# *SEQUENCE & STORE PROCEDURES*

Bruno A. N. Travençolo – FACOM

# SEQUENCES

- SEQUENCES são usados para gerar automaticamente números sequenciais únicos.
  - Ex: 1,2,3,4,5,....
- Podemos usar essas sequências para atribuir valores automaticamente para atributos de tabelas
  - Ex: chave primária.
- SINTAXE

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name
[ INCREMENT [ BY ] increment ]
[ MINVALUE minvalue | NO MINVALUE ]
[ MAXVALUE maxvalue | NO MAXVALUE ]
[ START [ WITH ] start ] [ CACHE cache ]
[ [ NO ] CYCLE ]
```

# SEQUENCES

```
/* cria uma sequencia iniciando em 1000 e incrementada em 1 */
-- DROP SEQUENCE Seq
CREATE SEQUENCE Seq
   START WITH 1001
   INCREMENT BY 1

-- testando a sequencia (rodar várias vezes)
SELECT NEXTVAL('Seq');
-- testando o valor ATUAL DA sequencia
SELECT CURRVAL('Seq'); -- CURRENT VAL

-- usando com INSERT INTO
CREATE TABLE teste(n int);
INSERT INTO teste VALUES (NEXTVAL('Seq'));

INSERT INTO Aluno VALUES('JOSE DA SILVA',
NEXTVAL('Seq'), 21, 'Araguari');
```

# FUNCTIONS

- O PostgreSQL permite ao desenvolvedor estender as funcionalidade do servidor de BD por meio da criação de novas função utilizando a linguagem C e carregando-as quando o servidor é iniciado.

- Uma extensão pode ser uma função bem simples, ou complexa como uma linguagem de programação. Essas linguagens permitem criar nossas próprias funções, conhecidas como **stored procedures**, de forma mais rápida e fácil que se fossem escritas em C.

# FUNCTIONS

- Comando SQL: CREATE FUNCTION

CREATE FUNCTION *nome-da-função ([<tipo_dado> [, ...] ] )*
RETURNS *<tipo_dado>*
AS *<definição>*
LANGUAGE *'nome_linguagem'*

*<definição>* é dada por uma string que pode conter várias linhas e pode ser escrita em qualquer linguagem suportada pelo PostgreSQL e carregada como uma linguagem procedural. *A linguagem* PL/pgSQL foi desenvolvida especialmente para criação de stored procedures no PostgreSQL

# FUNCTIONS

- Outras linguagens procedurais que podem ser usadas. Exemplos:
  - PL/Tcl
  - PL/Perl
  - PL/Python

- Quando uma função é criada, sua definição é armazenada no BD. Quando a função é chamada pela primeira vez, a definição é compilada e executada. Isso implica **que o programador não será avisado de erros enquanto não usar a função** (exceto para certos erros de sintaxe).

# Exemplo

```
CREATE LANGUAGE plpgsql;

CREATE FUNCTION add_one (int4)
RETURNS int4 as '
BEGIN
   return $1 + 1; -- $1 é a representação para o 1o. argumento
END;' language 'plpgsql';

-- Como chamar o store procedure SELECT +
   nome + parâmetros
SELECT add_one(47);
```

# PL/pgSQL stored procedures

- PL/pgSQL é uma linguagem estruturada por blocos. Sintaxe

```
[<<label>>]
[DECLARE declarations]
BEGIN
      statements
END;
```

Isso é um bloco. Ele contém um rótulo (*label* – opcional), uma seção de declaração de variáveis (DECLARE), e a declaração de início e fim de bloco (BEGIN-END)

```
-- For PostgreSQL 8.0 and later
CREATE FUNCTION name ( [ [arg] ftype [, ...] ] )
RETURNS rtype
AS $$
block definition
$$ LANGUAGE plpgsql;
```

# Argumentos da Função

- Uma função pode ter 0 ou mais parâmetros
- No corpo da função os parâmetros são referidos como $1, $2,..

```
-- geom_avg
-- Obtém a média geométrica de dois inteiros
CREATE FUNCTION geom_avg(int4, int4) RETURNS float8
AS
$$
BEGIN
   RETURN SQRT($1 * $2::float8);
END;
$$ language plpgsql;
```

# Argumentos da Função

- A partir do Postgre 8.0 é possível dar nomes para os parâmetros
- Toda *store procedure* deve retornar um valor
  - Mas pode retornar VOID

```
-- geom_avg
-- Obtém a média geométrica de dois inteiros
CREATE FUNCTION geom_avg(valor1 int4, valor2 int4)
RETURNS float8 AS
$$
BEGIN
   RETURN SQRT(valor1 * valor2::float8);
END;
$$ language plpgsql;
```

# Declarações de variáveis

- Declaração:

  *nome [CONSTANT] tipo [NOT NULL] [{ DEFAULT | := } value];*

- Exemplo

  - n INTEGER := 2; -- declaração com inicialização
  - Valor_de_PI CONSTANT FLOAT8 := pi();

# Declarações de variáveis - TIPOS

- Copiando tipos
- O tipo da variável pode ser especificado utilizando um tipo de outro item da base de dados

*nome_da_variável%TYPE*

- Exemplos

Variável declarada anteriormente

```
n INTEGER := 2
```

Declarando outra variável do mesmo tipo

```
n2  n%TYPE; -- n2 é do mesmo tipo tipo n, ou seja, inteiro.
```

# Declarações de variáveis - TIPOS

- O tipo da variável pode ser especificado utilizando um tipo já definido em um atributo de uma tabela

*nome_da_tabela.nome_da_coluna%TYPE*

- Exemplos

Tabela empregado

```
CREATE TABLE empregado(nome VARCHAR(255),
                       salario DECIMAL(10,2)..
```

Declarando uma variável baseando-se no tipo de um atributo

```
Soma_Salarios  empregado.salario%TYPE; -- Soma_Salarios é do mesmo tipo tipo do atributo salário da tabela empregado, ou seja, DECIMAL(10,2)
```

# Declarações de variáveis

- A variável é estruturada em blocos (como em C++)

```
DECLARE
n1 integer;
n2 integer;
BEGIN
        -- pode-se usar n1 e n2 aqui
        n2 := 1;
        DECLARE
        n2 integer; -- esconde a variável n2 definida anteriormente
        n3 integer;
        BEGIN
                -- n1, n2 e n3 estão disponíveis aqui
                n2 := 2;
        END;
        -- n3 não está mais disponível
        -- n2 ainda possui valor 1
END;
```

# SELECT INTO

- Variação do comando SELECT que permite atribuir o resultado de uma consulta a uma variável
- Vale lembrar que essa variação é própria para *store procedures*
- *Sintaxe:*

  SELECT expressão INTO variável [FROM ..];

  Obs: existe o comando SELECT INTO fora do pl/pgSQL. Ele serve para criar tabelas a partir de uma consulta. Mas não é recomendado. Veja mais em:

  http://www.postgresql.org/docs/8.1/static/sql-selectinto.html

# Exemplo 2

```
-- obtém o próximo número de matrícula (NMat)
CREATE OR REPLACE FUNCTION NOVO_NMat ()
RETURNS int4 AS
$$
DECLARE UltimoNumero INTEGER;
BEGIN
   SELECT max(NMat)
   FROM ALUNO
   INTO UltimoNumero; -- joga max(Nmat) na variável UltimoNumero
   RETURN (UltimoNumero + 1);
END;
$$ language 'plpgsql';
```

# Estruturas de Controle

- Return: retorna um valor de uma função:
  RETURN expression;

```
-- obtém o próximo número de matrícula (NMat)
CREATE FUNCTION NOVO_NMat ()
RETURNS int4 AS
$$
DECLARE UltimoNumero INTEGER;
BEGIN
   SELECT max(NMat) FROM ALUNO INTO UltimoNumero;
   RETURN (UltimoNumero + 1);
END;
$$ language 'plpgsql';
```

# Exceções e Mensagens

- Ao encontrar uma condição que impossibilita continuar a execução de uma *store procedure* pode-se usar exceções

RAISE *level 'format' [, variable ...];*

**Table 10-11.** *PostgreSQL Exception Levels*

| Level | Behavior |
|---|---|
| `DEBUG, LOG, INFO` | Writes a message in the log (usually suppressed) |
| `NOTICE, WARNING` | Writes a message in the log and sends it to the application |
| `EXCEPTION` | Writes a message in the log and terminates the stored procedure |

# Exceções e Mensagens

- Exemplo
  - DEBUG

```
-- Escreve o valor da variável n em um log

CREATE FUNCTION scope()
RETURNS integer AS
$$
DECLARE n INTEGER := 4;
BEGIN
  RAISE DEBUG  'O valor de N é %', n;
  RETURN (n);
END;
$$ language 'plpgsql';
```

# Exceções e Mensagens

- Exemplo
  - NOTICE

```
-- Mostra o valor da variável n (envia a mensagem para o aplicativo)
   e continua a execução do procedure


CREATE OR REPLACE FUNCTION scope()
RETURNS integer AS
$$
DECLARE n INTEGER := 4;
BEGIN
  RAISE NOTICE  'O valor de N é %', n;
  RETURN (n);
END;
$$ language 'plpgsql';
```

# Exceções e Mensagens

- Exemplo
  - EXCEPTION

```
-- Mostra o valor de n (escreve no log) e TERMINA A EXECUÇÃO NO PONTO


CREATE OR REPLACE FUNCTION scope()
RETURNS integer AS
$$
DECLARE n INTEGER := 4;
BEGIN
    RAISE EXCEPTION  'O valor de N é %', n;
   RETURN (n);
END;
$$ language 'plpgsql';
```

# Condicionais

- **IF-THEN-ELSE**

```
IF expression
THEN
        statements
[ELSE
        statements]
END IF;
```

- **NULLIF(*input, value)***
  - *retorna NULL se input=value*
- ***CASE***

```
CASE
    WHEN expression
    THEN expression
    ...
ELSE expression
END;
```

```
res := CASE
    WHEN n2 = 1
        THEN 5
    WHEN n2 = 2
        THEN 6
    ELSE 7
END;
```

# Condicionais

IF *boolean-expression* THEN
  *statements*
[ ELSIF *boolean-expression* THEN
   *statements*
[ ELSIF *boolean-expression* THEN
  *statements*
   ...]]
[ ELSE
  *statements* ]
END IF;

# LOOPS

- LOOP

```
[<<label>>]
LOOP
    statements
END LOOP;
```

  - Para sair do loop

```
EXIT [label] [WHEN
expression];
```

  - Exemplo

```
<<indefinite>>
LOOP
      n := n + 1;
EXIT indefinite WHEN n >= 10;
END LOOP;
```

# LOOPS

- WHILE

```
[<<label>>]
WHILE expression
LOOP
    statements
END LOOP;
```

- FOR

```
FOR name IN [REVERSE]
from .. to
LOOP
    statements
END LOOP;
```

```
[ <<label>> ]
FOR target IN query
LOOP
    statements
END LOOP [ label ];
```

# LOOPS

- FOR – exemplo para percorrer todas as tuplas

```
DECLARE
    tupla record;
    cid integer;
…
FOR cid IN 1 .. 15
LOOP
    SELECT * INTO tupla – joga o resultado em tupla
     FROM clientes
     WHERE cliente_id = cid;
     -- processamento sobre o cliente
      RAISE NOTICE  ‘Cliente: %', tupla.nome;
…
END LOOP;
```

# LOOPS

- FOR – exemplo para percorrer todas as tuplas

```
DECLARE
    tupla record;  _cliente integer
….
SELECT COUN  *)
FROM clientes
INTO n_clientes  obtém o número de clientes

FOR cid IN 1 ..  n_clientes
LOOP
   SELECT * INTO tupla-- joga o resultado em tupla
    FROM clientes
    WHERE cliente_id = cid;
     -- processamento sobre o cliente
     RAISE NOTICE  ‘Cliente: %', tupla.nome;
…
END LOOP;
```

# L

- tuplas

```
                                    oga o resultado em tupla
    WHERE cliente_id = cid;
    -- processamento sobre o cliente
     RAISE NOTICE  'Cliente: %', tupla.nome;
...
END LOOP;
```

# LOOPS

- FOR – exemplo

```
-- listar o nome e idade de todos os alunos via mensagem no programa aplicativo

DECLARE
    tupla record;
BEGIN
FOR tupla IN SELECT * FROM aluno
LOOP
   RAISE NOTICE 'Nome: %, Idade: %', tupla.nome, tupla.idade;
END LOOP;
END;
```

# LOOPS

- OBS: no FOR com SELECT, o comando SELECT é feito somente uma vez e não a cada passo do LOOP. Você pode, inclusive apagar os dados da tabela que estão no SELECT do loop e continuar o loop normalmente