PROCEDURES FUNCTION TRIGGER

Stored Procedure

Stored Procedure é um conjunto de comandos, ao qual é atribuído um nome. Este conjunto fica armazenado no Banco de Dados e pode ser chamado a qualquer momento tanto pelo SGBD (sistema Gerenciador de Banco de Dados) quanto por um sistema que faz interface com o mesmo.

A utilização de Stored Procedures é uma técnica eficiente de executarmos operações repetitivas. Ao invés de digitar os comandos cada vez que determinada operação necessite ser executada, criamos um Stored Procedure e o chamamos.

Stored Procedure

Vantagens do uso de Stored Procedures

- A maior vantagem do uso de *stored procedures* é a redução de tráfego de dados na rede. Já que as stored procedures são executadas pelo SGBD, num servidor de banco de dados, você pode utilizá-las para mover grande parte do seu código de manipulação de dados para o servidor.

Stored Procedure

Os parâmetros são delimitados entre parênteses após o nome da procedure.

CREATE PROCEDURE nm_proced([IN|OUT|INOUT] nome_do_parâmetro tipo_do_parâmetro) ...

- Parâmetros de entrada: IN
- Parâmetros de saída: OUT
- Parâmetros entrada e saída: INOUT

Stored Procedure – Parâmetros

Os procedimentos armazenados exigem parâmetros. Os parâmetros tornam o procedimento mais flexível e útil.

No MySQL, um parâmetro possui um dos três modos: IN, OUT ou INOUT.

IN - é o modo padrão. Quando você define um parâmetro IN em um procedimento armazenado, o programa de chamada tem que passar um argumento para o procedimento armazenado. Além disso, o valor de um parâmetro IN é protegido. Isso significa que mesmo o valor do parâmetro IN é alterado dentro do procedimento armazenado, seu valor original é mantido após o término do procedimento armazenado. Em outras palavras, o procedimento armazenado funciona somente na cópia do parâmetro IN.

OUT - o valor de um parâmetro OUT pode ser alterado dentro do procedimento armazenado e seu novo valor é passado de volta para o programa de chamada. Observe que o procedimento armazenado não pode acessar o valor inicial do parâmetro OUT quando ele é iniciado.

INOUT - um parâmetro INOUT é uma combinação dos parâmetros IN e OUT. Isso significa que o programa de chamada pode passar o argumento e o procedimento armazenado pode modificar o parâmetro INOUT e passar o novo valor de volta para o programa de chamada.

Stored Procedure

Sintaxe:

A sintaxe varia de acordo com o SGBD que se deseja trabalhar. No MySQL, o Stored Procedure possui a seguinte sintaxe:

Delimiter # (mudança de delimitador)

CREATE PROCEDURE <NOME>(parâmetros...)

BEGIN

--variáveis internas;

Comandos sql (Laços, condições, DML);

END #

Delimiter ; (retorno do delimitador original)

DROP PROCEDURE | FUNCTION [IF EXISTS] sp_nome

Este comando é usado para excluir uma stored procedure ou function. Isto é, a rotina especificada é removida do servidor.

A cláusula IF EXISTS é uma extensão do MySQL. Ela previne que um erro ocorra se o procedimento ou função não existe.

Stored Procedure – Exemplo parâmetro IN

```
DELIMITER //

CREATE PROCEDURE sp_listar_livro(IN descricao VARCHAR(60))

BEGIN

   SELECT *  FROM livros

   WHERE nome like concat(descricao,'%');

END //

DELIMITER ;


call sp_listar_livro('python');
```

Stored Procedure – Exemplo parâmetro OUT

```
DELIMITER $$
CREATE PROCEDURE sp_contar_vendas_livro( IN _id int, OUT total int)
BEGIN
   SELECT count(v.id)  INTO total
   FROM vendas as v where livro_id = _id;
END $$
DELIMITER ;

call sp_contar_vendas_livro(60, @total);
select @total;
```

Stored procedure – Exemplo parâmetro INOUT

```
DELIMITER $$
CREATE PROCEDURE sp_configura_cont (INOUT cont INT(4),IN inc INT(4))
BEGIN
    SET cont = cont + inc;
END $$
DELIMITER ;

set @contador=3;
call sp_configura_cont(@contador, 5);
select @contador;
```

Stored procedure - Exemplo parâmetro INOUT

```
DELIMITER $$
CREATE PROCEDURE sp_cont_lancamentos_por_ano( OUT ano2010 INT, OUT
ano2011 INT, OUT ano2012 INT, OUT ano2013 INT)
BEGIN
select count(id) into ano2010 from livros
where Year(data_de_lancamento) = 2010;
select count(id) into ano2011 from livros
where Year(data_de_lancamento) = 2011;
select count(id) into ano2012 from livros
where Year(data_de_lancamento) = 2012;
select count(id) into ano2013 from livros
where Year(data_de_lancamento) = 2013;
END
$$DELIMITER ;

CALL sp_cont_lancamentos_por_ano(@a,@b,@c,@d);
SELECT @a,@b,@c,@d;
```

Stored procedure - Declarar variável

- Primeiro, especifica o nome da variável após a palavra-chave DECLARE. O nome da variável deve seguir as regras de nomenclatura dos nomes das colunas da tabela MySQL.
- Segundo, especifica o tipo de dados da variável e seu tamanho. Uma variável pode ter qualquer tipo de dados do MySQL, como INT, VARCHAR e DATETIME.
- Em terceiro lugar, quando declarar uma variável, seu valor inicial é NULL. Pode atribuir à variável um valor padrão usando a palavra-chave DEFAULT.
- Exemplo:

```
DECLARE total INT DEFAULT 0;
```

Stored procedure - Declarar variável

- Declarar e atribuir valor a variavel

```
DECLARE total INT DEFAULT 0;

SET total = 10;
```

- Além da instrução SET pode ser usada a instrução SELECT INTO para atribuir o resultado de uma consulta, a uma variável.

Veja o seguinte exemplo:

```
DECLARE total INT DEFAULT 0;

SELECT COUNT(*) INTO total FROM livros;
```

Stored procedure – Escopo da variável

- Uma variável tem seu próprio escopo que define sua vida útil. Declarar uma variável dentro de uma procedure, estará fora do escopo quando a instrução END da procedure atingir.
- Declarar uma variável dentro do bloco BEGIN END, ela ficará fora do escopo se o END for atingido.
- Uma variável cujo nome começa com o sinal @ é uma variável de sessão. Está disponível e acessível até a sessão terminar.

MySQL IF

```
IF EXPRESSAO THEN
  COMANDOS;
END IF;
-------------------------------
IF EXPRESSAO THEN
  COMANDO_A;
ELSE
  COMANDOS_B;
END IF;
-------------------------------
```

MySQL IF

```
IF EXPRESSAO THEN
    COMANDO_A;
  ELSEIF EXPRESSAO THEN
    COMANDO_B;
  ELSEIF EXPRESSAO THEN
    COMANDO_C;
  END IF;
```

MYSQL - CASE

```
CASE  case_expression
   WHEN when_expression_1 THEN commands
   WHEN when_expression_2 THEN commands
   ...
   ELSE commands
END CASE;
```

MYSQL - CASE

```
delimiter $$
drop procedure if exists sp_case_exemplo $$
create procedure sp_case_exemplo(parametro int(10))
Begin
  case parametro
    when 0 then SELECT 'Zero';
    when 1 then SELECT 'Um';
    when 2 then SELECT 'dois';
    else SELECT 'O número não está entre zero e dois.';
  end case;
end $$
delimiter ;
```

Stored procedures – Loop - WHILE

```
delimiter //
drop procedure if exists sp_while_exemplo //
create procedure sp_while_exemplo(parametro int(10))
begin
  while (parametro<=10) do
     select parametro;
     set parametro = parametro + 1;
   end while;
  end //
delimiter ;
```

Stored procedures – Loop - REPEAT

```
delimiter $$
drop procedure if exists sp_repeat_exemplo $$
create procedure sp_repeat_exemplo(parametro int(10))
begin
  repeat
     select parametro;
     set parametro = parametro + 1;
   until parametro >10
  end repeat;
end  $$
delimiter ;
```

Stored procedures – Loop - REPEAT

```
delimiter //
drop procedure if exists p_loop //
create procedure p_loop(parametro int(10))
begin
  loop_label: LOOP
    select parametro;
    set parametro = parametro + 1;
    if parametro >10 then
        leave loop_label;
   end if;
  end loop;
end //
delimiter ;
```

Stored Procedure

```
DELIMITER $$
DROP PROCEDURE IF EXISTS reajustar_preco $$
CREATE PROCEDURE reajustar_preco(IN _id INT)
BEGIN
    update livros set preco = preco * 1.1 where id = _id;
END $$
DELIMITER ;
```

Para realizarmos a chamada a procedure utilizar o comando CALL.

```
call reajustar_preco(5);
```

Function

- A cláusula RETURNS pode ser especificada apenas para uma FUNÇÃO, para a qual é obrigatória. Indica o tipo de retorno da função e o corpo da função deve conter uma instrução de valor RETURN.
- Uma rotina é considerada " DETERMINISTIC " se sempre produzir o mesmo resultado para os mesmos parâmetros de entrada

Function

```
Delimiter $$
create function fc_calc_area_quadrado(lado double)
  returns double DETERMINISTIC
  begin
     declare valor double;
     set valor = lado * lado;
     return valor;
   end $$
DELIMITER ;
```

Trigger

Gatilho ou trigger é um recurso de programação armazenado no banco de dados e invocados **automaticamente** na ocorrência de algum evento especificado (*insert*, *update* e *delete*).

Trigger

A criação de um Trigger envolve duas etapas:

- Um comando SQL que vai disparar o Trigger (INSERT, DELETE, UPDATE)
- A ação que o Trigger vai executar (geralmente um bloco de códigos SQL)

Vantagens do uso de Trigger SQL

TRIGGERS:

- fornecem uma maneira alternativa de verificar a integridade dos dados.
- detectar erros na lógica de negócios na camada do banco de dados.
- são muito úteis para auditar as mudanças de dados nas tabelas.

Desvantagens do uso de Trigger SQL

TRIGGERS:

- São invocados e executados de forma invisível a partir dos aplicativos clientes, portanto, é difícil descobrir o que acontece na camada do banco de dados.
- Podem aumentar a sobrecarga do servidor de banco de dados.

Trigger - Sintaxe

```
DELIMITER $$

CREATE TRIGGER  trigger_name

[BEFORE|AFTER] [INSERT|UPDATE|DELETE] ON table_name

FOR EACH ROW

BEGIN

…

END $$

DELIMITER ;
```

Trigger - Comandos

Exibir as trigger criadas:

SHOW TRIGGERS;

Apagar a trigger da tabela:

DROP TRIGGER nome_da_tabela.nome_trigger;

Os registros NEW e OLD

Triggers, são executados em conjunto com operações de inclusão e exclusão, é necessário poder acessar os registros que estão sendo incluídos ou removidos.

Isso pode ser feito através das palavras NEW e OLD.

- NEW no caso de fazer um INSERT. A palavra reservada NEW dá acesso ao novo registro.
- OLD no caso de fazer um DELETE. A palavra reservada OLD dá acesso ao registro excluído.
- OLD – NEW no caso de fazer um UPDATE. A palavra reservada OLD dá acesso ao registro antes do UPDATE e a palavra reservada NEW dá acesso ao registro após o UPDATE.

Criar uma tabela livro_log

```
CREATE TABLE LIVROS_LOG (
id int auto_increment primary key,
id_livro int not null,
nome  varchar(100) NOT NULL,
data_de_lancamento date NOT NULL,
autor_id int(11) NOT NULL,
preco decimal(10,2) NOT NULL,
data_alteracao datetime,
usuario varchar(100) ,
Acao varchar(10)
);
```

Criar a trigger – Após INSERT

```
DELIMITER $$

CREATE TRIGGER tr_livro_insert

AFTER INSERT ON livros

FOR EACH ROW BEGIN

INSERT INTO LIVROS_LOG(id_livro, nome, data_de_lancamento,
autor_id, preco, data_alteracao, usuário, acao)  VALUES(new.id,
new.nome, new.data_de_lancamento, new.autor_id,   new.preco,
now(), user(),'Insert');

END $$

DELIMITER ;
```

Criar a trigger

```
DELIMITER $$

CREATE TRIGGER tr_livro_delete

AFTER DELETE ON livros

FOR EACH ROW BEGIN

INSERT INTO LIVROS_LOG(id_livro, nome, data_de_lancamento,
autor_id, preco, data_alteracao, usuário, acao)   VALUES (old.id,
old.nome, old.data_de_lancamento, old.autor_id,   old.preco, now(),
user(), 'Delete');

END $$

DELIMITER ;
```

Criar a trigger – Após Update

```
DELIMITER $$

CREATE TRIGGER tr_livro_update

AFTER UPDATE ON livros

FOR EACH ROW BEGIN

INSERT INTO LIVROS_LOG(id_livro, nome, data_de_lancamento,   autor_id,
preco, data_alteracao, usuário, acao)   VALUES (old.id, old.nome,
old.data_de_lancamento, old.autor_id,   old.preco, now(), user(), 'Update');

INSERT INTO LIVROS_LOG(id_livro, nome, data_de_lancamento,   autor_id,
preco, data_alteracao, usuário, acao)  VALUES(new.id, new.nome,
new.data_de_lancamento, new.autor_id,   new.preco, now(),
user(),'Update');

END $$

DELIMITER ;
```

Create user - Grant

CREATE USER 'novousuario'@'localhost' IDENTIFIED BY 'password';

Conceder permissões ao usuário.

GRANT [tipo de permissão] ON [nome da base de dados].[nome da tabela] TO ‘[nome do usuário]’@'localhost’;

Exemplo:

GRANT ALL PRIVILEGES ON * . * TO 'novousuario'@'localhost';

GRANT SELECT, UPDATE, DELETE ON ‘livraria'.* TO 'novousuario'@'localhost’;

Exibir as permissões:

SHOW GRANTS FOR 'novousuario'@'localhost’;

SHOW GRANTS FOR root@localhost;

Permissões

- ALL PRIVILEGES- acesso a todo o sistema
- CREATE- permite criar novas tabelas ou bases de dados
- DROP- permite deletar tableas ou bases de dados
- DELETE- permite deletar linhas das tabelas
- INSERT- permite inserir linhas nas tabelas
- SELECT- permite utilizar o comando Select para ler bases de dados
- UPDATE- permite atualizar linhas das tabelas
- GRANT OPTION- permite conceder ou revogar privilégios de outros usuários

Revoke e Drop user

Revogar uma permissão.

REVOKE [tipo de permissão] ON [nome da base de dados].[nome da tabela] FROM ‘[nome do usuário]’@‘localhost’;

REVOKE ALL PRIVILEGES FROM ‘novousuario’@‘localhost’;

REVOKE UPDATE, DELETE ON livraria.* FROM ‘novousuario’@‘localhost’;

Apagar um usuário

DROP USER ‘novousuario’@‘localhost’;

Desafio

Criar uma procedure para criar um novo usuário no banco de dados

E atribuir ao novo usuário permissões usando o comando grant