

PL/pgSQL

– Linguagem de Programação Estruturada com SQL

Prof. Dr. Ives Renê V. Pola

ivesr@utfpr.edu.br

Departamento Acadêmico de Informática – DAINF
UTFPR – Pato Branco

Apresentação da linguagem de programação estruturada PL/pgSQL
designada para a programação usando o PostgreSQL.

A linguagem de programação de consultas SQL

A linguagem de programação SQL as vezes tratada como uma Linguagem de Quarta-Geração, estende SQL para tratar:


- Variáveis e tipos;
- Estruturas de controle;
- Procedimentos e funções;
- Definir tipos de objeto e métodos;
- Fazer a adaptação entre SQL e as linguagens de programação imperativas.

Ela permite desenvolver programas segundo o paradigma imperativo (“procedural”) usando SQL, que segue o paradigma relacional.

A linguagem de programação de consultas SQL

Apesar de semelhantes ao padrão, cada gerenciador tem sua própria linguagem de programação de consulta:

- **ANSI/ISO Standard:** SQL/PSM (SQL/Persistent Stored Modules)
- **Oracle:** PL-SQL
- **Postgres:** PL/pgSQL
- **IBM DB2:** SQL PL
- **MS SQL Server, Sybase:** Transact-SQL
- **MySQL:** SQL/PSM

 A definição padrão visa a criação de procedimentos armazenados, mas cada produto estende o padrão para facilitar o desenvolvimento dos aplicativos, aumentando a coleção do que é “procedimento”.

A linguagem de programação de consultas SQL

Utilidades da **PL/pgSQL**

- Usada para definir triggers e stored procedures;
- Permite implementar estruturas de controle entre comandos SQL;
- Manipula todos tipos de dados usados em atributos e e seus operadores.

Vantagens da **PL/pgSQL**

- Roda inteiramente no servidor e evita sobrecarga de rede. Evita a sobrecarga de comunicação inter-processos e a sobrecarga na rede porque a programação via aplicação cliente receberia respostas de cada instrução SQL pela rede, e requisições futuras tráfegariam novamente ao servidor.
- Parsing de comandos previamente compilados.

A linguagem de programação de consultas **PL/pgSQL** para **PostgreSQL**

Recursos atendidos por **PL/pgSQL**:

- Estrutura em blocos;
- Declaração de Variáveis e Tipos;
- Tratamento de erros;
- Cursores;
- Estruturas de controle: Condições e Repetição;
- Procedimentos e funções;

Estrutura geral de um bloco

- Um módulo de programação é chamado um **Bloco de programa**, que pode ser uma *procedure*, uma função ou um **Bloco Anônimo**.
- Todo bloco de programa em **PL/pgSQL** tem uma estrutura em três partes:

Estrutura de um módulo **PL/pgSQL**

```
[<<Nome>>]
[DECLARE -- parte declaração
    <Variáveis, tipos, cursores, subprogramas>]
BEGIN -- parte executável
    <Instruções>
[EXCEPTION -- parte de tratamento de exceções
    <Tratamento de exceções>]
END [Nome];
```

- As partes executáveis e de tratamento de exceções podem conter qualquer número de sub-blocos aninhados;

Declaração de Variáveis

Declaração/Inicialização de Variáveis

```
DECLARE  
    nome [CONSTANT] tipo [NOT NULL]  
        [ DEFAULT | := | = expressão];
```

Exemplos: Declarar uma variável tipo `INTEGER` e outra tipo `VARCHAR`:

```
DECLARE user_id integer;  
        url varchar;
```

Declarar uma variável para receber o valor do atributo `Nome` da tabela `Aluno`:

```
DECLARE V_Nome Aluno.Nome%TYPE;
```

Declarar uma variável para receber uma linha da tabela `Aluno`:

```
DECLARE V_Aluno Aluno%ROWTYPE;
```

Atribuições em PL/pgSQL

Atribuições em PL/pgSQL

variável { := | = } expressão;

Exemplos

taxa := 0.05;

soma = 0;

- Se uma expressão não resultar no mesmo tipo de dado que a variável, o valor será convertido (*cast* dinâmico).
- Mas, se o gerenciador não conseguir converter automaticamente, usará algum método implementado no tipo que converta o dado, podendo gerar erros se não houver definido algum método que tenha parâmetro desejado.

Declaração de Variáveis

Atribuição de valores a variáveis

Existem três maneiras de atribuir valores a variáveis:

- 1 Usando o operador de atribuição `:=`:

```
nome := 'Jose da Silva';  
idade := CAST(Extract(year FROM NOW()) AS int)  
        - CAST(Extract(year FROM DataNasc) AS int);
```

- 2 Obtendo dados de uma relação com um comando `SELECT INTO`:

```
select nome into nomedep  
from departamento  
where id = 1;
```

- 3 Passando as variáveis como argumentos de um sub-programa.

Como usar um bloco anônimo

Assumindo que: create table departamento (id integer, nome varchar);
insert into departamento values (1, 'Financeiro');

```
DO $$  
declare  
  
idade integer;  
nomedep varchar;  
  
begin  
  
idade := CAST(Extract(year FROM NOW()) AS int) -  
CAST(Extract(year FROM DATE '23/06/1980') AS int);  
select nome into nomedep from departamento where id = 1;  
  
raise notice 'Idade:  ',idade;  
raise notice 'Departamento:  ',nomedep;  
  
end $$;
```

Declaração de Variáveis – Exemplos

Exemplos

Mostrar quantos alunos são de Pato Branco:

```
DO $$  
DECLARE  
    numeroalu INTEGER;  
    cidadealu Aluno.Cidade%TYPE := 'Pato Branco';  
BEGIN  
    SELECT count(*) INTO numeroalu  
        FROM Aluno  
        WHERE cidade=cidadealu;  
    raise notice 'Alunos de Pato Branco:  %',numeroalu;  
END $$;
```

- ✎ Essa construção requer que o comando **SELECT** retorne uma única tupla, caso contrário, um erro é lançado;
- ✎ Erros podem ser capturados no bloco **EXCEPTION**.

Declaração de Variáveis – Exemplos

Exibir o nome do aluno de RA=1 (com tratamento de erros):

```
DO $$  
DECLARE  
    nomealu varchar;  
    raalu Aluno.RA%TYPE:=1;  
BEGIN  
    SELECT nome INTO STRICT nomealu  
        FROM Aluno  
        WHERE ra=raalu;  
    raise notice 'Aluno: %',nomealu;  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        raise notice 'Não existe este aluno.';  
    WHEN TOO_MANY_ROWS THEN  
        raise notice 'Nesse caso não acontece...';  
    WHEN OTHERS THEN  
        raise notice 'Erro número: %. Mensagem:%', SQLSTATE, SQLERRM;  
END $$;
```

- STRICT gera erro se a consulta retornar mais de uma tupla, a ausência fará com que a primeira tupla do resultado será colocada na variável, ou senão NULO caso não retorne nada, e não gera erro.

Declaração de Variáveis – Exemplos

Obter os dados do aluno Zeca:

```
DO $$  
DECLARE  
    V_Aluno Aluno%ROWTYPE;  
BEGIN  
    SELECT * INTO STRICT V_Aluno  
        FROM Aluno  
        WHERE lower(Nome)='zeca';  
    raise notice 'Nome: %, RA: %, Cidade: %', V_Aluno.Nome,  
V_Aluno.RA, V_Aluno.Cidade;  
EXCEPTION  
    WHEN NO_DATA_FOUND THEN  
        raise notice 'Aluno não existe';  
    WHEN TOO_MANY_ROWS THEN  
        raise notice 'Mais de um Zeca.';  
END $$;
```

Consultas com uma única tupla no resultado

- Para executar comandos que retornem uma única tupla no resultado, ou mesmo um único valor, utiliza-se os comandos SQL acrescentando-se a cláusula INTO.
- Os planos de consulta ficarão no cache, e eles são definidos da seguinte maneira:

Sintaxe

SELECT <atribos> **INTO** [**STRICT**] <variavel> **FROM** ...

INSERT ... **RETURNING** <expressões> **INTO** [**STRICT**] <variavel>;

UPDATE ... **RETURNING** <expressões> **INTO** [**STRICT**] <variavel>;

DELETE ... **RETURNING** <expressões> **INTO** [**STRICT**] <variavel>;

Cláusula RETURNING..

- A cláusula RETURNING em um DML serve para mostrar os dados inserido/atualizado/excluído, de acordo com a expressão passada a ele. Por exemplo, seja:

```
funcionario ( ID bigserial, nome varchar(30) )
```

- O Comando

```
insert into funcionario (nome) values ('Juca')  
returning ID;
```
- Insere uma tupla em funcionario e a exibe o resultado da inserção.
- É muito útil para descobrir os valores de campos *auto-increment*, como ID no exemplo dado.

Comando WITH ... AS

- O comando WITH é usado para guardar o resultado de uma expressão DML em uma variável.
- Por exemplo, o código a seguir:

```
WITH temp AS
  (UPDATE funcionario
   SET vendas = vendas + 1
   WHERE ID = 143310
   RETURNING *)
INSERT INTO vendas_log
  SELECT ID, current_timestamp FROM temp;
```

- Atualiza o número de vendas de um funcionário específico e armazena no log de vendas quando esse fato aconteceu, gerando um histórico de vendas para funcionários.

Funções

- Uma função ou um Procedimento são chamados de “**Subprogramas**”.
- Possuem a estrutura:

```
<Nome Subprograma ([<parametro1>, <parametro2>...]);
```

- Uma função é chamada em uma expressão, como por exemplo:

```
Variavel := funcao (variavel1, variavel2);
```

Funções

- A sintaxe do comando de criação de uma Função é:

```
CREATE [OR REPLACE] FUNCTION
  <Nome> ( [modo] [nome] tipo [ DEFAULT expressão] [,...])
  [RETURNS tipo | TABLE (nome tipo [,...]) ]
  { LANGUAGE linguagem }
END <Nome function>;
```

- Uma função deve ter ao menos um comando **RETURN**.
- A cláusula [OR REPLACE] é muito útil para redefinir funções de modo que todas visões ou triggers que dependam delas não sejam afetadas. Se excluirmos e criarmos outra função, a identidade será outra, e os objetos que dependam dela não funcionarão e serão desabilitadas.

Exemplo de função

```
CREATE OR REPLACE FUNCTION soma (a integer, b integer)
RETURNS INTEGER AS $$
BEGIN
    return a+b;
END;
$$ LANGUAGE plpgsql;
```

```
CREATE OR REPLACE FUNCTION soma (integer, integer)
RETURNS INTEGER AS $$
DECLARE
    param1 ALIAS FOR $1;
    param2 ALIAS FOR $2;
BEGIN
    return param1 + param2;
END;
$$ LANGUAGE plpgsql;
```

Parâmetros de funções

- Funções podem ser declaradas com vários parâmetros de saída – OUT, diretamente no cabeçalho da função.
- Por exemplo:

```
CREATE FUNCTION soma2 (a integer, b integer, OUT soma integer,  
OUT prod integer) as $$  
BEGIN  
    soma := a + b;  
    prod := a * b;  
END;  
$$ LANGUAGE plpgsql;
```

- Isso é muito útil quando se quer retornar vários valores.

Retorno de funções

- Nas funções, também pode-se retornar tabelas.
- Para isso, utilize RETURNS TABLE

```
CREATE OR REPLACE FUNCTION Vendas (itemID int)
RETURNS TABLE (quantidade int, total numeric) as $$
BEGIN
    RETURN QUERY SELECT V.quantidade, V.quantidade*V.preco
                  FROM vendas as V
                  WHERE V.itemno = itemID;
END;
$$ LANGUAGE plpgsql;
```

- Isso é muito útil quando se quer retornar vários valores.
- Para o exemplo dado acima, considere a tabela:
create table vendas (itemno integer, quantidade integer, preco numeric);

Outro exemplo de função

```
CREATE FUNCTION fatorial (n INTEGER) RETURNS INTEGER AS $$
DECLARE
fat integer DEFAULT 1;
BEGIN
    FOR i IN REVERSE n..1 LOOP
        fat := fat*i;
    END LOOP;
    RETURN fat;
END;
$$ LANGUAGE plpgsql;

DO $$ BEGIN
    FOR i IN 1..5 LOOP
        RAISE NOTICE 'Fatorial( % ) = %', i, fatorial(i);
    END LOOP;
END $$;
```

RETURN NEXT e RETURN QUERY

Os comandos:

- RETURN NEXT <expressão>
- RETURN QUERY <consulta>

São usados para acrescentar tuplas no resultado, cada um de uma maneira.

- RETURN NEXT aceita expressões com escalares e subconsultas também, e acrescenta a tupla no resultado enquanto a função está executando.
- RETURN QUERY acrescenta o resultado de uma consulta no resultado, também enquanto a função está executando.

A função ainda precisará de um comando RETURN para finalizá-la e retornar o resultado geral.

Veja o Exemplo:

Exemplo de SETOF - RETURN NEXT

```
CREATE TABLE aluno (ra INT, nome varchar, datanasc date);
INSERT INTO aluno VALUES (1, 'Pedro', to_date('12-02-1995','DD-MM-YYYY') );
INSERT INTO aluno VALUES (2, 'Juca', to_date('20-03-1989','DD-MM-YYYY') );
```

```
CREATE OR REPLACE FUNCTION todos_alunos() RETURNS SETOF aluno
AS
$$
DECLARE
    r aluno%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM aluno WHERE ra > 0
    LOOP
        -- pode-se fazer algum processamento aqui
        RETURN NEXT r; -- retorna tupla atual do SELECT
    END LOOP;
    RETURN;
END
$$
LANGUAGE plpgsql;

SELECT * FROM todos_alunos();
```


Exemplo de Returns table

- A função abaixo retorna uma tabela calculando as idades atuais de cada aluno.

```
create function get_idades()  
RETURNS table (nome varchar, idade double precision) AS $$  
begin  
    return query select A.nome,extract( year from  
        age(A.datanasc) ) from aluno as A;  
end;  
$$ language plpgsql;
```

- Isso é útil para utilizarmos em consultas que as utiliza na cláusula FROM.
- Por exemplo:

```
select nome,idade from get_idades()
```

Condicionais

- O comando IF .. THEN .. ELSE executa os comandos dependendo da condição de teste ser verdadeira. Por exemplo:

```
DO $$
declare
    reg integer;
begin
    select ra into reg from aluno where nome = 'Juca';
    IF reg <> 0 THEN
        update aluno
        set datanasc = to_date('20-03-1990', 'DD-MM-YYYY')
        where ra = reg;
    END IF;
END $$
```

Condicional IF

- Atente-se que em plpgsql a sintaxe completa do condicional IF é:

```
IF number = 0 THEN
    result := 'zero';
ELSIF number > 0 THEN
    result := 'positive';
ELSIF number < 0 THEN
    result := 'negative';
ELSE
    result := 'NULL';
END IF;
```

Comando CASE

- O comando CASE executa os comandos de forma condicional baseado em expressões booleanas.
- Por exemplo, veja uma utilização dele:

```
CASE
  WHEN x BETWEEN 0 AND 10 THEN
    msg := 'valor está entre zero e dez';
  WHEN x BETWEEN 11 AND 20 THEN
    msg := 'valor está entre onze e vinte';
END CASE;
```

Estruturas de repetição

- Uma estrutura de repetição pode ser feita usando o comando LOOP, ele repete infinitamente o bloco de comando no interior dele, ao menos que encontre um comando EXIT, que força a saída do laço.

```
LOOP
    -- alguns cálculos podem vir aqui
    IF count > 0 THEN
        EXIT; -- força sair do laço
    END IF;
END LOOP;
```

```
LOOP
    -- alguns cálculos podem vir aqui
    EXIT WHEN count > 0; -- igual exemplo acima
END LOOP;
```

CONTINUE

- Pode-se interromper a execução dentro de um laço e forçar a próxima iteração através do comando CONTINUE.
- Por exemplo:

```
LOOP
-- alguns cálculos aqui
EXIT WHEN count > 100;
CONTINUE WHEN count < 50;
-- Daqui para frente apenas executará
-- se valor de count estiver entre [50 .. 100]
<comandos>
END LOOP;
```

WHILE

- O comando while executa o corpo de comandos enquanto a condição associada a ele for verdadeira.
- A checagem da condição é feita no início de cada repetição.
- A sintaxe do comando é:

```
WHILE <condição> LOOP  
    <comandos>  
END LOOP
```

FOR

- Cria um loop que repete a execução de um bloco de comandos.
- Veja alguns exemplos de utilização:

```
FOR i IN 1..5 LOOP
    -- Aqui a variável i irá ter os valores 1,2,3,4,5
END LOOP;
```

```
FOR i IN REVERSE 5..1 LOOP
    -- Aqui a variável i irá ter os valores 5,4,3,2,1
END LOOP;
```

```
FOR i IN REVERSE 5..1 BY 2 LOOP
    -- Aqui a variável i irá ter os valores 5,3,1
END LOOP;
```


Vetores e Matrizes

- O tipo array pode ser usado para manipular vetores e matrizes.
- Por exemplo, crie a função abaixo:

```
CREATE FUNCTION scan_rows(int[]) RETURNS void AS $$  
DECLARE  
    x int[];  
BEGIN  
    FOREACH x SLICE 1 IN ARRAY $1  
    LOOP  
        RAISE NOTICE 'row = %', x;  
    END LOOP;  
END;  
$$ LANGUAGE plpgsql;
```

- Agora use a função desta maneira (mensagens na aba Messages):

```
SELECT scan_rows(ARRAY[[1,2,3],[4,5,6],[7,8,9],[10,11,12]]);
```

Cursosores – Conceitos

- Ao invés de executar toda a consulta de uma só vez, um **ponteiro** para a **tupla corrente**, chamado **cursor**, pode ser criado e recuperar poucas tuplas por vez na memória.
- Um cursor pode ser **implícito** ou **explícito**.
- PL/pgSQL declara um cursor implícito para todo comando DML usado dentro de um laço FOR.
- Um cursor explícito é declarado associado a um comando DML.

```
nomecursor [[NO] SCROLL] CURSOR (argumentos) FOR query;
```

- Por exemplo:

```
DECLARE  
    curs1 refcursor;  
    curs2 CURSOR FOR SELECT * FROM aluno;  
    curs3 CURSOR (chave integer) FOR SELECT * FROM  
aluno WHERE ra = chave;
```

- O conjunto de tuplas retornados por um cursor é chamado **Conjunto-Resposta** (*result-set*).

Cursor Explícito – Conceitos

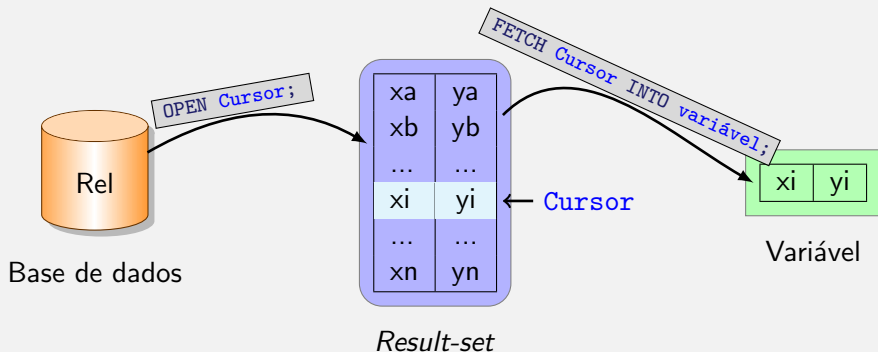
- Para usar um cursor, ele deve ser:

OPEN – Executa o comando associado ao cursor, recupera o *result-set* e posiciona o cursor antes da primeira tupla: `OPEN <Cursor>;`

FETCH – Recupera a tupla corrente e avança o cursor para a tupla seguinte do *result-set*; `FETCH <Cursor> INTO <variáveis>;`

CLOSE – Desabilita o cursor. `CLOSE <Cursor>;`

Cursor Explícito – Conceitos



Cursor Explícito - Exemplo

```
do $$  
DECLARE  
    C_Alunos CURSOR FOR SELECT * FROM Aluno;  
    V_Aluno record;  
BEGIN  
    OPEN C_Alunos; --Abre o cursor e obtém o Conjunto-resposta  
    LOOP  
        FETCH C_Alunos INTO V_Aluno; --recupera uma tupla  
        --Sai do laço quando não tem mais tuplas:  
        EXIT WHEN V_Aluno IS NULL;  
        raise notice 'RA: %, Nome: %', V_Aluno.ra, V_Aluno.Nome;  
    END LOOP;  
    CLOSE C_Alunos; --Fecha o cursor  
END; $$  
language plpgsql;
```

Cursors de laço FOR

- É frequente que o acesso a uma tabela possa ser feito simplesmente percorrendo todas as tuplas de um *result-set*.
- Isso pode ser feito usando um laço FOR, definido como:

```
FOR <recordvar> IN <Cursor> LOOP  
    <Corpo do FOR>  
END LOOP;
```

- A variável recordvar deve ser do tipo RECORD, e o cursor não pode ter sido aberto antes do FOR. O FOR nesse caso executa automaticamente sobre o cursor as operações OPEN, FETCH e CLOSE.
- Os campos individuais da tupla podem ser acessados usando a notação `tupla.atributo`.

Cursosos de laço FOR – Exemplo

Executar um determinado processamento sobre os atributos **RA**, **Nome** e **Idade** de todos os alunos de Pato Branco.

```
DO $$  
DECLARE  
    C_Aluno CURSOR FOR  
        SELECT RA, Nome, Extract(year from age(Datanasc)) AS Idade  
        FROM Aluno  
        WHERE Cidade='Pato Branco';  
BEGIN  
    FOR alu IN C_Aluno LOOP  
        raise notice 'Idade de %: %', alu.Nome, alu.Idade;  
    END LOOP;  
END; $$  
language plpgsql;
```

Variáveis Globais

Existem variáveis que podem ser consultadas para saber o que acontece nos comandos executados:

FOUND – Retorna **TRUE** se a última SQL executada alterou/selecionou alguma tupla e **FALSE** caso contrário;

ROWCOUNT – Número de linhas alteradas pelo comando anterior;
👉 GET DIAGNOSTICS count = ROW_COUNT

Visões

- Visões são consultas que geram tabelas temporárias a partir de uma consulta SELECT.
- Elas são úteis para gerar porções das tabelas e disponibilizá-las a outros usuários para terem acesso apenas aos dados que desejarmos.
- Por exemplo:
CREATE VIEW NomesDatas AS SELECT nome, datanasc from Aluno;
- Nesse caso uma visão apenas projetará o nome e a data de nascimento, gerando uma tabela temporária quando a visão é utilizada em consultas em cláusulas FROM.

Visões Materializadas Automáticas

- É possível criar visões como tabelas e já inserir tuplas nelas.
- O comando para criar visões materializadas é:

```
CREATE MATERIALIZED VIEW  
table_name [ (column_name [, ...] ) ]  
[ TABLESPACE tablespace_name ]  
AS query  
[ WITH [ NO ] DATA ]
```

- Existem limitações na criação deste tipo de visão, pois não podem haver comandos group by, junções, entre outras limitações.

Visões Materializadas Automáticas

- A consulta na view é executada e a tabela alvo é populada de modo imediato, a não ser que o comando WITH NO DATA seja emitido.
- Só que a visão materializada não é atualizada automaticamente.
- Deve-se executar o comando:

```
REFRESH MATERIALIZED VIEW [CONCURRENTLY] nomeview
```

- Concurrently: Atualiza a visão materializada sem bloquear selects concorrentes nela. Sem esta opção a atualização termina mais rápido porém bloqueia muitas consultas que querem ler desta visão.

Visões Materializadas Manuais

- Trabalhamos com visões materializadas quando queremos persistir os dados provenientes de uma consulta que pode envolver mais de uma tabela em uma tabela e armazenar esses dados no disco.
- Um problema é que temos que atualizar constantemente estas tabelas em disco, quando atualizações ocorrerem nas tabelas que elas acessam.
- Por exemplo, se quisermos trabalhar com uma visão da tabela aluno, reduzida:
create table aluno2 (nome varchar, datanasc date);
- Veja que esta relação está vazia e não possui o campo RA, que queremos proteger.

Atualizando Visões Materializadas

- Para atualizarmos esta visão materializada de maneira automática, usamos Triggers.
- Uma trigger executa uma função e ela é disparada por algum evento como por exemplo atualizações ou inserções.
- Para criarmos uma trigger precisamos ter uma função que execute os comandos que queremos, e depois criamos a trigger para executar esta função após um evento ocorrer.
- Por exemplo. Queremos que a trigger execute uma função quando alguma tupla for inserida na tabela aluno. Além disso, queremos que a função copie os dados para a tabela Aluno2, nossa visão materializada, automaticamente.
- Vamos ver o código:

Usando Triggers

```
CREATE OR REPLACE FUNCTION atualizaAluno2()  
RETURNS trigger AS $$  
BEGIN  
    INSERT INTO aluno2 VALUES (NEW.nome, NEW.datanasc);  
    RETURN NEW;  
END;  
$$ LANGUAGE plpgsql;  
  
CREATE TRIGGER TrigAluno2 BEFORE INSERT ON Aluno  
FOR EACH ROW EXECUTE PROCEDURE atualizaAluno2();
```

- O comando: insert into aluno values (3,'Joao',to_date('23-05-1986','DD-MM-YYYY'))
- Agora também adicionará cópia dos valores para a tabela Aluno2.

Auditoria de forma automática

- Podemos realizar uma auditoria de forma automática, armazenando em uma tabela todas operações que acontecem em uma tabela alvo.
- Por exemplo, considere as tabelas:

```
create table func ( nome varchar, salario numeric);  
  
create table audit_func (operacao char, data timestamp,  
usuario varchar, nomefunc varchar, salario numeric);
```

- Toda operação na tabela de funcionarios func, queremos registrar na tabela de auditoria de modo automático.
- Para isso escrevemos uma trigger que chama uma função para isso.

Auditoria de forma automática

```
CREATE OR REPLACE FUNCTION process_func_audit()
RETURNS TRIGGER AS $func_audit$
BEGIN
IF (TG_OP = 'DELETE') THEN
    INSERT INTO audit_func SELECT 'D', now(), user, OLD.*;
    RETURN OLD;
ELSIF (TG_OP = 'UPDATE') THEN
    INSERT INTO audit_func SELECT 'U', now(), user, NEW.*;
    RETURN NEW;
ELSIF (TG_OP = 'INSERT') THEN
    INSERT INTO audit_func SELECT 'I', now(), user, NEW.*;
    RETURN NEW;
END IF;
RETURN NULL; - resultado é ignorado pois será uma AFTER trigger
END;
$func_audit$ LANGUAGE plpgsql;

CREATE TRIGGER func_audit
AFTER INSERT OR UPDATE OR DELETE ON func
FOR EACH ROW EXECUTE PROCEDURE process_func_audit();
```


Tipos de Triggers

- As triggers podem ser do tipo:
 - 1 BEFORE
 - 2 AFTER
 - 3 INSTEAD OF
- E podem disparar em eventos de:
 - 1 INSERT
 - 2 UPDATE
 - 3 DELETE
 - 4 TRUNCATE
- Podem ser a nível de ROW ou a nível de STATEMENT
 - 1 FOR EACH ROW
 - 2 FOR EACH STATEMENT

Trigger BEFORE UPDATE

- A seguinte trigger verifica os dados antes da atualização ou inserção na tabela de funcionários, evitando erros provenientes de formulários, caso não sejam tratados na aplicação.

```
CREATE TRIGGER checa_update  
BEFORE UPDATE OR INSERT ON funcionario  
FOR EACH ROW  
EXECUTE PROCEDURE checa_func_dados();
```

Trigger BEFORE UPDATE

- A função de checagem dos dados seria como:

```
CREATE FUNCTION checa_func_dados() RETURNS trigger AS $$
BEGIN
IF NEW.nome IS NULL THEN
    RAISE EXCEPTION 'Nome funcionario vazio';
END IF;
IF NEW.salario IS NULL THEN
    RAISE EXCEPTION '%: Salário vazio', NEW.nome;
END IF;
IF NEW.salario < 0 THEN
    RAISE EXCEPTION '%: Salário negativo', NEW.nome;
END IF;
NEW.data_cadastro := current_timestamp;
NEW.usuario_cadastrou := current_user;
RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

Roteiro

- 1 Introdução
- 2 Declaração de Variáveis
- 3 Comandos básicos em PL/pgSQL
- 4 Funções em PLPGSQL
- 5 Cursores
- 6 Triggers e Visões

PL/pgSQL

– Linguagem de Programação Estruturada com SQL

Prof. Dr. Ives Renê V. Pola

ivesr@utfpr.edu.br

Departamento Acadêmico de Informática – DAINF
UTFPR – Pato Branco

FIM

