Laboratório de Banco de Dados

SQL Avançado

Rotinas Armazenadas e Gatilhos

UFMT

Julho de 2025

Agenda

- **Rotinas armazenadas: Motivação, diferenças, vantagens e desvantagens.**
- Funções armazenadas: declaração, chamada e exemplos.
- Procedimentos armazenados: declaração, chamada e exemplos.
- Construções de linguagem para funções e procedimentos armazenados.
- Chamadas de funções e procedimentos armazenados em Python.
- Chamadas de funções e procedimentos armazenados em Java.
- Gatilhos.

Introdução

- Funções e os Procedimentos Armazenados são funções e procedimentos armazenados e executados pelo SGBD, com o objetivo de desempenhar operação com o BD.
- **Vantagens**:
  - Facilitam o compartilhamento de código, uma vez que podem ser acessados por mais de uma aplicação escritas em linguagens diferentes ou que funcionam em plataformas diferentes, mas que precisam executar as mesmas operações no banco de dados.
  - Melhoram a segurança, pois limitam o acesso direto às tabelas do banco de dados.
  - Um único ponto de manutenção em caso de alterações na lógica da aplicação.
- **Desvantagens:**
  - Teste e depuração pode ser difícil, dependendo do SGBD. Performance deve ser analisada para decidir sobre a viabilidade.
  - Em geral, não são portáveis entre diferentes SGBDs.

Diferenças Básicas com Foco no MySQL

Funções armazenadas

- Recebem apenas parâmetros de entrada.
- Devem retornar um valor (escalar no mysql) via RETURN.
- Não podem chamar procedimentos armazenados.
- São usadas em comandos SQL select, insert, update, delete.
- Correntemente não é possível definir operações transacionais.

Procedimentos armazenados

- Recebem parâmetros de entrada e de saída.
- Não retornam valor via RETURN.
- Podem chamar funções e outros procedimentos armazenados.
- Não podem ser usados em comandos SQL select, insert, update, delete.
- Podem definir operações transacionais com `commit` e `rollback`.

Mais Detalhes sobre as Diferenças no MySQL

- As restrições associadas às rotinas armazenadas podem ser encontradas em:
  - https://starcat.dp.ua/doc/mysql-5.0/restrictions.html#routine-restrictions
- Perguntas frequentes sobre funções e procedimentos armazenados em MySQL podem ser encontradas no seguinte endereço:
  - https://starcat.dp.ua/doc/mysql-5.0/faqs.html#qandaitem-26-4-9
- Um fórum de discussão que pode ser útil:
  - https://forums.mysql.com/list.php?98

Agenda

- Rotinas armazenadas: Motivação, diferenças, vantagens e desvantagens.
- **Funções armazenadas: declaração, chamada e exemplos.**
- Procedimentos armazenados: declaração, chamada e exemplos.
- Construções de linguagem para funções e procedimentos armazenados.
- Chamadas de funções e procedimentos armazenados em Python.
- Chamadas de funções e procedimentos armazenados em Java.
- Gatilhos.

Visão Geral Sobre a Criação de Funções Armazenadas

- Funções armazenados podem ser definidas diretamente em SQL ou externamente usando uma linguagem de propósito geral, tais como Java, Python ou C++.
- Embora os conceitos apresentados aqui sejam bastante gerais, a sintaxe apresentada não é necessariamente a adotada por todos os SGBDs. Portanto, consulte sempre o manual do SGBD adotado por você.
- Os exemplos a seguir consideram o SGBD MySQL, o qual segue a sintaxe SQL:2003 para rotinas armazenadas.

Comando `CREATE FUNCTION`

- Funções armazenadas são definidas com o comando `CREATE FUNCTION`. Elas podem receber um ou mais parâmetros de entrada e retornar um valor via comando RETURN (um escalar no caso do MySQL).
- **Sintaxe geral**:

```
CREATE FUNCTION nome_func (par1: tipo, par2: tipo,...) RETURNS tipo
        corpo_da_função;
```

  - nome_func: nome da função.
  - par1, par2: parâmetros de entrada, cada qual seguido por seu tipo.
  - corpo_da_função: definido em um bloco `begin-end`, pode incluir variáveis locais definidas no começo do bloco com o comando `declare`, bem como comandos SQL e de linguagem que não alteram o banco de dados (por padrão, para serem determinísticas).

Detalhes Sobre Funções Armazenada no MySQL

- As condições para o uso de funções armazenadas no MySQL podem ser resumidas da seguinte forma.

- Ao criar uma função armazenada, você deve declará-la como determinística (DETERMINISTIC) ou como não modificadora de dados (NO SQL ou READS SQL DATA). Caso contrário ela pode ser insegura para replicação de dados (não deteminística).

- A avaliação da natureza de uma função é feita pelo criador da função. O MySQL não verifica se uma função declarada como DETERMINISTIC produz resultados não determinísticos.

Detalhes Sobre Funções Armazenada no MySQL

- Para relaxar as condições de criação de funções armazenadas, é possível definir a variável de sistema `log_bin_trust_function_creators` para 1.

- Por padrão, esta variável é 0 , mas você pode alterá-la usando o comando `SET GLOBAL`:

```
SET GLOBAL log_bin_trust_function_creators = 1;
```

- Isso faz com que o SGBD aceite a criação de funções sem que seja assegurado pelo programador que elas sejam determinísticas.

Exemplo de criação de função armazenada no MySQL

- Conta o número de professores de um dado departamento.

```
create function dep_count(nome_dept varchar(20))
returns integer reads SQL data
begin
  declare qtd integer;
  select count(*) into qtd
  from Professor as P
  where P.nome_dept = nome_dept;
  return qtd
end
```

- SELECT armazena o resultado diretamente na variável local `qtd`.

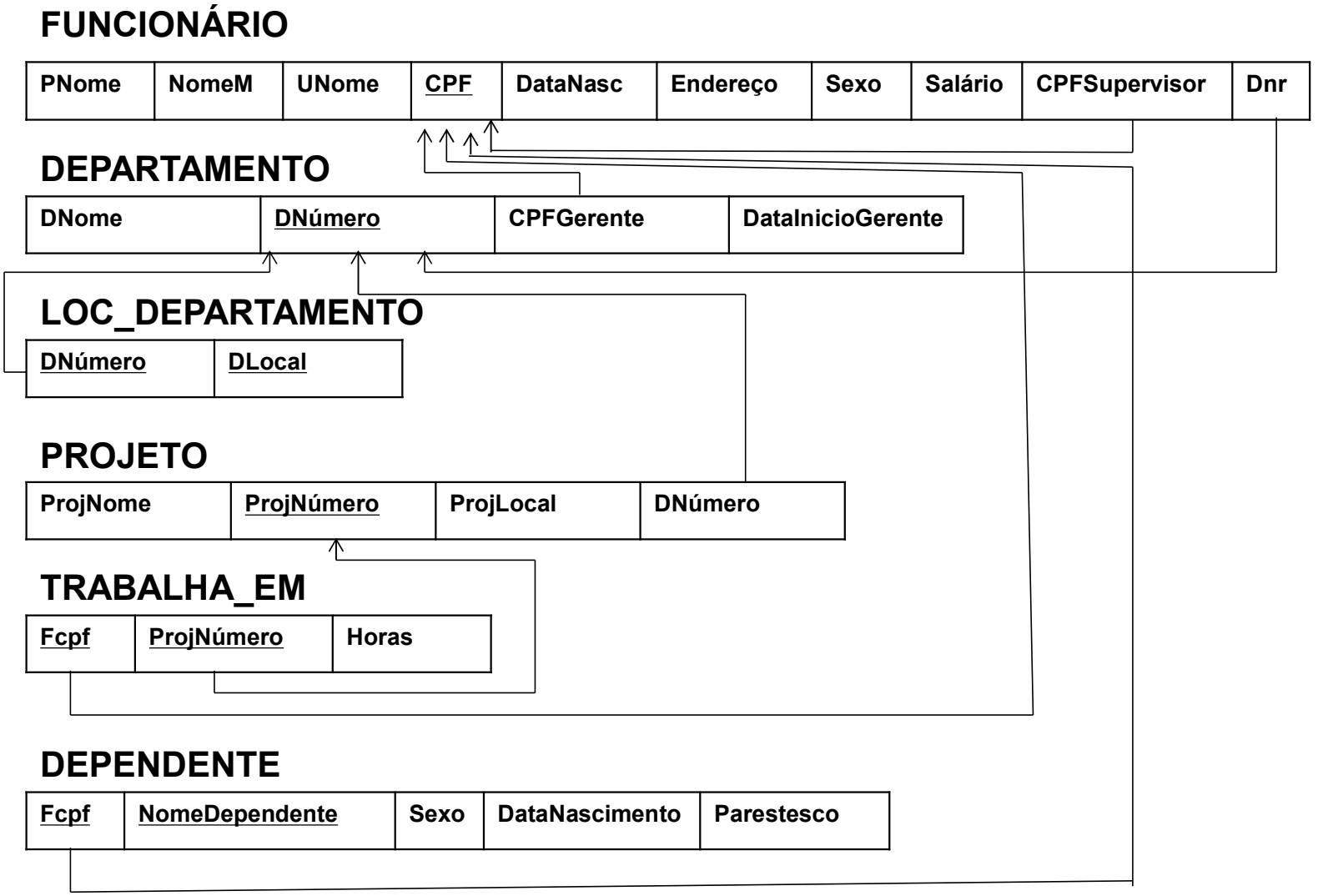
Professor

ID	*nome*	*nome_dept*	*salário*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Exemplos de Funções Armazenadas no MySQL

- Para o próximo exemplo de função armazenada, considere o banco de dados EMPRESA apresentado no próximo slide.
- Esse banco de dados é discutido em detalhes no livro "Fundamentals of Database Systems" by Ramez Elmasri.
- Faremos uma rápida discussão sobre a modelagem do banco de dados EMPRESA para, em seguida, implementarmos uma função armazenada que calcula o salário médio dos funcionários de um dado departamento.
- A implementação será feita usando a sintaxe do MySQL.

Banco de Dados EMPRESA



Função Armazenada para o BD Empresa

- Definir uma função armazenada que, dado o nome de um departamento, calcule o salário médio dos funcionários.

```
1. create function avg_sal(nome_dept varchar(20))returns int reads sql data
2. begin
3.     declare avg_salario float; --variável local
4.     select avg(F.salario) into avg_salario
5.     from FUNCIONARIO as F, DEPARTAMENTO as D
6.     where D.Dnome = nome_dept and F.Dnr = D.Dnumero;
7.     return avg_salario;
8. end
```

Função Armazenada para o BD Empresa

- Definir uma função armazenada que, dado o nome de um departamento, retorna o número de funcionários no departamento.

```
1.create function conta_func(nome_dept varchar(20))returns int reads sql data
2.begin
3.    declare qtd int;
4.    select count(*) into qtd
5.    from FUNCIONARIO as F, DEPARTAMENTO as D
6.    where D.Dnome = nome_dept AND F.Dnr = D.Dnumero;
7.    return qtd;
8.end
```

Chamada de Funções Armazenadas

- Funções armazenadas normalmente são usadas diretamente em comandos SQL, o que muitas vezes melhora a legibilidade do código.
- Por exemplo, podemos chamar a função armazenada `avg_sal()` em um comando SELECT simples:

```
select avg_sal('Research');
```

- Ou usar, `conta_func()` para facilitar uma consulta mais complexa, tal como recuperar o nome de todos os departamentos que tenham mais de 15 funcionários.

```
1. select D.DNome
2. from DEPARTAMENTO as D
3. where conta_func(D.Dnome) > 15;
```

Agenda

- Rotinas armazenadas: Motivação, diferenças, vantagens e desvantagens.
- Funções armazenadas: declaração, chamada e exemplos.
- **Procedimentos armazenados: declaração, chamada e exemplos.**
- Construções de linguagem para funções e procedimentos armazenados.
- Chamadas de funções e procedimentos armazenados em Python.
- Chamadas de funções e procedimentos armazenados em Java.
- Gatilhos.

Procedimentos Armazenados

- Assim como as funções, procedimentos armazenados também podem ser definidos diretamente em componentes procedurais em SQL ou externamente usando uma linguagem de propósito geral, tais como Java, Python ou C++.

- Embora os conceitos apresentados aqui sejam bastante gerais, a sintaxe apresentada não é necessariamente a adotada por todos os SGBDs. Portanto, consulte sempre o manual.

- Particularizando para o SGBD MySQL, a seguir apresentaremos a sintaxe geral para a criação de procedimentos armazenadas.

Comando `CREATE PROCEDURE`

- Procedimentos armazenados não possuem retorno e os parâmetros podem ser de entrada (`in`) ou saída (`out`). Além disso, procedimentos armazenados podem consultar e **modificar** o banco de dados.

- Sintaxe geral:

```
1.  CREATE PROCEDURE nome_proc (parâmetros)
2.      corpo_do_procedimento;
```

- `nome_proc`: nome do procedimento armazenado.
- `parâmetros`: parâmetros de entrada (IN) e saída (OUT), separados por vírgula, cada qual seguido por seu tipo. IN e OUT devem preceder nome do parâmetro.
- `corpo_do_procedimento`: comandos válidos (sql e de linguagem) que definem a lógica do procedimento.

Procedimento Armazenado para o BD Empresa

- Considere novamente o BD EMPRESA. Definiremos um procedimento armazenado que, dado o nome de um departamento, retorna o número de funcionários naquele departamento.
- Note que o procedimento não tem o comando RETURN, bem como não é preciso especificar READS SQL DATA.

```
1. create procedure conta_func2(IN nome_dept varchar(20), OUT res int)
2. begin
3.    declare qtd int;
4.    select count(*) into qtd
5.    from FUNCIONARIO as F, DEPARTAMENTO as D
6.    where D.DNome = nome_dept AND F.Dnr = D.Dnumero;
7.    set res = qtd;
8. end
```

Chamada de Procedimento Armazenado

- Procedimentos armazenados **não podem** ser chamados embutidos em um comando SQL, tal como funções armazenadas. Diferentemente, procedimentos armazenados são chamados com o comando `CALL`.

- `CALL` pode retornar valores para o chamador via parâmetros declarados `OUT` ou `INOUT`. No caso particular do MySQL, com acesso via terminal ou MySQLWorkbench, parâmetros de saída devem ser passados usando @.

- Por exemplo, o procedimento armazenado do slide anterior pode ser chamado com parâmetros tais a seguir, supondo a existência de um departamento chamado Research:

```
CALL conta_func2('Research', @qtd);
```

Agenda

- Rotinas armazenadas: Motivação, diferenças, vantagens e desvantagens.
- Funções armazenadas: declaração, chamada e exemplos.
- Procedimentos armazenados: declaração, chamada e exemplos.
- **Construções de linguagem para funções e procedimentos armazenados.**
- Chamadas de funções e procedimentos armazenados em Python.
- Chamadas de funções e procedimentos armazenados em Java.
- Gatilhos.

Variáveis Locais e Atribuição

- Variáveis locais à procedimentos ou funções armazenadas são declaradas com o comando `DECLARE` e podem ter qualquer tipo válido.
  - Sintaxe geral: `declare nome_da_variável tipo [valor DEFAULT];`
  - Exemplo: `declare qtd int;`
- Atribuições são realizadas via comando `SET`.
  - Sintaxe geral: `set nome_da_variável = valor;`
  - Exemplo: `set qtd = 10;`
- Resultados de consulta que consistem em uma única tupla podem ser atribuídos a variáveis locais usando `into`:

```
1. select ... into nome_da_variável_local
2. from ...
3. Where ...;
```

Variáveis Locais e Atribuição

- O comando `DECLARE` é permitido apenas dentro de um bloco `BEGIN-END` e deve estar no seu início, antes de qualquer outra instrução.
- O escopo de uma variável local é o bloco `BEGIN-END` dentro do qual ela é declarada. A variável pode ser referenciada em blocos aninhados dentro do bloco declarante, exceto aqueles blocos que declaram uma variável com o mesmo nome.
- Uma variável local não deve ter o mesmo nome de um atributo (coluna) de tabela. Se uma instrução SQL, como uma instrução `SELECT...INTO`, contém uma referência a uma coluna e a uma variável local declarada com o mesmo nome, o MySQL atualmente interpretará a referência como sendo para a variável local.

Comandos de Controle de Fluxo de Execução

- Existem vários comandos para controlar o fluxo de execução de uma função ou procedimento armazenado. No MySQL, as principais construções são as seguintes:
  - `IF-ELSEIF-ELSE-END IF`
  - `CASE-WHEN-THEN-ELSE-END CASE`
  - `LOOP`
  - `WHILE`
  - `REPEAT`
  - `RETURN` (só para o caso das funções armazenadas).

- Essas construções podem ser aninhadas. Por exemplo, uma instrução `IF` pode conter um loop `WHILE`, que por sua vez pode conter um comando `CASE`.

Sintaxe Geral do Comando `IF`

```
1. IF condição THEN
2.    --bloco de código
3. ELSEIF outra condição THEN
4.    --bloco de código
5. ELSE
6.    --último bloco de código
7. END IF;
```

- Se uma condição for avaliada como verdadeira, o `bloco de código` correspondente a cláusula é executado. Se nenhuma condição for avaliada como verdadeira, o `bloco de código` da cláusula `ELSE` sé executado (se existir).
- Cada `bloco de código` consiste em uma ou mais instruções SQL ou outras construções de linguagem. Para um `bloco de código` composto por mais de um comando use `BEGIN-END`.

Exemplo: Comando IF

```
1.  CREATE FUNCTION compara(n INTEGER, m INTEGER) RETURNS VARCHAR(20)
2.  BEGIN
3.    DECLARE s VARCHAR(20);
4.    IF n > m THEN
5.           SET s = '>’;
6.    ELSEIF n = m THEN
7.           SET s = '=‘;
8.    ELSE
9.           SET s = '<‘;
10.   END IF;
11.   SET s = CONCAT(n, ' ', s, ' ', m);
12.   RETURN s;
13. END
```

A função compara dois valores inteiros, `m` e `n`, e retorna a string `s` informando qual é o maior.

Sintaxe Geral do Comando CASE

Há duas possibilidades para definir um bloco CASE

```
1. CASE case_value
2.     WHEN v1 THEN
3.       --bloco de código
4.     WHEN v2 THEN
5.       --bloco de código
6.     ELSE
7.       --último bloco de código
8. END CASE;
```

```
1. CASE
2.     WHEN condição THEN
3.       --bloco de código
4.     WHEN condição THEN
5.       --bloco de código
6.     ELSE
7.       --último bloco de código
8. END CASE;
```

Sintaxe Geral do Comando `CASE`

- Para a primeira sintaxe, `case_value` é um valor que é comparado à valores (`v1`, `v2`) em cada cláusula `WHEN`. Quando um valor igual é encontrado, o bloco de código correspondente é executado. Se nenhum valor for igual, a cláusula `ELSE` será executada (se houver). Os `blocos de código` consistem de um ou mais comandos SQL ou construções de linguagem válidos.
- Para a segunda sintaxe, cada `condição` da cláusula `WHEN` é avaliada até que uma seja verdadeira, momento no qual sua lista de comandos correspondente é executada. Se nenhuma `condição` for avaliada como verdadeira, a cláusula `ELSE` será executada, se houver.
- Se nenhuma condição corresponder ao valor testado e a instrução `CASE` não contiver nenhuma cláusula `ELSE`, um erro é reportado (`case not found`). Use um `ELSE` com bloco `BEGIN-END` vazio para evitar esse tipo de erro.

Exemplo: comando CASE

```
1. create procedure detalhe(in id integer, out frase varchar(50))
2. begin
3.     declare tot_c decimal(3,0);
4.     select S.tot_cred into tot_c
5.     from student as S
6.     where S.id = ID;
7.     CASE
8.        when tot_c > 200 then
9.           set frase = 'Formando';
10.       when tot_c < 200 and tot_c > 150 then
11.          set frase = 'Terceiro ano';
12.       when tot_c < 150 and tot_c > 100 then
13.          set frase = 'Segundo ano';
14.       when tot_c < 100 then
15.          set frase = 'Primeiro ano';
16.    END CASE;
17.end
```

```
mysql> select * from student;
+-------+----------+------------+----------+
| ID    | name     | dept_name  | tot_cred |
+-------+----------+------------+----------+
| 00128 | Zhang    | Comp. Sci. |      102 |
| 12345 | Shankar  | Comp. Sci. |       32 |
| 19991 | Brandt   | History    |       80 |
| 23121 | Chavez   | Finance    |      110 |
| 44553 | Peltier  | Physics    |       56 |
| 45678 | Levy     | Physics    |       46 |
| 54321 | Williams | Comp. Sci. |       54 |
| 55739 | Sanchez  | Music      |       38 |
| 70557 | Snow     | Physics    |        0 |
| 76543 | Brown    | Comp. Sci. |       58 |
| 76653 | Aoi      | Elec. Eng. |       60 |
| 98765 | Bourikas | Elec. Eng. |       98 |
| 98988 | Tanaka   | Biology    |      120 |
+-------+----------+------------+----------+
13 rows in set (0.00 sec)
```

O procedimento “detalha” a situação dos Estudantes com base em tot_cred.

Comandos ITERATE e LEAVE

- ITERATE é usado nos comandos LOOP, REPEAT e WHILE para reiniciar o loop. Ou seja, para pular para a próxima iteração de um loop, semelhante ao continue em outras linguagens.

- LEAVE é usado para sair de um bloco que possuir o rótulo fornecido. Se o rótulo for para o bloco mais externo, LEAVE sai do programa. Pode ser usado entre BEGIN-END ou em construções de laços. Pode ser visto como o equivalente a um break em outras linguagens.

- Nos slides a seguir veremos alguns exemplos que ilustram a aplicação desses comandos.

Sitaxe Geral do Comando LOOP

```
1. nome_do_loop: LOOP
2.     --bloco de código;
3.     IF condição_de_parada THEN
4.       LEAVE nome_loop;
5.     END IF;
6. END LOOP;
```

- LOOP implementa uma construção que permite a execução repetida de um `bloco de código`.
- As instruções dentro do loop são repetidas até que o loop seja encerrado com uma instrução LEAVE. O comando RETURN também pode ser usado, no caso de uma função armazenada.

```
1. CREATE PROCEDURE doiterate(OUT res INT)
2. BEGIN
3.    declare x int;
4.    declare acc int;
5.    set x = 0;
6.    set acc = 0
7.    loop_soma: LOOP
8.       SET acc = acc + x;
9.       SET x = x + 1;
10.      IF x < 100 THEN
11.         ITERATE loop_soma;
12.      END IF;
13.      LEAVE loop_soma;
14.   END LOOP;
15.   SET res = acc;
16.END;
```

Exemplo

Soma de 0 a 100 e retorna o resultado no argumento de saída `res`.

Sintaxe Geral do Comando REPEAT

```
1. REPEAT
2.     --bloco de código
3. UNTIL condição END REPEAT;
```

- O comando `REPEAT` executa o `bloco de código` repetidamente até que a condição seja verdadeira. O comando `REPEAT` sempre entra no loop pelo menos uma vez.
- O `bloco de código` consiste em uma ou mais instruções, cada uma terminada por um delimitador de instrução (;).

Exemplo

```
1. CREATE PROCEDURE dorepeat(in p1 INT, out x int)
2. BEGIN
3.    declare aux int;
4.    SET aux = 1;
5.    SET x = 1;
6.    REPEAT
7.      SET x = x * aux;
8.      SET aux = aux + 1;
9.      UNTIL aux > p1
10.   END REPEAT;
11.END
```

Calcula o fatorial de p1.

Sintaxe Geral do Comando `WHILE`

```
1. WHILE condição DO
2.    --bloco de comandos
3. END WHILE;
```

- O comando `WHILE` executa repetidamente o `bloco de comandos` enquanto a condição for verdadeira.
- O `bloco de código` consiste em uma ou mais instruções SQL, cada uma terminada por um delimitador (;).

Comando `WHILE`: exemplo

```
1. CREATE PROCEDURE dowhile(in p1 INT, out x int )
2. BEGIN
3.    DECLARE v1 INT DEFAULT 1;
4.    SET x = 1;
5.    WHILE v1 < p1  DO
6.       SET x = x * v1;
7.       SET v1 = v1 + 1;
8.    END WHILE;
9. END;
```

Calcula o fatorial de p1, retorna o resultado em x.

Cursores

- Um cursor é um objeto que permite percorrer um conjunto de tuplas retornadas por uma consulta e processar cada tupla individualmente. Ele fornece uma maneira de trabalhar com um conjunto de resultados (resultset) tupla por tupla.

- **Quando usá-los**:
  - São úteis quando o processamento de cada tupla exige lógica complexa que não pode ser expressa em uma única consulta SQL.
  - São usados também quando há necessidade de operações sequenciais. Isto é, quando o resultado do processamento de uma tupla influencia o processamento da próxima tupla.

Cursores

- Tipos de cursores MySQL e suas propriedades:
  - Asensitive: Um cursor sensitive refletiria todas as alterações feitas nas tuplas subjacentes durante a sua execução. MySQL, porém, não oferece esse suporte.
  - READ ONLY (somente leitura) ou FOR UPDATE: somente para leitura (default) ou atualizável.
    - No caso de cursores atualizáveis, o SGBD bloqueia as linhas selecionadas para evitar que outros processos as modifiquem. A definição do cursor deve conter a cláusula "FOR UPDATE" no final de sua definição.
  - Nonscrollable (não rolável): pode ser percorrido apenas em uma direção e não pode pular tuplas.

Cursores

Fluxo básico de trabalho com Cursores:

1. **Declare o cursor**: O cursor é declarado para uma consulta SQL específica.
2. **Abra o Cursor**: O cursor é aberto para iniciar a leitura do conjunto de resultados relacionado à consulta definida no passo 1.
3. **Busca de tuplas**: As tuplas do resultado da consulta são buscadas uma a uma pelo cursor para processamento. Se o cursor for atualizável, pode-se realizar atualizações em tabelas envolvidas na consulta associada ao cursor.
4. **Feche o Cursor**: O cursor deve ser fechado após o uso para liberar recursos.

Declarações de Cursores

- Sintaxe geral:

```
DECLARE nome_cursor CURSOR FOR
comando_select
[FOR UPDATE];
```

- Esta instrução declara um cursor chamado `nome_cursor` e o associa a um comando `SELECT` que recupera as tuplas a serem percorridas pelo cursor.
- A instrução SELECT não pode ter uma cláusula `INTO`.
- As declarações do cursor devem aparecer antes das declarações do manipulador e depois das declarações de variáveis locais.
- Um programa armazenado pode conter várias declarações de cursor, mas cada cursor declarado em um determinado bloco deve ter um nome exclusivo.

Abertura e Fechamento de Cursor

- Para abrir: `OPEN nome_cursor;`
  - Esta instrução abre um cursor declarado anteriormente.

- Para fechar: `CLOSE nome_cursor;`
  - Esta instrução fecha um cursor aberto anteriormente. Ocorre um erro se o cursor não estiver aberto. Se não for fechado explicitamente, um cursor será fechado no final do bloco BEGIN ... END em que foi declarado.

Comando de Busca (FETCH) do Cursor

```
FETCH  nome_cursor INTO nome_var1, nome_var2,...;
```

- Essa instrução normalmente é colocada em um looping para buscar a próxima tupla do conjunto de resultados do cursor especificado (que deve estar aberto) e avançar o ponteiro do cursor.
- Se existir uma tupla, as colunas buscadas serão armazenadas nas variáveis nomeadas. O número de colunas recuperadas pela instrução SELECT deve corresponder ao número de variáveis de saída especificadas na instrução `FETCH`.
- Para detectar o final da leitura, você deve especificar um handler (depois da declaração do cursor), tal como explicado no slide a seguir.

Handler para Fim de Leitura do Cursor

```
DECLARE CONTINUE HANDLER FOR NOT FOUND SET var1 = valor;
```

- Esse handler detecta quando não há mais linhas para buscar (`NOT FOUND`), o que significa que o cursor atingiu o final.
- Nessa situação você deve atribuir um valor a alguma variável que encerra o looping de leitura do cursor.

Exemplo

O exemplo do slide a seguir supõe a existência da tabela ***Professor*** e usa um cursor para encontrar o maior valor de salário dentre todos os professores.

Professor

ID	*nome*	*nome_dept*	*salário*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Exemplo: encontra o maior salário usando cursor

```
1. CREATE PROCEDURE maior_salario(OUT valor FLOAT)
2. BEGIN
3.   DECLARE done INT DEFAULT 0;
4.   DECLARE maior FLOAT DEFAULT 0.0;
5.   DECLARE v FLOAT;
6.   DECLARE cur_sal CURSOR FOR SELECT salario
7.                          FROM professor;
8.   DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;
9.   OPEN cur_sal;
```

```
10. read_loop: LOOP
11.    FETCH cur_sal INTO v;
12.    IF done=1 THEN
13.       LEAVE read_loop;
14.     END IF;
15.      IF v > maior THEN
16.          SET maior = v;
17.      END IF;
18.   END LOOP;
19.   SET valor = maior;
20.   CLOSE cur_sal;
21.END
```

Agenda

- Rotinas armazenadas: Motivação, diferenças, vantagens e desvantagens.
- Funções armazenadas: declaração, chamada e exemplos.
- Procedimentos armazenados: declaração, chamada e exemplos.
- Construções de linguagem para funções e procedimentos armazenados.
- **Chamadas de funções e procedimentos armazenados em Python.**
- Chamadas de funções e procedimentos armazenados em Java.
- Gatilhos.

Funções e Proc. Armazenados: Chamada via Python

- Em Python, procedimentos armazenados são chamados usando o método `callproc()` de um objeto cursor.

- Funções armazenadas são executadas como parte de um comando SQL normalmente executados via método `execute()` de um objeto cursor.

- A seguir descreveremos os passos básicos para cada caso e ilustraremos com um exemplo.

Chamada de Proc. Armazenado MySQL via Python

1. Conecte ao servidor MySQL.
2. Obtenha um objeto cursor da conexão.
3. Execute o procedimento armazenado chamando o método `callproc()` do cursor. O nome do procedimento é o primeiro argumento de `callproc()`. Se o procedimento requer parâmetros, passe-os como uma lista para o segundo argumento de `callproc()`. Se o procedimento armazenado retornar um ou mais conjuntos de resultados, você pode invocar o método `stored_results()` do objeto cursor para obter um iterador e iterar nos conjuntos de resultados. Para cada conjunto resultados use o método `fetchall()` ou `fetchone()` para recuperar as tuplas.
4. Feche o objeto cursor e a conexão com o banco de dados. Lembre-se de comitar (`commit`) se o procedimento alterar o banco de dados.

Exemplo 1

- Considere o banco de dados EMPRESA. O seguinte procedimento armazenado recupera informações de funcionários com base no nome do departamento:

```
1. create procedure find_all(in nome_dept varchar(20))
2. begin
3.    select Pnome, Cpf
4.    from FUNCIONARIO as F, DEPARTAMENTO as D
5.    where D.DNome = nome_dept AND F.Dnr = D.DNumero
6. end
```

Exemplo 1 (cont.)

- Abaixo os passos principais para a chamada do procedimento em Python e recuperar um resultset:

```
1. con = mysql.connector.connect(..., database='EMPRESA')
2. cur = con.cursor()
3. cur.callproc('find_all', ['Research'])
4. for result in cur.stored_results():
5.      print(result.fetchall())
6. cur.close()
7. con.close()
```

Exemplo 2

- O seguinte procedimento armazenado recupera informações de funcionários com base no nome do departamento. Note que o resultado é um número retornado no parâmetro de saída, e não uma tabela.

```
1. create procedure conta_func(in nome_dept varchar(20), out qtd int)
2. begin
3.    select COUNT(*) into qtd
4.    from FUNCIONARIO as F, DEPARTAMENTO as D
5.    where D.DNome = nome_dept AND F.Dnr = D.DNumero
6. end
```

Exemplo 2 (cont.)

```
1. con = mysql.connector.connect(..., database='EMPRESA')
2. cur = con.cursor()
3. args = [‘Research', 0] # zero na posicao do parametro de saída
4. result_args = cursor.callproc('conta_func_por_dept', args)
5. print(result_args[1]) # 1 é a posição do parâmetro de saída
6. cur.close()
7. con.close()
```

Execução de Função Armazenada MySQL em Python

1. Conectar ao servidor MySQL
2. Obter via objeto de conexão um objeto cursor.
3. Executar a função armazenada embutida em alguma comando sql usando `cursor.execute()`.
4. Extrair os resultados da consulta usando `cursor.fetchall()`, `cursor.fetchone()` ou `cursor.fetmany()`.
5. Fechar o objeto cursor e a conexão com o banco de dados.

Exemplo 3

- Considere a seguinte função armazenada que recupera informações de funcionários com base no nome do departamento:

```
1. create function conta_func_por_dept(in nome_dept varchar(20)) returns int
   reads sql data
2. begin
3.     declare qtd int;
4.     select COUNT(*) into qtd
5.     from FUNCIONARIO as F, DEPARTAMENTO as D
6.     where D.DNome = nome_dept AND F.Dnr = D.Dnumero;
7.     return qtd;
8. end
```

Exemplo 3 (cont.)

```
1. con = mysql.connector.connect(..., database='EMPRESA')
2. cur = con.cursor()
3. depto = ‘research’
4. cur.execute(“SELECT conta_func_por_dept(%s) as num_func”, (dpto,))
5. numero = cur.fetone()[0]:
6. print(numero)
7. cur.close()
8. con.close()
```

Agenda

- Rotinas armazenadas: Motivação, diferenças, vantagens e desvantagens.
- Funções armazenadas: declaração, chamada e exemplos.
- Procedimentos armazenados: declaração, chamada e exemplos.
- Construções de linguagem para funções e procedimentos armazenados.
- Chamadas de funções e procedimentos armazenados em Python.
- **Chamadas de funções e procedimentos armazenados em Java.**
- Gatilhos.

Chamada de Proc. Armazenados via JDBC

- A interface `java.sql.CallableStatement` do JDBC expõe a funcionalidade do procedimento armazenado. A implementação concreta de `CallableStatement` é fornecida pelos drivers JDBC específicos. Por exemplo, o driver JDBC do MySQL (`com.mysql.cj.jdbc.Driver`) fornece uma implementação real da interface `CallableStatement`.
- Funções armazenadas são executadas como parte de um comando SQL normal executados via `executeUpdate()` e `executeQuery()`.
- A seguir descreveremos os passos básicos para cada caso e ilustraremos com um exemplo.

Como Executar um Proc. Armazenado via JDBC

1. **Estabeleça uma Conexão com o BD**: Utilize DriverManager para conectar-se ao banco de dados MySQL e obter o objeto de conexão.
2. **Prepare o CallableStatement**: Use `Conn.prepareCall()` para criar uma instância de `CallableStatement` representando o proc. armazenado.
3. **Defina os Parâmetros do Procedimento**: Caso existam, configure os parâmetros de entrada e saída para o procedimento armazenado.
4. **Execute o procedimento**: Use `CallableStatement.execute()` ou para chamar o procedimento.
5. **Processe os Resultados**: Se o procedimento retornar resultados, use métodos como `getResultSet()` ou `getUpdateCount()` para processá-los.
6. **Feche os Recursos**: `CallableStatement` e `Connection`.

Exemplo 1

- Considere o seguinte procedimento armazenado que recupera informações de funcionários com base no nome do departamento:

```
1. create procedure conta_func(in nome varchar(20), out qtd int )
2. begin
3.    select COUNT(*) into qtd
4.    from FUNCIONARIO as F, DEPARTAMENTO as D
5.    where D.DNome = nome AND F.Dnr = D.DNumero
6. end
```

Passo 1: Conectar ao Servidor MySQL

```
1. Connection conn = null;
2. ...
3. try {
4.    conn = DriverManager.getConnection(url, usuário, password);
5.          ...
6. } catch (SQLException ex) {
7.       System.out.println("SQLException: " + ex.getMessage());
8.       System.out.println("SQLState: " + ex.getSQLState());
9.       System.out.println("VendorError: " + ex.getErrorCode());
10. }
```

Passo 2: Preparar a Chamada de Procedimento

```
1. Connection conn = null;
2. ...
3. try {
4.   ...
5.   CallableStatement cStmt= conn.prepareCall("{call conta_func(?, ?)}");
6.   ...
7. } catch (SQLException ex) {
8.         System.out.println("SQLException: " + ex.getMessage());
9.         System.out.println("SQLState: " + ex.getSQLState());
10.        System.out.println("VendorError: " + ex.getErrorCode());
11. }
```

Passo 3: Registrar de Parâmetros

```
1. ...
2. try {
3.   ...
4.  cStmt.setString(1, ”Research"); //entrada
5.  cStmt.registerOutParameter(2, Types.INTEGER); //saída
6.   ...
7. } catch (SQLException ex) {
8.        System.out.println("SQLException: " + ex.getMessage());
9.        System.out.println("SQLState: " + ex.getSQLState());
10.       System.out.println("VendorError: " + ex.getErrorCode());
11. }
```

Passo 4: Executar Procedimento

```
1. ...
2. try {
3.   ...
4.  boolean hasResults = cStmt.execute(); //executa
5.   ...
6. } catch (SQLException ex) {
7.       System.out.println("SQLException: " + ex.getMessage());
8.       System.out.println("SQLState: " + ex.getSQLState());
9.       System.out.println("VendorError: " + ex.getErrorCode());
10. }
```

Passo 4: Processar Resultados

```
1.  ...
2.  try {
3.   boolean hasResults = ...
4.   while (hadResults) {
5.        ResultSet rs = cStmt.getResultSet(); //recupera resultset
6.        ...
7.        hadResults = cStmt.getMoreResults();
8.   }
9.   int outputValue = cStmt.getInt(2); //recupera parâmetro de saída
10.  ...
11. } catch (SQLException ex) {
12.       ...
13.  }
```

Agenda

- Rotinas armazenadas: Motivação, diferenças, vantagens e desvantagens.
- Funções armazenadas: declaração, chamada e exemplos.
- Procedimentos armazenados: declaração, chamada e exemplos.
- Construções de linguagem para funções e procedimentos armazenados.
- Chamadas de funções e procedimentos armazenados em Python.
- Chamadas de funções e procedimentos armazenados em Java.
- **Gatilhos.**

Triggers: Introdução

- Um trigger é um objeto de BD associado a uma tabela, que é executado automaticamente como consequência de uma modificação na tabela.
- Podem ser usados para realização de verificações de valores a serem inseridos em uma tabela ou a realização de cálculos com valores envolvidos em uma atualização.
- Um trigger é definido para ser ativado quando uma instrução `insert`, `update` ou `delete` ocorre na tabela associada. Para definir um trigger precisamos especificar duas coisas:
  - Quando o trigger deve ser executado.
  - As ações a serem tomadas quando o trigger executa.
- Definido o trigger, o SGBD fica responsável por dispará-lo sempre que as condições para sua execução forem satisfeitas.

Triggers: Introdução

- Como exemplo ilustrativo, considere um sistema de controle de estoque.
- Suponha que, sempre que a venda de um dado produto aconteça, seja necessário atualizar o estoque.
- Isso pode ser feito com um trigger que monitora as vendas. Quando uma venda ocorrer (`insert`), o trigger é disparado para fazer a atualização (`update`) da tabela de produtos.

Tipos de Triggers

- Triggers (gatilhos) são definidos para serem executados **ANTES** ou **DEPOIS** de certos eventos.
- No MySQL, os principais tipos são os seguintes:

1. **BEFORE Trigger**:
   - Uso: Executado antes da operação especificada (INSERT, UPDATE, DELETE).
   - Exemplo: Validar ou modificar dados antes que eles sejam efetivamente inseridos, atualizados ou excluídos.
2. **AFTER Trigger**:
   - Uso: Executado após a operação especificada ter sido completada (INSERT, UPDATE, DELETE).
   - Exemplo: Atualizar tabelas relacionadas, registrar logs, ou realizar outras ações após a conclusão de uma operação.

Triggers: Definição em MySQL

- Aqui a sintaxe é apresentada considerando o MySQL e alertando que isso varia bastante de SGBD para SGBD. Consulte sempre o manual. Em todo caso, os conceitos são todos aplicáveis.

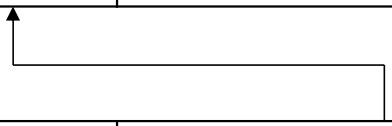
```
1. CREATE TRIGGER nome_do_trigger
2. [ BEFORE | AFTER ] [ INSERT | UPDATE | DELETE ]
3. ON nome_da_tabela
4. FOR EACH ROW
5. BEGIN
6.     -- Corpo do trigger
7. END;
```

Exemplo 1

- Considere o BD_ESTOQUE, com duas tabelas: PRODUTO e VENDAS.

PRODUTO

ID	DESCRICAO	QTD_ESTOQUE	VALOR_UNT
1	arroz	100	5
2	feijão	200	6
3	leite	70	2



VENDAS

ID	ID_PROD	QTD_VENDA	VALOR_VENDA

Exemplo 1

PRODUTO

ID	DESCRICAO	QTD_ESTOQUE	VALOR_UNT
1	arroz	100	5
2	feijão	200	6
3	leite	70	2

VENDA

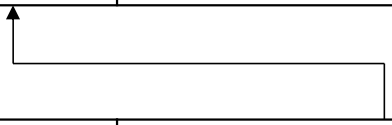
ID	ID_PROD	QTD_VENDA	VALOR_VENDA

- A quantidade de um produto em estoque deve ser sempre atualizada quando uma venda acontece.
- Um forma de se conseguir isso é criar um trigger que é disparado após uma inserção na tabela VENDA. A inserção só pode ser feita caso o estoque seja suficiente.

Exemplo 1

PRODUTO

ID	DESCRICAO	QTD_ESTOQUE	VALOR_UNT
1	arroz	100	5
2	feijão	200	6
3	leite	70	2



VENDA

ID	ID_PROD	QTD_VENDA	VALOR_VENDA

- Podemos implementar dois triggers:

1. BEFORE para verificar o estoque antes da inserção.
2. AFTER para atualizar o estoque no caso da venda ser efetivada.

Exemplo 1

```
1.CREATE TRIGGER verificar_estoque
2.BEFORE INSERT ON vendas
3.FOR EACH ROW
4.BEGIN
5.    DECLARE quantidade_estoque INT;
6.    SELECT qtd_estoque INTO quantidade_estoque
7.    FROM Produto
8.    WHERE id = NEW.id_prod;
9.    IF quantidade_estoque < NEW.qtd_venda THEN
10.        SIGNAL SQLSTATE '45000'
11.        SET MESSAGE_TEXT = 'Quantidade insuficiente em estoque para o produto.';
12.   END IF;
13.END;
```

Exemplo 1

```
1. create trigger update_estoque after insert on Vendas
2.   for each row
3.   begin
4.    update Produto
5.    set qtd_estoque = qtd_estoque – new.qtd_venda
6.    where id = new.id_prod;
7.   end
```

Exemplo 2

- Um gatilho pode chamar um procedimento armazenado usando o comando `CALL`.

- Por exemplo, suponha um banco de dados com uma única tabela que armazena contas bancárias:

CONTAS

ID	nome	saldo
1	João	1000,00
2	Maria	250,00
3	José	700,00

```
CREATE TABLE Contas (
 id int AUTO_INCREMENT,
 nome varchar(100) NOT NULL,
 saldo DECIMAL(10 , 2 ) NOT NULL ,
 primary key (id));
```

Exemplo 2

- O comando abaixo tenta atualizar a tabela contas, fazendo um saque no valor determinado por val.
- A princípio, devemos checar o valor disponível antes da atualização da tabela. Para isso usaremos um BEFORE trigger.

```
1. update contas
2.   set saldo = saldo - val
3.   where id = id_c;
4. end;
```

Exemplo 2

Esse gatilho não deixa atualizar se o saldo for insuficiente.

```
1. create trigger  before update on contas
2. FOR EACH ROW
3. BEGIN
4.     CALL verifica_saldo (OLD.id,  OLD.saldo - NEW.saldo );
5. END;
```

Exemplo 2

Verifica se o saldo é suficiente!

```
1.  create procedure verifica_saldo(id_c INT, valor dec(10,2))
2.  Begin
3.    declare aux dec(10, 2);
4.    select saldo into aux
5.    from contas
6.    where id = id_c;
7.    if valor > aux then
8.        signal SQLSTATE '45000’
9.            set MESSAGE_TEXT = ‘saque insuficiente’;
10.   end if;
11. End;
```