# I. PROCEDURI STOCATE/ХРАНИМЫЕ ПРОЦЕДУРЫ IN MYSQL

http://www.mysqltutorial.org/mysql-stored-procedure-tutorial.aspx

## Proceduri stocate MySQL

## Secțiunea 1. Elementele de bază ale procedurilor stocate

- Introducere în procedurile stocate în MySQL - se prezentă procedurile stocate, avantajele și dezavantajele acestora.

Procedurile stocate reprezinta secvente de cod SQL care pot fi rulate pe server pentru a indeplini anumite sarcini. Procedurile stocate sunt salvate in baza de date si pot fi apelate la un moment dat de un program, declansator (trigger) sau chiar de o alta procedura stocata.

**Principalele AVANTAJE ale utilizarii procedurilor stocate pot fi sintetizate astfel:**

• procedurile stocate cresc performantele aplicatiilor; dupa creare, procedurile stocate sunt compilate si salvate in baza de date; in felul acesta, ele vor rula mai rapid decat comenzile SQL care sunt trimise din aplicatii;
• procedurile stocate reduc traficul intre aplicatii si serverul de baze de date, deoarece aplicatiile nu mai trimit secvente de cod SQL necompilate, ci doar numele unor proceduri stocate pe server;
• procedurile stocate permit reutilizarea secventelor de cod SQL; prin intermediul lor sunt oferite anumite functionalitati, care nu trebuie implementate pentru fiecare aplicatie in parte;
• procedurile stocate sunt sigure; administratorii bazelor de date stabilesc aplicatiile care pot accesa anumite proceduri stocate, fara a acorda privilegii pe tabelele bazei de date.

**Totusi, procedurile stocate prezinta si anumite DEZAVANTAJE, dintre care pot fi amintite urmatoarele:**

• procedurile stocate contin, in general, instructiuni SQL, motiv pentru care nu este posibila implementarea unor facilitati complexe, oferite de limbajele de programare;
• procedurile stocate necesita aptitudini specializate in randul dezvoltatorilor;
• procedurile stocate solicita memoria si puterea de procesare a serverului de baze de date, prin rularea unor operatii complexe.

Sistemul MySQL ofera suport pentru proceduri stocate incepand cu versiunea 5.0. Crearea unei proceduri stocate se realizeaza cu ajutorul instructiunii CREATE PROCEDURE, care prezinta urmatoarea sintaxa in MySQL:

**Хранимые процедуры - это последовательности кода SQL, которые можно запустить на сервере для выполнения определенных задач. Хранимые процедуры сохраняются в базе данных и могут быть вызваны в любое время программой, триггером или даже другой хранимой процедурой.**

**Основные ПРЕИМУЩЕСТВА использования хранимых процедур можно резюмировать следующим образом:**

1. **хранимые процедуры увеличивают производительность приложений; после создания хранимые процедуры компилируются и сохраняются в базе данных; таким образом они будут выполняться быстрее, чем команды SQL, отправляемые из приложений;**
2. **хранимые процедуры уменьшают трафик данных между приложениями и сервером базы данных, поскольку приложения больше не отправляют некомпилированные последовательности кода SQL, а только имена процедур, хранящихся на сервере;**
3. **хранимые процедуры позволяют повторно использовать последовательности кода SQL; через них предлагаются определенные функции, которые не обязательно реализовывать для каждого приложения;**
4. **хранимые процедуры безопасны; администраторы баз данных создают приложения, которые могут обращаться к определенным хранимым процедурам без предоставления привилегий таблицам базы данных.**

**Однако хранимые процедуры также имеют определенные НЕДОСТАТКИ, из которых можно отметить следующие:**

1. **хранимые процедуры обычно содержат операторы SQL, поэтому невозможно реализовать сложные функции, предлагаемые языками программирования;**
2. **хранимые процедуры требуют специальных навыков от разработчиков;**
3. **Хранимые процедуры требуют памяти и вычислительной мощности сервера базы данных для выполнения сложных операций.**

**MySQL обеспечивает поддержку хранимых процедур, начиная с версии 5.0. Создание хранимой процедуры выполняется с помощью оператора CREATE PROCEDURE, который в MySQL имеет следующий синтаксис:**

```
CREATE
    [DEFINER = {utilizator | CURRENT_USER}]
    PROCEDURE nume_procedura_stocata ([IN | OUT | INOUT nume_parametru tip_parametru
    [,...]])
    SQL SECURITY {DEFINER | INVOKER}
    corp_procedura
```

- Clauzele **DEFINER si SQL SECURITY** specifica contul MySQL care urmeaza a fi utilizat pentru a verifica privilegiile, la rularea procedurii stocate.
- Daca pentru clauza **DEFINER** este specificata o valoare, aceasta trebuie sa corespunda unui utilizator de pe serverul MySQL (**user_name@host_name**). **Valoare implicita pentru clauza DEFINER este aceeasi cu numele utilizatorului care executa instructiunea CREATE PROCEDURE**.
- Valorile permise pentru clauza **SQL SECURITY** sunt **DEFINER si INVOKER**. Acestea indica faptul ca procedura stocata va fi executata cu privilegiile utilizatorului care a creat procedura stocata sau cu cele ale utilizatorului care o invoca***(o apelează).*** Utilizatorul care creeaza sau invoca o procedura stocata trebuie sa detina permisiunea de a accesa baza de date cu care procedura stocata este asociata.
- **Valoarea implicita** pentru clauza **SQL SECURITY** este **DEFINER**. Daca valoarea corespunzatoare clauzei **SQL SECURITY** este **DEFINER** si contul indicat in clauza **DEFINER** nu exista cand procedura stocata este executata, atunci este generata o eroare.
- **Pentru a utiliza** instructiunea **CREATE PROCEDURE** este necesar privilegiul **CREATE ROUTINE**. De exemplu, in cazul sistemelor de tip MySQL sunt automat alocate privilegiile **ALTER ROUTINE** si **EXECUTE** pentru utilizatorul care creeaza o procedura stocata.
- **Pentru a marca finalul** unei instructiuni **CREATE PROCEDURE** este utilizat un **delimitator**. **Acesta poate fi cel implicit, simbolul (;),** sau unul stabilit prin intermediul instructiunii **DELIMITER**.

**O astfel de abordare este utila, daca avem in vedere faptul ca o procedura stocata poate include mai multe instructiuni SQL delimitate prin simbolul (;).** In acest caz, trebuie utilizata instructiunea DELIMITER, **inaintea crearii unei proceduri stocate,** pentru a stabili simbolul care va marca finalul procedurii stocate.

## Corpul unei proceduri stocate este delimitat de cuvintele cheie BEGIN, respectiv END, si cuprinde instructiuni SQL.

**Folosirea procedurilor stocate mysql, cu phpmyadmin**

În cele din urmă vom începe munca de creare și interogare a procedurilor stocate cu instrumentul **phpmyadmin**, dar orice SGBD care acceptă interogări SQL de la MYSQL 5.0 poate fi utilizat pentru astfel de activitati!

**ÎN ACEST CAZ, VOM LUA BAZA DE DATE LIBRARIE.**

**1) Introduceți phpmyadmin și de acolo selectam baza de date**

Există 2 motoare de stocare care gestionează date în Mysql

1. **MyISAM:** motor implicit, foarte rapid pentru interogări, nu oferă integritatea datelor sau protecție referențială. Sisteme ideale cu multe întrebări
2. **InnoDB:** oferă protecție referențială și integritate a datelor pe lângă blocarea înregistrărilor, ideal dacă veți insera, edita sau șterge multe informații în mod constant. **În general, pentru procedurile stocate, este mai bine să folosiți InnoDB.**

- Modificarea delimitatorului implicit în MySQL - aflați cum puteți schimba delimitatorul implicit în MySQL.
- Crearea de noi proceduri stocate - se arată cum puteți crea folosi CREATE PROCEDURE pentru a crea o nouă procedură stocată în baza de date.
- Eliminarea procedurilor stocate - se arată cum să utilizați DROP PROCEDURE pentru a renunța la o procedură stocată existentă.
- Variabile - vă ghidează cum să utilizați variabile pentru a păstra rezultatul imediat în cadrul procedurilor stocate.
- Parametri - se prezintă diverse tipuri de parametri folosiți în procedurile stocate, inclusiv parametrii IN , OUT și INOUT .
- Modificarea procedurii stocate - se arată pas cu pas cum puteți modifica o procedură stocată folosind o secvență de DROP PROCEDURE și CREATE PROCEDURE instrucțiuni în MySQL Workbench.
- Listarea procedurilor stocate - se oferă câteva comenzi utile pentru listarea procedurilor stocate din bazele de date.

## Secțiunea 2. Declarații condiționale

- Instrucțiunea IF - se arată cum să utilizați instrucțiunea IF THEN în procedurile stocate.
- Instrucțiunea CASE - se prezintă declarațiile CASE , inclusiv instrucțiunile CASE simple și declarațiile CASE căutate.

## Secțiunea 3. Bucle

- LOOP - aflați cum să executați o listă de declarații în mod repetat pe baza unei condiții.
- WHILE Loop - se arată cum să executați o buclă atâta timp cât o condiție este adevărată.
- REPEAT Loop - se arată cum să executați o buclă până când o condiție de căutare este adevărată.
- Instrucțiunea LEAVE - vă îndrumă cum să ieșiți dintr-o buclă imediat.

## Secțiunea 4. Gestionarea erorilor

- Expedierea excepțiilor - se arată cum să gestionați excepțiile și erorile în procedurile stocate.

- Creșterea erorilor - aflați cum să folosiți SIGNAL și RESIGNAL pentru a ridica erorile în procedurile stocate.

**Secțiunea 5. Cursori**

- Cursori - aflați cum să utilizați cursoarele pentru a procesa rând pe rând într-un set de rezultate.

**Secțiunea 6. Funcții stocate**

- Crearea unei funcții stocate - se arată cum puteți utiliza funcțiile create stocate în baza de date.
- Eliminarea unei funcții stocate - utilizați DROP FUNCTION pentru a elimina o funcție stocată.
- Listarea funcțiilor stocate - aflați cum să enumerați toate funcțiile stocate în baza de date folosind SHOW FUNCTION STATUS sau interogând din dicționarul de date.

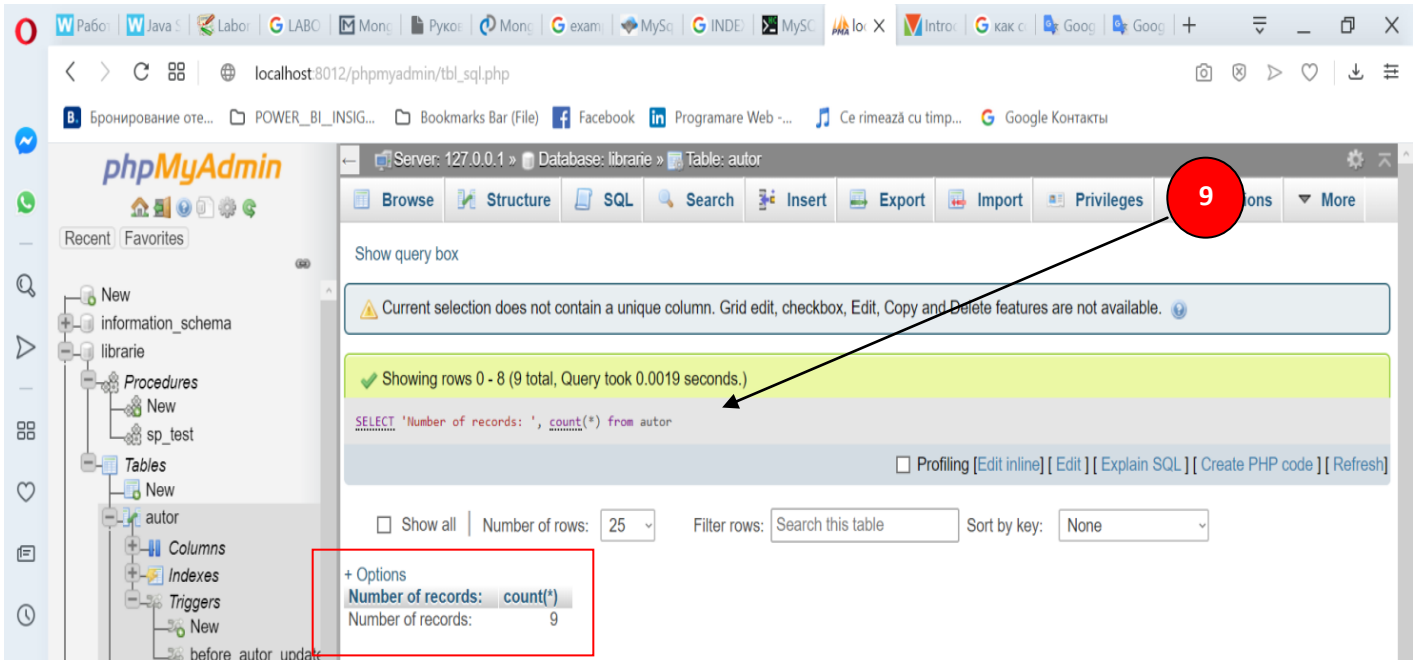
**Secțiunea 7. Securitatea programelor stocate**

- Control acces obiect - aflați cum puteți controla securitatea obiectelor stocate.

**SA CREĂM CEA MAI SIMPLA PROCEDURA STOCATA PENTRU TABELA LIBRARIE.AUTOR**

**Sa urmarin atent rezultatele comenzii**

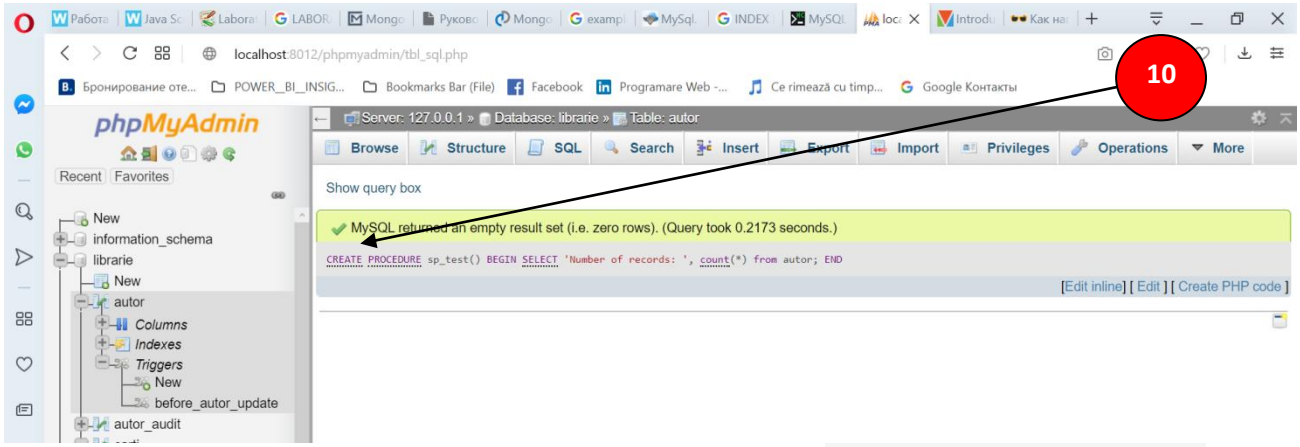
**SELECT 'Number of records:', COUNT(*) from autor**




**In cele ce urmeaza sa culegem comenzile in punctual de meniu SQL**

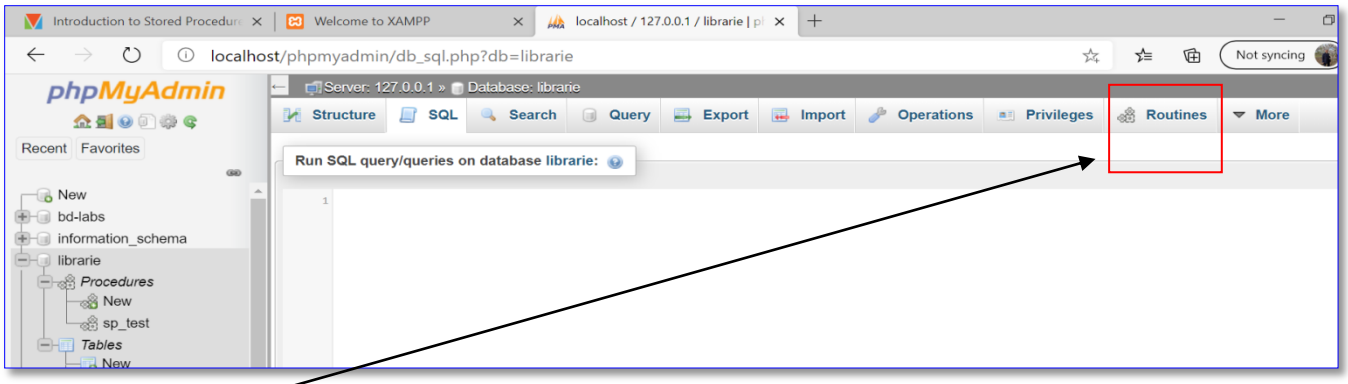
```
DELIMITER //
CREATE PROCEDURE sp_test()
BEGIN
  SELECT 'Number of records: ', count(*) from autor;
END//
DELIMITER ;
```

Apăsăm tasta F5, Refresh

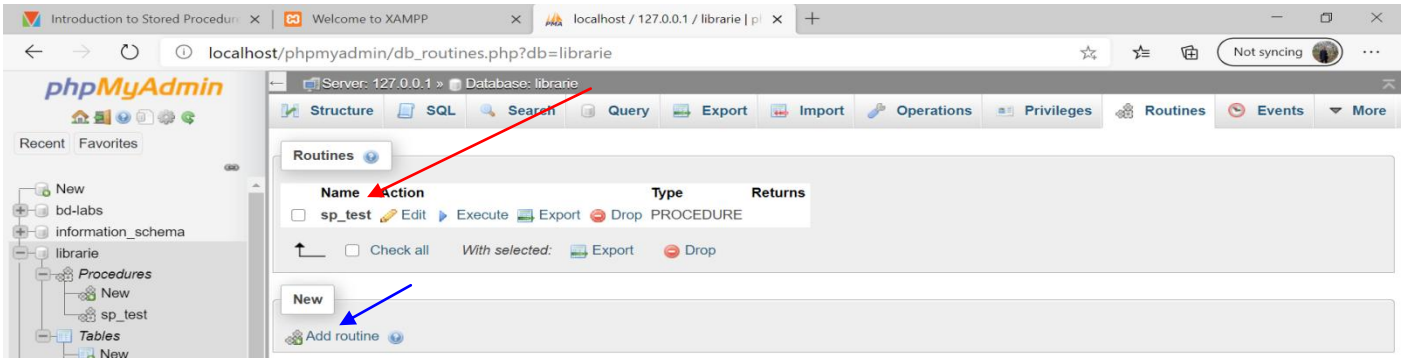


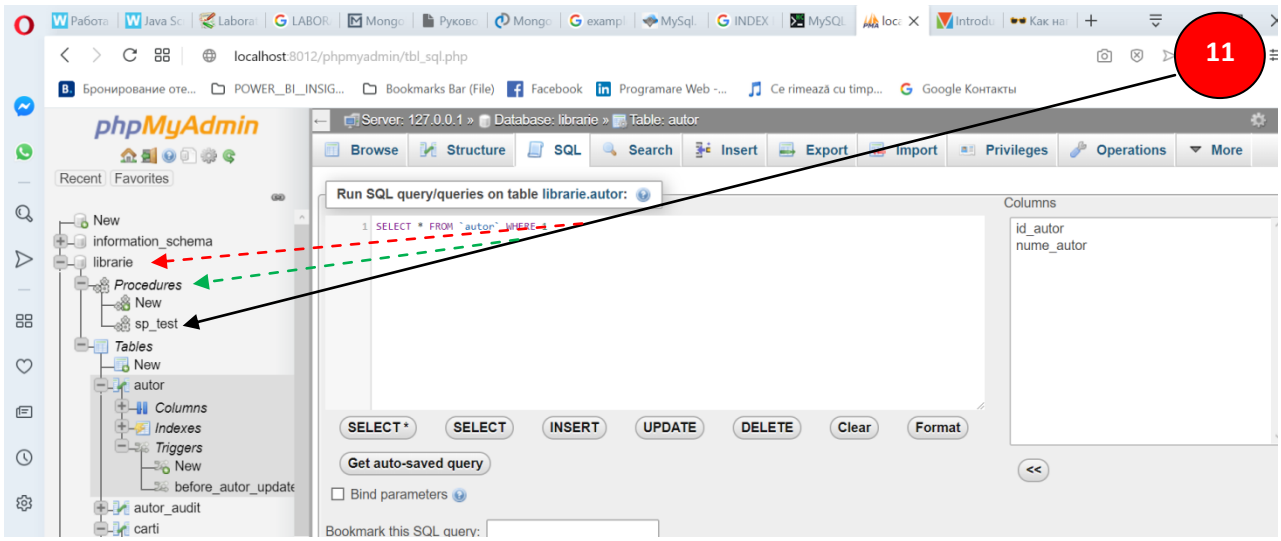

**Dupa creare aceasta procedura apare in Baza de date** Tastăm tasta F5, Refresh

**Sau**




Sau tastam si obtinem

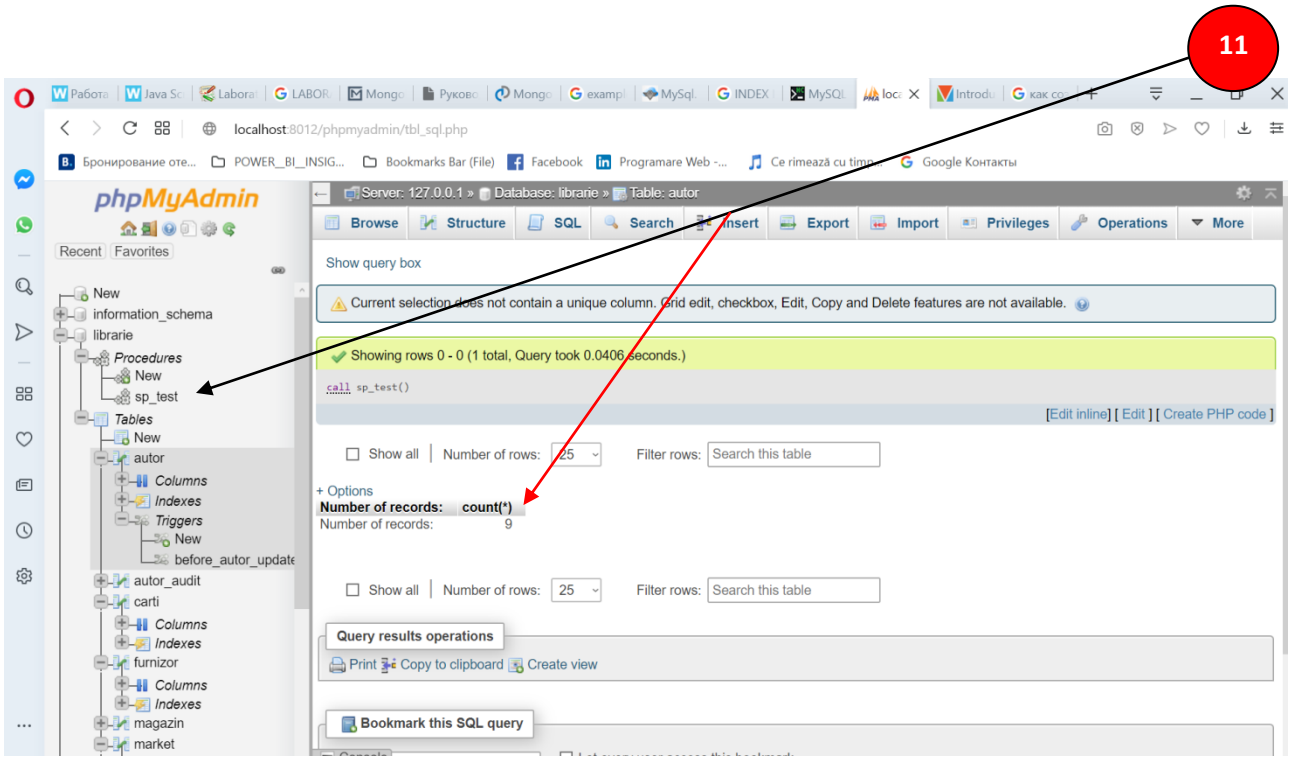



**Acum urmeaza sa utilizam procedura stocata in lucru**
**Lansând-o cu comanda Execute din punctual de meniu Routines sau in meniul SQL culegem**

**Call sp_test();**

**In rezultat vom obtine:**




## Noțiuni introductive cu procedurile stocate:

***Prima dată când invocăm (apelăm) o procedură stocată, MySQL*** /MariaDB, caută numele din ***catalogul de baze de date***, compilează codul procedurii stocate, îl așează într-o zonă de memorie cunoscută sub numele de ***cache*** și apoi se execută procedura stocată.

***Dacă invocăm (apelăm) din nou aceeași procedură stocată în aceeași sesiune***, MySQL execută doar procedura stocată din cache fără a fi nevoie să o recompilăm.

***O procedură stocată poate avea parametri***, astfel încât să îi putem transfera valori și să obțineți rezultatul înapoi.

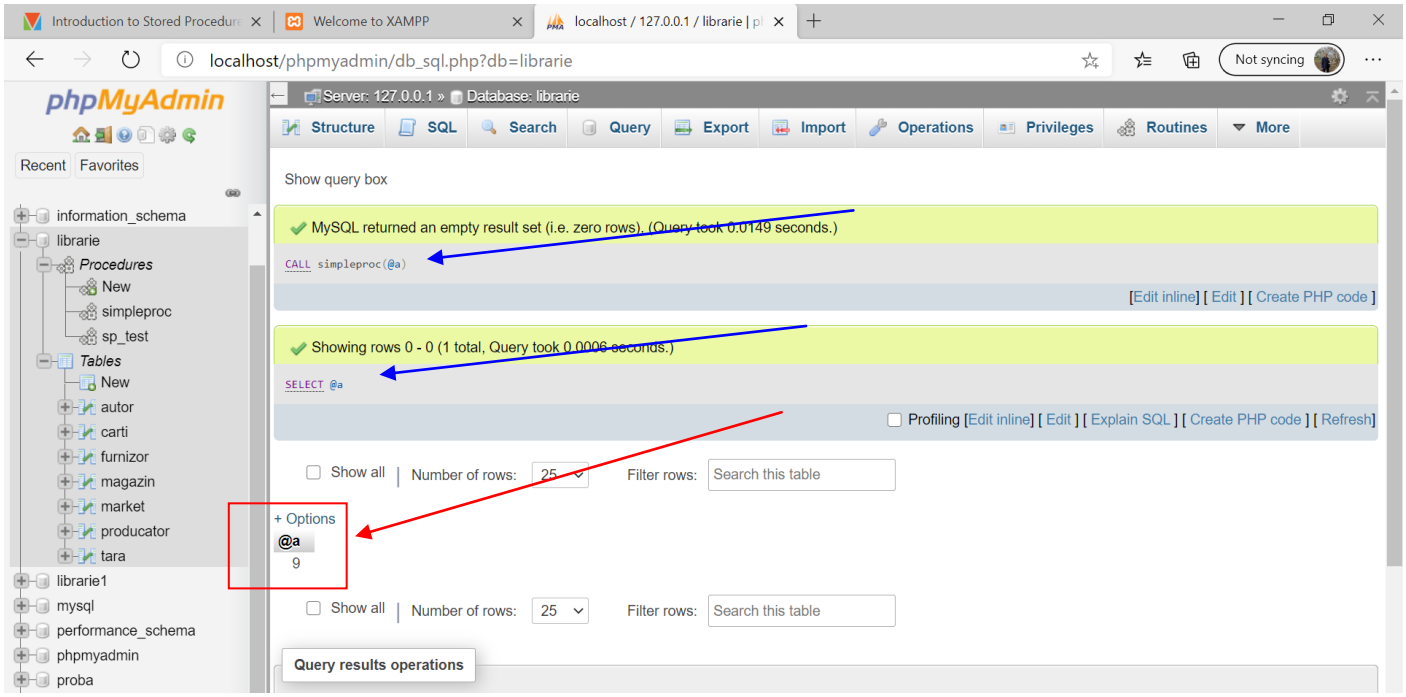
De exemplu, putem avea o procedură stocată care returnează clienții după țară și oraș. În acest caz, țara și orașul sunt parametrii procedurii stocate.

Ori in cazul nostrum, pentru BD librarie si tabelul autor vom avea:

```
DELIMITER //
CREATE PROCEDURE simpleproc (OUT param1 INT)
 BEGIN
  SELECT COUNT(*) INTO param1 FROM autor;
 END//
DELIMITER ;
```

## Adresarea la procedura stocată

```
CALL simpleproc(@a);
Select @a;
```




***O procedură stocată poate conține declarații de flux de control***, cum ar fi **IF , CASE și LOOP** care ne-ar permite să implementăm codul în mod procedural.

O procedură stocată poate apela alte proceduri stocate sau ***funcții stocate*** , ceea ce ne permite să modelam codul după necesități.

Să reținem că vom afla pas cu pas în tutorialul ce rumează cum să creăm o nouă procedură stocată

**Mai multe detalii despre sintaxa CREATE PROCEDURE - vezi ANEXA 1**

**ALTE EXEMPLE! VEZI ANEXA 2**

## Avantajele procedurilor MySQL/MariaDB stocate

**Procedurilor stocate au următoarele avantaje:**

***Reducem traficul de rețea***

Procedurile stocate ajută la reducerea traficului de rețea între aplicații și MySQL Server. Deoarece în loc să trimită mai multe declarații SQL de lungă durată, aplicațiile trebuie să trimită doar numele și parametrii procedurilor stocate.

***Centralizăm logica de afaceri în baza de date***

Putem utiliza procedurile stocate pentru a implementa logica de afaceri care este reutilizabilă de mai multe aplicații. Procedurile stocate ajută la reducerea eforturilor de duplicare a aceleiași logici în multe aplicații și fac baza de date mai consistentă.

***Creează baza de date mai sigură***

Administratorul bazei de date poate acorda privilegii adecvate aplicațiilor care accesează numai proceduri stocate specifice fără a acorda privilegii pe tabelele de bază.

**Dezavantajele procedurilor MySQL stocate**

**Pe lângă aceste avantaje, procedurile stocate prezintă și dezavantaje:**

***Utilizarea resurselor***

Dacă utilizam multe proceduri stocate, ***utilizarea memoriei*** fiecărei conexiuni va crește substanțial.

În plus, utilizarea excesivă a unui număr mare de operații logice în procedurile stocate va crește ***utilizarea procesorului***, deoarece MySQL nu este bine conceput pentru operații logice.

***Depanare***

Este dificil să depanați procedurile stocate. Din păcate, MySQL nu oferă nicio facilitate pentru a depana procedurile stocate, cum ar fi alte produse ale bazelor de date, cum ar fi ***Oracle și SQL Server***.

***Mentenanta***

Dezvoltarea și menținerea procedurilor stocate necesită adesea un set specializat de abilități pe care nu toți dezvoltatorii de aplicații îl dețin. Acest lucru poate duce la probleme atât în dezvoltarea aplicațiilor, cât și în întreținere.

## Atentie!! MySQL Delimiter

Când scriem instrucțiuni SQL, utilizați punct și virgulă **(;)** pentru a separa două instrucțiuni precum următorul exemplu:

```
SELECT * FROM products;
SELECT * FROM customers;
```

Un program client MySQL, folosește delimitatorul **(;)** pentru a separa instrucțiunile și a executa fiecare instrucțiune separat.

**O procedură stocată, însă, constă din mai multe enunțuri separate printr-un punct și virgulă (;).**

**Este evident, că dacă utilizam un program client MySQL pentru a defini o procedură stocată care conține caractere punct și virgulă, programul client MySQL nu va trata întreaga procedură stocată ca o singură instrucțiune, ci mai multe declarații.**

Prin urmare**, TREBUIE SĂ REDEFINIM DELIMITATORUL TEMPORAR** pentru a putea trece întreaga procedură stocată pe server ca o singură instrucțiune.

**Pentru a redefini delimitatorul implicit, utilizați comanda DELIMITER :**

```
DELIMITER delimiter_character
```

**delimiter_character poate consta dintr-un singur caracter sau mai multe caractere, de exemplu, // sau $$.**

**Notă:** Cu toate acestea, ar trebui să evităm utilizarea backslash-ului **( \ ),** deoarece acesta este caracterul de **Escape** din MySQL.

De exemplu, această afirmație

**DELIMITER //**

schimbă delimitatorul în **//:**

Odată schimbat delimitatorul, puteți utiliza noul delimiter pentru a încheia o declarație după cum urmează:

```
DELIMITER //
 SELECT * FROM customers //
 SELECT * FROM products //
```

Pentru a schimba delimitatorul înapoi în punct și virgulă, utilizați această afirmație:

**DELIMITER ;**

## Utilizarea MySQL DELIMITER pentru procedurile stocate

**O procedură stocată conține de obicei mai multe instrucțiuni separate prin punct și virgulă (;).** Pentru a utiliza compilați întreaga procedură stocată ca o singură instrucțiune compusă, trebuie să schimbați temporar delimitatorul de la punct și virgulă (;) la alte delimitare, cum ar fi **$$ sau // : de exemplu**

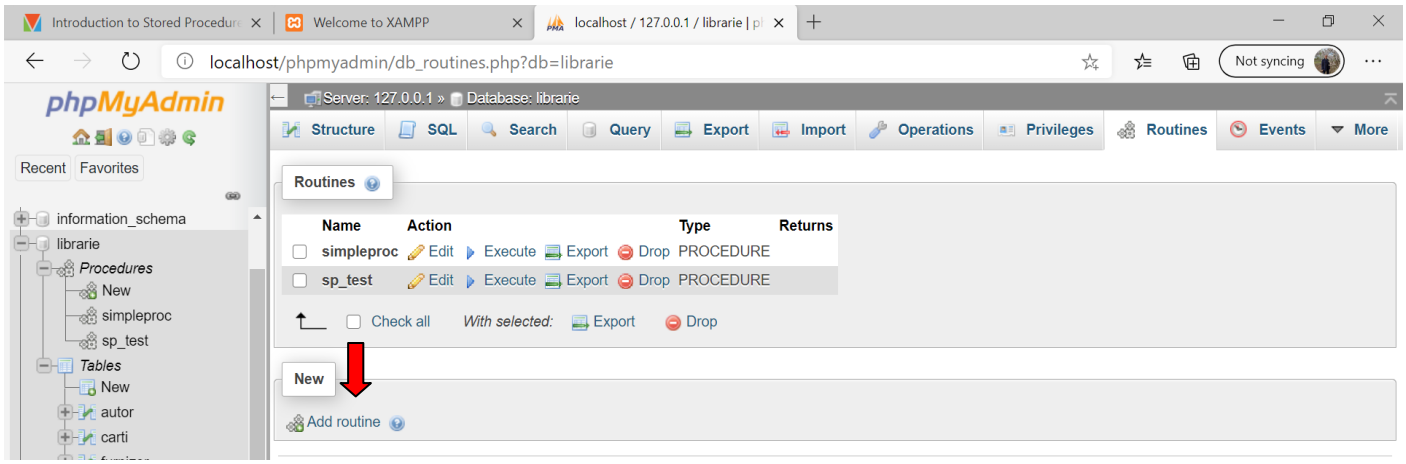
```
DELIMITER $$
CREATE PROCEDURE sp_name()
BEGIN
  -- statements
END $$
DELIMITER ;
```

← **revenim la delimitatorul prin Default!!!**

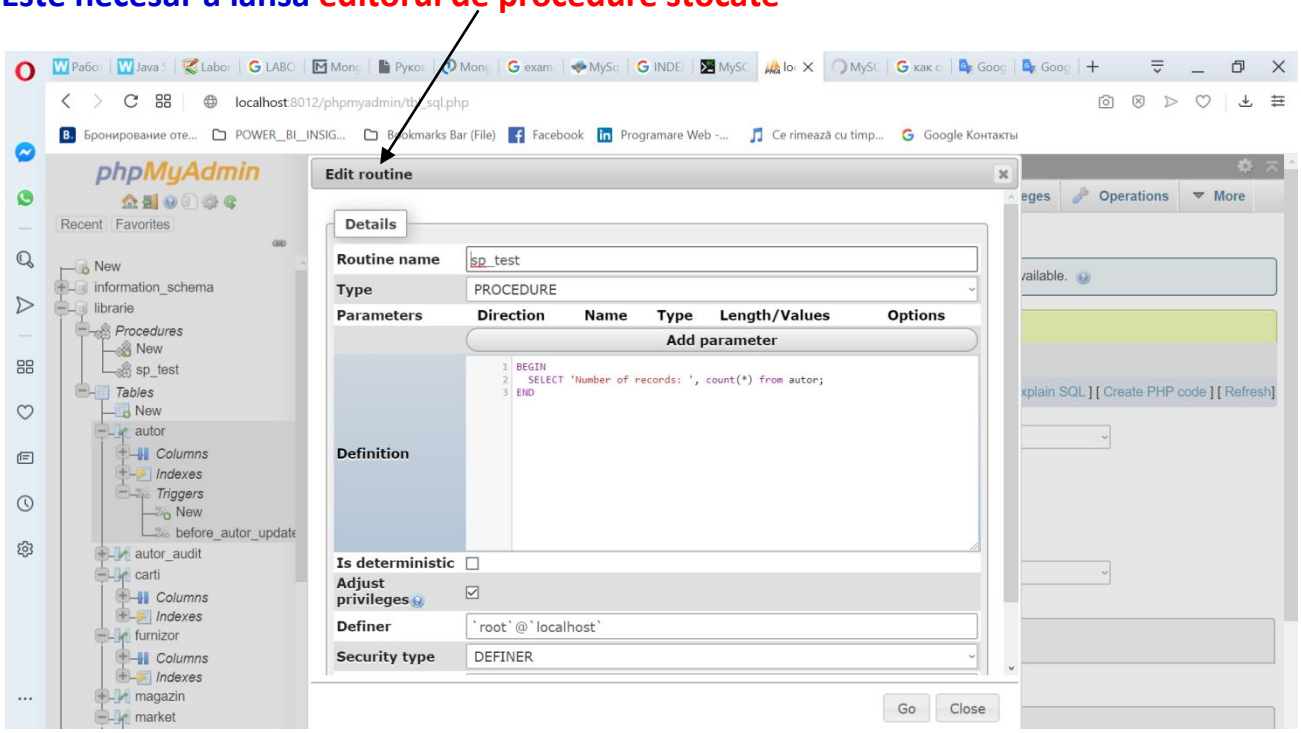
## Algoritmul utilizarii delimitatoarelor:

1. Mai întâi, schimbați delimitatorul implicit în **$$, sau //**
2. În al doilea rând, utilizați (;) în corpul procedurii stocate și $$ după cuvântul cheie END pentru a încheia procedura stocată.
3. În al treilea rând, schimbați delimitatorul implicit înapoi în punct și virgulă (;)

## MySQL CREATE PROCEDURE



**Este necesar a lansa editorul de procedure stocate**




**Notă:**

Instrumentele mai avansate cum ar fi **MS SQL Server, VS, WorkBench** si altele au instrumente mai moderne de editare a procedurilor stocate.

## MySQL DROP PROCEDURE

1. Se creeaza procedura stocată

***DELIMITER $$***

CREATE PROCEDURE ***Getautor()***

***BEGIN***

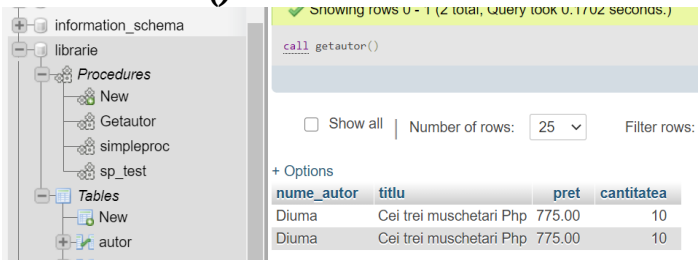
SELECT A.nume_autor, B.titlu, B.pret, B.cantitatea FROM autor A INNER JOIN carti B ON A.id_autor= B.id_autor Where B.autor='Diuma';

***END $$***

***DELIMITER ;***

**Lansarea procedurii este simpla**

***Call Getautor()***



**Folosim DROP PROCEDURE** pentru a distruge/delete procedura stocată ***GetEmployees()*** / sau ***Call Getautor()***

2.

```
DROP PROCEDURE GetEmployees;
```

3. Utilizati MySQL **DROP PROCEDURE** cu **IF EXISTS** , fiindca in cazul ca ea nu exista veti avea erori

```
DROP PROCEDURE abc;
```

MySQL issued the following error:

**Error Code: 1305.** PROCEDURE classicmodels.abc does not exist

Utilizînd optiunea **IF EXISTS** :

```
DROP PROCEDURE IF EXISTS abc;
```

Obtinem

0 row(s) affected, 1 warning(s): 1305 PROCEDURE classicmodels.abc does not exist

Dacă utilizăm SHOW WARNINGS

```
SHOW WARNINGS;
```

Vom optine

| | Level | Code | Message |
|---|---|---|---|
| ▶ | Note | 1305 | PROCEDURE classicmodels.abc does not exist |

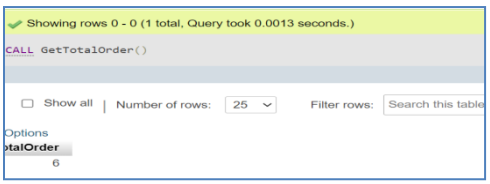
## MYSQL STORED PROCEDURE VARIABLES

```
DELIMITER $$
CREATE PROCEDURE GetTotalOrder()
BEGIN
   DECLARE totalOrder INT DEFAULT 0;
     SELECT COUNT(*)
   INTO totalOrder
   FROM carti;
   SELECT totalOrder;
END$$
DELIMITER ;
```

**This statement calls the stored procedure GetTotalOrder():**

```
CALL GetTotalOrder();
```

Iată rezultatul:



# PARAMETRII PROCEDURII STOCATE MYSQL

***Procedurile stocate*** pe care le dezvoltăm necesită parametri. Parametrii fac ca procedura stocată să fie mai flexibilă și mai utilă.

***În MySQL, un parametru are unul dintre cele trei moduri:*** **IN, OUT sau INOUT .**

## *Parametrii IN*

***IN este modul implicit***. Când definiți un parametru ***IN*** într-o procedură stocată, programul apelant trebuie să treacă un argument procedurii stocate. În plus, valoarea unui parametru ***IN*** este protejată. Înseamnă că chiar și valoarea parametrului ***IN*** este modificată în cadrul procedurii stocate, valoarea inițială este păstrată după încheierea procedurii stocate. Cu alte cuvinte, procedura stocată funcționează numai la copia parametrului ***IN*** .

## *Parametri OUT*

Valoarea unui parametru OUT poate fi modificată în cadrul procedurii stocate, iar noua sa valoare este transmisă înapoi la programul apelant. Observați că procedura stocată nu poate accesa valoarea inițială a parametrului OUT atunci când începe lucrul.

## *Parametrii INOUT*

Un parametru INOUT este o combinație de parametri IN și OUT . Înseamnă că programul apelant poate transmite argumentul, iar procedura stocată poate modifica parametrul INOUT și trece noua valoare înapoi la programul apelant.

## *Definirea unui parametru*

Iată sintaxa de bază a definirii unui parametru în procedurile stocate :

**[ IN | OUT | INOUT ] parameter_name datatype[( length )]**

În această sintaxă,

- Mai întâi, specificăm modul parametrului, care poate fi IN , OUT sau INOUT , în funcție de scopul parametrului din procedura stocată.
- În al doilea rând, specificați numele parametrului. Numele parametrului trebuie să respecte regulile de denumire a numelui de coloană din MySQL.
- În al treilea rând, specificați tipul de date și lungimea maximă a parametrului.

## *Exemple de parametri ai procedurii stocate MySQL*

Să luăm câteva exemple de utilizare a parametrilor procedurii stocate.

### Exemplul parametrului IN

Următorul exemplu creează o procedură stocată care găsește toate birourile care se localizează într-o țară specificată de parametrul de introducere ***countryName*** :

```
DELIMITER //
 CREATE PROCEDURE GetOfficeByCountry(IN  countryName  VARCHAR (255)
)
BEGIN
    SELECT *
    FROM offices
    WHERE country = countryName;
```

```
END //
 DELIMITER ;
```

În acest exemplu, ***countryName*** este parametrul IN al procedurii stocate.

Să presupunem că doriți să găsiți birouri localizate în SUA, trebuie să treceți un argument ( USA ) la procedura stocată, așa cum se arată în următoarea interogare:

```
CALL GetOfficeByCountry( 'USA' );
```

| officeCode | city | phone | addressLine1 | addressLine2 | state | country | postalCode | territory |
|---|---|---|---|---|---|---|---|---|
| 1 | San Francisco | +1 650 219 4782 | 100 Market Street | Suite 300 | CA | USA | 94080 | NA |
| 2 | Boston | +1 215 837 0825 | 1550 Court Place | Suite 102 | MA | USA | 02107 | NA |
| 3 | NYC | +1 212 555 3000 | 523 East 53rd Street | apt. 5A | NY | USA | 10022 | NA |

Pentru a găsi birouri în France , treceți șirul literal France la procedura stocată GetOfficeByCountry după cum urmează:

```
CALL GetOfficeByCountry( 'France' )
```

| officeCode | city | phone | addressLine1 | addressLine2 | state | country | postalCode | territory |
|---|---|---|---|---|---|---|---|---|
| 4 | Paris | +33 14 723 4404 | 43 Rue Jouffroy D'abbans | NULL | NULL | France | 75017 | EMEA |

Deoarece ***countryName*** este parametrul ***IN***, trebuie să treceți un argument. Dacă nu faceți acest lucru, va rezulta o eroare:

```
CALL GetOfficeByCountry();
```

Iată eroarea:

```
Error Code: 1318. Incorrect number of arguments for PROCEDURE
classicmodels.GetOfficeByCountry; expected 1, got 0
```

**Exemplul parametrului OUT**

Următoarea procedură stocată returnează nu***mărul de comenzi*** după starea comenzii.

```
DELIMITER $$
CREATE PROCEDURE GetOrderCountByStatus ( IN   orderStatus VARCHAR (25),
OUT total INT)
BEGIN
    SELECT COUNT (orderNumber)
    INTO total
    FROM orders
    WHERE status = orderStatus;
END $$
DELIMITER ;
```

***Procedura stocată GetOrderCountByStatus() are doi parametri:***

- ***orderStatus*** : este parametrul **IN** specifică starea comenzilor de returnare.
- ***total*** : este parametrul **OUT** care stochează numărul de comenzi într-o stare specifică.

Pentru a afla numărul de comenzi care au fost deja expediate, apelati la ***GetOrderCountByStatus*** și treceți si ***vedeti starea comenzii Shipped*** și, de asemenea, treceți o ***variabilă de sesiune*** ( **@total** ) pentru a primi valoarea returnată.

```
CALL GetOrderCountByStatus( 'Shipped' ,@total);
SELECT @total;
```

| | @total |
|---|---|
| ▶ | 303 |

Pentru a obține ***numărul de comenzi care sunt în proces***, apelați procedura stocată GetOrderCountByStatus după cum urmează:

```
CALL GetOrderCountByStatus( 'in process' ,@total);
SELECT @total AS   total_in_process;
```

| | total_in_process |
|---|---|
| ▶ | 6 |

**Exemplul parametrului INOUT**

Exemplul următor demonstrează modul de utilizare a unui parametru **INOUT** în procedura stocată.

```
DELIMITER $$
CREATE PROCEDURE SetCounter( INOUT counter INT , IN inc INT)
BEGIN
    SET counter = counter + inc;
END $$
DELIMITER ;
```

În acest exemplu, procedura stocată ***SetCounter()*** acceptă un parametru **INOUT** ( ***counter*** ) și un parametru **IN** ( ***inc*** ).

Crește contorul ( counter ) cu valoarea specificată de parametrul inc .

Următoarele  declarații ilustrează modul de apelare a procedurii stocate ***SetSounter*** :

```
SET @counter = 1;
CALL SetCounter(@counter,1);      -- 2
CALL SetCounter(@counter,1);      -- 3
CALL SetCounter(@counter,5);      -- 8
SELECT @counter;                  -- 8
```

**Iată rezultatul:**

| | @counter |
|---|---|
| ▶ | 8 |

# II. FUNCTII STOCATE IN MYSQL

O funcție stocată este un program memorat special care returnează o singură valoare. În mod obișnuit, utilizați funcții stocate pentru a încapsula formule comune sau reguli de afaceri care pot fi reutilizabile printre instrucțiunile SQL sau programele stocate.

**Diferență de o procedură stocată, putem utiliza o funcție stocată în instrucțiuni SQL oriunde este utilizată o expresie**. Acest lucru ajută la îmbunătățirea lizibilității /citibil/ și a mentenabilității codului procedural.

Pentru a crea o funcție stocată, utilizați instrucțiunea CREATE FUNCTION.

Sintaxa MySQL **CREATE FUNCTION**

**SINTAXA DE BAZĂ PENTRU CREAREA UNEI NOI FUNCȚII STOCATE ESTE DUPĂ CUM URMEAZĂ:**

```
DELIMITER $$
CREATE FUNCTION function_name (
   param1,
   param2,…
)
RETURNS datatype
[NOT] DETERMINISTIC
BEGIN
-- statements
END $$
DELIMITER ;
```

**În această sintaxă:**

1. **Mai întâi**, specificați numele funcției stocate pe care dorim să o cream după cuvintele cheie CREATE FUNCTION.
2. **În al doilea rând,** enumerăm toți parametrii funcției stocate în paranteze urmată de numele funcției. În mod implicit, toți parametrii sunt parametrii IN. ***Nu puteți specifica modificatorii IN, OUT sau INOUT la parametric***
3. **În al treilea rând**, specificăm tipul de date al valorii de returnare din instrucțiunea RETURNS, care poate fi orice tip de date MySQL valabil.
4. **În al patrulea rând,** specificăm dacă o funcție este deterministă sau nu utilizează cuvântul cheie DETERMINISTIC.

**O funcție DETERMINISTĂ întoarce întotdeauna același rezultat pentru aceiași parametri de intrare, în timp ce o funcție nedeterministă returnează rezultate diferite pentru aceiași parametri de intrare.**

**Dacă nu utilizați DETERMINISTIC sau NU DETERMINISTIC, MySQL utilizează opțiunea NOT DETERMINISTIC în mod implicit.**

**În al cincilea rând**, scriem codul în corpul funcției stocate în blocul **BEGIN -- END**. În secțiunea corpului, trebuie să specificați cel puțin o declarație **RETURN.** Instrucțiunea RETURN returnează o valoare programelor apelante. **Ori de câte ori se ajunge la declarația RETURN, execuția funcției stocate se încheie imediat.**

**Exemplu MySQL de creare a unei functii stocate CREATE FUNCTION**

Să luăm exemplul creării unei funcții stocate. Vom folosi tabelul clienților din baza de date de **probă** pentru demonstrație

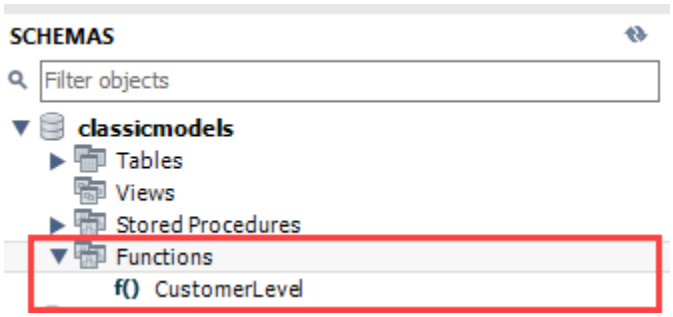
**Următoarea declarație CREATE FUNCȚIE creează o funcție care returnează nivelul clientului pe baza creditului:**

```
CREATE FUNCTION CustomerLevel (credit DECIMAL(10,2))
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
   DECLARE customerLevel VARCHAR(20);

   IF credit > 50000 THEN
      SET customerLevel = 'PLATINUM';
   ELSEIF (credit >= 50000 AND
         credit <= 10000) THEN
      SET customerLevel = 'GOLD';
   ELSEIF credit < 10000 THEN
      SET customerLevel = 'SILVER';
   END IF;
   -- return the customer level
   RETURN (customerLevel);
END$$
DELIMITER ;
```

```
DELIMITER $$
CREATE FUNCTION AutorLevel (numar_carti DECIMAL(10,2))
RETURNS VARCHAR(20)
DETERMINISTIC
BEGIN
   DECLARE AutorLevel VARCHAR(20);
   IF numar_carti > 150  THEN
      SET AutorLevel = 'PLATINUM';
   ELSEIF (numar_carti >= 150  AND
         numar_carti <= 200) THEN
      SET AutorLevel = 'GOLD';
   ELSEIF numar_carti< 100 THEN
      SET AutorLevel = 'SILVER';
   END IF;
   -- return the customer level
   RETURN (AutorLevel);
END$$
DELIMITER ;
```

**Din moment cind functia stocata este creată, ea poate fi urmărită MySQL**




Or, se poate urmări toate functiile stocate in structura curentă a BD utilizind comanda **SHOW FUNCTION STATUS**
După cum urmează:

```
SHOW FUNCTION STATUS
WHERE db = 'classicmodels';
```

| Db | Name | Type | Definer |
|---|---|---|---|
| classicmodels | CustomerLevel | FUNCTION | root@localhost |

## Lansarea unuei functii stocate intr-o interogare SQL.

Următoarea interogare foloseste functia stocată CustomerLevel:

```
SELECT
    customerName,
    CustomerLevel(creditLimit)
FROM
    customers
ORDER BY
    customerName;
```

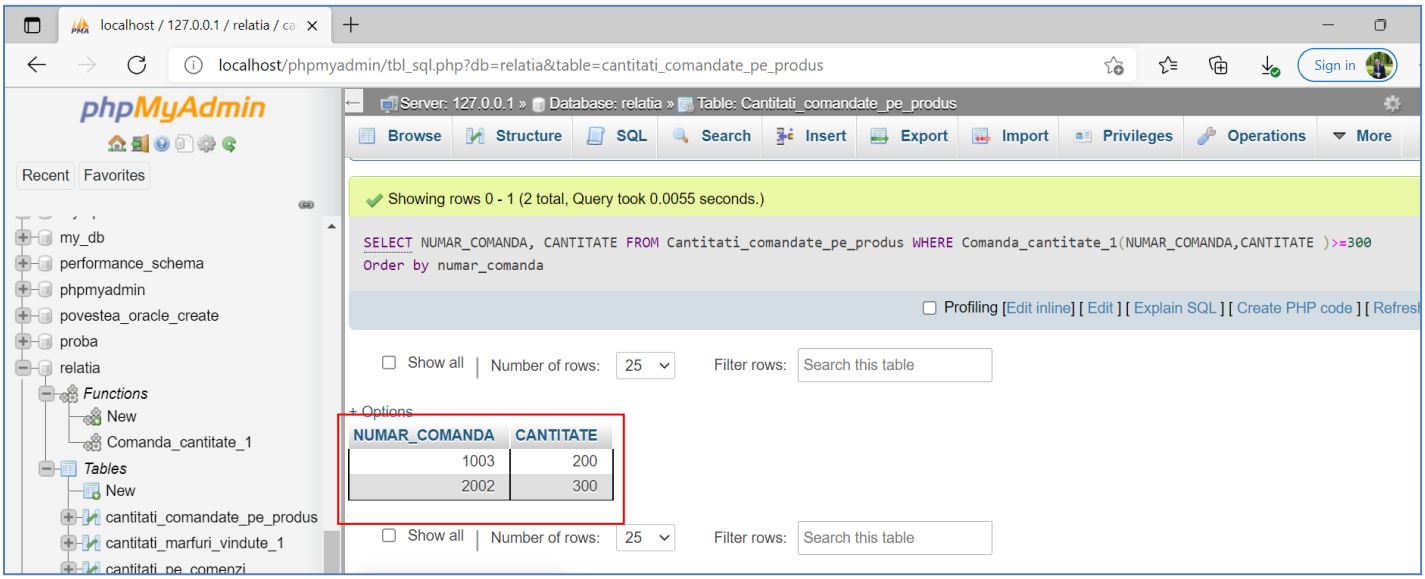
| customerName | CustomerLevel(creditLimit) |
|---|---|
| Alpha Cognac | PLATINUM |
| American Souvenirs Inc | SILVER |
| Amica Models & Co. | PLATINUM |
| ANG Resellers | SILVER |
| Anna's Decorations, Ltd | PLATINUM |
| Anton Designs, Ltd. | SILVER |
| Asian Shopping Network, Co | SILVER |
| Asian Treasures, Inc. | SILVER |
| Atelier graphique | GOLD |
| Australian Collectables, Ltd | PLATINUM |
| Australian Collectors, Co. | PLATINUM |

```
SELECT
   autor,
   AutorLevel(pret), pret as Pretul
FROM
   carti
ORDER BY
   autor;
```

EXEMPLU DIN BD RELATIA

```
DELIMITER //
CREATE FUNCTION  Comanda_cantitate_1  (NUMAR_COMANDA DECIMAL(10,2),
CANTITATE DECIMAL (10,2))
RETURNS DECIMAL(10,4)
NOT DETERMINISTIC
BEGIN
   DECLARE Comanda_cantitate_1 DECIMAL(10,2);
    IF NUMAR_COMANDA=1001
 THEN
      SET Comanda_cantitate_1  = CANTITATE*2;
   ELSEIF  NUMAR_COMANDA=1003
   THEN
      SET Comanda_cantitate_1  = CANTITATE*4;
   ELSEIF NUMAR_COMANDA=2002
   THEN
      SET Comanda_cantitate_1  = 300;
   END IF;
   -- return the customer level
   RETURN (Comanda_cantitate_1 );
END//
DELIMITER ;
```

```
SELECT  NUMAR_COMANDA, CANTITATE
 FROM Cantitati_comandate_pe_produs
WHERE Comanda_cantitate_1(NUMAR_COMANDA,CANTITATE )>=300
Order by numar_comanda
```



**Lansarea unuei functii stocate intr-o procedură stocată.**

Următoarea interogare ***creează o nouă procedură stocată*** / creates a new stored procedure care lansează functia stocată **CustomerLevel()** :

```
DELIMITER $$
CREATE PROCEDURE
GetCustomerLevel(
    IN  customerNo INT,
    OUT customerLevel VARCHAR(20)
)
BEGIN
  DECLARE credit DEC(10,2)
     DEFAULT 0;
    -- get credit limit of a customer
    SELECT
       creditLimit
    INTO credit
    FROM customers
    WHERE
       customerNumber = customerNo;
       -- call the function
    SET customerLevel =
CustomerLevel(credit);
END$$
DELIMITER ;
```

Următoarea comandă ilustrează cum se adresează procedura stocată how to call the **GetCustomerLevel()** :

```
CALL GetCustomerLevel(- 1,@customerLevel);
SELECT @customerLevel;
```

Este important să observați că, dacă o funcție stocată conține instrucțiuni SQL care interogează datele din tabele, nu ar trebui să le utilizați în alte instrucțiuni SQL; în caz contrar, funcția stocată va încetini viteza interogării.

În acest tutorial, ați învățat cum să creați o funcție stocată pentru a încapsula formula comună sau regulile de afaceri.

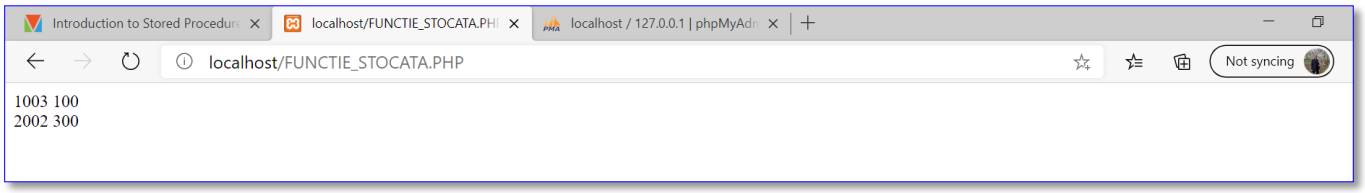
**Tutoriale conexe**

MySQL Stored Procedures That Return Multiple Values

## UTILIZAREA FUNCTIILOR STOCATE IN PHP SCRIPT

```
<?php
// enable error reporting
SESSION_START();
```

```
//mysqli_report(MYSQLI_REPORT_ERROR | MYSQLI_REPORT_STRICT);
//connect to database
$connect = mysqli_connect('localhost', 'root', '', 'relatia');
//run the store proc
//$NUMAR_COMANDA= Cantitati_comandate_pe_produs.NUMAR_COMANDA;
//$CANTITATE= Cantitati_comandate_pe_produs.CANTITATE;
$result = mysqli_query($connect, "select * from
Cantitati_comandate_pe_produs where
Comanda_cantitate_1(Cantitati_comandate_pe_produs.NUMAR_COMANDA,Cantit
ati_comandate_pe_produs.CANTITATE)>=300");
//loop the result set
while ($row = mysqli_fetch_row($result))
{
echo $row[0].'  '.+$row[1]."<br/>";
}
mysqli_close($connect);
?>
```




**ANEXA 1**

**Sintaxa CREATE PROCEDURE**

```
CREATE
   [OR REPLACE]
   [DEFINER = { user | CURRENT_USER | role | CURRENT_ROLE }]
   PROCEDURE sp_name ([proc_parameter[,...]])
   [characteristic ...] routine_body

proc_parameter:
   [ IN | OUT | INOUT ] param_name type

type:
   Any valid MariaDB data type

characteristic:
   LANGUAGE SQL
  | [NOT] DETERMINISTIC
  | { CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES SQL DATA }
```

**| SQL SECURITY { DEFINER | INVOKER }**
**| COMMENT 'string'**

**routine_body:**
**Valid SQL procedure statement**

**ANEXA 2**

**Введение в хранимые процедуры MySQL 5**

**https://ruseller.com/lessons.php?id=1189**

В MySQL 5 есть много новых функций, одной из самых весомых из которых является создание хранимых процедур. В этом уроке я расскажу о том, что они из себя представляют, а также о том, как они могут облегчить вам жизнь.

**Введение**

Хранимая процедура - это способ инкапсуляции повторяющихся действий. В хранимых процедурах можно объявлять переменные, управлять потоками данных, а также применять другие техники программирования.

Причина их создания ясна и подтверждается частым использованием. С другой стороны, если вы поговорите с теми, кто работает с ними нерегулярно, то мнения разделятся на два совершенно противоположных фланга. Не забывайте об этом.

**За**

- Разделение логики с другими приложениями. Хранимые процедуры инкапсулируют функциональность; это обеспечивает связность доступа к данным и управления ими между различными приложениями.

- Изоляция пользователей от таблиц базы данных. Это позволяет давать доступ к хранимым процедурам, но не к самим данным таблиц.
- Обеспечивает механизм защиты. В соответствии с предыдущим пунктом, если вы можете получить доступ к данным только через хранимые процедуры, никто другой не сможет стереть ваши данные через команду SQL DELETE.
- Улучшение выполнения как следствие сокращения сетевого трафика. С помощью хранимых процедур множество запросов могут быть объединены.

**Против**

- Повышение нагрузки на сервер баз данных в связи с тем, что большая часть работы выполняется на серверной части, а меньшая - на клиентской.
- Придется много чего подучить. Вам понадобится выучить синтаксис MySQL выражений для написания своих хранимых процедур.
- Вы дублируете логику своего приложения в двух местах: серверный код и код для хранимых процедур, тем самым усложняя процесс манипулирования данными.
- Миграция с одной СУБД на другую (DB2, SQL Server и др.) может привести к проблемам.

Инструмент, в котором я работаю, называется MySQL Query Browser, он достаточно стандартен для взаимодействия с базами данных. Инструмент командной строки MySQL - это еще один превосходный выбор. Я рассказываю вам об этом по той причине, что всеми любимый phpMyAdmin не поддерживает выполнение хранимых процедур.

Кстати, я использую элементарную структуру таблиц, чтобы вам было легче разобраться в этой теме. Я ведь рассказываю о хранимых процедурах, а они достаточно сложны, чтобы вникать еще и в громоздкую структуру таблиц.

**Шаг 1: Ставим ограничитель**

Ограничитель - это символ или строка символов, который используется для указания клиенту MySQL, что вы завершили написание выражения SQL. Целую вечность ограничителем был символ точки с запятой. Тем не менее, могут возникнуть проблемы, так как в хранимой процедуре может быть несколько выражений, каждое из которых должно заканчиваться точкой с запятой. В этом уроке я использую строку “//” в качестве ограничителя.

**Шаг 2: Как работать с хранимыми процедурами**

**Создание хранимой процедуры**

```
01 DELIMITER //
02
03 CREATE PROCEDURE `p2` ()
04 LANGUAGE SQL
05 DETERMINISTIC
06 SQL SECURITY DEFINER
07 COMMENT 'A procedure'
08 BEGIN
09     SELECT 'Hello World !';
10 END//
```

Первая часть кода создает хранимую процедуру. Следующая - содержит необязательные параметры. Затем идет название и, наконец, тело самой процедуры.

Названия хранимых процедур чувствительны к регистру. Вам также нельзя создавать несколько процедур с одинаковым названием. Внутри хранимой процедуры не может быть выражений, изменяющих саму базу данных.

4 характеристики хранимой процедуры:

- Language: в целях обеспечения переносимости, по умолчанию указан SQL.
- Deterministic: если процедура все время возвращает один и тот же результат, и принимает одни и те же входящие параметры. Это для репликации и процесса регистрации. Значение по умолчанию - NOT DETERMINISTIC.
- SQL Security: во время вызова идет проверка прав пользователя. INVOKER - это пользователь, вызывающий хранимую процедуру. DEFINER - это “создатель” процедуры. Значение по умолчанию - DEFINER.
- Comment: в целях документирования, значение по умолчанию - ""

**Вызов хранимой процедуры**

Чтобы вызвать хранимую процедуру, необходимо напечатать ключевое слово CALL, а затем название процедуры, а в скобках указать параметры (переменные или значения). Скобки обязательны.

```
1 CALL stored_procedure_name (param1, param2, ....)
2
3 CALL procedure1(10 , 'string parameter' , @parameter_var);
```

**Изменение хранимой процедуры**

В MySQL есть выражение ALTER PROCEDURE для изменения процедур, но оно подходит для изменения лишь некоторых характеристик. Если вам нужно изменить параметры или тело процедуры, вам следует удалить и создать ее заново.

**Удаление хранимой процедуры**

```
1 DROP PROCEDURE IF EXISTS p2;
```

Это простая команда. Выражение IF EXISTS отлавливает ошибку в случае, если такой процедуры не существует.

**Шаг 3: Параметры**

Давайте посмотрим, как можно передавать в хранимую процедуру параметры.

- CREATE PROCEDURE proc1 (): пустой список параметров
- CREATE PROCEDURE proc1 (IN varname DATA-TYPE): один входящий параметр. Слово IN необязательно, потому что параметры по умолчанию - IN (входящие).
- CREATE PROCEDURE proc1 (OUT varname DATA-TYPE): один возвращаемый параметр.
- CREATE PROCEDURE proc1 (INOUT varname DATA-TYPE): один параметр, одновременно входящий и возвращаемый.

Естественно, вы можете задавать несколько параметров разных типов.1

**Пример параметра IN**

```
1   DELIMITER //
2
3 CREATE PROCEDURE `proc_IN` (IN var1 INT)
4 BEGIN
5    SELECT var1 + 2 AS result;
6 END//
```

**Пример параметра OUT**

```
1 DELIMITER //
2
3 CREATE PROCEDURE `proc_OUT` (OUT var1 VARCHAR(100))
4 BEGIN
5     SET var1 = 'This is a test';
6 END //
```

**Пример параметра INOUT**

```
1 DELIMITER //
2
3 CREATE PROCEDURE `proc_INOUT` (OUT var1 INT)
4 BEGIN
5     SET var1 = var1 * 2;
6 END //
```

### Шаг 4: Переменные

Сейчас я научу вас создавать переменные и сохранять их внутри процедур. Вы должны объявлять их явно в начале блока BEGIN/END, вместе с их типами данных. Как только вы объявили переменную, вы можете использовать ее там же, где переменные сессии, литералы или имена колонок.
Синтаксис объявления переменной выглядит так:

```
1 DECLARE varname DATA-TYPE DEFAULT defaultvalue;
```

Давайте объявим несколько переменных:

```
1 DECLARE a, b INT DEFAULT 5;
2
3 DECLARE str VARCHAR(50);
4
5 DECLARE today TIMESTAMP DEFAULT CURRENT_DATE;
6
7 DECLARE v1, v2, v3 TINYINT;
```

**Работа с переменными**

Как только вы объявили переменную, вы можете задать ей значение с помощью команд SET или SELECT:

```
01 DELIMITER //
02
03 CREATE PROCEDURE `var_proc` (IN paramstr VARCHAR(20))
04 BEGIN
05     DECLARE a, b INT DEFAULT 5;
06     DECLARE str VARCHAR(50);
07     DECLARE today TIMESTAMP DEFAULT CURRENT_DATE;
08     DECLARE v1, v2, v3 TINYINT;
09
```

```
10    INSERT INTO table1 VALUES (a);
11    SET str = 'I am a string';
12    SELECT CONCAT(str,paramstr), today FROM table2 WHERE b >=5;
13 END //
```

### Шаг 5: Структуры управления потоками

MySQL поддерживает конструкции IF, CASE, ITERATE, LEAVE LOOP, WHILE и REPEAT для управления потоками в пределах хранимой процедуры. Мы рассмотрим, как использовать IF, CASE и WHILE, так как они наиболее часто используются.2

**Конструкция IF**

С помощью конструкции IF, мы можем выполнять задачи, содержащие условия:

```
01 DELIMITER //
02
03 CREATE PROCEDURE `proc_IF` (IN param1 INT)
04 BEGIN
05    DECLARE variable1 INT;
06    SET variable1 = param1 + 1;
07
08    IF variable1 = 0 THEN
09       SELECT variable1;
10    END IF;
11
12    IF param1 = 0 THEN
13       SELECT 'Parameter value = 0';
14    ELSE
15       SELECT 'Parameter value <> 0';
16    END IF;
17 END //
```

**Конструкция CASE**

CASE - это еще один метод проверки условий и выбора подходящего решения. Это отличный способ замены множества конструкций IF. Конструкцию можно описать двумя способами, предоставляя гибкость в управлении множеством условных выражений.

```
01 DELIMITER //
02
03 CREATE PROCEDURE `proc_CASE` (IN param1 INT)
04 BEGIN
05    DECLARE variable1 INT;
06    SET variable1 = param1 + 1;
07
08    CASE variable1
```

```
09      WHEN 0 THEN
10        INSERT INTO table1 VALUES (param1);
11      WHEN 1 THEN
12        INSERT INTO table1 VALUES (variable1);
13      ELSE
14        INSERT INTO table1 VALUES (99);
15    END CASE;
16
17 END //
```

или:

```
01 DELIMITER //
02
03 CREATE PROCEDURE `proc_CASE` (IN param1 INT)
04 BEGIN
05    DECLARE variable1 INT;
06    SET variable1 = param1 + 1;
07
08    CASE
09      WHEN variable1 = 0 THEN
10        INSERT INTO table1 VALUES (param1);
11      WHEN variable1 = 1 THEN
12        INSERT INTO table1 VALUES (variable1);
13      ELSE
14        INSERT INTO table1 VALUES (99);
15    END CASE;
16
17 END //
```

**Конструкция WHILE**

Технически, существует три вида циклов: цикл WHILE, цикл LOOP и цикл REPEAT. Вы также можете организовать цикл с помощью техники программирования “Дарта Вейдера”: выражения GOTO. Вот пример цикла:

```
01 DELIMITER //
02
03 CREATE PROCEDURE `proc_WHILE` (IN param1 INT)
04 BEGIN
05    DECLARE variable1, variable2 INT;
06    SET variable1 = 0;
07
```

```
08    WHILE variable1 < param1 DO
09       INSERT INTO table1 VALUES (param1);
10       SELECT COUNT(*) INTO variable2 FROM table1;
11       SET variable1 = variable1 + 1;
12    END WHILE;
13 END //
```

**Шаг 6: Курсоры**

Курсоры используются для прохождения по набору строк, возвращенному запросом, а также обработки каждой строки.
MySQL поддерживает курсоры в хранимых процедурах. Вот краткий синтаксис создания и использования курсора.

```
1 DECLARE cursor-name CURSOR FOR SELECT ...;      /*Объявление курсора и его заполнение */
2 DECLARE  CONTINUE HANDLER FOR NOT FOUND        /*Что делать, когда больше нет записей*/
3 OPEN cursor-name;                              /*Открыть курсор*/
4 FETCH cursor-name INTO variable [, variable];   /*Назначить значение переменной, равной текущему значению столбца*/
5 CLOSE cursor-name;                             /*Закрыть курсор*/
```

В этом примере мы проведем кое-какие простые операции с использованием курсора:

```
01 DELIMITER //
02
03 CREATE PROCEDURE `proc_CURSOR` (OUT param1 INT)
04 BEGIN
05    DECLARE a, b, c INT;
06    DECLARE cur1 CURSOR FOR SELECT col1 FROM table1;
07    DECLARE CONTINUE HANDLER FOR NOT FOUND SET b = 1;
08    OPEN cur1;
09
10    SET b = 0;
11    SET c = 0;
12
13    WHILE b = 0 DO
14       FETCH cur1 INTO a;
15       IF b = 0 THEN
16          SET c = c + a;
17    END IF;
18    END WHILE;
19
20    CLOSE cur1;
```

```
21    SET param1 = c;
22
23 END //
```

У курсоров есть три свойства, которые вам необходимо понять, чтобы избежать получения неожиданных результатов:

- Не чувствительный: открывшийся однажды курсор не будет отображать изменения в таблице, происшедшие позже. В действительности, MySQL не гарантирует то, что курсор обновится, так что не надейтесь на это.
- Доступен только для чтения: курсоры нельзя изменять.
- Без перемотки: курсор способен проходить только в одном направлении - вперед, вы не сможете пропускать строки, не выбирая их.

**Заключение**

В этом уроке я ознакомил вас с основами работы с хранимыми процедурами и с некоторыми специфическими свойствами, связанными с ней. Конечно, вам нужно будет углубить знания в таких областях, как безопасность, выражения SQL и оптимизация, прежде чем стать настоящим гуру MySQL процедур.

Вы должны подсчитать, какие преимущества даст вам использование хранимых процедур в вашем конкретном приложении, и только потом создавать лишь необходимые процедуры. В общем, я использую процедуры; по-моему, их стоит внедрять в проекты в следствие их безопасности, обслуживания кода и общего дизайна. К тому же, не забывайте, что над процедурами MySQL все еще ведется работа. Ожидайте улучшений, касающихся функциональности и улучшений. Прошу, не стесняйтесь делиться мнениями.+

Данный урок подготовлен для вас командой сайта ruseller.com
Источник урока: www.net.tutsplus.com/tutorials/an-introduction-to-stored-procedures/
Перевел: Станислав Протасевич
Урок создан: 7 Июля 2011
Просмотров: 224595