BANCO DE DADOS

AULA 6

Prof. Lucas Rafael Filipak

CONVERSA INICIAL

Segue a apresentação da aula com a estrutura de conteudos que vão ser trabalhados nos seguintes tópicos:

1. *Stored procedures*
1.1 Criando *stored procedures*
2. *Functions*
3. *Triggers*
4. Registros *New* e *Old*
5. Estruturas de programação

O objetivo desta aula é entender o que são os procedimentos armazenados (*stored procedures*) e quais são as vantagens e desvantagens da sua utilização. Primeiramente é feita uma abordagem sobre os procedimentos, mostrando sua sintaxe e aplicação. O próximo assunto são as funções e, por último, os gatilhos (*triggers*).

Para finalizar a aula, são explicadas algumas estruturas de programação (condicionais e laço) que podem ser implementadas em um procedimento.

TEMA 1 – *STORED PROCEDURES*

Muitas aplicações que acessam um banco de dados executam rotinas de manipulação de dados a partir da linguagem ou ferramenta utilizada para criar as aplicações. Essas rotinas podem utilizar várias instruções SQLs em sequência. Se essa rotina executada possuir muitas consultas e atualizações no banco de dados, ela vai gerar um consumo maior dos recursos da aplicação. Caso a aplicação for *web*, o problema se torna um pouco maior, pois vai gerar maior tráfego de informações na rede e maiores requisições para o servidor.

Para tentar amenizar o consumo de recursos pela aplicação, é transferida parte do processamento (programação) para o banco de dados. Essas sub-rotinas que são programadas no banco de dados recebem o nome de *stored procedure*. Para Relminiak (2000), “executar a lógica do aplicativo diretamente no banco de dados tem a vantagem de que a quantidade de dados transferida entre o servidor de banco de dados e o cliente emitindo o comando SQL pode ser minimizada, e, simultaneamente, utilizando o poder total do servidor de banco de dados”.

Uma *stored procedure* ou um procedimento de armazenagem (em português) foi implementada a partir da versão 5.0 do MySQL e pode ser utilizada para fazer a validação de dados, executar instruções SQLs, controle de acesso, receber parâmetros, retornar valores etc.

Para exemplificar a utilização de uma *stored procedure* a seguir é feita uma comparação de uma execução de uma rotina utilizando e não utilizando uma *stored procedure*. Analise o contexto:

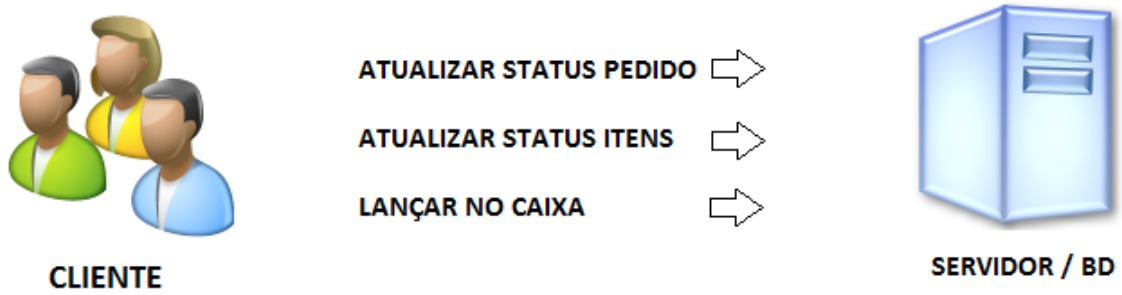
- O cliente realiza um pedido (que contém itens);
- O pedido precisa ser confirmado para sair do *status* de pendente;
- O usuário confirma o pedido, liberando para o cliente e registrando a transação no livro caixa.

O pedido deve ser confirmado para então liberar outros lançamentos no livro caixa, para isso é executado uma rotina de confirmação de pedido:

- Atualizar o *status* do pedido (de pendente para finalizado);
- Atualizar os *status* dos itens do pedido;
- Lançar as informações do pedido no livro caixa.

Para executar essa rotina, existem pelo menos três instruções SQLs inserção ou atualização de dados que estão representadas na Figura 1.

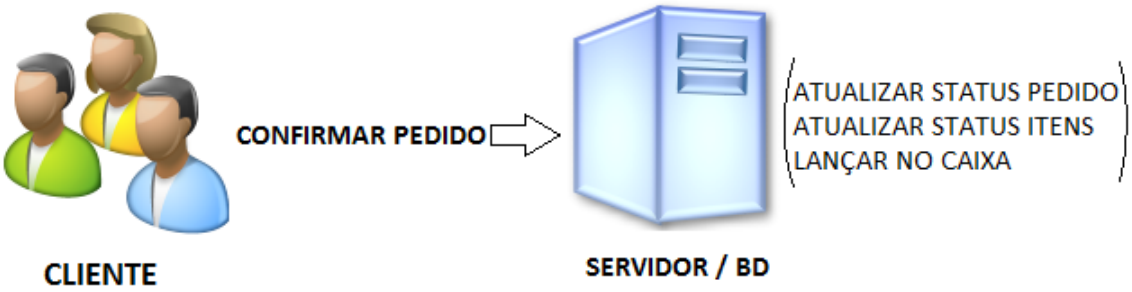
Figura 1 – Ilustração da execução de uma rotina sem *stored procedure*



Fonte: Rodrigues, 2016b.

Por outro lado, é possível criar um procedimento agrupando essas instruções ficando a cargo da aplicação apenas fazer a chamada desse procedimento. As ações realizadas (*update* e *insert*) ficam por conta do banco de dados (servidor). Na Figura 2 foi criada uma *stored procedure* chamada *CONFIRMAR PEDIDO.* Analise a Figura 2.

Figura 2 – Ilustração da execução de uma rotina com *stored procedure*



Fonte: Rodrigues, 2016b.

Após o entendimento da utilização de um procedimento de armazenagem, Rodrigues (2016b) destaca alguns pontos positivos e negativos da sua utilização:

Pontos positivos:

- Simplificação da execução de instruções SQL pela aplicação;
- Transferência de parte da responsabilidade de processamento para o servidor.
- Facilidade na manutenção, reduzindo a quantidade de alterações na aplicação.

Pontos negativos:

- Necessidade de maior conhecimento da sintaxe do banco de dados para escrita de rotinas em SQL;
- As rotinas ficam mais facilmente acessíveis. Alguém que tenha acesso ao banco poderá visualizar e alterar o código.

Um comando que pode ser utilizado em *stored procedures, functions* e *triggers* é o DELIMITER. Ele é utilizado para trocar o caractere de finalização. O mysql tem como delimitador o **;** (ponto e vírgula), ou seja, o mysql entende que ali o comando está encerrado e não há necessidade de continuidade. Ao utilizar o DELIMITER, você diz ao mysql onde seu *script* tem início, e onde encerra. Repare que na Figura 3 o DELIMITER foi alterado para $$.

Figura 3 – Exemplo do comando DELIMITER

```
DELIMITER $$
— seu objeto 1 aqui
$$
— seu objeto 2 aqui
$$
```

Para os exemplos e explicações sobre *stored procedures, functions* e *triggers* foram utilizadas o cenário de uma loja de livros, onde existem as seguintes tabelas:

- Clientes;
- Pedidos;
- Itens_venda;
- Livros.

1.1 Criando *stored procedures*

Para começar a utilizar as *stored procedures*, é importante entender que sua utilização é composta de duas partes: a criação da *procedure* e a sua chamada. A Figura 4 mostra a sintaxe da criação de uma *procedure*:

Figura 4 – Exemplo da sintaxe do comando CREATE PROCEDURE

```
CREATE PROCEDURE nome_procedure (parâmetros)
declarações;
```

Toda *procedure* deve ter um nome que será utilizado para fazer a sua chamada. Os parâmetros são opcionais. A Figura 5 exemplifica a criação de uma procedure sem parâmetros.

Figura 5 – Exemplo do comando para criação de uma *procedure*

```
CREATE  PROCEDURE ExibeNúmeroDePedidos
SELECT  C.id-cliente, C.nome-cliente, COUNT (*)
        FROM          Clientes C, Pedidos P
        WHERE         C.id-cliente = P.id-cliente
        GROUP BY      C.id-cliente, C.nome-cliente
```

Fonte: Ramarkrishnan; Gehrke, 2011, p. 174.

Repare na Figura 5 que a *procedure* criada possui o nome *ExibeNúmeroDePedidos*. Esse é o nome utilizado para fazer a chamada da *procedure*. Observe na Figura 6 a sintaxe da chamada de uma *procedure*.

Figura 6 – Exemplo da sintaxe do comando que chama uma *procedure*

```
CALL nome_procedure (parâmetros);
```

A chamada de uma *procedure* pode ser executada quantas vezes forem necessárias. A Figura 7 representa a chamada da *procedure* criada na Figura 5.

Figura 7 – Exemplo do comando que chama uma procedure

```
CALL ExibeNumeroDePedidos();
```

Na Figura 7 foi realizado a chamada da *procedure* ExibeNumeroDePedidos em que não havia parâmetros. Para não ficar dúvida, a Figura 8 traz um exemplo de criação e chamada de uma *procedure* com parâmetros.

Figura 8 – Exemplo do comando que cria uma procedure com parâmetros

```
DELIMITER $$
CREATE PROCEDURE valorPedido (varPedido smallint)
SELECT CONCAT (Valor_final , ' é o total do pedido ', varPedido ) AS Valor_total
FROM Pedidos
WHERE id_pedido = varPedido;

END $$
CALL valorPedido(3);
```

Note que a *procedure* valorPedido criada na Figura 8 possui um parâmetro (varPedido), que foi utilizado na sua chamada. Os parâmetros são classificados em três modos diferentes: IN, OUT ou INOUT.

- IN → é utilizado apenas para recebimento (entrada) de dados e não é utilizado para dar retorno;
- OUT → é um parâmetro de saída, não sendo informado um valor fixo (direto), apenas uma variável para o retorno;
- INOUT → esse modo de parâmetro pode ser utilizado como entrada ou saída, não podendo ser informado um valor fixo.

Figura 9 – Exemplo do comando que cria um *procedure* com parâmetros do modo IN

```
CREATE PROCEDURE AcrescInventário (
                IN livro_isbn CHAR(10),
                IN qtidadeAcresc INTEGER)
UPDATE Livros
        SET     qtidade_em_estoque = qtidade_em_estoque + qtidadeAcresc
        WHERE   livro_isbn = isbn
```

Fonte: Ramarkrishnan; Gehrke, 2011, p. 174.

Para finalizar os procedimentos armazenados, é importante salientar que eles não necessariamente são escritos em SQL. Ramarkrishnan e Gehrke (2011, p. 174) dizem que “os procedimentos armazenados não precisam ser escritos em SQL; eles podem ser escritos em qualquer linguagem hospedeira”. Observe a Figura 10.

Figura 10 – Exemplo do comando que cria um procedure armazenado em Java

```
CREATE PROCEDURE ListaClientes(IN número INTEGER)
LANGUAGE Java
EXTERNAL NAME 'file:///c:/storedProcedures/rank.jar'
```

Fonte: Ramarkrishnan; Gehrke, 2011, p. 175.

O exemplo da Figura 10 se trata de uma função em Java que é executada dinamicamente pelo banco dados cada vez que o cliente faz a chamada da procedure.

Para apagar uma *procedure*, utiliza-se o comando da Figura 11.

Figura 11 – Exemplo do comando utilizado para apagar uma *procedure*

```
DROP PROCEDURE nome_procedure;
```

TEMA 2 – *FUNCTIONS*

Segundo Puga, França e Goya (2013, p. 280), “as funções são muito semelhantes aos procedimentos, o que os difere, do ponto de vista estrutural, é a inclusão da cláusula RETURN. Nas funções, existe a obrigatoriedade de um retorno à rotina chamadora, que é feito por meio da cláusula RETURN.”.

O MySQL possui diversas *functions* (em português, *funções*) pré-programadas (internas) que o desenvolvedor pode utilizar, mas também permite que seja criado novas *functions*. A Figura 12 representa a sintaxe para a criação de uma *function*.

Figura 12 – Exemplo do comando para criar uma *function*

```
CREATE FUNCTION nome_funcao (parâmetros)
RETURNS tipo_dados
código_da_função
```

A chamada da *function* segue a mesma ideia da chamada das *procedures*, mas é utilizado o comando SELECT em vez do comando CALL (*procedures*). Observe na Figura 13.

Figura 13 – Exemplo da sintaxe de chamada de uma *function*

```
SELECT nome_funcao (parâmetros);
```

Para exemplificar a utilização das *functions* foi criada na Figura 14 uma função que recebe dois valores numéricos e retorna o resultado da multiplicação entre eles.

Figura 14 – Exemplo do comando para a criação de uma *function*

```
CREATE FUNCTION fn_teste (a DECIMAL(10,2), b INT)
RETURNS INT
RETURN a * b;
```

Fonte: Reis, 2014a.

A *function* criada na Figura 14 recebeu o nome de fn_teste e possui dois parâmetros o primeiro “a” que é do tipo decimal e o segundo “b” que é do tipo inteiro. Observe que o retorno é definido com o tipo inteiro e é a multiplicação do primeiro pelo segundo parâmetro. A Figura 15 faz a chamada da função.

Figura 15 – Exemplo do comando para fazer a chamada de uma *function*

```
SELECT fn_teste(2.5, 4) AS Resultado;
```

Fonte: Reis, 2014a.

A *function* fn_teste foi chamada passando o valor 2.5 como primeiro parâmetro e o valor 4 como segundo parâmetro. Apenas para melhorar a visualização, a coluna que mostrará o resultado 10 (que foi o retorno da função 2.5 * 4) foi apelidada de “Resultado”.

A Figura 14 representa uma *function* simples. Mesmo com 2 parâmetros a *function* simplesmente fazia um cálculo matemático. A Figura 16 traz a criação de uma função um pouco mais elaborada, utilizando uma instrução SQL.

Figura 16 – Exemplo do comando de criação de uma function com parâmetros

```
CREATE FUNCTION verPreço (a SMALINT)
RETURNS VARCHAR(60)
RETURN
(SELECT CONCAT (‘O preço do livro ’ , Nome_Livro,
‘ é ’, Preco_livro)
FROM Livros
WHERE id_livro = a ;
```

Fonte: Reis, 2014a.

A *function* verPreço possui apenas um parâmetro. A *function* recebe o parâmetro “a” (código de um livro) e faz um SELECT na tabela *Livros*, trazendo como resultado uma mensagem no seguinte padrão: “O preço do livro **nome_do_livro** é R$ **preco_do_livro**”.

Note que o retorno da *function* é do tipo VARCHAR com tamanho 60. Para finalizar o exemplo, a Figura 17 faz a chamada da *function*.

Figura 17 – Exemplo do comando que faz a chamada da *function* verPreço

```
SELECT verPreço (9);
```

Fonte: Reis, 2014a.

Na chamada da *function* foi passado o número 9 como valor do parâmetro. A *function* utiliza esse parâmetro e faz a seleção no registro que possui o código 9. Também é possível apagar uma função já programada, conforme se vê na Figura 18.

Figura 18 – Exemplo do comando que apaga uma *function*

```
DROP FUNCTION nome_funcao;
```

TEMA 3 – *TRIGGERS*

Algumas ações podem ser disparadas (executadas) em consequência ou resposta de uma ação. Esses disparos podem ser programados na aplicação ou diretamente no banco de dados. A utilização direta no banco de dados chama-se *triggers*. Para Puga, França e Goya (2013, p. 286), os “Gatilhos, ou *triggers*, são

blocos PL/SQL acionados ou disparados automaticamente, antes ou depois da ocorrência de um evento, que pode ser uma operação DML para inserção, alteração ou exclusão de dados ou um evento associado ao banco, como efetuar o *logon* do usuário.”.

Em minhas palavras, um *trigger* é um objeto do banco de dados que sempre está associado a uma tabela e é disparado automaticamente antes ou depois de um evento DML. Podem-se definir vários *triggers* em um banco de dados, mas cada comando pode disparar apenas um *trigger*. Rodrigues (2016a) destaca os pontos positivos e negativos da utilização dos *triggers*:

Pontos positivos:

- Parte do processamento que seria executado na aplicação passa para o banco, poupando recursos da máquina cliente;
- Facilita a manutenção, sem que seja necessário alterar o código fonte da aplicação (Rodrigues, 2016a).

Pontos negativos:

- Alguém que tenha acesso não autorizado ao banco de dados poderá visualizar e alterar o processamento realizado pelos gatilhos;
- Requer maior conhecimento de manipulação do banco de dados (SQL) para realizar as operações internamente (Rodrigues, 2016a).

Observe na Figura 19 a sintaxe de criação do *trigger*.

Figura 19 – Exemplo da sintaxe da criação do *trigger*

```
CREATE TRIGGER nome momento evento
ON tabela
FOR EACH ROW
BEGIN
/*corpo do código*/
END
```

Fonte: Rodrigues, 2016a

Lembre-se que a linguagem SQL não difere palavras escritas em maiúsculo ou minúsculo. As maiúsculas são utilizadas para identificar as partes que não variáveis da sintaxe. Os componentes variáveis da sintaxe são:

- nome → nome da *trigger*;
- momento → quando a *trigger* vai ser executada: BEFORE (antes) ou AFTER (depois);

- evento → qual o comando que vai fazer a *trigger* disparar: INSERT, UPDATE, DELETE ou REPLACE;
- tabela → é a tabela que o trigger está associado.

Lembre-se de que não é possível mais de um *trigger* para o mesmo evento e momento na mesma tabela. Como exemplo, é possível apenas um *trigger* BEFORE UPDATE na mesma tabela. Para facilitar a visualização foi criado na Figura 20 um comando com a sintaxe da criação do *trigger* com os possíveis valores.

Figura 20 – Exemplo da sintaxe da criação do *trigger* com todas as opções

```
CREATE TRIGGER nome AFTER/BEFORE INSERT/UPDATE/DELETE
ON tabela
FOR EACH ROW
BEGIN
  Instruções SQL
END
```

Agora que foi compreendida a sintaxe para criação, é criado um exemplo de utilização do *trigger*. Observe a Figura 21.

Figura 21 – Exemplo do comando CREATE TRIGGER

```
CREATE TRIGGER tr_desconto BEFORE INSERT
ON Livros
FOR EACH ROW
SET NEW.preco_desconto = (NEW.preco_produto * 0.90);
```

A Figura 21 representa a criação de um *trigger* que vai ser disparado antes (BEFORE) que um registro for inserido (INSERT) na tabela "*Livros*". O *trigger* vai mover o cálculo de 10% de desconto (*preco_produto* * 0.90) para a coluna *preco_desconto*. Toda vez que for inserido um registro na tabela *Livros*, o *trigger* é disparado e a coluna *preco_desconto* recebe o valor calculado.

TEMA 4 – REGISTROS *NEW* E *OLD*

As palavras *NEW* e *OLD* são utilizadas para acessar os registros antes ou depois da execução. Como exemplo pode-se acessar os registros que serão enviados para uma tabela antes (BEFORE) ou depois (AFTER) de um UPDATE. Bianchi (2008) explica os registros NEW e OLD em cada comando DML:

- INSERT: o operador NEW.nome_coluna, nos permite verificar o valor enviado para ser inserido em uma coluna de uma tabela. OLD.nome_coluna não está disponível.
- DELETE: o operador OLD.nome_coluna nos permite verificar o valor excluído ou a ser excluído. NEW.nome_coluna não está disponível.
- UPDATE: tanto OLD.nome_coluna quanto NEW.nome_coluna estão disponíveis, antes (BEFORE) ou depois (AFTER) da atualização de uma linha.

Como foi visto a palavra *NEW* acessa o novo registro após a execução de uma instrução. O operador OLD acessa o registro antigo. A Figura 22 exemplifica duas *triggers* utilizando os operadores NEW e OLD.

Figura 22 – Exemplo do comando CREARE TRIGGER com os operadores NEW e OLD

```
DELIMITER $

CREATE TRIGGER Tgr_ItensVenda_Insert AFTER INSERT
ON ItensVenda
FOR EACH ROW
BEGIN
    UPDATE Produtos SET Estoque = Estoque - NEW.Quantidade
WHERE Referencia = NEW.Produto;
END$

CREATE TRIGGER Tgr_ItensVenda_Delete AFTER DELETE
ON ItensVenda
FOR EACH ROW
BEGIN
    UPDATE Produtos SET Estoque = Estoque + OLD.Quantidade
WHERE Referencia = OLD.Produto;
END$

DELIMITER ;
```

Fonte: Rodrigues, 2016a.

Na Figura 22 foram criados dois *triggers*. O Tgr_ItensVenda_Insert é executado depois de cada INSERT na tabela *ItensVenda* e vai atualizar a tabela *Produtos* com a atualização da coluna *estoque*. O operador NEW é utilizado duas

vezes, uma na coluna *quantidade* outro na coluna *produto*. Lembre-se que no INSERT não há como utilizar o operador OLD.

Analisando o *trigger* Tgr_ItensVenda_Delete percebe-se que ele é ativado depois que uma instrução DELETE é executada na tabela *ItensVenda*. O operador OLD foi utilizado em duas colunas, atualizando a coluna *estoque* da tabela *Produtos* com o valor antigo da coluna *quantidade*. Com a instrução não é possível utilizar o operador NEW.

Vamos atentar para a utilização do comando DELIMITER na Figura 22. Por padrão, o **;** (ponto e vírgula) faz o encerramento, mas observe na Figura 22 que no meio do código há duas ocorrências de ; e nenhuma delas é no final do código. O código acaba no caractere que foi definido no DEMILITER.

Pode-se visualizar a lista de *triggers* com o comando SHOW TRIGGERS. Uma *trigger* também pode ser apagada. Observa a Figura 23.

Figura 23 – Exemplo do comando para apagar uma *trigger*

```
DROP TRIGGER nome_da_trigger;
```

Para finalizar a utilização das *triggers* ou gatilhos tem algumas limitações. Bianchi (2008) lista algumas:

- Não se pode chamar diretamente um TRIGGER com CALL, como se faz com um *stored procedures*;
- Não é permitido iniciar ou finalizar transações em meio à TRIGGERS;
- Não se pode criar um TRIGGERS para uma tabela temporária – TEMPORARY TABLE;
- TRIGGERS ainda não podem ser implementadas com a intenção de devolver para o usuário ou para uma aplicação mensagens de erros.

TEMA 5 – ESTRUTURAS DE PROGRAMAÇÃO

É possível que as *stored procedures* tenham incorporado dentro dos seus procedimentos algumas estruturas de programação como condicionais, laços de repetição, funções etc. Esse fato implica programas menores, exigindo menos do lado do cliente, mas em contrapartida exigindo mais processamento do SGBD. A Figura 24 traz um exemplo de uma sintaxe de uma condição.

Figura 24 – Exemplo da sintaxe da condicional IF

```
IF <condição> THEN
  comandos sql caso verdadeiro
ELSE
  comandos sql caso falso
END IF
```

O comando IF é uma estrutura condicional onde é testada uma condição e, se ela for verdadeira, é executado um bloco de comandos e, se a condição for falsa, é executado outro bloco de comandos. Dentro de uma *procedure* os comandos seguem a mesma dinâmica se eles estivessem escritos na linguagem de programação. A tabela da Figura 25 é utilizada no exemplo da Figura 26.

Figura 25 – Tabela *Cliente* utilizada como exemplo

codigo	nome	sexo
1	Pedro	M
2	Caio	M
3	Maria	F

Com base na tabela *Cliente* da Figura 25 é criado uma *procedure* que liste os clientes masculinos, femininos ou ambos dependendo do parâmetro passado na chamada da procedure. Como existem 3 possíveis resultados para serem apresentados (masculino, feminino ou ambos), não é possível fazer uma condicional simples como na Figura 24.

Figura 26 – Exemplo do comando que cria uma *procedure* com programação condicional

```
DELIMITER //
CREATE PROCEDURE lista_clientes (IN opcao integer)

BEGIN
   IF opcao = 0 THEN
      SELECT * FROM clientes where sexo = F;
   ELSE
      IF opcao = 1 THEN
         SELECT * FROM clientes WHERE sexo = M;
      ELSE
         SELECT * FROM clientes;
      END IF;
   END IF;
END//
```

Na criação da procedure foi informado que na chamada será preciso passar um parâmetro, do tipo inteiro e que internamente na procedure será armazenado dentro de *opcao*. Observe as chamadas na Figura 27.

Figura 27 – Exemplos de comandos que fazem a chamada da *procedure* lista_clientes

```
CALL lista_clientes(0);

CALL lista_clientes(1);

CALL lista_clientes(2);
```

A Figura 27 traz o exemplo de 3 chamadas da *procedure* lista_clientes, mas note que em cada chamada é passado um parâmetro diferente. Na primeira chamada é passado o parâmetro 0 e a *procedure* retorna todos os clientes que são do sexo feminino. Na segunda é passado o parâmetro 1 e a *procedure* retorna todos os clientes que são do sexo masculino. Agora se atente a terceira chamada onde é passado o parâmetro 2. A *procedure* vai retornar todos os clientes (masculinos e femininos), pois na condicional dentro da *procedure* essa opção entra no segundo ELSE (qualquer valor diferente de 0 e 1).

Para finalizar esta aula, será demonstrada a utilização de um laço de repetição dentro de uma *procedure*. Analise a Figura 28.

Figura 28 – Exemplo de *procedures* com laços de repetição

```
DELIMITER //
CREATE PROCEDURE acumulador (limite TINYINT UNSIGNED)
BEGIN
  DECLARE contador TINYINT UNSIGNED DEFAULT 0;
  DECLARE soma INT DEFAULT 0;
  WHILE contador < limite DO
    SET contador = contador + 1;
    SET soma = soma + contador;
  END WHILE;
  SELECT soma;
END//
```

A *procedure acumulador* (representada na Figura 28) tem um laço de repetição e a declaração de duas variáveis. O parâmetro *limite* vai ser informado na chamada da *procedure*. Foi declarado a variável contador do tipo TINYINT e a variável soma do tipo INT. As duas foram inicializadas com o valor 0. Repare na Figura 29 onde é feita a chamada dessa *procedure*.

Figura 29 – Exemplo do comando que faz a chamada da *procedure acumulador*

```
CALL acumulador(10);
```

Na chamada da *procedure*, Figura 29, foi passado o valor 10 como parâmetro. A variável interna limite recebe o valor 10 que é utilizado na condição do WHILE. Em cada repetição o contador é incrementado e a variável soma recebe o valor atual dela + o valor do contador até que a condição se torne falsa. Ao final o comando SELECT mostra o valor da variável soma.

FINALIZANDO

Nesta aula foi estudado sobre os procedimentos armazenados. Foi vista a sua utilização, vantagens e desvantagens. Ao final da aula foram vistas também algumas estruturas de programação (condicionais e laços) que podem ser programadas diretamente no SGBD. Foi estudado sobre as funções, que podem ou não ter parâmetros, mas sempre devem retornar alguma informação. As *triggers* (ou gatilhos) que são ações automáticas que são disparadas antes (BEFORE) ou depois (AFTER) de uma ação.

REFERÊNCIAS

BIANCHI, W. MySQL – Triggers. **Devmedia**, 2008. Disponível em: <https://www.devmedia.com.br/mysql-triggers/8088>. Acesso em: 5 set. 2019.

PUGA, S.; FRANÇA, E.; GOYA, M. **Banco de dados**: implementação em SQL, PL/SQL, Oracle 11g. São Paulo: Pearson Education do Brasil, 2013.

RAMARKRISHNAN, R.; GEHRKE, J. Sistemas de gerenciamento de banco de dados. Tradução: Célia Taniwake. 3. ed. Porto Alegre: AMGH, 2011.

REIS, F. dos. MySQL – Procedimentos armazenados. **Bóson treinamentos em tecnologia**, 13 fev. 2014a. Disponível em: <http://www.bosontreinamentos.com.br/mysql/mysql-procedimentos-armazenados-stored-procedures-basico-34/>. Acesso em: 5 set. 2019.

_____. MySQL – Rotinas armazenadas – Funções (CREATE FUNCTION). **Bóson treinamentos em tecnologia**, 13 fev. 2014b. Disponível em: <http://www.bosontreinamentos.com.br/mysql/mysql-rotinas-armazenadas-funcoes-create-function-33/>. Acesso em: 5 set. 2019.

RODRIGUES, J. MySQL básico: *triggers*. **Devmedia**, 2016a Disponível em: <https://www.devmedia.com.br/mysql-basico-triggers/37462>. Acesso em: 5 set. 2019.

_____. *Stored procedures* no MySQL. **Devmedia**, 2016b. Disponível em: <https://www.devmedia.com.br/stored-procedures-no-mysql/29030>. Acesso em: 5 set. 2019.