SQL Avançado

CURSO:
Administrador de Banco de Dados

PROFESSOR:
Denival Araújo dos Santos

CARGA HORÁRIA:
20 h/a

Índice Geral

Procedimento e Função

3.1 INTRODUÇÃO À PROGRAMAÇÃO EM SQL

A linguagem SQL foi estruturada como uma linguagem de programação comum, assumindo estrutura de controle, decisão, repetição, de forma que possa executar funções (*functions*), procedimentos (*store procedures*) e gatilhos (*triggers*) de maneira eficiente e efetiva.

Este tipo de programação com SQL diretamente no SGBD trás as seguintes vantagens:

- ✓ Reduz a diferença entre o SQL e a linguagem de programação;
- ✓ Por ser armazenada no SGBD permite que possa ser invocado por diferentes aplicações evitando assim a duplicação de código;
- ✓ Reduz custos de comunicação, por ser executada no SGBD;
- ✓ Pode ser utilizado diretamente na elaboração de *functions, store procedures* e *triggers*;

O MySQL utiliza o SQL/PSM (*SQL/Persistent Stored Modules*) que é uma extensão ao SQL. Ele define para a escrita de procedimentos e funções em SQL que juntamente com a utilização de estruturas de controle aumentam consideravelmente o poder expressivo do SQL.

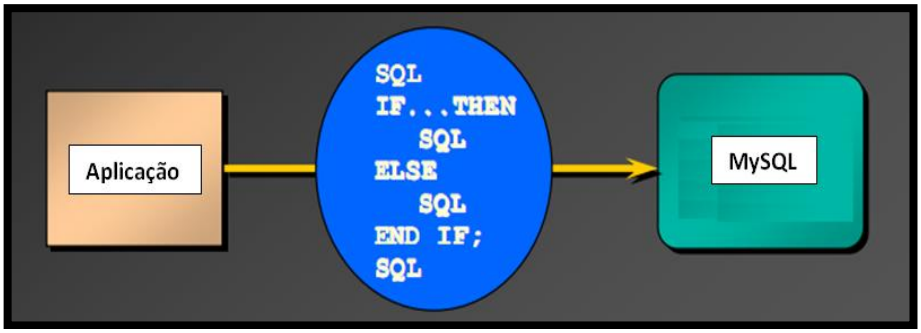

Figura 1 - Bloco de Comandos

Para se programar em SQL/PSM, ou seja, criar programas em SQL, se faz necessário familiaridade com as declarações de variáveis, cursores, atribuições de valores, operadores lógicos, condições, loops, rotinas de controle, comentários. Acumulado esse conhecimento, pode-se criar blocos de comandos para preparar funções (*function*), procedimentos (*store procedure*) e gatilhos (*triggers*), que serão compilados, executados e armazenados diretamente no SGBD. Fazendo com que dessa forma as regras de

negócio sejam disponibilizadas para todas as aplicações que acessam o SGBD.

Iniciaremos nosso estudo em SQL/PSM no MySQL, desde os blocos de comando até a elaboração das funções (*function*), procedimentos (*store procedure*), gatilhos (*triggers*) e ainda controle de acesso de usuário.

Vamos aprender a sintaxe, dos comandos para que possamos programar no MySQL.

3.1.1 COMANDO: BEGIN ... END

As palavras-reservas BEGIN e END atuam como delimitadoras de um bloco de comandos, na criação de programas SQL. Os blocos de comandos são programas compostos por uma ou mais instruções escritas em SQL. Essas mesmas instruções BEGIN... ENG também podem aparecer aninhadas. Temos a estrutura de um bloco de comandos em SQL, a seguir:

```
BEGIN
  [DECLARAÇÃO DE VARIÁVEIS];
  [DECLARAÇÃO DE CURSORES];
  COMANDOS SQL;
  COMANDOS SQL;
END
```

3.1.2 COMANDO: DECLARE

Para que se possa usar uma variável em um programa no MySQL, é necessário fazer a declaração de variável antes. A declaração de variáveis simplesmente informa ao MySQL quais são os nomes dados as variáveis e quais são os tipos usados. No MySQL o nome da variável consiste, basicamente, de caracteres alfanuméricos, podendo ainda ser utilizados os caracteres ‘‘_’, ‘$’ e o ‘.’.

O comando DECLARE, serve para fazer a declaração de variáveis locais, condições, cursores e *handlers*.

Introduzindo o conceito de cursor até então desconhecidos nas linguagens comuns de programação, o MySQL utiliza esse termo para armazenar resultados de algum tipo de processamento realizado no SGBD. No exemplo a seguir temos a declaração de dois cursores, o cur1 e o cur2, observemos que os dois acessam colunas e tabelas diferentes, portanto geram resultados diferentes.

Quando necessário o cursor está presente em funções (function), procedimentos (store procedure) e gatilhos (triggers).

Temos também o termo novo handler. Um handler funciona como um artifício que existem em torno de funções de acesso ao banco de dados MySQL. Ele procura estabelecer conexões persistentes ou não persistentes com o MySQL, executar consultas SQL, recuperar o número de

linhas entre um conjunto de resultados e além de obter o número de linhas afetadas por um INSERT, UPDATE ou DELETE numa consulta no MySQL. E ainda recupera mensagens de erros associadas à última consulta no MySQL que não chegou a ser finalizada devido a uma falha interna qualquer.

O comando DECLARE segue a precedência, assumida pelo MySQL, que determinar que os cursores devem ser declarados antes dos handlers, e as variáveis e condições, antes dos cursores.

Sintaxe do comando DECLARE para declarar variável:

```
DECLARE <nome da variável> <tipo de dados>;
```

Sintaxe do comando DECLARE para declarar condição:

```
DECLARE <nome da condição> CONDITION FOR <valor da
condição>;
```

Sintaxe do comando DECLARE para declarar cursor:

```
DECLARE <nome do cursor> CURSOR FOR <comando
SELECT>;
```

Sintaxe do comando DECLARE para declarar handler:

```
DECLARE <tipo de handler> HANDLER FOR <valor da
condição>;

<tipo de handler> pode ser: CONTINUE, EXIT ou UNDO.
```

Exemplificação do comando DECLARE, a seguir:

```
BEGIN
 DECLARE a CHAR(16);
 DECLARE b, c INT;
 DECLARE processa CONDITION FOR FOUND SET done=1;
 DECLARE cur1 CURSOR FOR SELECT id, data FROM test.t1;
 DECLARE cur2 CURSOR FOR SELECT i from test.t2;
 DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;

END;
```

3.1.3 - COMANDO: SET

Uma vez que já se tenha declarado uma variável no MySQL, deve-se atribuir a mesma algum valor. A operação de atribuição é uma operação muito simples, consiste de atribuir um valor de uma expressão a uma variável, utilizando para isso o comando SET.

O símbolo de atribuição é o = (igual) que no MySQL, também pode ser usado como o operador que representa a igualdade de valores.

As variáveis não precisam ser inicializadas. Elas contém o valor NULL por padrão e podem armazenar valores numéricos (inteiro ou real), string (seqüência de caracteres) ou data e hora.

Sintaxe do comando SET:

```
SET <nome da variável> = <valor a ser atribuído>;
```

O comando SET demonstrado, abaixo:

```
BEGIN
  DECLARE a CHAR(16);
  DECLARE b, c INT;

  SET a = 'EAD-POLÓ';
  SET b=10;
  SET c =76;
  .....
END
```

3.1.4 COMANDO: OPEN, FETCH, CLOSE

Já vimos como declarar um cursor, portanto, vamos agora aprender a utilizá-lo.

O curso age como um mecanismo para manipulação de linhas de uma tabela do MySQL, muitas vezes discriminadas linha por linha. E atuam ainda como ponteiros, uma vez que apontam para a(s) linha(s) do resultado dado pela consulta solicitada no MySQL.

Após a declaração do cursor, ele deve ser inicializado através do comando open.

Sintaxe do comando OPEN:

```
OPEN <nome do cursor>;
```

Posteriormente, a execução do comando OPEN, o MySQL está pronto para manipular o resultado dos comandos SQL. Então, o comando FETCH é executado para que o ponteiro seja posicionado numa linha e as informações atribuídas apontadas pra um conjunto de variáveis (o número de coluna do resultado apontado pelo cursor deve ser igual ao número de variáveis). Portanto, terminado a execução do FETCH, já se pode manipular as variáveis que receberam o valor do cursor.

O comando FETCH é usualmente encontrado nos comandos de iteração, como o REPEAT e o WHILE, que serão vistos em seções posteriores.

Sintaxe do comando FETCH:

```
FETCH <nome do cursor> INTO <nome(s) da(s) variável(s);
```

É para finalizar o uso do cursor, deve-se fechar o mesmo, através do comando CLOSE. A ação de não fechar o cursor pode causar problemas graves no MySQL.

Sintaxe do comando CLOSE:

```
CLOSE <nome do cursor>;
```

Os comandos OPEN, FETCH e CLOSE demonstrados, abaixo:

```
BEGIN
   DECLARE x1 CHAR(16);
   DECLARE y1, z1 INT;
   DECLARE CURSOR cursor1 CURSOR FOR SELECT rg, data1, FROM tabela1.dt1;
   DECLARE CURSOR cursor2 CURSOR FOR SELECT rg, data1, FROM tabela2.dt2;

   OPEN cursor1;
   OPEN cursor2;

   REPEAT

     FETCH cursor1 INTO x1, y1;
     FETCH cursor2 INTO z1;
     ..............................
   END REPEAT;
   .....
   CLOSE cursor1;
   CLOSE cursor2;

  END
```

3.1.5 - COMANDO: SELECT.... INTO

Esse comando é usado para armazenar, no MySQL, o resultado de uma consulta em uma variável. O resultado da consulta deve ter sempre como retorno somente uma linha, caso o resultado tenha mais de uma linha, deve ter o mesmo número de variáveis para receber esses valores.

Se a consulta tiver mais de uma linha como resultado, e não existir variável suficiente receber os valores da consulta, ocorrerá o erro 1172, e aparecerá no MySQL, a seguinte mensagem “Result consisted of more than one row”, caso a consulta retorne nenhum resultado, ocorrerá o erro 1329, e aparecerá no MySQL a mensagem “No data”.

Sintaxe do comando SELECT .... INTO:

```
SELECT <nome da coluna1, coluna2,...N coluna> INTO
<nome da variável1, variável2,...N variável> FROM
<nome da tabela>;
```

O comando SELECT.... INTO está demonstrado, abaixo:

```
SELECT rg, datal INTO xl, yl FROM tabelal.tdl;
```

3.1.6 - COMANDO: IF

A estrutura de decisão permite executar um entre dois ou mais blocos de instruções. No MySQL, temos a estrutura de decisão IF, ele testa se uma condição é verdadeira ou falsa, se for verdadeira executa um conjunto de comandos.

Sintaxe do comando IF:

```
IF <condição> THEN
<comandos SQL>;
<comandos SQL>;
[ELSE IF <condição> THEN <comandos SQL>;
<comandos SQL>;]...
[ELSE <comandos SQL>;<comandos SQL>;
END IF;
```

Exemplificação, do comando IF, abaixo:

```
BEGIN
 DECLARE simbol VARCHAR(10);

 IF num1 > num2 THEN SET simbol = 'maior';
 ELSEIF num1 = num2 THEN SET simbol = 'igual';
 ELSE SET simbol = 'menor';
 END IF;

 RETURN s;

END
```

3.1.7 COMANDO: CASE...WHEN

A estrutura CASE...WHEN é uma estrutura de decisão que permite a execução de um conjunto de instruções SQL, conforme a pesquisa e posterior encontro de um determinado valor.

Sintaxe do comando CASE...WHEN:

```
CASE <valor procurado>
 WHEN <valor da pesquisa1> THEN <comandos SQL>;
 WHEN <valor da pesquisa2> THEN <comandos SQL>;]...
 [ELSE <comandos SQL>;]
END CASE;
```

A Sintaxe do comando CASE...WHEN, também pode ser a seguinte:

```
CASE
 WHEN <valor da pesquisa1> THEN <comandos SQL>;
 WHEN <valor da pesquisa2> THEN <comandos
SQL>;]...
 [ELSE <comandos SQL>;]
END CASE;
```

O comando CASE..WHEN está demonstrado, abaixo:

```
BEGIN
 DECLARE VALOR INT DEFAULT 1;

 CASE VALOR
  WHEN 4 THEN SELECT VALOR;
  WHEN 6 THEN SELECT 0;
  ELSE
   BEGIN
   END;
 END CASE;
END
```

3.1.8 COMANDO: LOOP e ITERATE

O comando LOOP não tem uma condição a ser testada. Para que a repetição termine, o MySQL determina que o comando LEAVE finalize o laço. O comando ITERATE é usado dentro da construção LOOP... END LOOP, serve para reiniciar a repetição, ou seja, o loop.

Sintaxe do comando LOOP:

```
<nome do label> : LOOP
<comandos SQL>;
<comandos SQL>;
 ITERATE <nome do label>;
 <comandos SQL>
 <comandos SQL>;
END LOOP <nome do label>;
```

Os comandos LOOP..END LOOP e ITERATE está demonstrado, abaixo:

```
BEGIN
 label2 : LOOP

  SET num1 = num2 + 20;
  IF num1 < 200 THEN
     ITERATE label2;
  END IF;
  LEAVE label2;

  END LOOP label2;
END
```

3.1.9 COMANDO: REPEAT

Esse comando permite a repetição na execução de um conjunto de comandos. Os comandos serão executados ao menos uma vez, independente da condição testada. A execução do comando REPEAT será mantida enquanto a condição testada for falsa.

Sintaxe do comando REPEAT:

```
REPEAT
<comandos SQL>;
<comandos SQL>;
 UNTIL <condição testada>
END REPEAT;
```

Para melhor exemplificação, temos um exemplo comando REPEAT:

```
BEGIN
 label2 : LOOP

  SET num1 = num2 + 20;
  IF num1 < 200 THEN
     ITERATE label2;
  END IF;
  LEAVE label2;

  END LOOP label2;
END
```

3.1.10 COMANDO: WHILE...DO

Esta estrutura faz com que a condição seja avaliada em primeiro lugar. Se a condição é verdadeira os comandos SQL são executados uma vez e a condição é avaliada novamente. Caso a condição seja falsa a repetição é terminada sem a execução dos comandos SQL.

Sintaxe do comando WHILE..DO:

```
WHILE <condição testada> DO
<comandos SQL>;
<comandos SQL>;
<comandos SQL>;
END WHILE;
```

O comando WHILE...DO está demonstrado, abaixo:

```
BEGIN
  ......................
  WHILE num > xx DO
    .....
    SET xx = xx - 1;

  END WHILE;
END
```

3.1.11 COMANDO: LEAVE

Esse comando é utilizando para sair de uma estrutura de controle, seja de repetição (REPEAT, WHILE, LOOP, ITERATE) ou decisão (IF, CASE).

Sintaxe do comando LEAVE:

```
LEAVE <nome do label>;
OBSERVAÇÃO: o label pode ser o nome de uma função, procedimento ou gatilho, ou simplesmente o nome de um rótulo presente nas estrutura de controle.
```

O comando LEAVE está demonstrado, abaixo:

```
BEGIN
  ......................
  label1 : xxx
    .....
    .....
    LEAVE label1;
  END xxxx label1;
    .....
END
```

3.1.12 COMANDO: CALL

Esse comando é utilizando para chamar um procedimento (store procedure) no MySQL. Posteriormente, veremos como criar um procedimento (store procedure).

Sintaxe do comando CALL:

```
CALL <nome-procedimento> (parâmetros do procedimento);
ou

CALL <nome-procedimento> ();
```

O comando CALL está demonstrado, abaixo:

```
CALL procedimento ();
CALL procedimento1;
```

3.1.13 COMANDOS: RETURN e RETURNS

Esse comando é utilizando para retornar um valor de uma variável armazenada no MySQL. O comando RETURN não é utilizado em procedimentos (store procedure).

Sintaxe do comando RETURN:

```
RETURN <valor de variável>;
```

O comando RETURN está demonstrado, abaixo:

```
CREATE FUNCTION Aumenta_Sal(aumento INT) RETURNS double
 BEGIN
     DECLARE maior_sal REAL;
     UPDATE funcao SET sal=(sal*(aumento/100))+sal;

    SELECT MAX(sal) INTO maior_sal FROM funcao;

    RETURN maior_sal;

 END
```

O comando RETURN é diferente do comando RETURNS. Os dois são usados numa função, o 1º (RETURNS) serve para definir qual tipo de dados irá retornar na função, e o 2º (RETURN) diz o valor de qual variável será definida como retorno. No exemplo, acima, temos a função Aumenta_Sal (...) que irá retornar um valor do tipo DOUBLE, que está armazenado na variável maior_sal.

Sintaxe do comando RETURNS:

```
RETURNS <tipo da variável>;
```

O **MySQL** reconhece *store procedure* (procedimentos armazenados), na nossa apostila, vamos nos referir a eles somente como procedimentos.

3.2 PROCEDIMENTOS (*STORE PROCEDURE)*

Agora que já aprendemos a sintaxe, para que possamos construir as primeiras rotinas que serão executadas no MySQL. Temos que entender, que nossas rotinas poderão ser simples, com poucas linhas de código ou bastante complexas e de código extenso, vai depender muito do que se programa, como numa linguagem de programação comum. A diferença básica e essencial, é que esses nossos programas estarão armazenados no servidor e poderão ser chamados a partir da própria linguagem SQL.

Assim, teremos que utilizar os procedimentos (store procedure) e as funções (function), cujos conceitos estão definidos no SQL e ficam armazenados no servidor, podendo ser invocados, de acordo com a necessidade do SGBD.

Vamos a um assunto muito interessante referente à banco de dados, mais precisamente procedimentos armazenados (store procedure).

Um procedimento é um código procedural, semelhante ao utilizado em linguagens estruturadas, só que ao invés de ter que escrever seus próprios comandos SQL na aplicação, você cria o procedimento (store procedure) no

banco de dados e pode ser executado por qualquer aplicação cliente, o que melhor e muito sua performance de execução.

Já que entendemos o que é um procedimento (store procedure). Saberemos, a partir de agora, os motivos para se utilizar um procedimento (store procedure):

- **Modularização**: os procedimentos (*store procedures*) utilizam a programação modular. Eles encapsulam conjuntos de operações sobre os dados, ou seja, qualquer possível alteração no SGBD fica "escondida" da aplicação que fazem o acesso o banco por meio de procedimento (*store procedure*). E ainda permite que aplicações possam acessar o SGBD de maneira uniforme;
- **Performance**: quando um procedimento (*store procedure)* é executado no banco de dados, um "modelo" daquele procedimento continua na memória para se aproveitada posteriormente, o que melhorar a velocidade de execução;
- **Segurança:** utilizando procedimento (*store procedure)*, o acesso não é feito diretamente nas tabelas, portanto a probabilidade de um desenvolvedor da aplicação fazer algo de errado que possa comprometer a base do banco de dados diminui, o que aumenta a segurança da aplicação desenvolvida utilizando procedimento. Ou podem ter atributos de segurança, portanto os usuários podem ter permissões de executar procedimento (*store procedure)* , sem ter permissões sobre os objetos referenciados dentro do procedimento;
- **Tráfego de rede:** pode reduzir o tráfego na rede gerado pela aplicação, porque quando código SQL fica na aplicação, é necessário que o mesmo seja enviado ao banco (compilado e otimizado) a cada nova consulta, se tivermos muitas linhas de SQL isso irá gerar um tráfego maior, portanto é mais vantajoso ter uma linha de código para executar um procedimento (*store procedure)*.

Atenção!!

Agora é a hora de usar o *comando* **CALL,** toda vez que quiser executar um procedimento precisamos utilizar esse comando.

Os procedimentos (store procedure), podem ser de 03 (três) tipos:

- retornam registros do banco de dados: um simples SELECT em uma tabela;

- retornam um simples valor: pode ser o total de registros de uma tabela;
- não retorna valor (realiza uma ação): pode ser a inserção

Já que entendemos, o que é um procedimento, sua sintaxe é:

```
CREATE PROCEDURE nome-do-procedimento ([parâmetros[,...]])
BEGIN
  [características ...] corpo da função

  parâmetros:
    [ IN | OUT| INOUT ] nome do tipo do parâmetro

  tipo:
    Qualquer tipo de dado válido no MySQL

  características:
      Linguagem SQL
      | [NOT] DETERMINISTIC
      | {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES
         SQL DATA }
      | SQL SECURITY {DEFINER | INVOKER}
      | COMMENT string

  corpo do procedimento:
    comandos válido no SQL

END
```

Onde:

nome-do-procedimento representa o nome do procedimento que será criado.

Os três tipos de parâmetros que pode ser utilizados nos procedimentos são:

- **IN**: é um parâmetro de entrada, ou seja, um parâmetro cujo seu valor seu valor será utilizado no interior do procedimento para produzir algum;
- **OUT**: é um parâmetro de saída, retorna algo de dentro do procedimento, colocando os valores manipulados disponíveis na memória ou no conjunto de resultados;
- **INOUT**: faz o processamento dos **IN** ou **OUT** simultaneamente.

A característica **DETERMINISTIC** diz que o procedimento sempre retorna o mesmo resultado para os mesmos parâmetros de entrada, e a característica **NOT DETERMINISTIC** determina o contrário da característica *determinstc*. Atualmente, essas características são aceitas, pela MySQL, mas ainda não é usada.

Temos a característica que informa o modo de tratamentos dos dados durante a execução do

procedimento. Para **CONTAINS SQL** é o default, determina que os dados não podem ser lidos ou escritos, pois ele já devem ser determinados, no bloco de comandos. O **NO SQL** diz que o procedimento contém dados a serem lidos ou escritos. O **READS SQL DATA** indica que teremos somente leitura de dados, por meio do **SELECT**. E o **MODIFIES SQL DATA** determina que tem-se escrita ou remoção de dados, utilizando o **INSERT** ou **DELETE**.

A característica **SQL SECURITY** pode ser usada para especificar se o procedimento pode ser executado para usar as permissões do usuário que criou os procedimentos, ou do usuário que o invocou. O **DEFINER** é o valor padrão, foi um recurso novo introduzido no **SQL:2003**.

A cláusula **COMMENT** é uma extensão do MySQL, e pode ser usada para descrever o procedimento (*stored procedure*).

Depois que, aprendemos a sintaxe dos procedimentos, vamos ver alguns exemplos implementadas no MySQL, do nosso banco de dados da "*empresa de desenvolvimento de projetos"* utilizado desde o início da nossa apostila:

Antes de elaborar os procedimentos, temos que ter como objetivos, criá-los de forma que eles possam trazer algum tipo utilização, realmente prática, para empresa ou organização, na qual forem implementados.

Poderia ser interessante, para cada departamento, saber relacionar os projetos e seus orçamentos, o usuário pode determinar solicitar um relatório de todos os projetos cujo orçamento for maior do R$ 10.000,00 e a data de início desse projeto. Abaixo, esse exemplo bem simples de procedimento:

```
CREATE PROCEDURE Relatorio_Projeto()
   LANGUAGE SQL
   DETERMINISTIC
   READS SQL DATA

   BEGIN
   select nome, dt_ini from Projeto p
   where p.orcamento > 10000;

   END
```

A gerência de pessoal, pode solicitar os nomes dos funcionários que estão lotados em mais de um projeto, ou solicitar em qual ou quais projeto(s) seu funcionário está lotado, a partir do código desse funcionário. Observem que são dois procedimentos diferentes um sem e outro com parâmetros de entrada.

Atenção!!

A cláusula **IF EXISTS**, do comando **DROP PROCEDURE OU FUNCTION** é uma extensão do MySQL. Ela previne que um erro ocorra se a função ou o procedimento não existe mais no SGBD. Um aviso é produzido e pode ser visualizado

Para resolver estes problemas, vamos à utilização dos procedimentos abaixo.

```
CREATE PROCEDURE Relatorio_Func_Projeto()
    LANGUAGE SQL
    DETERMINISTIC
    READS SQL DATA

    BEGIN
     SELECT nome
     FROM funcionario f
     WHERE cod_func IN (SELECT cod_func FROM trabalha);

   END
```

```
CREATE PROCEDURE Relatorio_Cod_Func(IN cod_func INT)
   LANGUAGE SQL
   DETERMINISTIC
   READS SQL DATA

   BEGIN
    SELECT nome
    FROM projeto
    WHERE  cod_proj IN(SELECT cod_proj FROM trabalha WHERE cod_func);

   END
```

A chamada para a execução de um procedimento é diferente da chamada para função. No procedimento, utilizamos o comando (CALL). Daqui à pouco, aprenderemos como utilizar função.

Vamos invocar os procedimentos criados, e verificar se eles realmente funcionam, no MySQL

```
CALL Relatorio_Projeto;

CALL Relatorio_Func_Projeto;

CALL Relatorio_Cod_Func(2);
```

Algumas características de um procedimento podem ser alteradas, para isso, vamos ver a sintaxe da alteração de um procedimento, a seguir:

```
ALTER PROCEDURE nome-do-procedimento[características
...]
características:
   NAME novo-nome
  | {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES
     SQL DATA }
  | SQL SECURITY {DEFINER | INVOKER}
  | COMMENT string
```

Onde:

nome-do-procedimento representa o nome do procedimento que terá sua(s) característica(s) alterada(s).

O comando ALTER PROCEDURE não é muito utilizado no MySQL, uma vez que não se pode alterar nem os parâmetros nem o *corpo* do procedimento. Caso, se queira fazer muitas modificações no procedimento, recomenda-se apagar o procedimento e criar um novo.

A seguir, vamos aprender como remover uma função:

```
DROP PROCEDURE nome-do-procedimento [IF EXISTS]
nome_warning
```

Onde:

nome-do-procedimento representa o nome do procedimento que terá removido do servidor de banco de dados.

Veremos, a seguir, como remover os três procedimentos que foram criados no MySQL:

```
DROP PROCEDURE Relatorio_Cod_Func;

DROP PROCEDURE Relatorio_Func_Projeto;

DROP PROCEDURE Relatorio_Projeto;
```

Curiosidade!!

Para saber todas as informações sobre as funções do SGBD, podemos usar o comando **SHOW FUNCTION STATUS** e **SHOW PROCEDURE STATUS**, para os procedimentos.

3.3 FUNÇÕES (*Function*)

Vamos agora, aprender sobre funções (*functions)*.

Funções (também chamadas de rotinas, ou sub-programas) são segmentos de programa que executam uma determinada tarefa específica. É possível ao administrador do SGBD, escrever suas próprias rotinas, no MySQL. São as chamadas de funções definidas pelo usuário ou rotinas definidas pelo usuário (UDFS – *User Defined Functions*).

Criando nossas próprias funções, temos a vantagem de adequá-las a nosso ambiente de trabalho, de acordo com as nossas necessidades diárias. Isso tudo, sem ter que modificar a aplicação, uma vez a função está implementada direto na aplicação.

Temos como exemplo, um Banco Comercial, onde um de seus maiores patrimônio são os dados dos correntistas, nesse tipo de organização a mudança de SGBD é remota, procura-se dessa forma obter o máximo possível do servidor de banco de dados, utilizando os recursos que lhes são oferecidos, como a utilização de funções. Outras organizações por utilizar tudo centralizado no SGBD, centralizam também às regras de negócios, tornando-as iguais para qualquer aplicação que venha a acessar o

servidor do banco de dados. Dessa maneira, uma função que venha a ser padronizada no banco de dados, por ser executada por aplicações diferente, seja desenvolvida em Delphi, Java, C.

Então, função são programas armazenados no SGBD, pré-compilados, invocados de forma explícita para executar alguma lógica de manipulação de dados, e que sempre retorna algum valor.

A diferença básica entre o procedimento e uma função é que a função sempre retorna algum valor.

Já que entendemos, o que é uma função, sua sintaxe é:

```
CREATE FUNCTION nome-da-função ([parâmetros[,...]])
[RETURNS tipo]
BEGIN
  [características ...] corpo da função

  parâmetros:nome do tipo do parâmetro

  tipo:
    Qualquer tipo de dado válido no MySQL

  características:
      Linguagem SQL
      | [NOT] DETERMINISTIC
      | {CONTAINS SQL | NO SQL | READS SQL DATA |
         MODIFIES SQL DATA }
      | SQL SECURITY {DEFINER | INVOKER}
      | COMMENT string

  corpo da função:
    comandos válido no SQL
   RETURN <valor>
END
```

parâmetros vazia de **( )** deve ser usada.

A função também apresenta as características **DETERMINISTIC** e **NOT DETERMINISTIC**. As características **CONTAINS SQL**, **NO SQL**, **READS SQL DATA** e **MODIFIES SQL DATA**, possuem as mesmas funcionalidades utilizadas nos procedimentos.

E na questão de segurança, a característica **SQL SECURITY** pode ser usada para especificar se a função possa ser executada usando as permissões do usuário que criou as funções, ou do usuário que a invocou. O **DEFINER** é o valor padrão, foi um recurso novo introduzido no **SQL:2003**.

A cláusula **COMMENT** é uma extensão do MySQL, e pode ser usada para descrever a função (*function*).

A cláusula **RETURNS** pode ser especificada apenas por uma **FUNCTION**. É usada para indicar o tipo de retorno

da função, e o corpo da função deve conter uma instrução **RETURN** e o valor de retorno.

Depois que, aprendemos a sintaxe da função, vamos ver alguns exemplos de funções, implementadas no MySQL, do nosso banco de dados ***praticabd***, utilizado desde o início da nossa apostila.

Antes de elaborar as funções, temos que ter como objetivos, criá-las de forma que elas possam trazer algum tipo utilização, realmente prática, para empresa ou organização, na qual forem implementadas.

Caso cada departamento, queira saber a classificação dos projetos de acordo com o seu orçamento, a função está exemplificada a seguir:

```
CREATE FUNCTION Classifica_Orcament(orcamento INT)
       RETURNS VARCHAR(15)
   LANGUAGE SQL
   DETERMINISTIC
   READS SQL DATA

   BEGIN
      IF orcamento >=100000 THEN RETURN 'Muito Alto';
      ELSEIF orcamento >= 50000 THEN RETURN 'Alto' ;
      ELSEIF orcamento >= 30000 THEN RETURN 'Medio' ;
      ELSEIF orcamento >= 10000 THEN RETURN 'Baixo' ;
      ELSE  RETURN 'Muito Baixo' ;
      END IF;

   END
```

A gerência de pessoal poderia solicitar ao SGBD, um aumento salarial a todos os funcionários e ter como retorno o valor do maior salário, após o aumento. Vamos a elaboração dessa função no MySQL:

```
CREATE FUNCTION Aumenta_Sal(aumento INT)
       RETURNS REAL
   LANGUAGE SQL
   DETERMINISTIC
   MODIFIES SQL DATA

   BEGIN
     DECLARE maior_sal REAL;
     UPDATE funcao SET sal=(sal*(aumento/100))+sal;

     SELECT MAX(sal) INTO maior_sal FROM funcao;

    RETURN maior_sal;

   END
```

Para verificarmos, se as funções acima funcionarão e se retornaram o que se esperava dele, podemos usar os SELECT abaixo:

Vamos invocar as funções criadas para saber a classificação do orçamento, e verificar se elas realmente funcionam.

```
SELECT Classifica_Orcament(155000);
SELECT Classifica_Orcament(55000);
SELECT Classifica_Orcament(35000);
SELECT Classifica_Orcament(15000);
SELECT Classifica_Orcament(1500);
```

Agora, vamos invocar a Aumenta_Sal(*aumento* **INT**) e vê sua funcionalidade, para um aumento de 10%.

```
SELECT Aumenta_Sal(10);
```

Lembrando que uma função sempre retorna algum valor, por isso podemos utilizar o valor do retorno da função. E para utilizar esse valor, é preciso que seja criada uma variável do mesmo tipo do retorno da função, e o nome da função deve aparecer ao lado direito de um comando de atribuição de valores (SET). As funções armazenadas no MySQL podem ser utilizadas em comandos SQL da mesma forma que as outras funções já criadas na própria linguagem.

Algumas características de uma função podem ser alteradas, para isso usamos a sintaxe, a seguir:

```
ALTER FUNCTION nome-da-função [características ...]
características:
   | {CONTAINS SQL | NO SQL | READS SQL DATA | MODIFIES
     SQL DATA }
  | SQL SECURITY {DEFINER | INVOKER}
```

Onde:

nome-da-função representa o nome da função que terá sua(s) característica(s) alterada(s).

Esse comando não é muito utilizando, uma vez que não se pode alterar os parâmetros ou o *corpo* da função. Caso se queira alterar, o código de uma função o ideal é removê-la e criar uma nova função. A seguir, vamos aprender como remover uma função.

Portanto, para finalizar o nosso estudo sobre função, podemos deletar essa rotina do SGBD, para isso usamos a sintaxe, abaixo:

```
DROP FUNCTION nome-da-função [IF EXISTS] nome_warning
```

Onde:

nome-da-função representa o nome da função que terá removida do servidor de banco de dados.

Teremos, a seguir, a remoção das funções criadas no MySQL.

```
DROP FUNCTION Classifica_Orcament;

DROP FUNCTION Aumenta_Sal;
```

Para listarmos todos os procedimentos (*store procedure*) e as funções (*functions*) armazenados no MySQL, temos que utilizar a sintaxe a seguir:

```
SELECT * FROM INFORMATION_SCHEMA.ROUTINES;
```

Esse último comando lista somente os procedimentos e funções, gravadas na tabela ROUTINES do banco de dados INFORMATION_SCHEMA.

EXERCÍCIO AULA

01. A empresa de elaboração de projetos possui uma falha grave em seu banco de dados ***praticabd***, apesar de não trabalhar com projeto cujo orçamento está acima de R$ 100.000,00, existe esse tipo de registro em seu **SGBD,** implementado no ***MySQL***. Você foi contrato como técnico em informática, para resolver esse problema, utilizando um procedimento (*store procedure*).
02. Agora faça a chamada para o procedimento (*store procedure*) criado na questão anterior.
03. Para finalizar a utilização de procedimento (*store procedure*). Use o comando apagar o procedimento no ***MySql***.
04. Usando função, faça a classificação dos salários dos funcionários de acordo com a tabela abaixo:

VALOR DO SALÁRIO (R$)	RETORNO DA FUNÇÃO
Maior ou igual a R$ 10.000,00	Classe Alta
Maior ou igual a R$ 5.000,00	Classe Média Alta
Maior ou igual a R$ 1.650,00	Classe Média
Maior ou igual R$ 650,00	Classe Baixa
Menor ou igual R$ 649,00	Classe Muito Baixa

Observação: *a tabela acima, não é oficial, sendo elaborada unicamente, para efeito de atividade deste exercício. Para mais informações, acessar o site do IBGE <www.ibge.gov.br>*

05. Agora faça a chamada para a função (*function*) criado na questão anterior.
06. Para finalizar a utilização de função (*function*). Use o comando para apagar a função no ***MySql***.

RESUMO DO CAPÍTULO

Começamos uma introdução na linguagem SQL/PSM (*SQL/Persistent Stored Modules*), voltada para utilização no MySQL. Depois, utilizamos o estudo de caso de uma empresa de desenvolvimento de projeto, para verificar como modelar o banco de dados do sistema dessa empresa, utilizando procedimentos e funções.
Portanto, a Aula 3, procurar fazer uma iniciação no estudo de procedimentos (*store procedure*) e funções (*functions*) no MySQL, buscando ensinar o manejo destes recursos que é de grande proveito, principalmente no sistemas que necessitam de modularidade e otimização quanto à performance.

Informações para a próxima aula

Na Aula 4, teremos a oportunidade de conhecer e implementar um gatilho (*trigger*) o controle de acesso de usuários.

Bibliografia consultada

- DESCOBRE. Tipos de Linguagem de Programação. Disponível em <http://www.descobre.com/forum/showthread.php?t=697>. Acesso em 23. mar. 2010
- Manual MySQL 5.1< http:// dev.mysql.com/doc/refman/5.1/en/index.html>. Acesso em 24.mar. 2010

Gatilho e Controle de Aceso

4.1 GATILHO (*TRIGGER)*

A linguagem SQL além de tratar os procedimentos e funções, também permite a criação de gatilhos (*triggers*).

É considerado uma lógica de processamento procedural, armazenada no SGBD e disparada automaticamente pelo servidor sob condições específicas. Gatilhos (*triggers*) representam regras do mundo (negócio) que definem a integridade ou consistência do BD.

Passaram a ser padrão SQL em 1999. Seu principal objetivo é monitorar o SGBD e realizar alguma ação quando uma condição ocorre.

Os gatilhos (*triggers)* devem ser armazenados na base de dados como objetos independentes e não podem ser locais a um bloco ou pacote. São na verdade, procedimentos disparados automaticamente pelo SGBD em resposta a um evento específico do banco de dados. Portanto, são bastante semelhantes aos procedimentos (*store procedure*) só que tem sua execução disparada pelo SGBD quando ocorre um acontecimento/evento desencadeador de "*trigerring*" suceder e não aceita argumentos. O ato de executar um gatilho (trigger) é conhecido como ***disparar*** o gatilho (*trigger)*. O evento desencadeador pode ser uma operação **DML** (**INSERT**, **UPDATE**, ou **DELETE**) em uma tabela da base de dados.

Os gatilhos (*triggers)* podem usados para:

- Segurança sobre a base de dados, e ação de usuários, verificando quando uma operação é realizada sobre uma entidade, o gatilho (trigger) é disparado para verificar as permissões do usuário;
- Melhorar a segurança no acesso aos dados;
- Assegurar as restrições de integridade;
- Fazer a auditoria das informações de uma tabela, registrando as alterações efetuadas e quem as efetuou;
- Sinalizar automaticamente a outros programas que é necessário efetuar uma ação, quando são efetuadas alterações numa tabela.

A seguir são enumeradas algumas vantagens no uso de gatilhos (*triggers*):

- Um gatilho (*trigger)* sempre é disparado quando o evento ocorre, evitando assim esquecimentos ou falta de conhecimento sobre o banco de dados;

- São adminstrados de forma centralizada, ou seja, o DBA (Adminstrador de Banco de Dados) define suas situações, eventos e ações;
- A ativação central combina com o modelo cliente/servidor, portanto a execução da *trigger* é realizada no servidor, independente do aplicativo executado pelo cliente.

Com o uso de gatilho (trigger) procura-se eliminar o esforço manual e repetitivo de identificar e corrigir comandos e regras mal implementados, com baixo desempenho. Assim, o desenvolvedor da aplicação sairá ganhando em produtividade, uma vez que não irá mais perder seu tempo em corrigir e buscar regras já prontas e controladas no SGBD.

Por se tratar de mecanismo ativos, os gatilhos (*triggers*), utilizam como requisitos para sua elaboração, o paradigma **Evento**-**Condição**-**Ação** (**ECA**). Onde o "*Evento*", indica o momento do disparo da regra, a "*Condição*" precisa ser satisfeita para que a execução do gatilho (*trigger*) prossiga e a "*Ação*" determina o que ter de ser feito, caso a condição seja realmente válida.

Há três tipos principais de gatilhos (*triggers*): DML, *insted-of* e gatilhos (*triggers*) de sistema (DDL).

- Um gatilho (*trigger*) DML é acionado em uma operação INSERT, UPDATE ou DELETE de uma tabela de dados. Ele pode ser acionado antes ou depois que a instrução é executada e pode ser acionado uma vez por linha problemática ou uma vez por instrução;
- Os gatilhos (*triggers*) *insted-of* podem ser definidos apenas em visões (tanto de objeto como relacional).
- Um *trigger* de sistema é acionado quando um evento de sistema como uma inicialização ou desativação de banco de dados ocorrer, em vez de em uma operação DML em uma tabela. Um *trigger* de sistema também pode ser acionado em operações de DDL como a criação de uma tabela.

Devemos ter cuidado quando utilizar gatilhos (*triggers*), pois erros na execução de um gatilho (*trigger*) pode causar falhas nas funções de incluir/deletar/atualizar da função que o disparou. Não podemos criar gatilho (*trigger*) para disparar outro gatilho (*trigger*), ou seja, na pior das hipóteses gerar uma cadeia infinita de gatilhos (*triggers*) e também não se deve usar gatilho (*trigger*) na replicação da base dados, porque quando alguma modificação é feita na base dados

principal, um gatilho (*trigger*) aplica a mesma modificação na cópia.

Cada SGBD utiliza sua própria linguagem e sintaxe para gatilhos (*triggers*). Iremos aprender agora, como elaborar gatilhos (*trigger*) no MySQL. Importa ressaltar que iremos nos ater aos gatilhos (*triggers*) DML.

Já que aprendemos o que é um gatilho (*trigger*), sua sintaxe a seguir:

```
CREATE TRIGGER nome-do-gatilho momento-da-execução
               evento-disparador
ON nome-da-tabela FOR EACH ROW comandos válidos no
SQL OU
BEGIN

  corpo do gatilho:
    comandos válido no SQL
```

Onde:

nome-do-gatilho representa o nome do gatilho (*trigger*) que será criado;

momento-da-execução diz em que tempo a ação ocorrerá, antes (**BEFORE**) ou depois (**AFTER**) do evento;

evento-disparador representa o evento que dispara o gatilho (*trigger*), são os comandos **INSERT**, **UPDATE** e **DELETE** do SQL;

nome-da-tabela diz o nome da tabela que será utilizado pelo gatilho (*trigger*);

Para um melhor entendimento do que seja o momento da execução e um evento-disparador durante a execução de um gatilho (*trigger*), observemos a Tabela 1, a seguir:

Tipo de Gatilho	Descrição
BEFORE INSERT	**O gatilho é disparado antes de uma ação de inserção**
BEFORE UPDATE	**O gatilho é disparado antes de uma ação de alteração**
BEFORE DELETE	**O gatilho é disparado antes de uma ação de remoção**
AFTER INSERT	**O gatilho é disparado depois de uma ação de inserção**
AFTER UPDATE	**O gatilho é disparado depois de uma ação de alteração**
AFTER DELETE	**O gatilho é disparado depois de uma ação de remoção**

Tabela 1 – Tipos de Gatilhos

É possível combinar alguns dos modos, desde que tenham a operação de AFTER ou BEFORE em comum, conforme mostra a Tabela 2, abaixo:

Tipo de Gatilho	Descrição
BEFORE INSERT ou UPDATE ou DELETE	**O gatilho é disparado antes de uma ação de inserção ou de alteração ou de remoção**

Tabela 2 – Combinação dos Tipos de Gatilhos

Depois que, aprendemos a sintaxe do gatilho (*trigger*), vamos implementar no nosso banco de dados ***praticabd*** utilizando o MySQL, alguns gatilhos (*triggers*) realmente úteis para a regras de negócios da nossa empresa e seus projetos.

A empresa pode quer ter um controle de todas as datas de previsão de finalização prevista de seus projetos. Para isso podemos implementar um gatilho (*trigger*) para colher as datas de previsão de término do cadastro de todos os projetos e estas serão inseridas numa outra tabela, por exemplo, uma tabela com a seguinte estrutura.

```
CREATE TABLE DataProj
(novo_codproj INT auto_increment,
dt_prev_term DATE NOT NULL,
CONSTRAINT pk_DataProj PRIMARY KEY(novo_codproj));
```

Toda vez que um novo projeto for inserido no cadastro de projeto, tanto será acrescido na tabela do projeto, quanto na nova tabela criada (*DataProj*), isso automaticamente através do gatilho (*trigger*), abaixo:

```
CREATE TRIGGER Data_Final_Projeto AFTER INSERT on projeto
FOR EACH ROW BEGIN

  IF (NEW.dt_prev_term IS NOT NULL) THEN
    INSERT INTO dataproj SET dtprevterm = NEW.dt_prev_term;
  END IF;

END
```

Para testar o gatilho (*trigger*), fazemos uma inserção na tabela projeto, no campo referente à data de previsão do término do projeto, e depois fazemos um SELECT na tabela *dataproj*, pra verificarmos se a data 2015-01-12 também foi inserida automaticamente nela, sintaxe é:

```
INSERT INTO projeto SET dt_prev_term='2015-01-12';

SELECT * from dataproj;
```

Para finalizar o nosso estudo sobre gatilho (*trigger*), iremos verificar como deletar um gatilho (*trigger*), sintaxe é:

```
DROP TRIGGER nome-do-gatilho [IF EXISTS]
nome warning
```

Onde:

nome-do-gatilho representa o nome do gatilho que terá removido do servidor de banco de dados.

Teremos, a seguir, a remoção dos gatilhos que foram criados no MySQL.

```
DROP TRIGGER Data_Final_Projeto;
```

Para listarmos todos os gatilhos (*triggers*) armazenados no MySQL, temos que utilizar a sintaxe a seguir:

```
SELECT * FROM INFORMATION_SCHEMA.TRIGGERS;
```

Esse último comando lista somente os gatilhos, gravados na tabela TRIGGERS do banco de dados INFORMATION_SCHEMA.

4.2 CONTROLE DE ACESSO

A integridade do sistema de Banco de Dados depende muito do tipo de segurança implementado ao SGBD. Para isso existe um DBA (Administrador de Banco de Dados) responsável pela segurança e controle de acesso a esse banco de Dados. Por isso, iremos estudar como implementar o controle de acesso e melhorar a segurança no MySQL.

Quando iniciamos os trabalhos em um servidor MySQL, normalmente precisamos de uma identidade e uma senha. Essa identidade é o que vai fazer com que o MySQL reconheça o usuário. Cabe portanto ao DBA (Administrador de Banco de Dados) criar os usuários que poderão usar o MySQL, podendo também renomeá-los e removê-los permanentemente do SGBD.

A sintaxe para criar um usuário é:

```
CREATE USER usuário [IDENTIFIED BY[PASSWORD]]
'senha';
```

Onde:

usuário representa o nome do usuário que será criado;

'senha' representa a senha de acesso ao MySQL pelo usuário criado.

Teremos, a seguir, a criação de um usuário no MySQL:

```
CREATE USER aluno_EAD;
```

Temos também a possibilidade de criar o novo usuário no MySQL já com senha cada um:

```
CREATE USER alunoEAD IDENTIFIED BY '2010';
```

Depois de criarmos um usuário com senha no MySQL, temos a possibilidade de alterarmos a senha desse usuário, conforme a sintaxe abaixo:

```
SET PASSWORD FOR usuário PASSWORD ('nova senha');
```

Veremos como alterar a senha do usuário *alunoEAD*, exemplificado no a seguir:

```
SET PASSWORD FOR alunoEAD = PASSWORD ('2014');
```

Caso se queira, renomear um usuário, a sintaxe é:

```
RENAME USER usuário_velho TO usuário_novo;
```

Abaixo, um exemplo de renomeação de usuário:

```
RENAME USER alunoEAD TO aluno_E_A_D;
```

E para remover um usuário no MySQL, a sintaxe é:

```
DROP USER usuário;
```

A exemplificação de como remover um usuário no MySQL, a seguir:

```
DROP USER aluno_E_A_D;
```

Definido no MySQL, os usuários precisamos definir quais privilégios eles terão direito.

O sistema de privilégios do MySQL faz a autentição de um usuário a partir de uma determinada máquina e associa este usuário com privilégio ao banco de dados como usar os comandos **SELECT**, **INSERT**, **UPDATE** ou **DELETE**. Isso

garante que os usuários só possam fazer aquilo que lhes é permitido.

Quando nos conectamos a um servidor MySQL, nossa identidade é determinada pela máquina de onde nos conectamos e nosso nome de usuário que foi especificado. O MySQL concede os privilégios de acordo com a identidade e com o que se deseja fazer no SGBD.

O controle de acesso do MySQL é realizado em dois passos:

1º Passo: É feita a conferência para saber se o usuário pode ter ou não acesso. Nesse caso, o usuário tenta se conectar, e o MySQL aceita ou rejeita a conexão baseado na identidade do usuário e no uso de sua senha correta.

2º Passo: Caso o usuário posso se conectar, o SGBD verifica da solicitação feita pra saber se o usuário tem ou não privilégios para realizar cada solicitação requisitada. Tome como exemplo, o caso de um usuário que tenta altera a linha de uma tabela em um banco de dados ou uma tabela do mesmo banco, o MySQL irá se certificar que o usuário tem o privilégio **UPDATE** para a tabela ou o privilégio **DROP**.

Curiosidade!!

Existem mais privilégios que deve ser concedido apenas a administradores, são eles o ***REPLICATION CLIENT, REPLICATION SLAVE, SUPER, CREATE USER, DROP USER, RENAME USER.***

Os privilégios são armazenados em quatro tabelas interna, no banco MySQL, a saber: ***mysql.user****: privilégios globais aplicam-se para todos os banco de dados em um determinado servidor;* ***mysql.db****: privilégios de banco de dados aplicam-se a todas as tabelas em um determinado banco de dados.* ***mysql.tables_priv****: privilégios de tabelas que aplicam-se a todas as colunas em uma determinada tabela.* ***mysql.colums_priv****: privilégios de colunas aplicam-se a uma única coluna em uma determinada tabela.*

4.3 PRIVILÉGIOS

Quando nos conectamos a um servidor MySQL, nossa identidade é determinada pela máquina de onde nos conectamos e nosso nome de usuário que foi especificado. O MySQL concede os privilégios de acordo com a identidade e com o que se deseja fazer no SGBD.

Os privilégios fornecidos pelo MySQL são descritos na tabela abaixo:

Privilégios MySQL
SELECT – INSERT – UPDATE – DELETE – INDEX – ALTER – CREATE – DROP – GRANT OPTION – RELOAD – SHUTDOWN – PROCESS - FILE - ALL

Tabela 4 – Privilégios MySQL

Os privilégios **SELECT**, **INSERT**, **UPDATE** e **DELETE** permitem que sejam realizadas operações nos registros das tabelas existentes no SGBD.

O privilégio **INDEX** permite a criação ou remoção de índices.

O privilégio **ALTER** permite que se altere tabelas (adicione colunas, renomei colunas ou tabelas, altere tipos de dados de colunas), aplicado a tabelas;

Os privilégios **CREATE** e **DROP** permite a criação de banco de dados e tabelas, ou a remoção de banco de dados e tabelas existentes.

O privilégio **GRANT OPTION** permite que um usuário possa fornecer a outros usuários os privilégios que ele mesmo possui.

Os privilégios **RELOAD**, **SHUTDOWN**, **PROCESS**, **FILE** e **ALL** são usados para operações administrativas que são realizadas pelo DBA (Administrador do Banco de Dados).

No MySQL, ou em qualquer SGBD, o ideal é conceder privilégios somente para usuários que realmente necessitem, por isso deve-se ter muito cuidado ao conceder certos privilégios. Por exemplo, o privilégio **GRANT OPTION** permite que usuários possam repassar seus privilégios a outros usuários, portanto dois usuários com diferentes privilégios e com privilégio **GRANT OPTION** conseguem combinar seus privilégios, o que pode ser um risco numa organização, dependo do tipo de privilégio cedido a cada usuário desses. O privilégio **ALTER** pode ser usado para subverter o sistema de privilégios, quando o usuário pode renomear todas as tabelas do banco de dados.

Após o entendimento, do que sejam os privilégios, vamos vê como atribuir os privilégios a um usuário sua sintaxe é:

```
GRANT privilégios [(coluna)] [, privilégio
[(coluna)] ...]
ON {nome_tabela – função – procedimento | * | *.* |
nome_banco.*}
TO nome_usuario [IDENTIFIED BY 'senha']
[, nome_usuario [IDENTIFIED BY 'senha'] ...]
[WITH GRANT OPTION]
```

Onde:

privilégios: representa qual privilégio será concedido.

colunas: é opcional e especifica as colunas a que os privilégios se aplicam. Pode ser uma única coluna ou várias separadas por vírgula;

nome-tabela, função ou *procedimento,* *****, ***.*** , *nome_banco.******: é o banco de dados, tabela, função ou procedimento a que os privilégios se aplicam. Pode ser especificado como: ***.*** – neste caso os privilégios aplicam-se a todos os bancos de dados; ***** – os privilégios aplicam-se se não estiver utilizando nenhum banco de dados em particular; *nome_banco.******- neste caso, os privilégios aplicam-se a todas as tabelas do banco; *banco.tabela* – neste caso os privilégios aplicam-se a uma determinada tabelas. Você pode, ainda especificar alguma(s) coluna(s) sem particular inserindo esta(s) em colunas;

nome-usuário: especifica um usuário do MySQL. Para preservar a segurança este usuário não deve ser o mesmo usuário do sistema;

senha: senha do usuário para acessar o servidor MySQL;

WITH GRANT OPTION: se for especificado, o usuário poderá conceder privilégios a outros usuários.

Vejamos, a seguir, alguns exemplos de privilégios concedidos a alguns usuários.

Nesse primeiro exemplo, temos a concessão de todos os privilégios ao usuário *aluno_E_A_D*, incluindo o privilégio que permite que o usuário possa conceder privilégios a outros usuários, sintaxe a seguir:

```
GRANT ALL PRIVILEGES ON *.* TO aluno_E_A_D WITH GRANT OPTION;
```

No próximo exemplo, o usuário aluno_E_A_D terá os privilégios de realizar as operações SELECT, INSERT, UPDATE e DELETE no nosso banco de dados ***praticabd***, podemos constatar que diferente do exemplo anterior, o usuário agora não poderá conceder privilégios para os outros usuários, a sintaxe a seguir:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON praticabd.* TO aluno_E_A_D;
```

E para revogar direitos aos usuários do MySQL, usamos o REVOKE, sua sintaxe a seguir:

```
REVOKE privilégio [(coluna)] [, privilégio [(coluna)]
...]
ON {nome_tabela | * | *.* | nome_banco.*}
FROM nome_usuario [, nome_usuario ...]

ou

REVOKE ALL PRIVILEGES, GRANT OPTION FROM nome_usuário
```

Onde:

privilégios: representa qual privilégio será removido;

colunas: é opcional e especifica as colunas a que os privilégios serão removidos. Pode ser uma única coluna ou várias separadas por vírgula;

*nome-tabela, **, **.** , nome_banco.**: é o banco de dados ou tabela a que os privilégios se aplicam. Pode ser especificado como: **.** – neste caso os privilégios serão removidos de todos os bancos de dados; ***** – os privilégios removidos aplicam-se se não estiver utilizando nenhum banco de dados

em particular; *nome_banco.**- neste caso, os privilégios removidos aplicam-se a todas as tabelas do banco;

nome-usuário: especifica um usuário do MySQL que terá seus privilégios removidos;

Ou:

ALL PRIVILEGES: para remover todos privilégios do usuário;

GRANT OPTION: para remover o privilégio do usuário de conceder privilégio a outros usuários;

nome-usuário: especifica um usuário do MySQL que terá seus privilégios removidos;

Temos a seguir, a remoção dos privilégios do usuário aluno_E_A_D. Esse usuário tinha como privilégios manipular as operações de SELECT, INSERT, UPDATE e DELETE, no banco de dados ***praticabd***, a sintaxe é:

```
REVOKE SELECT, INSERT, UPDATE, DELETE ON praticabd.* FROM aluno_E_A_D
```

No próximo exemplo temos a revogação de todos os privilégios concedidos ao usuário *aluno_E_A_D*, inclusive o privilégio de conceder privilégio a outros usuários, a sintaxe é:

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM aluno_E_A_D;
```

EXERCÍCIO AULA

01. Aproveitando a tabela ***dataproj*** , do nosso banco de dados ***praticabd***. Vamos criar um gatilho (*trigger*) para remover automaticamente, a data da tabela ***dataproj***, toda vez que um projeto for removido da tabela ***projeto***.
02. Aproveitando a questão anterior. Vamos testar o gatilho (*trigger*) que foi gerado na questão anterior.
03. Aproveitando a questão 01. Apague o gatilho (*trigger*) que foi criado.
04. Tratando, agora de controle de acesso de usuário ao banco de dados ***praticabd***. Crie um usuário o ***aluno_info*** com a senha "4567".
05. Renomei o usuário ***aluno_info*** para o nome de usuário ***al_EAD***.
06. Altere a senha do usuário ***al_EAD*** para a nova senha "***1234***"

RESUMO DO CAPÍTULO

Nesta aula 4, conceituamos os gatilhos (triggers) que são uma adição valiosa ao banco de dados, podem ser utilizados para impor restrições de dados que são muito mais complexas do que restrições referenciais normais de integridade. Na verdade, os gatilhos (triggers) são executados quando um evento ocorre podendo ser uma inclusão, alteração ou exclusão ocorre em uma tabela do SGBD. Quando nos referimos a prática de banco de dados, não podemos deixar de focar no controle de acesso. No MySQL é necessário saber manipular o usuário do banco de dados, desde sua geração até saber como deletá-lo. Um usuário deve ter nível de permissões ou privilégios no SGBD para que possa executar suas tarefas. O ideal é que se um usuário só precisa escrever numa determinada tabela, dê a ele somente permissão pra escrever, se ele só precisa lê, então só dê a ele privilégio de lê. Portanto, quanto menos privilégio o usuário tiver para executar alguma tarefa, melhor para a segurança do MySQL.

Referências

- DESCOBRE. Tipos de Linguagem de Programação. Disponível em <http://www.descobre.com/forum/showthread.php?t=697>. Acesso em 23. mar. 2010
- Manual de Referência do MySQL, 1997. Disponível em <http://www.descobre.com/forum/showthread.php?t=697>. Acesso em 24.mar. 2010
- Manual MySQL 5.1< http:// dev.mysql.com/doc/refman/5.1/en/index.html>. Acesso em 24.mar. 2010
- LOUCOPOULOS, P. Conceptual Modeling, in: P. Loucopouls, Conceptual Modeling, Databases, and CASE – New York et, al: John Wiley & Sons, 1992.