CAPÍTULO 6

# PROCEDIMENTOS ARMAZENADOS E GATILHOS

Fala Professor

*Olá!*

*Alguns sistemas gerenciadores de bancos de dados – e o MySQL é um deles – permitem armazenar não somente dados, como também pequenos programas que acessem e manipulem esses mesmos dados.*

*Esses miniprogramas são geralmente identificados como rotinas armazenadas e podem ser basicamente de dois tipos: procedimentos armazenados ou gatilhos.*

*Neste capítulo abordaremos introdutoriamente o desenvolvimento e o alcance dessas rotinas armazenadas. Veremos como as mesmas podem ser criadas e executadas no MySQL.*

***Prof. Vanderson José Ildefonso Silva***

## 6.1 A importância de rotinas armazenadas

Conceitos

**Rotinas Armazenadas** são um conjunto de **comandos SQL** que podem convenientemente ser armazenados em um servidor.

**Procedimentos Armazenados** (***Stored Procedure***) são rotinas armazenadas que devem ser executadas por intermédio de invocações explícitas da parte do usuário (comando **CALL**).

**Gatilhos** (***Triggers***) são rotinas armazenadas que, associadas a eventos percebidos automaticamente pelo MySQL, não precisam ser invocados explicitamente pelo usuário para serem executados. Quando o evento programado acontece, o gatilho é disparado automaticamente.

Rotinas Armazenadas apresentam as seguintes vantagens no contexto de um banco de dados:

1. Centralizam as operações críticas do banco de dados. É o que ocorre quando programas aplicativos, escritos em diferentes linguagens de programação e rodando sob diferentes plataformas precisam executar as mesmas operações de bancos de dados.

2. Padronizam as operações de bancos de dados, que deixam de depender excessivamente da memória e da disciplina de usuários humanos. Se todo dia, por exemplo, há a necessidade de executar uma sequência de operações que envolve a inclusão de registros em uma tabela T1 e a consequente atualização de registros em uma tabela T2 para, somente então, gerar um relatório impresso R1 com base nos dados de T1 e T2, uma rotina armazenada pode ser útil. Executada todo dia, assegura que T2 seja atualizado apenas após a inclusão de registros em T1 e que R1 seja gerado apenas após a atualização de T2. Reduz-se assim a margem para erros humanos.

3. Os usuários finais deixam de manipular ou acessar diretamente as tabelas e passam a fazê-lo somente através das rotinas armazenadas, que foram testadas exaustivamente antes de entrarem em produção e fazem validação dos dados manipulados.

4. Podem melhorar a performance do banco de dados, providenciando um tempo de resposta menor para operações que exigem intensa troca de informações do Servidor com as aplicações Clientes. Com as rotinas armazenadas, validações de informações, que exigiriam um pesado intercâmbio de dados, passam a ser realizadas no Servidor de Banco de Dados em vez de nos Clientes. Um bom exemplo é quando um cliente solicita ao servidor de banco de dados que registre a venda de um produto (inclusão na tabela VENDAS). Após o servidor confirmar a conclusão da operação, o cliente solicita que o estoque do produto seja atualizado pelo Servidor (atualização na tabela PRODUTO). O fluxo excessivo de informações entre o Servidor e seus Clientes pode ser reduzido quando tudo que o Cliente deve fazer é executar uma rotina armazenada no Servidor. Esta se encarrega de inserir registros na tabela de VENDAS e atualizar os registros na tabela de PRODUTOS, sem que Servidor e Clientes necessitem trocar mais informações.

## 6.2 Procedimentos armazenados

Novamente devemos entrar no banco de dados EXEMPLO, acessando o MySQL como ***root*** (Administrador de Banco de Dados).

Para a criação de um Procedimento Armazenado (Stored Procedure), utilizamos o comando CREATE PROCEDURE, conforme exemplo abaixo:

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE bilheteriaMedia()
    -> BEGIN
    ->   SELECT ANO, AVG(BILHETERIA) AS BILHETERIA_MEDIA
    ->   FROM FILMES
    ->   GROUP BY ANO;
    -> END//
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql>
mysql> DELIMITER ;
```

Antes do comando CREATE PROCEDURE propriamente, devemos executar o comando "DELIMITER //", que substitui temporariamente o caractere ';" pelos "//" como finalizador de comandos no MySQL. Significa que ao deparar com um ponto-e-vírgula, o interpretador de comandos do MySQL não "entende" que o usuário terminou a digitação do comando. Isso foi muito útil, por exemplo, na linha do "**GROUP BY ANO;**". Se não tivéssemos usado o comando "**DELIMITER //**" antes, certamente haveria um erro exatamente nessa linha do procedimento armazenado. Note que a definição do procedimento armazenado termina com o comando "**END//**". Então, o interpretador de comandos do MySQL "entende" que o usuário já completou o comando.

Após a criação do procedimento armazenado, devemos executar o comando "**DELIMITER;**" para que o ponto-e-vírgula volte a ser o finalizador de comandos.

O procedimento armazenado acima foi criado com o nome de bilheteriaMedia ("**CREATE PROCEDURE bilheteriaMedia( )**"). Os comandos do procedimento propriamente começam com o comando "**BEGIN**" (Início). Em seguida, há uma consulta SQL normal que começa com "**SELECT**" e termina com "**GROUP BY ANO;**". Em seguida, o comando "**END//**" fecha o bloco de comando iniciado com "**BEGIN**". Ou seja, para todo comando "**BEGIN**" deve haver um comando "**END**".

O objetivo deste procedimento armazenado é exibir a média das bilheterias por ano de produção dos FILMES. Para executá-la, utilizamos o comando "**CALL bilheteriaMedia( )**".

```
mysql> CALL BILHETERIAMEDIA;
+------+------------------+
| ANO  | BILHETERIA MEDIA |
| 1972 |       600.000000 |
| 2007 |       390.890000 |
| 2009 |       660.296667 |
3 rows in set (0.00 sec)

Query OK, 0 rows affected (0.02 sec)
```

Geralmente criam-se procedimentos armazenados para –como o próprio nome indica – mantê-los armazenados no Servidor de Banco de Dados. Porém, eventualmente, pode ser conveniente excluir o procedimento. Para tanto, faz-se uso do comando DROP PROCEDURE.

```
mysql> DROP PROCEDURE BILHETERIAMEDIA;
Query OK, 0 rows affected (0.05 sec)
```

Observe que omitimos os parênteses que acompanham o nome do procedimento para a passagem de parâmetros.

Os procedimentos armazenados podem receber parâmetros de entrada ou de saída entre os parênteses após o seu nome.

```
mysql> DELIMITER //
mysql> CREATE PROCEDURE ANALISE_BILHETERIA(
    ->     OUT MAIOR_BIL NUMERIC(6,2),
    ->     OUT MENOR_BIL NUMERIC(6,2),
    ->     OUT MEDIA_BIL NUMERIC(6,2)
    -> )
    -> BEGIN
    ->     SELECT MIN(BILHETERIA)
    ->     INTO MENOR_BIL
    ->     FROM FILMES;

    ->     SELECT MAX(BILHETERIA)
    ->     INTO MAIOR_BIL
    ->     FROM FILMES;

    ->     SELECT AVG(BILHETERIA)
    ->     INTO MEDIA_BIL
    ->     FROM FILMES;
    -> END//
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> DELIMITER ;
```

O comando acima cria um procedimento armazenado (***stored procedure***) de nome ANALISE_BILHETERIA. Ao contrário do procedimento anterior, este recebe três parâmetros de saída (OUT): MAIOR__BIL, MENOR_BIL, e MEDIA_BIL. Todos esses parâmetros foram declarados do mesmo tipo que a coluna BILHETERIA em FILMES (**NUMERIC(6,2)**).

Esses parâmetros não levam valores para dentro do procedimento armazenado. Antes o contrário, esses parâmetros servem para levar para fora do procedimento armazenado os resultados gerados.

O operador INTO usado nas cláusulas SELECT armazena os resultados lidos nos respectivos parâmetros de saída. No comando "**SELECT MAX(BILHETERIA) INTO MAIOR_BIL**", por exemplo, o operador INTO armazena o valor encontrado da maior bilheteria no parâmetro MAIOR_BIL.

Antes de executar o procedimento armazenado por meio do comando CALL, devemos criar variáveis para armazenar seus resultados.

```
mysql> SET @MAIOR = 0;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> SET @MENOR = 0;
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> SET @MEDIA = 0;
Query OK, 0 rows affected (0.00 sec)
```

Nos comandos acima, foram criadas três variáveis: @MAIOR, @MENOR e @MEDIA. Todas foram inicializadas com valor zero. O caractere arroba ("@") utilizado no início de cada nome de variável é obrigatório.

```
mysql> CALL ANALISE_BILHETERIA(@MAIOR, @MENOR, @MEDIA);
Query OK, 0 rows affected, 1 warning (0.00 sec)
```

À primeira vista, a execução desse procedimento armazenado não parece produzir qualquer resultado. Se não fosse a mensagem de "Query Ok" gerada pelo MySQL, não teríamos como ter certeza de sua execução.

Para exibir o resultado, temos de executar outro comando:

```
mysql> SELECT @MAIOR AS MAIOR_BILHETERIA, @MENOR AS
MENOR_BILHETERIA, @MEDIA AS
BILHETERIA_MEDIA;
+------------------+------------------+------------------+
| MAIOR_BILHETERIA | MENOR_BILHETERIA | BILHETERIA_MEDIA |
|           990.90 |           390.89 |           594.36 |
1 row in set (0.00 sec)
```

Repare que o comando SELECT acima não busca os dados em tabelas (ausência da cláusula FROM), mas apenas nas variáveis que criamos e passamos ao procedimento armazenado como parâmetros de saída.

Sei que a execução desse procedimento pareceu algo muito complicado. No entanto, podemos suavizar o problema através da criação de um outro procedimento que faça uma chamada a este último.

```
mysql> DELIMITER //
mysql>
mysql> CREATE PROCEDURE ANALISE_RESULTADOS()
    -> BEGIN
    ->    SET @MAIOR = 0;
    ->    SET @MENOR = 0;
    ->    SET @MEDIA = 0;
    ->    CALL ANALISE_BILHETERIA(@MAIOR, @MENOR, @MEDIA);
    ->    SELECT @MAIOR AS MAIOR_BILHETERIA,
    ->     @MENOR AS MENOR_BILHETERIA,
    ->     @MEDIA AS BILHETERIA_MEDIA;
    -> END//
Query OK, 0 rows affected (0.00 sec)

mysql>
mysql> DELIMITER ;
```

Repare que toda a sequência de passos que era necessária para exibir os resultados foi encapsulada por um novo procedimento de nome ANALISE_RESULTADOS. É este procedimento que cria as variáveis de memória (@MAIOR, @MENOR e @MEDIA) e executa o procedimento ANALISE_BILHETERIA, para depois exibir os resultados gerados por este último procedimento ("**SELECT @MAIOR ...**").

```
mysql> CALL ANALISE_RESULTADOS();
+-----------------+-----------------+-----------------+
| MAIOR_BILHETERIA | MENOR_BILHETERIA | BILHETERIA_MEDIA |
+-----------------+-----------------+-----------------+
|           990.90 |           390.89 |           594.36 |
+-----------------+-----------------+-----------------+
1 row in set (0.00 sec)
Query OK, 0 rows affected, 1 warning (0.02 sec)
```

Agora, com a criação deste novo procedimento, o usuário final não precisa saber ou lembrar que variáveis devem ser criadas, ou mesmo a sequência correta de passagem dessas variáveis como parâmetros, ou ainda que é necessário executar um SELECT ao final de tudo. O outro procedimento se encarrega de tudo isso.

```
mysql> DELIMITER  //
mysql>
mysql> CREATE PROCEDURE BILHETERIA_ANO(
    ->   IN ANOP INT
    -> )
    -> BEGIN
    ->   SELECT ANO, MAX(BILHETERIA) AS
    ->          MAIOR_BILHETERIA,
    ->          MIN(BILHETERIA) AS MENOR_BILHETERIA,
    ->          AVG(BILHETERIA) AS MEDIA_BILHETERIA
    ->   FROM FILMES
    ->   WHERE ANO = ANOP;
    -> END//
Query OK, 0 rows affected (0.00 sec)

mysql> DELIMITER ;
```

O procedimento armazenado BILHETERIA_ANO recebe como parâmetro de entrada (IN) o ano de produção dos FILMES (ANOP) declarado como inteiro (INT). Observe que o parâmetro de entrada ANOP não serve para levar valores para fora do procedimento, mas apenas para trazer uma informação para dentro do procedimento.

```
mysql> CALL BILHETERIA_ANO(2009);
+------+-----------------+-----------------+-----------------+
| ANO  | MAIOR BILHETERIA | MENOR BILHETERIA | MEDIA BILHETERIA |
| 2009 |          990.90 |          391.00 |       660.296667 |
1 row in set (0.01 sec)
Query OK, 0 rows affected (0.01 sec)
```

O comando acima gera os resultados somente para o ano de 2009. Caso pretendamos exibir os resultados para o ano de 1972:

```
mysql> CALL BILHETERIA_ANO(1972);
+------+-----------------+-----------------+-----------------+
| ANO  | MAIOR BILHETERIA | MENOR BILHETERIA | MEDIA BILHETERIA |
| 1972 |          600.00 |          600.00 |       600.000000 |
1 row in set (0.00 sec)
```

Como existe apenas um filme com ANO de produção igual a 1972, os resultados para MAIOR_BILHETERIA, MENOR_BILHETERIA e MEDIA_BILHETERIA são idênticos (R$ 600.000,00).

Se quiser rever o conteúdo de um procedimento armazenado, basta usar o comando SHOW CREATE PROCEDURE:

```
mysql> SHOW CREATE PROCEDURE BILHETERIA_ANO;
CREATE DEFINER=`root`@`localhost` PROCEDURE
`BILHETERIA_ANO`(
     IN ANOP INT
)
BEGIN
   SELECT ANO, MAX(BILHETERIA) AS MAIOR_BILHETERIA,
          MIN(BILHETERIA) AS MENOR_BILHETERIA,
          AVG(BILHETERIA) AS MEDIA_BILHETERIA
   FROM FILMES
   WHERE ANO = ANOP;
END
1 row in set (0.00 sec)
```

Considere agora uma tabela chamada CONTA_CORRENTE, criada conforme o comando abaixo:

```
mysql> CREATE TABLE CONTA_CORRENTE (
    ->         NR_CONTA INTEGER NOT NULL,
    ->         ID_AGENCIA INTEGER NOT NULL,
    ->         ID_CLIENTE INTEGER NOT NULL,
    ->         SALDO NUMERIC(8,2) NOT NULL,
    ->
    ->         PRIMARY KEY(NR_CONTA)
    -> );
Query OK, 0 rows affected (0.09 sec)
```

Para transferir uma quantia de uma conta para outra, vamos criar um procedimento armazenado, conforme especificado logo abaixo.

```
mysql> DELIMITER //
mysql>
mysql> CREATE PROCEDURE TRANSFERENCIA (
    ->   IN CONTA_ORIGEM INT,
    ->   IN CONTA_DESTINO INT,
    ->   IN VALOR NUMERIC(8,2)

    -> BEGIN

    ->  DECLARE SALDO_ORIGEM NUMERIC(8,2);
    ->  DECLARE SALDO_DESTINO NUMERIC(8,2);

    ->  SELECT SALDO INTO SALDO_ORIGEM
    ->  FROM CONTA_CORRENTE
    ->  WHERE NR_CONTA = CONTA_ORIGEM;

    ->  IF SALDO_ORIGEM < VALOR THEN

    ->    SELECT "FALHA NA TRANSFERENCIA - SALDO
          INSUFICIENTE";

    ->  ELSE

    ->    SELECT "TRANSFERENCIA AUTORIZADA - SALDO
          SUFICIENTE";

    ->    UPDATE CONTA_CORRENTE SET SALDO =
                (SALDO_ORIGEM - VALOR)
    ->    WHERE NR_CONTA = CONTA_ORIGEM;

    ->    SELECT "VALOR R$ ",VALOR," RETIRADO DA CONTA
                ",CONTA_ORIGEM;

    ->    SELECT SALDO INTO SALDO_DESTINO
    ->    FROM CONTA_CORRENTE
    ->    WHERE NR_CONTA = CONTA_DESTINO;

    ->    UPDATE CONTA_CORRENTE SET SALDO =
                (SALDO_DESTINO + VALOR)
    ->    WHERE NR_CONTA = CONTA_DESTINO;

    ->    SELECT "VALOR R$ ",VALOR, " DEPOSITADO NA
                CONTA", CONTA_DESTINO;

    ->  END IF;
    -> END//
Query OK, 0 rows affected (0.16 sec)
mysql> DELIMITER ;
```

O procedimento armazenado acima recebe três parâmetros de entrada. Estes são: CONTA_ORIGEM (número da conta corrente de onde o valor será retirado), CONTA_DESTINO (número da conta corrente onde o valor será depositado) e VALOR (a quantia em dinheiro que será transferida de uma conta corrente para outra).

Duas variáveis de memória foram criadas: SALDO_ORIGEM e SALDO_DESTINO. Ambas foram definidas como sendo NUMERIC(8,2).

Em seguida, o procedimento armazenado executa uma consulta SQL (SELECT) em que a variável de memória SALDO_ORIGEM recebe o valor do SALDO da CONTA_CORRENTE em que NR_CONTA =

CONTA_ORIGEM.

Um comando condicional IF avalia se SALDO_ORIGEM é menor que VALOR. Caso essa condição seja verdadeira, uma mensagem é exibida, informando que ocorreu uma "**FALHA NA TRANSFERÊNCIA – SALDO INSUFICIENTE**". Por outro lado, se a condição for falsa, ou seja, se SALDO_ORIGEM for igual ou maior que o VALOR a ser transferido, uma outra mensagem informa que "**TRANSFERENCIA AUTORIZADA – SALDO SUFICIENTE**".

Se esta condição for falsa, o procedimento armazenado também altera o valor do SALDO da CONTA_CORRENTE em que NR_CONTA = CONTA_ORIGEM. Na verdade, o VALOR é retirado do SALDO ("**SET SALDO = (SALDO_ORIGEM – VALOR)**").

Então uma mensagem é exibida na tela informando que uma quantia igual a VALOR foi retirada da CONTA_CORRENTE de número igual a CONTA_ORIGEM.

A variável de memória SALDO_DESTINO é alimentada através de uma consulta SQL com o SALDO da CONTA_CORRENTE em que NR_CONTA = CONTA_DESTINO.

O SALDO da CONTA_CORRENTE em que NR_CONTA = CONTA_DESTINO é atualizado com o resultado da soma de SALDO_DESTINO com VALOR transferido.

Uma mensagem é exibida na tela para informar que o VALOR foi depositado na CONTA_CORRENTE em que NR_CONTA = CONTA_DESTINO.

Para testar esse procedimento armazenado, precisaremos antes inserir alguns registros na tabela CONTA_CORRENTE, conforme os comandos abaixo:

```
mysql> INSERT INTO CONTA_CORRENTE VALUES
(107,662,335,2900.80);
Query OK, 1 row affected (0.03 sec)

mysql> INSERT INTO CONTA_CORRENTE VALUES
(129,662,335,200.10);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO CONTA_CORRENTE VALUES
(108,662,343,55200.00);
Query OK, 1 row affected (0.00 sec)
```

Com esses comandos de inserção de registros, passamos a ter três contas correntes na tabela. Duas pertencem ao mesmo cliente.

```
mysql> SELECT * FROM CONTA_CORRENTE;
+----------+------------+------------+----------+
| NR CONTA | ID AGENCIA | ID CLIENTE | SALDO    |
|      107 |        662 |        335 |  2900.80 |
|      108 |        662 |        343 | 55200.00 |
|      129 |        662 |        335 |   200.10 |
3 rows in set (0.00 sec)
```

Para testar o procedimento armazenado, vamos tentar a transferência de R$ 300,00 da conta 107 (com SALDO de R$ 2.900,80) para a conta 129 (com SALDO de R$ 200,10).

```
mysql> CALL TRANSFERENCIA(107,129,300);
+-----------------------------------------+
| TRANSFERENCIA AUTORIZADA - SALDO SUFICIENTE |
| TRANSFERENCIA AUTORIZADA - SALDO SUFICIENTE |
1 row in set (0.06 sec)

+-----------+--------+---------------------+---------------+
| VALOR R$  | VALOR  | RETIRADO DA CONTA   | CONTA ORIGEM |
| VALOR R$  | 300.00 |  RETIRADO DA CONTA  |          107 |
1 row in set (0.08 sec)

+-----------+--------+----------------------+---------------+
| VALOR R$  | VALOR  | DEPOSITADO NA CONTA  | CONTA DESTINO |
| VALOR R$  | 300.00 |  DEPOSITADO NA CONTA |          129 |
1 row in set (0.09 sec)

Query OK, 0 rows affected (0.11 sec)
```

Como a consulta abaixo comprova, tudo transcorreu como esperávamos.

```
mysql> SELECT * FROM CONTA_CORRENTE;
+----------+------------+------------+----------+
| NR CONTA | ID AGENCIA | ID CLIENTE | SALDO    |
|      107 |        662 |        335 |  2600.80 |
|      108 |        662 |        343 | 55200.00 |
|      129 |        662 |        335 |   500.10 |
3 rows in set (0.00 sec)
```

A conta 107, após a retirada de R$ 300,00, ficou com SALDO de R$ 2.600,00. Por sua vez, a conta 129 teve seu SALDO alterado para R$ 500,10.

Ainda testando o procedimento armazenado TRANSFERENCIA, tentaremos transferir R$ 900,00 da conta 129 para a conta 107. Como o VALOR a ser transferido é superior ao existente na conta 129, a transferência não poderá ocorrer.

```
mysql> CALL TRANSFERENCIA(129,107,900);
+-----------------------------------------+
| FALHA NA TRANSFERENCIA - SALDO INSUFICIENTE |
| FALHA NA TRANSFERENCIA - SALDO INSUFICIENTE |
1 row in set (0.02 sec)
Query OK, 0 rows affected (0.02 sec)

mysql> SELECT * FROM CONTA_CORRENTE;
+----------+------------+------------+----------+
| NR CONTA | ID AGENCIA | ID CLIENTE | SALDO    |
|      107 |        662 |        335 |  2600.80 |
|      108 |        662 |        343 | 55200.00 |
|      129 |        662 |        335 |   500.10 |
3 rows in set (0.00 sec)
```

De fato, o procedimento armazenado TRANSFERENCIA não apenas exibiu uma mensagem informando que a transferência falhou por insuficiência de SALDO, como não permitiu alterações nos SALDOS das contas envolvidas.

**Atividade 30**

Crie um procedimento armazenado de nome CONSULTA_FUNCIONARIO, que receba como parâmetro de entrada uma MATRICULA e exiba os dados relativos ao funcionário com essa matrícula. Este procedimento armazenado deverá lidar com a visão FUNC criada na atividade anterior em vez da tabela FUNCIONARIO.

Conceitos

## 6.3 Gatilhos

Os gatilhos (***triggers***) se diferenciam dos procedimentos armazenados por não necessitarem do comando CALL para serem executados. Sua execução fica condicionada à ocorrência de um evento pré-definido. Sempre que esse evento ocorrer, o gatilho é "disparado" automaticamente.

Para exemplificarmos a utilização de um gatilho, vamos inicialmente criar duas novas tabelas em nosso banco de dados EXEMPLO. Elas são PRODUTO e VENDA.

```
mysql> CREATE TABLE PRODUTO (
      ID INTEGER NOT NULL PRIMARY KEY AUTO_INCREMENT,
      NOME VARCHAR(40) NOT NULL,
      ESTOQUE INTEGER NOT NULL,
      PRECO NUMERIC(6,2));
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE VENDA (
      NR_VENDA INTEGER NOT NULL PRIMARY KEY
         AUTO_INCREMENT,
      ID INTEGER NOT NULL,
      QUANTIDADE INTEGER NOT NULL,
      PRECO NUMERIC(6,2));
Query OK, 0 rows affected (0.02 sec)
```

Em seguida vamos inserir dois novos registros na tabela PRODUTO.

```
mysql> INSERT INTO PRODUTO VALUES (1,"DVD PLAYER",
40,299.99);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO PRODUTO VALUES (2,"APARELHO DE
TELEVISÃO LCD", 20,1429.99);
Query OK, 1 row affected (0.05 sec)
```

Assim, se fizermos uma rápida consulta à tabela PRODUTO, obteremos o seguinte resultado:

```
mysql> SELECT * FROM PRODUTO;
+----+---------------------------+---------+---------+
| ID | NOME                      | ESTOQUE | PRECO   |
|  1 | DVD PLAYER                |      40 |  299.99 |
|  2 | APARELHO DE TELEVISÃO LCD |      20 | 1429.99 |
2 rows in set (0.00 sec)
```

Note que no estoque da loja existem 40 aparelhos de DVD e 20 aparelhos de TV com tela LCD.

Antes de inserirmos registros na tabela VENDA, criaremos um gatilho (***trigger***) que será executado sempre que um novo registro de VENDA for inserido em nosso banco de dados.

```
mysql> DELIMITER //
mysql>
mysql> CREATE TRIGGER ATUALIZANDO_ESTOQUE
    -> AFTER INSERT ON VENDA
    -> FOR EACH ROW BEGIN
    ->     SET @VELHO_ESTOQUE = 0;
    ->     SELECT ESTOQUE INTO @VELHO_ESTOQUE
               FROM PRODUTO WHERE ID = NEW.ID;
    ->     UPDATE PRODUTO SET ESTOQUE =
               @VELHO_ESTOQUE - NEW.QUANTIDADE
    ->     WHERE ID = NEW.ID;
    -> END//
Query OK, 0 rows affected (0.06 sec)
mysql> DELIMITER ;
```

O gatilho (***trigger***) foi criado com o nome ATUALIZANDO_ESTOQUE ("**CREATE TRIGGER ATUALIZANDO_ESTOQUE**") e o evento que o dispara é a inserção de um novo registro na tabela VENDA ("**AFTER INSERT ON VENDA**").

Para cada linha ("**FOR EACH ROW**") inserida na tabela VENDA o gatilho cria uma variável chamada @VELHO_ESTOQUE. Essa variável é inicializada com zero, mas em seguida é atualizada a quantidade em estoque do produto antes da venda. **NEW.ID** corresponde ao valor da coluna ID inserido no novo ("NEW") registro de VENDA.

Em seguida, a coluna ESTOQUE da tabela PRODUTO é alterada com o resultado da subtração da quantidade vendida da quantidade em estoque do produto.

Agora vamos inserir um registro em VENDA e verificar as consequências dessa ação.

```
mysql> INSERT INTO VENDA VALUES (1,1,10,298.50);
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM VENDA;
+----------+----+------------+--------+
| NR_VENDA | ID | QUANTIDADE | PRECO  |
|        1 |  1 |         10 | 298.50 |
1 row in set (0.00 sec)

mysql> SELECT * FROM PRODUTO;
+----+--------------------------+---------+---------+
| ID | NOME                     | ESTOQUE | PRECO   |
|  1 | DVD PLAYER               |      30 |  299.99 |
|  2 | APARELHO DE TELEVISÃO LCD |      20 | 1429.99 |
2 rows in set (0.00 sec)
```

Observe que foram vendidos 10 aparelhos de DVD. Essa mesma quantidade foi descontada da coluna ESTOQUE em PRODUTO. Antes havia 40 aparelhos de DVD e agora restam apenas 30.

O interessante é que não precisamos alterar o estoque, pois o gatilho nos poupou dessa tarefa.

Para o nosso próximo exemplo de gatilho, precisamos criar uma tabela para armazenar alterações efetuadas em registros da tabela FILMES.

```
mysql> CREATE TABLE FILMES_LOG(
       NR_ENTRADA INTEGER NOT NULL PRIMARY KEY
          AUTO_INCREMENT,
       ID_VELHO INTEGER,
       ID_NOVO INTEGER,
       TITULO_VELHO VARCHAR(60),
       TITULO_NOVO VARCHAR(60),
       ANO_VELHO INTEGER,
       ANO_NOVO INTEGER,
       BILHETERIA_VELHA NUMERIC(6,2),
       BILHETERIA_NOVA NUMERIC(6,2));
Query OK, 0 rows affected (0.00 sec)
```

Assim, a tabela FILMES_LOG armazena os dados de antes da alteração do registro (ID_VELHO, TITULO_VELHO, ANO_VELHO e BILHETERIA_VELHA), assim como os dados após essa alteração (ID_NOVO, TITULO_NOVO, ANO_NOVO, BILHETERIA_NOVA).

```
mysql> DELIMITER //
mysql>
mysql> CREATE TRIGGER MANTER_LOG_ALTERACAO
    -> AFTER UPDATE ON FILMES
    -> FOR EACH ROW BEGIN
    ->     INSERT INTO FILMES_LOG (ID_VELHO,
              ID_NOVO, TITULO_VELHO, TITULO_NOVO,
              ANO_VELHO, ANO_NOVO, BILHETERIA_VELHA,
              BILHETERIA_NOVA)
           VALUES (OLD.ID, NEW.ID, OLD.TITULO,
              NEW.TITULO, OLD.ANO, NEW.ANO,
              OLD.BILHETERIA, NEW.BILHETERIA);
    -> END //
Query OK, 0 rows affected (0.00 sec)
mysql> DELIMITER ;
```

O gatilho acima dispara sempre que uma alteração é produzida em algum registro da tabela FILMES. Em consequência, ele insere um novo registro na tabela FILMES_LOG.

Repare que para obter o valor antigo de uma coluna utilizamos o prefixo "**OLD.**". Analogamente, para obter o novo valor de uma coluna utilizamos o prefixo "**NEW.**".

```
mysql> SELECT * FROM FILMES;
+----+--------------------------+------+------------+
| ID | TITULO                   | ANO  | BILHETERIA |
+----+--------------------------+------+------------+
|  1 | TRANSFORMERS             | 2007 |     390.89 |
|  2 | TRANSFORMERS 2           | 2009 |     990.90 |
|  3 | STAR TREK                | 2009 |     598.99 |
|  4 | X-MEN ORIGENS: WOLVERINE | 2009 |     391.00 |
|  5 | THE GODFATHER            | 1972 |     600.00 |
+----+--------------------------+------+------------+
5 rows in set (0.00 sec)

mysql> UPDATE FILMES SET TITULO = "O PODEROSO CHEFÃO",
ANO = 1973 WHERE ID = 5;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Note que alteramos simultaneamente o TITULO e ANO de produção do FILME com ID = 5.

```
mysql> SELECT * FROM FILMES;
+----+--------------------------+------+------------+
| ID | TITULO                   | ANO  | BILHETERIA |
+----+--------------------------+------+------------+
|  1 | TRANSFORMERS             | 2007 |     390.89 |
|  2 | TRANSFORMERS 2           | 2009 |     990.90 |
|  3 | STAR TREK                | 2009 |     598.99 |
|  4 | X-MEN ORIGENS: WOLVERINE | 2009 |     391.00 |
|  5 | O PODEROSO CHEFÃO        | 1973 |     600.00 |
+----+--------------------------+------+------------+
5 rows in set (0.00 sec)
```

A alteração foi bem-sucedida e o gatilho gerou um novo registro na tabela – criada de FILMES_LOG.

```
mysql> SELECT TITULO_NOVO, TITULO_VELHO, ANO_NOVO,
ANO_VELHO FROM FILMES_LOG;
+------------------+---------------+----------+-----------+
| TITULO NOVO      | TITULO VELHO  | ANO NOVO | ANO VELHO |
+------------------+---------------+----------+-----------+
| O PODEROSO CHEFÃO | THE GODFATHER |     1973 |      1972 |
+------------------+---------------+----------+-----------+
1 row in set (0.00 sec)
```

Um último exemplo de gatilho traz como novidade o evento antes da exclusão na tabela FILMES ("**BEFORE DELETE ON FILMES**"). Afinal, até aqui usávamos apenas "AFTER" (depois). Esta é a primeira vez que usamos "BEFORE" (antes).

```
mysql> DELIMITER //
mysql>
mysql> CREATE TRIGGER MANTER_LOG_EXCLUSAO
    -> BEFORE DELETE ON FILMES
    -> FOR EACH ROW BEGIN
    ->     INSERT INTO FILMES_LOG
             (ID_VELHO, TITULO_VELHO, ANO_VELHO,
              BILHETERIA_VELHA) VALUES (OLD.ID,
              OLD.TITULO, OLD.ANO,
              OLD.BILHETERIA);
    -> END //
Query OK, 0 rows affected (0.00 sec)
mysql>
mysql> DELIMITER ;
```

Imediatamente antes que uma exclusão de registro ocorra na tabela FILMES_LOG, o gatilho MANTER_LOG_EXCLUSAO é executado e um novo registro é inserido na tabela FILMES_LOG.

```
mysql> DELETE FROM FILMES WHERE ID = 3;
Query OK, 1 row affected (0.01 sec)

mysql>
mysql>
mysql> SELECT * FROM FILMES;
+----+--------------------------+------+------------+
| ID | TITULO                   | ANO  | BILHETERIA |
+----+--------------------------+------+------------+
|  1 | TRANSFORMERS             | 2007 |     390.89 |
|  2 | TRANSFORMERS 2           | 2009 |     990.90 |
|  4 | X-MEN ORIGENS: WOLVERINE | 2009 |     391.00 |
|  5 | O PODEROSO CHEFÃO        | 1973 |     600.00 |
+----+--------------------------+------+------------+
4 rows in set (0.00 sec)

mysql> SELECT ID_VELHO, TITULO_VELHO, ANO_VELHO,
BILHETERIA_VELHA FROM FILMES_LOG;
+----------+---------------+----------+------------------+
| ID VELHO | TITULO VELHO  | ANO VELHO | BILHETERIA VELHA |
+----------+---------------+----------+------------------+
|        5 | THE GODFATHER |     1972 |           600.00 |
|        3 | STAR TREK     |     2009 |           598.99 |
+----------+---------------+----------+------------------+
2 rows in set (0.02 sec)
```

Atividades

**Atividade 31**

Crie um gatilho (*trigger*) a ser disparado sempre que uma exclusão for efetuada na tabela FUNCIONARIO (a mesma da Atividade 29). O gatilho deverá armazenar os dados do registro excluído na tabela FUNCIONARIOS_EXCLUIDOS. Esta tabela deverá apresentar as seguintes colunas:

NR_EXCLUSAO integer not null primary key auto_increment,
DATA_EXCLUSAO date not null,
MATRICULA integer not null primary key,
NOME varchar(50) not null,
SEXO char(1) not null,
TELEFONE varchar(15),
SETOR integer not null,
SALARIO numeric(6,2), e
SENHA varchar(20).