из строк и столбцов. *Сразу же следует отметить, что указанные таблицы, строки и столбцы не являются конструкциями SQL с теми же именами, а также непосредственно не соответствуют отношениям, кортежам или атрибутам на пользовательском уровне.*
- Средний уровень представляет собой уровень перенаправления между другими двумя уровнями — отношения верхнего уровня отображаются на файлы среднего уровня, а затем эти файлы

отображаются на таблицы низкого уровня. Кроме того, указанные файлы состоят из записей и полей; записи соответствуют кортежам, а поля — атрибутам верхнего уровня.

■ *Примечание.* На основании этого описания не следует делать ошибочный вывод, что файлы хранятся физически; они представляют собой такую же абстракцию физически хранимых данных, как и сами переменные отношения (а в конечном итоге такой же абстракцией хранимых данных являются и таблицы TR). Но вполне допустимо рассматривать их "немного более близкими к физическому уровню", чем переменные отношения (и вместе с тем менее близкими к физическому уровню, чем таблицы TR).

Начиная с этого момента, в настоящем приложении скрупулезно соблюдается соглашение о том, что реляционная терминология используется при описании верхнего уровня, файловая терминология — среднего уровня, а табличная терминология — нижнего уровня. Для упрощения предполагается также, что все отношения и переменные отношения являются именно базовыми отношениями и переменными отношения, если явно не указано иное.

Таким образом, первым этапом преобразования любой конкретной переменной отношения в соответствующее представление TR является отображение соответствующего отношения на файл, записи которого соответствуют кортежам, а поля — атрибутам. Например, на рис. А3 показан один из возможных файлов, соответствующих обычно применяемому отношению поставщиков. В таком файле записи имеют упорядочение сверху вниз, а поля — упорядочение слева направо (как показано с помощью номеров записей и номеров полей на этом рисунке). Но рассматриваемое упорядочение фактически выбрано произвольным образом, поэтому, например, данное отношение поставщиков вполне можно было бы отобразить на любой из 2880 различных файлов (поскольку имеется 120 разных вариантов упорядочения для пяти записей[1] и 24 разных вариантов упорядочения для четырех полей). С другой стороны, по меньшей мере, все эти 2880 различных файлов эквивалентны друг другу, в том смысле, что все они представляют точно одну и ту же информацию, поэтому иногда удобнее рассматривать их не просто как 2880 различных файлов как таковых, а скорее как 2880 различных версий "одного и того же" файла.



Последовательность полей: 1 2 3 4

Последовательность записей:	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London
2	S5	Adams	30	Athens
3	S2	Jones	10	Paris
4	S1	Smith	20	London
5	S3	Blake	30	Paris

Рис. А.3. Файл для обычно применяемого отношения поставщиков

Теперь файл, подобный приведенному на рис. А.3, может быть представлен с помощью таблиц на уровне TR и реконструирован из этих таблиц TR. При этом следует учитывать тот (очень важный!) факт, что все возможные различные версии одного и того же файла могут быть реконструированы из одних и тех таблиц TR одинаково легко (здесь термин *версия* используется в том смысле, который был только что определен); это означает, что в разных версиях файла упорядочение записей и полей может отличаться, а содержание остается одним и тем же. Как достигается такая возможность, будет описано в следующем разделе. В этих таблицах TR строки имеют упорядочение сверху вниз, а столбцы — упорядочение слева направо. Кроме того (это еще одно важное замечание!), места пересечения строк и столбцов в такой таблице (которые будут именоваться ячейками), можно адресовать в стиле адресации массивов, [i, j], где i — номер строки, а j — номер столбца.

[1] Безусловно, не все из этих 120 упорядочений могут быть получены с помощью простой конструкции ORDER BY (например, не может быть получено упорядочение, показанное на рис. А.3).

Подробные сведения об отображении файлов на таблицы TR приведены в следующем разделе, а в этом разделе достаточно подчеркнуть тот факт, что оно ничем не напоминает тот вид непосредственного отображения, который был описан в разделе А.1. В частности, строки таблиц TR не имеют какого-либо взаимно однозначного соответствия записям на файловом уровне, а в силу этого и не имеют какого-либо взаимно однозначного соответствия кортежам на реляционном уровне. В качестве иллюстрации на рис. А.4 показана таблица TR, называемая **таблицей значений полей,** которая соответствует файлу на рис. А.3; в частности, заслуживает внимания то, что (как уже было сказано) ее строки не соответствуют каким-либо очевидным образом тем записям, которые показаны на рис. А3.

Для того чтобы иметь возможность реконструировать показанный на рис. А.3 файл из таблицы значений полей на рис. А.4, требуется еще одна таблица — **таблица реконструкции записей** (рис. А.5). Следует отметить, что значениями в ячейках этой таблицы больше не являются номера поставщиков **или** значения статуса (и т.д.), несмотря на обозначения столбцов; вместо этого в ней находятся номера строк. Дополнительное описание приведено в следующем разделе.

Последовательность столбцов: 1 2 3 4

Последовательность строк:	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	London
3	S3	Clark	20	London
4	S4	Jones	30	Paris
5	S5	Smith	30	Paris

Последовательность столбцов: 1 2 3 4

Последовательность строк:	S#	SNAME	STATUS	CITY
1	5	4	4	5
2	4	5	2	4
3	2	2	3	1
4	3	1	1	2
5	1	3	5	3

Рис. А.5. Таблица реконструкции записей для файла, приведенного на рис. А.3

А.3. ОСНОВНАЯ ИДЕЯ

Наиболее важная мысль, лежащая в основе модели TR, может быть описана следующим образом. Допустим, что r — запись некоторого файла на файловом уровне. В таком случае справедливо приведенное ниже утверждение.

Хранимая форма записи r состоит из двух логически различимых частей — множества значений полей и множества "связующих" данных, позволяющих связать друг с другом эти значения полей, причем для физического хранения каждой из этих частей можно воспользоваться широким спектром возможностей.

В системах с непосредственным отображением эти две части хранятся вместе; иными словами, в таких системах связующая информация задается по принципу физической близости. В отличие от этого, в модели TR эти две части хранятся отдельно — значения полей находятся в таблице значений полей, а связующая информация — в таблице реконструкции записей. А именно такое разделение является основным источником многочисленных преимуществ, которые способна предоставить модель TR.

Таблица значений полей

Итак, читатель, по-видимому, уже определил самостоятельно, как была получена приведенная на рис. А.4 таблица значений полей из файла, который показан на рис. А.3: фактически каждый столбец таблицы содержит значения из соответствующего поля файла, отсортированные в порядке возрастания. Поэтому следует сразу же отметить, что независимо от первоначального расположения записей в файле всегда формируется одна и та же таблица значений полей (в рассматриваемом примере на одну и ту же таблицу значений полей отображаются все 2880 версий файла). Кроме того, даже несмотря на то, что мы еще не имеем возможности описать, как используется эта таблица (поскольку вначале требуется рассмотреть таблицу реконструкции записей), можно сразу же подчеркнуть несколько приведенных ниже особенностей, позволяющих проще понять ее назначение.

- Тот факт, что каждый столбец в таблице значений полей находится в отсортированном порядке, безусловно, способствует более успешному выполнению пользовательских запросов с конструкцией ORDER BY. Например, для выполнения запроса на получение номеров поставщиков в порядке возрастания лексикографического значения названия города не требуется ни сортировка на этапе выполнения, ни индекс.
- То же утверждение остается справедливым применительно к запросу на получение номеров поставщиков в порядке убывания лексикографического значения названия города (т.е. в обратном порядке сортировки), поскольку данная реализация позволяет просто обрабатывать для этого таблицу значений полей снизу вверх, а не сверху вниз.
- Аналогичные утверждения относятся к каждому отдельному атрибуту; это означает, что таблица значений полей фактически представляет множество различных вариантов сортировки одновременно (по существу, в этой таблице каждый отдельный атрибут отсортирован сразу в обоих направлениях — возрастающем и убывающем).
- Запросы, требующие поиска конкретного значения (например, запрос на получение номеров поставщиков из Лондона) может быть реализован с помощью эффективного бинарного поиска. И аналогичные замечания снова касаются каждого атрибута.

В завершение этого подраздела отметим, что таблица значений полей может рассматриваться как своего рода *мост* между пользовательским восприятием данных (под этим подразумевается первоначальное отношение пользовательского уровня и/или соответствующий файл) и другими внутренними структурами TR. В частности, таблица значений полей — это единственная таблица TR, которая содержит пользовательские данные как таковые; все остальные таблицы содержат внутреннюю информацию (как правило, указатели), а эта информация имеет смысл только для модели TR, но непосредственно не касается пользователя и даже вообще ему не предъявляется.

Таблица реконструкции записей

На рис. А.6 таблица значений полей (которая была показана на рис. А.4) находится рядом с таблицей реконструкции записей, приведенной на рис. А.5. Следует отметить, что эти две таблицы являются изоморфными; в действительности, как вскоре будет показано, существует непосредственное взаимно однозначное соответствие между ячейками этих двух таблиц (т.е. обе эти таблицы имеют такое же количество строк и столбцов, сколько записей и полей, соответственно, имеется в файле на рис. А.3). Заслуживает также внимания следующее — как указано в разделе А.2, данные в ячейках таблицы реконструкции записей не представляют собой больше номера поставщиков, имена поставщиков (или другую фактическую информацию); вместо этого они представляют собой **номера строк,** а эти номера строк могут рассматриваться как указатели на строки таблицы значений полей или таблицы реконструкции записей или той и другой таблицы, в зависимости от контекста, в котором указатели используются. (По этой причине фактически столбцы в таблице реконструкции записей не обязательно должны были обозначаться надписями S#, SNAME и т.д., как показано на этом рисунке; тем не менее, применение таких надписей позволит проще следить за некоторыми дальнейшими описаниями.)

Прежде чем приступить к рассмотрению того, как формируется таблица реконструкции записей, кратко опишем, как она используется. Рассмотрим приведенную ниже последовательность операций.

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1	Adams	10	Athens
2	S2	Blake	20	London
3	S3	Clark	20	London
4	S4	Jones	30	Paris
5	S5	Smith	30	Paris

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	5	4	4	5
2	4	5	2	4
3	3	2	3	1
4	2	1	1	2
5	1	3	5	3

Рис. А.6. Таблица значений полей, показанная на рис. А.4, и соответствующая таблица реконструкции записей

Шаг 1. Перейти в ячейку [1,1] таблицы значений полей и считать хранящееся здесь значение, а именно номер поставщика S1. Это значение является первым значением поля (т.е. значением поля S#) в некоторой записи с данными о поставщике в файле поставщиков.

Шаг 2. Перейти в ту же ячейку (т.е. в ячейку [1,1]) таблицы реконструкции записей и считать хранящееся в ней значение, а именно номер строки 5. Этот номер строки интерпретируется как означающий, что значение следующего поля (которое представляет собой значение второго поля, или значение поля SNAME) в записи поставщика, значение поля S# в которой равно S1, можно найти в позиции SNAME пятой строки таблицы значений полей, иными словами, в ячейке [5,2] таблицы значений полей. Перейти в эту ячейку и считать хранящееся там значение (имя поставщика Smith).

Шаг 3. Перейти в соответствующую ячейку таблицы реконструкции записей [5,2] и считать хранящийся в ней номер строки (3). Значение следующего поля (третьего поля, или поля STATUS) в реконструируемой записи поставщика находится в позиции STATUS третьей строки таблицы значений полей, иными словами, в ячейке [3,3]. Перейти в эту ячейку и считать хранящееся в ней значение (статус 20).

Шаг 4. Перейти в соответствующую ячейку таблицы реконструкции записей [3,3] и считать хранящееся в ней значение (которое снова равно 3). Значение следующего поля (четвертого поля, или поля CITY) реконструируемой записи поставщика находится в позиции CITY третьей строки таблицы значений полей, иными словами, в ячейке [3,4]. Перейти в эту ячейку и считать хранящееся в ней значение (название города London).

Шаг 5. Перейти в соответствующую ячейку таблицы реконструкции записей [3,4] и считать хранящееся в ней значение (1). Теперь на первый взгляд может показаться, что значением "следующего" поля в реконструируемой записи поставщика должно быть значение пятого столбца, но записи поставщиков имеют только четыре поля, поэтому "пятое" поле замыкает цикл и становится первым. Таким образом, значение "следующего" поля (первого поля, или поля S#) в реконструируемой записи поставщика находится в позиции S# первой строки таблицы значений полей, иными словами, в ячейке [1,1]. Но с этого значения и началось считывание записи, поэтому весь процесс останавливается.

Безусловно, что приведенная выше последовательность операций приводит к реконструкции одной конкретной записи из файла поставщиков, а именно той записи, которая показана под номером 4 на рис. А.3:

	S#	SNAME	STATUS	CITY
4	S1	Smith	20	London

Кстати, следует отметить, что указатели номеров строк, по которым мы переходили в приведенном выше примере, образуют *кольцо,* а фактически образуют два изоморфных кольца — одно из них находится в таблице значений полей, а другое — в таблице реконструкции записей (рис. А.7).

Примечание. По очевидным причинам эти кольца часто называют также *зигзагами,* а сам алгоритм реконструкции носит неформальное название *алгоритма зигзага.*

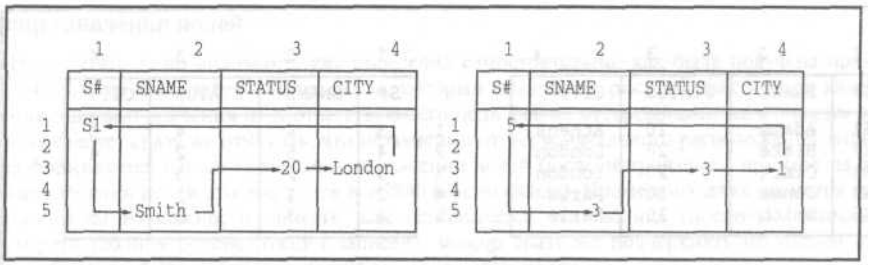


Рис. А.7. Кольца указателей (примеры)

В качестве упражнения попытайтесь сами реконструировать еще одну запись с данными поставщика. Если вы начнете с ячейки [2,1] таблицы значений полей, то получите запись 3 файла, показанного на рис. А.3. Аналогичным образом, начав с ячейки [3,1], можно получить запись 5, начав с ячейки [4,1] — запись 1, а начав с ячейки [5,1] — запись 2. Здесь заслуживает внимания то, в чем состоит суммарный эффект: если обработка всей таблицы значений полей осуществляется в последовательности номеров поставщиков путем прохождения сверху вниз по столбцу S# (т.е. если процесс реконструкции записи выполняется пять раз, начиная последовательно с ячеек [1,1], [2,1], [3,1], [4,1] и [5,1]), то происходит реконструкция той версии всего файла поставщиков, в которой записи присутствуют в порядке возрастания номеров поставщиков. Иными словами, при этом реализуется следующий запрос SQL.

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY St ;
```

Аналогичным образом, для реализации приведенного ниже запроса достаточно выполнить обработку столбца таблицы значений полей с номерами поставщиков в обратном порядке и провести операции реконструкции, начиная с ячейки [5,1], затем [4,1] и т.д.

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
ORDER BY Sf DESC ;
```

Причем и в том, и в ином случае не требуется ни выполнять сортировку на этапе прогона, ни использовать индекс.

Более того, именно в связи с тем, что указатели в таблице реконструкции записей образуют кольца, можно входить в эти кольца в любой точке. Поэтому процесс применения алгоритма реконструкции можно начинать с любой требуемой ячейки. Например, если начать с ячейки [1,3] (т.е. с первой ячейки в столбце STATUS), то будет получена следующая запись.

	S#	SNAME	STATUS	CITY
3	S2	Jones	10	Paris

(Точнее, будет получена та версия этой записи, в которой упорядочением полей слева направо является STATUS, затем CITY, затем S#, затем SNAME.) Следуя вниз по столбцу STATUS (т.е. последовательно продолжая процесс реконструкции с применением ячеек [2,3], [3,3], [4,3] и [5,3]) можно в конечном итоге получить весь файл поставщиков в порядке возрастания статуса.

```
SELECT S.STATUS, S.CITY, S.S#, S.SNAME
FROM S
ORDER BY STATUS ;
```

Аналогичным образом, обработка таблицы реконструкции записей в порядке значений указателей в столбце SNAME позволяет получить файл поставщиков, отсортированный по возрастанию лексикографических значений имен поставщиков, а обработка этой таблицы в порядке значений указателей в столбце CITY — файл, отсортированный по возрастанию лексикографических значений названий

городов. Иными словами, при совместном использовании таблица реконструкции записей и таблица значений полей служат одновременно представителями всех этих способов упорядочения; при этом (еще раз повторяем) не возникает необходимость ни в использовании индексов, ни в выполнении сортировки на этапе прогона.

Теперь рассмотрим следующий запрос, который требует выполнения операции сокращения с проверкой на равенство.

```
SELECT S.S#, S.SNAME, S.STATUS, S.CITY
FROM S
WHERE S.CITY = 'London' ;
```

Поскольку столбец CITY (как и все столбцы) в таблице значений полей хранится в отсортированном порядке, то для поиска ячеек, содержащих значение London, может применяться бинарный поиск. Если рассматривается таблица значений полей, приведенная на рис. А.6, то такими ячейками являются [2,4] и [3,4]. Теперь можно сформировать зигзаги, следуя по кольцам указателей, проходящим через ячейки [2,4] и [3,4] таблицы реконструкции записей. В данном примере указанные зигзаги выглядят следующим образом.

```
[2,4], [4,1], [3,2], [2,3] И
[3,4], [1,1], [5,2], [3,3]
```

Наложив эти зигзаги на таблицу значений полей, получим следующие значения полей для требуемых записей.

	S#	SNAME	STATUS	CITY
1	S4	Clark	20	London
4	S1	Smith	20	London

При совместном использовании таблица значений полей и таблица реконструкции записей предоставляют также непосредственную поддержку многих других операций пользовательского уровня, а не только поддержку простых запросов с конструкциями ORDER BY и операциями сокращения с проверкой на равенство. В действительности, основная часть или даже все фундаментальные реляционные операции (сокращение, проекция, соединение, агрегирование и др., не говоря уже об операции удаления дубликатов, необходимость в использовании которой на промежуточных этапах возникает всегда, даже в истинно реляционных системах) имеют алгоритмы реализации, основанные на способности получать доступ к данным в некоторой конкретной последовательности. В качестве примера рассмотрим операцию соединения. В главе 18 было сказано, что удобным способом реализации соединения является сортировка—слияние. А модель TR позволяет выполнять соединения по методу сортировки-слияния без необходимости выполнения самой сортировки! (Или, по меньшей мере, без необходимости выполнять сортировку на этапе прогона, поскольку сортировка осуществляется при формировании таблиц значений полей и реконструкции записей, иными словами, неформально можно утверждать, что она выполняется на этапе загрузки данных.) Например, для соединения отношений поставщиков и деталей по названиям городов достаточно просто обратиться к каждой из двух соответствующих таблиц значений полей в порядке названий городов и выполнить соединение в стиле операции слияния.

Одним важным следствием из всего сказанного является то, что работа оптимизатора системы намного упрощается, а именно, гораздо проще становится процесс выбора пути доступа (см. главу 18), а в некоторых случаях необходимость в его использовании полностью исключается. Еще одним следствием является то, что становятся ненужными также многие из вспомогательных структур (индексы, хэши и т.д.), применяемых в традиционных СУБД. Наконец, важным следствием становится то, что физическое проектирование базы данных значительно упрощается в связи с тем, что количество различных опций и вариантов выбора структур данных становится намного меньше; то же самое касается и настройки производительности.

Формирование таблицы реконструкции записей

Рассмотрим результаты применения различных вариантов упорядочения с помощью сортировки к файлу, показанному на рис. А.3. Например, предположим, что была выполнена его сортировка в порядке возрастания номеров поставщиков, а затем получены записи в последовательности 4, 3, 5, 1, 2. Назовем эту последовательность "перестановкой, соответствующей упорядочению с возрастанием атрибута S#" (или сокращенно *перестановкой* S#). Другие варианты перестановки показаны ниже.

- По возрастанию значения SNAME: 2,5,1,3,4.
- По возрастанию значения STATUS: 3, 1,4,2,5.
- По возрастанию значения CITY: 2, 1,4, 3, 5.

Итоговые данные по этим перестановкам приведены в следующей таблице перестановок, в которой ячейка [i, j] содержит номер в файле поставщиков той записи, которая присутствует в i-й позиции, если файл отсортирован по возрастанию значений в j-м поле.

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	2	3	2
2	3	5	1	1
3	5	1	4	4
4	1	3	2	3
5	2	4	5	5

Как показывает эта таблица, перестановка S# (еще раз повторяем) представляет собой следующую последовательность. 4, 3, 5, 1, 2 Обратной по отношению к этой перестановке является следующая последовательность.

4, 5, 2, 1, 3

Эта обратная перестановка является такой уникальной перестановкой, которая после ее применения к первоначальной последовательности 4, 3, 5, 1,2 приводит к получению последовательности 1, 2, 3, 4, 5. (Если SEQ — первоначальная последовательность 4, 3, 5, 1, 2, то четвертым элементом в последовательности SEQ является элемент 1, пятым— 2, вторым— 3 и т.д.) Вообще говоря, если рассматривать любую заданную перестановку как вектор V, то обратная перестановка V может быть получена в соответствии с простым правилом, что если $V[i) = i^1$, то $V[i^1] = i$. Применяя это правило к каждой из перестановок в

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	4	3	2	2
2	5	1	4	1
3	2	4	1	4
4	1	5	3	3
5	3	2	5	5

[6] Теперь можно приступить к формированию таблицы реконструкции записей. Например, первый столбец (s#) можно сформировать, как описано ниже.
приведенной выше таблице перестановок, получим следующую таблицу обратных перестановок.

Перейти в ячейку [i,1] таблицы обратных перестановок. Пусть эта ячейка содержит значение r; кроме того, допустим, что следующая ячейка справа (ячейка [i,2]) содержит значение r'. Перейти в r-ю строку таблицы реконструкции записей и поместить значение r' в ячейку [r,1].

Выполнение этого алгоритма для i = 1, 2, ..., 5 позволяет получить весь столбец S# таблицы реконструкции записей. Другие столбцы формируются аналогичным образом.

Примечание. Настоятельно рекомендуем читателю в качестве упражнения проработать весь этот алгоритм и полностью сформировать таблицу реконструкции записей. Выполнение такого упражнения позволяет получить ключ к пониманию работы алгоритма. Кстати, следует отметить, что таблица реконструкции записей формируется исключительно на основании файла, поскольку в этом процессе таблица значений полей вообще не играет никакой роли.

Неуникальная таблица реконструкции записей

В предыдущем подразделе (кроме всего прочего) было сказано, что перестановка CITY для файла поставщиков, приведенного на рис. А.3, имела последовательность 2, 1, 4, 3, 5. Но из того, что и поставщик S1, и поставщик S4 находятся в одном и том же городе, так же, как и поставщики S2 и S3, следует, что можно было бы с такой же уверенностью утверждать, что перестановкой по атрибуту CITY была последовательность 2,4,1,3,5, или3, 1, 4, 2, 5, или 3, 4, 1, 2, 5. Иными словами, перестановка CITY не является уникальной[2]. Из этого следует, что таблица перестановок не уникальна и поэтому также не уникальна и таблица реконструкции записей. Но оказалось, что применительно к каждому конкретному отношению пользовательского уровня всегда имеются определенные таблицы реконструкции записей, являющиеся "предпочтительными", в том смысле, что они обладают некоторыми желательными свойствами, которых другие таблицы реконструкции записей не имеют. Но подробные сведения об этих особенностях выходят за рамки настоящего приложения; дополнительная информация приведена в [А. 1].

А.4. СЖАТЫЕ СТОЛБЦЫ

Рассмотрим рис. А.8, на котором показан возможный вариант файла, соответствующего обычно применяемому в этой книге отношению деталей, рис. А.9, на котором приведена соответствующая таблица значений полей, и рис. А.10, где показана соответствующая "предпочтительная" таблица реконструкции записей.

Теперь следует отметить, что таблица значений полей, показанная на рис. А.9, содержит значитель ный объем *избыточной информации,* например, название города London появляется в ней три раза, вес 17.0 — два раза и т.д. Сжатие столбцов этой таблицы позволяет просто устранить указанную избыточность. Поэтому результат становится таким, что каждый столбец содержит только соответствующие различимые значения, как показано на рис. А. 11. Из сказанного следуют приведенные ниже выводы.

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Nut	Red	12.0	London
2	P2	Bolt	Green	17.0	Paris
3	P3	Screw	Blue	17.0	Oslo
4	P4	Screw	Red	14.0	London
5	P5	Cam	Blue	12.0	Paris
6	P6	Cog	Red	19.0	London

Рис. А.8. Вариант файла для обычно применяемого отношения деталей

[2] Аналогичное утверждение справедливо и по отношению к перестановке STATUS, но, как оказалось, не относится к перестановке SNAME (а также, безусловно, к перестановке S#).

Рис. А. 11. Сжатая версия таблицы значений полей, соответствующая таблице,

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Bolt	Blue	12.0	London
2	P2	Cam	Blue	12.0	London
3	P3	Cog	Green	14.0	London
4	P4	Nut	Red	17.0	Oslo
5	P5	Screw	Red	17.0	Paris
6	P6	Screw	Red	19.0	Paris

Рис. А.9. Таблица значений полей, соответствующая файлу на рис. А.8

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	4	3	2	1	1
2	1	1	4	6	4
3	5	6	5	2	6
4	6	4	1	4	3
5	2	2	3	5	2
6	3	5	6	3	5

Рис. А.10. Таблица реконструкции записей, соответствующая файлу на рис. А.8

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Bolt	Blue	12.0	London
2	P2	Cam	Green	14.0	Oslo
3	P3	Cog	Red	17.0	Paris
4	P4	Nut		19.0	
5	P5	Screw			
6	P6				

показанной на рис. А.9

1. Допустимо и даже желательно предусмотреть *избирательное применение* процесса сжатия. В част ности, нет смысла выполнять сжатие столбца p#, поскольку номера деталей являются уникаль ными
2. Значения полей в сжатых столбцах фактически используются *совместно* в разных записях файла деталей. Например, имя города London в ячейке [1,5] используется в трех записях с данными о деталях, а именно с данными о деталях P1, P4 и P6. Одним из следствий этого является возмож ность быстрее выполнять операции обновления, особенно INSERT, поскольку в них можно ис пользовать уже существующие значения полей, фактически обеспечивая при этом совместное применение этих значений с другими записями; например, достаточно представить себе, что произойдет, если пользователь вставит кортеж с данными о детали P7, имеющей название Nut, цвет Red, вес 18 . О, город London. Но, как было указано в разделе АЛ, в целом операции обнов ления выходят за рамки данного приложения.
3. Сжатые столбцы, безусловно, становятся своего рода основой метода сжатия данных (причем та кого метода, который, как правило, не встречается в обычных реализациях с непосредственным

отображением), но заслуживает внимания то, насколько *эффективное* сжатие они позволяют обеспечить. Например, предположим, что в Министерстве автотранспорта штата используется отношение с данными о водительских правах, где находится по одному кортежу с данными о правах каждого водителя, выданных (скажем) в штате Калифорния, причем общее количество таких кортежей составляет около 20 млн. Но, безусловно, количество различных значений веса, роста, цвета волос водителей, а также значений срока действия этих прав не составляет 20 млн! Иными словами, коэффициент сжатия иногда может достигать порядка одного к миллиону **или** примерно такого значения.

Диапазоны строк

Вернемся к анализу рис. А. 11. Вне всякого сомнения, нельзя просто заменить (например) три первоначальных вхождений названия города London только одним таким вхождением, поскольку подобное решение привело бы к потере информации. (Сжатый столбец CITY содержит три значения, а количество деталей равно шести. Как в таком случае узнать, в каком городе находится та или иная деталь?) Поэтому необходимо хранить некоторую дополнительную информацию, которая, по существу, позволяет реконструировать первоначальную, несжатую таблицу значений полей из ее сжатого аналога. Один из способов решения этой задачи состоит в том, что наряду с каждым значением поля в каждом сжатом столбце таблицы значений полей необходимо хранить данные о диапазоне номеров строк[3] в несжатой версии этой таблицы, в которых первоначально находилось это значение, как показано на рис. А.12. Например, рассмотрим ячейку [3,4] на этом рисунке, которая содержит значение веса 17. 0. Рядом с этим значением веса показан диапазон строк "[4:5]". Этот диапазон строк означает, что если потребуется отменить "сжатие" таблицы значений полей, т.е. восстановить ее в первоначальном виде, то значение веса 17.0 появится (разумеется, в столбце WEIGHT, т.е. в столбце 4) в строках 4 и 5, включительно, как в той таблице, какой она была до сжатия.

	1	2	3	4	5
	P#	PNAME	COLOR	WEIGHT	CITY
1	P1	Bolt [1:1]	Blue [1:2]	12.0 [1:2]	London [1:3]
2	P2	Cam [2:2]	Green [3:3]	14.0 [3:3]	Oslo [4:4]
3	P3	Cog [3:3]	Red [4:6]	17.0 [4:5]	Paris [5:6]
4	P4	Nut [4:4]		19.0 [6:6]	
5	P5	Screw [5:6]			
6	P6				

Рис. А.12. Сжатая таблица значений полей с указанием диапазонов строк

Кстати, не следует путать спецификацию в форме [4:5] со спецификацией [4,5]. Первая из них (с разделителем в виде двоеточия) обозначает определенный диапазон номеров строк, как было описано выше, а последняя (с разделителем в виде запятой) — это индекс, обозначающий конкретную ячейку, которая находится на пересечении указанных в этом индексе строки и столбца.

В отношении диапазонов номеров строк необходимо сделать еще одно замечание. Рассмотрим (например) столбец 3 (столбец COLOR) в таблице значений полей, приведенной на рис. А.12. Очевидно, что в этом столбце указано именно следующее: во-первых, множество значений COLOR, которые в настоящее время присутствуют в файле деталей, и наряду с этим, во-вторых, для каждого такого

[3] Автор намеренно выбрал для обозначения диапазонов строк на этом рисунке такой же формат, как и для интервалов в главе 23, но нельзя отрицать, что информация о диапазоне строк может быть физически представлена с помощью многих других, отличных от этого способов. Один из способов может предусматривать хранение информации только о начале или только о конце диапазона. Еще один способ может состоять в том, что вместо обозначений начала и конца диапазона может быть указано количество строк в диапазоне. Существуют и другие способы.

значения — количество случаев появления этого значения в заданном файле. Иными словами, данный столбец может рассматриваться как **гистограмма** (рис. А. 13). **В** действительности, вообще говоря, вся сжатая таблица значений полей со своими диапазонами номеров строк может с очень большой пользой для дела рассматриваться как множество гистограмм, по одной для каждого сжатого столбца. Одним из следствий из указанного факта является то, что должна быть весьма высокой производительность тех запросов, которые концептуально требуют применения подобных гистограмм (таковым является, например, запрос: "Сколько имеется деталей каждого цвета?"). См. раздел А.6.

Приведем еще одно короткое замечание на ту же тему: если таблица значений полей может рассматриваться как множество гистограмм, то и таблица реконструкции записей (как уже было сказано в предыдущем разделе) может фактически рассматриваться как множество **перестановок.** Например, после реконструкции файла деталей с использованием столбца 3 таблицы реконструкции записей, показанной на рис. АЛО, будет получена версия файла, отсортированная по цвету деталей; иными словами, будет получено то, что можно было бы назвать "перестановкой COLOR" этого файла. Поэтому представление **с** помощью модели TR любого конкретного набора данных можно неформально охарактеризовать как множество гистограмм в сочетании с множеством перестановок (рассматриваемых данных). Такие гистограммы и перестановки, по сути, составляют все, что лежит в основе представления данных **в** модели TR.

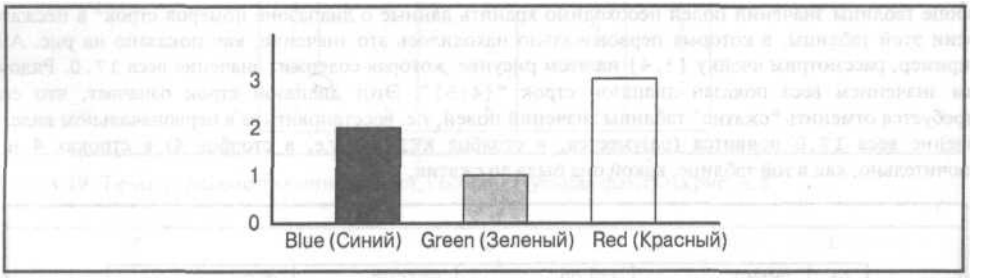


Рис. А. 13. Гистограмма цветов (основанная на данных рис. А. 12)

Применение сжатых таблиц в процессе реконструкции записей

Безусловно, что при сжатии таблицы значений полей информация о взаимно однозначном соответствии между ячейками этой таблицы и ячейками таблицы реконструкции записей уничтожается. Из этого следует, что применявшийся до сих пор алгоритм реконструкции записей перестает действовать. Тем не менее, эту проблему можно решить довольно просто, как описано ниже.

Рассмотрим ячейку `[i,j]` таблицы реконструкции записей. Вместо перехода в ячейку `[i,j]` таблицы значений полей следует перейти в ячейку `[i',j]` этой таблицы, где ячейка `[i',j]` является уникальной ячейкой в столбце `j` данной таблицы, которая содержит диапазон строк, включающий строку `i`.

В качестве примера рассмотрим ячейку [3,4] таблицы реконструкции записей, приведенной на рис. АЛО, которая присутствует (безусловно) в столбце 4 (столбце WEIGHT) этой таблицы. Чтобы найти соответствующее значение веса в таблице значений полей, показанной на рис. А.11, выполним поиск в столбце WEIGHT этой таблицы и найдем в этом столбце уникальную запись, содержащую диапазон строк, в который входит строка 3. Данный рисунок показывает, что рассматриваемым элементом является ячейка [2,4] (притом что соответствующим диапазоном строк является [3:3]), а требуемое значение веса равно 14 . 0. В качестве упражнения воспользуйтесь приведенной на рис. АЛО таблицей реконструкции записей наряду со сжатой таблицей значений полей на рис. А. 12 для полной реконструкции файла деталей (начните со столбца 5, чтобы получить результат по возрастанию последовательности лексикографических значений названий городов).

А.5. СЛИВШИЕСЯ СТОЛБЦЫ

В предыдущем разделе рассматривались сжатые столбцы, которые могут использоваться в качестве способа совместного использования значений полей в нескольких записях, но все рассматриваемые записи находятся в одном файле. В отличие от этого, **слившиеся** столбцы могут применяться в качестве способа совместного использования значений полей в нескольких записях, в котором все рассматриваемые записи могут находиться в одном или в разных файлах. Основная идея этого способа состоит в том, что разные поля на файловом уровне могут соответствовать одному и тому же столбцу таблицы значений полей на уровне TR, разумеется, при условии, что все рассматриваемые поля имеют один и тот же тип данных.

Начнем с примера, в котором применяется только один файл. Рассмотрим отношение спецификации материалов MMQ, показанное на рис. А. 14 (это — вариант рис. 4.6 из главы 4). Вначале определим, что произойдет в этом примере без слившихся столбцов, затем рассмотрим, как изменится ситуация после перехода к использованию метода слияния столбцов. Таким образом, на рис. А.15 показан возможный вариант файла, соответствующий отношению на рис. А. 14, на рис. А. 16 приведена соответствующая сжатая таблица значений полей, а на рис. А. 17— соответствующая "предпочтительная" таблица реконструкции

MMQ

MAJOR_P#	MINOR_P#	QTY
P1	P2	2
P1	P3	4
P1	P4	1
P2	P3	3
P2	P4	8
P2	P5	6
P3	P4	3
P3	P6	4
P5	P6	3

Рис. А. 14. Отношение спецификации материалов MMQ

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	P3	P4	3
2	P1	P3	4
3	P2	P4	8
4	P1	P4	1
5	P2	P5	6
6	P3	P6	4
7	P1	P2	2
8	P5	P6	3
9	P2	P3	3

записей.

Рис. А.15. Файл для отношения спецификации материалов, показанного на рис. А. 14

Теперь перейдем к изучению способа, в котором применяются слившиеся столбцы. Снова вернувшись к анализу отношения MMQ (рис. А.14), можно убедиться в том, что атрибуты MAJOR_P# и MINOR_P# относятся к одному и тому же типу; это означает, что поля MAJOR P# и MINOR P# соответствующего файла также относятся к одному и тому же типу. Поэтому их можно поставить в соответствие одному и тому же столбцу таблицы значений полей. На рис. А. 18 показано, что при этом происходит. Из данного описания следует приведенные ниже выводы.

1. Выполнено слияние столбцов MAJOR_P# и MINOR_P# в один столбец. Этот столбец содержит все значения полей (т.е. номера деталей), которые прежде находились либо в столбце MAJOR_P#, либо в столбце MINOR_P# той таблицы, какой она была до слияния. После слияния дубликаты были удалены.

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	P1 [1:3]	P2 [1:1]	1 [1:1]
2	P2 [4:6]	P3 [2:3]	2 [2:2]
3	P3 [7:8]	P4 [4:6]	3 [3:5]
4	P5 [9:9]	P5 [7:7]	4 [6:7]
5		P6 [8:9]	6 [8:8]
6			8 [9:9]

Рис. А. 16. Сжатая таблица значений полей для файла, приведенного на рис. А. 15

	1	2	3
	MAJOR_P#	MINOR_P#	QTY
1	1	2	3
2	2	6	1
3	4	3	4
4	3	1	7
5	5	9	9
6	7	4	2
7	6	8	8
8	8	7	6
9	9	5	5

Рис. А. 17. Таблица реконструкции записей для файла, приведенного на рис. А. 15

	1	2
	MAJOR_P# + MINOR_P#	QTY
1	P1 [1:3] [:]	1 [1:1]
2	P2 [4:6] [1:1]	2 [2:2]
3	P3 [7:8] [2:3]	3 [3:5]
4	P4 [:] [4:6]	4 [6:7]
5	P5 [9:9] [7:7]	6 [8:8]
6	P6 [:] [8:9]	8 [9:9]

Рис. А.18. Приведенная на рис. А. 16 таблица значений полей после слияния первых двух столбцов

2. Поэтому каждая ячейка в слившемся столбце содержит единственный номер детали вместе с двумя диапазонами строк. Первый диапазон строк показывает, в каких строках несжатой таблицы значений полей соответствующий номер детали присутствует в качестве номера "основной" детали, а второй диапазон показывает, в каких строках этой несжатой таблицы значений полей соответствующий номер детали присутствует в качестве номера "дополнительной" детали. Эти диапазоны строк являются по сути теми же, как и в предыдущей версии таблицы значений полей, если не считать того, что среди них иногда появляется специальный пустой диапазон строк "[:]", который используется, если указанный номер детали вообще не присутствует в

соответствующем столбце несжатой таблицы значений полей (например, pi никогда не появляется в качества номера "дополнительной" детали).

3. В этой слившейся таблице слившимся столбцом является столбец 1, а столбец QTY становится столбцом 2 (теперь в конечном итоге эта таблица имеет только два стол-бца, а не три). Столбец 2 (столбец QTY) является таким же, каким он был на рис. А. 16.

4. Таблица реконструкции записей остается неизменной[4]. Но теперь столбцы 1 и 2 этой таблицы оба соответствуют столбцу 1 (слившемуся столбцу) таблицы значений полей; столбец 1 относится к первому диапазону строк в этом слившемся столбце, а столбец 2 — ко второму. Столбец 3 таблицы реконструкции записей теперь относится к столбцу 2 таблицы значений полей.

Эта идея слияния столбцов позволяет достичь многих преимуществ; но в данном разделе будет описано только одно из них. Оно касается операций соединения. Предположим, что необходимо соединить отношение MMQ с самим собой, сопоставляя номера дополнительных деталей из "первой копии" (в том виде, какая она есть) данного отношения с номерами основных деталей из "второй копии". В таком случае достаточно выполнить единственный проход по слившейся таблице значений полей, чтобы узнать, какие кортежи с какими соединяются! Например, строка 3 этой таблицы (которая содержит номер детали РЗ) показывает "диапазон строк с номерами дополнительных деталей", равный [2 :31, а "с номерами основных деталей " — равный [7 : 8]. Из этого непосредственно следует, что и второй, и третий кортежи из "первой копии" отношения MMQ оба соединяются к седьмым и восьмым кортежами из "второй копии". И аналогичные замечания относятся ко всем остальным строкам этой слившейся таблицы значений полей. Поэтому фактически соединение по методу сортировки—слияния может быть получено без выполнения[5] сортировки (как было указано выше в данном приложении) и даже вообще без слияния!

Примечание. Безусловно, ни в одном отношении нет такого понятия, как "второй" кортеж, "третий" кортеж или "i-й" кортеж для любого значения i; кортежи в отношении не упорядочены. Под этой сомнительной формулировкой подразумеваются приведенные ниже соглашения.

■ Допустим, что F1 — файл, реконструированный из таблицы значений полей, приведенной на рис. А.18, путем обработки столбца MAJOR_P# таблицы реконструкции записей на рис. А.17 в последовательности сверху вниз. Тогда "первой копией" отношения MMQ является файл Fl, а "i-м кортежем" этой копии — уникальный кортеж отношения MMQ, который соответствует i-й записи файла F1.

■ Аналогичным образом, допустим, что F2 — это файл, реконструированный из таблицы значений полей, приведенной на рис. А.18, путем обработки столбца MINOR_P# таблицы реконструкции записей на рис. А.17 в последовательности сверху вниз. Тогда "второй копией" отношения MMQ является файл F2, а "i-м кортежем" этой копии является уникальный кортеж отношения MMQ, который соответствует i-й записи файла F2.

В завершение данного раздела еще раз подчеркнем то, что метод слияния столбцов может применяться не только к одному, но и к нескольким файлам. Например, в случае отношения поставщиков и деталей можно было бы предусмотреть только одну таблицу значений полей (подвергнутую и слиянию, и сжатию) для всей базы данных, и в этой таблице предусмотреть один столбец для номеров поставщиков (из переменных отношения S и SP), один для номеров деталей (из переменных отношения SP и P), один для названий городов (из переменных отношения S и P) и т.д. В действительности, модель TR позволяет включать в таблицу значений полей такие значения, которые в данный момент фактически не находятся ни в одном из отношений базы данных, поэтому TR может рассматриваться в полном смысле этого слова как "доменно-ориентированное" представление всей базы данных. Дополнительная информация на эту тему приведена в[А.1].

---

[4] Эта таблица фактически может быть усовершенствована многими способами, но подробные сведе ния об этом выходят за рамки данного приложения.

[5] Точнее, эти операции сортировки и слияния не осуществляются на этапе прогона; вместо этого они выполняются заблаговременно, при формировании таблиц значений полей и таблиц реконструкции записей (по сути, на этапе загрузки данных).

А.6. РЕАЛИЗАЦИЯ РЕЛЯЦИОННЫХ ОПЕРАТОРОВ

В данном разделе кратко описано, какие следствия связаны с применением модели TR для реализации некоторых реляционных операторов. Соответствующие примеры основаны на переменных отношения S и SPJ из базы данных поставщиков, деталей и проектов (примеры значений показаны на рис. А. 19). Полученная после слияния и сжатия таблица значений полей приведена на рис. А.20, а "предпочтительные" таблицы реконструкции записей — на рис. А.21.

S

S#	SNAME	STATUS	CITY
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

SPJ

S#	P#	J#	QTY
S1	P1	J1	200
S1	P3	J2	100
S2	P1	J1	200
S2	P1	J2	500
S2	P2	J2	500
S3	P1	J1	100
S3	P2	J2	500
S3	P3	J1	200
S3	P3	J2	200

Рис. А. 19. Переменные отношения S и SPJ (примеры значений)

	1	2	3	4
	S#	SNAME	STATUS	CITY
1	S1 [1:2]	Adams [1:1]	10 [1:1]	Athens [1:1]
2	S2 [3:5]	Blake [2:2]	20 [2:3]	London [2:3]
3	S3 [6:9]	Clark [3:3]	30 [4:5]	Paris [4:5]
4	S4 [:]	Jones [4:4]		
5	S5 [:]	Smith [5:5]		

	5	6	7
	P#	J#	QTY
	P1 [1:4]	J1 [1:4]	100 [1:2]
	P2 [5:6]	J2 [5:9]	200 [3:6]
	P3 [7:9]		500 [7:9]

Рис. А.20. Слившаяся таблица значений полей для поставщиков и отгрузок

Сокращение

Рассмотрим следующий запрос на выполнение операции сокращения[6].

SPJ WHERE QTY = 200

Для реализации этого запроса необходимо выполнить бинарный поиск в столбце QTY таблицы значений полей (см. рис. А.20) и найти ячейку, содержащую значение 200; следует отметить, что такая ячейка должна быть уникальной (если она вообще существует), поскольку данный столбец — сжатый. Если поиск оканчивается неудачей, это позволяет сразу же определить, что результат запроса пуст. Но в

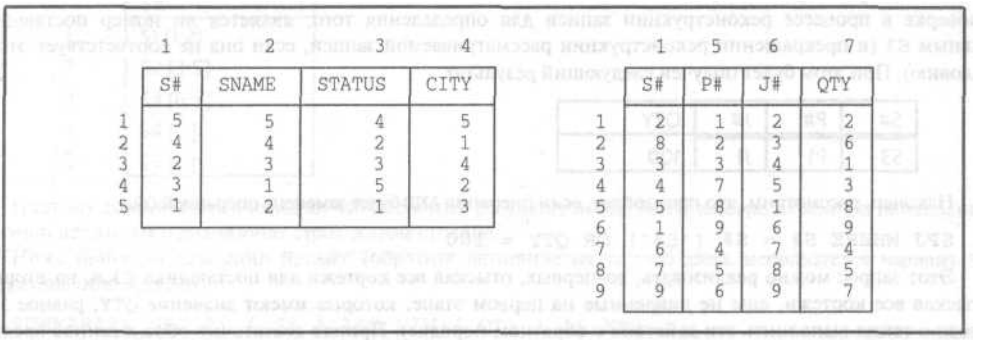
[6] В качестве основы для всех остальных примеров в этом приложении используется язык Tutorial D, а не SQL.

рассматриваемом случае поиск приводит к обнаружению ячейки [2,7], которая, кроме заданного значения QTY, содержит диапазон строки [3:6]. На основании этого можно сразу же сделать вывод, что ячейки [3,7], [4,7], [5,7] и [6,7] таблицы реконструкции записей поставок соответствуют описанным ниже требованиям.

а) Содержат номера строк для ячеек таблицы значений полей, содержащих значение QTY, равное 200
(и, безусловно, все они включают номер строки 2).

б) Содержат номера строк для "следующей" ячейки в таблице реконструкции записей поставок.

Поэтому зигзаги могут быть сформированы путем прохождения по соответствующим кольцам указателей в таблице реконструкции записей поставок. В данном примере эти зигзаги выглядят следующим образом.



	1	2	3	4
	S#	SNAME	STATUS	CITY
1	5	5	4	5
2	4	4	2	1
3	2	3	3	4
4	3	1	5	2
5	1	2	1	3

	1	5	6	7
	S#	P#	J#	QTY
1	2	1	2	2
2	8	2	3	6
3	3	3	4	1
4	4	7	5	3
5	5	8	1	8
6	1	9	6	9
7	6	4	7	4
8	7	5	8	5
9	9	6	9	7

Рис. А.21. Таблицы реконструкции записей для поставщиков и отгрузок

[3,7], [1,1], [2,5], [2,6]
[4,7], [3,1], [3,5], [3,6]
[5,7], [8,1], [7,5], [4,6]
[6,7], [9,1], [9,5], [6,6]

Проходя по этим зигзагам через таблицу реконструкции записей поставок и обращаясь соответствующим образом к таблице значений полей, получим показанный ниже требуемый результат.

S#	P#	J#	QTY
S1	P1	J1	200
S2	P1	J1	200
S3	P3	J1	200
S3	P3	J2	200

В качестве второго примера рассмотрим запрос, в котором используется сокращение с помощью оператора "<", а не с помощью оператора "=".

SPJ WHERE QTY < 150

Вполне очевидно, что этот запрос также может быть выполнен очень просто, но на этот раз с помощью описанной ниже процедуры.

а) Выполнить последовательный поиск в столбце QTY таблицы значений полей.

б) Реконструировать все соответствующие записи, следовательно, кортежи пользовательского уровня для каждой ячейки, обнаруженной в процессе этого поиска.

в) Остановиться, как только в столбце QTY таблицы значений полей будет найдена ячейка, содер жащая значение QTY, большее или равное 150.

Теперь рассмотрим следующий пример.

При этом будет получен следующий результат.

S#	P#	J#	QTY
S1	P3	J2	100
S3	P1	J1	100

SPJ WHERE S# = S# ('S3') AND QTY = 100

Выполняя поиск в столбцах S# и QTY таблицы значений полей, можно обнаружить из соответствующих диапазонов строк, что есть четыре поставки с номером поставщика S3, но только две с количеством 100. Поэтому наилучшая стратегия состоит в использовании зигзагов, связанных с количеством 100, и проверке в процессе реконструкции записи для определения того, является ли номер поставщика равным S3 (и прекращении реконструкции рассматриваемой записи, если она не соответствует этому условию). При этом будет получен следующий результат.

S#	P#	J#	QTY
S3	P1	J1	100

Наконец, рассмотрим, что произойдет, если операция AND будет заменена операцией OR.

SPJ WHERE S# = S# ('S3') OR QTY = 100

Этот запрос можно реализовать, во-первых, отыскав все кортежи для поставщика S3 и, во-вторых, отыскав все кортежи, еще не найденные на первом этапе, которые имеют значение QTY, равное 100 (можно также выполнить эти действия в обратном порядке). Приняв достаточно обоснованное предположение, будто эти два описанных выше этапа выполняются таким образом, что происходит определенное упорядочение двух вырабатываемых результатов (скажем, в порядке возрастания значений S#), можно сделать вывод о том, что их слияние позволяет получить общий желаемый результат.

S#	P#	J#	QTY
S1	P3	J2	100
S3	P1	J1	100
S3	P2	J2	500
S3	P3	J1	200
S3	P3	J2	200

Проекция

Чтобы вычислить (скажем) проекцию SPJ{S#, P#, J#}, достаточно пройти просто через обычный процесс реконструкции для поставок, но пропустить этап реконструкции атрибута QTY в каждой записи. Но чтобы вычислить (скажем) проекцию SPJ{S#, P#}, которая, в отличие от предыдущего примера, требует устранения некоторых дубликатов, желательно обработать таблицу реконструкции записей для поставок в такой последовательности, которая позволяет получать кортежи в соответствии с упорядочением от основной детали к дополнительной "S#, затем P#" (или "P#, затем S#"); при таком упорядочении дубликаты станут смежными и поэтому могут быть легко удалены. Здесь подробные сведения об этом не приведены, но отметим лишь то, что "предпочтительные" таблицы реконструкции записей являются таковыми именно потому, что они поддерживают указанные варианты упорядочения.

Агрегирование

Рассмотрим следующий запрос.

SUMMARIZE SPJ PER S { S# } ADD COUNT AS SHIP_COUNT

Поэтому должно быть очевидно, что желаемый результат может быть непосредственно и немедленно

Ниже приведен полученный при этом результат.

S#	SHIP_COUNT
S1	2
S2	3
S3	4
S4	0
S5	0

Итак, столбец S# таблицы значений полей выглядит следующим образом (см. рис. А.20).

	S#
1	S1 [1:2]
2	S2 [3:5]
3	S3 [6:9]
4	S4 [:]
5	S5 [:]

получен из данных о диапазонах строк в этом столбце.

Ниже приведен еще один пример (обратите внимание на то, что здесь используется вариант BY оператора SUMMARIZE).

```
SUMMARIZE SPJ BY { S# } ADD MIN ( QTY ) AS MNQ
```

Для того чтобы определить, как может быть реализован этот запрос, рассмотрим данные о поставщике S2. Из диапазона строк [3:5] для этого поставщика в таблице значений полей (см. рис. А.20) можно определить, что строками таблицы реконструкции записей поставок, которые относятся к этому поставщику, являются строки 3, 4 и 5 (это означает, что применимыми ячейками этой таблицы являются, соответственно, [3,1], [4,1] и [5,1]. Проходя по зигзагам к соответствующим ячейкам QTY в этой таблице, можно определить, что эти ячейки, соответственно, содержат указатели на строки 2, 3 и 3 таблицы значений полей. Поскольку столбец QTY (как и все столбцы) в этой таблице отсортирован в порядке возрастания, становится сразу же очевидно, что минимальным значением QTY для поставщика S2 является значение, находящееся в строке 2 таблицы значений полей, а именно значение QTY, равное 200.

Соединение

В предыдущих разделах уже было показано, что требуется для реализации соединений. Поэтому ниже просто указано несколько дополнительных требований.

- Поскольку в модели TR всегда выполняется соединение по методу сортировки—слияния, а операции сортировки и слияния выполняются заблаговременно, то на этапе прогона затраты на выполнение операции соединения имеют аддитивный (линейный), а не мультипликативный характер. В [АЛ] приведен пример, который предусматривает выполнения соединения пяти отношений. В этом примере в реализации TR на выполнение этой операции потребовалось 5 с, а в реализации с последовательным просмотром (см. главу 18) потребовалось бы свыше 3 млрд. лет, или время, превышающее примерно в 200 раз возраст Вселенной (!).
- Чем больше отношений участвуют в операции соединения, тем больше выигрыш. Иными словами, чем сложнее запрос, тем более значительными становятся преимущества системы, основанной на модели TR, над системами с непосредственным отображением.
- Поскольку все соединения реализуются одинаковым способом, нет необходимости выполнять сложный процесс выбора пути доступа, который является обязательным в системах с непосред ственным отображением.

- Более того, указанный процесс выбора пути доступа, который приходится выполнять в системах с непосредственным доступом, в любом случае имеет сомнительную точность (кроме всего прочего) из-за сложности оценки размеров промежуточных результатов.

Объединение, пересечение и разность

Для использования в качестве основы для примеров данного подраздела дополним рассматриваемую базу данных, включив в нее переменную отношения детали P с обычно применяемыми в данной книге значениями. Дополним также столбец CITY слившейся таблицы значений полей, чтобы включить в нее названия городов, в которых находятся детали, и соответствующие диапазоны строк. Этот столбец будет выглядеть следующим образом.

	CITY
1	Athens [1:1] [:]
2	London [2:3] [1:3]
3	Oslo [:] [4:4]
4	Paris [4:5] [5:6]

Теперь рассмотрим операцию в следующей форме.

X { CITY } op Y { CITY }

Здесь X — переменная отношения S или P, Y — другая из этих двух переменных отношения, а op — операция UNION, INTERSECT или MINUS. Метод использования слившегося столбца CITY в таблице значений полей при реализации таких операций должен быть очевидным. По сути, он должен быть таким, как описано ниже.

- UNION. Каждое конкретное название города появляется в результате тогда и только тогда, когда имеется непустой диапазон строк для поставщиков или деталей, или тех и других. Иными словами, объединение — это просто множество всех названий городов в слившемся столбце.
- INTERSECT. Каждое конкретное название города появляется в результате тогда и только тогда, когда имеется непустой диапазон строк и для поставщиков, и для деталей.
- MINUS. Если переменной отношения X является S, то каждое конкретное название города появляется в результате тогда и только тогда, когда имеется непустой диапазон строк для поставщиков и пустой— для деталей. Аналогичным образом, если переменной отношения X является P, то каждое конкретное название города появляется в результате тогда и только тогда, когда имеется непустой диапазон строк для деталей и пустой — для поставщиков.

Очевидно, что все эти операции могут быть реализованы за один проход по столбцу CITY слившейся таблицы значений полей.

Заключительные замечания

По вопросу использования модели TR для реализации реляционных операций может быть сказано гораздо больше, но примеров, приведенных в этом разделе, должно быть достаточно для демонстрации основной идеи. В завершение приведем еще несколько замечаний.

- В реализациях с непосредственным отображением иногда возникает необходимость материали зовать промежуточные результирующие отношения. Такая же необходимость возникает и при использовании модели TR, но при этом указанная материализация, во-первых, обычно требуется менее часто и, во-вторых, является более эффективной в случае ее вынужденного выполнения (в основном благодаря всей предварительной сортировке).
- В модели TR могут быть очень эффективно реализованы ограничения потенциального ключа (уникальности) и внешнего ключа (ссылочной целостности) благодаря тому, что таблицы значений полей обычно являются одновременно и сжатыми, и слившимися.
- Ниже приведена цитата из самой первой статьи Кодда, посвященной реляционной модели [6.1].

Зная о том, что существует определенное отношение, пользователь будет рассчитыватъ на то, что у него есть возможность эксплуатировать это отношение с использованием любой комбинации его атрибутов как "известного", а оставшихся атрибутов — как "неизвестного", поскольку информация, даже если она не видна, все равно присутствует в этом отношении (как не исчезает и Эверест, даже если его нельзя разглядеть за облаками). В этом и состоит одно из свойств реляционной системы (отсутствующее во многих других современных информационных системах), которое мы в дальнейшем будем называть (вполне обоснованно) симметричной эксплуатацией *отношений. Но, вполне естественно, нельзя рассчитывать на то, что такая же симметрия будет наблюдаться и в части производительности.*

Но модель TR, кроме симметрии в эксплуатации, обеспечивает также симметрию в производительности! По крайней мере, она позволяет намного ближе подойти к этой цели, чем реализации с непосредственным отображением. Это происходит благодаря отделению значений полей от связующей информации, что фактически позволяет обеспечить физическое хранение данных во многих разных отсортированных последовательностях одновременно.

А.7. РЕЗЮМЕ

В этой главе кратко описана модель TR, представляющая собой принципиально новый подход к реализации реляционных СУБД. Модель TR представляет собой конкретное приложение более общей технологии, известной под названием метода преобразования Тарена, который предназначен для использования в качестве технологии реализации систем хранения и выборки данных многих типов (а не только СУБД). Метод преобразования Тарена является объектом действия патента США [А.2] и интеллектуальной собственностью компании Required Technologies, Inc. (http: / /www. requiredtech. com).

В данной главе было показано применение модели TR для баз данных, предназначенных только для чтения и находящихся в оперативной памяти. Хотя приведенное здесь описание, к сожалению, не могло не оказаться весьма поверхностным, автор надеется, что оно было достаточным для того, чтобы показать, насколько значительно модель TR отличается от обычной технологии с непосредственным отображением. Вместе с тем, автор не сомневается в том, что у читателя возникло много вопросов. Ниже приведены наиболее характерные вопросы, на которые нельзя найти ответ в данном приложении.

- Можно ли обеспечить эффективное сопровождение таблиц значений полей и реконструкции записей в условиях применения к базе данных произвольных операций обновления?
- Поскольку таблица реконструкции записей изоморфна первоначальному файлу (по сути, первоначальному отношению), но в каждой ячейке вместо значения данных содержит указатель, разве модель TR не потребует гораздо больше памяти по сравнению с той обычной реализацией, в которой применяется непосредственное отображение?
- Разве не следует ожидать, что в системе с хранимой на диске информацией зигзагообразное обращение к таблицам потребует большого количества операций произвольного доступа и повлечет за собой резкое снижение производительности? И можно ли добиться эффективного осуществления бинарного поиска на диске?

Подобные вопросы, безусловно, являются очень серьезными, но (к сожалению) в данной книге не предусмотрен дополнительный объем для их освещения; достаточно сказать, что они могли быть и были успешно решены, а эти решения реализованы. Дополнительная информация приведена в [АЛ] и [А.2].

СПИСОК ЛИТЕРАТУРЫ

АЛ. Date C. J. Go Faster! The TransRelational™ Approach to DBMS Implementation // Готовится к выходу.

Настоящее приложение в значительной степени основано на данной книге, но в ней приведено намного больше подробностей и описаны многие дополнительные темы. В частности, в ней рассматриваются операции обновления и базы данных, хранимые на диске, а также приведены ответы на вопросы, которые были подняты в разделе А.7.

A.2. U.S. Patent and Trademark Office: Value-lnstance-Connectivity Computer-Implemented Database. U.S. Patent No. 6,009,432. - December 28, 1999.

Это — оригинальный патент по методу преобразования Тарена; данный метод является интеллектуальной собственностью компании Required Technologies, Inc. (http: //www. requiredtech.com). Но следует отметить, что метод преобразования Тарена в значительной степени расширен, и вышел за пределы того, что было описано в этом оригинальном патенте. В частности, он теперь охватывает целый ряд методов обновления и способов организации работы с базами данных на диске (см. [A.1]). Кроме того, заслуживает внимания само название этого источника: оригинальный патент был сформулирован в терминах (кроме всего прочего) того, что в нем было названо *хранилищами значений, хранилищами экземпляров* и *хранилищами связности.* В отличие от этого, в настоящем приложении и в [АЛ) используются такие термины, как *таблицы значений полей* и *таблицы реконструкции записей,* в надежде на то, что эти последние термины являются более удобными для пользователя в определенных аспектах и лучше передают суть происходящих действий.

ПРИЛОЖЕНИЕ Б

Выражения SQL

Б.1. ВВЕДЕНИЕ

Табличные и логические выражения составляют основу языка SQL. В данном приложении представлена грамматика указанных выражений в форме Бэкуса-Наура. Кроме того, в некоторых случаях подробно описана семантика таких выражений. Тем не менее, здесь преднамеренно исключены следующие описания:

- подробные сведения о скалярных выражениях;
- подробные сведения о форме RECURSIVE конструкции WITH;
- нескалярные элементы оператора выборки *<select item>;*
- варианты ONLY конструкций < table *ref> и <type speo;*
- ОПЦИИ GROUPING SETS, ROLLUP И CUBE конструкции GROUP BY;
- условия BETWEEN, OVERLAPS И SIMILAR;
- все, что касается неопределенных значений (NULL-значений).

Необходимо также отметить, что применяемые в данном приложении названия для синтаксических категорий и конструкций языка SQL чаще всего отличаются от используемых в самом стандарте SQL [4.23], поскольку, по мнению автора, стандартные термины не всегда удачны. В частности, здесь вместо опций <table *value constructor* и *<row value constructors,* соответственно, используются просто их сокращенные варианты < *table constructor* и *<row constructor.*

Б.2. ТАБЛИЧНЫЕ ВЫРАЖЕНИЯ

Вначале рассмотрим приведенную ниже грамматику табличных выражений < *table exp*> в форме Бэкуса-Наура.

<table *exp*>
::= <*with exp*> \ <*nonwith exp*>

<*with exp*>
::= WITH [RECURSIVE]
<*table name*> [(<*column name commalist*>)]] AS (
<*table exp*> *)* <*nonwith exp*>

```
<nonwith exp>
    ::= <join table exp> | <nonjoin table exp>

<join table exp>
    ::= <table ref> I NATURAL ] JOIN <table ref>          [ ON <bool exp>
                        | USING ( <column name conmalist> ) ] | <table ref> CROSS
        JOIN <table ref> | ( <join table exp> )

<table ref>
    ::= <table name> [ [ AS ] <range var name>
                        [ ( <column name commalist> ) ] ] | ( <nonwi th exp> ) [ AS ]
        <range var name>
                            [ ( <column name commalist> ) ] | <join table exp>

<nonjoin table exp>
    ::= <nonjoin table term>
        | <nonwith exp> UNION [ ALL | DISTINCT ]
          [ CORRESPONDING [ BY ( <column name commalist> ) ] ]
            <table term> <nonwith exp> EXCEPT [ ALL | DISTINCT ]
          [ CORRESPONDING [ BY ( <column name commalist> ) ] ]
            <table term>

<nonjoin table term>
    ::= <nonjoin table primary>
        | <table term> INTERSECT [ ALL | DISTINCT ]
          [ CORRESPONDING [ BY ( <column name commalist> ) ] ] <table primary>

<table term>
    ::= <nonjoin table term> \ <join table exp>

<table primary>
    ::= <nonjoin table primary> \ <join table exp>

<nonjoin table primary>
    ::= TABLE <table name> | <table constructor> |
        <select exp> | ( <nonjoin table exp> )

<table constructor
    ::= VALUES <row constructor commalist>

<row constructor
    ::= <scalar exp>
        | ( <scalar exp commalist> ) I ( <table exp> )

<select exp>
    ::= SELECT [ ALL | DISTINCT ] <select item commalist> FROM <table ref
            commalist> [ WHERE <bool exp> ]
                [ GROUP BY <column name commalist> ] [ HAVING <bool
                    exp> ]
```

```
<select item>
        ::=    <scalar exp>  [  [ AS  ]  <column name>  ] |
             [  <range var name>.  ]  *
```

Теперь перейдем к конкретному случаю выражений выборки <*select* exp>, которые, безусловно, представляют наибольший интерес с точки зрения практики. Выражения <*select exp*> можно неформально определить как табличные выражения <*table exp*>, в которых не применяются операторы JOIN, UNION, EXCEPT или INTERSECT. В данном контексте слово "неформально" означает, что такие операторы, разумеется, могут применяться в выражениях, которые вложены в рассматриваемое выражение <select *exp*>. Как показывает грамматическая структура выражения <*select* exp>, в нее могут входить, кроме конструкции SELECT, конструкция FROM, необязательная конструкция WHERE, необязательная конструкция GROUP BY и необязательная конструкция HAVING, в указанной последовательности. Ниже каждая из этих конструкций рассматривается по очереди.

Конструкция SELECT

Конструкция SELECT принимает следующую форму.

SELECT [ALL | DISTINCT] <*select item commalist*>

Пояснение

1. Разделенный запятыми список элементов выборки <*select item commalist*> должен быть непустым' (подробное описание опций с обозначением элемента выборки <select i tem> при ведено ниже).
2. Если не заданы ни ключевое слово ALL, ни ключевое слово DISTINCT, по умолчанию применя ется ключевое слово ALL.
3. Примем на время предположение, что вычисление выражений в конструкциях FROM, WHERE, GROUP BY и HAVING уже выполнено. Независимо от того, какие из этих конструкций определе ны и какие опущены, результатом их вычисления концептуально всегда становится таблица (возможно, "сгруппированная" таблица, как описано ниже), которая будет именоваться табли цей T1 (хотя этот концептуальный результат фактически является безымянным).
4. Допустим, что T2 — таблица, полученная из таблицы T1 путем вычисления указанного элемента выборки <*select item*> применительно к T1.
5. Допустим, что T3 — таблица, которая получена из таблицы T2 путем устранения избыточных дубликатов строк из T2, если задано ключевое слово DISTINCT, или таблица, идентичная T2, в противном случае.
6. Таблица T3 представляет собой окончательный результат.

Теперь перейдем к описанию элементов выборки <*select item*>. Для этого необходимо рассмотреть два случая, причем второй из них относится просто к форме, в которой применяется сокращенный вариант разделенного запятыми списка <*select item*> из первой формы; таким образом, первый случай действительно является более фундаментальным.

Случай 1. Опция <*select i ten»* принимает следующую форму.

<*scalar exp*> [[AS] <*column name*>]

Скалярное выражение <*scalar exp*> обычно (но не обязательно) включает один или несколько столбцов таблицы T1 (см. п. 3). Для каждой строки T1 вычисление выражения <*scalar exp*> приводит

---

[1] Фактически, все простые списки и разделенные запятыми списки, рассматриваемые в данном при ложении, должны быть непустыми.

[2] Иными словами, в контексте оператора SELECT применяемым по умолчанию значением является ALL. В отличие от этого, в контексте UNION, INTERSECT или EXCEPT применяемым по умолчанию зна чением является DISTINCT.

к получению скалярного результата. Разделенный запятыми список таких результатов (соответствующий вычислению всех элементов выборки <*select item*> в конструкции SELECT применительно к отдельной строке T1) составляет одну строку таблицы T2 (см. п. 4). Если элемент <*select item*> включает конструкцию AS, то неуточненное имя столбца <*column name*> из этой конструкции назначается как имя соответствующего столбца T2 (необязательное ключевое слово AS предусмотрено просто для удобства и может быть опущено; в результате этого смысл конструкции не изменится). Если элемент <*select i ten»* не включает конструкцию AS, могут рассматриваться следующие два варианта: во-первых, если этот элемент состоит просто из имени столбца <*column name*> (возможно, уточненного), то имя <*column name*> присваивается в качестве имени соответствующему столбцу таблицы T2; во-вторых, в противном случае соответствующий столбец T2, по сути, остается безымянным (но фактически этому столбцу присваивается внутреннее имя, зависящее от реализации). Из этого следуют приведенные ниже выводы.

- Поскольку имя, введенное с помощью конструкции AS, безусловно, относится к столбцу таблицы T2, а не таблицы T1, то это имя не может использоваться в конструкциях WHERE, GROUP BY и HAVING, непосредственно участвующих в формировании таблицы T1 (если они имеются). Однако на него можно ссылаться в соответствующей конструкции ORDER BY, если она имеется, и также во "внешнем" выражении <*table* exp>, которое содержит вложенное в него рассматриваемое выражение <*select exp*>.
- Если элемент <*select item*> включает вызов агрегирующего оператора, а выражение <*select exp*> не включает конструкцию GROUP BY (см. ниже), то ни один элемент <*select i tem*> в конструкции SELECT не может включать каких-либо ссылок на столбец T1, если только эта ссылка на столбец не является фактическим параметром (или не входит в состав фактического параметра) в вызове агрегирующего оператора.

Случай 2. Опция <*select i tem*> принимает следующую форму.

[<*range var name*>.] *

Если спецификатор опущен (т.е. опция <select *item*> представляет собой просто звездочку и не имеет уточнения), то данная опция <*select item*> должна быть единственной опцией <*select item*> в конструкции SELECT. Такая форма является сокращенным обозначением разделенного запятыми списка всех опций <*column name*> для таблицы T1 в последовательности расположения столбцов слева направо. Если же спецификатор включен (т.е. опция <*select item*> представляет собой звездочку, уточненную именем переменной области значений R, например, "R.*"), то опция <*select item*> представляет собой разделенный запятыми список имен столбцов <*co1umn name*>, который включает все столбцы таблицы, связанные с переменной области значений R в последовательности расположения столбцов слева направо. (Напомним, что в разделе 8.6 было отмечено, что в качестве неявно заданной переменной области значений может использоваться и часто используется имя таблицы. Поэтому опция <*select i tem*> чаще применяется в форме "T. *", а не "R. *".)

Конструкция FROM

Конструкция FROM принимает следующую форму.

FROM <*table ref commalist*>

Допустим, что заданные выражения со ссылками на таблицу <*table ref*> после вычисления принимают, соответственно, значения таблиц A, B, . . ., Z. Тогда результатом вычисления конструкции FROM является таблица, равная декартову произведению (в стиле SQL) таблиц A, B, **. . . , Z.**

Конструкция WHERE

Конструкция WHERE принимает следующую форму.

WHERE <*bool exp*>

Допустим, что т — результат вычисления конструкции FROM, непосредственно предшествующей этой конструкции WHERE. Тогда результат вычисления конструкции WHERE представляет собой таблицу, полученную из Т путем удаления всех строк, для которых логическое выражение <*bool exp*> не принимает значения TRUE. Если же конструкция WHERE опущена, то результатом становится просто Т.

Конструкция GROUP BY

Конструкция GROUP BY принимает следующую форму.

GROUP BY <*column name commalist*>

Допустим, что Т — результат вычисления непосредственно предшествующей конструкции FROM и конструкции WHERE (если она имеется). Каждое имя столбца <*column n*ame>, упомянутое в конструкции GROUP BY, должно представлять собой имя столбца таблицы Т с необязательным уточнителем. Результатом вычисления конструкции GROUP BY является сгруппированная таблица (иными словами, множество групп строк, полученное из таблицы Т путем концептуального переупорядочения строк с разбивкой на минимальное количество групп, таких что в каждой отдельной группе все строки имеют одинаковое значение для комбинации столбцов, обозначенной конструкцией GROUP BY). Поэтому заслуживает особого внимания то, что этот результат фактически не представляет собой "настоящую таблицу" (еще раз повторяем, что эта таблица состоит из групп строк, а не из отдельных строк). Но конструкция GROUP BY никогда не применяется без соответствующей конструкции SELECT, в результате действия которой из таблицы групп создается настоящая таблица, поэтому из-за указанного временного отклонения от инфраструктуры "настоящей таблицы" никаких нарушений в работе не возникает.

Если выражение выборки <*select exp*> включает конструкцию GROUP BY, то каждый элемент выборки <*select* item> в конструкции SELECT (включая все те элементы, которые становятся следствием применения сокращенного обозначения в виде звездочки), должен быть однозначным в расчете на каждую группу.

Конструкция HAVING

Конструкция HAVING принимает следующую форму.

HAVING <*bool exp*>

Допустим, что G — сгруппированная таблица, полученная в результате вычисления непосредственно предшествующих конструкций: конструкции FROM, конструкции WHERE (если она имеется) и конструкции GROUP BY (если она имеется). Если конструкция GROUP BY отсутствует, то таблица G рассматривается как результат вычисления одних только конструкций FROM и WHERE, поэтому считается сгруппированной таблицей, содержащей одну и только одну группу;[3] иными словами, в данном случае применяется неявная концептуальная конструкция GROUP BY, которая вообще не задает ни одного группирующего столбца. Результатом применения конструкции HAVING является сгруппированная таблица, полученная из таблицы G путем удаления всех групп, для которых логическое выражение <*bool exp*> не принимает значения TRUE. Из этого вытекают приведенные ниже выводы.

- Если конструкция HAVING опущена, а конструкция GROUP BY включена, то результатом вычисления конструкции HAVING является просто таблица G. Если опущены обе конструкции, и HAVING, и GROUP BY, то результатом становится просто "настоящая" (т.е. несгруппированная) таблица Т, полученная в результате применения конструкций FROM и WHERE.
- Любое скалярное выражение <*scalar exp*> в конструкции HAVING должно быть однозначным в расчете на каждую группу (как и выражения <*scalar exp*> в конструкции SELECT, если имеется конструкция GROUP BY, как было описано в предыдущем подразделе).

[3] Именно это указано в стандарте, хотя было бы точнее применить здесь формулировку "не больше одной группы" (поскольку фупп не должно быть вообще, если применение конструкций FROM и WHERE приводит к получению пустой таблицы).

Всеобъемлющий пример

В завершение этого описания выражений выборки <*select exp*> приведем довольно сложный пример, который иллюстрирует несколько особенностей этих выражений, описанных а предыдущих подразделах (но, разумеется, не все). Соответствующий запрос приведен ниже.

Для всех деталей красного и синего цвета, таких что общий объем их поставки превышает 350 (но из итогов исключены все поставки, для которых это количество меньше или равно 200), получить номера деталей, вес в граммах, цвет и максимальное поставляемое количество этих деталей.

Ниже приведена возможная формулировка этого запроса на языке SQL.

```
SELECT  P.P#,
         'Weight in grains =' AS TEXT1,
        P.WEIGHT * 454 AS GMWT,
        P.COLOR,
'Max quantity =' AS TEXT2, MAX (
SP.QTY ) AS MXQTY FROM   P, SP WHERE
P.P# = SP.P#
AND  ( P.COLOR = COLOR ('Red') OR P.COLOR = COLOR ('Blue') ) AND
SP.QTY > QTY ( 200 ) GROUP  BY P.P#, P.WEIGHT, P.COLOR HAVING SUM (
SP.QTY ) > QTY ( 350 ) ;
```

Пояснение. Прежде всего необходимо отметить, что (как указано в предыдущих подразделах) конструкции выражения <select *exp*> концептуально вычисляются в том порядке, в каком они записаны. Единственным исключением из этого правила является сама конструкция SELECT, которая вычисляется в последнюю очередь. Поэтому в данном примере можно предположить, что результат формируется, как описано ниже.

1. Конструкция FROM. В приведенном выше предложении конструкция FROM вычисляется для по лучения новой таблицы, которая представляет собой декартово произведение таблиц P и SP.
2. Конструкция WHERE. К результату, полученному в шаге 1, применяется операция сокращения, выполняемая путем удаления всех строк, которые не удовлетворяют условию конструкции WHERE. Поэтому в данном примере удаляются строки, которые не соответствуют следующему логическому выражению <*Ьоо1 exp*>.

```
          P.P#  =  SP.P#
AND  ( P.COLOR = COLOR ('Red') OR P.COLOR = COLOR ('Blue') ) AND
SP.QTY > QTY ( 200 )
```

3. Конструкция GROUP BY. Результат, полученный в шаге 2, группируется по значениям в столбце (столбцах), названном в конструкции GROUP BY. В данном примере такими столбцами являются P. P#, P . WEIGHT И P. COLOR.
4. Конструкция HAVING. Из результата, полученного в шаге 3, удаляются группы, не удовлетво ряющие следующему логическому выражению <*bool* exp>: SUM (SP.QTY) > QTY (350).
5. Конструкция SELECT. Каждая группа в результате шага 4 применяется для выработки одной строки окончательного результата следующим образом. Во-первых, из группы извлекаются дан ные о номере детали, весе, цвете и максимальном количестве. Во-вторых, вес преобразуется в граммы. В-третьих, в соответствующие места данной строки вставляются две символьные строки "Weight in grams =" и "Max quantity =". Кстати, следует отметить, что здесь использует ся тот факт, что в языке SQL столбцы таблиц имеют упорядочение слева направо (о чем говорит фраза "соответствующие места данной строки"); если бы текстовые надписи появлялись не в этих "соответствующих местах", то они имели бы мало смысла.

Окончательный результат выглядит следующим образом.

P#	TEXT1	GMWT	COLOR	TEXT2	MXQTY
P1	Weight in grams =	5448	Red	Max quantity =	300
P5	Weight in grams =	5448	Blue	Max quantity =	400
P3	Weight in grams =	7718	Blue	Max quantity =	400

Читатель должен учитывать, что приведенное выше описание алгоритма выполнения запроса предназначено исключительно в качестве концептуального объяснения того, как происходит вычисление выражения *<select exp>*. Этот алгоритм, безусловно, является правильным в том смысле, что он гарантирует получение правильного результата. Но если бы он действительно выполнялся в соответствии с этим описанием, то оказался бы довольно неэффективным. Например, вряд ли было бы целесообразно фактически формировать декартово произведение в шаге 1. Именно такие соображения и являются той причиной, по которой в реляционных системах требуется оптимизатор, как было описано в главе 18. Безусловно, задачу оптимизатора в системе SQL можно охарактеризовать как поиск процедуры реализации, позволяющей получить точно такой же результат, как при использовании только что (схематически) описанного концептуального алгоритма, но являющейся более эффективной, чем этот алгоритм.

Б.3. ЛОГИЧЕСКИЕ ВЫРАЖЕНИЯ

Как и в предыдущем разделе, начнем с рассмотрения грамматики в форме Бэкуса-Наура, затем перейдем к более подробному описанию условий *<like cond>, <match cond>* и *<all* or *any cond>*.

```
<bool exp>
   :: =   <bool term> | <bool exp> OR <bool term>

<bool term>
   ::=     <bool factor>  [ <bool term> AND <bool factor>

<bool factor>m
   ::=     [ NOT ]  <bool primary>

<bool primary>
   ::=    <simple cond>  |  ( <bool exp> )

<simple cond>
   ::=    <comp cond>  \  <in cond>  \  <like cond>  \  <match cond>
        |  <all or any cond>  \  <exists cond>  \  <unigue cond>
        j  <distinct cond>  |  <type cond>

<comp cond>
   : : =     <row constructor> <comp op>  <row constructor>

<comp op>
   ::=     =  |  <  |  <=  |  >  |  >=  |  <>

<in cond>
   ::=     <row constructor  [ NOT ]  IN  (  <table exp>  )
        |  <scalar exp>  [ NOT ]  IN  (  <sca.lar exp commalist>  )

<like cond>
   ::=    <char string exp> [ NOT ] LIKE <pattern>
                          [ ESCAPE <escape> ]
```

```
<match cond>
    ::=   <row constructors MATCH UNIQUE ( <table exp> )

<all or any cond>
    ::=   <row constructor <comp op> ALL ( <table exp> )
| <xow constructor <comp op> ANY ( <table exp> )

<exists cond>
    ::=   EXISTS ( <table exp> )
<unique cond>
    ::=   UNIQUE ( <table exp> )

<distinct cond>
    ::=   <row constructor IS DISTINCT FROM <row constructor>

<type cond>
    ::=  TYPE ( <scalar exp> )
           IS [ NOT ] OF ( <type spec commalist> )

<type spec>
    : :=   <type name>
```

Условия LIKE

Условия LIKE предназначены для сопоставления символьных строк с простым образцом, т.е. для проверки заданной символьной строки в целях определения того, соответствует ли она некоторому установленному образцу. Ниже (еще раз) показан синтаксис такого условия.

<char *string exp*> [NOT] LIKE <*pattern*> [ESCAPE <*escape*>]

Здесь <*pattern*> — произвольное выражение в виде символьной строки, а обозначение маскирующего символа <*escape*>, если оно определено, представляет собой выражение в виде символьной строки, результатом вычисления которого является один символ. Пример применения условия LIKE приведен ниже.

```
SELECT P.P#,  P.PNAME
FROM P
WHERE P.PNAME LIKE 'C%';
```

("Определить номера и названия деталей, названия которых начинаются с буквы С"). При использовании данных, обычно рассматриваемых в данной книге, будет получен следующий результат.

P#	PNAME
P5	Cam
P6	Cog

Если конструкция ESCAPE не определена, то символы, входящие в состав шаблона <pattern>, интерпретируются следующим образом.

- Символ подчеркивания "_" обозначает любой отдельный символ.
- Знак процента "%" представляет любую последовательность из п символов (где п может быть
  равно нулю).
- Все другие символы обозначают сами себя.

Поэтому в приведенном выше примере запрос возвращает из таблицы P такие строки, в которых значение PNAME начинается с прописной буквы C, за которой следует последовательность любых символов в количестве от нуля и больше. Ниже приведено еще несколько примеров.

```
ADDRESS LIKE '%Berkeley%'
```

- Принимает значение TRUE, если в любом месте поля ADDRESS содержится строка "Berkeley".

```
S# LIKE 'S  '
```

- Принимает значение TRUE, если содержимое поля S# имеет длину точно три символа, и первым из них является "S".

```
PNAME LIKE '%C  '
```

- Принимает значение TRUE, если содержимое поля PNAME имеет длину четыре символа или больше, и четвертым справа символом является "c".

```
MYTEXT LIKE '=_%' ESCAPE '='
```

- Принимает значение TRUE, если конструкция MYTEXT начинается с символа подчеркивания (см. следующий абзац).

В данном последнем примере символ "=" был определен как маскирующий. Это означает, что при желании можно отменить специальную интерпретацию, применяемую к символам "_" и "%", поставив перед ними символ "=", обозначенный как маскирующий символ.

Наконец, такое условие *<like cond>,* как

X NOT LIKE y [ESCAPE z] определено в качестве

семантического эквивалента следующего выражения.

```
NOT ( x LIKE y [ ESCAPE z ] )
```

Условия MATCH

Условие согласования *<ma tch cond>* принимает следующую форму.

<row constructor MATCH UNIQUE (*<table exp>*)

Допустим, что rl — строка, полученная в результате вычисления выражения конструктора строки *<row constructor,* а T — таблица, которая получена в результате вычисления табличного выражения *<table* exp>. В таком случае условие *<match cond>* принимает значение TRUE тогда и только тогда, когда T содержит точно одну строку (скажем, r2), такую что приведенная ниже операция сравнения принимает значение TRUE.

```
rl = r2
```

Например, рассмотрим следующее предложение.

```
SELECT SP.*
FROM SP
WHERE NOT ( SP.S# MATCH UNIQUE ( SELECT S.S# FROM S ) ) ;
```

("Определить поставки, которым не соответствует один и только один поставщик из таблицы поставщиков".) Такой запрос может применяться при проверке целостности базы данных, поскольку, безусловно, если база данных является правильной, то не должно быть ни одной такой поставки. Но следует отметить, что для выполнения точно такой же проверки может использоваться условие IN *<in cond>*.

Кстати, следует отметить, что ключевое слово UNIQUE в конструкции MATCH UNIQUE может быть опущено, но тогда конструкция MATCH становится синонимом конструкции IN (в отсутствие неопределенных значений).

Условия ALL или ANY

Условие "для всех или для некоторого", <all *or any cond*>, имеет следующую общую форму.

<row constructor <comp op> <qualifier> (<table exp>)

Здесь операцией сравнения <*comp op*> может быть любая операция из обычного набора ("=", "о" и т.д.). а уточнитель <*qualifier*> может принимать значение ALL или ANY[4]. Вообще говоря, условие <*all or any cond*> принимает значение TRUE тогда и только тогда, когда соответствующее сравнение без ключевого слова ALL (соответственно, ANY) принимает значение TRUE для всех (соответственно, некоторых) строк таблицы, представленной с помощью табличного выражения < *table exp*>. (Если эта таблица пуста, то условия ALL принимают значение TRUE, а условия ANY — значение FALSE.) Ниже приведен один пример ("Определить названия деталей, имеющих вес больше по сравнению со всеми деталями синего цвета").

```
SELECT DISTINCT
PX.PNAME FROM P AS PX
 WHERE PX.WEIGHT >ALL ( SELECT
PY.WEIGHT FROM P AS PY WHERE
PY.COLOR = 'Blue' ) ;
```

При использовании данных, обычно рассматриваемых в этой книге в качестве примера, полученный результат будет выглядеть следующим образом.

PNAME
Cog

Пояснение. Вложенное табличное выражение <table exp> возвращает множество значений веса деталей синего цвета. Затем внешняя конструкция SELECT возвращает названия деталей, имеющих вес, больший по сравнению с любым из значений в этом множестве. Безусловно, в общем случае окончательный результат может включать любое количество названий деталей (включая нуль).

Примечание. Необходимо сделать одно предостережение, которое касается смыслового значения применяемых здесьключевых слов. Фактически условия <*all or any cond*> могут стать источником ошибки. Подбирая ключевое слово для формулировки приведенного выше запроса, пользователь может ошибочно предположить, что слово ANY (любой) означает "каждый", и поэтому (неправильно) применить выражение >ANY вместо >ALL. Аналогичное замечание относится ко всем (любым?) операторам с ключевыми словами ANY и ALL.

Приложение B

Сокращения и специальные символы

1NF	first normal form	первая нормальная форма (1НФ)
2NF	second normal form	вторая нормальная форма (2НФ)
2PC	two-phase commit	двухфазная фиксация
2PL	two-phase locking	двухфазная блокировка
2VL	two-valued logic	двузначная логика
20C		то же, что и 2PC
20L		то же, что и 2PL
3GL	third-generation language	язык (программирования) третьего поколения
3NF	third normal form	третья нормальная форма (3НФ)
3VL	three-valued logic	трехзначная логика
4GL	fourth-generation language	язык (программирования) четвертого поколения
4NF	fourth normal form	четвертая нормальная форма (4НФ)
4VL	four-valued logic	четырехзначная логика
5NF	fifth normal form (same as PJ/NF)	пятая нормальная форма (5НФ) (то же, что и PJ/NF)
6NF	sixth normal form	шестая нормальная форма (6НФ)
A	ALGEBRA	см. ниже
ACID	atomicity-consistency-isolation-durability	неразрывность-согласованность-изолированность-устойчивость
ACM	Association for Computing Machinery	Ассоциация по вычислительной технике
ADT	abstract data type	абстрактный тип данных
AES	Advanced Encryption System	усовершенствованная система шифрования
ALGEBRA	A Logical Genesis Explains Basic Relational Algebra	рекомендация: "Основы реляционной алгебры можно понять, изучая процесс развития логики"
ANSI	American National Standards Institute	Американский национальный институт стандартов
ANSI/SPARC	ANSI/Systems Planning and Requirements Committee	Американский национальный институт стандартов/Комитет системного планирования и определения требований (упоминался в главе 2 в связи с трехуровневой архитектурой систем баз данных)
API	application programming interface	интерфейс прикладного программирования
ARIES	Algorithms for Recovery and Isolation Exploiting Semantics	алгоритмы восстановления и изоляции на основе семантики
AST	automatic summary table	итоговая таблица с автоматизированным расчетом данных

ВВ		то же, что и гигабайт (Гбайт)
BCNF	Boyce/Codd normal form	нормальная форма Бойса-Кодда (НФБК)
BLOB	binary large object	большой двоичный объект
BNF	Backus-Naur form; Backus normal form	форма Бэкуса-Наура, или нормальная форма Бэкуса
CACM	Communications of the ACM	издание ассоциации ACM
CAD/CAM	computer-aided design/computer-aided manufacturing	автоматизированное проектирование/автоматизированное производство (САПР/АСУП)
CASE	computer-aided software engineering	автоматизированное проектирование и создание программ
CDO	class-defining object	объект, определяющей класс
CIM	computer-integrated manufacturing	комплексно автоматизированное производство
CLI	Call-Level Interface	интерфейс уровня вызова (процедур)
CLOB	character large object	большой символьный объект
CNF	conjunctive normal form	конъюнктивная нормальная форма
CODASYL	Conference on Data Systems Languages	Конференция по языкам обработки данных; используется при ссылке на определенные дореляционные (фактически сетевые) системы, такие как IDMS
CPU	central processing unit	центральный процессор (ЦП)
CS	cursor stability (DB2)	стабильность курсора (в СУБД DB2)
CWA	Closed World Assumption	предположение о замкнутости мира
DA	data administrator	администратор данных (АД)
DB/DC	database/data communications	база данных/передача данных
DBA	database administrator	администратор базы данных (АБД)
DBMS	database management system	система управления базой данных (СУБД)
DBP&D	Database Programming & Design	первоначально - журнал, издаваемый типографским способом, затем журнал, предоставляемый в оперативном режиме по адресу http: //www.dbpd.com; в настоящее время вместо него издается журнал Intelligent Enterprise
DBTG	Data Base Task Group	Рабочая группа по базам данных; в контексте баз данных эта аббревиатура используется взаимозаменяемо с CODASYL
DC	data communications	передача данных
DCO	domain check override	замещение проверки домена
DDB	distributed database	распределенная база данных
DDBMS	distributed DBMS	распределенная СУБД (РСУБД)
DDL	data definition language	язык определения данных (ЯОД)
DES	Data Encryption Standard	стандарт шифрования данных
DK/NF	domain-key normal form	доменно-ключевая нормальная форма
DML	data manipulation language	язык манипулирования данными (ЯМД)
DNF	disjunctive normal form	дизъюнктивная нормальная форма
DOM	Document Object Model	объектная модель документа (XML)
DRDA	Distributed Relational Database Architecture	архитектура распределенных реляционных баз данных (компании IBM)
DSL	data sublanguage	подъязык данных
DSS	decision support system	система поддержки принятия решений
DTD	Document Type Definition	определение типа документа (XML)

DUW	distributed unit of work	распределенная единица работы •'
E/R	entity/relationship	"сущность/связь"
EB		то же, что и XB
ECA	event-condition-action	событие-условие-действие
EDB	extensional database	экстенсиональная база данных
EDI	Electronic Data Interchange	электронный обмен данными
EKNF	elementary key normal form	нормальная форма с элементарными ключами
EMVD	embedded MVD	встроенная многозначная зависимость
EOT	end of transaction	конец транзакции
FD	functional dependence	функциональная зависимость (ФЗ)
FLWOR	for-let-where-order by-return	выражения языка XQuery (XML)
FTP	File Transfer Protocol	протокол передачи данных
GB	gigabyte (1024MB)	гигабайт (Гбайт); равен 1024 мегабайт (Мбайт)
GIS	geographic information system	геоинформационная (географическая информационная) система
GML	Generalized Markup Language	обобщенный язык разметки
HOLAP	hybrid OLAP	гибридная система OLAP
HTML	HyperText Markup Language	язык разметки гипертекста
HTTP	Hypertext Transfer Protocol	протокол передачи гипертекста
I/O	input/output	ввод-вывод
IDB	intensional database	интенсиональная база данных
IDMS	Integrated Database Management System	интегрированная система управления базой данных — одна из первых СУБД
IFIP	International Federation for Information Processing	Международная федерация по обработке информации
IEEE	Institute for Electrical and Electronics Engineers	Институт инженеров по электротехнике и электронике
IMS	Information Management System	информационно-управляющая система - одна из первых СУБД
INCITS	ANSI International Committee on Information Technology Standards	Международный комитет по стандартам информационной технологии института ANSI (прежде носил название NCITS, а перед этим - X3)
INCITS/H2	INCITS database committee	Комитет INCITS по базам данных
IND	inclusion dependence	зависимость включения
IS	intent shared (lock); information system	намеченная разделяемая блокировка; информационная система
ISBL	Information System Base Language	базовый язык информационной системы (в системе PRTV - Peterlee Relational Test Vehicle)
ISO	International Organization for Standardization	Международная организация по стандартизации
IT	information technology	информационная технология (ИТ)
IX	intent exclusive (lock)	намеченная исключительная блокировка
JACM	Journal of the ACM	издание ассоциации ACM
JD	join dependence	зависимость соединения (ЗС)
JDBC	Java DataBase Connectivity	средства организации доступа Java-приложений к базам данных в сети; JDBC - официальное обозначение данной технологии, а не аббревиатура
K		1024 байт (иногда этой буквой обозначают 1000 байт)
KB	kilobyte (1024 bytes)	килобайт;! Кбайт = 1024байт

LAN	local area network	локальная сеть (ЛВС)
LOB	large object	большой объект
USP	Liskov Substitution Principle	принцип замены Лисков (получивший название в честь Барбары Лисков)
MB	megabyte (1024KB)	мегабайт; 1 Мбайт = 1024 килобайт (Кбайт)
MLS	multi-level secure	многоуровневая система защиты
MOLAP	multi-dimensional OLAP	многомерная система OLAP
MQT	materialized query table	материализованная таблица запроса
MVD	multi-valued dependence	многозначная зависимость (М33)
NCITS		см. INCITS
NCITS/H2		см. INCITS/H2
NF^2	"NF squared" = NFNF = non first normal form	"NF квадрат" = NFNF = не первая нормальная форма (с точки зрения автора - весьма сомнительная концепция)
ODBC	Open Database Connectivity	открытый интерфейс доступа к базам данных
ODMG	Object Data Management Group	Рабочая группа по выработке и согласованию стандартов объектных баз данных
ODS	operational data store	банк оперативных данных
OID	object ID	идентификатор объекта
OLAP	online analytic processing	оперативная аналитическая обработка
OLCP	online complex processing	оперативная комплексная обработка
OLDM	online decision management	оперативное управление процессом принятия решений
OLTP	online transaction processing	оперативная обработка транзакций
OMG	Object Management Group	Рабочая группа по развитию стандартов объектного программирования
OO	object-oriented; object orientation	объектно-ориентированный; объектная ориентация (00)
OODB	object-oriented database (= object database)	объектно-ориентированная база данных (то же, что и объектная база данных)
OODBMS	object-oriented DBMS (= object DBMS)	объектно-ориентированная СУБД (тоже, что и объектная СУБД)
OOPL	object-oriented programming language (= object programming language)	объектно-ориентированный язык программирования (то же, что и объектный язык программирования)
OQL	Object Query Language	объектный язык запросов (часть предложения 0DMG)
OSI	Open Systems Interconnection	взаимодействие открытых систем
0SQ1	"Object SQL"	"объектный SQL"
PB	petabyte (1024TB)	петабайт; 1 Пбайт= 1024 терабайт (Тбайт)
PC	personal computer	персональный компьютер (ПК)
PJ/NF	projection-join normal form	проекционно-соединительная нормальная форма
PODS	Principles of Database Systems	конференция ассоциации ACM
PRTV	Peterlee Relational Test Vehicle	экспериментальная реляционная система, разработанная в городе Питерли
PSM	Persistent Stored Modules	постоянные хранимые модули (часть стандарта SQL)
PSVi	Post Schema Validation Infoset	инфонабор, полученный после проверки по схеме (язык XML)
QBE	Query-By-Example	запрос по образцу (язык формирования запросов)
QUEL	Query Language	язык запросов (применявшийся в СУБД Ingres)
RAID	redundant array of inexpensive disks	массив недорогих дисковых накопителей с избыточностью
R0A	Remote Data Access	удаленный доступ к данным

RDB	relational database	реляционная база данных
RDBMS	relational DBMS	реляционная СУБД
RID	record ID; row ID	идентификатор записи; идентификатор строки
ROLAP	relational OLAP	реляционная система OLAP
RM/T	relational model/Tasmania	реляционная модель/Тасмания
RM/V1	relational model/Version 1	реляционная модель/версия 1
RM/V2	relational model/Version 2	реляционная модель/версия 2
RPC	remote procedure call	дистанционный вызов процедур
RR	read-read; repeatable read	чтение-чтение; повторяемое чтение (применяется в СУБД DB2)
RSA	Rivest-Shamir-Adleman	метод шифрования
RUW	remote unit of work	удаленная единица работы
RVA	relation-valued attribute	атрибут со значением в виде отношения
S	shared (lock)	разделяемая блокировка
SGML	Standard GML	стандартный обобщенный язык разметки
SIGMOD	Special Interest Group on Management of Data	Специальная группа по управлению данными (специальная группа ACM)
SIX	shared intent exclusive (lock)	разделяемая намеченная исключительная блокировка
SOAP	Simple Object Access Protocol	простой протокол доступа к объектам
SPARC		см. ANSI/SPARC
SQL	первоначальная расшифровка - Structured Query Language; иногда применяется расшифровка Standard Query Language	официальное обозначение языка запросов, а не аббревиатура
SQL/MM	SQL/Multimedia	часть стандарта SQL
SVG	Scalable Vector Graphics	масштабируемая векторная графика
TB	terabyte (1024GB)	терабайт; 1 Тбайт = 1024 гигабайт (Гбайт)
TCB	Trusted Computing Base	высоконадежная вычислительная база
TCP/IP	Transmission Control Protocol/Internet Protocol	протокол управления передачей/межсетевой протокол
TID	tuple ID	идентификатор кортежа
TODS	Transactions on Database Systems	издание ассоциации ACM
TP	transaction processing	обработка транзакций
TPC	Transaction Processing Council	Совет по средствам обработки транзакций
TR	TransRelational	модель TransRelational™
U	update (lock)	блокировка обновления
UDF	user-defined function	функция, определяемая пользователем
UDO	user-defined operator	оператор, определяемый пользователем
UDT	user-defined type	тип, определяемый пользователем
UML	Unified Modeling Language	универсальный язык моделирования
unk	unknown	неизвестное значение (истинностное значение)
UNK	unknown (null)	неизвестное значение (NULL-значение)
UOW	unit of work	единица работы
URI	Uniform Resource Identifier	унифицированный идентификатор информационного ресурса
URL	Uniform Resource Locator	унифицированный локатор информационного ресурса
VLDB	very large database; Very Large Data Bases	очень большая база данных; Very Large Data Bases (ежегодная конференция)

VSAM	Virtual Storage Access Method	метод доступа к виртуальной памяти
W3C	World Wide Web Consortium	Консорциум World Wide Web
WAL	write-ahead log	журнал для опережающей записи
WAN	wide area network	глобальная сеть
WFF	well-formed formula	правильно построенная формула
WORM	write once/read many times	с однократной записью и многократным считыванием
WWW	World Wide Web	Всемирная Паутина
WYSIWYG	what you see is what you get	принцип наглядного отображения
X	exclusive (lock)	исключительная блокировка
X3		см. INCITS
X3H2		см. INCITS/H2
XB	exabyte (1024PB)	эксабайт; 1 Эбайт = 1024 петабайт(Пбайт)
XML	Extensible Markup Language	расширяемый язык разметки
XQuery	XML Query	язык запросов XML
XQL	XML Query Language	язык запросов XML (отличный от XQuery)
XSL	XML Stylesheet Language	язык таблиц стилей XML
XSLT	XML Stylesheet and Transformation Language	язык таблиц стилей и преобразований XML
YB	yottabyte (1024ZB)	йотабайт; 1 Йбайт = 1024дзетабайт (Дбайт)
ZB	zettabyte (1024XB)	дзетабайт; 1 Дбайт = 1024 эксабайт (Эбайт)
∈		принадлежит; является членом; содержится; относится
∋		содержит
⊆		является подмножеством; включено
⊂		является строгим подмножеством; строго включено
⊇		является надмножеством; включает
⊃		является строгим надмножеством; строго включает
Θ		оператор сравнения (=, < и т.д.); полярная координата
∅		пустое множество
→		функционально определяет
→→		определяет с помощью многозначной функциональной зависимости
≡		является эквивалентным; является тождественно равным
=>		следует(логическая связка)
⊢		следует (металингвистический символ)
⊨		всегда справедливо (металингвистический символ)

ПРИЛОЖЕНИЕ Г

Структуры хранения и методы доступа

Г1. ВВЕДЕНИЕ

В данном приложении приведен учебный обзор методов, обычно используемых в современных системах для физического представления и доступа к базе данных на диске. *(Примечание.* Во всем этом приложении термин *диск* применяется как общее обозначение для всех носителей информации с непосредственным доступом, включая, например, массивы RAID, запоминающие устройства большой емкости, оптические диски и т.д., а также обычные магнитные диски с подвижными головками как таковые.) Предполагается, что читатель имеет основное представление об архитектуре диска и знает, что подразумевается под терминами *время поиска, частота вращения, цилиндр, дорожка, головка чтения-записи* и т.д. Хорошие учебные руководства по этой теме можно легко найти; см., например, [Г.4].

Основная причина, побуждающая к постоянному совершенствованию всей технологии организации структур хранения и методов доступа, состоит в том, что характеристики доступа к диску намного хуже по сравнению с соответствующими характеристиками доступа к оперативной памяти. Как правило, значения времени поиска составляют порядка 5 или 6 мс, частоты вращения — до 10000 мин $^{-1}$, а скорость передачи данных обычно находится в пределах 5—10 Мбайт/с, поэтому, как правило, в любой конкретной системе обмен данными с оперативной памятью происходит по меньшей мере на четыре или пять порядков быстрее по сравнению с диском. Таким образом, наиболее важное направление повышения производительности состоит в **уменьшении до минимума количества операций доступа к диску** (или дисковых операций ввода—вывода). В данном приложении рассматриваются методы, способствующие достижению этой цели, иными словами, методы такого размещения данных на диске, чтобы любой необходимый фрагмент данных, скажем, некоторую конкретную запись можно было найти за минимально возможное количество операций ввода—вывода.

Примечание. Поскольку в данном приложении все описание ведется на уровне хранения данных, в нем вместо реляционной терминологии используется терминология, характерная для уровня хранения (в частности, такие термины, как *файл, запись* и *поле,* под которыми, соответственно, подразумеваются *хранимые* файл, запись и поле).

Как уже было указано, любое конкретное размещение данных на диске называется **структурой хранения.** Уже было предложено много различных структур хранения (и на этом все возможные варианты не исчерпываются), а разные структуры, безусловно, обладают различными характеристиками производительности; некоторые из них больше подходят для одних приложений, а другие — для других. По-видимому, невозможно найти единственную структуру, которая была бы оптимальной для всех возможных приложений. Из этого следует, что качественно спроектированная система должна поддерживать целый ряд различных структур, чтобы разные части базы данных можно было хранить различными способами, а также изменять структуру хранения для любой конкретной части базы по мере изменения требований **к** производительности или достижения лучшего понимания требований к хранению.

Настоящее приложение имеет следующую структуру. Вслед за этим вступительным разделом находится раздел Г.2, в котором кратко описано, в чем состоит весь процесс поиска и доступа к некоторой конкретной записи, а также указаны основные программные компоненты, участвующие в этом процессе. После этого в разделе Г.3 немного более подробно описаны два из таких компонентов — диспетчер файлов (file manager) и диспетчер диска (disk manager). Эти два раздела (Г.2 и Г.3) при первом чтении достаточно просто просмотреть; значительное количество содержащихся в них подробных сведений действительно не требуется для понимания последующего материала. Но следующие четыре раздела (Г.4—Г.7) необходимо изучить очень внимательно, поскольку они содержат наиболее важные сведения по всей обсуждаемой теме; в них описаны некоторые из структур хранения, наиболее широко применяемых в современных системах, соответственно, подзаголовками "Индексация", "Хэширование", "Цепочки указателей" и "Методы сжатия". Наконец, в разделе Г.8 приведено резюме и сделано краткое заключение.

Примечание. В данном приложении основное внимание уделено изложению концепций, а не деталей. Но оно предназначено для описания основных идей, лежащих в основе таких понятий, как *индексация, хэширование* и т.д., без лишнего углубления в конкретные особенности любой отдельно взятой системы или конкретного метода. Если же читателю необходимо ознакомиться с такими подробностями, он может воспользоваться книгами и статьями, которые перечислены в разделе "Список литературы" в конце данного приложения.

Г.2. ДОСТУП К БАЗЕ ДАННЫХ - КРАТКИЙ ОБЗОР

Прежде чем перейти к обсуждению структур хранения как таковых, необходимо вначале рассмотреть, в чем, вообще говоря, состоит весь процесс доступа к данным. В решении задачи поиска конкретного фрагмента данных в базе данных и передачи его пользователю участвует несколько различных уровней программного обеспечения. Безусловно, подробности устройства этих уровней в значительной степени зависят от конкретной системы (к тому же в разных системах часто применяется различная терминология), но используемые при этом принципы являются довольно стандартными, и эти принципы кратко описаны ниже (рис. Г.1).

1. Вначале СУБД определяет, какая ей требуется запись, и передает **диспетчеру файлов** запрос на вы борку этой записи. (В целях этого простого описания предполагается, что СУБД обладает спо собностью заблаговременно и точно определять, какая именно запись ей потребуется. На практике чаще всего возникает необходимость сделать выборку набора из нескольких записей и выполнить поиск среди этих записи в оперативной памяти, чтобы найти ту конкретную запись, которая действительно требуется. Но, в принципе, это означает лишь то, что последовательность шагов 1—3 иногда приходится повторять для каждой записи из этого набора.)
2. Диспетчер файлов в свою очередь определяет, какая страница содержит требуемую запись, и пе редает **диспетчеру диска** запрос на выборку этой страницы.
3. Наконец, диспетчер диска определяет физическое местонахождение желаемой страницы на дис ке и выдает необходимый запрос на выполнение операции ввода-вывода на диске.

Примечание. Безусловно, что иногда требуемая страница может уже находиться в буфере оперативной памяти в результате ранее выполненной операции выборки, и в этом случае, безусловно, необходимость повторно осуществлять ее выборку не возникает.

Поэтому, выражаясь неформально, СУБД обладает представлением о базе данных как о коллекции записей, и это представление поддерживается диспетчером файлов; диспетчер файлов, в свою очередь, обладает представлением **о** базе данных как о коллекции страниц, и это представление поддерживается диспетчером диска, а диспетчер диска обладает таким представлением о диске, "каким диск является в действительности". В следующих трех подразделах эти неформальные определения рассматриваются более подробно. Затем в разделе Г.3 по этим же темам приведена еще более подробная информация.

Рис. ГЛ. Общая схема взаимодействия СУБД, диспетчера файлов **и** диспетчера диска

Диспетчер диска

Диспетчер диска — это компонент базовой операционной системы. Он представляет собой компонент, отвечающий за все физические операции ввода—вывода (в некоторых системах его называют компонентом "основных служб ввода-вывода"). Как таковой, этот компонент, безусловно, должен обладать информацией об **адресах на физическом диске.** Например, после получения запроса от диспетчера файлов на выборку некоторой указанной страницы р диспетчер диска должен точно определить, где страница р находится на диске. С другой стороны, компонент, пользующийся услугами диспетчера диска (а именно диспетчер файлов) не должен обладать подобной информацией. Вместо этого в диспетчере файлов диск рассматривается просто как логическая коллекция **наборов страниц** (page set), где каждый набор страниц представляет собой коллекцию страниц постоянного размера. Каждый набор страниц обозначается уникальным **идентификатором набора страниц.** Каждая страница, в свою очередь, обозначается **номером страницы,** который является уникальным в пределах всего диска; различные наборы страниц являются непересекающимися (т.е. не имеют общих страниц). Отображение между номерами страниц и адресами на физическом диске поддерживается и сопровождается диспетчером диска. Основным преимуществом такой организации доступа (не единственно возможной) является то, что весь код, относящийся к конкретному устройству, можно изолировать в единственном компоненте

системы (а именно в диспетчере диска), в результате чего все компоненты более высокого уровня (в частности диспетчер файлов) могут стать независимыми от физического устройства.

Как уже было сказано, все множество страниц на диске разбивается на коллекцию непересекающихся подмножеств, называемых *наборами страниц.* Один из этих наборов страниц **(набор свободных страниц)** служит в качестве пула доступных (т.е. не используемых в настоящее время) страниц; все другие страницы рассматриваются как содержащие значимые данные. Включение страниц в наборы страниц и исключение страниц из этих наборов осуществляется диспетчером диска в ответ на запросы диспетчера файлов. Ниже перечислены некоторые операции над наборами страниц, поддерживаемые диспетчером диска (иными словами, операции, выполнения которых может потребовать диспетчер файлов).

- Выборка страницы p из набора страниц s.
- Замена страницы p в наборе страниц s.
- Добавление новой страницы к набору страниц s (т.е. получение пустой страницы из набора свободных страниц и определение номера новой страницы p).
- Удаление страницы p из набора страниц s (т.е. возвращение страницы p в набор свободных страниц).

Безусловно, первые две из этих операций являются наиболее важными операциями ввода—вывода страничного уровня, которые могут потребоваться диспетчеру файлов. Остальные две операции позволяют по мере необходимости увеличивать и уменьшать наборы страниц.

Диспетчер файлов

В диспетчере файлов только что описанные средства диспетчера диска применяются таким образом, чтобы в компоненте более высокого уровня, пользующемся его услугами (т.е. в СУБД), можно было рассматривать диск как коллекцию **файлов.** В таком случае каждый набор страниц состоит из файлов в количестве от нуля и больше.

Примечание. Для организации работы СУБД может потребоваться учитывать существование наборов страниц (даже несмотря на то, что СУБД не отвечает за управление ими во всех деталях), по причинам, которые указаны в следующем подразделе. В частности, для СУБД может потребоваться информация о том, что в двух файлах совместно используется один и тот же набор страниц или в двух записях совместно используется одна и та же страница.

Каждый файл обозначается **именем файла** или **идентификатором файла,** уникальным, по меньшей мере, в содержащем его наборе страниц, а каждая запись, в свою очередь, обозначается **номером записи** или **идентификатором записи** (Record I D — RID), уникальным, по меньшей мере, в содержащем его файле. (Обычно на практике идентификатор записи является уникальным не только в содержащем его файле, но фактически и на всем диске, поскольку он, как правило, состоит из комбинации номера страницы и некоторого значения, уникального в пределах этой страницы. См. раздел Г.3.)

Операции с файлами, поддерживаемые диспетчером файлов (т.е. операции, которые имеет возможность затребовать СУБД), включают перечисленные ниже.

- Выборка записи r из файла f.
- Замена записи r в файле f.
- Добавление новой записи в файл f и определение идентификатора новой записи r.
- Удаление записи r из файла f.
- Создание нового файла f.
- Уничтожение файла f.

Применение этих примитивных операций управления файлами позволяет в СУБД формировать и манипулировать структурами хранения, которые являются основной темой данного приложения (см. разделы Г.4—Г.7).

Примечание. В некоторых системах диспетчер файлов является компонентом базовой операционной системы; в других он поставляется вместе с СУБД. В целях данного описания такое различие не имеет особого значения. Но следует, в частности, отметить, что такой компонент обязательно входит в состав операционной системы; тем не менее, чаще всего диспетчер файлов общего назначения, предоставляемый

операционной системой, не совсем идеально соответствует требованиям такого "приложения" специального назначения, каким является СУБД. Дополнительная информация по этой теме приведена в [Г.44].

Кластеризация

Это обзорное описание не будет полным без краткого упоминания такой темы, как кластеризация данных. Основная идея, лежащая в основе кластеризации, заключается в том, что необходимо обеспечить хранение логически связанных записей (которые поэтому часто используются совместно) в непосредственной близости друг от друга на физическом диске. Физическая кластеризация данных играет исключительно важную роль с точки зрения производительности, в чем можно легко убедиться на следующем примере. Предположим, что последней по времени записью, к которой был выполнен доступ, является rl, а следующая требуемая запись— это r2. Допустим также, что rl хранится на странице pi, а r2 — на странице p2. В таком случае справедливы приведенные ниже утверждения.

1. Если pi и p2 представляют собой одну и ту же страницу, то для доступа к r2 вообще не потребу ется каких-либо физических операций ввода—вывода, поскольку необходимая страница p2 уже будет находиться в буфере оперативной памяти.
2. Если страницы p1 и p2 являются разными, но расположенными на физическом устройстве дос таточно близко друг к другу (в частности, если они являются физически смежными), то для дос тупа к r2 потребуется физическая операция ввода—вывода (безусловно, за исключением того случая, что p2 также будет находиться в буфере оперативной памяти), но время поиска, требуе мое в этой операции ввода—вывода, будет предельно сокращено, поскольку головки чтения-записи уже должны находиться недалеко от нужной позиции на диске. В частности, время пере хода с дорожки на дорожку будет равно нулю, если страницы p1 и p2 находятся на одном и том же цилиндре.

В качестве примера кластеризации рассмотрим обычно применяемую в данной книге базу данных поставщиков и деталей[1].

- Если в приложении часто возникает требование обеспечить последовательный доступ к инфор мации обо всех поставщиках в порядке номеров поставщиков, то записи поставщиков должны быть кластеризованы таким образом, чтобы запись поставщика S1 была физически расположена близко к записи поставщика S2, запись поставщика S2 — близко к записи поставщика S3 и т.д. Это— пример внутрифайловой кластеризации, т.е. кластеризации, применяемой в пределах одного файла.
- Если же, с другой стороны, в приложении часто возникает требование обеспечить доступ к данным некоторого конкретного поставщика наряду с данными обо всех поставках этого поставщика, то записи поставщиков и поставок должны храниться с чередованием таким образом, чтобы записи поставок для поставщика S1 были физически расположены близко к записи поставщика S1, записи поставок для поставщика S2 — близко к записи поставщика S2 и т.д. Это — пример междуфайловой кластеризации, т.е. кластеризации, которая применяется больше чем к одному файлу.

Безусловно, любой конкретный файл или набор файлов в любой конкретный момент времени может быть физически кластеризован не больше чем одним способом.

СУБД может поддерживать кластеризацию, как внутрифайловую, так и междуфайловую, путем организации хранения логически связанных записей на одной и той же странице, если это возможно, и на смежных страницах, если такой возможности нет (именно поэтому СУБД должна иметь информацию не только о файлах, но и о страницах). После создания в СУБД новой записи диспетчер файлов должен предоставить ей возможность указать, что новая запись должна храниться "поблизости" (т.е. на той же странице или, по крайней мере, на логически близко расположенной странице) от некоторой

---

[1] Во всем этом приложении для упрощения предполагается, что каждый отдельный кортеж с данными о поставщике, детали или поставке отображается на единственную запись, хранимую на диске, что обычно также предусмотрено в большинстве современных коммерческих систем. Дополнительная информация по этой теме приведена в приложении А.

существующей записи. Диспетчер диска в свою очередь должен сделать все от него зависящее, чтобы две логически смежные страницы были физически смежными и на диске (см. раздел Г.3).

Безусловно, как правило, СУБД будет иметь информацию о том, что требуется тот или иной способ кластеризации, только если эту информацию ей сможет передать администратор базы данных. Поэтому качественная СУБД должна позволять администратору базы данных задавать различные способы кластеризации для разных файлов. Она должна также обеспечивать смену способа кластеризации для конкретного файла или набора файлов после изменения требований к производительности. Кроме того, разумеется, любое такое изменение физической кластеризации не должно требовать внесения каких-либо явных и связанных с этим изменений в прикладные программы, поскольку в противном случае нарушается такое требование, как независимость отданных.

Г.3. НАБОРЫ СТРАНИЦ И ФАЙЛЫ

Как было описано в предыдущем разделе, основная задача диспетчера диска состоит в том, что он должен дать возможность диспетчеру файлов игнорировать все детали дисковых операций физического ввода-вывода и вместо этого действовать в рамках (логических) "операций ввода—вывода страниц". Указанную область функционирования диспетчера диска принято называть управлением страницами. Ниже приведен очень простой пример, позволяющий показать, как обычно осуществляется управление страницами.

Еще раз рассмотрим базу данных поставщиков и деталей. Предположим, что требуемое логическое упорядочение записей (неформально говоря) определяется последовательностью первичных ключей, иными словами, записи поставщиков должны быть упорядочены по номерам поставщиков, записи деталей — по номерам деталей, а записи поставок — по порядку номеров деталей, в пределах последовательности номеров поставщиков[2]. Для дальнейшего упрощения предположим также, что каждый файл хранится в виде отдельного набора страниц, а каждая запись находится на отдельной странице. Теперь рассмотрим описанную ниже последовательность событий.

1. Первоначально база данных вообще не содержит информации. Существует только один набор страниц (набор свободных страниц), который включает все страницы на диске, кроме нулевой страницы, выполняющей особую роль (как описано ниже). Остальные страницы пронумерованы последовательно, начиная от единицы.
2. Диспетчер файлов инициирует выполнение операции создания набора страниц для записей поставщиков и вставляет пять записей поставщиков, которые относятся к поставщикам S1-S5. Диспетчер диска удаляет страницы 1—5 из набора свободных страниц и отмечает их как "набор страниц поставщиков".
3. Аналогичные действия выполняются для размещения информации о деталях и поставках. Теперь существуют четыре набора страниц: набор страниц поставщиков (страницы 1—5), набор страниц деталей (страницы 6-11), набор страниц поставок (страницы 12—23) и набор свободных страниц (страницы 24, 25, 26 и т.д.). Ситуация, сложившаяся к этому времени, показана на рис. Г.2.

Продолжим описание данного примера.

4. Затем диспетчер файлов вставляет новую запись поставщика (с информацией о новом поставщике с номером S6). Диспетчер диска находит первую свободную страницу в наборе свободных страниц (а именно страницу 24) и добавляет ее к набору страниц поставщиков.
5. Диспетчер файлов удаляет запись с данными о поставщике S2. Диспетчер диска возвращает страницу поставщика S2 (страницу 2) в набор свободных страниц.

---

[2] Здесь употребляется оборот "неформально говоря", поскольку термин *последовательность первичных ключей* определен не полностью, если первичный ключ является составным. Например, применительно к данным о поставках он может означать либо упорядочение номеров деталей в пределах последовательности номеров поставщиков, либо упорядочение по противоположному принципу. (Так или иначе, первичные ключи — это реляционное понятие, а не понятие файлового уровня, поэтому в действительности, рассуждая о структурах хранения, мы вообще не должны были вести речь о первичных ключах.)

6. Диспетчер файлов вставляет новую запись детали (с данными о детали P7). Диспетчер диска находит первую свободную страницу в наборе свободных страниц (а именно страницу 2) и добавляет ее к набору страниц деталей.
7. Диспетчер файлов удаляет запись с данными о поставщике S4. Диспетчер диска возвращает страницу для поставщика S4 (страницу 4) в набор свободных страниц.

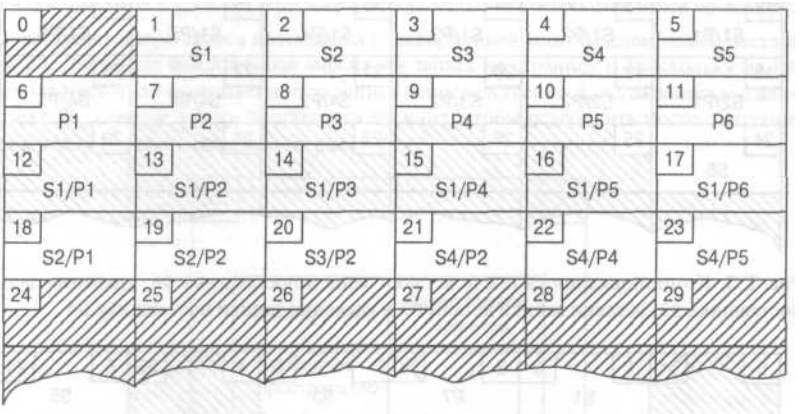


Рис. Г.2. Компоновка диска после создания и первоначальной загрузки базы данных поставщиков и деталей

В дальнейшем продолжаются аналогичные действия, а ситуация, сложившаяся к этому моменту, показана на рис. Г.3. Основной вывод, который может быть сделан после анализа этой ситуации, состоит в следующем: после того как система эксплуатировалась определенное время, она больше не может гарантировать, что логически смежные страницы будут по-прежнему физически смежными (даже если в самом начале они были таковыми). По этой причине логическая последовательность страниц в любом конкретном наборе страниц должна быть представлена не с помощью постоянной поддержки физической смежности, а с помощью указателей. Каждая страница должна содержать заголовок страницы, иными словами, набор управляющей информации, которая (кроме всего прочего) включает физический адрес на диске той страницы, которая непосредственно следует за этой страницей в логической последовательности (рис. Г.4).

На основании рассматриваемого примера можно сделать приведенные ниже выводы.

- Заголовками страниц (в частности, указателями "следующей страницы") управляет диспетчер диска; они должны быть полностью невидимыми для диспетчера файлов.
- Как было описано в подразделе о кластеризации в конце раздела Г.2, желательно хранить логически смежные страницы в физически смежных местах на диске (насколько это возможно). По этой причине диспетчер диска обычно включает и исключает страницы из наборов страниц не по одной, как было описано в данном примере, а скорее в виде физически смежных групп (или экстентов), что позволяет включать в набор одновременно (скажем) 64 страницы.
- Возникает резонный вопрос: "Как диспетчер диска определяет, где находятся различные наборы страниц?" или точнее: "Как он определяет для каждого набора страниц, где находится (логически) первая страница этого набора страниц?" (Безусловно, достаточно найти первую страницу, поскольку вторую и все другие страницы можно после этого обнаружить, следуя по указателям в заголовках страниц.) Ответ на этот вопрос состоит в том, что для хранения страницы, содержащей именно эту информацию, используется некоторое постоянное место на диске (как правило, нулевой цилиндр и нулевая дорожка). Поэтому такая страница (которая упоминается под разными названиями, такими как оглавление диска, каталог диска, каталог набора страниц или просто нулевая страница) обычно содержит список наборов страниц, существующих в настоящее время на диске, в котором имеются указатели на первую страницу каждого такого набора страниц (рис. Г.5).

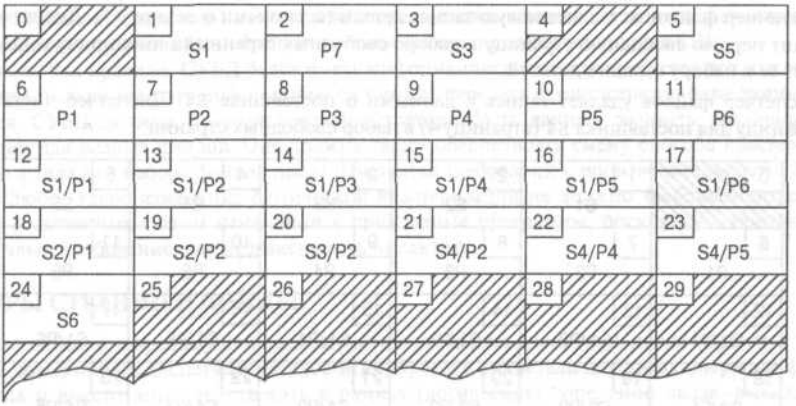


Рис. Г.3. Компоновка диска после вставки записи поставщика S6, удаления записи поставщика S2, вставки записи детали P7 и удаления записи поставщика S4

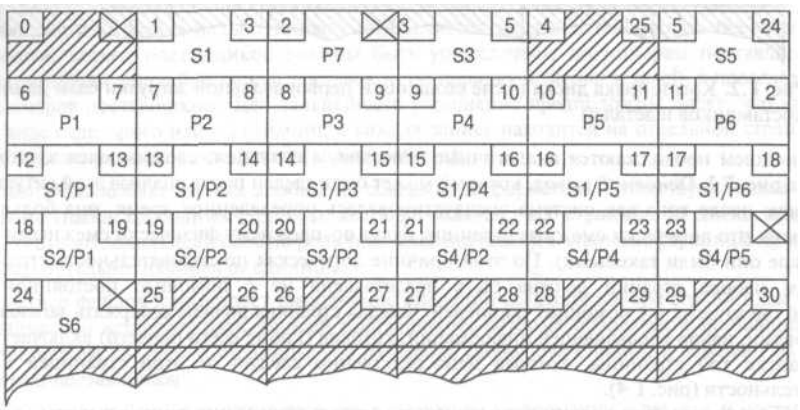


Рис. Г.4. Пересмотренный вариант рис. Г.3, на котором показаны указатели "следующей страницы" (в верхнем правом углу каждой страницы)

Теперь перейдем к описанию диспетчера файлов. Так же, как диспетчер диска позволяет диспетчеру файлов не учитывать нюансы физических операций ввода—вывода на диске и в основном функционировать на основе понятия логических страниц, так и диспетчер файлов позволяет СУБД игнорировать нюансы операций страничного ввода-вывода и функционировать на основе понятий файлов и записей. Такая функция диспетчера файлов называется **управлением записями.** Здесь эта функция рассматривается очень кратко и в качестве основы для примеров снова используется база данных поставщиков и деталей.

Итак, предположим, что на одной странице может находиться несколько записей, а не только одна, как было в примере управления страницами (и это предположение уже более реалистично). Допустим также, что, как и прежде, требуемое логическое упорядочение для записей поставщиков должно соответствовать возрастанию номеров поставщиков. Рассмотрим описанную ниже последовательность событий.

1. Прежде всего, выполняется вставка пяти записей с данными о поставщиках S1-S5, которые хранятся вместе на некоторой странице p, как показано на рис. Г.6. Обратите внимание на то, что страница p все еще содержит значительный объем свободного пространства.

2. Теперь допустим, что СУБД вставляет новую запись поставщика (с информацией о новом по ставщике, скажем, с номером S9). Диспетчер файлов вносит эту запись на страницу p (поскольку на ней еще есть свободное место) непосредственно вслед за записью с данными о по ставщике S5.
3. СУБД удаляет запись поставщика S2. Диспетчер файлов стирает запись S2 со страницы p и сдвигает записи поставщиков S3, S4, S5 и S9 вверх, чтобы заполнить освободившийся промежуток.
4. СУБД вставляет новую запись поставщика с информацией еще об одном новом поставщике, с но мером S7. Диспетчер файлов снова вносит эту запись на страницу p (поскольку на ней еще есть свободное место); он помещает новую запись непосредственно вслед за записью с данными о по ставщике S5, сдвигая запись поставщика S9 вниз, чтобы освободить место. Ситуация, сложив шаяся к этому времени, показана на рис. Г.7.

Рис. Г.5. Каталог диска ("нулевая страница")

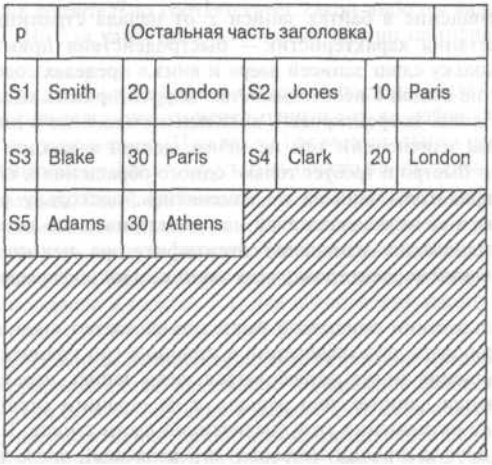


Рис. Г.6. Компоновка страницы p после первоначальной загрузки пяти записей с

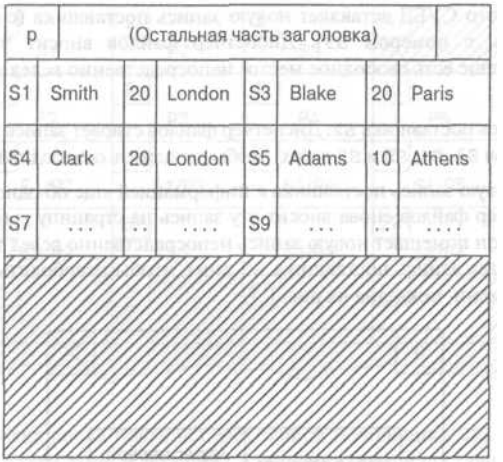


Рис. Г.7. Компоновка страницы p после вставки записи поставщика S9, удаления записи поставщика S2 и вставки записи поставщика S7

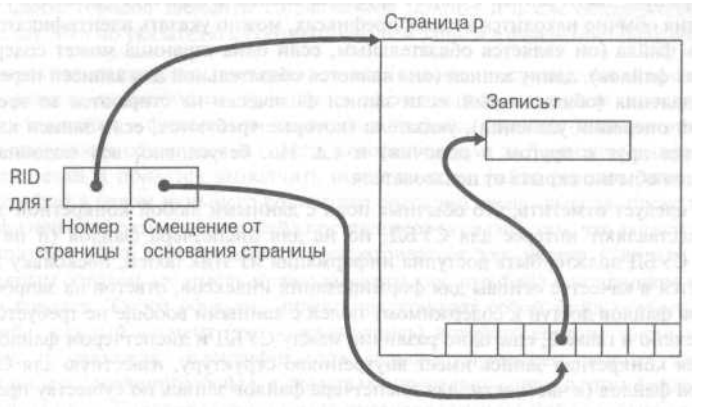
Аналогичные действия продолжаются и в дальнейшем. Основной вывод из сказанного состоит в том, что логическая последовательность записей на любой конкретной странице может быть представлена с помощью физической последовательности записей на этой странице; для достижения этого эффекта диспетчер файлов сдвигает записи вверх и вниз и обеспечивает то, что все записи с данными постоянно находятся в верхней части страницы, а все свободное пространство — в нижней его части. (Разумеется, что логическая последовательность записей, рассматриваемая с учетом не одной, а нескольких страниц, представлена путем упорядочения этих страниц в содержащем их наборе страниц, как было описано в приведенном выше примере управления страницами.)

Как было показано в разделе Г.2, в качестве внутреннего обозначения записей используются идентификаторы записей (Record ID — RID). На рис. Г.8 показано, как обычно реализованы идентификаторы записей. Идентификатор записи r состоит из двух частей: во-первых, номера страницы p, содержащей запись r, и, во-вторых, смещения в байтах от конца страницы p, обозначающего тот слот, который в свою очередь содержит смещение в байтах записи r от начала страницы p. Эта схема позволяет достичь приемлемого сочетания характеристик — быстродействия прямой адресации и гибкости косвенной адресации, поскольку сдвиг записей вверх и вниз в пределах содержащей их страницы (как показано на рис. Г.7 и Г.8) не связан с необходимостью корректировать идентификаторы записей (при каждом сдвиге приходится лишь корректировать локальные смещения в нижней части той страницы, где должны быть проведены изменения); тем не менее, доступ к каждой конкретной записи по ее идентификатору происходит быстро и требует только одного обращения к странице. (К тому же весьма желательно, чтобы идентификаторы записей не изменялись, поскольку они, как правило, широко используются в базе данных в качестве указателей на рассматриваемые записи, например, в индексах. Если же действительно происходит изменение идентификатора некоторой записи, то возникает необходимость выполнить также корректировку всех таких ссылок, оформленных в виде указателей, во всей базе данных.)

Примечание. При использовании описанной выше схемы доступ к какой-то конкретной записи в редких случаях может потребовать двух обращений к странице (но количество обращений никогда не превышает двух). Два обращения могут потребоваться, если запись переменной длины обновляется таким образом, что становится длиннее, чем раньше, а на странице отсутствует достаточный объем свободного пространства для размещения этой увеличенной части записи. В такой ситуации обновленная запись переносится на другую страницу (страницу переполнения), после чего первоначальная запись заменяется указателем (еще одним идентификатором записи) на новое место. Если же снова произойдет аналогичная ситуация, так что обновленную запись придется переместить со второй страницы на

Рис. Г.8. Принцип практической реализации идентификаторов записей

третью, то на первоначальной странице указатель будет откорректирован таким образом, чтобы он указывал на то место, где находится запись в настоящее время.



Теперь мы почти полностью готовы приступить к обсуждению структур хранения. В дальнейшем предполагается, что любой конкретный файл главным образом представляет собой просто коллекцию записей, каждая из которых обозначена уникальным идентификатором записи, никогда не изменяющимся, пока эта запись продолжает существовать (и аналогичные предположения обычно применяются практически во всех СУБД). Ниже приведено несколько заключительных замечаний, которые завершают тему данного раздела.

- Заслуживает внимания одно следствие из приведенного выше описания, которое состоит в том, что при использовании любого конкретного файла всегда возможно обеспечить доступ ко всем записям этого файла последовательно. Здесь под термином "последовательно" подразумевается "в последовательности записей, находящихся на последовательно упорядоченных страницах в наборе страниц" (как правило, в порядке возрастания идентификаторов страниц). Такую после довательность часто неформально называют физической последовательностью, хотя должно быть очевидно, что она не обязательно соответствует какой-либо очевидной физической последова тельности на диске. Но для удобства аналогичный термин применяется и в дальнейшем изло жении.
- Следует отметить, что доступ к файлу в физической последовательности возможен, даже если в нескольких файлах совместно используется один и тот же набор страниц (т.е. если страницы, занимаемые файлами, чередуются). При таком последовательном просмотре записи, не принад лежащие к рассматриваемому файлу, могут быть просто пропущены.
- Необходимо подчеркнуть, что физическая последовательность часто оказывается, по меньшей мере, вполне приемлемой для использования в качестве пути доступа к какому-то конкретному файлу. Иногда такой путь доступа может даже оказаться оптимальным (особенно если файл невелик). Но гораздо чаще встречается ситуация, когда требуется какой-то лучший способ доступа. А как было указано в разделе Г.1, для обеспечения такого "лучшего способа доступа" применяется чрезвычайно разнообразный набор методов.
- В целях упрощения до конца данного приложения обычно предполагается, что (уникальной) "физической" последовательностью для любого конкретного файла является последовательность первичных ключей (как было определено в разделе Г.3), если явно не указано иное. Но следует учитывать, что это предположение принято исключительно в целях упрощения приведенного ниже описания; автор признает, что на практике могут возникать серьезные основания для применения физического упорядочения данного конкретного файла в некоторой другой форме, например, по значению (значениям) некоторого другого поля (полей) или просто по времени поступления данных (в хронологической последовательности).

- Вполне вероятно, что по разным причинам кроме обычных полей с данными в запись придется включить определенную управляющую информацию. Такая информация обычно сосредотачив ается в передней части записи в форме **префикса записи.** В качестве примеров того, какого рода информация обычно находится в таких префиксах, можно указать идентификатор включающего эту запись файла (он является обязательным, если одна страница может содержать записи из нескольких файлов), длину записи (она является обязательной для записей переменной длины), флажок удаления (обязательный, если записи физически не стираются во время выполнения логической операции удаления), указатели (которые требуются, если записи каким-то образом соединяются друг с другом в цепочки) и т.д. Но, безусловно, вся подобная управляющая информация обычно скрыта от пользователя.
- Наконец, следует отметить, что обычные поля с данными любой конкретной записи, как пра вило, представляют интерес для СУБД, но не для диспетчера файлов (и не для диспетчера диска). В СУБД должна быть доступна информация из этих полей, поскольку эта информация используется в качестве основы для формирования индексов, ответов на запросы и т.д. Но для диспетчера файлов доступ к содержимому полей с данными вообще не требуется. Поэтому, как было отмечено в главе 2, еще одно различие между СУБД и диспетчером файлов состоит в том, что каждая конкретная запись имеет внутреннюю структуру, известную для СУБД, но не для диспетчера файлов (в частности, для диспетчера файлов запись по существу представляет собой просто строку байтов).

В оставшейся части этого приложения описаны некоторые из наиболее важных методов обеспечения "лучшего способа доступа" (т.е. создания пути доступа, лучшего по сравнению с физической последовательностью). Эти методы рассматриваются под общими заголовками "Индексация", "Хэширование", "Цепочки указателей" и "Методы сжатия". Приведем еще одно заключительное общее замечание: эти разнообразные методы не должны рассматриваться как взаимно исключающие друг друга. Например, вполне возможно применение такого файла, который допускает (скажем) и хэшированный, и индексированный доступ к нему на основе одного и того же поля или хэшированный доступ на основе одного поля и доступ с помощью цепочки указателей на основе другого поля.

Г.4. ИНДЕКСАЦИЯ

Еще раз рассмотрим данные о поставщиках. Предположим, что одним из самых важных (т.е. часто выполняемых и поэтому требующих высокой производительности) является запрос: "Определить всех поставщиков из города c" (где c — формальный параметр). С учетом такого требования администратор базы данных может выбрать хранимое представление, показанное на рис. Г.9. В этом представлении применяются два файла — файл поставщиков и файл городов (возможно, определенные в разных наборах страниц); файл городов, который предполагается хранить в последовательности городов (поскольку CITY — первичный ключ), включает указатели (идентификаторы записей) на файл поставщиков. Теперь, чтобы определить всех поставщиков (скажем) из Лондона, в СУБД можно применить одну из двух стратегий, описанных ниже.

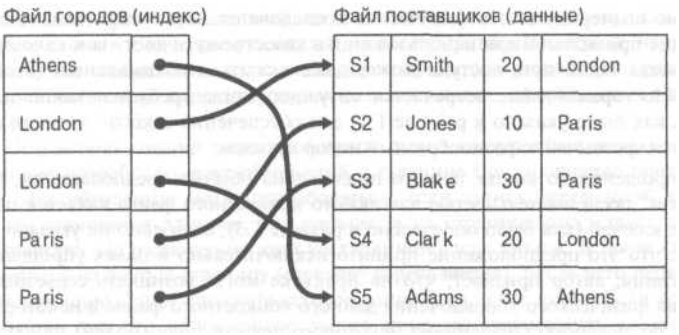


Рис. Г.9. Индексация файла поставщиков по атрибуту CITY

1. Выполнить поиск во всем файле поставщиков, отбирая те записи, в которых значение города
   равно London.
2. Найти в файле городов элементы со значением London и после обнаружения каждого такого элемента перейти по указателю к соответствующей записи в файле поставщиков.

Если относительное количество поставщиков из Лондона по сравнению с другими невелико, то вторая из этих стратегий, по-видимому, будет более эффективной, чем первая, поскольку, во-первых, в СУБД имеется информация о физическом упорядочении файла городов (и поиск в этом файле может быть прекращен сразу после обнаружения города, который следует за Лондоном в алфавитном порядке), и, во-вторых, даже если и придется выполнить поиск во всем файле городов, этот поиск все равно, скорее всего, потребует в целом меньшего количества операций ввода-вывода, поскольку файл городов имеет меньшие физические размеры, чем файл поставщиков (в связи с тем, что записи в нем короче).

В данном примере файл городов может рассматриваться как индекс ("индекс CITY") к файлу поставщиков; равным образом эту мысль можно выразить так, что файл поставщиков проиндексирован с помощью файла городов. Таким образом, индекс представляет собой файл особого рода. А именно, индекс — это файл, каждый элемент (т.е. каждая запись) которого состоит точно из двух значений: значения данных и указателя (идентификатора записи); значением данных является значение некоторого поля индексированного файла, а указатель определяет запись в этом файле, имеющую такое же значение этого же поля. Соответствующее поле индексированного файла называют индексированным полем или иногда *ключом индекса* (но автор не использует последний термин).

Примечание. Индексы получили такое название по аналогии с обычными индексами (предметными указателями), предусмотренными в книгах, которые также состоят из элементов, содержащих "указатели" (номера страниц), для облегчения процесса выборки информации из "индексированного файла" (т.е. из основного объема книги). Но следует отметить, что, в отличие от индекса CITY, показанного на рис. Г.9, предметные указатели являются иерархически сжатыми (т.е. элементы обычно содержат ссылки на несколько номеров страниц, а не только на один номер). См. раздел Г.7.

Дополнительная терминология. Индекс на первичном ключе (например, в случае поставщиков индекс на поле S#) иногда называют *первичным индексом.* Индекс на любом другом поле (т.е. в данном примере индекс CITY) иногда называют *вторичным индексом.* Кроме того, индекс на первичном ключе или, вообще говоря, на любом потенциальном ключе часто называют *уникальным индексом.*

Способы использования индексов

Фундаментальным преимуществом любого индекса по сравнению с другими путями доступа является то, что он ускоряет поиск. Но применение индексов связано также с определенным недостатком — они замедляют операции обновления. Например, после вставки каждой новой записи в индексированный файл необходимо также вводить новый элемент в индекс. В качестве более конкретного примера достаточно представить себе, какие действия СУБД должны выполнить над индексом CITY, показанным на рис. Г.9, если поставщик S2 переезжает, скажем, из Парижа в Лондон. Поэтому, вообще говоря, рассматривая возможность использования некоторого поля в качестве кандидата для индексации, необходимо прежде всего найти ответ на такой вопрос: "Что важнее, эффективная выборка на основе значений рассматриваемого иоля или издержки обновления, связанные с обеспечением такой эффективной выборки?".

В оставшейся части этого раздела речь пойдет именно об операциях выборки.

По сути, индексы могут использоваться двумя различными способами. Во-первых, с их помощью может осуществляться последовательный доступ к индексированному файлу (здесь под термином "последовательный" подразумевается "в последовательности, определяемой значениями индексированного поля"). Например, индекс CITY, показанный на рис. Г.9, позволяет обращаться к записям файла поставщиков в алфавитной последовательности названий городов. Во-вторых, индексы могут использоваться для прямого доступа к отдельным записям в индексированном файле с учетом конкретного значения индексированного поля. Этот второй случай можно проиллюстрировать на примере запроса: "Определить поставщиков из Лондона", который рассматривался в начале данного раздела.

Две только что высказанные основные идеи фактически могут быть немного обобщены, как описано ниже.

- *Последовательный доступ.* Индекс позволяет также упростить выполнение запросов с применением диапазона значений, например: "Определить поставщиков из городов, названия которых начинаются с букв, принадлежащих к указанному диапазону в алфавитном порядке" (допустим, начинаются с одной из букв в диапазоне L—R). При этом следует учитывать также два важных частных случая: а) "Определить всех поставщиков из городов, названия которых в алфавитном порядке предшествуют некоторому указанному значению (или следуют за этим значением)", и б) "Определить всех поставщиков из города, название которого находится на первом (или на последнем месте) в алфавитном порядке".
- *Прямой доступ.* Индекс позволяет также упростить выполнение запросов с применением списка, например: "Определить поставщиков из городов, названия которых указаны в некотором заданном списке" (допустим, Лондон, Париж и Нью-Йорк).

Кроме того, имеются некоторые запросы, например, с проверкой наличия данных (на которые можно получить ответ исключительно из индекса, вообще без какого-либо обращения к индексированному файлу. Предположим, что нужно найти ответ на запрос: "Имеются ли какие-либо поставщики в Афинах?". Ответ на этот запрос, безусловно, будет положительным тогда и только тогда, когда в индексе CITY имеется элемент, относящийся к Афинам. Аналогичные замечания относятся и к запросам, в которых используются определенные агрегирующие операторы. В качестве примера можно указать запрос: "Определить первый город в алфавитном порядке, где имеются поставщики"; в данном случае может применяться агрегирующий оператор MIN (напомним, что такая возможность рассматривалась в главе 18).

Каждый конкретный файл может иметь любое количество индексов. Например, для файла поставщиков можно задать и индекс CITY, и индекс STATUS (рис. Г. 10). После этого указанные индексы могут использоваться для обеспечения эффективного доступа к записям поставщиков на основе заданных значений либо CITY, либо STATUS, либо обоих значений. В качестве иллюстрации такого случая, в котором применяются оба значения, рассмотрим запрос: "Определить поставщиков из Парижа со статусом 30". Индекс CITY позволяет определить идентификаторы записей (скажем, г2 и г3) для поставщиков из Парижа; аналогичным образом, индекс STATUS указывает на идентификаторы записей (скажем, г3 и г5) для поставщиков со статусом 30. Сравнение этих двух множеств идентификаторов записей явно показывает, что единственным поставщиком, который соответствует первоначальному запросу, является поставщик с идентификатором записи, равным г3 (а именно поставщик S3). В самой СУБД доступ к файлу поставщиков можно выполнить только после проверки индексов, чтобы сразу же получить желаемую запись.

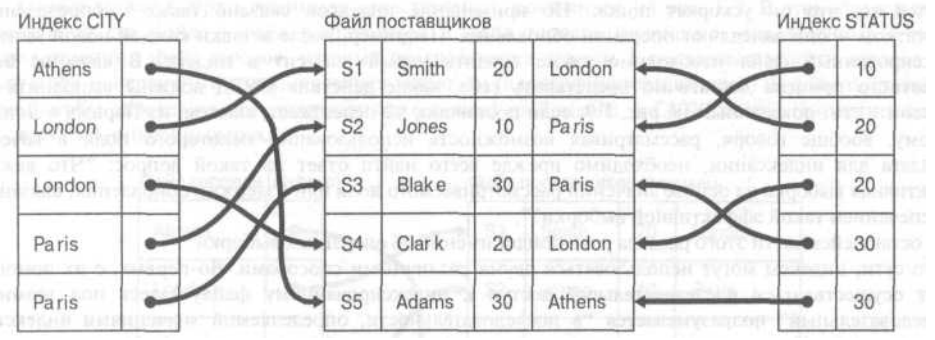


рис. **Г.10.** Индексация файла поставщиков по двум полям, CITY и STATUS

Дополнительная терминология. Индексы иногда называют **инвертированными списками** (inverted list) по следующей причине. Прежде всего, "обычный" файл (в этом смысле как типичный пример обычного файла может рассматриваться файл поставщиков, показанный на рис. Г.9 и Г.10), вообще говоря, содержит для каждой записи список значений полей данной записи. В отличие от этого, индекс для каждого значения индексированного поля содержит список записей, имеющих это значение. (Системы

баз данных с инвертированными списками, которые были кратко упомянуты в разделе 1.6 главы 1, получили свое название от указанного термина.) Приведем еще один важный термин — файл, в котором индекс имеется на каждом поле, иногда называют *полностью инвертированным* (fully inverted).

Индексация **с применением** комбинаций полей

Существует также возможность сформировать индекс на основе значений двух или нескольких полей, составляющих единую комбинацию. Например, на рис. Г. 11 показан индекс на файле поставщиков, в котором используется комбинация полей CITY и STATUS в указанном порядке. С применением такого индекса в СУБД можно получить ответ на запрос: "Определить поставщиков из Парижа со статусом 30" за один просмотр единственного индекса. А если бы этот комбинированный индекс был заменен двумя отдельными индексами, то для выполнения такого запроса потребовалось бы два отдельных просмотра индексов (как было описано выше). Кроме того, в данном случае может оказаться сложно определить, какой из этих двух просмотров следует выполнить в первую очередь; поскольку две возможные последовательности могут обладать весьма различными характеристиками производительности, такой выбор может оказаться весьма важным.

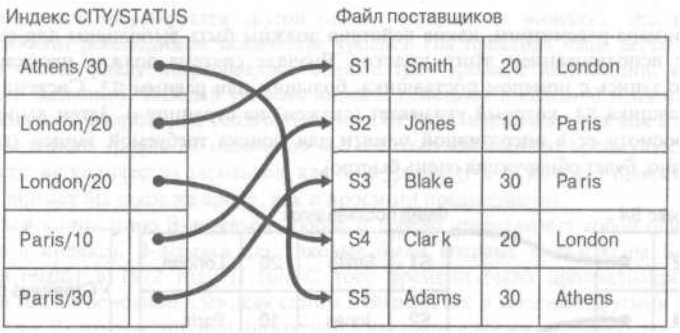


Рис. Г.11. Индексация файла поставщиков по комбинации полей CITY И STATUS

Кроме того, следует отметить, что комбинированный индекс CITY/STATUS может также служить в качестве индекса только к одному полю CITY, поскольку все записи, относящиеся к данному конкретному городу, по меньшей мере, продолжают оставаться последовательно расположенными и в этом комбинированном индексе. (Но если потребуется также индексация по полю STATUS, ТО необходимо будет предусмотреть еще один, отдельный индекс.) Вообще говоря, индекс на комбинации полей Fl, F2, F3, . . ., Fn (в указанном порядке) может также использоваться в качестве таких индексов — только на одном поле F1, на комбинации полей F1F2 (или F2F1), на комбинации полей F1F2F3 (в любом порядке) и т.д. Таким образом, общее количество индексов, которые требуются для обеспечения полной индексации с помощью указанного способа, не так уж велико, как может показаться на первый взгляд (см. упражнение Г.9 в конце этого приложения).

Плотная **и** неплотная индексация

Как уже было несколько раз отмечено, основное назначение любого индекса состоит в повышении быстродействия выборки, а именно, индекс предназначен для уменьшения количества дисковых операций ввода—вывода, необходимых для выборки некоторой указанной записи. По сути, эта задача выполняется с помощью указателей, а вплоть до этого момента предполагалось, что все такие указатели представляют собой указатели на записи (т.е. идентификаторы записей). Но фактически для достижения указанной цели достаточно, чтобы эти указатели представляли собой просто указатели на страницы (т.е. номера страниц). Верно, что после этого для поиска требуемой записи на указанной странице система должна выполнить некоторую дополнительную работу, которая заключается в том, что в оперативной памяти осуществляется поиск данных на странице, но количество операций ввода—вывода остается неизменным.

Примечание. В действительности, приведенная выше аналогия с предметным указателем книги представляет собой пример индекса, в котором указатели являются указателями на страницы, а не указателями на "записи" (под чем подразумеваются строки или слова).

Рассмотрим эту идею более подробно. Напомним, что любой конкретный файл имеет единственную "физическую" последовательность, представленную в виде комбинации, во-первых, последовательности записей на каждой странице и, во-вторых, последовательности страниц в содержащем этот файл наборе страниц. Предположим, что хранение файла поставщиков на диске организовано таким образом, что его физическая последовательность соответствует логической последовательности, которая определена с помощью значений некоторого поля, скажем, поля номеров поставщиков. Иными словами, допустим, что файл поставщиков кластеризован по этому полю (см. описание внутрифайловой кластеризации в конце раздела Г.2). Предположим также, что для этого поля требуется индекс. В таком случае нет необходимости, чтобы этот индекс содержал отдельный элемент для каждой записи в индексированном файле (т.е. в данном примере для каждой записи в файле поставщиков); требуется лишь предусмотреть для каждой страницы по одному элементу, в котором указан максимальный номер поставщика на этой странице и соответствующий номер страницы, как показано на рис. Г.12 (здесь для упрощения предполагается, что каждая конкретная страница может хранить не больше двух записей поставщиков).

В качестве примера рассмотрим, какие действия должны быть выполнены для выборки данных о поставщике S3 с использованием этого индекса. Вначале система должна просканировать индекс, отыскивая первую запись с номером поставщика, большим или равным S3. Система находит элемент индекса для поставщика S4, который указывает (скажем) на страницу p. Затем выполняется выборка страницы p и просмотр ее в оперативной памяти для поиска требуемой записи (которая в данном примере, безусловно, будет обнаружена очень быстро).

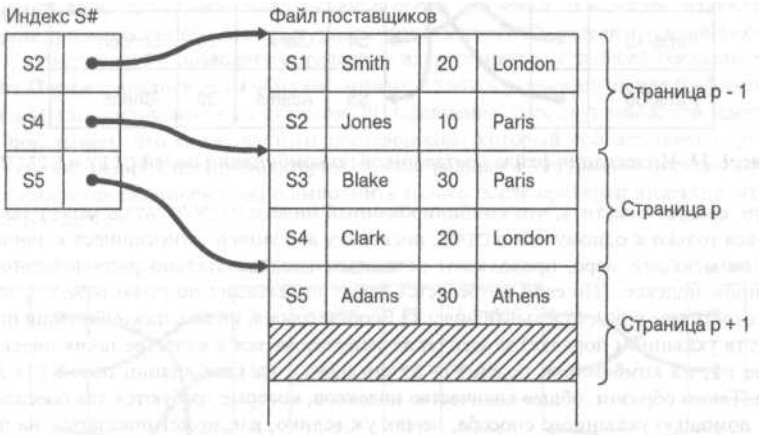


Рис. Г. 12. Пример неплотного индекса

Индекс, подобный приведенному на рис. Г.12, принято называть **неплотным—** nondense (или иногда **разреженным** — sparse), поскольку он не содержит по одному элементу для каждой записи индексированного файла. (В отличие от этого, все индексы, рассматриваемые в данном приложении, до этого момента были **плотными** — dense.) Одно из преимуществ неплотного индекса состоит в том, что он занимает меньше места по сравнению с соответствующим плотным индексом, по той очевидной причине, что он содержит меньше элементов. В результате просмотр этого индекса также, скорее всего, происходит быстрее. Недостатком такого индекса может оказаться то, что при его использовании больше нет возможности проводить проверки на наличие данных с применением лишь одного индекса (см. краткое замечание по поводу выполнения проверок на наличие данных в подразделе "Способы использования индексов" выше в этом разделе).

Вообще говоря, следует отметить, что любой конкретный файл может иметь не больше одного неплотного индекса, поскольку такой индекс основан на (уникальном) физическом упорядочении рассматриваемого файла. Все другие индексы обязательно должны быть плотными.

B-деревья

Наиболее широко применяемым и важным типом индекса являются **B-деревья** (в действительности, большинство реляционных систем поддерживают B-деревья в качестве основной формы структуры хранения, а некоторые системы вообще не поддерживают других форм хранения). Но прежде чем перейти к описанию того, что представляют собой B-деревья, необходимо вначале привести определение еще одного понятия, а именно понятия многоуровневого, или древовидного, индекса.

Основная причина применения любого индекса состоит в том, что он позволяет исключить необходимость в физическом последовательном просмотре индексированного файла. Но физический последовательный просмотр все еще требуется выполнять в индексе. Если индексированный файл становится очень большим, то и сам индекс может приобрести весьма значительные размеры, поэтому последовательный просмотр даже только одного индекса может потребовать довольно продолжительного времени. Решение этой проблемы является таким же, как и прежде — индекс рассматривается просто как обычный файл и на нем формируется другой индекс (индекс на индексе). Эта идея может быть осуществлена на любом необходимом количестве уровней (на практике чаще всего применяется три уровня; для того чтобы файлу потребовалось больше трех уровней индексации, он действительно должен стать очень большим). Каждый уровень индекса действует в качестве неплотного индекса для нижележащего уровня (разумеется, он обязательно должен быть неплотным, так как в противном случае использование еще одного уровня не позволило бы добиться каких-либо преимуществ — на уровне n находилось бы такое же количество элементов, как и на уровне n+1, и поэтому просмотр индекса более высокого уровня занимал бы такое же время, как и просмотр предыдущего).

Теперь перейдем к описанию B-деревьев. Любое B-дерево представляет собой один из конкретных типов древовидного индекса. B-деревья как таковые были впервые предложены Байером (Bayer) и Маккрейтом (McCreight) в 1972 году [Г.16]. С того времени было проанализировано множество вариантов одной и той же основной идеи, как самим Байером, так и многими другими исследователями; как уже было сказано, B-деревья того или иного типа в настоящее время, вероятно, составляют наиболее широко применяемую структуру хранения во всех современных системах баз данных. Ниже описан вариант, который рассматривается в работе Кнута (Knuth) [Г. 1]. Кстати, следует отметить, что структура индекса в методе виртуального доступа к памяти (Virtual Storage Access Method — VSAM) компании IBM [Г. 18] весьма напоминает структуру Кнута; тем не менее, версия VSAM была разработана независимо и включает собственные дополнительные средства, такие как использование методов сжатия. В действительности, вариант, предшествующий структуре VSAM, был описан еще в 1969 году [Г. 19].

В варианте Кнута, описанном ниже, индекс состоит из двух частей (для обозначения которых используется терминология VSAM) — *последовательного набора* (sequence set) и *индексного набора* (index set).

- **Последовательный набор** представляет собой одноуровневый индекс к фактическим данным; этот индекс обычно является плотным, но может быть и неплотным, если индексированный файл кластеризован по индексированному полю. Элементы этого индекса (безусловно) сгруппи рованы в страницы, а страницы (безусловно) соединены цепочками указателей таким образом, что логическое упорядочение, представленное индексом, можно восстановить, считывая записи в физической последовательности на первой странице в цепочке, затем — записи в физической последовательности на второй странице в цепочке и т.д. Таким образом, последовательный набор обеспечивает быстрый последовательный доступ к индексированным данным.
- **Индексный набор,** в свою очередь, дает возможность получить быстрый прямой доступ к после довательному набору (и таким образом также и к самим данным). Индексный набор фактически представляет собой древовидный индекс к последовательному набору; в действительности, строго говоря, настоящим B-деревом является именно индексный набор. Комбинация индекс ного набора и последовательного набора иногда именуется "B*-деревом". Верхний уровень индексного набора состоит из единственного узла (т.е. из одной страницы, но, безусловно, содержащей много элементов индекса, как и все другие узлы). Верхний узел индекса называется **корнем.**

На рис. Г. 13 приведен простой пример. Этот рисунок можно пояснить следующим образом. Прежде всего, значения 6, 8, 12, ..., 97, 99— это значения индексированного поля, скажем, F. Рассмотрим верхний узел, который состоит из двух значений F (50 и 82) и трех указателей (фактически номеров страниц). Записи данных со значением F, меньшим или равным 50, можно найти (в конечном итоге), проследовав по левому указателю из этого узла, аналогичным образом, записи со значением F, большим чем 50 и меньшим или равным 82, можно найти, следуя по среднему указателю, а записи со значением F, большим чем 82, можно получить, следуя по правому указателю. Другие узлы в этом индексном наборе интерпретируются аналогичным образом; обратите внимание на то, что (например), перейдя по правому указателю из первого узла на второй узел, можно получить все записи со значением F, большим чем 32, а также меньшим или равным 50 (в силу того факта, что мы уже перешли по левому указателю из узла более высокого уровня).

Тем не менее, B-дерево (т.е. индексный набор), показанное на рис. Г. 13, не совсем реалистично по описанным ниже причинам.

- Во-первых, все узлы B-дерева обычно не содержат одинаковое количество значений данных.
- Во вторых, в этих узлах, как правило, находится определенный объем свободного пространства.

Вообще говоря, любое "B-дерево порядка n" имеет не меньше n и не больше 2n значений данных в любом конкретном узле (и если в узле находится к значений данных, то имеется также к+1 указатель). Ни одно значение данных не встречается в дереве больше одного раза. Ниже приведен алгоритм поиска конкретного значения V в структуре, показанной на рис. Г.13 (см. стр. 1334); алгоритм для B*-дерева общего вида порядка n представляет собой просто обобщение данного алгоритма.

```
set N to the root node ;
do until N is a sequence-set node ;
   let X, Y (X < Y) be the data values in node N ;
   if V < X     then set N to the left   lower node of N ;
   if X < V < Y then set N to the middle lower node of N ;
   if V > Y     then set N to the right  lower node of N ;
end ;
if V occurs in node N then exit /* найдено */ ;       if V does
not occur in node N then exit /* не найдено */ ;
```

Выполнение операций вставки и удаления может привести к тому, что дерево станет несбалансированным, и в этом состоит важная проблема, связанная с использованием любых древовидных структур в целом. Дерево является несбалансированным, если не все листовые узлы находятся на одном и том же уровне, иными словами, если различные листовые узлы находятся на разных расстояниях от корневого узла. Поскольку в процессе поиска в дереве доступ к диску требуется для перехода к каждому узлу, в несбалансированном дереве такой показатель, как время поиска, становится почти непредсказуемым.

Примечание. На практике верхний уровень индекса (а также чаще всего отдельные фрагменты других уровней) большую часть времени хранится в оперативной памяти, что позволяет в конечном итоге уменьшить среднее количество операций доступа к диску. Но указанный основной недостаток несбалансированных деревьев тем самым не устраняется.

В отличие от этого, замечательным преимуществом B-деревьев является то, что алгоритмы вставки и удаления гарантируют, что дерево всегда остается сбалансированным. (По этой причине иногда приходится слышать, что аббревиатура "B" в термине "B-дерево" обозначает "balanced" — сбалансированный.) Ниже кратко рассматривается процедура вставки нового значения, скажем, V, в B-дерево порядка n. В применяемом при этом алгоритме рассматривается только индексный набор, поскольку (как было описано выше) собственно B-деревом является именно индексный набор; чтобы распространить действие этого алгоритма также на последовательный набор, требуется лишь простейшее дополнение.

- Вначале выполняется алгоритм поиска для обнаружения не узла последовательного набора, а того узла (скажем, N) на самом нижнем уровне индексного набора, к которому логически относится значение V. Если узел N содержит свободное пространство, то значение V вставляется в N и процесс завершается.

- В противном случае узел N (который таким образом содержит 2n значений) разделяется **на два** узла, N1 и N2. Допустим, что S — ряд первоначальных 2n значений наряду с новым значением V в их логической последовательности. Наименьшие n значений из ряда S помещаются в левый узел, N1, наибольшие n значений из ряда S помещаются в правый узел, N2, а среднее значение, скажем, W, продвигается в родительский узел узла N, скажем, P, для использования в качестве разделительного значения для узлов N1 и N2. В дальнейшем **при** поиске значения и **этот поиск** после достижения узла P будет направлен в узел N1, если U < W, **и в** узел N2, если и > W.
- Теперь предпринимается попытка вставить значение W **в** узел P **и** описанный процесс повто ряется.

В наихудшем случае разделение узлов происходит вплоть до вершины дерева, создается новый корневой узел (родительский по отношению к старому корневому узлу, который теперь разделился на два узла) и дерево растет в высоту на один уровень (но даже в этом случае остается сбалансированным).

Разумеется, алгоритм удаления по существу является обратным по отношению к только **что** описанному алгоритму вставки. Изменение значения осуществляется **путем удаления старого значения и** вставки нового.

Г.5. ХЭШИРОВАНИЕ

Хэширование (называемое также **хэш-адресацией,** а иногда, не совсем правильно, **хэш-индексацией)** представляет собой метод обеспечения быстрого прямого доступа к конкретной записи с учетом заданного значения определенного поля. Рассматриваемое поле обычно (но не обязательно) является первичным ключом. Ниже приведено краткое описание метода хэширования.

- Каждая запись помещается в базу данных в такое место, адрес которого (т.е. идентификатор записи или, возможно, просто номер страницы) вычисляется так некоторая функция (назы ваемая **хэш-функцией)** от значения определенного поля этой записи (называемое *хэшированным полем* — hash field, или иногда *хэшированным ключом* — hash key; но последний термин в данном приложении не используется). Адрес, вычисленный с помощью хэш-функции, называется **хэши рованным адресом.**
- Для первоначального сохранения рассматриваемой записи СУБД вычисляет хэшированный адрес для новой записи и передает диспетчеру файлов команду на размещение этой записи по указанному адресу.
- Для последующей выборки записи с использованием значения хэшированного поля СУБД проводит такие же вычисления, как и прежде, и выдает диспетчеру файлов команду чтения/записи по вычисленному адресу.

В качестве простого примера предположим, что значения номеров поставщиков составляют S100, S200, S300, S400, S500 (а не SI, S2, S3, S4, S5) и для каждой записи поставщика требуется целая отдельная страница, а также рассмотрим следующую хэш-функцию:

```
хэшированный адрес (т.е. номер страницы) = остаток после деления
числовой части значения S# на 13
```

Это — простейший пример очень широко применяемого класса хэш-функций, называемых **функциями хэширования с использованием остатка от деления,** или просто *хэш-функциями "деление/остаток"* — "division/remainder" hash function. (По причинам, описание которых выходит за рамки данного приложения, в качестве делителя в хэш-функции "деление/остаток" обычно используется простое число, как в данном примере.) Таким образом, номера страниц для этих пяти поставщиков, соответственно, принимают значения 9, 5, 1, 10, 6, в результате чего создается отображение между номерами поставщиком и номерами страниц, показанное на рис. Г. 14.

Из приведенного выше описания должно быть очевидно, что хэширование отличается от индексации по следующему важному признаку — в то время как любой конкретный файл может иметь любое количество индексов, для него может быть предусмотрено не больше одной хэшированной структуры. Иначе эту мысль можно выразить таким образом — файл может иметь любое количество индексированных полей, но только одно хэшированное поле. (Данное замечание относится к методу

прямого хэширования — direct hashing. В отличие от этого, с файлом может быть связано произвольное количество структур *косвенного хэширования* — indirect hash. См. [Г.24].)

Рассматриваемый пример показывает не только принципы работы метода хэширования, но и позволяет понять, почему требуется хэш-функция. Было бы теоретически возможно использовать "идентичную" хэш-функцию, иными словами, в качестве хэшированного адреса непосредственно применять значение первичного ключа (при этом, безусловно, подразумевается, что первичный ключ является числовым). Но такой метод обычно не может использоваться на практике, поскольку диапазон возможных значений первичного ключа чаще всего намного превышает диапазон доступных адресов. Например, предположим, что фактически номера поставщиков находятся в пределах от S000 до S999, как показано в приведенном выше примере. В таком случае количество возможных различных номеров поставщиков будет равно 1000, тогда как в действительности количество действующих поставщиков приблизительно равно 10. Поэтому для предотвращения значительных непроизводительных затрат пространства памяти в идеале следует найти такую хэш-функцию, которая будет преобразовывать любое значение в диапазоне 000—999 в одно значение (скажем) в диапазоне 0-9. Но для создания небольшого резерва для будущего роста обычно принято расширять целевой диапазон адресов примерно на 20%; именно поэтому в данном примере выбрана функция, которая вырабатывает значения в диапазоне 0-12, а не 0-9.

Этот пример также показывает один из недостатков хэширования: "физическая последовательность" записей в файле почти наверняка не будет совпадать с последовательностью первичных ключей, а также, разумеется, с любой другой последовательностью, которая имеет какую-либо осмысленную логическую интерпретацию. (Кроме того, между подряд идущими записями могут находиться промежутки произвольных размеров.) В действительности, физическая последовательность записей в файле с хэшированной структурой чаще всего (но не всегда) рассматривается как не представляющая какой-либо определенной логической последовательности.

Примечание. Безусловно, всегда возможно наложить любую желаемую логическую последовательность на хэшированный файл с помощью индекса; а в действительности, возможно наложить несколько таких последовательностей с помощью нескольких индексов, по одному на каждую такую последовательность. См. также [Г.35] и [Г.37], где рассматриваются возможности создания схем хэширования, сохраняющих логическую последовательность записей в хранимом файле.

Еще одним недостатком метода хэширования в целом является то, что всегда остается возможность **коллизий,** т.е. появления двух или нескольких разных записей ("синонимов"), хэширование которых происходит по одному и тому же адресу. Например, предположим, что файл поставщиков (с данными о поставщиках S100, S200 и т.д.) включает также поставщика с номером S1400. При использовании хэш-функции, рассматриваемой в данном примере ("поделить на 13 и взять остаток"), запись этого поставщика будет конфликтовать с записью поставщика S100, находящейся по хэшированному адресу 9. Таким образом, очевидно, что хэш-функция в том виде, в котором она здесь описана, не обеспечивает нормальной работы и должна быть каким-то образом дополнена, чтобы можно было справиться с проблемой коллизий.

В терминах данного первоначального примера одним возможным дополнением является то, что остаток от деления на 13 должен рассматриваться не как хэшированный адрес как таковой, а скорее как отправная точка для последовательного просмотра. Таким образом, чтобы вставить запись поставщика S1400 (при условии, что записи поставщиков S100—S500 уже существуют), необходимо перейти на страницу 9 и выполнить поиск вперед от этой позиции до первой свободной страницы. Допустим, что новая запись поставщика будет сохранена на странице 11. В дальнейшем для выборки данных об этом поставщике необходимо пройти аналогичную процедуру. Такой метод, называемый **линейным поиском** (linear search), может оказаться вполне приемлемым, если на каждой странице хранится немного записей (как обычно и бывает на практике). Предположим, что на каждой странице может находиться п записей. В таком случае все п первых записей с одним и тем же хэшированным адресом р, попавших в коллизию, будут сохранены на странице р, а линейный поиск среди этих записей будет полностью ограничиваться одной этой страницей. Но следующая, (п+1)-я коллизия, безусловно, приведет к тому, что соответствующую запись придется сохранить на какой-то отдельной **странице переполнения** и потребуется еще одна операция ввода—вывода.

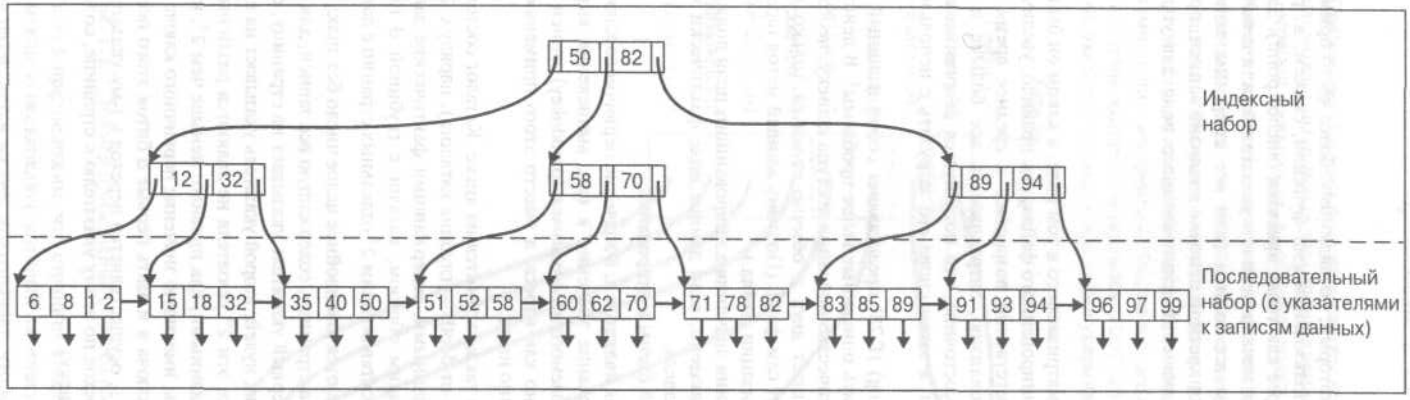


Рис. Г.13. Часть простого B-дерева (вариант Кнута)

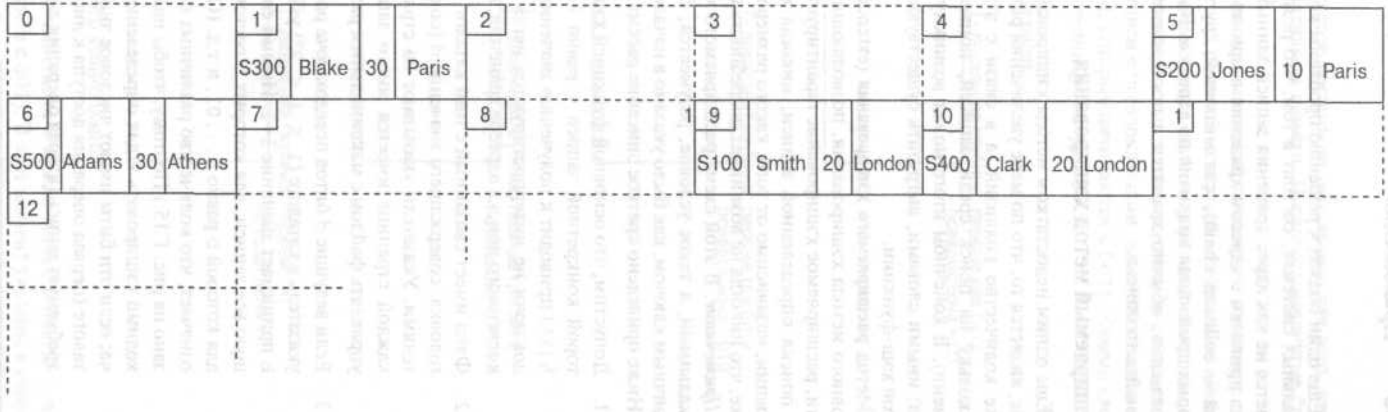


Рис. Г.14. Пример хэшированной структуры

Еще один подход к решению проблемы коллизий, который, по-видимому, более часто применяется в реальных системах, состоит в том, что результат использования хэш-функции, скажем, а, рассматривается не как адрес хранения записи данных, но скорее как **точка привязки** (anchor point). Затем эта точка привязки с адресом хранения а применяется как начало цепочки указателей (или **цепочки коллизий** — collision chain), связывающей друг с другом все записи (или все страницы записей), в которых произошла коллизия по адресу а. В каждой конкретной цепочке коллизий записи, попавшие в коллизию, обычно хранятся в последовательности значений хэшированного поля для упрощения дальнейшего поиска.

Расширяемый метод хэширования

Еще одним недостатком метода хэширования, рассматриваемого в том виде, в каком он был описан выше, является то, что по мере увеличения размеров хэшированного файла, как правило, увеличивается также количество коллизий, а в связи с этим соответственно возрастает и среднее время доступа (поскольку на поиск среди записей, попавших в коллизию, затрачивается все больше и больше времени). В конечном итоге может возникнуть такое состояние, что потребуется реорганизовать этот файл; иными словами, выгрузить существующий файл, а затем снова его загрузить с использованием другой хэш-функции.

Метод **расширяемого хэширования** (extendable hashing) [Г.28] представляет собой изящный вариант основного метода хэширования, позволяющий устранить описанные выше проблемы[3]. В действительности, расширяемое хэширование гарантирует, что количество операций доступа к диску, необходимых для поиска определенной записи, никогда не превышает двух и обычно сводится только к одной операции, независимо от того, какого размера достигает сам файл. (Поэтому данный метод гарантирует также, что никогда не возникнет потребность в реорганизации файла.)

Примечание. В этой схеме расширяемого хэширования значения хэшированного поля должны быть уникальными, а такое условие, разумеется, будет соблюдено, если данное поле фактически является первичным ключом, как было указано в начале этого раздела.

Ниже приведено краткое описание рассматриваемой схемы хэширования.

1. Допустим, что основной функцией хэширования является h, а значение первичного ключа неко торой конкретной записи г равно к. Хэширование значения к (т.е. вычисление выражения h (к)) приводит к получению значения з, называемого *псевдоключом* (pseudokey) записи к. Псевдоключи не интерпретируются непосредственно как адреса, а вместо этого применяются для косвенного поиска адресов хранения, как описано ниже.
2. Файл имеет связанный с ним каталог, который также хранится на диске. Каталог состоит из за головка, содержащего значение d (сокращение от depth — глубина каталога), наряду с 2^d указа телями. Указатели указывают на страницы с данными, содержащими фактические записи (на каждой странице имеется много записей). Таким образом, каталог с глубиной d позволяет управлять файлом с максимальным размером, составляющим 2^d отдельных страниц с данными.
3. Если ведущие d битов псевдоключа рассматриваются как двойное целое число без знака Ь, то i-й указатель в каталоге ($1 < i < 2^d$) указывает на страницу, содержащую все записи, для которых b принимает значение i-l. Иными словами, первый указатель указывает на страницу, содержа щую все записи, для которых b состоит из одних нулей, второй указатель указывает на страницу, **для** которой b равно 0… 01, и т.д. (Обычно все эти 2^d указателя не являются различными; это означает, что количество различных страниц сданными, как правило, меньше чем 2^d, как пока зано на рис. Г. 15.) Поэтому, чтобы найти запись, имеющую значение первичного ключа к, необ ходимо хэшировать к для определения псевдоключа s и взять первые d битов этого псевдоклю ча; если эти биты имеют числовое значение i-l, осуществляется переход к i-му указателю в ка талоге (первая операция доступа к диску) и переход по этому указателю к странице, содержащей требуемую запись (вторая операция доступа к диску).

---

[3] В [Г.28] применяется англоязычное написание термина "extendable" (расширяемый) с буквой i, т.е. "extendible".

Примечание. На практике этот каталог обычно остается достаточно небольшим для того, чтобы его можно было почти все время хранить в оперативной памяти. Поэтому упомянутые "две" операции доступа к диску обычно сводятся к одной.

4. Каждая страница с данными имеет также заголовок, указывающий локальную глубину p этой страницы ($p < d$). Предположим, например, что d равно 3 и первый указатель в каталоге (указатель 000) указывает на страницу, для которой локальная глубина p равна 2. В данном случае локальная глубина 2 означает, что эта страница содержит не только все записи с псевдоключами, начинающимися с 000, но и содержит все записи с псевдоключами, начинающимися с 00 (т.е. теми, которые начинаются с 000, а также теми, которые начинаются с 001). Иными словами, указатель каталога 001 также указывает на эту страницу (см. рис. Г. 15).

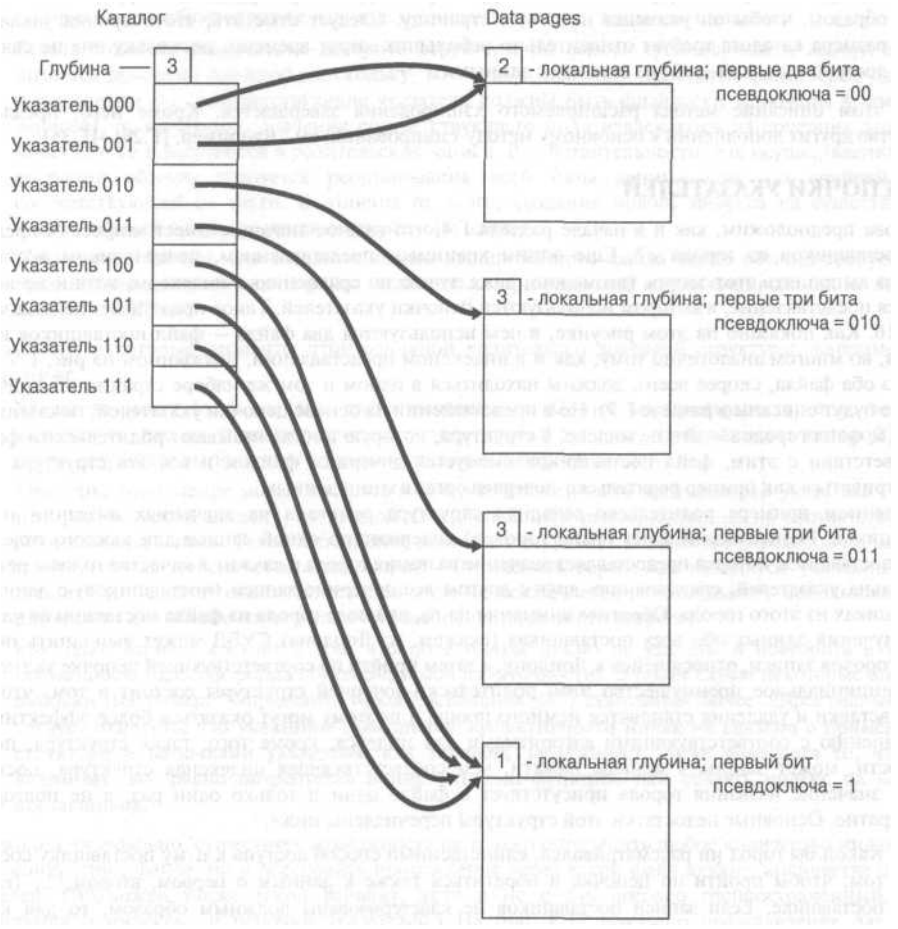


Рис. Г. 15. Пример применения способа расширяемого хэширования

5. Теперь предположим, что страница с данными 000 заполнена и нужно вставить новую запись, имеющую псевдоключ, который начинается с 000 (или с 001). В этот момент указанная страница разделяется на две; это означает, что из пула свободных страниц берется новая, пустая страница, а все записи 001 изымаются из старой страницы и переносятся в новую. Указатель 001 в

каталоге корректируется таким образом, чтобы он указывал на новую страницу (указатель 000 все еще указывает на старую страницу). Локальная глубина p для каждой из этих двух страниц теперь будет равна 3, а не 2.

6. Теперь допустим, что страница с данными, соответствующая начальной строке битов 000, снова заполняется и должна быть опять разделена. Существующий каталог не позволяет обеспечить такое разделение, поскольку локальная глубина разделяемой страницы уже равна глубине каталога. Поэтому размер каталога удваивается; это означает, что значение d увеличивается на единицу и каждый указатель заменяется парой смежных, идентичных указателей. Теперь появляется возможность разделить страницу с данными; записи 0000 остаются на старой странице, а записи 0001 перемещаются на новую страницу; первый указатель в каталоге остается неизменным (т.е. по-прежнему указывает на старую страницу), а второй указатель корректируется таким образом, чтобы он указывал на новую страницу. Следует отметить, что операция удваивания размера каталога требует относительно небольших затрат времени, поскольку она не связана с доступом к какой-либо из страниц с данными.

На этом описание метода расширяемого хэширования завершается. Кроме него, предложено множество других дополнений к основному методу хэширования; см., например, [Г.29]—[Г.36].

Г.6. ЦЕПОЧКИ УКАЗАТЕЛЕЙ

Снова предположим, как и в начале раздела Г.4, что важное значение имеет запрос: "Определить всех поставщиков из города c". Еще одним хранимым представлением, позволяющим достаточно успешно выполнять этот запрос (возможно, даже лучше по сравнению с индексом, хотя и не всегда), является представление, в котором используются цепочки указателей. Такое представление показано на рис. Г. 16. Как показано на этом рисунке, в нем используются два файла — файл поставщиков и файл городов, во многом аналогично тому, как и в индексном представлении, показанном на рис. Г.9 (но на этот раз оба файла, скорее всего, должны находиться в одном и том же наборе страниц по причинам, которые будут описаны в разделе Г.7). Но в представлении на основе цепочки указателей, показанном на рис. Г. 16, файл городов — это не индекс, а структура, которую иногда называют родительским файлом. В соответствии с этим, файл поставщиков именуется дочерним файлом и вся эта структура может рассматриваться как пример родительско-дочерней организации данных.

В данном примере родительско-дочерняя структура основана на значениях названий городов поставщиков. Родительский файл (файл городов) содержит по одной записи для каждого отдельного города поставщика, которая предоставляет значение названия города и служит в качестве головы цепочки, или кольца указателей, связывающих друг с другом все дочерние записи (поставщиков) с данными о поставщиках из этого города. Обратите внимание на то, что поле города из файла поставщиков удалено; для получения данных обо всех поставщиках (скажем, из Лондона) СУБД может выполнить поиск в файле городов записи, относящейся к Лондону, а затем пройти по соответствующей цепочке указателей.

Принципиальное преимущество этой родительско-дочерней структуры состоит в том, что алгоритмы вставки и удаления становятся немного проще и поэтому могут оказаться более эффективными по сравнению с соответствующими алгоритмами для индекса; кроме того, такая структура, по всей видимости, может занимать меньше памяти, чем соответствующая индексная структура, поскольку каждое значение названия города присутствует в файле один и только один раз, а не повторяется многократно. Основные недостатки этой структуры перечислены ниже.

- Какой бы город ни рассматривался, единственный способ доступа к n-му поставщику состоит в том, чтобы пройти по цепочке и обратиться также к данным о первом, втором, ..., (n-1)-м поставщике. Если записи поставщиков не кластеризованы должным образом, то для каждой операции доступа потребуется выполнить отдельную операцию поиска на диске, и время, требуемое для доступа к n-му поставщику, может оказаться довольно значительным.
- Хотя эта структура может оказаться подходящей для запроса: "Определить поставщиков из указанного города", она не способствует (фактически может даже воспрепятствовать) успешному выполнению противоположного запроса: "Определить название города, где находится указанный поставщик" (где поставщик указан с помощью номера поставщика). Для выполнения последнего запроса, по-видимому, потребуется хэширование или индексирование файла

поставщиков; следует отметить, что теперь родительско-дочерняя структура, основанная на номерах поставщиков, не будет иметь какого-либо смысла (объясните, почему).

- И даже если будет найдена запись конкретного поставщика, все равно не отпадет необходимость пройти по цепочке к родительской записи, чтобы узнать требуемое название города (тем самым подтверждается приведенное выше замечание, что такая родительско-дочерняя структура фактически препятствует успешному выполнению запросов этого класса, поскольку при ее использовании требуется указанный дополнительный шаг).
- Кроме того, следует отметить, что для родительского файла (файла городов) может также потребоваться индексированный или хэшированный доступ, если этот файл имеет достаточно значительные размеры. Поэтому цепочки указателей, отдельно взятые, фактически не могут служить приемлемой основой для создания каких-либо структур хранения, поскольку почти наверняка потребуются также и другие механизмы доступа, такие как индексы.
- Задача создания родительско-дочерней структуры на основе существующего набора записей является довольно сложной, поскольку, во-первых, цепочки указателей фактически проходят через записи (т.е. соответствующие указатели должны быть физически включены в префиксы записей), и, во-вторых, значения соответствующего поля извлекаются из дочерних записей и вместо этого помещаются в родительские записи. В действительности, для осуществления такой операции обычно требуется реорганизация всей базы данных или, по крайней мере, соответствующей ее части. В отличие от этого, создание нового индекса на существующем наборе записей осуществляется относительно просто.
- *Примечание.* Кстати, для создания нового хэшированного файла также обычно требуется ре организация существующего файла, если только применяемый метод хэширования не является косвенным [Г.24].

Возможны некоторые варианты этой основной родительско-дочерней структуры, например, как описано ниже.

- Указатели могут быть сделаны двунаправленными. Одним из преимуществ этого варианта является то, что он упрощает процесс корректировки записей, необходимость которой возни кает в результате выполнения операции удаления дочерней записи.
- Еще одно дополнение может состоять в том, что должен быть предусмотрен указатель ("указа тель родительской записи") от каждой дочерней записи непосредственно на соответствующую родительскую запись. Такое дополнение позволяет сократить количество переходов по цепочке, требуемых для поиска ответа на запрос: "Определить город, где находится указанный пос тавщик" (но следует отметить, что при этом не исключается необходимость использовать хэш или индекс для обеспечения успешного выполнения данного запроса).
- Еще один вариант состоит в том, что поле города нужно не удалить, а повторить в записях поставщиков (простая форма контролируемой избыточности). В таком случае некоторые операции выборки (например: "Определить город поставщика S4") становятся более эффективными. Но следует отметить, что указанное повышение эффективности никак не связано с применением структуры с цепочками указателей как таковой; заслуживает также внимания то, что, по-видимому, все равно потребуется применить хэш-функцию или создать индекс на номерах поставщиков.

Наконец, безусловно, существует возможность не только определить любое количество индексов на любом конкретном файле, но и обеспечить прохождение через такой файл любого количества цепочек указателей. (Возможен также такой вариант, хотя и не очень широко распространенный, когда применяются и индексы, и цепочки указателей.) На рис. Г. 17 показано представление для файла поставщиков, в котором используются две различные цепочки указателей, и поэтому определены две разные родительско-дочерние структуры, в одной из которых в качестве родительского используется файл городов (как на рис. Г. 16), а в другой — файл статуса. Файл поставщиков является дочерним для обеих этих структур.

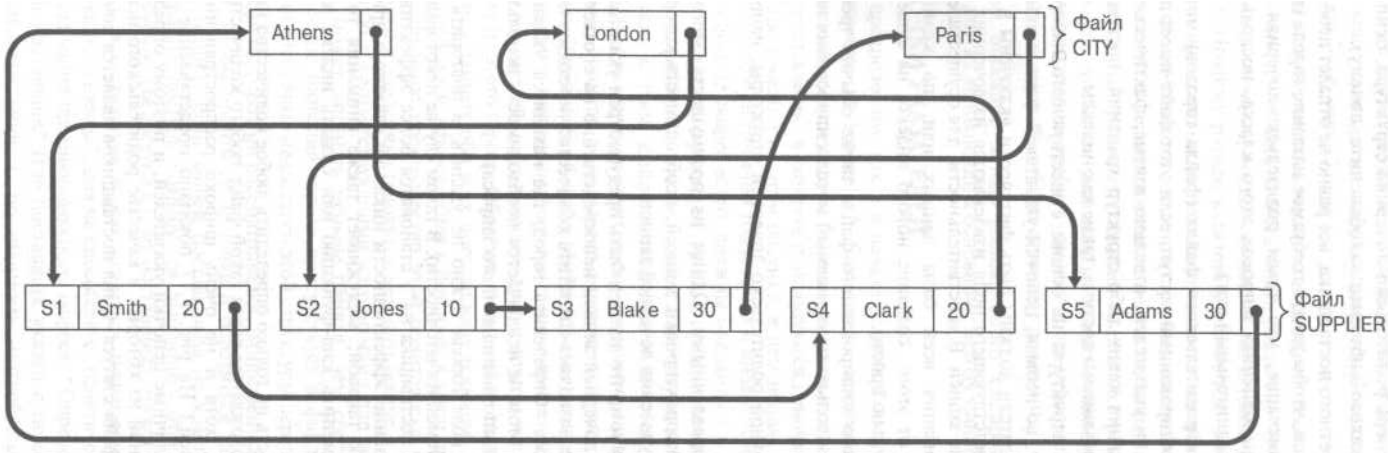


Рис. Г.16. Пример родительско-дочерней структуры

Г.7. МЕТОДЫ СЖАТИЯ

Методы сжатия используются для уменьшения объема памяти, необходимого для хранения определенной коллекции данных. Очень часто результатом такого сжатия становится не только экономия пространства памяти, но и сокращение количества операций ввода-вывода на диске (причем, возможно, еще более значительное по сравнению с экономией памяти). Дело в том, что если данные занимают меньше места, то для доступа к ним требуется меньше операций ввода-вывода. С другой стороны, требуется дополнительная обработка для восстановления данных (преобразования сжатых данных в исходный формат) после их выборки. Но в конечном итоге достигнутое сокращение количества операций ввода—вывода чаще всего перевешивает недостатки такой дополнительной обработки.

В основе методов сжатия лежит тот факт, что значения данных почти никогда не бывают полностью случайными и характеризуются определенной степенью предсказуемости. В качестве простейшего примера можно указать, что если имя некоторого лица в файле имен и адресов начинается с буквы R, то весьма вероятно, что имя следующего лица также будет начинаться с буквы R, разумеется, при условии, что файл отсортирован по именам в алфавитном порядке.

Поэтому один из широко применяемых методов сжатия состоит в том, что каждое отдельное значение данных заменяется некоторым обозначением различия между ним и непосредственно предшествующим ему значением. В этом заключается метод **дифференциального сжатия** (differential compression). Но следует отметить, что для успешного применения такого метода требуется, чтобы доступ к рассматриваемым данным осуществлялся последовательно, поскольку, чтобы распаковать любое конкретное значение, необходимо знать непосредственно предшествующее ему хранимое значение. Поэтому метод дифференциального сжатия главным образом применяется в таких ситуациях, когда доступ к данным должен всегда осуществляться последовательно, как в случае (например) использования элементов одноуровневого индекса. Однако необходимо учитывать, что именно в случае индекса может быть предусмотрено сжатие не только данных, но и указателей, поскольку, если логическое упорядочение данных, которым характеризуется индекс, является таким же или близким к физическому упорядочению файла с данными, то подряд идущие значения указателей в индексе будут весьма аналогичными друг другу, в связи с чем может оказаться выгодным и сжатие указателей. В действительности, индексы почти всегда позволяют добиться определенного выигрыша благодаря использованию сжатия, по меньшей мере, применительно к данным, если не к указателям.

Чтобы продемонстрировать применение метода дифференциального сжатия, оставим на время пример с поставщиками и деталями и рассмотрим страницу с записями из "индекса" с фамилиями служащих. Предположим, что первые четыре элемента на этой странице относятся к следующим служащим.

```
Roberton
Robertson
Robertstone
Robinson
```

Допустим также, что поле фамилий служащих имеет длину 12 символов, поэтому каждая из этих фамилий должна рассматриваться (в распакованном виде) как дополненная справа соответствующим количеством пробелов. Один из способов применения дифференциального сжатия к этому множеству значений состоит в том, что символы в начале каждого элемента, совпадающие с символами в предыдущем элементе индекса, заменяются числом, указывающим количество совпадающих символов. Такой метод сжатия называется **префиксным сжатием** (front compression). В результате применения этого метода к указанным выше данным, будет получено следующее (теперь заключительные пробелы показаны явно с помощью знаков " + ").

```
0 - Roberton++++
6 - son+++
7 - tone+
3 - inson++++
```

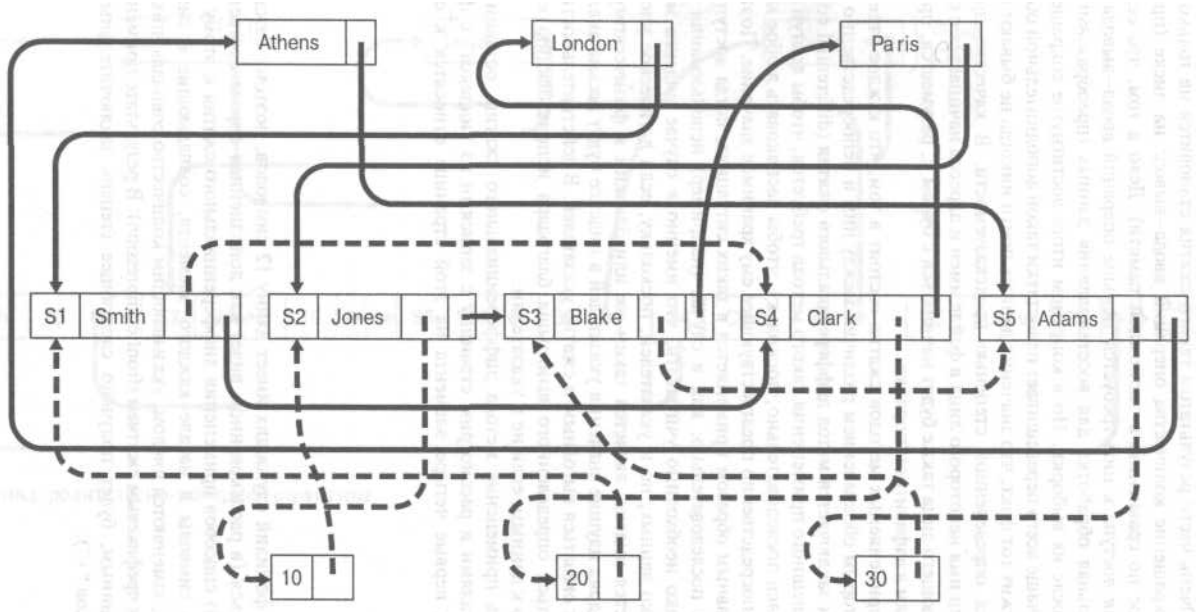


Рис. Г.17. Еще один пример родительско-дочерней структуры

Еще один возможный метод сжатия для этого набора данных состоит в том, что могут быть просто удалены все заключительные пробелы (при том условии, что они снова заменяются числом, указывающим количество удаленных пробелов). Такой метод представляет собой пример **суффиксного сжатия** (rear compression). Эффективность суффиксного сжатия можно дополнительно повысить, удалив все символы справа от того, который требуется, чтобы отличить рассматриваемый элемент от двух его непосредственных соседей, как показано ниже.

```
O - 7  -  Roberto
6 - 2  -  so
7 - 1  -  t
3 - 1  -  i
```

В данном случае первое из двух чисел в каждом элементе остается таким же, как и в предыдущем примере, а второе представляет собой количество зарегистрированных символов (предполагается, что Следующий элемент после его восстановления не будет иметь подстроку "Robi" в качестве своих первых четырех символов). Но следует отметить, что в этом индексе фактически потеряна некоторая информация. Поэтому после его восстановления он будет выглядеть следующим образом (где "?" обозначает неизвестный символ).

```
Roberto????
?
Robertso???
?
Robertst???
?
Robi???????
?
```

Безусловно, что такая потеря информации допустима, только если данные полностью зарегистрированы где-то в другом месте (в данном примере — в соответствующем файле служащих).

Иерархические методы сжатия

Предположим, что рассматриваемый файл упорядочен физически (т.е. кластеризован) по значениям некоторого поля F, и допустим также, что каждое отдельное значение F встречается в нескольких подряд идущих записях этого файла. Например, файл поставщиков может быть кластеризован по значениям поля города; в этом случае будут храниться вместе данные обо всех поставщиках из Лондона, из Парижа и т.д. В такой ситуации может оказаться, что множество всех записей поставщиков, относящихся к конкретному городу, целесообразно представить в сжатом виде как одну **иерархическую** запись, в которой название рассматриваемого города представлено только один раз, а за ним следует информация о номере, имени и статусе для каждого поставщика, который в данное время находится в этом городе (рис. Г. 18).

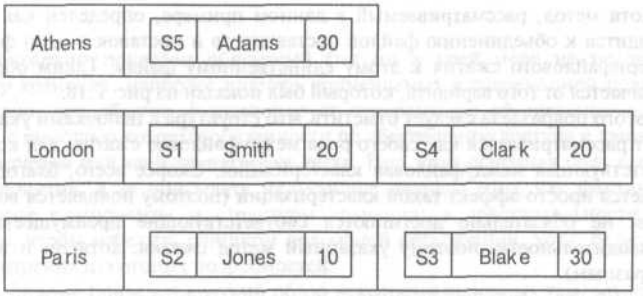


Рис. Г. 18. Пример иерархического сжатия (внутрифайлового)

Записи, показанные на рис. Г.18, состоят из двух частей — из постоянной части (а именно поля города) и переменной части (множества элементов с данными о поставщиках).

Примечание. Последняя часть называется *переменной,* поскольку количество содержащихся в ней элементов (т.е. количество поставщиков в рассматриваемом городе) зависит от конкретной записи. Как было указано в главе 6, такое множество элементов **с** изменяющимся количеством в записи иногда

называют *повторяющейся группой*. Таким образом, можно утверждать, что иерархические записи, приведенные на рис. Г. 18, состоят из одного поля города и повторяющейся группы с информацией о поставщиках, а информация о поставщиках в свою очередь состоит из поля номера поставщика, поля имени поставщика и поля статуса поставщика (по одной такой группе полей для каждого поставщика из соответствующего города).

Иерархическое сжатие по принципу, описанному выше, чаше всего является наиболее подходящим для индекса, если он характеризуется тем, что в нем целый ряд подряд идущих элементов имеет одинаковое значение данных (но, безусловно, разные значения указателей).

Из указанного выше следует, что иерархическое сжатие в той форме, которая здесь описана, применимо, только если файл организован по принципу внутрифайловой кластеризации. Но, как уже мог догадаться читатель, аналогичного рода сжатие может также применяться и в случае междуфайловой кластеризации. Предположим, что кластеризованы данные о поставщиках и поставках, как было указано в конце раздела Г.2; это означает, что данные о поставках поставщика S1 непосредственно следуют за записью с данными о поставщике S1, данные о поставках поставщика S2 — за данными о поставщике S2 и т.д. А именно, предположим, что информация о поставщике S1 и поставках поставщика S1 хранится на странице pi, информация о поставщике S2 и поставках поставщика S2 — на странице p2 и т.д. В таком случае может быть применен метод междуфайлового сжатия, как показано на рис. Г. 19.

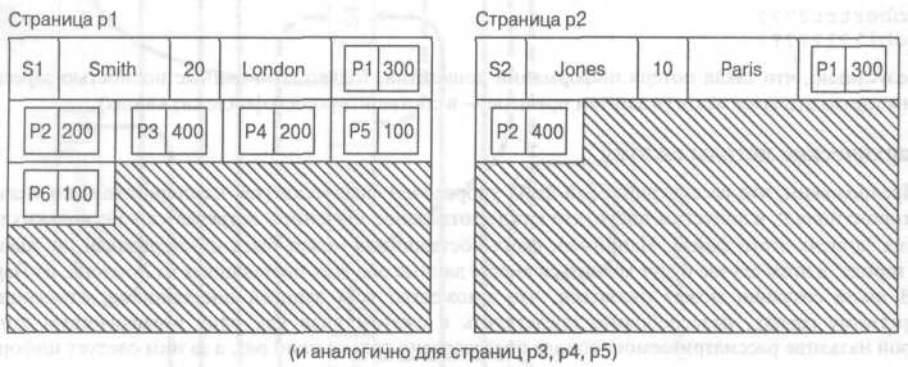


Рис. Г. 19. Пример иерархического сжатия (междуфайлового)

Примечание. Хотя метод, рассматриваемый в данном примере, определен как "междуфайловый", фактически он сводится к объединению файлов поставщиков и поставок в один файл с последующим применением внутрифайлового сжатия к этому единственному файлу. Таким образом, этот вариант фактически не отличается от того варианта, который был показан на рис. Г. 18.

В завершение этого подраздела следует отметить, что структура с цепочками указателей, показанная на рис. Г. 16, может рассматриваться как своего рода междуфайловое сжатие, для которого не требуется какая-либо соответствующая междуфайловая кластеризация. Скорее всего, благодаря использованию указателей достигается просто эффект такой кластеризации (поэтому появляется возможность сжатия), но наряду с этим не обязательно достигаются соответствующие преимущества с точки зрения физической производительности, поэтому указанный метод сжатия, хотя он и возможен, не всегда является целесообразным).

Кодирование по методу Хаффмена

Кодирование по методу Хаффмена (Huffman coding) [Г.39] представляет собой метод символьного кодирования, который позволяет в принципе достичь значительной степени сжатия данных, если в них различные символы встречаются с разными частотами (что, безусловно, обычно и происходит на практике), хотя этот метод редко применяется в современных системах. Основная его идея состоит в следующем: используется кодирование с применением битовых строк для представления символов

таким образом, что разные символы заменяются битовыми строками различной длины, причем наиболее часто встречающиеся символы заменяются самыми короткими строками. Кроме того, ни одному символу не поставлен в соответствие такой код (длиной, скажем, п битов), что эти п битов идентичны первым п битам кода какого-то другого символа.

В качестве простого примера предположим, что в кодируемых данных встречаются только символы A, в, C, D и E, а также допустим, что относительная частота появления этих пяти символов характеризуется приведенной ниже таблицей.

Символ	Частота	Код
E	35%	1
A	30%	01
D	20%	001
C	10%	0001
B	5%	0000

Символ E имеет максимальную частоту и поэтому ему присваивается самый короткий код, состоящий из одного бита, скажем, бита 1. Поэтому все другие коды должны начинаться с бита 0 и иметь длину не меньше 2 битов (единственный бит 0 нельзя использовать в качестве кода, поскольку его невозможно будет отличить от начальной части любого другого кода). Символу A присваивается код с длиной на единицу больше по сравнению с самым коротким кодом, скажем, 01, поэтому все другие коды должны начинаться с 00. Аналогичным образом, символам D, C и в присваиваются коды, соответственно, 001, 0001 и 0000.

Упражнение. Какие английские слова закодированы в приведенных ниже строках?

```
00110001010011
010001000110011
```

При использовании приведенных здесь значений кодов ожидаемая средняя длина закодированного символа в битах выражается следующим значением.

```
0,35 * 1 + 0,30 * 2 + 0,20 * 3 + 0,10 * 4 + 0,05 * 4 = 2,15 битов
```

Если бы каждому символу было назначено одинаковое количество битов, как в обычной схеме кодирования символов, то потребовалось бы 3 бита в расчете на символ (чтобы можно было охватить все эти пять вариантов).

Г.8. РЕЗЮМЕ

В данном приложении приведен обширный (но ни в коей мере не исчерпывающий!) обзор некоторых структур хранения, наиболее широко применяемых в современной практике. В нем также было кратко описано, как обычно функционирует программное обеспечение доступа к данным и показаны способы, с помощью которых обязанности по обеспечению доступа к данным распределяются между СУБД, диспетчером файлов и диспетчером диска. При этом основная цель состояла в том, чтобы объяснить общие понятия, а не описывать мельчайшие нюансы того, как фактически применяются различные системные компоненты и структуры хранения; в действительности, автор сделал все возможное для того, чтобы избежать слишком подробной детализации, хотя, безусловно, нельзя было обойтись без рассмотрения некоторых подробностей.

Ниже в данном резюме приведен краткий обзор некоторых основных тем, затронутых в настоящем приложении. Прежде всего, был описан метод кластеризации, основная идея которого состоит в том, что записи, часто используемые совместно, должны храниться физически достаточно близко друг от друга. В нем также показано, что в качестве внутреннего обозначения записей широко применяются идентификаторы записей (Record ID — RID). Затем были описаны наиболее важные структуры хранения, используемые на практике.

- **Индексы** (и некоторые их варианты, включая, в частности, **В-деревья)** и их использование как для последовательного, так и для прямого доступа.
- Методы **хэширования** (включая, в частности, **расширяемое** хэширование) и их применение для **прямого** доступа.
- **Цепочки указателей** (называемые также **родительско-дочерними структурами)** и их многочисленные варианты.

В данном приложении был также описан целый ряд методов **сжатия.**

В заключение следует еще раз подчеркнуть, что для большинства пользователей этот материал чаще всего не представляет (или не должен представлять) особого интереса. Единственным "пользователем", который должен достаточно подробно изучить эти сведения, является администратор базы данных, отвечающий за физическое проектирование базы данных, контроль и настройку производительности. Что касается остальных пользователей, то желательно, чтобы в процессе работы у них не возникало потребности в ознакомлении с изложенным здесь материалом, хотя и нельзя не отметить, что пользователи лучше справляются со своими обязанностями, если имеют определенное представление о том, как организовано внутреннее функционирование системы. С другой стороны, что касается разработчиков СУБД, то для них, безусловно, становится желательным или даже обязательным овладение изложенным здесь материалом (к тому же знания по этой теме должны быть **у** данных специалистов гораздо более глубокими).

УПРАЖНЕНИЯ

Упражнения Г. 1—Г.8 могут оказаться подходящими для использования в качестве основы для обсуждения в учебной группе; их выполнение должно привести к более глубокому пониманию различных сторон физического проектирования базы данных. Упражнения Г.9 и Г. 10 в большей степени имеют теоретическую (математическую) направленность.

ГЛ. Исследуйте все системы базы данных, которые могут быть вам доступны (чем больше, тем лучше). В каждой такой системе выделите компоненты, выполняющие функции, которые в данном приложении отнесены к сфере действия, соответственно, диспетчера диска, диспетчера файлов и самой СУБД. Какие диски или другие носители данных поддерживает эта система? Каковы размеры страницы? Какова емкость диска, как теоретическая (в байтах), так и фактическая (в страницах)? Каковы скорости передачи данных? Чему равны значения времени доступа? Как выглядят эти значения времени доступа по сравнению с быстродействием оперативной памяти? Есть ли какие-либо пределы размера базы данных или размера файла? Если да, то каковыми они являются? Какую из структур хранения, описанных в этом приложении, поддерживает система? Поддерживает ли она какие-либо другие структуры хранения? Если да, то какие?

Г.2. База данных отдела кадров компании должна содержать информацию об отделениях, отделах и служащих этой компании. Каждый служащий работает в одном отделе; каждый отдел входит в состав одного отделения. Подготовьте некоторые примеры данных и кратко опишите некоторые возможные структуры хранения, которые подходят для этих данных. Там, где это возможно, отметьте относительные преимущества каждой из таких структур (т.е. в каждом случае опишите, как с их помощью можно успешно выполнять типичные операции поиска и обновления).

Подсказка. Ограничения "каждый служащий работает в одном отделе" и "каждый отдел входит в состав одного отделения" по своей структуре аналогичны ограничению "каждый поставщик находится в одном городе" (все они являются примерами связей "многие к одному"). Различие между ними состоит в том, что в рассматриваемой базе данных, по-видимому, потребуется хранить более подробную информацию об отделах и отделениях, чем о городах.

Г.З. Повторно выполните упр. Г.2 применительно к базе данных, которая должна содержать информацию о клиентах и товарах. Каждый клиент может заказать любое количество товаров; каждый товар может присутствовать в заказах любого количества клиентов.

Подсказка. В данном случае проявляется связь "многие ко многим" между клиентами и товарами. Один из способов представления такой связи состоит в использовании *двойного индекса.* Двойным индексом называется индекс, который используется для индексации сразу двух файлов

данных; каждый конкретный элемент в таком индексе соответствует паре связанных записей данных, по одной для каждого из этих двух файлов, и содержит два значения данных и два указателя. Можете ли вы предложить какие-либо иные способы представления связей "многие ко многим (и ко многим ...)"?

Г.4. Повторно выполните упр. Г.2 применительно к базе данных, которая должна содержать информацию о деталях и узлах, где каждый узел рассматривается как отдельная деталь и может состоять из узлов (деталей) более низкого уровня.

Подсказка. Проанализируйте, в чем это задание отличается от задания, приведенного в упр. Г.3.

Г.5. Файл с записями данных без дополнительной структуры доступа иногда называют кучей (heap). Новые записи вставляют в кучу там, где обнаруживается свободное место. Для небольших файлов (таковыми являются любые файлы, для которых на устройстве хранения данных требуется не больше чем, скажем, девяти или десяти страниц) куча, по-видимому, представляет собой наиболее подходящую структуру хранения. Но основная часть файлов имеет гораздо более значительные размеры, и на практике все файлы, кроме наименьших, должны иметь некоторую дополнительную структуру доступа, такую как (по меньшей мере) индекс на первичном ключе. Назовите относительные преимущества и недостатки индексированной структуры по сравнению с структурой кучи.

Г.6. В настоящем приложении несколько раз упоминалась физическая кластеризация. Например, может оказаться выгодной такая организация физического хранения записей поставщиков, чтобы их физическое упорядочение было одинаковым или достаточно близким к логическому упорядочению, которое определено значениями поля номера поставщика (поля кластеризации). Каким образом СУБД может обеспечить такую физическую кластеризацию?

Г.7. В разделе Г.5 было указано, что один из способов организации хэшированного доступа с учетом коллизий предусматривает использование результатов вызова хэш-функции в качестве отправной точки для последовательного просмотра (метод линейного поиска). Какие сложности возникают, по вашему мнению, при использовании этой схемы?

Г.8. Каковы относительные преимущества и недостатки метода организации доступа с применением сразу нескольких родительско-дочерних структур? (В конце раздела Г.6 приведен обзор преимуществ и недостатков метода организации доступа с применением нескольких индексов, который может помочь при выполнении данного задания. В чем заключается сходство между этими методами? В чем состоят различия?)

Г.9. Определим понятие "полной индексации" как наличие индекса для каждой отдельной (и отдельно упорядоченной) комбинации полей в индексированном файле. Например, для полной индексации файла с двумя полями, A и B , требуются два индекса — один для комбинации AB (в указанном порядке), другой — для комбинации BA (в указанном порядке). Сколько индексов необходимо для обеспечения полной индексации файла, который определен:

а) на 3 полях;

б) на 4 полях;

в) на N полях.

Г.10. Рассмотрим упрощенное B-дерево (состоящее из индексного набора и последовательного набора), в котором последовательный набор содержит указатель на каждую из записей данных N, а на каждом уровне, более высоком по сравнению с последовательным набором (т.е. на каждом уровне индексного набора) содержится указатель на каждую страницу нижележащего уровня. На верхнем (корневом) уровне, безусловно, находится единственная страница. Предположим также, что каждая страница индексного набора содержит элементы индекса n. Выведите формулы для определения количества уровней и количества страниц во всем B-дереве.

Г.11. Ниже приведены первые 10 значений индексированного поля в некотором индексированном файле.

```
 Abrahams,GK
Ackermann, LZ
Ackroyd,S
Adams,T
Adams,TR
Adamson,CR
Allen,S
Ayres,ST
Bailey,TE
Baileyman,D
```

Каждое поле дополняется справа пробелами до полной длины в 15 символов. Покажите значения, фактически зарегистрированные в индексе при использовании методов префиксного и суффиксного сжатия, описанных в разделе Г.7. Чему равна экономия пространства в процентах? Какие шаги должны быть выполнены при выборке (или попытке выборки) записей с данными "Ackroyd. S" и "Adams, V". Перечислите также шаги, которые должны быть выполнены при вставке новой записи сданными "Allingham,M".

СПИСОК ЛИТЕРАТУРЫ

Приведенный ниже список литературы организован по группам следующим образом. В [Г.1]—[Г.10] перечислены учебные пособия, которые полностью посвящены теме этого приложения или, по крайней мере, включают подробные сведения по данной теме. В [Г.11]—[Г.15] приведены некоторые формальные трактовки этого материала. В [Г. 16]—[Г.23] в основном рассматриваются методы индексации, в частности, на основе B-деревьев; в [Г.24]—[Г.38] представлена часть очень обширной литературы по хэшированию; в [Г.39]-[Г.4О] рассматриваются методы сжатия; наконец, в [Г.41]-[Г.59] представлены различные структуры хранения и связанные с ними вопросы (например, в [Г.48]-[Г.59] описаны некоторые новые средства хранения данных и типы приложений, а также структуры хранения для этих средств и приложений).

ГЛ. Knuth D.E. The Art of Computer Programming. Volume III: Sorting and Searching (2d ed.). Reading, Mass.: Addison-Wesley, 1998.

Том 3 классической серии книг Кнута содержит результаты исчерпывающего анализа алгоритмов поиска. Что касается поиска в базе данных, где данные представлены во вторичной памяти, то к этой теме непосредственно относятся такие разделы этого тома, как 6.2.4 ("Сильно ветвящиеся деревья"), 6.4 ("Хэширование") и 6.5 ("Выборка по вторичным ключам").

Г.2. Martin J. Computer Data-Base Organization (2d ed.). Englewood Cliffs, N.J.: Prentice-Hall, 1977.

Эта книга разделена на две основные части: "Логическая организация" и "Физическая организация". Последняя часть представляет собой исчерпывающее описание структур хранения и соответствующих методов доступа (которое превышает по объему 300 страниц).

Г.3. (То же, что и [14.45].) Teorey T.J., Fry J.P. Design of Database Structures. Englewood Cliffs, N.J.: Prentice-Hall, 1982.

Учебное руководство по проектированию базы данных, как физическому, так и логическому. Свыше 200 страниц посвящены физическому проектированию.

F.4. Wiederhold G. Database Design (2d ed.). New York, N.Y.: McGraw-Hill, 1983.

Эта книга, состоящая из 15 глав, включает широкий обзор устройств вторичной памяти и их параметров производительности (одна глава, примерно 50 страниц) и исчерпывающий анализ вторичных структур хранения (четыре главы, свыше 250 страниц).

Г.5. Merrett T. H. Relational Information Systems. Reston, Va.: Reston Publishing Company, Inc., 1984.

Включает обширное введение и анализ различных структур хранения (около 100 страниц), где рассматриваются не только структуры, описанные в данном приложении, но и некоторые другие.

Г.6. Ullman J.D. Principles of Database and Knowledge-Base Systems: Volume I. Rockville, Md.: Computer Science Press, 1988.

Приведено описание структур хранения, которое содержит гораздо больше теоретических сведений по сравнению с настоящим приложением.

Г.7. (То же, что и [16.21].) Silberschatz A., Korth H.F., Sudarshan S. Database System Concepts (4th ed.). New York, N.Y.: McGraw-Hill, 2002.

Г.8. Smith P.D., Barnes G. M. Files and Databases: An Introduction. Reading, Mass.: Addison-Wesley, 1987.

Г.9. (То же, что и [14.18].) Elmasri R., Navathe S.B. Fundamentals of Database Systems (3d ed.). Redwood City, Calif.: Benjamin/Cummings, 2000.

В [Г.7], [Г.8] и [Г.9] приведены учебные руководства по системам баз данных. Каждая из этих книг включает материал по структурам хранения, который во многом является более подробным, чем данное приложение (это особенно касается [Г.8]).

Г.10. Ghosh S.P. Data Base Organization for Data Management (2d ed.). Orlando, Fla.: Academic Press, 1986.

В этой книге в основном рассматриваются структуры хранения и связанные с ними методы доступа (из десяти глав данной книги указанной теме посвящено не меньше шести глав). Способ изложения материала является довольно абстрактным.

Г.11. Hsiao D.K., Harary F. A Formal System for Information Retrieval from Files, CACM . — February 1970.-13, №2.

В этой статье представлены результаты, по-видимому, одной из самых ранних попыток привести все идеи в области создания различных структур хранения (в основном индексов и цепочек указателей) к одной общей модели и создать тем самым основу для формальной теории таких структур. Представлен обобщенный алгоритм выборки, позволяющий осуществлять выборку записей из обобщенной структуры, который удовлетворяет произвольной логической комбинации условий "поле = значение".

Г.12. Severance D.G. Identifier Search Mechanisms: A Survey and Generalized Model, ACM Cотр. Surv. — September 1974. -6, № 3.

Эта статья состоит из двух частей. В первой части приведено учебное руководство по некоторым структурам хранения (главным образом основанным на хэшировании и индексации). Во второй части рассматривается такая же тема, как и в [Г. 11]; в ней определена унифицированная структура, именуемая в данной статье структурой trie-дерева, в которой объединены и обобщены идеи создания структур, описанных в первой части. (Термин trie, который произносится как "трай", впервые был предложен в статье Фредкина [Г.42].) Результирующая структура позволяет создать общую модель, способную представить широкий ряд разнообразных структур с помощью небольшого количества формальных параметров, поэтому она может использоваться (и фактически уже используется) как вспомогательное средство при выборе конкретной структуры в ходе реализации процесса физического проектирования базы данных.

Различие между этой статьей и [Г. 11] состоит в том, что структура на основе trie-дерева позволяет представлять хэши, а не цепочки указателей, тогда как структура, предложенная в [Г.11], поддерживает цепочки указателей, но не хэши.

Г.13. Senko M. E., Altman E. B., Astrahan M. M., Fehder P. L. Data Structures and Accessing in Data-Base Systems, IBM Sys. J. - 1973. - 12, № 1.

Эта статья состоит из трех перечисленных ниже частей

1. Эволюция информационных систем.
2. Организация информации.
3. Способы представления данных и модель доступа, независимая от данных.

Первая часть представляет собой краткий исторический обзор процесса разработки систем баз данных до 1973 года. Во второй части описана так называемая модель множества сущностей (entity set), которая может служить основой для описания данного конкретного предприятия в терминах сущностей и множеств сущностей (эта модель соответствует концептуальному уровню архитектуры ANSI/SPARC). Третья часть является наиболее оригинальной и значимой частью этой

статьи; в ней содержится вводное описание модели доступа, независимой от данных (Data Independent Accessing Model — DIAM), которая представляет собой попытку описать базу данных в терминах четырех последовательных уровней абстракции: множество сущностей (наиболее высокий уровень), строковый уровень, уровень кодирования и уровень физического устройства. Эти четыре уровня могут рассматриваться как более детализированное, но все еще достаточно абстрактное определение концептуальной и внутренней частей архитектуры ANSI/SPARC. Краткое описание этих уровней приведено ниже.

- Уровень множества сущностей (entity set level). Аналогичен концептуальному уровню . . ANSI/SPARC.
- Строковый уровень (string level). Пути доступа к данным определены как упорядоченные множества или так называемые строки объектов данных.

  Описаны три типа строк: атомарные строки (например, строки, соединяющие экземпляры полей для формирования экземпляра записи с данными о детали), строки сущностей (например, строки, соединяющие экземпляры записей с данными о деталях красного цвета) и строки связей (например, строки, соединяющие экземпляры записей поставщиков с экземплярами записей деталей, поставляемых этим поставщиком).
- Уровень кодирования (encoding level). Объекты данных и строки отображаются на линейные адресные пространства с использованием простого примитива представления, известного под названием базовой единицы кодирования.
- Уровень физического устройства (physical device level). Линейные адресные пространства распределяются по отформатированным физическим подразделам реального регистрирую щего данные носителя данных.

Назначение модели DIAM, как и структур, описанных в [Г.11]—[Г.12], состоит (отчасти) в том, что она должна служить основой для теоретической систематизации структур хранения *1* и методов доступа. Важное критическое замечание (которое, кстати, относится и к формальным средствам, представленным в [Г.11] и [Г.12]), заключается в том, что иногда наилучшим методом выполнения некоторого конкретного запроса доступа является просто сортировка данных. Но сортировка, безусловно, является динамической операцией, а структуры, представленные с помощью модели DIAM (и моделей, описанных в [Г.11] и [Г.12]) по определению всегда остаются статическими.

Г. 14. Yao S. B. An Attribute Based Model for Database Access Cost Analysis, ACM TODS. — March 1977. — 2, № 1.

Назначение этой статьи аналогично работам [Г.11] и [Г.12]; в некоторых аспектах ее можно считать следствием этих более ранних работ, поскольку в ней представлена обобщенная модель структур хранения, которую можно рассматривать как объединение и дополнение предложений этих работ. В ней также представлен ряд обобщенных алгоритмов доступа и уравнений стоимости для этой обобщенной модели. Приведено большое количество ссылок на другие статьи, содержащие отчеты об экспериментах с анализатором физических проектов файлов, созданным на основе идей этой статьи.

Г.15. Batory D.S. Modeling the Storage Architectures of Commercial Database Systems, ACM TODS. — December 1985. - 10, № 4.

Представлено множество примитивных операций, называемых элементарными трансформациями, с помощью которых можно сделать явным и поэтому полностью изучить процесс преобразования из концептуальной схемы в соответствующую внутреннюю схему (т.е. процесс концептуально-внутреннего преобразования — см. главу 2). Элементарные трансформации включают операции дополнения (расширения записи путем включения данных префикса, а также пользовательских данных), кодирования (преобразования данных во внутреннюю форму, например, с помощью сжатия), сегментации (разбиения записи на несколько частей с учетом условий хранения) и некоторые другие операции. В статье приведено утверждение, что любое концептуально-внутреннее преобразование может быть представлено с помощью соответствующей последовательности таких элементарных трансформаций. Из этого следует, что указанные трансформации могут служить основой одного из подходов к автоматизации разработки программного обеспечения

управления данными. В качестве иллюстрации в данной статье изложенные идеи применяются для анализа трех коммерческих систем: INQUIRE, ADABAS и System 2000.

Примечание. Сравните и сопоставьте эту статью с (появившимися гораздо позже) предложениями GMAP, приведенными в [2.5]. Некоторые описания, которые могут рассматриваться как контрпримеры, можно найти также в приложении А.

Г.16. Bayer R., McCreight C. Organization and Maintenance of Large Ordered Indexes, Acta Informatica. 1972.-1,№3.

Г.17. Comer D. The Ubiquitous B-Tree, ACM Сотр. Surv. — June 1979. — 11, № 2.

Хорошее учебное руководство по B-деревьям.

Г.18. Wagner R. E. Indexing Design Considerations, IBM Sys. J. — 1973. — 12, № 4.

Описаны основные понятия индексации и приведены подробные сведения о методах индексации (включая методы сжатия), которые используются в методе доступа к виртуальной памяти (Virtual Storage Access Method — VSAM) компании IBM.

Г.19. Chang H. K. Compressed Indexing Method, IBM Technical Disclosure Bulletin. — April 1969. — 2, № 11

Г.20. Gupta G.K. A Self-Assessment Procedure Dealing with Binary Search Trees and B-Trees, CACM. — May 1984. - 27, № 5.

Г.21. Dim V.Y. Multi-attribute Retrieval with Combined Indexes, CACM. — November, 1970. — 13, № 11.

Именно в этой статье был впервые представлен метод индексации с использованием комбинаций полей.

Г.22. Mullin J.K. Retrieval-Update Speed Tradeoff Using Combined Indices, CACM. — December 1971. — 14, №12.

Дополнение к работе [Г.21], в котором приведены статистические данные о производительности, достигнутой при использовании схемы комбинированной индексации для различных соотношений операций выборки/обновления.

Г.23. Shneiderman B. Reduced Combined Indexes for Efficient Multiple Attribute Retrieval, Information Systems. 1976. — 2, № 4.

Предложено усовершенствование комбинированного метода индексации Лама [Г.21], позволяющее значительно уменьшить объем памяти и сократить время поиска. Например, все комбинации индексов ABCD, BCDA, CDAB, DABC, ACBD, BDAC (см. ответ к упражнению Г.9, *б)* можно заменить комбинацией ABCD, BCD, CDA, DAB, AC, BD. Если предположить, что в каждом из полей A, B, C и D могут находиться 10 различных значений, то в наихудшем случае для первоначальной комбинации потребуется 60 000 элементов индекса, а для сокращенной комбинации — только 13 200 элементов.

Г.24. Morris R. Scatter Storage Techniques, CACM. — January 1968. — 11, № 1.

Эта статья в основном посвящена методам хэширования, применяемым при организации таблицы символов ассемблера или компилятора. Ее основное назначение состоит в описании **косвенной** схемы хэширования, основанной на *таблицах разброса.* Таблицей разброса (scatter table) называется таблица адресов записей, которая немного напоминает каталог, используемый в методе расширяемого хэширования [Г.28]. Как и в методе расширяемого хэширования, хэш-функция применяется для доступа к таблице разброса, а не к самим записям; записи могут храниться в любом подходящем месте. Поэтому таблица разброса может рассматриваться как одноуровневый индекс к соответствующим данным, но к этому индексу может быть получен непосредственный доступ с помощью хэш-функции, без необходимости осуществления последовательного поиска. Следует отметить, что для любого конкретного файла данных вполне может быть предусмотрено несколько различных таблиц разброса, что, по сути, позволяет обеспечить хэшированный доступ к данным с помощью нескольких различных полей хэширования (за счет дополнительной операции ввода—вывода в расчете на любую конкретную операцию хэшированного доступа).

Несмотря на то, что эта работа в основном предназначена для описания способа реализации рассматриваемых методов на языках программирования, в ней дано хорошее вводное описание методов хэширования в целом, и основная часть материала применима также к средствам хэширования базы данных.

Г.25. Maurer W. D., Lewis T. G. Hash Table Methods, ACM Сотр. Surv. - March 1975. - 7, № 1.

Хорошее учебное руководство, хотя и немного устаревшее (в нем, естественно, не рассматриваются некоторые из новейших подходов, такие как расширяемое хэширование). В этой работе рассматриваются, в частности, такие темы, как основные методы хэширования (не только метод на основе деления с остатком, но и методы с использованием случайных чисел, середин квадратов, результатов извлечения квадратного корня, алгебраического кодирования, свертки и числового анализа), устранение коллизий и переполнения сегмента, некоторые результаты теоретического анализа различных методов, а также методы, альтернативные хэшированию (которые следует применять, если в данных обстоятельствах использовать хэширование либо невозможно, либо недопустимо).

Примечание. Сегментом (bucket) в терминологии хэширования называется единица хранения (как правило, страница), адрес которой вычисляется с помощью хэш-функции. Сегмент обычно содержит несколько записей.

Г.26. Lum V. Y., Yuen P.S.T., Dodd M. Key-to-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files, CACM. — April 1971. — 14, № 4.

Исследование производительности некоторых "базовых" (т.е. нерасширяемых) алгоритмов хэширования. В статье сделан вывод, что метод с использованием деления с остатком, по-видимому, обладает самой высокой производительностью по сравнению с другими методами.

Г.27. Ramakrishna M. V. Hashing in Practice: Analysis of Hashing and Universal Hashing, Proc. 1988 ACM SIGMOD Int. Conf. on Management of Data, Chicago, 111. — June 1988.

Как указано в этой статье (на основании работы Кнута [Г.1]), любая система, в которой реализован метод хэширования, должна обеспечивать решение двух проблем, почти полностью независимых друг от друга. Во-первых, в ней необходимо выбрать наиболее эффективные из всего широкого ряда имеющихся хэш-функций, а во-вторых, предусмотреть также эффективный метод устранения коллизий. Автор данной статьи утверждает, что второй из этих проблем посвящено много исследований, тогда как для решения первой проблемы сделано очень мало и предпринято лишь несколько попыток сравнить практические показатели производительности хэширования с теоретически достижимыми показателями (в качестве одного из исключений можно назвать [Г.26]). В связи с этим у тех, кто занимается созданием реальных систем, возникают сложности при выборе подходящей хэш-функции. В статье приведено утверждение, что возможно выбрать такую хэш-функцию, которая позволяет на практике достичь производительности, близкой к теоретически предсказанной, а также представлен ряд теоретических результатов в поддержку этого утверждения.

Г.28. Fagin R., Nievergelt J., Pippenger N., Strong H. R. Extendible Hashing - A Fast Access Method for Dynamic Files, ACM TODS. - September 1979. - 4, № 3.

Г.29. Knott G. D. Expandable Open Addressing Hash Table Storage and Retrieval, Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access, and Control, San Diego, Calif. — November 1971.

Г.30. Larson P.-E. Dynamic Hashing, BIT. - 1978. - 18.

Г.31. Litwin W. Virtual Hashing: A Dynamically Changing Hashing, Proc. 4th Int. Conf. on Very Large Data Bases, Berlin, FDR. — September 1978.

Г.32. Litwin W. Linear Hashing: A New Tool for File and Table Addressing, Proc. 6th Int. Conf. on Very Large Data Bases, Montreal, Canada. — October, 1980.

Г.33. Larson P.-E. Linear Hashing with Overflow-Handling by Linear Probing, ACM TODS. — March 1985.- 10, № 1.

Г.34. Larson P.-E. Linear Hashing with Separators — A Dynamic Hashing Scheme Achieving One-Access Retrieval, ACM TODS. - September 1988. - 13, № 3.

Во всех работах [Г.28]—[Г.34] представлены схемы расширяемого хэширования того или иного типа. Предложения работы [Г.29] по дополняемому хэшированию (expandable hashing) появились раньше по сравнению со всеми прочими работами (и поэтому, безусловно, являются совершенно независимыми от них). Тем не менее, дополняемое хэширование весьма напоминает расширяемое хэширование, которое определено в [Г.28], и таковым же является "динамическое" хэширование [Г.ЗО], за исключением того, что в обоих этих схемах используются древовидный каталог, а не просто непрерывный каталог, предложенный в [Г.28]. Метод "виртуального" хэширования [Г.31] характеризуется определенными отличиями; подробные сведения об этом можно найти в указанной статье. Метод "линейного" хэширования, впервые представленный в [Г.32] и дополненный в [Г.ЗЗ] и [Г.34], представляет собой усовершенствование метода виртуального хэширования.

Г.35. Litwin W. Trie Hashing, Proc. 1981 ACM SIGMOD Int. Conf. on Management of Data, Ann Arbor, Mich.-April 1981.

Представлена схема расширяемого хэширования, которая обладает целым рядом описанных ниже желательных свойств.

- Она позволяет сохранить упорядочение (это означает, что "физическая" последовательность записей соответствует логической последовательности этих записей, которая определяется значениями поля хэширования).
- Эта схема позволяет избежать возрастания сложности алгоритмов, а также других проблем, которые обычно связаны с применением хэшей, сохраняющих упорядочение.
- Доступ к произвольной записи можно осуществить (или показать, что эта запись отсутствует) с помощью одной операции доступа к диску, даже если файл содержит много миллионов записей.
- Файл может обладать произвольной изменчивостью (в отличие от этого, многие другие схемы хэширования, по крайней мере, те, которые относятся к нерасширяемому типу, обнаруживают тенденцию к значительному снижению производительности при возрастании объемов операций вставки).

Сама хэш-функция (изменяющаяся во времени, как и во всех алгоритмах расширяемого хэширования) представлена с помощью структуры trie-дерева [Г.42], которая хранится в оперативной памяти в течение всего периода времени использования файла и бесперебойно наращивается по мере возрастания файла данных. В самом файле данных, как уже было указано, поддерживается "физическая" последовательность значений поля хэширования, а логическая последовательность листовых элементов в структуре trie-дерева точно соответствует этой "физической" последовательности записей данных. Переполнение в файле данных устраняется с помощью метода разбиения страниц, по сути аналогичного методу разбиения страниц, используемому в B-дереве.

Методы хэширования с помощью trie-дерева представляются весьма перспективными. Как и другие схемы хэширования, они обеспечивают более высокую производительность по сравнению с индексацией для непосредственного доступа (в них обычно применяется одна операция ввода—вывода вместо двух или трех операций, необходимых при использовании B-дерева); к тому же, эти методы являются более предпочтительными по сравнению с большинством других схем хэширования, в том отношении, что они сохраняют упорядочение. Это означает, что последовательный доступ также осуществляется очень быстро. Для обеспечения быстрого последовательного доступа не требуются B-деревья или другие дополнительные структуры. Тем не менее, необходимо учитывать такую предпосылку, что trie-дерево должно постоянно находиться в оперативной памяти (которая, по-видимому, вполне реальна). Но если эта предпосылка оказывается недействительной (т.е. если файл данных слишком велик) или не требуется свойство сохранения упорядоченности, то более предпочтительной альтернативой становится линейное хэширование [Г.32] или какой-то другой метод.

Г.36. Lomet D.B. Bounded Index Exponential Hashing, ACM TODS. - March 1983. - 8, № 1.

Еще одна схема расширяемого хэширования. В этой работе содержатся приведенные ниже утверждения.

- Описанная схема предоставляет непосредственный доступ к любой записи в среднем при мерно за одну операцию ввода—вывода (а в общем количество таких операций никогда не превышает двух).
- Эта схема позволяет достичь производительности, не зависящей от размера файла. (В отличие от этого, большинство схем расширяемого хэширования характеризуются временным снижением производительности на том этапе, когда происходит разделение переполненных страниц каталога, поскольку обычно приходится разделять все такие страницы приблизите льно в одно и то же время.)
- Она позволяет эффективно использовать доступное дисковое пространство (т.е. с ее помощью можно добиться очень экономичного распределения пространства памяти).
- Реализация этой схемы является несложной.

Г.37. GargA.K.,GotliebC. C. Order-Preserving Key Transformations, ACM TODS. —June 1986. — 11,№2.

Как было описано в аннотации к [Г.35], сохраняющей упорядочение хэш-функцией (или "трансформацией ключей") называется такая функция, при использовании которой физическая последовательность записей соответствует логической последовательности этих записей, которая определена значениями поля хэширования. Хэши с сохранением упорядочения являются предпочтительными по очевидным причинам. Одна из простых функций, которая явно сохраняет упорядочение, представлена ниже.

хэшированный адрес = частное от деления значения хэшированного поля на некоторую константу (скажем, 10 000)

Однако один из очевидных недостатков применения подобной функции состоит в том, что соответствующий метод доступа обладает очень низкой производительностью, если значения поля хэширования распределены неравномерно (а такая ситуация, безусловно, является самой распространенной). Поэтому некоторые исследователи выдвинули идею использования независимых от распределения (но сохраняющих упорядочение) хэш-функций или, иными словами, функций, которые преобразуют неравномерно распределенные значения поля хэширования в равномерно распределенные хэш-адреса, поддерживая при этом свойство сохранение упорядочения. *(Примечание.* Одним из примеров такого подхода является хэширование с помощью trie-дерева [Г.35].) В данной статье приведен метод формирования подобных хэш-функций для файлов данных, которые встречаются в реальных приложениях, и показана практическая применимость этих функций.

Г.38. Ramakrishna M. V., Larson P.E. File Organization Using Composite Perfect Hashing, ACM TODS. — June 1989. - 14, № 2.

Хэш-функция называется *идеальной,* если она не создает переполнения. *(Примечание.* "Отсутствие переполнений" не означает "отсутствие коллизий". Например, если предположить, что хэш-функция вырабатывает адреса страниц, а не адреса записей, как указано в примечании о сегментах в аннотации к [Г.25], и что на каждой странице могут находиться п записей, то хэш-функция является идеальной, если она никогда не направляет больше чем п записей на одну и ту же страницу.) Идеальная хэш-функция обладает таким свойством, что выборка любой записи может быть выполнена за одну операцию ввода—вывода на диске. В данной статье представлен практический метод поиска и применения таких идеальных функций.

Г.39. Huffman D.A. A Method for the Construction of Minimum Redundancy Codes, Proc. IRE 40. — September 1952. **Г.40.** Matron B. A., de Maine P. A. D. Automatic Data Compression, CACM. — November 1967. — 10, № 11.

Описаны два алгоритма сжатия/восстановления: NUPAK, который оперирует с числовыми данными, и ANPAK, который оперирует с алфавитно-цифровыми, или "любыми", данными (т.е. с любыми строками битов).

Г.41. Severance D.G., Lohman G.M. Differential Files: Their Application to the Maintenance of Large Databases, ACM TODS. - September 1976. - 1, № 3.

Рассматриваются "дифференциальные файлы" и их преимущества. Основная идея состоит в том, что обновления не вносятся непосредственно в саму базу данных, а вместо этого регистрируются в физически отдельном файле (дифференциальном файле); в дальнейшем в некоторый подходящий момент времени происходит слияние этого файла с самой базой данных. Авторы статьи утверждают, что этот подход обладает описанными ниже преимуществами.

- Сокращаются расходы на резервное копирование базы данных.
- Упрощается инкрементное резервное копирование.
- И операции резервного копирования, и операции реорганизации базы данных могут выполняться одновременно с операциями обновления.
- Обеспечивается быстрое восстановление после аварийного завершения прикладной программы.
- Обеспечивается быстрое восстановление после отказа аппаратных средств.
- Сокращается риск серьезной потери данных.
- Обеспечивается эффективная поддержка эталонных файлов (см. приведенное ниже описание).
- Упрощается разработка прикладного программного обеспечения.
- Упрощается структура основных программных средств управления файлами.
- Существует возможность сократить в будущем расходы на приобретение устройств памяти.

Примечание. Эталонный файл (memo file) представляет собой своего рода временную копию определенной части базы данных, применяемую для обеспечения быстрого доступа к данным, которые, по всей вероятности, являются актуальными и правильными, но нет гарантии, что они действительно таковы. См. описание снимков в главе 10.

В статье не рассматривается одна потенциальная проблема — обеспечение эффективного последовательного доступа к данным (например, с помощью индекса), когда некоторые записи находятся в реальной базе данных, а другие — в дифференциальном файле.

Fredkin E. TRIE Memory, CACM. - September 1960. - 3, № 9.

Trie-деревом называется древовидный файл данных (а не древовидный путь доступа к такому файлу; это означает, что в виде дерева представлены сами данные, а не указатели на эти данные — исключением из этого правила можно считать "файл данных", который в действительности является индексом к некоторому другому файлу, как фактически обстоит дело при использовании хэширования на основе trie-дерева [Г.35]). Каждый узел в trie-дереве логически состоит из n записей, где n — количество различных символов, предназначенных для представления значений данных. Например, если каждый элемент данных представляет собой десятичное целое число, то каждый узел включает ровно 10 элементов, соответствующих десятичным цифрам 0, 1,2, ..., 9. Рассмотрим элемент данных "4285". На вершине дерева находится (уникальный) узел, который включает указатель на элемент "4". Этот указатель указывает на узел, соответствующий всем существующим элементам данных, содержащим "4" в качестве первой цифры. Этот узел ("узел 4") содержит в своем элементе "2" указатель на узел, соответствующий всем элементам данных, содержащим "42" в качестве их первых двух цифр ("узел 42"). Узел "42" содержит указатель на свой элемент "8" в узле "428" и т. д. А если (например) не существуют элементы данных, начинающиеся с "429", то элемент "9" в узле "42" будет пуст (в нем не будет находиться указатель); иными словами, дерево усекается таким образом, чтобы в нем содержались только непустые узлы. (Поэтому, вообще говоря, trie-дерево не является сбалансированным деревом.)

Примечание. Термин trie происходит от слова "retrieval" — выборка (которое произносится как "ретривал"), но, тем не менее, для него обычно предусмотрен вариант чтения "трай". Trie-деревья упоминаются также под названием поразрядных поисковых деревьев (radix search tree) или цифровых поисковых деревьев (digital search tree).

Г.43. Wong E., Chiang T. C. Canonical Structure in Attribute Based File Organization, CACM. — September 1971.-14, №9.

Предложена принципиально новая структура хранения, основанная на алгебре логики. В ней предполагается, что все запросы доступа выражаются как логические комбинации элементарных условий "поля = значение" и что все эти элементарные условия известны. В таком случае перед выводом на устройство хранения файл может быть секционирован на непересекающиеся подмножества. Эти подмножества рассматриваются как "элементарные объекты" алгебры логики, состоящие из множества всех множеств записей, выборка которых осуществима с помощью первоначальных запросов логического доступа. Ниже перечислены преимущества такой организации хранения.

- Для элементарных объектов никогда не требуется выполнять операцию пересечения множеств.
- Любой произвольный логический запрос можно легко преобразовать в запрос, предусматривающий объединение одного или нескольких элементарных объектов.
- После такого объединения никогда не возникает необходимость устранять дубликаты.

Г.44. Stonebraker M. Operating System Support for Database Management, CACM. — July 1981. — 24, № 7.

В работе рассматриваются причины, по которым различные средства операционной системы (в частности, средства управления файлами операционной системы) часто не предоставляют такие услуги, которые необходимы для СУБД, и предложены некоторые способы усовершенствования указанных средств.

Г.45. Schkolnick M. A Survey of Physical Database Design Methodology and Techniques, Proc. 4th Int. Conf. on Very Large Data Bases, Berlin, FDR. — September 1978.

Г.46. Finkelstein S., Schkolnick M., Tiberio P. Physical Database Design for Relational Databases, ACM TODS. - March 1988. - 13, № 1.

В определенных отношениях в реляционных системах проблема физического проектирования базы данных является более сложной по сравнению с другими системами. Это связано с тем, что решение по выбору способов "перемещения" по структуре хранения принимает система, а не пользователь, поэтому система сможет достичь высокой производительности, только если структуры хранения, выбранные проектировщиком базы данных, полностью соответствуют реальным потребностям системы. Из этого следует, что проектировщик должен быть знаком с достаточно подробными сведениями о внутреннем функционировании системы. Но проектировщики, как правило, не обладают такими знаниями (к тому же и не требуется, чтобы они обладали этими знаниями или стремились ими обладать). В связи с этим весьма желательно иметь своего рода инструментальное средство автоматизированного физического проектирования. В данной статье сообщается о создании такого инструмента, называемого DBDSGN, который был разработан для использования в системе System **R [4.1]—[4.3], [4.12]—[4.14].** На вход инструментального средства DBDSGN подается определение рабочей нагрузки (т.е. множества пользовательских запросов и соответствующих им частот выполнения), а на выходе формируется предлагаемый физический проект (т.е. множество индексов для каждого файла, которое в каждом случае обычно включает "кластеризующий индекс" — см. упражнение Г.6). Это средство взаимодействует с оптимизатором системы (см. главу 18) для получения дополнительной информации, например, о том, какие сведения о базе данных (в частности, о размерах файла) накоплены оптимизатором, и выявления формул стоимости, используемых оптимизатором.

Инструментальное средство DBDSGN применялось в качестве основы для программного продукта компании IBM, называемого RDT, который представляет собой инструментальное средство проектирования для SQL/DS [4.14].

Г.47. Sevcik K.C. Data Base System Performance Prediction Using an Analytical Model, Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France. — September 1981.

Как явствует из названия этой статьи ("Прогнозирование производительности системы базы данных с использованием аналитической модели"), тема этой статьи гораздо шире по сравнению

с материалом настоящего приложения; в ней рассматриваются общие проблемы производительности всей системы, а не только структур хранения как таковых. Автор статьи предлагает многоуровневую инфраструктуру, в рамках которой может быть организовано систематическое исследование различных проектных решений и взаимосвязи между этими решениями. Уровни этой инфраструктуры представляют систему со все более возрастающей степенью детализации описания, поэтому каждый следующий уровень является более конкретным (т.е. находится на более низком уровне абстракции), чем предыдущий. Названия уровней дают определенное представление о соответствующей степени детализации: абстрактное представление мира, логическая база данных, физическая база данных, доступ к единице данных, доступ на основе физического ввода—вывода и загрузка устройств. Автор утверждает, что аналитическая модель, основанная на этой инфраструктуре, может использоваться для прогнозирования многих характеристик производительности, включая коэффициент использования устройств, производительность выполнения транзакций и время отклика.

Статья включает обширный аннотированный список литературы по производительности компьютерной системы. В частности, в ней приведен краткий обзор работ по производительности различных структур хранения.

Г.48. Samet H. The Quadtree and Related Hierarchical Data Structures, ACM Cotp. Surv. — June 1984. — 16, №2.

Структуры хранения, описанные в настоящем приложении, достаточно хорошо подходят для традиционных коммерческих баз данных. Но по мере того как область применения технологии баз данных расширяется и охватывает данные новых типов (например, пространственные данные, наподобие тех, что используются при обработке изображений или в картографических приложениях), возникает необходимость в использовании новых методов представления данных на уровне хранения. Данная статья представляет собой вводное учебное описание некоторых из этих новых методов. Дополнительные сведения можно найти в книге автора данной статьи, Самета [Г.49], атакже в [Г.50]-[Г.59].

Г.49. (То же, что и [26.37].) Samet H. The Design and Analysis of Spatial Data Structures. Reading, Mass.: Addison-Wesley, 1990.

См. аннотацию к предыдущей работе.

Г.50. Christodoulakis S., Ford D. A. Retrieval Performance Versus Disc Space Utilization on WORM Optical Discs, Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. — May/June 1989. См. аннотацию к [Г.51].

Г.51. Lomet D., Salzberg B. Access Methods for Multiversion Data, Proc. 1989 ACM SIGMOD Int. Conf. on Management of Data, Portland, Ore. — May/June 1989.

Аббревиатурой WORM обозначается оптический диск, обладающий тем свойством, что после сохранения на нем какой-то записи ее больше нельзя стереть, т.е. обновить эту запись на месте (WORM расшифровывается как "Write Once, Read Many" — однократная запись и многократное считывание). Подобные диски обладают очевидными преимуществами при использовании в некоторых приложениях, особенно в таких, где требуется доступ к архивной информации. Но для этих дисков не применимы традиционные структуры хранения, такие как B-деревья, именно в связи с тем фактом, что их перезапись невозможна. Поэтому (как было описано в аннотации к [Г.48], но в связи с другими причинами) требуются новые структуры хранения. Некоторые такие структуры предложены и проанализированы в [Г.50] и [Г.51]; в частности, в [Г.51] рассматриваются структуры, подходящие для таких приложений, в которых должна быть предусмотрена полная регистрация хронологических сведений, т.е. приложений, в которых данные только добавляются в базу данных, но никогда не удаляются (см. главу 23).

Г.52. Encarnacao J., Krause F.-L. (eds.). File Structures and Data Bases for CAD. New York, N.Y.: North-Holland, 1982.

Один из основных стимулов к проведению исследований в области новых структур хранения связан с созданием приложений автоматизированного проектирования (Computer-Aided Design — CAD).

Эта книга состоит из докладов на семинарах по этой теме и включает перечисленные ниже основные разделы

1. Моделирование 'Данных для автоматизированного проектирования.
2. Модели данных для геометрического моделирования.
3. Базы данных для геометрического моделирования.
4. Аппаратные структуры.
5. Проблемы исследования баз данных для автоматизированного проектирования.
6. Проблемы реализации в системах базы данных для автоматизированного проектирования.
7. Производственные приложения.

Г.53. Finkei R. A., Bentley J. L. Quad-Trees - A Data Structure for Retrieval on Composite Keys, Acta Informatica. — 1974. — 4.

Г.54. Nievergelt J., Hinterberger H., Sevcik K. C The Grid File: An Adaptable, Symmetric, Multikey File Structure, ACM TODS. - March 1984. - 9, № 1.

Г.55. Guttman A. R-Trees: A Dynamic Index Structure for Spatial Searching, Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, Mass. — June 1984.

Г.56. Roussopoulos N., Leifker D. Direct Spatial Search on Pictorial Databases Using Packed R-Trees, Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Tex. — May 1985.

Г.57. Beckley D.A., Evens M. W., Raman V. K. Multikey Retrieval from K-D Trees and Quad-Trees, Proc. ACM SIGMOD Int. Conf. on Management of Data, Austin, Tex. — May 1985.

Г.58. Freeston M. The BANG File: A New Kind of Grid File, Proc. ACM SIGMOD Int. Conf. on Management of Data, San Francisco, Calif. — May 1987.

Г.59. Barnsley M.F., Sloan A.D. A Better Way to Compress Images, BYTE. — January 1988. - 13, № 1.

Описан принципиально новый метод сжатия для изображений, называемый *фрактальным сжатием.* Метод вообще не предусматривает запись на внешнее устройство самого изображения. Вместо этого на хранение записываются значения кодов, которые могут использоваться для восстановления желаемого изображения по мере необходимости с помощью соответствующих фрактальных уравнений. В результате, как указано в статье, можно достичь "коэффициентов сжатия 10 000 к 1 или даже выше".

ПРИЛОЖЕНИЕ Д

Ответы к отдельным упражнениям

ВВЕДЕНИЕ

В данном приложении приведены ответы к разделам "Упражнения" (приблизительно на 30% упражнений). Главным образом исключены ответы к упражнениям обзорного характера.

ОТВЕТЫ К ГЛАВЕ 1

1.2. Описание преимуществ приведено в самой главе. Некоторые недостатки перечислены ниже.

- Может быть нарушена защита данных (если отсутствуют надлежащие средства управления данными).
- Может быть нарушена целостность данных (если отсутствуют надлежащие средства управления данными).
- Может потребоваться дополнительное аппаратное обеспечение.
- Дополнительная нагрузка на вычислительную систему может оказаться весьма значительной.
- Задача обеспечения бесперебойного функционирования становится исключительно важной (предприятие может оказаться весьма уязвимым к сбоям системы).
- Велика вероятность того, что система станет более сложной (хотя все сложности должны быть скрыты от конечного пользователя).

1.5.

а)

WINE	PRODUCER
Zinfandel	Rafanelli

б)

WINE	PRODUCER
Chardonnay	Buena Vista
Chardonnay	Geyser Peak
Joh. Riesling	Jekel
Fumé Blanc	Ch. St. Jean
Gewurztraminer	Ch. St. Jean

в)

BIN#	WINE	YEAR
6	Chardonnay	2002
22	Fumé Blanc	2000
52	Pinot Noir	1999

г)

WINE	BIN#	YEAR
Cab. Sauvignon	48	1997

а) Указанная строка (относящаяся к ячейке с номером 80) добавляется к таблице CELLAR.

б) Строки, которые относятся к ячейкам с номерами 45, 48, 64 и 72, удаляются из таблицы CELLAR.

в) В строке, которая относится к ячейке номер 50, количество бутылок становится равным 5.

г) То же, что и в варианте в).

1.8.

а)
```
SELECT BIN#, WINE, BOTTLES
FROM   CELLAR
WHERE    PRODUCER  =  'Geyser  Peak'   ;
```

б)
```
SELECT  BIN#,   WINE
FROM     CELLAR
WHERE    BOTTLES  >  5  ;
```

В)
```
SELECT BIN#
FROM    CELLAR
WHERE  WINE  =   'Cab.   Sauvignon'
OR       WINE  =   'Pinot  Noir'
OR .     WINE  =   "Zinfandel'
OR       WINE  =   ' Syrah'
OR       . . .  ;
```

На этот вопрос нет краткого ответа, поскольку данные о цвете вина не зарегистрированы в базе данных в явном виде; это означает, что СУБД не может непосредственно найти информацию о том, что, например, вино "Pinot Noir" — красное.

Г)
```
UPDATE CELLAR
  SET   BOTTLES = BOTTLES + 3
 WHERE  BIN# = 30 ;
```

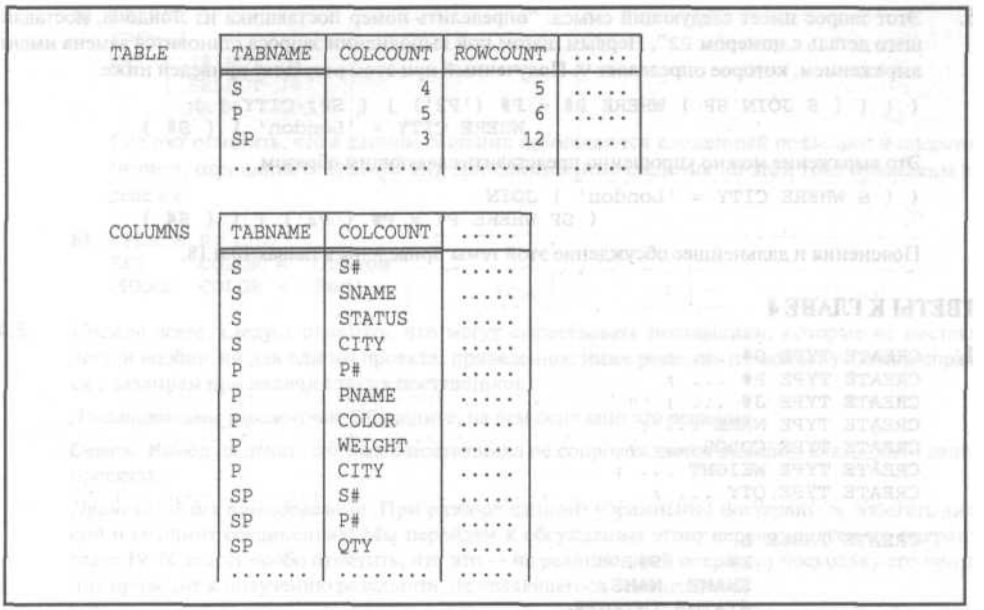
Д)
```
DELETE
      FROM   CELLAR WHERE  WINE =
      'Chardonnay' ;
```

е)
```
INSERT
      INTO   CELLAR ( BIN#, WINE, PRODUCER, YEAR, BOTTLES, READY ) VALUES ( 55,
      'Merlof, 'Gary Farrell', 2000, 12, 2005 ) ;
```

ОТВЕТЫ К ГЛАВЕ 3

3.2. На рис. Д.1 не показаны элементы каталога, соответствующие самим переменным отношения TABLE И COLUMN.

Примечание. На этом рисунке отсутствует также много другой информации. См. упр. 5.10 в главе 5.



TABLE

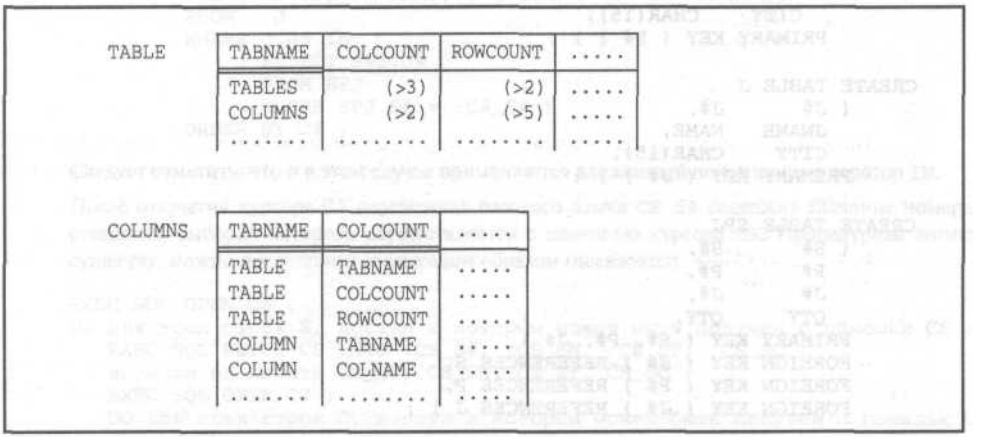
TABNAME	COLCOUNT	ROWCOUNT	.....
S	4	5	.....
P	5	6	.....
SP	3	12	.....
.....	.....	.....	.....

COLUMNS

TABNAME	COLCOUNT	.....
S	S#	.....
S	SNAME	.....
S	STATUS	.....
S	CITY	.....
P	P#	.....
P	PNAME	.....
P	COLOR	.....
P	WEIGHT	.....
P	CITY	.....
SP	S#	.....
SP	P#	.....
SP	QTY	.....
.....	.....	.....

Рис. Д. 1. Содержимое переменных отношения каталога TABLE и COLUMN

3.3. На рис. Д.2 показаны только те элементы каталога, которые соответствуют переменным отношения TABLE и COLUMN (т.е. элементы, соответствующие собственным пользовательским переменным отношения, опущены). Очевидно, что привести точные значения COLC0UNT и ROWCOUNT невозможно.



TABLE

TABNAME	COLCOUNT	ROWCOUNT	.....
TABLES	(>3)	(>2)	.....
COLUMNS	(>2)	(>5)	.....
.....	.....	.....	.....

COLUMNS

TABNAME	COLCOUNT	.....
TABLE	TABNAME	.....
TABLE	COLCOUNT	.....
TABLE	ROWCOUNT	.....
COLUMN	TABNAME	.....
COLUMN	COLNAME	.....
.....	.....	.....

Рис. Д.2. Сведения о переменных отношения TABLE и COLUMN

3.4. С помощью этого запроса выполняется выборка данных о номерах поставщиков и городах, где находятся поставщики, которые поставляют деталь P2.

3.5. Этот запрос имеет следующий смысл: "определить номер поставщика из Лондона, поставляющего деталь с номером P2". Первым шагом при выполнении запроса становится замена имени V выражением, которое определяет V. Полученный при этом результат приведен ниже.

```
( ( { ( S JOIN SP ) WHERE P# = P# CP2[1]) ) { S#, CITY } )
                                        WHERE CITY = 'London' ) { S# }
```

Это выражение можно упрощенно представить следующим образом. j

```
{ ( S WHERE CITY = 'London' ) JOIN
                                ( SP WHERE P# = P# CP2') ) ) { S# }
```

Пояснения и дальнейшее обсуждение этой темы приведены в главах 10 и 18.

ОТВЕТЫ К ГЛАВЕ 4

4.1.
```
CREATE TYPE S# ... ; CREATE TYPE P#
... ; CREATE TYPE J# ... ; CREATE
TYPE NAME ... ; CREATE TYPE
COLOR ... ; CREATE TYPE
WEIGHT ... ; CREATE TYPE QTY ...
;

CREATE TABLE S
                ( S#    S#, SNAME  NAME,
                STATUS INTEGER, CITY
                CHAR(15), PRIMARY KEY ( S# )
                ) ;

CREATE TABLE P
            (P#          P#,
            PNAME        NAME,
              COLOR      COLOR,
              WEIGHT     WEIGHT,
              CITY       CHAR(15),
            PRIMARY KEY ( P# ) ) ;

CREATE TABLE J
        ( J#     J#, JNAME   NAME, CITY
        CHAR(15), PRIMARY KEY ( J# ) )
        ;

CREATE TABLE SPJ
 ( S#     S#, P#     P#, J#     J#,
 QTY  QTY,
   PRIMARY KEY ( S#, P#, J# ), FOREIGN KEY ( S# )
   REFERENCES S,    FOREIGN KEY ( P# )
   REFERENCES P,
        FOREIGN KEY ( J# ) REFERENCES J ) ;
```

44.

```
a) INSERT INTO S ( S#, SNAME, CITY )
                VALUES ( S# ('S10'), NAME ('Smith'), 'New York' ) ;
```

В данном случае в качестве значения STATUS используется соответствующее значение по у молчанию (см. главу 6, раздел 6.6).

б) DELETE FROM J
WHERE J# NOT IN (SELECT J#
FROM SPJ) ;

Следует отметить, что в данном решении применяются вложенный подзапрос и оператор IN (вернее, отрицание оператора IN). Дополнительные сведения по этой теме приведены в разделе 8.6.

В) UPDATE P
SET COLOR = 'Orange' WHERE
COLOR = 'Red' ;

Прежде всего, следует отметить, что могут существовать поставщики, которые не поставляют детали вообще ни для одного проекта; приведенное ниже решение позволяет успешно справиться с заданием при наличии таких поставщиков. *Дополнительное упражнение.* Объясните, на чем основано это решение.

Ответ. Вывод на печать данных о поставщика не сопровождается выводом каких-либо данных о проектах.

Примечание для преподавателя. При разборе данного упражнения постарайтесь избегать дискуссий о внешних соединениях! Мы перейдем к обсуждению этого нерекомендуемого оператора в главе 19. (Следует особо отметить, что это — не реляционный оператор, поскольку его применение приводит к получению результата, не являющегося отношением.)

На первом этапе определим два курсора, CS и CJ, следующим образом.

```
EXEC SQL DECLARE CS CURSOR FOR
            SELECT S.S#, S.SNAME, S.STATUS, S.CITY FROM   S ORDER
            BY S# ;

EXEC SQL DECLARE CJ CURSOR FOR
            SELECT J.J#, J.JNAME, J.CITY FROM   J
            WHERE J.J# IN
                  { SELECT SPJ.J# FROM SPJ
                    WHERE SPJ.S# = :CS_S# ) „-... ORDERBYJ#;
```

Следует отметить, что и в этом случае применяются вложенный подзапрос и оператор IN.

После открытия курсора CJ переменная базового языка CS_S# содержит значение номера поставщика, выборка которого осуществляется с помощью курсора CS. Процедурная логику по существу, можно представить следующим образом (псевдокод).

```
EXEC SQL OPEN CS ;
DO для всех строк S, доступ к которым может быть получен с помощью CS ;
    EXEC SQL FETCH CS INTO :CS_S#, :CS_SN, :CS_ST, :CS_SC ;
    вывести на печать CS_S#, CS_SN, CS_ST, CS_SC ;
    EXEC SQL OPEN CJ ;
    DO для всех строк J, доступ к которым может быть получен с помощью CJ

     EXEC SQL FETCH CJ INTO :CJ_J#, :CJ_JN, :CJ_JC ; вывести на печать CJ_J#,
     CJ_JN, CJ_JC ; END DO ;
    EXEC SQL CLOSE CJ ; END DO ;
EXEC SQL CLOSE CS ;
```

4.6. В данном случае основная проблема состоит в следующем: необходимо "разузловать" рассматриваемую деталь на компоненты вплоть до n-го уровня, притом что само значение n не известно. Теперь, после ввода в действие спецификации SQL: 1999, решение указанной задачи упростилось, поскольку в этой спецификации предусмотрена возможность записывать рекурсивные выражения. С использованием такого средства запрос может быть сформулирован, как показано ниже.

```
WITH RECURSIVE TEMP ( MINOR_P# ) AS ( ( SELECT MINOR_P# /*
  Первоначальный подзапрос */ FROM PART_STRUCTURE WHERE
  MAJOR_P# = :GIVENP# ) UNION
     ( SELECT PP.MINOR_P# /* Рекурсивный подзапрос */ FROM PP,
       TEMP
       WHERE PP.MAJOR_P# = TEMP.MINOR_P# ) )
SELECT DISTINCT MINOR_P# /* Окончательный подзапрос */ FROM TEMP
;
```

Но если СУБД не поддерживает рекурсивных выражений, то для выполнения этого задания необходимо написать программу. В качестве одного из решений можно предложить примерно следующую рекурсивную программу (на псевдокоде).

```
CALL RECURSION ( GIVENP* ) ;

RECURSION: PROC ( UPPER_P# ) RECURSIVE ; DCL UPPER_P# ...
      ; DCL LOWER_P# ... ; EXEC SQL DECLARE C "reopenable"
      CURSOR FOR
                  SELECT MINOR_P#
                  FROM PART_STRUCTURE
                  WHERE MAJOR_P# = :UPPER_P# ;

      вывести на печать UPPER_P# ; EXEC
      SQL OPEN C ;
        DO для всех строк PART_STRUCTURE, доступ к которым может быть
получен с помощью C ;
      EXEC SQL FETCH C INTO :LOWER_P# ; CALL
      RECURSION ( LOWER_P# ) ; END DO ;
      EXEC SQL CLOSE C ; END
PROC ;
```

В данном случае при каждом рекурсивном вызове создается новый курсор; предполагается, что (фиктивная) спецификация "reopenable" (переоткрываемый) в конструкции DECLARE CURSOR указывает на допустимость применения команды OPEN (открыть) к этому курсору, даже если он уже открыт, и что результатом выполнения такой команды OPEN становится создание нового экземпляра курсора для указанного табличного выражения (с использованием текущих значений всех переменных базового языка, на которые имеется ссылка в этом выражении). Кроме того, предполагается, что ссылки на такой курсор в операторе FETCH (и тому подобных операторах) являются ссылками на "текущий" экземпляр и что команда CLOSE уничтожает данный экземпляр и восстанавливает статус предыдущего экземпляра как "текущего". Иными словами, предполагается, что переоткрываемый курсор образует стек, а команды OPEN и CLOSE служат для этого стека в качестве операторов "push" (протолкнуть) и "pop" (вытолкнуть).

К сожалению, приведенные здесь предположения в настоящее время являются чисто гипотетическими. В наши дни в языке SQL нет такого понятия, как переоткрываемый курсор (в действительности, попытка применить команду OPEN для открытия уже открытого курсора оканчивается неудачей). Приведенный выше код является недопустимым. Но этот пример позволяет наглядно

продемонстрировать, что "переоткрываемые курсоры" были бы весьма желательным дополнением к существующей спецификации[1] SQL.

Поскольку описанный выше подход является неприменимым, здесь приведено краткое описание возможного (но весьма неэффективного) подхода, который можно продемонстрировать следующим образом.

```
CALL RECURSION ( GIVENP# ) ;

RECURSION: PROC ( UPPER_P# ) RECURSIVE ;
      DCL UPPER_P# ... ;
      DCL LOWER_P# ... INITIAL ( ' ' ) ;   EXEC SQL
      DECLARE C CURSOR FOR
                                 SELECT MINOR_P#
                                 FROM PART_STRUCTURE
                                  WHERE MAJOR_P# = :UPPER_P#
                                  AND MINOR_P# > :LOWER_P#
                                 ORDER BY MINOR_P# ;

      вывести на печать UPPER_P# ; DO "до
           бесконечности" ; EXEC SQL OPEN C ;
           EXEC SQL FETCH C INTO :LOWER_P# ;
           EXEC SQL CLOSE C ;
      IF не выполнена выборка "меньшего значения P#" THEN RETURN ; END IF ; ■ i -  IF выполнена
выборка "меньшего значения P#" THEN
          CALL RECURSION ( LOWER_P# ) ; END IF ; END DO ;
END PROC ;
```

В этом решении заслуживает внимания то, что при каждом вызове Процедуры RECURSION используется один и тот же курсор. (В отличие от этого, новые экземпляры переменных UPPER_P# и LOWER_p# создаются динамически при каждом вызове процедуры RECURSION; эти экземпляры уничтожаются после завершения указанного вызова.) В силу этого факта, для того, чтобы после каждого вызова процедуры RECURSION можно было игнорировать все непосредственные компоненты (детали LOWER_P#) текущей детали UPPER_P#, которые уже были обработаны, приходится использовать определенный прием следующим образом.

```
. . .  AND MINOR_P#  >  : LOWER_P#  ORDER  BY  MINOR_P#
```

Дополнительные примечания

а) В [4.4] приведено исчерпывающее описание альтернативного подхода к решению проблем, подобных рассматриваемой в данной упражнении, и дан краткий обзор (нереляционных) . расширений CONNECT BY и START WITH компании Oracle, которые также предназначены для решения проблемы указанного типа. (Объяснение того, по какой причине эти расширения Oracle, как уже было указано, фактически являются нереляционными, приведено в краткой статье " *The Importance of Closure"* из книги C.J.Date. *Relational Database Writings 1991-1994,* Reading, Mass.:Addison-Wesley, 1995.)

б) В [4.8] приведено подробное описание подхода к обработке рекурсивных запросов, приня того в СУБД DB2 компании IBM. Рекурсивные выражения, предложенные в спецификации SQL: 1999, основаны на этом подходе. К сожалению, подход, принятый в IBM, характеризуется

---

[1] Кстати, следует отметить, что решение, весьма подобное показанному в данном примере, возможно в спецификации SQLJ [4.7], т.е. в том случае, если в качестве базового языка применяется Java, поскольку курсоры в SQLJ заменяются "итераторными объектами" (iterator object) Java, которые могут записываться в стек при рекурсивных вызовах (автор выражает свою благодарность за эти замечания рецензенту, не указавшему своей фамилии).

наличием многочисленных ограничений, которые сложно понять, объяснить, оправдать или запомнить, но, с другой стороны, средства поддержки, предусмотренные в спецификации SQL: 1999, к счастью, устраняют большинство ограничений IBM или даже все эти ограничения.

В главе 7 (в конце раздела 7.8) описан соответствующий реляционный оператор, получивший название *транзитивное замыкание*.

ОТВЕТЫ К ГЛАВЕ 5

56.
```
OPERATOR FG ( P POINT ) RETURNS POINT ;
    RETURN ( CARTESIAN ( F ( THE_X ( P ) ) ,
                                G ( THE_Y ( P ) ) ) ) ;   END
OPERATOR ;
```

57.
```
OPERATOR FG ( P POINT ) UPDATES P ;
THE_X ( P ) := F ( THE_X ( P ) ) , THE_Y ( P ) := G
( THE_Y ( P ) ) ; END OPERATOR ;
```

Здесь заслуживает внимания то, что применяется множественное присваивание. Следует также отметить, что явно заданный оператор RETURN отсутствует; вместо этого после достижения оператора END OPERATOR выполняется неявно заданный оператор RETURN.

58.
```
TYPE LENGTH POSSREP { RATIONAL } ;
TYPE POINT POSSREP { X RATIONAL, Y RATIONAL } ; TYPE
CIRCLE POSSREP { R LENGTH, CTR POINT } ;
   /* Переменная R обозначает радиус окружности (вернее, его длину), */
   /* а CTR - координаты центра */
```

Ниже приведен единственный селектор, который применяется к типу CIRCLE.

```
CIRCLE  (  r,  ctr  )
/* Возвращает окружность с радиусом r и центром ctr */
```

Операторы THE_ перечислены ниже.

```
THE_R ( c )
/* Возвращает длину радиуса окружности c */
                                                    c

THE_CTR ( c )
/* Возвращает точку, которая является центром окружности c */
```

а)
```
OPERATOR DIAMETER ( C CIRCLE ) RETURNS LENGTH ;
      RETURN ( 2 * THE_R ( C ) ) ; END
   OPERATOR ;

   OPERATOR CIRCUMFERENCE ( C CIRCLE ) RETURNS LENGTH ;
      RETURN ( 3.14159 * DIAMETER ( C ) ) ; END
   OPERATOR ;

   OPERATOR AREA ( C CIRCLE ) RETURNS AREA ;
      RETURN ( 3.14159 * ( THE_R ( C ) ** 2 ) ) ; END
   OPERATOR;
```

В этих определениях операторов предполагается, во-первых, что операция умножения длины на целое число или число с плавающей точкой возвращает значение длины[2], а во-вторых, что операция умножения длины на длину возвращает значение площади (где AREA — еще один определяемый пользователем тип).

---

[2] *Тема для обсуждения.* Что произойдет, если целое число или число с плавающей точкой будет отрицательным или равным нулю?

```
б) OPERATOR DOUBLE_R ( C CIRCLE ) UPDATES C ;
       THE_R ( C ) := 2 * THE_R ( C ) ;
   END OPERATOR ;
```

5.9. Треугольник может быть представлен, например, тремя его вершинами или серединами трех его сторон[3], для представления отрезка прямой, в частности, можно использовать начальную и конечную точки или его середину, длину и наклон.

5.14.

а) Допустимое; BOOLEAN.

б) Недопустимое; NAME (THE_NAME (JNAME) |I THE_NAME (PNAME)).
Примечание. Идея здесь состоит в том, чтобы выполнить конкатенацию (возможных) *представлений в виде символьной строки,* а затем выполнить обратное "преобразование" результата этой конкатенации в значение типа NAME. Безусловно, если результат операции конкатенации не может быть преобразован в допустимое имя, то само это преобразование окончится неудачей из-за ошибки, связанной с несоответствием типов.

в) Допустимое; QTY.

г) Недопустимое; QTY + QTY (100).

д) Допустимое; INTEGER.

е) Допустимое; BOOLEAN.

ж) Недопустимое; THE_COLOR (COLOR) = P.CITY.

з) Допустимое.

ОТВЕТЫ К ГЛАВЕ 6

6.1. По существу, *кардинальность* — это концепция, которая применяется к множествам. Кардинальностью множества называется количество содержащихся в нем элементов. Но эта концепция была также распространена на другие виды "коллекций", что позволяет вести речь о кардинальности *мультимножества,* кардинальности *списка* и т.д. В частности, кардинальностью *отношения* называется количество кортежей в теле этого отношения, а кардинальностью *переменной отношения* — кардинальность отношения, которое является текущим значением этой переменной отношения. Иногда данный термин может даже применяться к какому-то атрибуту отношения. В таком случае он может рассматриваться, во-первых, как кардинальность мультимножества значений, присутствующих в данном атрибуте отношения, либо, во-вторых, как кардинальность множества значений (после устранения дубликатов).

Примечание. Поскольку первая интерпретация гарантирует получение результата, идентичного результату определения кардинальности отношения, содержащего данный атрибут, вероятно, чаще всего применяется вторая интерпретация, но при этом нельзя игнорировать возможность возникновения путаницы в отношении того, к чему относится термин кардинальность. Такая вероятность обусловлена также тем, что (еще раз подчеркиваем) кардинальность по сути представляет собой понятие, которое относится к множествам, а не к мультимножествам.

6.4. Соответствующие предикаты перечислены ниже.

- Предикат переменной отношения S. Поставщик S# работает по контракту, носит имя SNAME, имеет статус STATUS и находится в городе CITY.
- Предикат переменной отношения P. Деталь P# представляет для нас интерес[4], называется PNAME, имеет цвет COLOR и вес WEIGHT, хранится на складе в городе CITY.

---

[3] В качестве вспомогательного упражнения рекомендуем читателю попытаться доказать, что треугольник действительно можно однозначно представить, указав середины его сторон.

[4] По какой-то неуказанной причине!

■ Предикат переменной отношения J. Проект J# находится в стадии создания, имеет имя JNAME и осуществляется в городе CITY.

■ Предикат переменной отношения SPJ. Поставщик S# поставляет детали P# для проекта J# в количестве QTY.

Определения на языке Tutorial D приведены ниже.

```
VAR S BASE RELATION {
        S# S#,
SNAME NAME, STATUS INTEGER, CITY
CHAR }      PRIMARY KEY { S# );

VAR P BASE RELATION
      { P# P#,   PNAME NAME,
      COLOR COLOR, WEIGHT
      WEIGHT, CITY CHAR }
      PRIMARY KEY { P# } ;

VAR J BASE RELATION {
        J# J#,
        JNAME NAME, CITY CHAR )
        PRIMARY KEY ( J# } ;

VAR SPJ BASE RELATION { S#
         S#, P# P#,
J# J#,
 QTY QTY )
 PRIMARY KEY          {  S#, P#, J# )
 FOREIGN KEY          {  S# ) REFERENCES S
 FOREIGN KEY          {  P# } REFERENCES P
        FOREIGN KEY   (  J# } REFERENCES J ;
```

6.7. Все они представляют собой вызовы селектора отношения и обозначают, соответственно, а) пустое отношение такого же типа, как и у переменной отношения SPJ; б) отношение такого же типа, как и у переменной отношения SPJ, содержащее только один кортеж (где S# равно S1, P# равно P1, J# равно л и QTY равно 200); в) нуль-арное отношение, содержащее только один кортеж, или, другими словами, таблицу TABLE_DEE; г) то же, что и в); д) таблицу TABLE_DUM.

6.10. Некоторые возможные варианты показаны ниже.

а) Пример отношения с одним атрибутом, имеющим значения в виде отношения, приведен на рис.Д.3.

Но следует отметить, что отношение, подобное приведенному слева на рис. Д.3, позволяет представить данные о поставщике, который не поставляет никаких деталей, а отношение, показанное справа, — нет.

б) Пример отношения с двумя атрибутами, имеющими значения в виде отношения, приведен на рис. Д.4.

Более конкретный пример, напоминающий эту вторую пару отношений, будет обсуждаться в главе 13.

Рис. Д.4. Пример отношения с двумя атрибутами, имеющими значения в виде отношения

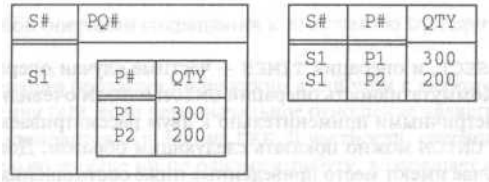
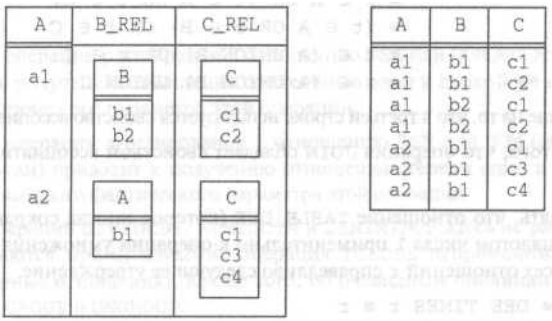


Рис. Д.3. Пример отношения с одним атрибутом, имеющим значения в виде отношения



6.11. P{} = TABLE_DUM.

Пояснение. В этом выражении с оператором сравнения левый операнд представляет собой проекцию переменной отношения P вообще ни по одному атрибуту. Выполнение такой операции проекции приведет к получению отношения TABLE_DEE, если P в настоящее время содержит не меньше одного кортежа, а в противном случае — отношения TABLE_DUM. 6.15.

а) INSERT SPJ RELATION { TUPLE { S# SJ('S1'), P# P#('P1'),
J# J#('J2'), QTY QTY(500) } } ;

б) INSERT S RELATION { TUPLE { S# S#('S10'), SNAME NAME('Smith'),
CITY 'New York[1] } } ;

В качестве значения статуса для нового поставщика будет задано соответствующее значение по умолчанию, если оно предусмотрено; в противном случае (т.е. если для атрибута STATUS указано, что "этому атрибуту нельзя присваивать значения по умолчанию"), операция INSERT окончится неудачей. Следует отметить, что такая ошибка (если это действительно ошибка), может быть выявлена на этапе компиляции.

В) DELETE P WHERE COLOR = COLOR('Blue') ;

Г) DELETE J WHERE IS_EMPTY (((J JOIN SPJ) RENAME J# AS X)
WHERE X = J#) ;

Это решение основано на использовании оператора RENAME, который рассматривается в главе 7.

Д) UPDATE P WHERE COLOR = COLOR('Red')
{ COLOR := COLOR('Orange') } ;

е) UPDATE SPJ WHERE S# = S#('SI') { S# := S#('S9') } ,
UPDATE S WHERE S# = S#('SI') { S# := S#('S9') } ;

Следует отметить, что здесь возникает необходимость использовать множественное присваивание (если бы использовались два отдельных оператора UPDATE, это привело бы к нарушению целостности внешнего ключа).

ОТВЕТЫ К ГЛАВЕ 7

7.4. И операция INTERSECT, и операция TIMES — частные случаи операции JOIN, здесь их можно не рассматривать. Коммутативность операций UNION и JOIN очевидна из их определений, которые являются симметричными применительно к двум рассматриваемым отношениям. Ассоциативность операции UNION можно показать следующим образом. Допустим, что t — некоторый кортеж. В таком случае имеют место приведенные ниже соотношения[5].

```
t G A UNION (B UNION C ) ≡ t ∈A O R t ∈ (B UNION C)
                         ≡ t ∈A O R ( t ∈B O R t ∈C )
                         ≡ ( t ∈A O R t ∈ B) 0R t ∈ C
                         = t ∈(A UNION B) OR t ∈ C
                         ≡ t ∈(A UNION B) UNION C
```

Обратите внимание на то, что в третьей строке используется свойство ассоциативности оператора OR.

Доказательство того, что операция JOIN обладает свойством ассоциативности, является аналогичным.

7.8. Можно утверждать, что отношение TABLE_DEE (которое иногда сокращенно обозначается как DEE) является аналогом числа 1 применительно к операции умножения в обычной арифметике, поскольку для всех отношений r справедливо следующее утверждение.

r TIMES DEE ≡ DEE TIMES r = r

Иными словами, отношение DEE представляет собой **единицу** применительно к операции TIMES и, вообще говоря, применительно к операции JOIN. Однако не существует никакого отношения, которое действовало бы применительно к операции TIMES полностью аналогично тому, как действует число 0 применительно к операции умножения. Тем не менее, результаты применения отношения TABLE_DUM (которое иногда сокращенно обозначается как DUM) немного напоминают то, как действует число 0, поскольку для всех отношений r справедливо следующее утверждение.

r TIMES DUM ≡ DUM TIMES r ≡ пустое отношение с таким же заголовком, как и у отношения r

Теперь рассмотрим, каковы результаты применения операторов реляционной алгебры к отношениям DEE и DUM. Прежде всего, следует отметить, что единственными отношениями, которые имеют такой же тип, как DEE и DUM, являются сами отношения DEE и DUM. Соответствующие результаты приведены ниже.

UNION	DEE	DUM
DEE	DEE	DEE
DUM	DEE	DUM

INTERSECT	DEE	DUM
DEE	DEE	DUM
DUM	DUM	DUM

MINUS	DEE	DUM
DEE	DUM	DEE
DUM	DUM	DUM

В случае оператора MINUS первый операнд показан слева, а второй — сверху (безусловно, при использовании других операторов операнды являются взаимозаменяемыми). Здесь заслуживает внимания то, насколько эти таблицы напоминают таблицы истинности, соответственно, для операторов OR, AND и AND NOT; разумеется, такое подобие — не простое совпадение. Что касается операций сокращения и проекции, то для них справедливы приведенные ниже утверждения.

- Применение любой операции сокращения к отношению DEE приводит к получению отношения DEE, если условие сокращения принимает значение TRUE, и отношения DUM, если это условие принимает значение FALSE.

---

[5] Символ "≡" обозначает "тогда и только тогда, когда".

ш Применение любой операции сокращения к отношению DUM приводит к получению отношения DUM.

- Применение операции проекции ни по каким атрибутам к любому отношению приводит к получению отношения DUM, если первоначальное отношение является пустым, в противном случае результатом становится отношение DEE. В частности, операция проекции отношения DEE или DUM, обязательно вообще ни по одному атрибуту, возвращает ее входной фактический параметр.

Что касается операций расширения и агрегирования, то для них справедливы приведенные ниже утверждения.

- Применение операции расширения к отношению DEE или DUM для добавления нового атрибута приводит к получению отношения со степенью один и с такой же кардинальностью, как и у входного фактического параметра этой операции.
- Применение операции агрегирования к отношению DEE или DUM (обязательно вообще ни по каким атрибутам) приводит к получению отношения степени один и с такой же кардинальностью, как и у входного фактического параметра этой операции.

Примечание. Операции DIVIDEBY, SEMI JOIN и SEMIMINUS здесь не рассматриваются, поскольку они не являются примитивными. Операция TCLOSE неприменима (она распространяется только на бинарные отношения). Кроме того, по очевидным причинам из рассмотрения исключены операции GROUP и UNGROUP.

7.11. В каждом случае результатом становится отношение степени один. Если отношение r является непустым, то все четыре выражения возвращают отношение с одним кортежем, который содержит данные о кардинальности n отношения r. Если отношение r является пустым, то выражения а) и в) возвращают пустой результат, тогда как выражения б) и г) оба возвращают отношение с одним кортежем, содержащим нуль (кардинальность отношения r).

7.13. J

7.16. SPJ WHERE QTY > QTY (300) AND QTY < QTY (750)

7.19.
```
( ( ( S RENAME CITY AS SCITY ) TIMES
    ( P RENAME CITY AS PCITY ) TIMES
    ( J RENAME CITY AS JCITY ) )
WHERE SCITY ≠ PCITY
OR    PCITY ≠ JCITY
OR    JCITY ≠ SCITY ) { S#, P#, J# }
```

7.22. Ниже приведено поэтапное решение данного упражнения (с использованием обозначений для промежуточных выражений), которое позволяет еще раз показать, насколько удобным является такой способ формирования сложных запросов.

```
WITH ( S WHERE CITY = 'London' ) AS T1,
     ( J WHERE CITY = 'London' ) AS T2,
     ( SPJ JOIN T1 ) AS T3, T3 { P#,
     J# } AS T4,
     ( T4 JOIN T2 ) AS T5 : T5 { P# }
```

Ниже показан тот же запрос, в котором не используется конструкция WITH.

```
( ( SPJ JOIN ( S WHERE CITY = 'London' ) ) { P#, J# }
                    JOIN ( J WHERE CITY = 'London' ) ) { P# }
```

Решения к остальным упражнениям будут приведены в смешанной форме (в одних решениях конструкция WITH используется, а в других — нет).

```
7.25. ( ( ( J RENAME CITY AS JCITY ) JOIN SPJ JOIN ( S RENAME CITY
            AS SCITY ) )
                                        WHERE JCITY ≠ SCITY ) { J# }

7.28. ( SUMMARIZE SPJ { S#, P#, QTY }
          PER RELATION { TUPLE { S# S# ( 'S1' ), P# P# ( 'P1' ) } } ADD SUM ( QTY ) AS Q
          ) { Q }

7.31. ( J JOIN ( SPJ WHERE S# = S# ( 'S1' ) ) ) { JNAME }

7.34. ( SPJ JOIN ( SPJ WHERE S# = S# ( 'S1' ) ) { P# } ) { J# }

7.37. ( ( EXTEND J ADD MIN ( J, CITY ) AS FIRST ) WHERE CITY = FIRST
        ) { J# }

7.40. WITH (      S WHERE CITY = 'London' ) { S# } AS T1,
                ( P WHERE COLOR = COLOR ( 'Red' ) ) AS T2,
      (           T1 JOIN SPJ JOIN T2 ) AS T3 :
                J { J# } MINUS T3 { J# }

7.43. S { S#, P# } DIVIDEBY J { J# } PER SPJ { S#, P#, J# }

7.46. ( SPJ JOIN ( S WHERE CITY = 'London' ) ) { P# }
          UNION ( SPJ JOIN ( J WHERE CITY = 'London' ) ) { P# }

7.49. SPJ GROUP ( J#, QTY ) AS JQ
```

ОТВЕТЫ К ГЛАВЕ 8

8.1. а) Не верно; б) не верно; в) верно; г) верно; д) не верно; е) не верно; ж) не верно.

Примечание. Причина, по которой утверждение д) не верно, состоит в том, что применение квантора FORALL к пустому множеству всегда приводит к получению значения TRUE. И наоборот, применение квантора EXISTS к пустому множеству всегда приводит к получению значения FALSE. Поэтому, например, если утверждение "Все детали фиолетового цвета весят больше 100 фунтов" истинно, это не обязательно означает, что действительно существуют какие-либо фиолетовые детали.

Следует отметить, что эти тождества и импликации (но только верные!) могут использоваться в качестве основы для множества правил преобразования выражений исчисления, во многом аналогично правилам преобразования алгебраических выражений, которые упоминались в главе 7 и подробно обсуждались в главе 18. Аналогичное замечание относится также к ответам на упр. 8.2 ' и 8.3.

8.2. а) Верно; б) верно; в) верно (этот пример рассматривался в основном тексте главы); г) верно (следовательно, каждый из этих кванторов может быть определен в терминах другого); д) не вер но; е) верно.

Примечание. Как показывают упр. а) и б), последовательность **одинаковых** кванторов можно записывать в любом порядке, и от этого смысл выражения не изменяется. Вместе с тем, как показывает упр. д), для **разных** кванторов порядок играет важную роль. В качестве иллюстрации последнего утверждения можно привести один пример. Допустим, что x и y пробегают множество целых чисел, а p — правильно построенная формула "y > x". В таком случае должно быть очевидно, что следующая правильно построенная формула (которая имеет смысл "для всех целых **x** существует целое y, большее x").

FORALL x EXISTS y (y > x)

принимает значение TRUE, тогда как приведенная ниже правильно построенная формула (которая имеет смысл "для всех целых x существует целое y, большее x") принимает значение FALSE.

EXISTS y FORALL X { y > X)

Таким образом, перестановка неодинаковых кванторов приводит к изменению смысла выражения. В связи с этим изменяется и смысл запроса на языке запросов, основанном на реляционном исчислении, после перестановки неодинаковых кванторов в предложении WHERE (см. [8.3]).

8.11. В языке SQL операторы реляционных сравнений непосредственно не поддерживаются. Однако такие операции можно промоделировать, хотя лишь только весьма громоздким способом. Например, следующую операцию сравнения (где A и B — переменные отношения).

A = B

можно промоделировать с помощью представленного выражения SQL.

```
NOT EXISTS ( SELECT * FROM A
                WHERE NOT EXISTS      ( SELECT * FROM B
                                        WHERE A-row = B-row ) )
AND
NOT EXISTS ( SELECT * FROM B
                WHERE NOT EXISTS      ( SELECT * FROM A
                                        WHERE B-row = A-row ) )
```

A-row и B-row— это выражения конструктора значения строки <*row value constructor* представляющие, соответственно, всю строку A и всю строку B.

8.13. Приведенные ниже решения перенумерованы как 8.13.П, где п — номер исходного упражнения] главе 7, т.е. упр. 7.п. Предполагается, что SX, SY, PX, PY, JX, JY, SPJX, SPJY (и т.д.) — перемен ные области значений, пробегающие, соответственно, отношения поставщиков, деталей, проек тов и поставок; определения этих переменных области значений в ответах не показаны, i

8.13.13. JX

8.13.16. SPJX WHERE SPJX.QTY > QTY (300) AND SPJX.QTY < QTY (750)

8.13.19. { SX.S#, PX.P#, JX.J# } WHERE SX.-CITY ≠ PX.CITY OR PX.CITY ≠ JX.CITY OR JX.CITY ≠ SX.CITY

8.13.22. SPJX.P# WHERE EXISTS SX EXISTS JX (SX.S# = SPJX.S# AND SX.CITY = 'London' AND JX.J# = SPJX.J# AND JX.CITY = 'London')

8.13.25. SPJX.J* WHERE EXISTS SX EXISTS JX (SX.CITY ≠ JX.CITY AND SPJX.S# = SX.S# AND SPJX.J# = JX.J#)

8.13.28. SUM (SPJX WHERE SPJX.S# = S# ('SI[1]) AND SPJX.P» = P# ('P1'), QTY) AS Q

Примечание. Следующее "решение" не верно (объясните, почему).

SUM (SPJX.QTY WHERE SPJX. S# = S# ('S1') AND SPJX.P# = P# ('P1[1])) AS Q

Ответ. Поскольку теперь дубликаты значений QTY будут устранены перед вычислением суммы.

8.13.31. JX.JNAME WHERE **EXISTS SPJX** (SPJX.J# = JX.J# AND SPJX.S# = S# ('S 1'))

8.13.34. SPJX.J# WHERE EXISTS SPJY (SPJX.P# = SPJY.P# AND SPJY.S# = S# ('S1'))

8.13.37. JX.J# WHERE FORALL JY (JY.CITY > JX.CITY) Или

JX.J# WHERE JX.CITY = MIN (JY.CITY)

8.13.40. JX.J# WHERE NOT EXISTS SPJX EXISTS SX EXISTS PX (SX.CITY = 'London' AND PX.COLOR = COLOR ('Red') AND SPJX.S# = SX.S# AND SPJX.P* = PX.P# AND SPJX.J# = JX.J#)

8.13.43. SX.S# WHERE EXISTS PX FORALL JX EXISTS SPJY (SPJY.S# = SX.S# AND SPJY.P* = PX.P# AND SPJY.J# = JX.J#)

8.13.46. SPJX.P# WHERE EXISTS SX (SX.S# = SPJX.S* AND SX.CITY = 'London') OR . EXISTS JX (JX.J# = SPJX.J* AND JX.CITY = 'London').

8.13.49. { SPJX.S#, SPJX.P*, { SPJY.J*, SPJY.QTY WHERE ' SPJY.S# = SPJX.S# AND SPJY.P* = SPJX.P» } AS JQ }

8.14. Приведенные ниже решения перенумерованы как8.14.п, гдеп — номер исходного упражнения в главе 7, т.е. упр. *Т.п.*

8.14.13.
```
SELECT *
FROM   J ;
```
Или просто
```
TABLE J ;
```

8.14.16.
```
SELECT SPJ.*
FROM   SPJ
WHERE  SPJ.QTY  >=  QTY(3 00)
AND    SPJ.QTY  <=  QTY(750)  ;
```

8.14.19.
```
SELECT  S.S#,  P.P#,  J.J#
FROM    S,  P,  J
WHERE  NOT ( S.CITY = P. CITY AND P.CITY
           = J.CITY ) ;
```

8.14.22.
```
SELECT DISTINCT SPJ.P* FROM
    SPJ WHERE ( SELECT S.CITY
                FROM  S
                WHERE  S.S# = SPJ.S* ) = 'London' AND  (
    SELECT J.CITY
                FROM  J
                WHERE  J.J# = SPJ.J# ) = 'London' ;
```

8.14.25.
```
SELECT DISTINCT SPJ.J* FROM
    SPJ
    WHERE ( SELECT S.CITY
                FROM S
                  WHERE S.S* = SPJ.S* ) <> (
               SELECT J.CITY FROM J WHERE
               J.J# = SPJ.J* ) ;
```

8.14.28.
```
SELECT    SUM ( SPJ.QTY ) AS X
  FROM    SPJ
  WHERE   SPJ.S# = S№('Sl')
  AND     SPJ.P# = P#('P1') ;
```

8.14.31.
```
SELECT DISTINCT J.JNAME FROM  J,
  SPJ WHERE  J.J# = SPJ.J# AND
  SPJ.S# = St('Sl')
```

8.14.34.
```
SELECT DISTINCT SPJX.J#
  FROM  SPJ AS SPJX, SPJ AS SPJY WHERE
  SPJX.P# = SPJY.P# AND  SPJY.S* = S#('S1' ) ;
```

8.14.37.
```
SELECT J.J#
  FROM  J
  WHERE  J.CITY = ( SELECT MIN ( J.CITY ) FROM  J ) ;
```

8.14.40.
```
SELECT J.J# FROM  J WHERE
  NOT EXISTS
            ( SELECT *
              FROM  SPJ, P, S
      WHERE SPJ.J# = J.J# AND   SPJ.P# = P.P# AND
              SPJ.S# = S.S# AND   P.COLOR =
              COLOR('Red') AND   S. CITY = ' London' ) ;

  SELECT S.S#
  FROM  S WHERE
  EXISTS
    ( SELECT * FROM  P     WHERE
    NOT EXISTS
                    ( SELECT *
   FROM J
            WHERE NOT EXISTS
                              ( SELECT *
                                FROM  SPJ
                    WHERE  SPJ.S# = S.S# AND  SPJ.P# = P.P#
                    AND  SPJ.J# = J.J# ) ) ) ;
```

8.14.47.
```
SELECT S.S#, P.P# FROM  S, P
  EXCEPT
  SELECT SPJ.S#, SPJ.P# FROM
  SPJ ;  ;
```

8.15. Приведенные ниже решения перенумерованы как 8.15.П, где п — номер исходного упражнения в главе 7, т.е. упр. 7.п. Что касается определений и имен переменных области значений, то применяются такие же соглашения, как и в разделе 8.7.

8.15.13.
```
( JX, NAMEX, CITYX )
WHERE J ( J#:JX, JNAME:NAMEX, CITY:CITYX )
```

8.15.16.
```
( SX, PX, JX, QTYX )
  WHERE SPJ ( S#:SX, P#:PX, J#:JX, QTY:QTYX ) AND  QTYX >
  QTY ( 300 ) AND QTYX < QTY ( 750 )
```

8.15.19. (SX, PX, JX)
```
WHERE EXISTS CITYX EXISTS CITYY EXISTS CITYZ
                ( S ( S#:SX, CITY:CITYX ) AND
                  P ( P#:PX, CITY:CITYY ) AND
                    J ( J#:JX, CITY:CITYZ )
                  AND ( CITYX ≠ CITYY OR CITYY ≠ CITYZ OR CITYZ ≠ CITYX )
                )
```

8.15.22. PX WHERE EXISTS SX EXISTS JX
```
                ( SPJ ( S#:SX, P#:PX, J#:JX )
        AND S ( S#:SX, CITY:'London' )
                  AND J ( J#:JX, CITY:'London' )
```

8.15.25. JY WHERE EXISTS SX EXISTS CITYX EXISTS CITYY
```
                ( SPJ ( S#:SX, J#:JY )        AND S ( S#:SX, CITY:CITYX )
                  AND J ( J#:JY, CITY:CITYY )
    AND CITYX ≠ CITYY )
```

8.15.31. NAMEX WHERE EXISTS JX
```
                ( J ( J#:JX, JNAME:NAMEX )
                  AND SPJ ( S#:S#('S1'), J#:JX ) )
```

8.15.34. JX WHERE EXISTS PX
```
                ( SPJ ( P#:PX, J#:JX ) AND
                  SPJ ( P#:PX, S#:S#('S1') ) }
```

8.15.37. JX WHERE EXISTS CITYX
```
                ( J ( J#:JX, CITY:CITYX ) AND
                  FORALL CITYY ( IF J ( CITY:CITYY )
                                 THEN CITYY ≥ CITYX
                                 END IF )
```

8.15.40. JX WHERE J (J#:JX) AND
```
    NOT                 EXISTS SX EXISTS PX
                (       SPJ ( S#:SX, P#:PX, J#:JX > AND
                        S ( S#:SX, CITY:'London' ) AND
                        P ( P#:PX, COLOR:COLOR!'Red' ) ) )
```

8.15.43. SX WHERE S (S#:SX)
```
            AND EXISTS PX FORALL JX
            . ( SPJ ( S#:SX, P#:PX, J#:JX ) )
```

8.15.46. PX WHERE EXISTS SX (SPJ (S#:SX, P#: PX) AND S (S#:SX, CITY:'London'))
```
            OR  EXISTS JX ( SPJ ( J#:JX, P#: PX ) AND
                            J ( J#:JX, CITY:'London' ) )
```

ОТВЕТЫ К ГЛАВЕ 9

a) TYPE CITY
```
   POSSREP { C CHAR CONSTRAINT C = 'London'
                               OR C = 'Paris'
                               OR C = 'Rome'
                                  OR C = 'Athens'
                               OR C = 'Oslo'
                               OR C = 'Stockholm'
                               OR C = 'Madrid'
                               OR C = 'Amsterdam' }
```

Очевидно, что в данном случае можно также применить следующее сокращение.

б) TYPE CITY
POSSREP { C CHAR CONSTRAINT C IN { 'London[1], 'Paris',
[1] 'Rome', 'Athens', 'Oslo', 'Stockholm', 'Madrid',
'Amsterdam' } ;

Примечание. Лучшее решение может состоять в следующем: сохранить допустимые названия городов в переменной отношения и использовать внешние ключи для обеспечения того, что бы ни одна другая переменная отношения не включала название города, которое не является одним из допустимых (этот подход, по-видимому, позволял бы проще учесть такую ситуацию, когда становится допустимым новое название города). Таким образом, указанное решение дало бы возможность заменить приведенное выше ограничение типа (и соответствующие ог раничения атрибутов) множеством ограничений базы данных.

В) TYPE S# POSSREP { C CHAR CONSTRAINT
LENGTH (C) > 2 AND LENGTH (C) < 5 AND SUBSTR
(C, 1, 1) = 'S' AND CAST_AS_INTEGER (SUBSTR (C,
2) > 0 AND CAST_AS_INTEGER (SUBSTR (C, 2) <
9999 } ;

Здесь предполагается, что операторы LENGTH, SUBSTR и CAST_AS_INTEGER доступны и имеют очевидную семантику.

Г) CONSTRAINT C FORALL PX (IF PX. COLOR = COLOR ('Red')
THEN PX.WEIGHT < WEIGHT (50.0) END IF) ;

Здесь и во всех остальных ответах применяются обычные соглашения, касающиеся определений и имен переменных области значений.

Д) CONSTRAINT D
FORALL JX FORALL JY (IF JX.J# ≠ JY.J#
THEN JX.CITY ≠ JY.CITY END IF) ;

е) CONSTRAINT E COUNT (SX WHERE SX.CITY = 'Athens') < 1 r

Ж) CONSTRAINT F
FORALL SPJX (SPJX.QTY ≤ 2 * AVG (SPJY, QTY)) ;

3) CONSTRAINT G
FORALL SX FORALL SY (IF SX. STATUS = MAX (S, STATUS) AND
SY. STATUS = MIN (S, STATUS)
THEN SX.CITY ≠ SY.CITY END IF) ;

Фактически термины "поставщик с самым высоким статусом" и "поставщик с самым низким статусом" недостаточно четко определены, поскольку значения статуса не уникальны. Здесь требование соответствующего задания интерпретируется так, что если Sx и Sy — любые поставщики, соответственно, с "самым высоким статусом" и с "самым низким статусом", то Sx и Sy не должны находиться в одном и том же городе. Следует отметить, что это ограничение обязательно будет нарушено, если "самый высокий" и "самый низкий" статус равны! В частности, оно будет нарушено, если в городе есть только один поставщик. Эту проблему можно устранить, вставив выражение AND SX. STATUS ≠ SY . STATUS непосредственно перед THEN.

И) CONSTRAINT H FORALL JX EXISTS SX (SX.CITY = JX.CITY) ;

К) CONSTRAINT I FORALL JX EXISTS SX EXISTS SPJX (SX.CITY =
JX.CITY AND
SX.S# = SPJX.S# AND SPJX.J# = JX. J#) ;

Л) CONSTRAINT J EXISTS PX (PX.COLOR = COLOR ('Red')) ;

Это ограничение будет нарушено, если вообще нет никаких деталей. Лучшая формулировка может быть представлена следующим образом.

```
CONSTRAINT J NOT EXISTS PX ( TRUE ) OR
                EXISTS PX ( PX. COLOR = COLOR ( 'Red' ) ) ;
```

M) CONSTRAINT K FORALL SX (AVG (SY, STATUS) > 19) ;

Здесь приведенная в начале конструкция "FORALL SX" позволяет избежать ошибки, кото-*рая* в противном случае возникала бы при попытке проверить это ограничение в системе, когда вообще не существует ни одного поставщика.

```
H) CONSTRAINT L
     FORALL SX ( IF SX.CITY = 'London'          THEN
                 EXISTS SPJX ( SPJX.S#          = SX.S# AND
                                 SPJX.P#    = P# ( 'P2' ) END IF ) ;
```

```
O) CONSTRAINT M NOT EXISTS PX ( PX. COLOR = COLOR ( 'Red' ) ) OR EXISTS PX (
                 PX.COLOR = COLOR ( 'Red' ) AND PX.WEIGHT < WEIGHT ( 50.0
                 ) ) ;
```

```
П) CONSTRAINT N      COUNT ( SPJX.P#
WHERE
                 EXISTS SX ( SX.S# = SPJX.S# AND
                                 SX.CITY  =  'London'   )   )  >
        COUNT  (  SPJY.P#  WHERE
                 EXISTS SY ( SY.S# = SPJY.S# AND
                                 SY.CITY = 'Paris' ) ) ;
```

```
p) CONSTRAINT O
        SUM ( SPJX WHERE
              EXISTS SX ( SX.S# = SPJX.S* AND
SX.CITY = 'London' ), QTY ) >
        SUM ( SPJY WHERE
              EXISTS SY ( SY.S# = SPJY.S# AND
                            SY.CITY = 'Paris' ), QTY ) ;
```

```
C) CONSTRAINT P
     FORALL SPJX' FORALL SPJX ( SPJX' . S# ≠ SPJX.S# OR
    SPJX'.P# ≠ SPJX.P# OR SPJX'.J# ≠ SPJX.J# OR                0.5 * SPJX'.QTY < SPJX.
    QTY ) ;
```

```
T) CONSTRAINT Q
     FORALL SX' FORALL SX ( SX'.S# ≠ SX.S# OR
                        ( IF SX'.CITY =        'Athens' THEN
                              SX. CITY =       'Athens' OR
                              SX. CITY =       'London' OR
                              SX. CITY =       'Paris' END IF ) OR
                        (  IF  SX'.CITY  =  'London'  THEN
                              SX.  CITY  =  'London'  OR
                              SX.  CITY  =  "Paris'  END  IF  )  )
```

а) Допустимая операция.

б) Недопустимая операция (нарушение уникальности потенциального ключа).

В) Недопустимая операция (нарушение спецификации RESTRICT).

г) Допустимая операция (удаление данных о поставщике S3 и обо всех поставках поставщика S3).

д) Недопустимая операция (нарушение спецификации RESTRICT).

е) Допустимая операция (удаление данных о проекте J4 и обо всех поставках для проекта J4).

ж) Допустимая операция.

з) Недопустимая операция (нарушение уникальности потенциального ключа).

и) Недопустимая операция (нарушение ссылочной целостности).

к) Допустимая операция.

л) Недопустимая операция (нарушение ссылочной целостности).

м) Недопустимая операция (нарушение ссылочной целостности — в переменной отношения J не существует заданный по умолчанию номер проекта j j j).

9.7. Ссылочная диаграмма показана на рис. Д.5, а возможное определение базы данных приведено ниже. Для простоты здесь не определены какие-либо ограничения типа, безусловно, если не считать того, что в данном определении типа априорным ограничением типа служит спецификация POSSREP.

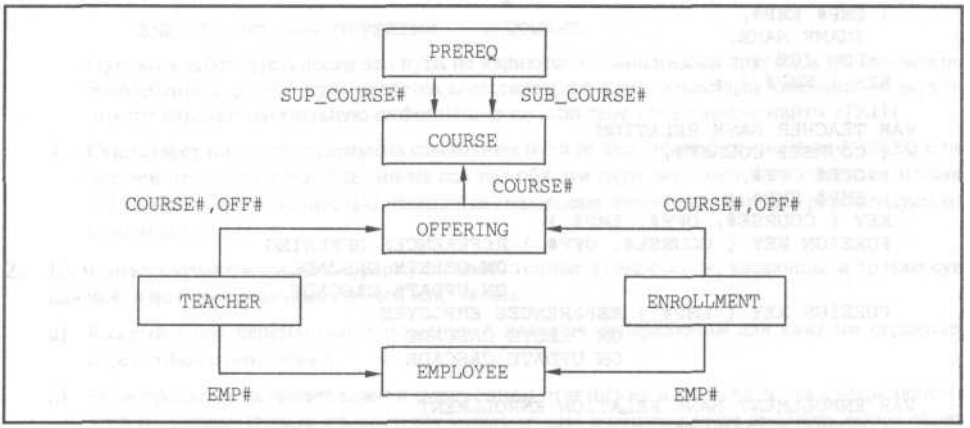


Рис. Д.5. Ссылочная диаграмма

```
TYPE COURSE# POSSREP   { CHAR   } ;
TYPE TITLE   POSSREP   { CHAR   } ;
TYPE OFF#    POSSREP   { CHAR   } ;
TYPE OFFDATE POSSREP   { DATE   } ;
TYPE CITY    POSSREP   { CHAR   } ;
TYPE EMP#    POSSREP   { CHAR   } ;
TYPE NAME    POSSREP   { NAME   } ;
TYPE JOB     POSSREP   { CHAR   } ;
TYPE GRADE   POSSREP   { CHAR   } ;

VAR COURSE BASE RELATION {
    COURSE# COURSE#,
       TITLE TITLE } PRIMARY KEY {
    COURSE# } ;

VAR PREREQ BASE RELATION {
    SUP_COURSE# COURSE#,
       SUB_COURSE# COURSE# } KEY { SUP_COURSE#,
    SUB_COURSE# } FOREIGN KEY { RENAME SUP_COURSE# AS
    COURSE# }
                                          REFERENCES COURSE
                                          ON DELETE CASCADE
```

```
                                                    ON UPDATE CASCADE
FOREIGN KEY { RENAME SUB_COURSE# AS COURSE» }
                                                    REFERENCES    COURSE
                                                    ON  DELETE  CASCADE
                                                    ON UPDATE CASCADE ;

 VAR OFFERING BASE RELATION {
     COURSE# COURSE»,
        OFF# OFF#,
OFFDATE OFFDATE,
        LOCATION CITY } KEY {
    COURSE#, OFF# }
  FOREIGN KEY { COURSE» } REFERENCES COURSE
                                    ON DELETE CASCADE
                                    ON UPDATE CASCADE ;

 VAR EMPLOYEE BASE RELATION {
    EMP# EMP#, ENAME NAME, JOB JOB
    } KEY { EMP# } ;

 VAR TEACHER BASE RELATION {
     COURSE# COURSE#, OFF# OFF#,
     EMP# EMP# }
    KEY { COURSE#, OFF#, EMP# }
    FOREIGN KEY { COURSE#, OFF# } REFERENCES OFFERING
    ON DELETE CASCADE ON UPDATE CASCADE FOREIGN KEY {
    EMP# } REFERENCES EMPLOYEE
                                ON DELETE CASCADE
                          ON UPDATE CASCADE ;

 VAR ENROLLMENT BASE RELATION ENROLLMENT
     { COURSE# COURSE»,
       OFF#  OFF#,  EMP#
       EMP»,        GRADE
       GRADE }
    KEY { COURSE#, OFF#, EMP# }
    FOREIGN KEY { COURSE#, OFF# } REFERENCES OFFERING
                                            ON DELETE CASCADE ON
                                            UPDATE CASCADE
    FOREIGN KEY { EMP# } REFERENCES EMPLOYEE
                               ON DELETE CASCADE
                                  ON UPDATE  CASCADE  ;
```

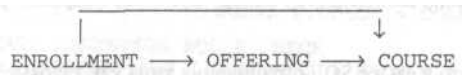
Пояснения

1. Такие (одноэлементные) множества атрибутов, как {COURSE#} в переменной отношения TEACHER и {COURSE#} в переменной отношения ENROLLMENT, могут также рассматриваться как внешние ключи, поскольку оба соответствующих атрибута ссылаются на переменную отношения COURSE. Однако, если все ограничения ссылочной целостности, направленные от переменной отношения TEACHER к переменной отношения OFFERING, от переменной отношения ENROLLMENT к переменной отношения OFFERING и от переменной отношения OFFERING к переменной отношения COURSE, поддерживаются должным образом, то ограничения ссылочной целостности, направленные от переменной отношения TEACHER к переменной отношения COURSE и от переменной отношения ENROLLMENT к переменной отношения COURSE, будут поддерживаться автоматически. Дополнительная информация на эту тему приведена в [9.11].

2. Здесь OFFERING — это пример переменной отношения, которая одновременно является и ссылающейся, и ссылочной, поскольку существует и ограничение ссылочной целостности, направленное к переменной отношения OFFERING от переменной отношения ENROLLMENT (а также фактически от переменной отношения TEACHER), и ограничения ссылочной целостности, направленное от переменной отношения OFFERING к переменной отношения COURSE.

   ENROLLMENT → OFFERING → COURSE

3. Следует отметить, что существует два разных ссылочных пути от переменной отношения ENROLLMENT к переменной отношения COURSE: во-первых, прямой (внешний ключ {COURSE*} в переменной отношения ENROLLMENT), а во-вторых, косвенный, через переменную отношения OFFERING (внешние ключи {COURSE#,OFF#} в переменной отношения ENROLLMENT и {COURSE*} в переменной отношения OFFERING).



Однако в действительности эти пути не являются независимыми друг от друга (возможность применения верхнего пути является следствием того, что существует комбинация двух нижних путей). Дополнительную информацию по этой теме также можно найти в [9.11].

4. Существует также два отдельных ссылочных пути от переменной отношения PREREQ к переменной отношения COURSE, но на сей раз оба эти пути действительно являются независимыми (они имеют полностью отдельные смысловые значения). Еще раз рекомендуем ознакомиться с [9.11].

9.12. Во многих случаях возможно сформулировать точные утверждения, касающиеся только суперключей, а не потенциальных ключей как таковых.

а) Каждый ключ переменной отношения A является суперключом для каждого ограничения переменной отношения A.

б) Если проекция включает ключ K переменной отношения A, то K является суперключом для этой проекции. Иными словами эту мысль можно в общем случае выразить таким образом: комбинация всех атрибутов проекции является суперключом для данной проекции.

в) Каждая комбинация к ключа KA переменной отношения A и ключа KB переменной отношения B является ключом для произведения A TIMES B.

г) Комбинация всех атрибутов является суперключом для объединения A UNION B.

д) Каждый ключ переменной отношения A или переменной отношения B является суперключом для пересечения A INTERSECT B.

е) Каждый ключ переменной отношения A является суперключом для разности A MINUS B.

ж) Каждая комбинация K ключа KA переменной отношения A и ключа KB переменной отношения B является суперключом для соединения A JOIN B.
Примечание. В том частном случае, когда атрибуты соединения в переменной отношения A включают ключ переменной отношения A, каждый ключ переменной отношения B является суперключом для этого соединения.

з) Каждый ключ переменной отношения A является ключом для каждого расширения переменной отношения A.

и) Каждый ключ переменной отношения B является суперключом для результата произвольной операции агрегирования переменной отношения A "по переменной отношения B".

к) Каждый ключ переменной отношения A является суперключом для полуобъединения A SEMIJOIN B.

л) Каждый ключ переменной отношения A является суперключом для полуразности A SEMIMINUS B.

Однако многие из приведенных здесь операторов в некоторых ситуациях могут быть немного усовершенствованы, например, как описано ниже.

- Комбинация {S#,P#, J#} — не единственный суперключ для сокращения SPJ WHERE S# = S# (' S1'), потому что комбинация {P#, J#} также является суперключом.
- Если переменная отношения A имеет заголовок {X, Y, Z} и единственный потенциальный ключ x, а также удовлетворяет функциональной зависимости Y → Z (см. главу 11), то Y является суперключом для проекции переменной отношения A по Y и Z.
- Если и переменная отношения A, и переменная отношения B являются сокращениями переменной отношения C, то каждый ключ переменной отношения C является суперключом для объединения A UNION B.

Вся эта тема, касающаяся логических зависимостей между ключами, подробно рассматривается в[11.7].

9.14. Прежде всего, следует отметить, что в языке SQL ограничения типа как таковые вообще не поддерживаются. Поэтому часть а) этого упражнения невозможно решить непосредственно. Но допустимые названия городов можно хранить в одной из базовых таблиц и использовать внешние ключи для обеспечения того, чтобы никакая другая таблица не могла включать название города, которое не является одним из допустимых[6]. Аналогичные замечания относятся и к части б). Здесь подробные сведения по этой теме не приведены.

В)
```
CREATE ASSERTION SQL_C CHECK
        ( P.COLOR <> COLOR { 'Red') OR
          P.WEIGHT < WEIGHT ( 50.0 ) ) ;
```

В этом и во всех остальных ответах было решено использовать "утверждения" (конструкция ASSERTION), а не "ограничения проверки базовой таблицы".

Г)
```
CREATE ASSERTION SQL_D CHECK
        ( NOT EXISTS ( SELECT * FROM J AS JX WHERE EXISTS
                ( SELECT * FROM J AS JY WHERE ( JX.J# <>
                JY.J# AND
                                JX.CITY = JY.CITY ) ) ) ) ;
```

Д)
```
CREATE ASSERTION SQL_E CHECK ( (
    SELECT COUNT(*) FROM S
        WHERE  S.CITY = 'Athens' ) < 1 ) ;
```

е)
```
CREATE ASSERTION SQL_F CHECK
  ( NOT EXISTS ( SELECT *
                        FROM   SPJ AS SPJX WHERE  SPJX.QTY
                        > 2 * ( SELECT AVG ( SPJY.QTY ) FROM
                        SPJ AS SPJY ) ) ) ;
```

Ж)
```
CREATE ASSERTION SQL_G CHECK
        ( NOT EXISTS ( SELECT * FROM S SX WHERE
                EXISTS ( SELECT * FROM S SY WHERE
                                SX.STATUS = ( SELECT MAX ( S.STATUS )
                                        FROM S ) AND SY. STATUS = (
                        SELECT MIN ( S. STATUS )
                                        FROM S ) AND
                                SX.STATUS  <>  SY.STATUS  AND
                        SX.CITY  =  SY.CITY  )  )  )  ;
```

[6] Могут также использоваться "домены" в стиле языка SQL [4.20].

3) CREATE ASSERTION SQL_H CHECK
(NOT EXISTS (SELECT * FROM J WHERE NOT
EXISTS (SELECT * FROM S WHERE
S.CITY = J.CITY))) ;

И) CREATE ASSERTION SQL_I CHECK
(NOT EXISTS (SELECT * FROM J WHERE NOT EXISTS (SELECT
* FROM S WHERE S.CITY = J.CITY AND EXISTS (SELECT * FROM
SPJ
WHERE SPJ.S# = S.S# AND SPJ.J# = J.J#)))) ;

К) CREATE ASSERTION SQL_J CHECK
(NOT EXISTS (SELECT * FROM P) OR EXISTS (
SELECT * FROM P
WHERE P.COLOR = COLOR ('Red'))) ;

Л) CREATE ASSERTION SQL_K CHECK
((SELECT AVG (S. STATUS) FROM S) > 19) ;

Если таблица поставщиков пуста, то оператор AVG языка SQL возвращает (ошибочно!) неопределенное значение, результатом сравнения становится unknown (неизвестное значение) и ограничение больше не рассматривается как нарушенное. Дополнительные пояснения приведены в главе 19.

М) CREATE ASSERTION SQL_L CHECK
(NOT EXISTS (SELECT * FROM S
WHERE S.CITY = 'London' AND NOT EXISTS (SELECT *
FROM SPJ WHERE SPJ.S* = S.S# AND SPJ.P# = P# ('P2')))
) ;

Н) CREATE ASSERTION SQL_M CHECK
(NOT EXISTS (SELECT * FROM P
WHERE P.COLOR = COLOR ('Red')) OR EXISTS (
SELECT * FROM P
WHERE P. COLOR = COLOR ('Red')
AND P.WEIGHT < WEIGHT (50.0))) ;

О) CREATE ASSERTION SQL_N CHECK
((SELECT COUNT (DISTINCT P#) FROM SPJ WHERE EXISTS (
SELECT ♦ FROM S WHERE (S.S# = SPJ.S# AND
S.CITY = 'London'))) > (SELECT COUNT (DISTINCT P#) FROM
SPJ WHERE EXISTS (SELECT * FROM S WHERE (S.S# = SPJ.S#
AND
S.CITY = 'Paris')))) ;

П) CREATE ASSERTION SQL_O CHECK
((SELECT SUM (SPJ.QTY) FROM SPJ WHERE
(SELECT S.CITY FROM S
WHERE S.S# = SPJ.S#) = 'London') > (SELECT
SUM (SPJ.QTY) FROM SPJ WHERE (SELECT S.CITY FROM S
WHERE S.S# = SPJ.S*) = 'Paris')) ; Обратите внимание на использование в данном примере двух скалярных подзапросов.

р) Это задание не может быть выполнено непосредственно (язык SQL не поддерживает переходных ограничений), но может быть разработан триггер. Более подробный ответ на это упражнение не предусмотрен.

с) То же, что и в варианте р).

ОТВЕТЫ К ГЛАВЕ 10

10.2. VAR NON_COLOCATED VIEW
(S { S# } JOIN P { P# }) MINUS (S JOIN P) { S#, P# } ;

В данном выражении при желании первый оператор JOIN можно заменить оператором TIMES.

10.7. Приведенные ниже решения перенумерованы как 10.7.п, где п— номер исходного примера в разделе 10.1. Что касается переменных области значений, то применяются обычные соглашения.

10.7.1.
```
VAR REDPART VIEW
    { PX.P#, PX.PNAME, PX.WEIGHT AS WT, PX.CITY }    WHERE
PX.COLOR = COLOR ( 'Red' ) ;
```

10.7.2.
```
VAR PQ VIEW
    { PX.P#,
      SUM ( SPX WHERE SPX.P# = PX.P#, QTY ) AS TOTQTY } ;
```

10.7.3.
```
VAR CITY_PAIR VIEW
    { SX.CITY AS SCITY, PX.CITY AS PCITY }
      WHERE EXISTS SPX ( SPX.S# = SX.S# AND
      SPX.P# = PX.P# ) ;
```

10.7.4.
```
VAR HEAVY_REDPART VIEW
      RPX WHERE RPX.WT > WEIGHT ( 12.0 ) ;
```

Здесь RPX — переменная области значений, которая пробегает по переменной отношения REDPART.

10.8. Поскольку результатом применения конструкции ORDER BY не является отношение.

10.11. Ответ на этот вопрос является положительным, но необходимо отметить следующее. Предположим, что переменная отношения поставщиков S заменена двумя сокращениями, скажем, SA и SB, где SA — поставщики, находящиеся в Лондоне, а SB — поставщики, не находящиеся в Лондоне. Теперь можно определить объединение SA и SB как представление с именем S. Если после этого будет предпринята попытка обновить (с помощью этого представления) значение города поставщика из Лондона, указав в операторе UPDATE город, отличный от Лондона, или обновить значение города поставщика не из Лондона, указав в операторе UPDATE город Лондон, то данная реализация должна преобразовать этот оператор UPDATE в оператор DELETE, применяемый к одному из этих двух сокращений, и в оператор INSERT, применяемый, соответственно, к другому сокращению. Итак, правила, приведенные в разделе 10.4, позволяют успешно справиться с этим заданием; в действительности, операция UPDATE (намеренно) определена как операция DELETE, за которой следует операция INSERT. Тем не менее, по умолчанию применяется такое предположение, что в данной реализации фактически должен использоваться оператор UPDATE в целях повышения эффективности. Настоящий пример показывает, что иногда преобразование одного оператора UPDATE в другой оператор UPDATE является недопустимым; фактически определение тех случаев, в которых такое преобразование возможно, следует рассматривать как одно из направлений оптимизации.

10.15. Автор предлагает ознакомиться со следующими комментариями. Прежде всего, сам процесс замены состоит из нескольких этапов, которые могут быть кратко представлены, как показано ниже. (В дальнейшем изложении эта последовательность операций будет усовершенствована.)

```
/* Определить новую переменную отношения */

VAR SNC BASE RELATION
        { S# S#, SNAME NAME, CITY CHAR } KEY {
          S# } ;

VAR ST BASE RELATION
      { S# S#, STATUS INTEGER }
```

```
    KEY { S# };

/* Скопировать данные в новую переменную отношения */ INSERT SNC S {

S#, SNAME, CITY } ; INSERT ST S { S#, STATUS } ; /* Удалить старую

переменную отношения */

DROP VAR S ;
```

Теперь может быть создано требуемое представление.

```
VAR S VIEW
        SNC JOIN ST ;
```

В данный момент следует отметить, что каждый из двух атрибутов S# (в переменных отношения SNC и ST) представляет собой внешний ключ, который ссылается на другой атрибут. В действительности, существует строгая взаимно однозначная связь между переменными отношения SNC и ST и поэтому в данном случае приходится сталкиваться с многочисленными сложностями[7] применения "взаимно однозначных" связей, которые были подробно описаны автором данной книги в другой работе (см. [14.8]).

Следует также отметить, что в переменной отношения SP необходимо выполнить определенные действия с внешним ключом, который ссылается на старую базовую переменную отношения S. Безусловно, было бы лучше, если бы этот внешний ключ теперь можно было трактовать как ссылающийся вместо этого на представление S;[8] если это невозможно (как в действительности обычно имеет место в современных программных продуктах), то было бы лучше ввести в базу данных третью проекцию базовой переменной отношения S.

```
VAR SS BASE RELATION
        { S# S# } KEY { S# } ;

INSERT SS S { S# } ;
```

(В действительности, этот проект, так или иначе, рекомендован в [9.11], хотя и по другим причинам.) Теперь откорректируем определение представления S следующим образом.

```
VAR S VIEW
        SS JOIN SNC JOIN ST ;*
```

Добавим также следующую спецификацию внешнего ключа к определениям переменных отношения SNC И ST.

```
FOREIGN KEY { S# } REFERENCES SS
                        ON DELETE CASCADE
                        ON UPDATE CASCADE
```

Наконец, необходимо изменить спецификацию для внешнего ключа {S#} в переменной отношения SP, чтобы он ссылался на переменную отношения SS, а не S.

10.16. Что касается части а) этого упражнения, то ниже приведен один пример выборки представления, который ко времени написания данной книги, безусловно, должен был оканчиваться неудачей при его выполнении в некоторых программных продуктах. Рассмотрим следующее определение представления SQL.

---

[7] Некоторых из этих сложностей можно было бы избежать при наличии в системе поддержки множественного присваивания, но эта возможность в [14.8] не обсуждалась.

[8] В действительности, весомым доводом в пользу того, чтобы ограничения, вообще говоря, можно было определять не только для базовых переменных отношения, но и для представлений, является логическая независимость от данных.

```
CREATE VIEW PQ AS
      SELECT SP.P#, SUM ( SP.QTY ) AS TOTQTY    FROM   SP
  GROUP BY SP.P#;
```

Рассмотрим также следующую попытку выполнения запроса.

```
SELECT AVG ( PQ.TOTQTY ) AS PT FROM
PQ ;
```

Если следовать простой процедуре подстановки, описанной в основной части главы 10 (т.е. если попытаться заменить ссылки на имя представления выражением, в котором определено это представление), то будет получен примерно следующий результат.

```
SELECT AVG ( SUM ( SP.QTY ) ) AS PT
FROM   SP
GROUP  BY SP.P# ;
```

А этот оператор SELECT не является допустимым, поскольку (как было отмечено в комментарии, который следует за формулировкой упр. 8.6.7 в главе 8) в языке SQL не разрешено использовать в такой форме вложенные конструкции, состоящие из агрегирующих операторов.

Ниже приведен еще один пример запроса, применяемого к тому же представлению PQ, который также оканчивается неудачей при выполнении в некоторых программных продуктах, в основном по той же причине.

```
SELECT PQ.P#
FROM   PQ
WHERE  PQ.TOTQTY > 500 ;
```

Кстати, следует отметить, что именно из-за той проблемы, которая была проиллюстрирована в этих примерах, в некоторых программных продуктах (в данном случае речь идет о СУБД DB2 компании IBM) иногда осуществляется физическая материализация этого представления (вместо использования более обычной процедуры подстановки), а затем запрос применяется к этой материализованной версии. Безусловно, что такой метод всегда позволяет добиться успеха, но обладает тем недостатком, что с ним связано значительное снижение производительности. Более того, что касается, в частности, СУБД DB2, все равно остается фактом, что некоторые операции выборки на некоторых представлениях не завершаются успешно; это означает, во-первых, что в СУБД DB2 не всегда используется материализация, даже если подстановка является неприменимой, а во-вторых, что невозможно точно определить, какие варианты выборки из представлений в этой СУБД являются осуществимыми и какие нет.

10.20. Реляционная модель состоит из пяти компонентов, описанных ниже.

1. Неограниченная коллекция скалярных типов (включая, в частности, логический тип, или ис тинностное значение).

Комментарий. Вообще говоря, эти скалярные типы могут быть определяемыми системой или пользователем, поэтому должны быть предусмотрены средства, с помощью которых пользователи могли бы определять свои собственные типы (такое требование частично обусловлено тем, что, как сказано выше, коллекция скалярных типов является "неограниченной"). В соответствии с этим, пользователям должны быть также предоставлены средства для определения их собственных операторов, поскольку типы без операторов являются бесполезными. Автор настаивает на том, что единственным встроенным (т.е. определяемым системой) типом должен быть тип BOOLEAN, но реальные системы должны, безусловно, поддерживать целые числа, строки и т.д.

2. Генератор типа отношения и намеченная интерпретация для отношений с типами, созданны ми с помощью этого генератора.

Комментарий. Генератор типа отношения позволяет пользователям определять свои собственные типы отношений (в языке Tutorial D определение данного конкретного типа отношения обычно совмещается с определением переменной отношения этого типа; по причинам, подробно описанным в [3.3], отдельный оператор "определения типа отношения" не предусмотрен).

Поэтому намеченная интерпретация для данного конкретного типа отношения может рассматриваться с точки зрения предиката отношения.

3. Средства определения переменных отношения для указанных выше сгенерированных типов отношения.

   Комментарий. Это требование является обязательным! Следует отметить, что переменные отношения являются единственными переменными, разрешенными для использования в реляционной базе данных (по существу, это следует из информационного принципа).

4. Операция реляционного присваивания, позволяющая присваивать значения отношений таким переменным отношения.

   Комментарий. Переменные являются обновляемыми по определению (именно это и подразумевается под словом "переменная"), поэтому переменная любого типа может стать объектом действия операции присваивания (именно так осуществляется обновление), и переменные отношения не являются исключением. Безусловно, такие сокращения, как INSERT, UPDATE и DELETE, являются допустимыми и действительно полезными, но, строго говоря, они были и остаются просто сокращениями.

5. Неограниченная коллекция универсальных реляционных операторов для получения значений отношения из других значений отношения.

   Комментарий. Из этих операторов состоит реляционная алгебра и поэтому они являются встроенными (хотя и не существует каких-либо непреодолимых причин, из-за которых пользователям нельзя было бы дать возможность определять дополнительные операторы). Следует отметить, что эти операторы являются универсальными. Это означает, что они, выражаясь неформально, будут применяться ко всем возможным отношениям.

ОТВЕТЫ К ГЛАВЕ 11

11.3. Правило рефлексивности гласит, что если множество функциональных зависимостей B является подмножеством множества функциональных зависимостей A, то A → B.

Доказательство. Предположим, что рассматриваемой переменной отношения является R, а t1 и t2 — любые два кортежа R, которые соответствуют множеству функциональных зависимостей A. В таком случае, безусловно, t1 и t2 соответствуют множеству функциональных зависимостей B. Следовательно, A → B.

Правило дополнения гласит, что если A → B, то AC → BC.

Доказательство. Снова предположим, что рассматриваемой переменной отношения является R, а tl и t2 — любые два кортежа R, которые соответствуют множеству функциональных зависимостей AC. В таком случае, безусловно, tl и t2 соответствуют множеству функциональных зависимостей C. Они также соответствуют множеству функциональных зависимостей A, а поэтому и множеству функциональных зависимостей B, поскольку A → B. Следовательно, они соответствуют множеству функциональных зависимостей BC. Это означает, что AC → BC.

Правило транзитивности гласит, что если A → B и B → C, то A → C.

Доказательство. Снова предположим, что рассматриваемой переменной отношения является R, а tl и t2 — любые два кортежа R, которые соответствуют множеству функциональных зависимостей A. В таком случае, безусловно, tl и t2 соответствуют множеству функциональных зависимостей B, поскольку A → B. Следовательно, они также соответствуют множеству функциональных зависимостей C, поскольку B → C. Это означает, что A → C.

11.8. $\{A,C\}^+ = \{A, B, C, D, E\}$. Ответна вторую часть этого вопроса является положительным.

11.11. Они эквивалентны. Пронумеруем функциональные зависимости первого множества следующим образом.

1. A → B '
2. AB → C
3. D → AC
4. D → E

После этого функциональную зависимость 3 можно заменить следующим образом

5. D → A И D → C

Затем отметим, что из функциональных зависимостей 1 и 2, вместе взятых, следует, что ФЗ 2 можно заменить следующий функциональной зависимостью.

6. A → C

Но теперь имеют место ФЗ D → A и A→ C, поэтому ФЗ D → C можно вывести как следствие (по правилу транзитивности). Это означает, что ее можно удалить, получив следующую функциональную зависимость.

7. D →A

Таким образом, первое множество функциональных зависимостей эквивалентно следующему неприводимому множеству.

A → B A → C D → A D → E Очевидно, что второе заданное множество функциональных зависимостей

A → BC
D → AE

также эквивалентно этому неприводимому множеству. Таким образом, два заданных множества эквивалентны.

11.14. Сокращенно обозначив NAME, STREET, CITY, STATE и ZIP[9], соответственно, как N, R, C, T и Z, получим следующий результат.

N → RCT RCT
→ Z Z → CT

Очевидно, что эквивалентным неприводимым множеством является следующее множество.

N → R N → C N v T RCT → Z Z → C Z
→ T Здесь N — единственный потенциальный ключ.

[9] Кстати, известно ли читателю, что слово ZIP, которое мы привыкли рассматривать как обозначение почтового кода, — это сокращенное обозначение "Программы усовершенствования зонирования" (Zoning Improvement Program), принятой в США?

ОТВЕТЫ К ГЛАВЕ 12

12.1. Теорема Хита (Heath) гласит, что если переменная отношения R{A,B,C} удовлетворяет функциональной зависимости A → B (где A, B и C — множества атрибутов), то R равна соединению своих проекций: R1 по {A, B) и R2 по {A, C}. В приведенном ниже доказательстве этой теоремы принято обычно применяемое в данной книге неформальное сокращенное обозначение для кортежей.

Прежде всего, необходимо доказать, что ни один кортеж переменной отношения R не будет потерян после получения этих проекций, а затем повторного соединения этих проекций друг с другом. Допустим, что (a,b,c) ∈ R, в таком случае (a,b) ∈ R1 и (a, c) ∈ R2 и поэтому является истинным выражение (a,b,c) ∈ R1 JOIN R2.

Затем необходимо доказать, что каждый кортеж полученного соединения действительно является кортежем переменной отношения R (т.е. доказать, что операция соединения не вырабатывает каких-либо "фиктивных" кортежей). Допустим, что (a, b, c) ∈ R1 JOIN R2. Для того чтобы кортеж (a,b,c) появился в соединении, необходимо, чтобы существовали кортежи (a,b) ∈ R1 и (a, c) ∈ R2. Поэтому, для того чтобы можно было сформировать кортеж (a, c) ∈ R2, должен существовать кортеж (a, b' , c) ∈ R для некоторого b'. Это означает, что должен существовать кортеж (a, b') ∈ R1. Итак, известно, что существуют два кортежа, (a, b) ∈ R1 и (a,b') ∈ R1; следовательно, должна иметь место эквивалентность b = b', поскольку A → B. Таким, образом, (a,b,c) ∈ R.

Предположим, что принято решение сформулировать теорему, обратную теореме Хита, которая гласит, что если переменная отношения R{A,B,C} равна соединению своих проекций по {A, в} и по {A, C}, то R удовлетворяет функциональной зависимости A —> в. Но это утверждение является ложным. Например, на рис. 13.2 главе 13 показано отношение, которое, безусловно, равно соединению двух своих проекций и, тем не менее, вообще не удовлетворяет ни одной (нетривиальной) функциональной зависимости.

12.3. На рис. Д.6 показаны наиболее важные функциональные зависимости — и те, которые следуют из формулировки данного упражнения и те, которые соответствуют обоснованным семантическим допущениям (явно сформулированным ниже). Предполагается, что имена атрибутов не требуют дополнительных пояснений.

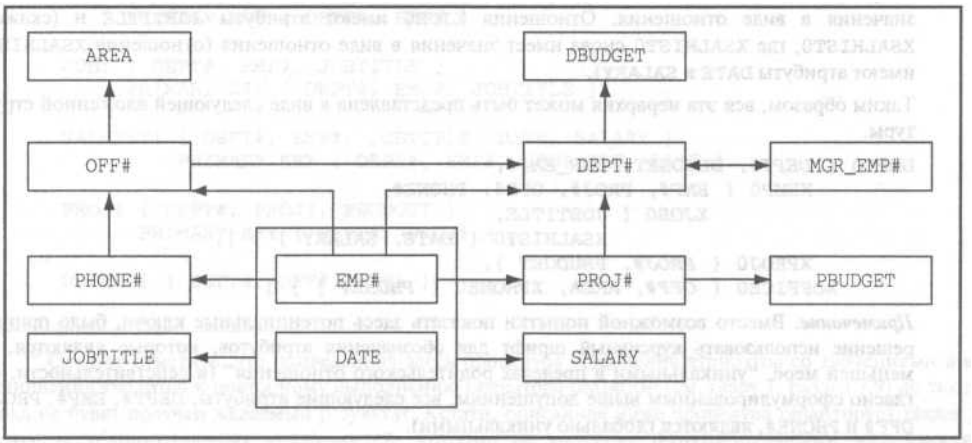


Рис. Д.6. Наиболее важные функциональные зависимости

Применяемые семантические допущения перечислены ниже.

- Ни один из служащих не является менеджером больше чем одного отдела одновременно.
- Ни один из служащих не работает больше чем в одном отделе одновременно.
- Ни один из служащих не работает больше чем над одним проектом одновременно.
- Ни один из служащих не имеет больше одного офиса одновременно.
- Ни один из служащих не имеет больше одного телефона одновременно.
- Ни один из служащих не имеет больше одного задания одновременно.
- Ни один из проектов не поручается для выполнения больше чем одному отделу одновременно.
- Ни один из офисов не выделяется больше чем одному отделу одновременно.
- Все номера отделов, табельные номера, номера проектов, номера офисов и номера телефонов являются "глобально" уникальными.

Шаг 0. Определить первоначальную структуру переменной отношения

Прежде всего, следует отметить, что первоначальная иерархическая структура может рассматриваться как переменная отношения DEPT0 в первой нормальной форме (1НФ) со значениями в виде следующего отношения.

```
DEPT0 { DEPT#, DBUDGET, MGR_EMP#, XEMP0, XPROJ0, XQFFICEQ } KEY {
    DEPT# } KEY { MGR_EMP# }
```

Атрибуты DEPT#, DBUDGET и MGR_EMP# не требуют дополнительных пояснений, а атрибуты XEMPO, XPROJ0 и XOFFICE0 имеют значения в виде отношения, и ниже приведены некоторые пояснения к этим атрибутам.

- Значение XPROJ0 в каждом конкретном кортеже DEPT0 представляет собой отношение с атрибутами PROJ# и PBUDGET.
- Аналогичным образом, значение XOFFICE0 в каждом конкретном кортеже DEPT0 представляет собой отношение с атрибутами OFF#, AREA и (скажем) XPHONE0, где XPHONE0, в свою очередь, имеет значения в виде отношения. Отношения XPHONE0 имеют только один атрибут, PHONE*.
- Наконец, значение XEMPO в каждом конкретном кортеже DEPT0 представляет собой отношение с атрибутами EMP#, PROJ#, OFF#, PHONEt и (скажем) XJOB0, где XJOB0, в свою очередь, имеет значения в виде отношения. Отношения XJOB0 имеют атрибуты JOBTITLE и (скажем) XSALHIST0, где XSALHIST0 снова имеет значения в виде отношения (отношения XSALHIST0 имеют атрибуты DATE и SALARY).

  Таким образом, вся эта иерархия может быть представлена в виде следующей вложенной структуры.

```
DEPT0  {  DEPTi,  DBUDGET,  MGR EMP#,
            XEMPO  {  EMP#,  PROJ#,  OFF#,  PHONE#,
                        XJOB0  {  JOBTITLE,
                                  XSALHISTO  {  DATE,  SALARY  )  }  },
        XPROJ0  {  PROJ#,  PBUDGET  }, XOFFICE0  {  OFF#,  AREA,
        XPHONE0  {  PHONE# }  }  }
```

Примечание. Вместо возможной попытки показать здесь потенциальные ключи, было принято решение использовать курсивный шрифт для обозначения атрибутов, которые являются, по меньшей мере, "уникальными в пределах родительского отношения" (в действительности, согласно сформулированным выше допущениям, все следующие атрибуты, DEPT#, EMP#, PROJ#, OFF# и PHONE#, являются глобально уникальными).

Шаг 1. Устранение атрибутов со значениями в виде отношения

Теперь для упрощения предположим, что выдвинуто требование, чтобы для каждой переменной отношения был специально предусмотрен первичный ключ. Это означает, что на каком-то основании (и это основание здесь не имеет значения) один из потенциальных ключей всегда обозначается как первичный. В частности, в случае переменной отношения DEPT0 выберем в качестве первичного ключа множество атрибутов {DEPT#} (и поэтому множество атрибутов {MGR_EMP#} станет альтернативным ключом).

Теперь перейдем к решению задачи исключения всех атрибутов со значениями в виде отношения из переменной отношения DEPT0, как описано ниже, поскольку, как отмечалось в разделе 12.6, такие атрибуты обычно нежелательны[10].

- Для каждого атрибута со значениями в виде отношения в переменной отношения DEPT0 (т.е. для атрибутов XEMPO, XPROJ0 и XOFFICE0) сформировать новую переменную отношения с атрибутами, состоящими из атрибутов соответствующего типа отношения, наряду с первичным ключом DEPT0. Первичный ключ каждой такой переменной отношения представляет собой комбинацию тех атрибутов, которые перед этим обеспечивали "уникальность в пределах родительского отношения", наряду с первичным ключом DEPT0. (Но следует отметить, что многие из этих "первичных ключей" будут содержать атрибуты, которые являются избыточными с точки зрения уникальной идентификации, и поэтому в дальнейшем будут удалены в общей процедуре сокращения.) Удалить атрибуты XEMPO, XPROJ0 и XOFFICE0 из переменной отношения DEPT0.
- Если какая-либо переменная отношения R все еще включает какие-либо атрибуты со значениями в виде отношения, выполнить аналогичную последовательность операций с переменной отношения R.

На этом этапе будет получена следующая коллекция переменных отношения, в которых (как было указано) удалены все атрибуты со значениями в виде отношения. Но необходимо отметить следующее: хотя результирующие переменные отношения (безусловно) обязательно находятся в первой нормальной форме, они не должны обязательно находиться в какой-либо более высокой нормальной форме.

```
DEPT1 { DEPT#, DBUDGET, MGR_EMP# }
        PRIMARY KEY { DEPT# } ALTERNATE
        KEY { MGR_EMP# }

EMP1 { DEPT#, EMP#, PROJ#, OFF#, PHONE# ) PRIMARY
       KEY { DEPT#, EMP# }

J0B1 { DEPT#, EMP#, JOBTITLE }
       PRIMARY KEY { DEPT#, EMP#, JOBTITLE }

SALHIST1 { DEPT#, EMP#, JOBTITLE, DATE, SALARY }
           PRIMARY KEY { DEPT#, EMP#, JOBTITLE, DATE }

PR0J1 { DEPT#, PROJ#, PBUDGET }
        PRIMARY KEY { DEPT#, PROJ# }

OFFICE1 { DEPT#, OFF#, AREA }
```

[10] Следует отметить, что приведенная здесь процедура устранения атрибутов со значениями в виде отношения сводится к повторному выполнению оператора UNGROUP (см. главу 7, раздел 7.9) до тех пор, пока не будет получен желаемый результат. Кстати, описанная ниже процедура гарантирует также удаление всех многозначных зависимостей, которые не являются функциональными зависимостями. Вследствие этого переменные отношения, которые будут получены в конечном итоге, будут фактически находиться в четвертой нормальной форме, а не просто в нормальной форме Бойса-Кодда, НФБК (см. главу 13).

PRIMARY KEY { DEPT#, OFF#)

PH0NE1 { DEPT#, OFF#, PHONE# } PRIMARY KEY { DEPT#, OFF#, PHONE# }

Шаг 2. Приведение ко второй нормальной форме

Теперь преобразуем переменные отношения, полученные в шаге 1, в эквивалентную коллекцию переменных отношения во второй нормальной форме (2НФ) путем устранения всех функциональных зависимостей, которые не являются неприводимыми. Рассмотрим эти переменные отношения одну за другой.

- DEPT1. Эта переменная отношения уже находится в 2НФ.
- EMP1. Прежде всего, следует отметить, что атрибут DEPT# фактически является избыточным как компонент первичного ключа для этой переменной отношения. В качестве первичного ключа можно взять лишь множество атрибутов {EMP#} и в таком случае данная переменная отношения в своем непосредственном виде станет принадлежащей ко второй нормальной форме.
- JOB1. Снова отметим, что атрибут DEPT# не является обязательным компонентом первичного ключа. Поскольку атрибут DEPT# является функционально зависимым от EMP#, то мы имеем дело с неключевым атрибутом (DEPT#), который не является неприводимо зависимым от первичного ключа (комбинации атрибутов {EMP#, JOBTITLE}), и поэтому переменная отношения J0B1 не находится в 2НФ. Ее можно заменить переменными отношения

  J0B2A { EMP#, JOBTITLE }
  PRIMARY KEY { EMP#, JOBTITLE } И

  JOB2B { EMP#, DEPT# }
  PRIMARY KEY { EMP# }

Но J0B2A — проекция переменной отношения SALHIST2 (см. ниже), а J0B2B — проекция переменной отношения EMP1 (переименованной ниже в EMP2), поэтому обе эти переменные отношения могут быть отброшены.

- SALHIST1. Здесь, как и в переменной отношения JOB1, с помощью операции проекции можно полностью исключить атрибут DEPT#. Кроме того, атрибут JOBTITLE не является обязательным компонентом первичного ключа; в качестве первичного ключа можно взять комбинацию {EMP#, DATE}, чтобы получить следующую переменную отношения в 2НФ.

  SALHIST2 { EMP#, DATE, JOBTITLE, SALARY }
  PRIMARY KEY { EMP#, DATE }

- PR0J1. Как и в переменной отношения EMP1, атрибут DEPT# может рассматриваться как неключевой атрибут; в таком случае эта переменная отношения в своем непосредственном виде может рассматриваться как находящаяся в 2НФ.
- OFFICE1. Применимы аналогичные замечания.
- PHONE 1. Атрибут DEPT# может быть полностью исключен с помощью операции проекции, поскольку переменная отношения (DEPT#,OFF#) является проекцией переменной отношения OFFICE1 (переименованной ниже в OFFICE2). Кроме того, атрибут OFF# является функционально зависимым от атрибута PHONE*, поэтому для получения переменной отношения в 2НФ в качестве первичного ключа можно взять только множество атрибутов {PHONE#}.

  PHONE2 { PHONE#, OFF# }
  PRIMARY KEY { PHONE» }

Следует отметить, что эта переменная отношения не обязательно является проекцией переменной отношения EMP2 (телефоны или офисы могут существовать, не будучи назначенными конкретным служащим), поэтому ее нельзя отбрасывать. Таким образом, коллекция переменных отношения в 2НФ принимает следующий вид.

```
DEPT2 { DEPT#, DBUDGET, MGR_EMP# }
        PRIMARY KEY { DEPT# }
        ALTERNATE KEY { MGR_EMP# }

EMP2 { EMP#, DEPT#, PROJ#, OFF#, PHONE# } PRIMARY
      KEY { EMP# }

SALHIST2 { EMP#, DATE, JOBTITLE, SALARY }
           PRIMARY KEY { EMP#, DATE }

PROJ2 { PROJ#, DEPT#, PBUDGET }
        PRIMARY KEY { PROJ# }

OFFICE2 { OFF#, DEPT#, AREA }
          PRIMARY KEY { OFF# }

PHONE2 { PHONE#, OFF# }
         PRIMARY KEY { PHONE# }
```

Шаг 3. Приведение к третьей нормальной форме

Теперь приведем эти переменные отношения во второй нормальной форме в эквивалентное множество в третьей нормальной форме (3НФ), устранив транзитивные функциональные зависимости. Единственной переменной отношения 2НФ, которая еще не находится в 3НФ, является переменная отношения EMP2, в которой и атрибут OFF#, и атрибут DEPT# являются транзитивно зависимыми от первичного ключа {EMP#} — зависимость OFF# проявляется через PHONE#, а DEPT# — через PROJ#, а также через OFF# (и поэтому через PHONE#). Ниже приведены переменные отношения 3НФ (проекции), соответствующие EMP2.

```
EMP3 { EMP#, PROJ#, PHONE# }
       PRIMARY KEY { EMP# }

X { PHONE#, OFF#
   PRIMARY KEY { PHONE# }

Y { PROJ#, DEPT# }     ; PRIMARY
   KEY { PROJ# }

Z { OFF#, DEPT# )
    PRIMARY KEY { OFF# }
```

Но X — переменная отношения PHONE2, Y — проекция PROJ2, а Z — проекция OFFICE2. Поэтому коллекция переменных отношения 3НФ просто принимает следующий вид.

```
DEPT3 { DEPT#, DBUDGET, MGR_EMP# }
        PRIMARY KEY { DEPT# }
        ALTERNATE KEY { MGR_EMP# }

EMP3 { EMP#, PROJ#, PHONE# }
       PRIMARY KEY { EMP# }

SALHIST3 { EMP#, DATE, JOBTITLE, SALARY } PRIMARY
           KEY { EMP#, DATE }

PROJ3 { PROJ#, DEPT#, PBUDGET }
        PRIMARY KEY { PROJ# }

OFFICE3 { OFF#, DEPT#, AREA }
          PRIMARY KEY { OFF# }
```

```
PH0NE3 { PHONE#, OFF# }
        PRIMARY KEY { PHONE# }
```

Наконец, можно легко обнаружить, что каждая из этих переменных отношения ЗНФ фактически находится в НФБК.

Следует отметить, что при некоторых (обоснованных) дополнительных семантических ограничениях эта коллекция переменных отношения НФБК является строго избыточной (см. [6.1]), поскольку проекция переменной отношения PROJ3 по {PROJ#, DEPT#} всегда равна проекции соединения переменных отношения EMP3, PHONE3 И OFFICE3.

Наконец, следует отметить, что переменные отношения НФБК можно "выявить" на диаграмме функциональных зависимостей, которая показана на рис. Д.6 (объясните, как).

Ответ. Неформально выражаясь, для каждого прямоугольника, показанного на этом рисунке, должна быть только одна такая переменная отношения, из которой исходит стрелка; эта переменная отношения должна включать атрибуты из данного исходного прямоугольника, применяемые в качестве потенциального ключа, наряду с отдельно атрибутом для каждого прямоугольника, на который указывает стрелка, исходящая из этого прямоугольника (и не включать каких-либо иных атрибутов). Безусловно, это неформальное утверждение требует определенного уточнения для того, чтобы его можно было распространить на такие переменные отношения, как DEPT3, которые имеют два или несколько потенциальных ключей.

Примечание. Автор не утверждает, что всегда возможно "выявить" декомпозицию НФБК, а лишь указывает, что в тех случаях, которые обычно возникают на практике, часто существует такая возможность.

Вернемся к примеру с базой данных компании. В качестве дополнительного упражнения (которое, вообще говоря, не относится к задаче нормализации, но непосредственно касается проблемы проектирования базы данных в целом) попытайтесь дополнить описанный выше проект, чтобы включить в него также все необходимые спецификации внешнего ключа.

Ниже приведен ответ к этому заданию.

```
    DEPT3 { DEPT#, DBUDGET, MGR_EMP# }
  PRIMARY KEY { DEPT# }
            ALTERNATE KEY { MGR_EMP# }      FOREIGN KEY { RENAME MGR_EMP#
    AS EMP# } REFERENCES EMP3

    EMP3 { EMP#, PROJ#, PHONE# }
          PRIMARY KEY { EMP# }
          FOREIGN KEY { PROJ# } REFERENCES PROJ3
          FOREIGN KEY { PHONE# } REFERENCES PHONE3

    SALHIST3 { EMP#, DATE, JOBTITLE, SALARY } PRIMARY
              KEY { EMP#, DATE } FOREIGN KEY { EMP# }
              REFERENCES EMP3

    PR0J3 { PROJ#, DEPT#, PBUDGET )          PRIMARY KEY {
PROJ# }        FOREIGN KEY { DEPT# } REFERENCES DEPT3

    OFFICE3 { OFF#, DEPT#, AREA }
              PRIMARY KEY { OFF# }       FOREIGN KEY { DEPT# }
    REFERENCES DEPT3

    PHONE3   {  PHONE#,  OFF#  }
              PRIMARY KEY  {  PHONE#  } FOREIGN KEY  {
            OFF#  }  REFERENCES  OFFICE3
```

12.6. На рис. Д.7 показаны наиболее важные функциональные зависимости. Одной из возможных коллекций переменных отношения является приведенная ниже.

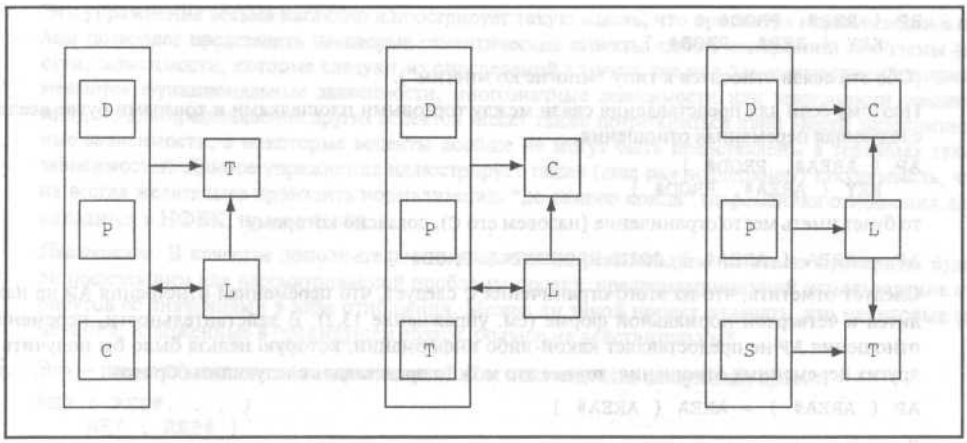


Рис. Д.7. Наиболее важные функциональные зависимости

```
SCHED { L, T, C, D,          P}
        KEY { L }
        KEY { T, D, P        }
        KEY {C, D, P         }

STUDY { S, L }
        KEY { S, L }
```

12.8. Это упражнение оказалось удивительно сложным, но для него подходит следующее решение. Допустим, что spqt — произвольный кортеж, присутствующий в переменной отношения SPQ, а s и pq, соответственно, — значения S# и PQ, присутствующие в кортеже spqt. Предположим также, что pqt — произвольный кортеж, присутствующий в значении pq, а p и q, соответственно, — значения P# и QTY, содержащиеся в этом кортеже pqt. В таком случае, во-первых, значение s не появляется ни в одном кортеже SPQ, кроме spqt, во-вторых, значение p не появляется ни в одном кортеже pq, кроме pqt, в-третьих, поставщик s поставляет детали p в количестве q, и, в-четвертых, поставщик s не поставляет каких-либо других деталей.

ОТВЕТЫ К ГЛАВЕ 13

13.3. Вначале введем следующие три переменные отношения с очевидной интерпретацией.

```
REP     { REP#, ... } KEY { REP# }

AREA    { AREA#, ... } KEY {
            AREA# }

PRODUCT { PROD#, ... } KEY {
            PROD# }
```

После этого можно представить связь между торговыми представителями и торговыми площадками с помощью следующей переменной отношения.

```
RA { REP#, AREA# }
     KEY { REP#, AREA# }
```

А связь между торговыми представителями и товарами с помощью следующей переменной отношения.

```
RP { REP#, PROD# }
    KEY { REP#, PROD# }
```

(Обе эти связи относятся к типу "многие ко многим").

Поэтому если для представления связи между торговыми площадками и товарами будет введена следующая переменная отношения.

```
AP { AREA#, PROD# }
    KEY { AREA#, PROD# }
```

то будет иметь место ограничение (назовем его С), согласно которому

```
AP = AREA { AREA# } JOIN PRODUCT { PROD# }
```

Следует отметить, что из этого ограничения С следует, что переменная отношения AP не находится в четвертой нормальной форме (см. упражнение 13.2). В действительности, переменная отношения AP не предоставляет какой-либо информации, которую нельзя было бы получить из других переменных отношения; точнее это можно представить следующим образом.

```
AP { AREA# } = AREA { AREA# }
```

И

```
AP { PROD# ) = PRODUCT { PROD# }
```

Но на время предположим, что переменная отношения AP все равно включена в проект. Никакие два торговых представителя не продают один и тот же товар на одной и той же площадке. Иными словами, если задана определенная комбинация {AREA*, PROD*}, то соответствует один и только один торговый представитель (REP#), поэтому можно ввести следующую переменную отношения.

```
APR  {  AREA#,  PROD#,  REP#  }
      KEY  {  AREA#,  PROD#  }
```

Здесь присутствует следующая функциональная зависимость (безусловно, для представления этой функциональной зависимости достаточно определить комбинацию атрибутов {AREA#, PROD#} в качестве ключа).

```
{  AREA#,  PROD#  }  →  REP#
```

Но теперь переменные отношения RA, RP и AP становятся избыточными, поскольку все они являются проекциями переменной отношения APR, следовательно, все они могут быть удалены. В таком случае вместо ограничения С потребуется ввести следующее ограничение c1.

```
APR { AREA#, PROD# } = AREA { AREA# } JOIN PRODUCT { PROD# }
```

Это ограничение необходимо сформулировать отдельно и явно (оно "не следует из определения ключей").

Кроме того, поскольку каждый торговый представитель продает все товары, относящиеся к этому торговому представителю, на всех торговых площадках, закрепленных за этим торговым представителем, необходимо ввести дополнительное ограничение С2, которое распространяется на переменную отношения APR (это — нетривиальная многозначная зависимость; переменная отношения APR не находится в четвертой нормальной форме).

```
REP#  →  AREA#  |  PROD#
```

И снова это ограничение должно быть сформулировано отдельно и явно.

Таким образом, окончательный проект состоит из переменных отношения REP, AREA, PRODUCT и APR, наряду с ограничениями С1 и С2, как показано ниже.

```
CONSTRAINT Cl APR { AREA#, PROD# } =
                    AREA { AREA# } JOIN PRODUCT { PROD# } ;

CONSTRAINT C2 APR =
                    APR { REP#, AREA# } JOIN APR { REP#, PROD# } ;
```

Это упражнение весьма наглядно иллюстрирует такую мысль, что процедура нормализации в целом позволяет представить некоторые семантические аспекты любой конкретной проблемы (по сути, зависимости, которые следуют из определений ключей, где под "зависимостями" подразумеваются функциональные зависимости, многозначные зависимости или зависимости соединения), но для представления других аспектов может также потребоваться явно ввести дополнительные зависимости, а некоторые аспекты вообще не могут быть представлены в терминах таких зависимостей. Данное упражнение иллюстрирует также (еще раз подчеркнем) такую мысль, что не всегда желательно проводить нормализацию "до самого конца" (переменная отношения APR находится в НФБК, но не в 4НФ).

Примечание. В качестве дополнительного упражнения рекомендуем читателю проверить, будет ли подходящим для рассматриваемой проблемы проект, предусматривающий использование атрибутов со значениями в виде отношения. Может ли такой проект означать, что некоторые замечания, изложенные в предыдущем абзаце, больше недействительны?

13.5. Это — пример "циклического ограничения". Для него подходит следующий проект.

```
REP { REP#, ... )
      KEY { REP# }

AREA { AREA#, ... } KEY {
        AREA# }

PRODUCT { PROD#, ... } KEY {
            PROD# }

RA { REP#, AREA# }
     KEY { REP#, AREA# }

AP { AREA#, PROD# }
     KEY { AREA#, PROD# }

PR { PROD#, REP# }
     KEY { PROD#, REP# )
```

Кроме того, пользователю необходимо сообщить, что соединение переменных отношения RA, AP и PR не приводит к возникновению какого-либо "дефекта соединения".

```
CONSTRAINT NO_TRAP
   ( RA JOIN AP JOIN PR )        { REP#, AREA# } = RA AND
   ( RA JOIN AP JOIN PR )        { AREA#, PROD# } = AP AND
   ( RA JOIN AP JOIN PR )        { PROD#, REP# } = PR ;
```

Примечание. Как и в случае упражнения 13.3, рекомендуем читателю проверить, подходит ли для решения рассматриваемой задачи такой проект, в котором используются атрибуты со значениями в виде отношения.

ОТВЕТЫ К ГЛАВЕ 14

14.4.

```
CONSTRAINT TUTD_14_4
      FORALL PX ( EXISTS SPX ( SPX.P# = PX.P# ) ) ;

CREATE ASSERTION SQL_14_4 CHECK (
          NOT EXISTS ( SELECT PX.* FROM P PX
                         WHERE NOT EXISTS ( SELECT SPX.*
                                              FROM SP SPX
                                              WHERE SPX.P# = PX.P# ) ) );
```

14.5.

а) Предположим, что служащие имеют иждивенцев, а иждивенцы имеют друзей, и рассмотрим связь между иждивенцами и друзьями.

б) Допустим, что поставки выражают связь между поставщиками и деталями и рассмотрим связь между поставками и проектами.

в) Рассмотрим "крупные поставки", где под крупной поставкой подразумевается поставка в количестве, скажем, больше 1000.

г) Допустим, что (только) крупные поставки могут осуществляться в контейнерах, и поэтому будем рассматривать контейнеры как соответствующие слабые сущности.

ОТВЕТЫ К ГЛАВЕ 15

15.4.

а) Вслед за аварийным остановом системы никогда не возникает необходимость выполнить накат.

б) Физический откат никогда не требуется и поэтому не нужны также записи журнала отката.

15.6. Это упражнение является типичным для широкого класса приложений и поэтому типичными также можно назвать приведенные ниже схематические решения. Вначале будет показано решение без использования средств CHAIN и WITH HOLD, которые были введены в спецификации SQL: 1999.

```
EXEC SQL DECLARE CP CURSOR FOR
            SELECT P.P#, P.PNAME, P.COLOR, P.WEIGHT, P.CITY
            FROM P
            WHERE P.P# > предыдущий_P#
            ORDER BY P#;

предыдущий_P# := '' ;
eof := FALSE ;
DO WHILE ( NOT eof ) ;
    EXEC SQL START TRANSACTION ; EXEC
    SQL OPEN CP ;
    DO count := 1 TO 10 ;
        EXEC SQL FETCH CP INTO :P#, ...; IF SQLSTATE =
        '02000' THEN DO ;
                EXEC SQL CLOSE CP ; EXEC
                SQL COMMIT ;
                    eof := TRUE ;
            END DO ;
    ELSE вывести на печать P#, ...; END IF ;
    END DO ;
EXEC SQL DELETE FROM P WHERE P.P# := P# ;
EXEC SQL CLOSE CP ; EXEC SQL COMMIT ;
предыдущий_P# := P# ; END DO ;
```

Следует отметить, что в конце каждой транзакции теряется информация о положении курсора в таблице деталей p (даже если курсор CP не закрывается явно, операция COMMIT все равно закрывает его автоматически). Поэтому приведенный выше код не особенно эффективен, поскольку в каждой новой транзакции приходится выполнять поиск в таблице деталей для того, чтобы снова установить правильную позицию курсора. Такое положение дел можно было бы немного улучшить при наличии индекса на столбце P# (как в действительности и должно быть в данном случае, поскольку {P#} — это первичный ключ), притом что оптимизатор выберет этот индекс в качестве пути доступа к данной таблице.

Ниже для сравнения приведено решение с использованием новых средств CHAIN и WITH HOLD.

```
EXEC SQL DECLARE CP CURSOR WITH HOLD FOR
            SELECT P.P#, P.PNAME, P.COLOR, P.WEIGHT, P.CITY FROM   P
            ORDER  BY PJ ;

eof := FALSE ;
EXEC SQL START TRANSACTION ;
EXEC SQL OPEN CP ; DO WHILE ( NOT
eof ) ; DO count := 1 TO 10 ;
        EXECSQLFETCHCPINTO:P#,..,IF SQLSTATE =
        '02000' THEN      DO ;
            EXEC SQL CLOSE CP ; EXEC
            SQL COMMIT ; eof := TRUE ;
            END DO ;
    ELSE вывести на печать P#, ... ; END IF ; END DO ;
EXEC SQL DELETE FROM P WHERE P.P# = :P# ; EXEC SQL
COMMIT AND CHAIN ; END DO ;
```

Тщательное сравнение этих двух решений оставляем читателю в качестве дополнительного упражнения.

ОТВЕТЫ К ГЛАВЕ 16

16.3.

а) Ниже приведено шесть возможных правильных результатов, соответствующих шести воз можным последовательным расписаниям.

```
Первоначально A = 0
T1-T2-T3    :  A =  1
T1-T3-T2    :  A =  2
T2-T1-T3    :  A =  1
T2-T3-T1    :  A =  2
T3-T1-T2    :  A =  4
T3-T2-T1    :  A =  3
```

Безусловно, не все шесть возможных правильных результатов являются различными. В данном конкретном примере фактически оказалось, что все возможные правильные результаты не зависят от первоначального состояния базы данных в силу самого характера транзакции Т3.

б) Существует 90 возможных различных расписаний. Эти варианты можно представить, как показано ниже. (Ri, Rj, Rk обозначают три операции RETRIEVE — Rl, R2, R3, не обязатель но в указанном порядке; аналогичным образом, Up, Uq, Ur обозначают три операции UPDATE — Ul, U2, U3, и в этом случае не обязательно в указанном порядке.)

```
Ri-Rj-Rk-Up-Uq-Ur      : 3 * 2 * 1 * 3 * 2 * 1 = 3 б      вариантов
Ri-Rj-Up-Rk-Uq-Ur      :3 * 2 * 2 * 1 * 2 * 1 =  24       варианта
Ri-Rj-Up-Uq-Rk-Ur      : 3 * 2 * 2 * 1 * 1 * 1 = 1 2     вариантов
Ri-Up-Rj-Rk-Uq-Ur      : 3 * 1 * 2 * 1 * 2 * 1 = 1 2     вариантов
Ri-Up-Rj-Uq-Rk-Ur      : 3 * 1 * 2 * 1 * 1 * 1 = 6       вариантов

                                Общее количество вариантов = 90
```

в) Ответ на этот вопрос является положительным. Например, расписание R1-R2-R3-U3-U2-U1 приводит к получению того же результата (одного), как и два из шести возможных последо вательных расписаний (в качестве дополнительного упражнения рекомендуем читателю

проверить это утверждение), поэтому оказывается "правильным" для данного начального значения, равного нулю. Но следует четко понимать, что эта "правильность" — просто является следствием удачного стечения обстоятельств и следует исключительно из того факта, что начальное значение данных оказалось равным нулю, а не чему-либо иному. В качестве контрпримера достаточно рассмотреть, что произошло бы, если бы начальное значение A было равно 10, а не нулю. По-прежнему ли расписание R1-R2-R3-U3-U2-U1, приведенное выше, вырабатывало один из изначально правильных результатов? (Объясните, что в данном случае подразумевается под выражением "изначально правильные результаты"?) Если ответ на этот вопрос отрицателен, то расписание не является упорядочиваемым. г) Ответ на этот вопрос является положительным. Например, расписание R1-R3-U1-U3-R2-U2 является упорядочиваемым (оно эквивалентно последовательному расписанию T1-T3-T2), но оно не может быть выработано, если все транзакции Tl, T2 и T3 подчиняются протоколу двухфазной блокировки. Дело в том, что в соответствии с этим протоколом операция R3 должна приобрести блокировку S на переменной отношения A от имени транзакции T3, поэтому операция U1 в транзакции T1 не сможет продолжаться до тех пор, пока эта блокировка не будет освобождена, а этого не произойдет до завершения транзакции T3 (в действительности, после достижения операции из транзакции T3 и T1 перейдут в состояние взаимоблокировки).

16.4. Во время tn ни одна из транзакций вообще не выполняет какой-либо полезной работы! Имеет место взаимоблокировка, в которой участвуют транзакции T2, T3, T9 и T8; кроме того, транзакция T4 ожидает T9, T12 ожидает T4, а T10 и T11 обе ожидают T12. Эту ситуацию можно представить с помощью графа (графа ожидания), в котором узлы обозначают транзакции, а ориентированные ребра от узла Ti к узлу т j указывают, что транзакция Ti ожидает транзакцию т j (рис. Д.8). Ребра имеют обозначения, которые указывают имена объектов базы данных и уровни блокировки, освобождения которых ожидают транзакции.

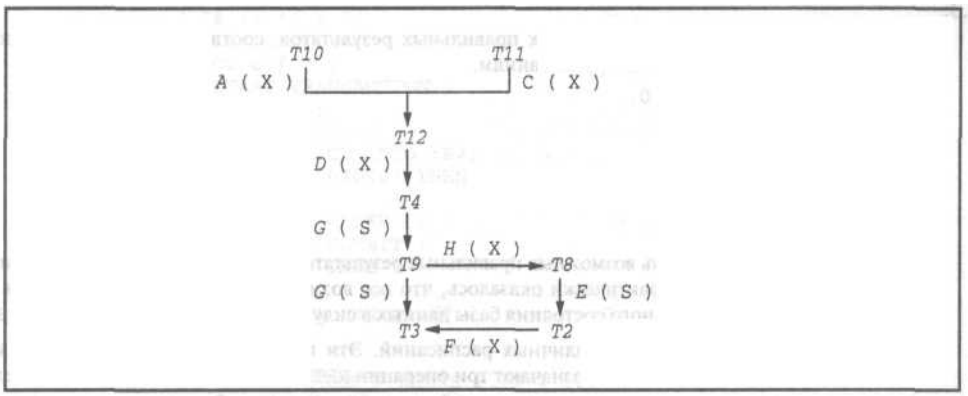


Рис. Д.8. Граф ожидания

ОТВЕТЫ К ГЛАВЕ 17

17.1

а) AUTHORITY AAA
GRANT RETRIEVE ON STATS TO Ford ;

б) AUTHORITY BBB
GRANT INSERT, DELETE ON STATS TO Smith ;

B) AUTHORITY CCC
GRANT RETRIEVE

```
      ON    STATS
      WHEN  USER () = NAME
      TO    ALL ;
```

В данном случае предполагается, что пользователи в качестве своих идентификаторов применяют собственные регистрационные имена. Обратите внимание на использование конструкции WHEN и нуль-арного встроенного оператора USER ().

Г)
```
AUTHORITY DDD
   GRANT RETRIEVE, UPDATE { SALARY, TAX } ON
   STATS TO    Nash ;
```

Д)
```
AUTHORITY EEE
GRANT RETRIEVE { NAME, SALARY, TAX } ON    STATS
TO    Todd ;
```

е)
```
AUTHORITY FFF
 GRANT RETRIEVE { NAME, SALARY, TAX },
 UPDATE { SALARY, TAX }   ON    STATS TO   Ward ;
```

Ж)
```
VAR PREACHERS VIEW
    STATS WHERE OCCUPATION = 'Preacher' ;

AUTHORITY GGG
   GRANT ALL
   ON    PREACHERS
   TO    Pope  ;
```

Следует отметить, что в данном примере необходимо использовать представление.

З)
```
VAR NONSPECIALIST VIEW
     WITH ( STATS RENAME OCCUPATION AS X ) AS T1, (
          EXTEND STATS
            ADD COUNT ( T1 WHERE X = OCCUPATION ) AS Y ) AS T2, ( T2
          WHERE Y > 10 ) AS T3 :
            T3 { ALL BUT Y }
```

AUTHORITY HHH
```
   GRANT DELETE
   ON    NONSPECIALIST TO    Jones ;
```

И)
```
VAR JOBMAXMIN VIEW
WITH ( STATS RENAME OCCUPATION AS X ) AS T1, (
     EXTEND STATS ADD
       ( MAX ( T1 WHERE X = OCCUPATION, SALARY ) AS MAXSAL, MIN (
         T1 WHERE X = OCCUPATION, SALARY ) AS MINSAL ) )
                                          AS T2 : T2 {
OCCUPATION, MAXSAL, MINSAL }

AUTHORITY III
   GRANT RETRIEVE ON JOBMAXMIN TO King ;
```

Индивидуальное средство слежения для лица с идентификатором Hal приведено ниже.

CHILDREN > 1 AND NOT (OCCUPATION = 'Homemaker') Рассмотрим следующую последовательность запросов.

COUNT (STATS WHERE CHILDREN > 1)

Результат: 6.

```
COUNT ( STATS WHERE CHILDREN > 1 AND NOT
                    ( OCCUPATION = 'Homemaker' ) )
```

Результат: 5.

Таким образом, следующее выражение

CHILDREN > 1 AND OCCUPATION = 'Homemaker'

однозначно идентифицирует Hal.

```
SUM ( STATS WHERE CHILDREN > 1, TAX )
```

Результат: 48000.

```
SUM ( STATS WHERE CHILDREN > 1 AND NOT
                ( OCCUPATION = 'Homemaker' ), TAX )
```

Результат: 46000.

Следовательно, размер налогов, выплачиваемых лицом с идентификатором Hal, равен 2000.

17.4. Общее средство слежения: SEX = 'F'.

```
SUM ( STATS WHERE SEX = 'F', TAX )
```

Результат: 70000.

```
SUM ( STATS WHERE NOT ( SEX = 'F' ), TAX )
```

Результат: 16000.

Следовательно, суммарный размер налогов равен 86000.

```
SUM ( STATS WHERE ( CHILDREN > 1 AND
                    OCCUPATION = 'Homemaker' ) OR
                    SEX = 'F', TAX )
```

Результат: 72000.

```
SUM ( STATS WHERE ( CHILDREN > 1 AND
                    OCCUPATION = 'Homemaker' ) OR NOT (
                  SEX = 'F' ), TAX )
```

Результат: 16000.

Складывая эти результаты и вычитая ранее вычисленное суммарное значение, получим, что размер налогов, выплачиваемых лицом с идентификатором Hal, равен 88000 - 86000 = 2000.

ОТВЕТЫ К ГЛАВЕ 18

18.1. а) верно; б) верно; в) верно; г) не верно; д) верно; е) не верно (это утверждение стало бы верным после замены оператора AND оператором OR); ж) не верно; з) не верно; и) верно.

18.4. Здесь показано решение для части а), что операция UNION распределяется по операции INTERSECT. Доказательство для части б), что операция INTERSECT распределяется по операции UNION, является аналогичным.

- Если t ∈ A UNION (B INTERSECT C), то t ∈ A или t ∈ (B INTERSECT C).
- Если t ∈ A, то t ∈ A UNION B и t ∈ A UNION C и, следовательно, t ∈ (A UNION B) INTERSECT (A UNION C).
- Если t ∈ B INTERSECT C, то t ∈ B и t ∈ C, поэтому t ∈ A UNION B и t ∈ A UNION C и, следовательно, (опять-таки) t ∈ (A UNION B) INTERSECT (A UNION C).
- И наоборот, если t ∈ (A UNION B) INTERSECT (A UNION C), то t ∈ A UNION B и t ∈ A UNION C. Следовательно, t ∈ A или t ∈ (принадлежит) одновременно B и C. Таким образом, t ∈ A UNION (B INTERSECT C).

18.11.

а) Будет получено множество поставщиков "не из Лондона", которые не поставляют деталь P2.

б) Будет получено пустое множество поставщиков.

в) Будет получено множество поставщиков "не из Лондона", таких, что ни один поставщик не поставляет меньшее количество видов деталей.

г) Будет получено пустое множество поставщиков.

д) Никакое упрощение не возможно.

е) Будет получено пустое множество пар поставщиков.

ж) Будет получено пустое множество деталей.

з) Будет получено множество поставщиков "не из Парижа", таких что ни один поставщик не поставляет большее количество видов деталей.

Следует отметить, что ответы на некоторые запросы, точнее, на запросы б), г), е) и ж), можно получить непосредственно с помощью ограничений.

ОТВЕТЫ К ГЛАВЕ 19

19.5. Четыре унарных оператора могут быть определены следующим образом (где A — единственный операнд).

```
A
NOT (A)
A OR NOT (A)
A  AND NOT (A)
```

Шестнадцать бинарных операторов могут быть определены следующим образом (где двумя операндами являются A и B).

```
A OR NOT (A) OR B OR NOT (B) A
AND NOT (A) AND B AND NOT (B) A
NOT (A)
B
NOT(B) A OR B A AND B A OR NOT(B)
A AND NOT(B) NOT(A) OR B NOT(A)
AND B NOT(A) OR NOT(B) NOT(A) AND
NOT(B) (NOT (A) OR B) AND (NOT(B)
OR A) (NOT (A) AND B) OR (NOT(B)
AND A)
```

Кстати, следует еще раз подчеркнуть, что в этих выражениях вполне можно применять либо операторы AND, либо операторы OR, а не те и другие вместе, поскольку, например, справедливо следующее тождество.

```
A OR B ≡ NOT ( NOT ( A ) AND NOT ( B ) )
```

19.10. Краткие определения этих операторов приведены ниже.

- Выражение EXISTS *(<table exp>)* возвращает FALSE, если таблица, обозначенная табличным выражением *<table exp>,* пуста, а в противном случае возвращает TRUE.
- Выражение UNIQUE (<table exp>) возвращает TRUE, если таблица, обозначенная табличным выражением *<table exp>,* не содержит двух разных строк, скажем, rl и r2, таких что результат операции сравнения rl = r2 равен TRUE; в противном случае выражение UNIQUE (*< table exp>)* возвращает FALSE.
- Выражение Left IS DISTINCT FROM Right (где Left и Right— выражения, обозначающие строки одной и той же самой степени, скажем, n, и i-е компоненты выражений Left и Right сопоставимы) возвращает FALSE тогда и только тогда, когда для всех i либо, во-первых, результаты операции сравнения Li = Ri равны TRUE, либо,

во-вторых, и компонент Li, и компонент Ri одновременно имеют неопределенное значение; в противном случае выражение Left IS DISTINCT FROM Right возвращает FALSE.

Эти операторы не примитивны. Ниже приведено неформальное обоснование данного утверждения.

- Выражение EXISTS(T) возвращает TRUE, если выполнение выражения SELECT COUNT!*) FROM T приводит к получению результата, отличного от нуля; в противном случае выражение EXISTS (T) возвращает FALSE.
- Выражение UNIQUE (т) возвращает TRUE, если выполнение выражения SELECT COUNT(*) FROM T = SELECT COUNT(*) FROM (SELECT DISTINCT * FROM T)) приводит К получению результата, равного TRUE; в противном случае выражение UNIQUE (T) возвращает FALSE.
- Очевидно, что конструкция IS DISTINCT FROM является сокращением (она фактически и определена именно так).

По определенным причинам оператор IS NOT DISTINCT FROM не предусмотрен (следует ли рассматривать такую ситуацию как еще один пример подхода, согласно которому "была бы несогласованной попытка исправлять несогласованности языка SQL"?).

Рассмотрим следующий запрос.

```
SELECT SPJX.S# FROM SPJ AS SPJX
WHERE SPJX.P# = P# ( 'PI[1] )
AND NOT EXISTS ( SELECT * FROM SPJ AS SPJY
                 WHERE  SPJY.S# = SPJX.S#
                 AND    SPJY.P# = SPJX.P#
                 AND    SPJY.QTY = 1000 ) ;
```

("Определить номера поставщиков, которые поставляют деталь P1, по меньшей мере для одного проекта, но только если они не поставляют деталь P1 для какого-либо из этих проектов в количестве 1000"). Предположим, что переменная отношения SPJ содержит только один кортеж с номером поставщика S1, номером детали P1, номером проекта Л и количеством QTY, равным неопределенному значению. В таком случае этот запрос возвращает номер поставщика S1, тогда как в действительности неизвестно, отвечает ли поставщик S1 заданному условию. Таким образом, полученный результат является неправильным.

Рассмотрим следующий запрос ("Верно ли, что не существует двух таких проектов, для которых поставщик S1 поставляет деталь P1 в одном и том же количестве?").

```
SELECT UNIQUE  (  SPJ. QTY  )
FROM     SPJ
WHERE    SPJ.S#  =  S#  (  'S1'  )
AND      SPJ. P#  =  P#  (  'P1'  )  ;
```

Предположим, что переменная отношения SPJ содержит только два кортежа, (S1, P1, J1, nul 1) и (S1, P1, J2, null). В таком случае данный запрос возвратит TRUE, тогда как в действительности неизвестно, должен ли он возвратить TRUE или FALSE. Таким образом, этот результат является неправильным.

ОТВЕТЫ К ГЛАВЕ 20

20.6. Поскольку центры всех прямоугольников находятся в начале координат, прямоугольник ABCD можно однозначно определить с помощью двух смежных вершин, скажем, А и В. Для того чтобы более точно определить условия данной задачи (и перейти к использованию декартовых координат), предположим, что А — это точка (xa, ya), а В — точка (xb, yb), в таком случае С — точка (-xa, -ya), а D — точка (-xb, -yb). Поскольку очевидно, что A, B, C и D лежат на окружности с центром в начале координат, безусловно, должно соблюдаться тождество $xa^2 + ya^2 = xb^2 + yb^2$. Поэтому тип RECTANGLE можно определить следующим образом.

```
TYPE RECTANGLE IS PLANE FIGURE
POSSREP { A POINT, B POINT
     CONSTRAINT THE X ( A ) ** 2 + THE Y ( A ) ** 2 =
                THE_X ( B ) ** 2 + THE_Y ( B ) ** 2 } ;
```

Такой прямоугольник является квадратом тогда и только тогда, когда для любой из вершин, скажем, для вершины B справедливо равенство (xb, yb) = (ya, -xa). Поэтому можно определить тип SQUARE следующим образом.

```
TYPE SQUARE IS RECTANGLE
     CONSTRAINT THE X ( THE B ( RECTANGLE ) ) =
     THE Y ( THE A ( RECTANGLE ) ) AND THE Y ( THE B (
     RECTANGLE ) ) = - THE X ( THE A ( RECTANGLE )
     POSSREP { A = THE_A ( RECTANGLE ) } ;
```

Примечание. Подробное описание синтаксиса спецификаций POSSREP и CONSTRAINT (который, как можно видеть в приведенных здесь двух случаях, становится другим), см. в [3.3]. Для сравнения ниже приведено еще одно решение, в котором вместо декартовых координат в качестве основы возможного представления используются полярные координаты.

```
TYPE RECTANGLE IS PLANE FIGURE POSSREP { A POINT, B
     POINT CONSTRAINT THE_R ( A ) = THE_R ( B ) } ;

TYPE SQUARE IS RECTANGLE
     CONSTRAINT ABS ( THE_θ  ( THE_A ( RECTANGLE ) ) - THE_θ ( THE_B (
                      RECTANGLE ) ) ) = Π / 2
      POSSREP { A = THE_A ( RECTANGLE ) } ;
```

Операторы, приведенные ниже, являются именно операторами обновления.

```
OPERATOR ROTATE ( T RECTANGLE ) UPDATES T
   VERSION ROTATE RECTANGLE ;
      THE_X ( THE_A ( T ) ) := - THE_X ( THE_B ( T ) ) ,
      THE_Y ( THE_A ( T ) ) := THE_Y ( THE_B ( T ) ) ,
      THE_X ( THE_B ( T ) ) := - THE_X ( THE_A ( T ) ) ,
      THE Y ( THE B ( T ) ) := THE Y ( THE A ( T ) ) ;
END OPERATOR ;

OPERATOR ROTATE ( S SQUARE ) UPDATES S
   VERSION ROTATE SQUARE ; END
OPERATOR ;
```

Следует отметить, что версию ROTATE_SQUARE по сути (вполне обоснованно) можно считать просто "пустой операцией".
Аналогичные операторы с использованием полярных координат приведены ниже.

```
OPERATOR ROTATE ( T RECTANGLE ) UPDATES T VERSION
   ROTATE RECTANGLE ;
      THE_θ ( THE_A ( T ) ) := THE_θ ( THE_A ( T ) ) + Π / 2 , THE_θ (
      THE_B ( T ) ) := THE_θ ( THE_B ( T ) ) + Π / 2 ;
END OPERATOR ;

OPERATOR ROTATE ( S SQUARE ) UPDATES S
   VERSION ROTATE SQUARE ; END
OPERATOR ;
```

В качестве дополнительного упражнения предлагаем читателю определить некоторыё'аналоги этих операторов, предназначенные только для чтения.

Ответ к этому упражнению приведен ниже.

```
OPERATOR ROTATE ( T RECTANGLE ) RETURNS RECTANGLE
   VERSION ROTATE_RECTANGLE ; RETURN RECTANGLE ( POINT (
   .- THE_X ( THE_B ( T ) ) ,
                                THE_Y ( THE_B ( T ) ) ),
                           POINT ( - THE X ( THE A ( T ) ) ,
                                THE Y ( THE A ( T ) ) ) ) ;
END OPERATOR ;

OPERATOR ROTATE ( S SQUARE ) RETURNS SQUARE
   VERSION ROTATE SQUARE ;
RETURN S ; END OPERATOR ;
```

Аналогичные операторы с использованием полярных координат приведены ниже.

```
OPERATOR ROTATE ( T RECTANGLE ) RETURNS RECTANGLE
VERSION ROTATE RECTANGLE ; RETURN       RECTANGLE (
POINT ( THE_R ( THE_A ( T ) ),
                           THE θ  ( THE A ( T ) ) + Π / 2 ) ,
                    POINT ( THE_R ( THE_B ( T ) ),
                           THE θ  ( THE B ( T ) ) + Π / 2 ) ) ;
END OPERATOR ;

OPERATOR ROTATE ( S SQUARE ) RETURNS SQUARE
   VERSION ROTATE_SQUARE ;
   RETURN S ;
END OPERATOR ;
```

ОТВЕТЫ К ГЛАВЕ 22

22.8. Существует восемь ($= 2^3$) возможных группировок для каждой иерархии, поэтому общее коли чество вариантов составляет $8^4 = 4096$. В качестве дополнительного упражнения рекомендуем читателю определить, как можно получить все эти агрегированные данные с помощью языка SQL. Более подробный ответ к этому упражнению не предусмотрен (к тому же данный вопрос в общем-то является риторическим).

22.9. Применительно к этим запросам SQL, ниже показаны только конструкции GROUP BY.

а) `GROUP BY GROUPING SETS ( (S#,P#), (P#,J#), (J#,S#) )`

б) `GROUP BY GROUPING SETS ( J#, (J#,P#), () )`

в) Ловушка здесь в том, что этот запрос является двусмысленным, поскольку выражение (например) "формирование свертки вдоль размерности поставщиков" может иметь много различных толкований. Но одна возможная интерпретация этого требования приводит к по лучению конструкции GROUP BY, которая выглядит следующим образом.

```
GROUP BY ROLLUP (S#), ROLLUP (P#)
```

г) `GROUP BY CUBE ( S#, P# )`

Результирующие таблицы SQL здесь не показаны. А что касается перекрестных таблиц, то должно быть очевидно, что перекрестные таблицы — не очень удобный способ отображения результата, который включает больше двух измерений (и чем больше имеется измерений, тем хуже обстоят дела). Например, одна из таких перекрестных таблиц, соответствующих конструкции GROUP BY S#, P#, J#, может частично выглядеть, как показано в табл. Д. 1.

По существу, общий вывод состоит в следующем: заголовки перекрестных таблиц являются гро моздкими, а представленные в них массивы — разреженными.

Таблица Д.1. Пример перекрестной таблицы

	P1				P2				...
	J1	J2	J3	...	J1	J2	J3	...	...
S1	200	0	0	...	0	0	0	...	...
S2	0	0	0	...	0	0	0	...	...
S3	0	0	0	...	0	0	0	...	...
S4	0	0	0	...	0	0	0	...	...
S5	0	200	0	...	0	0	0	...	...
..	...	...	...	...	...	...	...	...	...

ОТВЕТЫ К ГЛАВЕ 23

23.6. INTERVAL_INTEGER
[MIN (MIN (BEGIN(i1), BEGIN(i2)), BEGIN(i3)) :
MAX (MAX (END(i1), END(i2)), END(i3))]

Для определенности предполагается, что INTEGER используется в качестве базового типа точки. Следует отметить, что приведенное ниже выражение может оказаться неприменимым, поскольку операция UNION не обязательно определена для каждой пары интервалов, взятых из указанных трех.

```
i1 UNION i2 UNION i3
```

23.7. Ответ на этот вопрос является положительным, если выражение в правой части определено; в противном случае он отрицателен. Ниже приведено три примера (в упрощенной системе обозначений).

- a = [2:6), b = [4:9]; a INTERSECT b = [4;6], a MINUS **(a MINUS b)** = [2:6] MINUS [2:3] = [4:6].
- a = [4:6], Ь = [2:6]; a INTERSECT b = [4:6], a MINUS b **(И поэтому выражение** a MINUS (a MINUS b) **не определено).**
- **а** = [4:6], b = [8:9]; **выражение** a INTERSECT b **не определено, а** MINUS **b = [4 : б], выражение** a MINUS (a MINUS b) **не определено.**

23.8.

а) Предположим, что существует общее упорядочение номеров деталей, скажем, P1 < P2 < P3 (и т.д.). В таком случае следующее отношение может интерпретироваться как содержащее информацию о том, что некоторые поставщики были способны поставлять некоторые детали, указанные с помощью интервалов номеров деталей, в течение некоторых интервалов времени.

S#	PARTS	DURING
S1	[P1:P3]	[d01:d04]
S1	[P2:P4]	[d07:d08]
S1	[P5:P6]	[d09:d09]
S2	[P1:P1]	[d08:d09]
S2	[P1:P2]	[d08:d08]
S2	[P3:P4]	[d07:d08]
S3	[P2:P4]	[d01:d04]
S3	[P3:P5]	[d01:d04]
S3	[P2:P4]	[d05:d06]
S3	[P2:P4]	[d06:d09]
S4	[P3:P4]	[d05:d08]

б) Следующее отношение может интерпретироваться как содержащее информацию о том, что некоторые поставщики, указанные с помощью интервалов номеров поставщиков, были способны поставлять некоторые детали, указанные с помощью интервалов номеров деталей, в течение некоторых интервалов времени.

в) См. приведенный выше вариант б).

```
23.10. WITH ( FEDERAL GOVT RENAME DURING AS FD ) AS FG ,
            ( STATE GOVT RENAME DURING AS SD ) AS SG ,
            ( FG JOIN SG ) AS T1 , ( T1 WHERE FD
           OVERLAPS SD ) AS T2 ,
            ( EXTEND T2 ADD ( FD INTERSECT SD ) AS DURING ) AS T3 :
       T3 { ALL BUT FD, SD }
```

ОТВЕТЫ К ГЛАВЕ 24

24.1. а) Верно; б) Верно; в) Не верно.

24.2. В приведенных ниже выражениях a, Ь, и c — скулемовские константы, а f — скулемовская функция с двумя фактическими параметрами.

а) p (x, y) $\Rightarrow$ q (x, f (x, y))

б) p (a, Ь) $\Rightarrow$ q (a, z)

В) p (a , b) $\Rightarrow$ q (a , c)

24.3. Рассмотрим только часть а) этого упражнения. В соответствии с его заданием, определены при веденные ниже предложения.

1. WOMAN (Eve)
2. PARENT (Eve, Cain)
3. MOTHER (x, y) $\Leftarrow$ PARENT (x, y) AND WOMAN (X)

Представим предложение 3 в другой форме, чтобы устранить оператор " $\Leftarrow$ ".

4. MOTHER (X, y) OR NOT PARENT (X, y) OR NOT WOMAN (X)

Сформулируем отрицание заключения и примем его в качестве предпосылки.

5. NOT MOTHER (Eve, Cain)

Подставим Eve вместо x и Cain вместо y в строке 4 и применим правило резолюции к полученному результату и к строке 5.

6. NOT PARENT (Eve, Cain) OR NOT WOMAN (Eve)

Применим правило резолюции к строкам 2 и 6.

7. NOT WOMAN (Eve)

Применим правило резолюции к строкам 1 и 7. Будет получено пустое множество предложений, [ ].

24.6. Приведенные ниже решения перенумерованы как 24.6.п, где п — номер исходного упражнения в главе 7, т.е. упр. 7.п. Как и в основной части главы 24, например, литерал 300 применяется в качестве удобного краткого обозначения вместо QTY(300) и т.д.

24.6.13. `? ⇐ j ( j, jn, jc )`

24.6.16.
```
? ⇐ SPJ ( s, p, j, q ) AND 300 < q AND q < 750
```

24.6.22.
```
RES ( p ) ⇐ SPJ( s, p, j, q ) AND
            S ( s, sn, st, London ) AND
            J ( j, jn, London.
? ⇐ RES ( p )
```

24.6.31.
```
RES ( jn ) ⇐ J ( j, jn, jc ) AND
SPJ ( S1, p, j, q )
? ⇐ RES ( jn )
```

24.6.34.
```
RES ( j ) ⇐ SPJ ( s, p, j, q ) AND
            SPJ ( S1, p, J2, q2 )
? ⇐ RES ( j )
```

24.6.46.
```
RES ( p ) ⇐ SPJ( s, p, j, q ) AND
            S ( s, sn, st, London )
          RES ( p ) ⇐ SPJ ( S, p, j, q ) AND
            J ( j, jn, London )
? ⇐ RES ( p )
```

ОТВЕТЫ К ГЛАВЕ 25

25.10. Существуют, по меньшей мере, следующие девять возможных иерархий.

S contains (P contains (J))
S contains (J contains (P))
S contains (P and J)
P contains (J contains (S))
P contains (S contains (J)),
P contains (J and S)
J contains (S contains (P))
J contains (P contains (S))
J contains (S and P)

На вопрос о том, какая из них является наилучшей, нельзя найти ответ без дополнительной информации, но почти наверняка все они являются неприемлемыми. Это означает, что при выборе любой из этих иерархий всегда будут возникать многочисленные проблемы, трудно разрешимые в рамках данной конкретной иерархии.

25.11. Прежде всего, существует девять "очевидных" проектов, рассматриваемых в ответе на предыдущее упражнение. Но может быть также предусмотрено много других потенциальных проектов, например, с применением класса "SP", который непосредственно показывает, какие поставщики поставляют те или иные детали, а также включает два вложенных множества проектов — один для поставщиков и один для деталей. Возможен также очень простой проект, вообще не предусматривающий использование каких-либо (нетривиальных) иерархий и состоящий из класса "SP", класса "PJ" и класса " JS".

ОТВЕТЫ К ГЛАВЕ 26

26.3. Аналогичная проблема при использовании внешних ключей не возникает. Для того чтобы определить, с чем это связано, возвратимся к первоначальному примеру из раздела 26.3. ***Примечание.*** Приведенное ниже пояснение взято из [3.3], приложение G, стр. 421—422.

```
VAR E ELLIPSE ;
VAR XC REF_TO_CIRCLE ;
```

```
E := CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) ;
XC := TREAT DOWN AS REF TO CIRCLE ( REF TO ( E ) ) ;
THE_A ( E ) := LENGTH ( 6.0 ) ;
```

Игнорируя подробности, не относящиеся к соответствующему способу представления, можно предложить примерно следующий реляционный аналог данного примера.

```
VAR R1 ... RELATION { K ELLIPSE ... } KEY { K } ;

VAR R2 . . . RELATION { K CIRCLE . . . }
           FOREIGN KEY { K } REFERENCES R1 ;
```

Для простоты здесь предполагается, что не должны быть заданы какие-либо "ссылочные действия", например, каскадное обновление и т.д. (это упрощающее предположение никоем образом существенно не влияет на приведенные ниже доводы). Следует отметить, что каждое значение K в R1, которое "согласуется" с некоторым значением K в R2, должно принадлежать к типу CIRCLE, а не просто к типу ELLIPSE.

Теперь вставим в каждую из этих двух переменных отношения такое отношение, которое содержит только один кортеж.

```
INSERT R1
RELATION {
TUPLE { K CIRCLE ( LENGTH ( 5.0 ), POINT ( 0.0, 0.0 ) ) } } ;

INSERT R2
RELATION {
TUPLE { K CIRCLE ( LENGTH ( 5.0 ), POINT (0.0,0.0) ) } } ;
```

Наконец, попытаемся обновить кортеж в R1.

```
UPDATE R1 { THE_A ( K ) := LENGTH ( 6.0 ) } ;
```

В данной операции UPDATE предпринимается попытка обновить окружность в одном кортеже отношения R1, чтобы преобразовать ее в объект типа ELLIPSE (Безусловно, в этом описании применяются совершенно неформальные выражения!). Если бы эта попытка завершилась успешно, то значение K в R2 ссылалось бы на "некруглую окружность", но эта попытка не будет успешной; вместо этого выполнение этой операции UPDATE закончится неудачей из-за нарушения ограничения ссылочной целостности.

Примечание. Верно, что могут происходить ошибки на этапе прогона (а именно, ошибки из-за нарушения ограничения ссылочной целостности), однако, вообще говоря, нарушения целостности на этапе прогона всегда возможны. По крайней мере, будет создана такая система, в которой с помощью ограничений поддерживаются операции выборки и обобщения, поддерживаются также ограничения типа, и не могут появляться такие аномалии, как "некруглые окружности" и т.п. (И, в частности, ошибки преобразования типа на этапе прогона могут возникать только в связи с выполнением оператора TREAT DOWN.)

ОТВЕТЫ К ГЛАВЕ 27

27.3. Для краткости приведенный ниже ответ был во многом упрощен, например, удалены номера глав и разделов, а также номера страниц, кроме того, не указана кодировка символов в символьных данных". Но в целом приведенные здесь решения вполне позволяют получить общее представление о формате документов XML.

```
<?xml version="1.0"?>
<!-- Документ XML, представляющий содержание книги -->
<!DOCTYPE Contents [
   <!ELEMENT Contents (Preface?, Part+, Appendixes*, Index)>
   <!ELEMENT Preface (#PCDATA)>
   <!ELEMENT Part (Chapter+)>
      <!ATTLIST Part title CDATA #REQUIRED>
```

[11] Но поскольку элементы присутствуют в определенном порядке, на основании этого представления на языке XML можно, по крайней мере, определить номера глав и разделов. В отличие от этого, номера страниц, безусловно, определить невозможно.

```
   <!ELEMENT Chapter (Introduction, Section+, Summary,
   Exercises?, Refs-Bib, Answers?)> <!ATTLIST Chapter title
   CDATA #REQUIRED> <!ELEMENT Introduction EMPTY> <!ELEMENT
   Section (#PCDATA)> <!ELEMENT Summary EMPTY> <!ELEMENT
   Exercises EMPTY> <!ELEMENT Refs-Bib EMPTY> <!ELEMENT
   Answers EMPTY> <!ELEMENT Appendixes (Appendix+)>
   <!ELEMENT Appendix (Introduction?, Section*)>
   <!ATTLIST Appendix title CDATA #REQUIRED> <!ELEMENT
Index EMPTY> ]> <Contents>
   <Preface>Предисловие к восьмому изданикк/Preface> <Part
   title="Основные понятия">
      <Chapter title="Базы данных и управление ими">
         <Introduction/>
         <Section>Общее определение системы баз данных</Section>
         <Section>Общее определение базы данных</Section>
         <Section>Назначение баз данных</Section> <Section>Независимость
         от данных</Section>
        <Sec^on>Реляционные и другие системы</Section>
      <Summary/> <Exercises/> <Refs-Bib/> </Chapter> <Chapter
      title="Архитектура системы баз данных">

      </Chapter> </Part> <Part
   title="Реляционная модель">

   </Part>
   <Appendixes>

      <Appendix title="Выражения SQL">
         <Introduction>

   </Appendix>
</Appendixes> <Index/>
</Contents>
```

Ниже приведен еще один возможный ответ. В нем используется меньше структурных ограничений и не так широко представлены средства, которые не рассматривались в основной части главы 27.

```
<?xml version="l.0"?>
<!-- Документ XML, представляющий содержание книги -->
<!DOCTYPE Contents [
   <!ELEMENT Contents (Preface?, Part+, Appendixes*, Index)>
   <!ELEMENT Preface (#PCDATA)> <!ELEMENT Part (Chapter+J>
      <!ATTLIST Part title CDATA #REQUIRED>
   <!ELEMENT Chapter (Section+)>
      <!ATTLIST Chapter title CDATA #REQUIRED>
   <!ELEMENT Section (#PCDATA)> <!ELEMENT Appendixes
   (Appendix+)> <!ELEMENT Appendix (Section*)>
      <!ATTLIST Appendix title CDATA #REQUIRED>
```

```
   <!ELEMENT Index EMPTY>
]> <Contents>
   <Preface>Предисловие к восьмому изданию</Preface>
   <Part title="Основные понятия">
      <Chapter title="Базы данных и управление ими">
         <Section>Вводный пример</Section>
      <Section>Общее определение системы баз данных</Section>
      <Section>Обшее определение базы данных</Section>
      <Section>Назначение баз данных</Section>
      <Section>Независимость от данных</Section>
      <Sec^on>Реляционные и другие системы</Section>
      <Section>Резюме</Section> <Section>Упражнения</Section>
      <Section>Список литературы</Section> </Chapter> <Chapter
      title="Архитектура системы баз данных">

      </Chapter> </Part> <Part
   title="Реляционная модель">

    </Part>
    <Appendixes>

      <Appendix title="Выражения SQL">
         <Section>Введение</Section>

   </Appendix>
</Appendixes>
<Index/>
</Contents>
```

27.12. Схемы могут быть сформулированы с помощью многих различных способов. Один из радикальных подходов состоит в том, что все элементы могут быть определены как глобальные (т.е. как непосредственные дочерние элементы элемента xsd: schema), после чего на них по мере необходимости сформированы перекрестные ссылки. Этот подход является особенно удобным, если определения типов или элементов должны быть разделяемыми; он позволяет избежать применения избыточных и потенциально несовместимых определений. Еще один радикальный подход, который иллюстрируется приведенным ниже ответом, состоит в том, что глобальным должен быть объявлен только корневой элемент, а все дочерние элементы определены как содержащиеся в этом корневом элементе (на некотором уровне). Обратите внимание на то, что в данном случае приходится повторять определение элемента Section (раздел), поскольку разделы имеются и в главах, и в приложениях). Но так как элемент Section является весьма простым (это связано с тем, что он представляет собой просто данные типа xsd: string), необходимость его повторения нельзя назвать такой уж обременительной.

```
<?xml version="1.0"?>
<!-- Схема ко второму ответу на упр. 27.3 -->
<!DOCTYPE xsd:schema SYSTEM
          "http://www.w3.org/2001/XMLSchema.dtd">

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd: element name=" Contents ">'*
    <xsd:complexType>
      <xsd:sequence>

        <xsd:element name="Preface" type="xsd:string"
                                    minOccurs="0" />
```

```
<xsd:element name="Part" maxOccurs="unbounded">
  <xsd:complexType> <xsd:sequence>

      <xsd:element name="Chapter">
        <xsd:complexType>
        <xsd:sequence>

            <xsd:element name="Section" type="xsd:string"
                                        maxOccurs="unbounded" />
          </xsd:sequence>
        <xsd:attribute name="title" type="xsd:string" />
    </xsd:complexType> </xsd:element> </xsd:sequence>
  <xsd:attribute name="title" type="xsd:string" />,
</xsd:complexType> </xsd:element>

<xsd:element name="Appendixes">
  <xsd:complexType>
  <xsd:sequence>

      <xsd:element name="Appendix">
        <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="Section" type="xsd:string"
                                        maxOccurs="unbounded" />
          </xsd:sequence>
          <xsd:attribute name="title" type="xsd:string" />
        </xsd:complexType>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

   <xsd:element name="Index" type="xsd:string" />
</xsd:sequence>       </xsd:complexType>      </xsd:element>
</xsd:schema>
```

27.19.

```
<Result>
   { for $p in document("PartsRelation.xml")//PartTuple
                                        [SCOLOR = "Green"]
     return <GreenPart> {$p} </GreenPart>
   }
</Result>
```

27.20. Полученный результат имеет примерно следующий вид.

```
<Parts>
6
</Parts>
```

27.22. Поскольку этот документ не включает каких-либо непосредственных дочерних элементов типа Supplier, конструкция return никогда не выполняется и результатом становится пустая последовательность.

Примечание. Если бы этот запрос был сформулирован несколько **иначе, например:**

```
<Result>
{ for $sx in document("SuppliersOverShipments.xml")/
```

```
                Supplier [CITY = 'London']
  return
    <whatever>
    { $sx/SNUM, $sx/SNAME, $sx/STATUS, $sx/CITY }
</whatever> } </Result>
```

то был бы получен примерно следующий результат.

```
<Result>
</Result>
```

27.25. На ум сразу же приходят, по крайней мере, изложенные ниже замечания.

- Некоторые из этих функций выполняют такие действия, которые по существу представляют собой преобразование типов. Например, выражение XMLFILETOCL0B (' Bol tDrawing. svg') обычно принято записывать примерно следующим образом.

  ```
  CAST_AS_CLOB ( 'BoltDrawing.svg' )
  ```

  Иными словами, XMLDOC следует рассматривать как вполне сформировавшийся тип (см. раздел 27.6, подраздел "Документы как значения атрибутов").

- Аналогичным образом, выражение XMLCONTENT (DRAWING, 'RetrievedBoltDraw ing . svg') обычно принято записывать следующим образом.

  ```
  DRAWING := CAST_AS_XMLDOC ( 'RetrievedBoltDrawing.svg' ) ;
  ```

  В действительности, XMLCONTENT представляет собой оператор обновления (см. главу 5) и вся эта идея, состоящая в том, чтобы иметь возможность вызывать его из операции только чтения (т.е. из операции, аналогичной операции SELECT в языке SQL), является более чем подозрительной [3.3].

- Еще раз рассмотрим выражение XMLFILETOCLOB ('BoltDrawing. svg'). Очевидно, что применяемый здесь фактический параметр по своему типу представляет собой символьную строку. Но эта символьная строка интерпретируется (в действительности, она разадресовывается — см. главу 26), а это означает, что она не может быть просто любой символьной строкой традиционного типа. В действительности, функция XMLFILETOCLOB весьма напоминает операцию EXECUTE IMMEDIATE динамического SQL (см. главу 4).

- Замечания, аналогичные приведенным в предыдущем абзаце, относятся также к таким фактическим параметрам, как представленный ниже (см. пример применения операции XMLEXTRACTREAL).

  ```
  '//PartTuplefPNUM = "P3"]/WEIGHT'
  ```

Предметный указатель

Научно-популярное издание К.Дж.Дейт

Введение в системы баз данных
8-е издание

Литературный редактор *С. Г. Татаренко*
Верстка *М.А. Смолина* Художественный редактор *С.А. Чернокозинский*
Корректоры *Горохова В.Ю. Орындаш А.А.*
Сканирование *Петраки О.*

Издательский дом "Вильяме". 101509, Москва, ул. Лесная, д. 43, стр. 1.

Подписано в печать 28.02.2005. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 118,68. Уч.-изд. л. 86,19.
Тираж 3000 экз. Заказ № 859.

Отпечатано с диапозитивов в ФГУП "Печатный двор"
Министерства РФ по делам печати,
телерадиовещания и средств массовых коммуникаций.
197110, Санкт-Петербург, Чкаловский пр., 15.

Ученье - свет, а неученье - тьма
народная мудрость.

Да будет Свет! - сказал Господь
божественная мудрость

NataHaus - Знание без границ:
Скромное воплощение народной и божественной мудрости.:-)

библиотека
форум
каталог
