TÉCNICO LISBOA

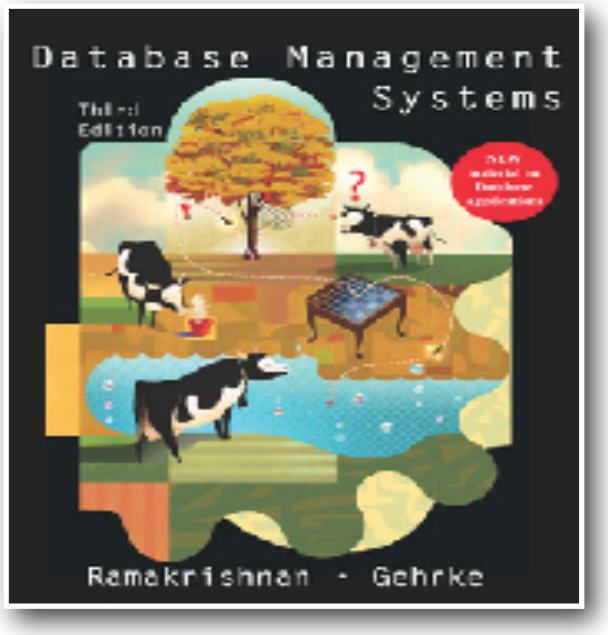
Bases de Dados

Aula 12: Triggers e Stored Procedures

Prof. Paulo Carreira

slides não são livros

Bibliografia



Capítulos 3 e 5

- Manual referência Postgres
  - https://www.postgresql.org/docs/9.5/static/sql-createtrigger.html
  - https://www.postgresql.org/docs/9.5/static/plpgsql-structure.html
  - https://www.postgresql.org/docs/9.5/static/plpgsql-control-structures.html

Sumário

- *Persistent Stored Modules (PSM)* em Postgres
  - Funções e Procedimentos
  - Cursores
  - Triggers
  - Excepções

Introdução

Persistent Stored Module

- Conjunto de instruções (tal como um sub-programa em qualquer linguagem de programação) que é guardado na BD também conhecido como stored program/stored routine

- Corresponde a rotina que pode ser de dois tipos:
  - *Stored procedures* que chamadas com parâmetros
  - *Funções* que devolvem um valor

- Pode ser invocado por:
  - *Triggers, Stored Procedures, Aplicações (em Java, C++, etc…)*

Funções e Procedimentos

- PSM permitem definir funções e procedimentos
  - Estrutura de controlo (*if-then-else*, ciclos, excepções), declaração de variáveis, atribuição de valores a variáveis, etc…
- Alguns SGBDs suportam ainda
  - Funções que devolvem relações (SQL:2003)
  - Outras funcionalidades proprietárias

Linguagens para Extensões Procedimentais de SGBDs

- **PL/pgSQL – Postgres (=PL/SQL)**
- SQL/PSM – Standard SQL (IBM DB2 e MySQL)
- Transact-SQL – MS SQL Server + Sybase
- PL/SQL – Oracle
- …

Vantagens

- Permitem que as aplicações sejam mais eficientes
  - Compilados e mantidos na BD
- Permitem reduzir o tráfego de informação na rede entre aplicação e servidor de BD
- São reutilizáveis e transparentes para a aplicação
  - Útil quando aplicações codificadas em diferentes linguagens executam as mesmas operações na BD
- Suportam acesso restrito à informação guardada na BD
  - Operações são encapsuladas em *stored programs*; aplicações só as executam e não acedem às tabelas base directamente

Inconvenientes

- Desenvolvimento e manutenção de PSMs é relativamente complexa (sobretudo quando a lógica de domínio é complexa)
- É difícil fazer *debugging* e *profiling*
- Sintaxe e ambiente de desenvolvimento altamente dependente do SGBD utilizado

Exemplo de sintaxe (PSM Standard)

```
CREATE PROCEDURE procedure1(IN parameter1 INTEGER) AS
                                          -- nome e parâmetros
DECLARE variable1 CHAR(10);               -- variáveis
BEGIN                                     -- início de bloco
     IF parameter1 = 17                   -- início de IF
     THEN
       SET variable1 := 'birds';              – atribuição
     ELSE
       SET variable1 := 'beasts';
     END IF;                             -- fim de IF
     INSERT INTO table1
        VALUES (variable1);              -- instrução SQL
END                                      -- fim de bloco
```

Definição vs. Chamada

- Definição de procedimento e função

```
CREATE PROCEDURE/FUNCTION …

DROP PROCEDURE/FUNCTION …
```

- Redefinição de procedimento e função

```
CREATE OR REPLACE PROCEDURE

CREATE OR REPLACE FUNCTION
```

- Invocação de procedimento e função

```
CALL nome_procedimento [(param1, param2,…)]

nome_função [(param1, param2,…)]
```

https://www.postgresql.org/docs/9.1/static/sql-createfunction.html

Stored Procedures in Postgres

- PL/PgSQL does not support true stored procedures; only stored functions are supported!

```
CREATE FUNCTION myfunc([params]) RETURNS
  VOID AS …
```

- They are not capable of starting autonomous transactions
- Within a function, we can abort a running transaction but we cannot commit or open a new separate transaction

Transactional Behavior

- Stored Procedures are not able to start autonomous transactions (open a new separate transaction)
- A function can abort a running transaction but cannot commit

Instruções
(corpo de um Stored Module)

Exemplo

```
CREATE OR REPLACE FUNCTION
   account_c(name_count VARCHAR(40))
 RETURNS INTEGER AS
$$
   DECLARE a_count INTEGER;
BEGIN
   SELECT COUNT(*) INTO a_count
   FROM depositor
   WHERE customer_name = c_name;
    RETURN a_count;
END
$$ LANGUAGE plpgsql;
```

Elementos Procedimentais Básicos

- Declaração de variáveis

```
DECLARE var_name [, var_name ...]
  type [DEFAULT value]
```

  - o âmbito da variável local está restrito ao bloco onde foi declarada

Elementos Procedimentais Básicos

- Atribuição de valores a variáveis simples

```
var_name := expr
```

```
LET var_name := expr
```

```
SET var_name := expr
```

Em alguns SGBDs

- Captura do resultado de uma query:

```
SELECT col_name, … INTO var_name FROM …
```

Condições e Ciclos

```
IF condicao THEN lista_instrucoes1
ELSE lista_instrucoes2
END IF;

WHILE condicao
DO lista_instrucoes
END WHILE;

REPEAT lista_instrucoes
UNTIL condicao
END REPEAT;

LOOP lista_instrucoes
END LOOP;
```

Exemplo de função

```
CREATE FUNCTION add_me(x NUMERIC, y NUMERIC) RETURNS
   DECIMAL(10,0) AS
$$
BEGIN
  RETURN x + y;
END
$$ LANGUAGE plpgsql;
```

- Como executar a função?

```
SELECT add_me(2,3);
```

```
SELECT add_me(salary, 100)
FROM Employee
```

```
SELECT *
FROM Employee
WHERE add_me(salary, -1000) < 5000
```

Sintaxe das funções

- A instrução RETURNS pode ser especificada apenas numa função, na qual é obrigatória
- O corpo da função tem que conter uma instrução RETURN *valor*
- Uma função é considerada *determinística* se produzir sempre os mesmos resultados para os mesmos parâmetros de entrada.
  - Por omissão é não determinística (especificar **IMMUTABLE** caso contrário)
  - Exemplo de uma função não determinística: invoca a função NOW() ou RAND()

https://www.postgresql.org/docs/9.2/static/sql-createfunction.html

Parâmetros

- Lista de parâmetros entre () tem que estar sempre presente
  - Se não existirem parâmetros, colocar ()
- Cada parâmetro é **IN** por omissão
  - Se não for o caso, colocar **OUT** ou **INOUT** antes do nome do parâmetro
- Parâmetros **IN**, **OUT**, **INOUT** só são válidos para procedimentos; funções só têm parâmetros **IN**
- Procedimento pode mudar o valor do parâmetro **IN** mas o novo valor não é visível fora do procedimento depois de **RETURN**
- Um parâmetro **OUT** tem o seu valor inicial igual a **NULL**
- Um parâmetro INOUT tem o seu valor inicial dado por quem invoca o procedimento, pode ser modificado pelo procedimento e o seu valor novo é visível depois do procedimento retornar

Exemplos

Outro exemplo de função

Definir uma função que, dado o nome de um cliente, devolve o número de contas desse cliente

```
CREATE OR REPLACE FUNCTION account_count(c_name VARCHAR(40))
 RETURNS INTEGER AS
$$
DECLARE a_count INTEGER;
BEGIN
  SELECT COUNT(*) INTO a_count
  FROM depositor
  WHERE customer_name = c_name;
   RETURN a_count;
END
$$ LANGUAGE plpgsql;
```

Funções que devolvem relações

Definir uma função que, dado o nome de um cliente, devolve todas as contas do cliente

```
CREATE FUNCTION accounts_of(name VARCHAR(80))
 RETURNS SETOF account
AS
$$
 SELECT a.account_number, branch_name, balance
     from account as a, depositor as d
 WHERE a.account_number = d.account_number
       and d.customer_name = name;
$$
LANGUAGE sql;
```

Funções que devolvem uma linha de uma tabela

Definir uma função que devolve os dados de uma conta

```
CREATE OR REPLACE FUNCTION accounts(acct_num VARCHAR(10))
     RETURNS account AS

$BODY$
DECLARE
     acct account%ROWTYPE;
BEGIN
     SELECT *
     INTO acct
     FROM account
     WHERE acct_number = acct_num;

     RETURN acct;
END

$BODY$
LANGUAGE plpgsql;
```

Funções com Tipos Complexos

Definir uma função que, dado o nome de um cliente, devolve informação das contas desse cliente

```
CREATE TYPE account_data AS (
     account_number VARCHAR(10),
     branch_name VARCHAR(40),
     balance NUMERIC(12,2)
);
```

```
DROP FUNCTION accounts_of(varchar);

CREATE FUNCTION accounts_of(name VARCHAR(80))
RETURNS account_data
AS $$
     SELECT a.account_number, branch_name, balance
     FROM account AS a, depositor AS d
     WHERE a.account_number = d.account_number
          AND d.customer_name = name;
$$ LANGUGE sql;
```

Utilização

Obter o Nome, Rua e Cidade dos clientes com mais do que uma conta

```
SELECT customer_name,
    customer_street,
 customer_city
FROM customer
WHERE account_count(customer_name) > 1
```

Exemplo apenas para demonstração.
Não usar na prática!

Exemplo de parâmetro OUT

```
CREATE OR REPLACE FUNCTION out_test(OUT x text, OUT y text)
AS $$
BEGIN
   x := 10;
   y := 20;
END;
$$  LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION get_test_read()
RETURNS VOID AS $$
DECLARE
   a text;
   b text;
BEGIN
   SELECT out_test() INTO a, b;

   RAISE INFO 'a: <%>', a;
   RAISE INFO 'b: <%>', b;

END;
$$  LANGUAGE plpgsql;
```

Cursores

- Abstração para ler tabela a partir de um programa como se fosse um ficheiro
  - `OPEN` / `FETCH` / `CLOSE`

- Podem ser usados dentro de Stored Modules

Cursores

```
CREATE OR REPLACE FUNCTION average_balance() RETURN REAL AS
$$
  DECLARE balance_var REAL DEFAULT 0.0;
  DECLARE sum_balance REAL DEFAULT 0.0;
  DECLARE count_balance INTEGER DEFAULT 0;
  DECLARE cursor_account CURSOR FOR
    SELECT balance FROM account;
BEGIN
  OPEN cursor_account;
  LOOP
    FETCH cursor_account INTO balance_var;
    sum_balance := sum_balance + balance_var;
    count_balance := count_balance + 1;
  END LOOP;
  CLOSE cursor_account;
  RETURN sum_balance / count_balance;
end
$$ LANGUAGE plpgsql;
```

TÉCNICO LISBOA

Bases de Dados

Aula 13: Triggers, Stored Procedures, Views

Prof. Paulo Carreira

slides não são livros

Sumário

- Triggers
- Excepções
- Views

Triggers

Triggers

- Um *trigger* é uma instrução executada em reacção a uma modificação na BD

- Para especificar um trigger é necessário saber:
  - as condições em que o trigger é disparado
  - as acções a fazer quando o trigger é executado

- Trigger é um conceito antigo, mas só foi standardizado na norma SQL:1999

Triggers

- Procedimento que é automaticamente invocado em reposta a actualizações específicas da base de dados
- A sua definição contém:
  - **Evento**: qual o tipo de actualização que activa o *trigger*
  - **Condição**: uma interrogação ou um teste para verificar se a acção deve ser executada
  - **Acção**: um procedimento que é executado quando o *trigger* é activado e a condição anterior é verdadeira

Triggers: Exemplo 1

Criar um trigger que evite que o saldo seja negativo ou maior que 100

```
CREATE OR REPLACE FUNCTION chk_balance_interval_proc()
RETURNS TRIGGER AS $BODY$
BEGIN
  IF NEW.balance < 0 THEN
    NEW.balance := 0;
  ELSEIF NEW.balance > 100 THEN
    NEW.balance := 100;
  END IF;

  RETURN NEW;
END;
$BODY$ LANGUAGE plpgsql;

CREATE TRIGGER chk_balance_interval BEFORE UPDATE ON account
FOR EACH ROW EXECUTE PROCEDURE chk_balance_interval_proc();
```

```
UPDATE account
SET balance = balance - 500
WHERE account_number = 'A-101';
```

Sintaxe

- Criação de um Trigger

```
CREATE TRIGGER <trigger_name>
   { BEFORE | AFTER } { INSERT | UPDATE | DELETE }

ON <tbl_name>

WHEN <condition>

FOR EACH { ROW | STATEMENT } EXECUTE PROCEDURE
   <proc>
```

- Remoção de um Trigger

```
DROP TRIGGER trigger_name
```

Tratamento de erros

- Se um trigger BEFORE falha, a operação sobre a linha ou tabela correspondente não é executada
- Um erro durante um trigger AFTER ou BEFORE resulta na falha da instrução completa que desencadeou o trigger
- Um trigger AFTER só é executado se quaisquer triggers BEFORE sobre a mesma tabela e relativos à mesma operação forem executados com sucesso

Trigger: Exemplo 2

- O cliente levanta uma quantia superior ao saldo

- Em vez de resultar num saldo negativo, o banco:
  - cria um empréstimo igual à quantia em falta
  - dá ao empréstimo o mesmo número que a conta
  - coloca o saldo da conta a zero

- Condição de disparo: um **update** que resulte em saldo negativo na conta

Exemplo 2

- Levantamento de 800 € da conta A-102

account_number	branch_name	balance
A-101	Downtown	500.0000
A-215	Metro	600.0000
A-102	Uptown	700.0000
A-305	Round Hill	800.0000
A-201	Uptown	900.0000
A-222	Central	550.0000
A-217	University	650.0000
A-333	Central	750.0000
A-444	Downtown	850.0000

Exemplo 2

- Levantamento de 800 € da conta A-102 cria um empréstimo

Account

account_number	branch_name	balance
A-101	Downtown	500.0000
A-215	Metro	600.0000
A-102	Uptown	700.0000
A-305	Round Hill	800.0000
A-201	Uptown	900.0000
A-222	Central	550.0000
A-217	University	650.0000
A-333	Central	750.0000
A-444	Downtown	850.0000

Loan

loan_number	branch_name	amount
L-17	Downtown	1000.0000
L-23	Central	2000.0000
L-15	Uptown	3000.0000
L-14	Downtown	4000.0000
L-93	Metro	5000.0000
L-11	Round Hill	6000.0000
L-16	Uptown	7000.0000
L-20	Downtown	8000.0000
L-21	Central	9000.0000
A-102	UpTown	100.0000

Depositor

customer_name	account_number
Johnson	A-101
Brown	A-215
Cook	A-102
Cook	A-101
Flores	A-305
Johnson	A-201
Iacocca	A-217
Evans	A-222
Oliver	A-333
Brown	A-444

Borrower

customer_name	loan_number
Iacocca	L-17
Brown	L-23
Cook	L-15
Nguyen	L-14
Davis	L-93
Brown	L-11
Gonzalez	L-17
Iacocca	L-16
Parker	L-20
Brown	L-21
Cook	A-102

Exemplo 2

```
CREATE OR REPLACE FUNCTION overdraft_proc()
RETURNS TRIGGER
AS $$
BEGIN
   IF NEW.balance < 0 THEN
     INSERT INTO loan
     VALUES (NEW.account_number,
             NEW.branch_name,
             NEW.balance);

     INSERT INTO borrower
     SELECT customer_name, account_number
     FROM depositor
     WHERE depositor.account_number = NEW.account_number;
```

```
CREATE TRIGGER overdraft_trigger
BEFORE UPDATE
ON account
FOR EACH ROW EXECUTE PROCEDURE overdraft_proc();
```

Problemas dos *Triggers*

- O seu efeito pode ser complexo e imprevisíveis
- Vários *triggers* podem ser accionados numa só operação
- A acção de um *trigger* pode activar um outro *trigger* (triggers recursivos)
  - Cadeias de eventos intermináveis

Problemas dos *Triggers*

- Execuções não intencionadas
  - Ex. quando a BD está a ser reposta a partir de *backup*
- Ocorrência de erros
  - Se o *trigger* falha, toda a operação falha
  - Tempos de recuperação dilatados

Quando não usar *triggers*

- Actualizar um total quando se insere/actualiza um registo
  - Usar vistas em vez de *triggers,* se possível

- Replicação de BDs
  - Usar mecanismos próprios do SGBD (só funciona se SGBDs forem do mesmo fabricante)

Sintaxe *Triggers* em SQL:1999

```
CREATE TRIGGER set_count
    AFTER INSERT ON Students
REFERENCING NEW TABLE AS InsertedTuples
FOR EACH STATEMENT
    INSERT INTO StatisticsTable
        (ModifiedTable, ModificationType, Count)
    SELECT 'Students', 'Insert', COUNT(*)
    FROM InsertedTuples I
    WHERE I.age < 18;
```

- Mantêm *log* das operações na StatisticsTable
- Elegante mas não suportado em todos os SGBD

CHECK vs. Triggers

- Clausula `CHECK`
  - São **declarativos**
  - Mais fáceis de entender
  - Mais eficientes
  - Suporte em POSTGRES (com limitações)

- *Triggers*
  - São **procedimentais**
  - Algumas restrições só podem ser implementadas através de triggers
  - Podem ser usados para outras objectivos
    - Logs, Warnings, Security

Excepções

Lançamento de Excepções

```
RAISE EXCEPTION <mensagem>
USING HINT 'texto'
```

```
CREATE FUNCTION search(uid INT) RETURNS VOID AS
$$
BEGIN
  IF NOT EXISTS(
            SELECT *
            FROM users
            WHERE user_id=uid)
  THEN
     RAISE EXCEPTION 'Nonexistent ID %', user_id
     USING HINT = 'Please check your user ID';
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Recuperação de Excepções

```
CREATE OR REPLACE FUNCTION test_excep(arg INTEGER)
RETURNS INTEGER
AS $$
DECLARE res INTEGER;
BEGIN
     res := 100 / arg;
     RETURN res;
EXCEPTION
     WHEN division_by_zero
     THEN RETURN 0;
END;
$$
LANGUAGE plpgsql;
```

Vistas

Vistas

Definidas com base numa instrução SELECT

`CREATE VIEW` *`myview`* `AS SELECT …`

- Uma vez criada, pode ser usada como uma relação
  - a vista exprime uma relação em interrogações
  - mas não é o mesmo que criar uma tabela

Criação de Vistas

- Uma vista é apenas uma relação, mas realizamo-la guardando a sua definição em vez de um conjunto de linhas

```
CREATE VIEW account_stats(name, num_accts)
AS SELECT name, count(*) AS num_accts
FROM depositor
GROUP BY customer_name
```

Remoção de Vistas

```
DROP VIEW BestStudents
```

Criação de Vistas: Exemplo

```
CREATE VIEW BestStudents(name, sid, course)
AS SELECT S.sname, S.sid, E.cid
FROM Students S, Enrolled E
WHERE S.sid = E.studid
AND E.grade = ’20'
```

```
SELECT COUNT(*)
FROM BestStudents
WHERE name = 'Jones'
```

Interrogações sobre vistas

- **Expansão de vistas** - técnica para avaliação de interrogações sobre vistas
  - Referências a uma vista são substituídas pela sua definição

Vistas, independência de dados e segurança

- Suporta a independência lógica dos dados do modelo relacional
- Úteis no contexto de segurança
  - Definir vistas que dão acesso, a um grupo de utilizadores, só à informação que eles estão autorizados a ver.

Actualização de vistas

- SQL:1999 distingue entre dois tipos de vistas:
  - Cujas linhas podem ser modificadas (**updatable views**)
    - Uma coluna de uma vista pode ser actualizada se for obtida a partir de exactamente uma tabela base e a chave primária da tabela base estiver incluída nas colunas da vista
  - E vistas onde novas linhas podem ser inseridas (**insertable views**)
  - Tem de existir uma relação de um para um entre as linhas da vista e as das respectivas tabelas base.

Vistas Materializadas

```
CREATE MATERIALIZED VIEW account_stats AS …
```

- Actualização automática (imediata ou diferida no tempo)
- Permite re-escrita sobre vistas

```
SELECT name, count(*)
FROM depositor d
GROUP BY name
HAVING count(*) > 2
```

>>

```
SELECT name, num_accts
FROM  account_stats
WHERE num_accts > 2
```