

# FUNCTIONS E STORED PROCEDURES

---

Sérgio Mergen

# Sumário

- **Functions**
- Cursores
- Stored Procedures

# Functions

- Executa código diretamente a partir do SGBD
- O código pode ser composto por comandos SQL e as estruturas algorítmicas suportadas (IF/ELSE, REPEAT, DECLARE, SET, CASE, LOOP, WHILE...)

# Functions

- Uma função pode receber parâmetros e **necessariamente** devolve um resultado
- Esse resultado pode ser usado dentro de uma consulta, assim como as funções MySQL pré-existentes
  - Ex. a função de manipulação de string **concat**

```
SELECT concat(nome, sobrenome) FROM func
```

# Sintaxe de uma Function(MySQL)

CREATE FUNCTION nome ([parâmetro,...])

RETURNS tipo

[característica ...] corpo\_da\_função

parâmetro: nome tipo

tipo: Qualquer tipo de dado suportado pelo MySQL

característica :

[NOT] DETERMINISTIC

| SQL SECURITY {DEFINER | INVOKER}

| COMMENT string

# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION soma (  
    a INTEGER,  
    b INTEGER  
) RETURNS INTEGER  
BEGIN  
    RETURN a + b;  
END  
$$  
DELIMITER ;
```

A função **soma** realiza a adição de dois valores recebidos como parâmetro

# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION soma (  
    a INTEGER,  
    b INTEGER  
) RETURNS INTEGER  
BEGIN  
    RETURN a + b;  
END  
$$  
DELIMITER ;
```

Como nas triggers, troca-se temporariamente o delimitador de comandos SQL, para que o ponto-e-vírgula possa ser usado dentro da função

# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION soma (  
    a INTEGER,  
    b INTEGER  
) RETURNS INTEGER  
BEGIN  
    RETURN a + b;  
END  
$$  
DELIMITER ;
```

(Declaração de parâmetros)

Os tipos de dados suportados são os mesmos usados nas colunas de tabelas: DECIMAL, INTEGER, CHAR, ...



# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION soma (  
    a INTEGER,  
    b INTEGER  
) RETURNS INTEGER  
BEGIN  
    RETURN a + b;  
END  
$$  
DELIMITER ;
```

O retorno é do tipo inteiro

# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION soma (  
    a INTEGER,  
    b INTEGER  
) RETURNS INTEGER  
BEGIN  
    RETURN a + b;  
END  
$$  
DELIMITER ;
```

O corpo da função fica delimitado pelos comandos BEGIN e END

# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION soma (  
    a INTEGER,  
    b INTEGER  
) RETURNS INTEGER  
BEGIN  
    RETURN a + b;  
END  
$$  
DELIMITER ;
```

O retorno é informado usando o comando RETURN

# Chamada a uma função

- Funções são chamadas a partir de um SELECT

```
SELECT nomeFunc, soma(salario, bonus) FROM func
```

- Também dá para usar funções em consultas sem usar dados provindos de colunas

```
SELECT soma(3,6)
```

# Quando usar

- No caso anterior, poderíamos chegar ao mesmo resultado sem recorrer a funções

```
SELECT nomeFunc, salario + bonus FROM func;
```

```
SELECT 3+6
```

- Assim sendo, quando usar funções?
  - Quando elas simplificam expressões mais complexas
    - Ex. cálculo de fatorial

# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION fatorial (n INTEGER)  
RETURNS INTEGER  
BEGIN  
    DECLARE fatorial INTEGER DEFAULT 1;  
    DECLARE cont INTEGER;  
    SET cont = n;  
    REPEAT  
        SET fatorial = fatorial * cont;  
        SET cont = cont - 1;  
    UNTIL cont < 2  
    END REPEAT;  
    RETURN fatorial;  
END $$
```

Variáveis locais

# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION fatorial (n INTEGER)  
RETURNS INTEGER  
BEGIN  
    DECLARE fatorial INTEGER DEFAULT 1;  
    DECLARE cont INTEGER;  
    SET cont = n;  
    REPEAT  
        SET fatorial = fatorial * cont;  
        SET cont = cont - 1;  
    UNTIL cont < 2  
    END REPEAT;  
    RETURN fatorial;  
END $$
```

(Laço de repetição)  
Continua no laço enquanto o  
contador for superior a 2.

# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION fatorial (n INTEGER)  
RETURNS INTEGER  
BEGIN  
    DECLARE fatorial INTEGER DEFAULT 1;  
    DECLARE cont INTEGER;  
    SET cont = n;  
    REPEAT  
        SET fatorial = fatorial * cont;  
        SET cont = cont - 1;  
    UNTIL cont < 2  
    END REPEAT;  
    RETURN fatorial;  
END $$
```

A cada iteração, o resultado é multiplicado pelo valor do contador



# Function - Exemplo

- Geralmente, funções utilizam como parâmetros dados provindos de colunas.
  - Para exemplificar, considere a função `calculaGratificacaoAnual(ano de ingresso, bônus)`
    - Calcula o reajuste a ser aplicado para cada funcionário com base em um bônus
    - A cada ano trabalhado, o funcionário recebe um bônus

```
SELECT nomeFunc, salario,  
       calculaGratificacaoAnual(anoContratacao, bonus) AS bonus  
FROM func;
```

- A implementação dessa função aparece a seguir

# Function - Exemplo

```
DELIMITER $$  
CREATE FUNCTION calculaGratificacaoAnual (  
ano INTEGER, bonus INTEGER  
) RETURNS INTEGER  
BEGIN  
    DECLARE anoAtual INTEGER;  
    SET anoAtual = YEAR(CURDATE());  
    RETURN (anoAtual - ano) * bonus;  
END  
$$
```

```
SELECT nomeFunc, salario,  
       calculaGratificacaoAnual(anoContratacao, bonus) AS bonus  
FROM func;
```

CURDATE()

recupera a data atual

YEAR()

recupera a parte referente ao ano

# Function - Exemplo

```
DELIMITER $$
CREATE FUNCTION calculaGratificacaoAnual (
ano INTEGER, bonus INTEGER
) RETURNS INTEGER
BEGIN
    DECLARE anoAtual INTEGER;
    SET anoAtual = YEAR(CURDATE());
    RETURN (anoAtual - ano) * bonus;
END
$$

SELECT nomeFunc, salario,
        calculaGratificacaoAnual(anoContratacao, bonus) AS bonus
FROM func;
```

O cálculo usa como base o ano de ingresso e bônus de cada funcionário

# Quando usar

- Note que aqui também poderíamos chegar no mesmo resultado sem recorrer a funções

```
SELECT nomeFunc, salario,  
       (YEAR(CURDATE()) - anoContratacao)*bonus  
FROM func;
```

- Nesse caso, vale a pena usar função?
  - Vale caso o cálculo envolvido seja comum a muitas consultas diferentes
  - Com funções, o cálculo fica codificado em um único lugar (**ponto único de mudança**)

# Sumário

- Functions
- **Cursores**
- Stored Procedures

# Cursors

- Para descobrir o custo total de todos os reajustes, poderíamos usar a seguinte consulta

```
SELECT  
    SUM(calculaGratificacaoAnual (anoContratacao, bonus))  
FROM func;
```

- No entanto, isso envolve múltiplas chamadas à função.
- Uma alternativa seria criar uma função que descubra esse valor total.
  - Essa função teria que acessar todos os registros de funcionário
  - Isso é possível através do uso de **cursors**

# Uso de Cursores

- Um cursor permite iterar pelos registros retornados por uma consulta
- Um cursor é:
  - **Read only:** não se consegue modificar os registros recuperados
  - **Non-scrollable:** Só se consegue avançar o cursor (um registro por vez), e não retroceder
  - **Asensitive:** O cursor aponta para o registro original. Caso o registro seja modificado, essa modificação é visível pelo cursor

# Uso de Cursores

- Comandos úteis ao se trabalhar com cursores:
- DECLARE **c** CURSOR FOR **consulta**:
  - Cria uma variável **c** do tipo cursor, associado a uma **consulta**
- OPEN **c**:
  - Abre o cursor **c** (executa a **consulta** e aponta o cursor para o primeiro registro)
- FETCH **c** INTO variáveis:
  - Coloca em variáveis os valores das colunas do registro apontado pelo cursor **c**
- CLOSE **c**:
  - Libera todos os recursos usados pela **consulta** aberta pelo cursor **c**



# Uso de Cursores

- Para exemplificar, considere a função **calculaGratificacaoAnualTotal(bônus)**
  - Calcula o reajuste a ser aplicado para cada funcionário com base em um bônus anual
  - A cada ano trabalhado, o funcionário recebe um bônus de 200 reais (igual para todos funcionários)
  - Retorna o reajuste total

```
SELECT calculaGratificacaoAnualTotal(200) AS bonus;
```

- A implementação dessa função aparece a seguir

# Uso de Cursores - Exemplo

```
DELIMITER $$
CREATE FUNCTION calculaGratificacaoAnualTotal (
bonus INTEGER
) RETURNS DECIMAL
BEGIN
    DECLARE total DECIMAL DEFAULT 0;
    DECLARE anoAtual INTEGER;
    DECLARE ano_ DECIMAL;
    DECLARE noMoreRows INTEGER DEFAULT 0;
    DECLARE cursor_ CURSOR FOR SELECT anoContratacao FROM func;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET noMoreRows = 1;

    ...
    RETURN total;
END $$
```

Variáveis usadas para o cálculo do total

# Uso de Cursores - Exemplo

```
DELIMITER $$
CREATE FUNCTION calculaGratificacaoAnual
bonus INTEGER
) RETURNS DECIMAL
BEGIN
    DECLARE total DECIMAL DEFAULT 0;
    DECLARE anoAtual INTEGER;
    DECLARE ano_ DECIMAL;
    DECLARE noMoreRows INTEGER DEFAULT 0;
    DECLARE cursor_ CURSOR FOR SELECT anoContratacao FROM func;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET noMoreRows = 1;

    ...
    RETURN total;
END $$
```

Variável do tipo **cursor**.  
Um **cursor** itera sobre o  
resultado de uma **consulta**

**IMPORTANTE:** Todas  
variáveis devem ser declaradas  
antes do cursor

# Uso de Cursores - Exemplo

```
DELIMITER $$
CREATE FUNCTION calculaGratificacaoAnualTotal (
bonus INTEGER
) RETURNS DECIMAL
BEGIN
    DECLARE total DECIMAL DEFAULT 0;
    DECLARE anoAtual INTEGER;
    DECLARE ano_ DECIMAL;
    DECLARE noMoreRows INTEGER DEFAULT 0;
    DECLARE cursor_ CURSOR FOR SELECT anoContratacao FROM func;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET noMoreRows = 1;

    ...
    RETURN total;
END $$
```

Variável que será usada para  
acessar dados retornados pelo  
cursor

# Uso de Cursores - Exemplo

```
DELIMITER $$
CREATE FUNCTION calculaGratificacaoAnualTotal (
bonus INTEGER
) RETURNS DECIMAL
BEGIN
    DECLARE total DECIMAL DEFAULT 0;
    DECLARE anoAtual INTEGER;
    DECLARE ano_ DECIMAL;
    DECLARE noMoreRows INTEGER DEFAULT 0;
    DECLARE cursor_ CURSOR FOR SELECT anoContratacao FROM func;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET noMoreRows = 1;

    ...
    RETURN total;
END $$
```

Variável de controle usada para verificar quando o cursor chegar ao final dos registros

# Uso de Cursores - Exemplo

```
DELIMITER $$
CREATE FUNCTION calculaGratificacaoAno
bonus INTEGER
) RETURNS DECIMAL
BEGIN
    DECLARE total DECIMAL DEFAULT 0;
    DECLARE anoAtual INTEGER;
    DECLARE ano_ DECIMAL;
    DECLARE noMoreRows INTEGER DEFAULT 0;
    DECLARE cursor_ CURSOR FOR SELECT anoContratacao FROM func;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET noMoreRows = 1;

    ...
    RETURN total;
END $$
```

Handler que é chamado quando ocorrer um evento chamado 'No Data' (SQLSTATE 02000).

O evento é disparado quando não houver mais registros a processar

Caso ocorra o evento, o valor da variável de controle é alterado

# Uso de Cursores - Exemplo

```
DELIMITER $$
CREATE FUNCTION calculaGratificacaoAnualTotal (
bonus INTEGER
) RETURNS DECIMAL
BEGIN
    DECLARE total DECIMAL DEFAULT 0;
    DECLARE anoAtual INTEGER;
    DECLARE ano_ DECIMAL;
    DECLARE noMoreRows INTEGER DEFAULT 0;
    DECLARE cursor_ CURSOR FOR SELECT anoContratacao FROM func;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET noMoreRows = 1;
    ...
    RETURN total;
END $$
```

A parte central da função  
(próximo slide)

# Uso de Cursores - Exemplo

```
OPEN cursor_;  
check_loop: LOOP
```

```
FETCH cursor_ INTO ano_;
```

```
IF noMoreRows = 1 THEN  
  LEAVE check_loop;  
END IF;
```

```
SET anoAtual = YEAR(CURDATE());  
SET total = total + (anoAtual - ano_) * bonus;
```

```
END LOOP check_loop;  
CLOSE cursor_;
```

consulta:  
**SELECT ano FROM func**

Abre o cursor.  
A consulta é executada e o  
cursor é posicionado no primeiro  
registro



# Uso de Cursores - Exemplo

```
OPEN cursor_  
check_loop: LOOP
```

```
    FETCH cursor_ INTO ano_;
```

```
    IF noMoreRows = 1 THEN  
        LEAVE check_loop;  
    END IF;
```

```
    SET anoAtual = YEAR(CURDATE());  
    SET total = total + (anoAtual - ano_) * bonus;
```

```
END LOOP check_loop;  
CLOSE cursor_;
```

consulta:  
**SELECT ano FROM func**

Laço de repetição.

O laço recebe um nome para que se consiga sair dele através de um comando específico (**LEAVE**)

# Uso de Cursores - Exemplo

```
OPEN cursor_  
check_loop: LOOP
```

```
    FETCH cursor_ INTO ano_;
```

```
    IF noMoreRows = 1 THEN  
        LEAVE check_loop;  
    END IF;
```

```
    SET anoAtual = YEAR(CURDATE());  
    SET total = total + (anoAtual - ano_) * bonus;
```

```
END LOOP check_loop;  
CLOSE cursor_;
```

consulta:  
**SELECT ano FROM func**

Se a variável de controle tiver o valor 1, significa que ocorreu o evento 'No Data'

Nesse caso, deve-se sair do laço

# Uso de Cursores - Exemplo

```
OPEN cursor_  
check_loop: LOOP
```

```
FETCH cursor_ INTO ano_;
```

```
IF noMoreRows = 1 THEN  
    LEAVE check_loop;  
END IF;
```

```
SET anoAtual = YEAR(CURDATE());  
SET total = total + (anoAtual - ano_) * bonus;
```

```
END LOOP check_loop;  
CLOSE cursor_;
```

consulta:  
**SELECT ano FROM func**

A atribuição das colunas da consulta para variáveis é feita com o FETCH

# Uso de Cursores - Exemplo

```
OPEN cursor_;  
check_loop: LOOP
```

```
    FETCH cursor_ INTO ano_;
```

```
    IF noMoreRows = 1 THEN  
        LEAVE check_loop;  
    END IF;
```

```
    SET anoAtual = YEAR(CURDATE());  
    SET total = total + (anoAtual - ano_) * bonus,
```

```
END LOOP check_loop;  
CLOSE cursor_;
```

consulta:  
**SELECT ano FROM func**

O total é reajustado com o bônus calculado para um funcionário específico

É usado o parâmetro **bônus** e o **ano de ingresso** trazido pelo cursor

# Uso de Cursores - Exemplo

```
OPEN cursor_;  
check_loop: LOOP
```

```
    FETCH cursor_ INTO ano_;
```

```
    IF noMoreRows = 1 THEN  
        LEAVE check_loop;  
    END IF;
```

```
    SET anoAtual = YEAR(CURDATE());  
    SET total = total + (anoAtual - ano_) * bonus;
```

```
END LOOP check_loop;  
CLOSE cursor_;
```

consulta:  
**SELECT ano FROM func**

É importante fechar o cursor no final do processamento para evitar que recursos continuem alocados sem necessidade

# Sumário

- Functions
- Cursores
- **Stored Procedures**

# Stored Procedure

- Assim como funções, uma stored procedure também executa código diretamente do SGBD
- Diferenças
  - Não usa a cláusula RETURN
    - Mas se pode retornar dados a partir de parâmetros de saída
  - Não se consegue usar dentro de comandos SQL
    - O procedimento deve ser chamado usando a cláusula CALL
  - Aceita comandos transacionais e comandos DDL

# Sintaxe de uma Stored Procedure (MySQL)

CREATE PROCEDURE nome ([parâmetro,...])  
[característica ...] corpo\_da\_função

parâmetro: [ IN | OUT | INOUT ] nome tipo

tipo: Qualquer tipo de dado suportado pelo MySQL

característica :

[NOT] DETERMINISTIC  
| SQL SECURITY {DEFINER | INVOKER}  
| COMMENT string



# Sintaxe de uma Stored Procedure (MySQL)

CREATE PROCEDURE nome ([parâmetro,...])  
[característica ...] corpo\_da\_função

parâmetro: [ IN | OUT | INOUT ] nome tipo

tipo: Qualquer tipo de dado suportado pelo MySQL

característica :

[NOT] DETERMINISTIC

| SQL SECURITY {DEFINER | INVOKER}

| COMMENT string

Em vez do RETURN, pode-se retornar dados a partir de um parâmetro do tipo OUT ou INOUT

# Stored Procedure - Exemplo

```
DELIMITER $$  
CREATE PROCEDURE fatorial2(INOUT n INTEGER)  
BEGIN  
    DECLARE fatorial INTEGER DEFAULT 1;  
    DECLARE cont INTEGER;  
    SET cont = n;  
    REPEAT  
        SET fatorial = fatorial * cont;  
        SET cont = cont - 1;  
    UNTIL cont < 2  
    END REPEAT;  
  
    SET n = fatorial;  
END $$  
DELIMITER ;
```

O parâmetro **n** é tanto de entrada quanto de saída (INOUT)

Ele guardará o fatorial calculado

# Stored Procedure - Exemplo

```
DELIMITER $$
CREATE PROCEDURE fatorial2(INOUT n INTEGER)
BEGIN
    DECLARE fatorial INTEGER DEFAULT 1;
    DECLARE cont INTEGER;
    SET cont = n;
    REPEAT
        SET fatorial = fatorial * cont;
        SET cont = cont - 1;
    UNTIL cont < 2
    END REPEAT;

    SET n = fatorial;
END $$
DELIMITER ;
```

O cálculo do fatorial é exatamente o mesmo que foi usado dentro da função

A única diferença é a forma de retorno

# Chamada a uma Procedure

- Um procedimento é chamado através da cláusula CALL
- Pode-se criar variáveis fora de um procedimento usando @
  - **SET** serve para atribuição de valor
  - **SELECT** para consumo do valor

```
SET @valor = 5;  
CALL fatorial2(@valor);  
SELECT @valor;
```

# Quando Usar uma Procedure

- Em muitos casos, pode-se tanto usar funções quanto procedimentos
  - Como no exemplo anterior (fatorial)
- Para tomar uma decisão, verifique se o trecho de código precisa ser executado a partir de uma consulta
  - Nesse caso, use uma função
  - Caso contrário, use procedimento

# Quando Usar uma Procedure

- Para exemplificar, considere que se deseja atualizar dados de funcionário
  - A cada ano trabalhado, o funcionário recebe um bônus de 200 reais
  - O total deve ser guardado dentro da coluna gratificação de cada funcionário
- Nesse caso, é mais adequado usar um procedimento

```
CALL atualizaGratificacaoAnual (200);
```

- A implementação aparece a seguir

# Procedure - exemplo

```
DELIMITER $$
CREATE PROCEDURE atualizaGratificacaoAnual(
IN bonus INTEGER )
BEGIN
    DECLARE total DECIMAL DEFAULT 0;
    DECLARE idFunc_ INTEGER;
    DECLARE ano_ INTEGER;
    DECLARE anoAtual INTEGER;
    DECLARE noMoreRows INTEGER DEFAULT 0;
    DECLARE cursor_ CURSOR FOR
        SELECT idFunc, anoContratacao FROM func;

    DECLARE CONTINUE HANDLER FOR NOT FOUND
        SET noMoreRows = 1;

    ...
END $$
```

O cursor agora dá acesso à duas colunas (**idFunc** e **ano**)

# Procedure - exemplo

```
OPEN cursor_;  
check_loop: LOOP
```

Consulta  
SELECT **idFunc**, **ano** FROM func;

```
    FETCH cursor_ INTO idFunc_, ano_;
```

```
    IF noMoreRows = 1 THEN  
        LEAVE check_loop;  
    END IF;
```

**IdFunc** e **ano** são recuperados  
usando FETCH INTO

A ordem deve ser a mesma que  
aparece no comando SELECT

```
    SET anoAtual = YEAR(CURDATE());  
    SET total = (anoAtual - ano_) * bonus;  
    UPDATE func SET bonus = total WHERE idFunc = idFunc_;
```

```
END LOOP check_loop;  
CLOSE cursor_;
```



# Procedure - exemplo

```
OPEN cursor_;  
check_loop: LOOP
```

```
    FETCH cursor_ INTO idFunc_, ano_;
```

```
    IF noMoreRows = 1 THEN  
        LEAVE check_loop;  
    END IF;
```

```
    SET anoAtual = YEAR(CURDATE());  
    SET total = (anoAtual - ano_) * bonus;  
    UPDATE func SET bonus = total WHERE idFunc = idFunc_;
```

```
END LOOP check_loop;  
CLOSE cursor_;
```

Consulta

```
SELECT idFunc, ano FROM func;
```

A variável **ano\_** calcula o bônus de um funcionário específico

O comando de UPDATE atualiza esse funcionário, com base no seu **idFunc\_**

# Quando Usar uma Procedure

- Em alguns casos, apenas procedimentos podem ser usados.
- Exemplos
  - O código compreende comandos que devem fazer parte de uma transação
  - O código implementa uma solução complexa que recorre a comandos DDL.

# Procedure - Exemplo

```
DELIMITER $$  
CREATE PROCEDURE atualizaSalarios(  
IN base DECIMAL )  
BEGIN
```

```
START TRANSACTION;  
    UPDATE func SET salario = salario * 1.05  
        WHERE salario > base;  
    UPDATE func SET salario = salario * 1.1  
        WHERE salario <= base;  
COMMIT;
```

```
END $$  
DELIMITER ;
```

```
CALL atualizaSalarios (2000);
```

Esse procedimento atualiza todos os salários a partir de um salário base (**2000**)

# Procedure - Exemplo

```
DELIMITER $$  
CREATE PROCEDURE atualizaSalarios(  
  IN base DECIMAL )  
BEGIN
```

```
  START TRANSACTION;  
    UPDATE func SET salario = salario * 1.05  
      WHERE salario > base;  
    UPDATE func SET salario = salario * 1.1  
      WHERE salario <= base;  
  COMMIT;
```

```
END $$  
DELIMITER ;
```

```
CALL atualizaSalarios (2000);
```

Quem ganha mais do que o  
salário base recebe um aumento  
menor

# Procedure - Exemplo

```
DELIMITER $$  
CREATE PROCEDURE atualizaSalarios(  
IN base DECIMAL )  
BEGIN
```

```
START TRANSACTION;
```

```
    UPDATE func SET salario = salario * 1.05
```

```
        WHERE salario > base;
```

```
    UPDATE func SET salario = salario * 1.1
```

```
        WHERE salario <= base;
```

```
COMMIT;
```

```
END $$
```

```
DELIMITER ;
```

```
CALL atualizaSalarios (2000);
```

Os comandos transacionais servem para garantir que todas alterações sejam desfeitas caso ocorra um erro no meio do processo

Ou tudo é feito, ou nada

# Quando Usar uma Procedure

- Em alguns casos, apenas procedimentos podem ser usadas.
- Exemplos
  - O código compreende comandos que devem fazer parte de uma transação
  - O código implementa uma solução complexa que recorre a comandos DDL.

# Quando Usar uma Procedure

- Para exemplificar, suponha que se deseja obter todos os subordinados (diretos e indiretos) de um funcionário específico (raiz)
  - Para cada funcionário, deve-se retornar quem é seu chefe direto e qual a distância (nível de subordinação) até o raiz

# Quando Usar uma Procedure

- Ex. Deseja-se obter os subordinados (diretos e indiretos) do funcionário 1

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Resposta

idFunc	idChefe	Nivel
2	1	0
3	1	0
4	2	1
5	2	1
6	4	2



# Quando Usar uma Procedure

- Ex. Deseja-se obter os subordinados (diretos e indiretos) do funcionário 2

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Resposta

idFunc	idChefe	Nivel
4	2	0
5	2	0
6	4	1

# Quando Usar uma Procedure

- Ex. Deseja-se obter os subordinados (diretos e indiretos) do funcionário 3

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Resposta

idFunc	idChefe	Nivel
--------	---------	-------

# Quando Usar uma Procedure

- Não se consegue resolver esse problema a partir de uma consulta simples
- É necessário recorrer a um script mais elaborado
- Uma solução envolve criar uma tabela temporária que armazenará o resultado
- Como é necessário um comando DDL, só se pode usar um procedimento

# Procedure - Exemplo

```
CREATE PROCEDURE calculaSub( raiz INTEGER )
BEGIN
    DECLARE cont INTEGER DEFAULT 0;
    DROP TABLE IF EXISTS arvore;
    CREATE TABLE arvore
        SELECT idFunc, idChefe, 0 AS nivel
        FROM func WHERE idChefe = raiz;
    ALTER TABLE arvore
        ADD PRIMARY KEY(idFunc);

    REPEAT
        INSERT INTO arvore
            SELECT f.idFunc, f.idChefe, a.nivel+1
            FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc
            WHERE nivel = cont;
        SET cont = cont + 1;
    UNTIL Row_Count() = 0
    END REPEAT;
END ;
```

Esse procedimento encontra a resposta a partir de um funcionário **raiz**

# Procedure - Exemplo

```
CREATE PROCEDURE calculaSub( raiz INTEGER )
BEGIN
  DECLARE cont INTEGER DEFAULT 0;
  DROP TABLE IF EXISTS arvore;
  CREATE TABLE arvore
    SELECT idFunc, idChefe, 0 AS nivel
    FROM func WHERE idChefe = raiz;
  ALTER TABLE arvore
    ADD PRIMARY KEY(idFunc);

  REPEAT
    INSERT INTO arvore
      SELECT f.idFunc, f.idChefe, a.nivel+1
      FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc
      WHERE nivel = cont;
    SET cont = cont + 1;
  UNTIL Row_Count() = 0
  END REPEAT;
END ;
```

O processo envolve a criação de uma tabela que guardará o resultado (tabela **arvore**)

A tabela precisa ser recriada a cada execução da procedure, para evitar que registros de execuções anteriores interfiram no resultado

# Procedure - Exemplo

```
CREATE PROCEDURE calculaSub( raiz INTEGER )
BEGIN
  DECLARE cont INTEGER DEFAULT 0;
  DROP TABLE IF EXISTS arvore;
  CREATE TABLE arvore
    SELECT idFunc, idChefe, 0 AS nivel
    FROM func WHERE idChefe = raiz;
  ALTER TABLE arvore
    ADD PRIMARY KEY(idFunc);

  REPEAT
    INSERT INTO arvore
      SELECT f.idFunc, f.idChefe, a.nivel+1
      FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc
      WHERE nivel = cont;
    SET cont = cont + 1;
  UNTIL Row_Count() = 0
  END REPEAT;
END ;
```

A tabela possui três colunas:

- id do funcionário
- id do seu chefe direto
- distância até o chefe raiz

Ela é criada a partir de um SELECT

# Procedure - Exemplo

```
CREATE PROCEDURE calculaSub( raiz INTEGER )
BEGIN
  DECLARE cont INTEGER DEFAULT 0;
  DROP TABLE IF EXISTS arvore;
  CREATE TABLE arvore
    SELECT idFunc, idChefe, 0 AS nivel
    FROM func WHERE idChefe = raiz;
  ALTER TABLE arvore
    ADD PRIMARY KEY(idFunc);

  REPEAT
    INSERT INTO arvore
      SELECT f.idFunc, f.idChefe, a.nivel+1
      FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc
      WHERE nivel = cont;
    SET cont = cont + 1;
  UNTIL Row_Count() = 0
  END REPEAT;
END ;
```

Inicialmente a tabela guarda apenas os subordinados diretos do func raiz:

- o idFunc (subordinado)
- o idChefe (raiz)
- o nível 0

# Procedure - Exemplo

```
CREATE PROCEDURE calculaSub( raiz INTEGER )
```

```
BEGIN
```

```
  DECLARE cont INTEGER DEFAULT 0;
```

```
  DROP TABLE IF EXISTS arvore;
```

```
  CREATE TABLE arvore
```

```
    SELECT idFunc, idChefe, 0 AS nivel
```

```
    FROM func WHERE idChefe = raiz;
```

```
  ALTER TABLE arvore
```

```
    ADD PRIMARY KEY(idFunc);
```

```
REPEAT
```

```
  INSERT INTO arvore
```

```
    SELECT f.idFunc, f.idChefe, a.nivel+1
```

```
    FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc
```

```
    WHERE nivel = cont;
```

```
  SET cont = cont + 1;
```

```
UNTIL Row_Count() = 0
```

```
END REPEAT;
```

```
END ;
```

(O laço de repetição)

Cada funcionário existente na tabela **arvore** pode levar à criação de novos registros.

A ideia é inserir registros referentes àqueles que são subordinados desse funcionário



# Procedure - Exemplo

```
CREATE PROCEDURE calculaSub( raiz INTEGER )
BEGIN
  DECLARE cont INTEGER DEFAULT 0;
  DROP TABLE IF EXISTS arvore;
  CREATE TABLE arvore
    SELECT idFunc, idChefe, 0 AS nivel
    FROM func WHERE idChefe = raiz;
  ALTER TABLE arvore
    ADD PRIMARY KEY(idFunc);

  REPEAT
    INSERT INTO arvore
      SELECT f.idFunc, f.idChefe, a.nivel+1
      FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc
      WHERE nivel = cont;
    SET cont = cont + 1;
  UNTIL Row_Count() = 0
  END REPEAT;
END ;
```

Um contador (iniciado em zero) é incrementado à cada iteração.

Ele é usado na consulta para selecionar os últimos registros inseridos na tabela **arvore**

# Procedure - Exemplo

```
CREATE PROCEDURE calculaSub( raiz INTEGER )
BEGIN
  DECLARE cont INTEGER DEFAULT 0;
  DROP TABLE IF EXISTS arvore;
  CREATE TABLE arvore
    SELECT idFunc, idChefe, 0 AS nivel
    FROM func WHERE idChefe = raiz;
  ALTER TABLE arvore
    ADD PRIMARY KEY(idFunc);

  REPEAT
    INSERT INTO arvore
      SELECT f.idFunc, f.idChefe, a.nivel+1
      FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc
      WHERE nivel = cont;
    SET cont = cont + 1;
  UNTIL Row_Count() = 0
  END REPEAT;
END ;
```

O row\_count() guarda o número de registros atualizados pelo último comando de atualização (**insert**, update ou delete).

Quando chegar a zero, significa que os últimos funcionários inseridos não levaram à geração de novos registros

# Explicando a Lógica

```
CREATE TABLE arvore  
  SELECT idFunc, idChefe, 0 AS nível  
  FROM func WHERE idChefe = raiz;
```

Tabela func

<b>idFunc</b>	<b>idChefe</b>
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

<b>idFunc</b>	<b>idChefe</b>	<b>Nivel</b>
---------------	----------------	--------------

O processamento começa com a criação da tabela arvore a partir de um select

# Explicando a Lógica

```
CREATE TABLE arvore  
  SELECT idFunc, idChefe, 0 AS nível  
  FROM func WHERE idChefe = 1;
```

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

idFunc	idChefe	Nível
--------	---------	-------

O select retorna os subordinados do raiz

# Explicando a Lógica

```
CREATE TABLE arvore  
  SELECT idFunc, idChefe, 0 AS nível  
  FROM func WHERE idChefe = 1;
```

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

idFunc	idChefe	Nível
2	1	0
3	1	0

Eles são inseridos na  
tabela no nível 0

# Explicando a Lógica

```
INSERT INTO arvore
  SELECT f.idFunc, f.idChefe, a.nivel+1
  FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc
  WHERE nivel = 0;
```

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

idFunc	idChefe	Nivel
2	1	0
3	1	0

A partir daí, começa  
o laço REPEAT

O contador começa  
em 0

# Explicando a Lógica

INSERT INTO **arvore**

SELECT **f.idFunc**, **f.idChefe**, **a.nivel+1**

FROM **func** AS **f** JOIN **arvore** AS **a** ON **f.idChefe** = **a.idFunc**

WHERE **nivel** = 0;

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

idFunc	idChefe	Nivel
2	1	0
3	1	0

Na primeira  
iteração, se descobre  
os subordinados dos  
funcionários no  
nível 0

# Explicando a Lógica

INSERT INTO **arvore**

SELECT **f.idFunc**, **f.idChefe**, **a.nivel+1**

FROM **func** AS **f** JOIN **arvore** AS **a** ON **f.idChefe** = **a.idFunc**

WHERE **nivel = 0**;

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

idFunc	idChefe	Nivel
2	1	0
3	1	0
4	2	1
5	2	1

Eles são inseridos na  
tabela no nível 1



# Explicando a Lógica

INSERT INTO **arvore**

SELECT **f.idFunc**, **f.idChefe**, **a.nivel+1**

FROM **func** AS **f** JOIN **arvore** AS **a** ON **f.idChefe** = **a.idFunc**

WHERE **nivel** = **1**;

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

idFunc	idChefe	Nivel
2	1	0
3	1	0
4	2	1
5	2	1

Na próxima  
iteração, se descobre  
os subordinados dos  
funcionários no  
nível **1**

# Explicando a Lógica

```
INSERT INTO arvore  
  SELECT f.idFunc, f.idChefe, a.nivel+1  
  FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc  
  WHERE nivel = 1;
```

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

idFunc	idChefe	Nivel
2	1	0
3	1	0
4	2	1
5	2	1
6	4	2

Ele é inserido na  
tabela no nível 2

# Explicando a Lógica

```
INSERT INTO arvore
SELECT f.idFunc, f.idChefe, a.nivel+1
FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc
WHERE nivel = 2;
```

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

idFunc	idChefe	Nivel
2	1	0
3	1	0
4	2	1
5	2	1
6	4	2

Na próxima  
iteração, se descobre  
os subordinados dos  
funcionários no  
nível 2

# Explicando a Lógica

```
INSERT INTO arvore  
  SELECT f.idFunc, f.idChefe, a.nivel+1  
  FROM func AS f JOIN arvore AS a ON f.idChefe = a.idFunc  
  WHERE nivel = 2;
```

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4

Tabela arvore (raiz = 1)

idFunc	idChefe	Nivel
2	1	0
3	1	0
4	2	1
5	2	1
6	4	2

Nada é retornado e  
row\_count chega a  
zero

O laço é encerrado

# Exemplos de Uso da Procedure

```
CALL calculaSub(1);  
SELECT * FROM arvore;
```

idFunc	idChefe	Nivel
2	1	0
3	1	0
4	2	1
5	2	1
6	4	2

```
SELECT CONCAT(SPACE(nivel),idChefe) AS chefe,  
       Group_Concat(idFunc ORDER BY idFunc) AS subordinado  
FROM arvore  
GROUP BY idChefe;
```

chefe	subordinado
1	2,3
2	4,5
4	6

# Dados hierárquicos

- O problema anterior envolve busca sobre dados hierárquicos
- Esse tipo de busca pode ser resolvida usando consultas recursivas
- Alguns bancos de dados dão suporte à consultas recursivas por meio de um recurso chamado **CTE** (Common Table Expression)
- Falaremos sobre isso em outra aula

# Fechamento

- Principais finalidades de funções e procedimentos
  - Encapsular código
    - isso aumentar o reuso
  - Reduzir tráfego de rede
    - Em vez de passar comandos (e possivelmente dados) entre o cliente e o SGBD, um único comando é passado
  - Reduzir problemas de segurança
    - Como poucos dados são trafegados, isso reduz a exposição de informações que poderiam ser alvo de interceptação por hackers
  - Aumento de performance
    - Executar todo o script dentro do banco pode ser mais eficiente do que dividir o processamento com o cliente

# Exercício 1

- Crie uma função (chamada arranjo) que calcule o número de arranjos de  $n$  elementos, tomados  $k$  a  $k$ .
- Fórmula para calcular o arranjo:

$$a = \frac{n!}{(n - k)!}$$

- A função recebe dois parâmetros( $n$  e  $k$ ).
- Aproveite a função de cálculo de fatorial, que já está pronta
- Demonstre o uso da função



# Exercício 2

- Crie uma procedure (chamada `arranjo2`) que calcule o número de arranjos de  $n$  elementos, tomados  $k$  a  $k$ .
- A procedure recebe dois parâmetros de entrada ( $n$  e  $k$ ) e um de saída.
- A procedure também pode aproveitar a função de cálculo de fatorial, que já está pronta.
- Demonstre o uso da procedure.

# Exercício 3

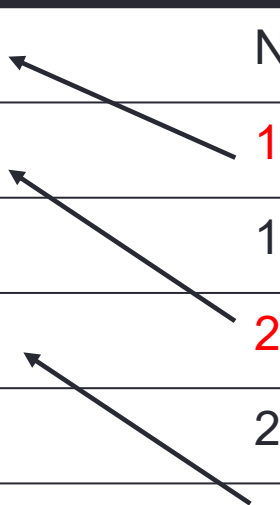
- Crie uma procedure (chamada calculaChefes) que descubra todos os chefes de um funcionário base (incluindo os indiretos).
- Para cada chefe, deve-se saber a distância dele até o funcionário base.
- A procedure recebe um parâmetro (o func base).
- A resposta pode ser colocada em uma tabela temporária (com os campos idChefe e nível).
- Demonstre o uso da procedure.

# Exercício 3

- Ex. Deseja-se obter os chefes do funcionário 6

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4



Resposta

idChefe	Nivel
4	0
2	1
1	2

# Exercício 3

- Ex. Deseja-se obter os chefes do funcionário 2

Tabela func

idFunc	idChefe
1	Null
2	1
3	1
4	2
5	2
6	4



Resposta

idChefe	Nivel
1	0

# Atividade Individual

- A tabela pedido tem uma coluna multivalorada (produtos)
  - Os produtos são armazenados em uma string, como uma lista separada por vírgulas
    - Ex. ('Laptop,Mouse,Teclado')
- De certa forma, pode-se considerar que essa tabela não está na primeira forma normal

Pedido	
*idPedido	int
idCliente	int
produtos	

# Atividade Individual

- Para normalizá-la, decidiu-se aplicar o processo de divisão de tabelas, conforme descrito abaixo
- Agora cada um dos produtos do pedido fica armazenado em um registro exclusivo

Pedido	
*idPedido	int
idCliente	int
produtos	

Antes da divisão

Pedido	
*idPedido	int
idCliente	char(30)

Depois da divisão

ItemPedido	
*idPedido	int
*numItem	int
produto	

# Atividade Individual

- Para realizar esse processo, será necessário
  - Criar a tabela itemPedido
  - Realizar a migração dos dados de Pedido para ItemPedido
  - Remover a coluna produtos de Pedido

Pedido	
*idPedido	int
idCliente	int
produtos	

Antes da divisão

Pedido	
*idPedido	int
idCliente	char(30)

1

n

ItemPedido	
*idPedido	int
*numItem	int
produto	

Depois da divisão

# Atividade Individual

- Para a migração, crie uma stored procedure chamada `split_products()`
- Essa procedure deve
  - percorrer toda a tabela de pedido
  - Para cada pedido
    - Identificar cada um dos produtos do pedido
    - Para cada produto identificado
      - Armazená-lo na tabela de ItemPedido



# Atividade Individual

- A função abaixo pode ser útil
  - implementada no script de criação do banco
- **split\_string**(str TEXT, delim CHAR(1), pos INT)
- Exemplo de uso: str = 'Laptop, Mouse, Teclado'
  - Split\_string(str, ',', 1) => 'Laptop'
  - Split\_string(str, ',', 2) => 'Mouse'
  - Split\_string(str, ',', 3) => 'Teclado'