



Disciplina: Laboratório de Banco de Dados - BD

Funções e Stored Procedures



Objetivo

- Compreender a importância e necessidade de uso de Stored procedure
- Aprender a criar Stored procedure em linguagem SQL
- Aprender uma linguagem procedural sql
- Diferenciar Stored procedure e funções.

Introdução

- **Stored procedures (SP)** (procedimentos armazenados) são um importante recurso suportado pelos SGBDs há vários anos
- **Stored Procedures:** conjunto de código SQL armazenados no servidor do SGBD que podem ser invocados a qualquer momento por meio de um comando especial

Vantagens(1)

- **Centralização:** Em casos onde o banco de dados é acessado por diversas aplicações desenvolvidas em diferentes linguagens e por diferentes equipes, com o uso de SPs as alterações e regras são mantidos em um único local.
- **Segurança :** as vezes é necessário utilizar alguns recursos de segurança para evitar que usuários e desenvolvedores comentam erros.
- **Suporte à transação:** permite garantir que as transações que sejam interrompidas de alguma forma gerem inconsistência no banco de dados. Desta forma nenhuma transação ocorre pela metade

Vantagens(2)

● Desempenho:

- Um plano de execução para a SP é traçado apenas na primeira vez em que ela é executada, nas utilizações subsequentes o mesmo plano de execução é utilizado evitando assim o *re-parsing* dos comandos SQL.
- É possível empacotar várias instruções SQL em uma SP evitando um aumento na comunicação inter-processos e também um aumento no tráfego da rede.

- **Abrangência da linguagem SQL:** como é possível utilizar comandos de controle de fluxo e iterações estendeu-se o poder da linguagem SQL.

Stored Procedures(SP) X Funções

- Apesar de serem semelhantes por definição: conjunto de blocos de código SQL armazenados no servidor do SGBD que podem ser invocados a qualquer momento
- Funções e Stored procedures se divergem pelo seu uso:
 - Funções □ objetivo de realizar pequenas operações, normalmente auxiliares que possam ser solicitadas em uma transação.
 - Responsáveis por tratamento de texto, variáveis, formatação, operações repetitivas e rotineiras
 - Podem ser chamadas dentro de uma consulta em SQL
 - SP □ objetivo de realizar transações completas que garantem regras de negócios, não tem obrigação de retornar valores para o usuário

Implementação de Stored Procedures no Postgresql

- No PostgreSQL funções e Stored procedures são diferenciadas apenas pelo conceito de uso.
- A sintaxe para criação de funções e Stored procedures é a mesma.
- Em geral, em outros SGBDs, Stored procedures e funções são objetos distintos.

Implementação de Stored Procedures no Postgresql

- O PostgreSQL tem suporte para funções definidas pelo usuário, que podem ser escritas em outra linguagem diferente de SQL.
 - Estas outras linguagens são denominadas genericamente de “PROCEDURAL LANGUAGES (PL)”.
 - Existem quatro procedural languages disponibilizadas por padrão:
 - PL/pgSQL
 - PL/Tcl
 - PL/Pearl
 - PL/Python
- Obs: No Windows há atualmente somente a PL/pgSQL instalado.

Linguagem Procedural (PL/pgSQL)

- A PL/pgSQL é uma linguagem procedural que é carregada sob demanda no banco de dados.
- A arquitetura desta linguagem foi produzida de forma que:
 - Possa ser utilizada para escrever triggers e funções.
 - Adiciona estruturas de controle na linguagem SQL.
 - Pode executar cálculos complexos.
 - Herda os tipos de dados, funções e operações.
 - É fácil de usar.

Introdução

- Para poder ser utilizada, a PL deve ser “instalada” no Banco de Dados em que se deseja desenvolver procedimentos.
- SINTAXE:

```
CREATE LANGUAGE <nome_linguagem>  
HANDLER <função_handler>
```

- Exemplo:

```
CREATE LANGUAGE plpgsql  
HANDLER plpgsql_call_handler
```

Sintaxe:

CREATE OR REPLACE FUNCTION <NOME> ([*Lista_de_Parâmetros*])

RETURNS AS \$\$

[DECLARE *declaração_de_variáveis*]

BEGIN

comandos

END;

\$\$LANGUAGE nome_LinguagemProcedural;

Em que

- **DECLARE** – onde são declaradas as variáveis que serão utilizadas neste bloco.
 - SINTAXE: nome_variável tipo_dado [:= valor_inicial];
- **BEGIN e END** – delimitam um bloco dentro da função. No mínimo, uma função deve ter um bloco declarado. É possível declarar um bloco dentro de outro bloco (sub-blocos).
- O conjunto de comandos em SQL são encadeados por ;

Lista de Parâmetros

```
[TIPO_PARÂMETRO] nome_parâmetro TIPO_DADO  
[, [TIPO_PARÂMETRO] nome_parâmetro TIPO_DADO ]
```

● TIPO_PARÂMETRO:

● Default: entrada (IN)

- IN : indica parâmetro de entrada
- OUT: indica parâmetro de saída
- INOUT: indica parâmetro de entrada e saída

● TIPO_DADO:

- tipo de dado válido (INTEGER, DATE, INTERVAL, VARCHAR, TEXT ...)

Exemplo

```
CREATE OR REPLACE FUNCTION fn_retorna() RETURNS integer
AS $$
DECLARE
    inteiro integer := 10;

BEGIN

    return inteiro;

END;
$$LANGUAGE 'plpgsql'
```

- Para executar esta função:

```
select fn_retorna()
```

Exemplo

```
CREATE OR REPLACE FUNCTION fn_subtrai(vlr_a decimal, vlr_b
decimal) returns decimal AS
$$
declare
    resultado decimal(10,2);
begin
    resultado := vlr_a - vlr_b;

    return resultado;

end;
$$' language 'plpgsql'
```

- Para executar esta função:

```
select fn_subtrai(20.52,10.19)    => 10,33
```

Expressões

- As expressões em uma função podem ser aritméticas, booleanas, relacionais ou um comando SQL.

- Aritmética

Resultado := (valor_a + valor_b)/10;

- Boolean

Resultado_b := condicao1 AND condicao2;

- Relacional

valor_a >= valor_b;

- SQL

Insert into cliente (nom_cliente) values ('Rodrigo');

Exemplo de sub-bloco:

```
CREATE FUNCTION fn_retorna2()
```

```
RETURNS integer AS $$
```

```
DECLARE
```

```
    qtd integer := 30;
```

```
BEGIN
```

```
    RAISE NOTICE 'qtd here is %', qtd; -- qtd here is 30
```

```
    qtd := 50;
```

```
--
```

```
-- Criação de um sub-bloco
```

```
--
```

```
    DECLARE
```

```
        qtd integer := 80;
```

```
    BEGIN
```

```
        RAISE NOTICE 'qtd é %', qtd; -- qtd aqui é 80
```

```
    END;
```

```
    RAISE NOTICE 'qtd é %', qtd; -- qtd aqui é 50
```

```
    RETURN qtd;
```

```
END;
```

```
$$ LANGUAGE plpgsql;
```

Estrutura de Controle de Fluxo

- Como em qualquer outra linguagem procedural, a PL/pgSQL possui estruturas de controle para manipular os dados.
- As estruturas de controle são:
 - RETURN.
 - IF / THEN / ELSE (encadeado, aninhado).
 - LOOP (simples, while, for)

Estrutura de Controle de Fluxo

- RETURN – retorna qualquer expressão que a função possa retornar. Uma função que seja declarada para retornar algum valor deve usar o return dentro de sua declaração, nem que seja nulo.

```
return valor_a + valor_b;
```

```
return null;
```

Estrutura de Controle

- IF / THEN / ELSE

```
IF boolean-expression THEN  
    statements  
END IF;
```

- O PL/pgSQL apresenta cinco formas de IFs:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSE IF
- IF ... THEN ... ELSIF ... THEN ... ELSE
- IF ... THEN ... ELSEIF ... THEN ... ELSE

Estrutura de Controle de Fluxo – IF-ELSE

● Exemplo:

```
IF number = 0
  THEN result := 'zero';

ELSIF number > 0
  THEN result := 'positive';

ELSIF number < 0
  THEN result := 'negative';

ELSE
  result := 'NULL';

END IF;
```

Estrutura de Controle de Fluxo - LOOP

- LOOP – é um laço que não apresenta nenhuma condição de parada na sua estrutura, mas conta com o EXIT para sair do laço.

LOOP

result := result + count;

count := count + 1;

IF count > 10

THEN EXIT; -- exit loop

END IF;

END LOOP;

Estrutura de Controle de Fluxo - WHILE

- WHILE – é um laço que apresenta uma condição de parada na sua estrutura. Ou seja, avalia uma condição e enquanto ela for verdadeira o laço continua.

```
WHILE count < 10 LOOP
```

```
    result := result + count;
```

```
    count := count + 1;
```

```
    RAISE NOTICE 'O valor do resultado é: %', result;
```

```
END LOOP;
```

Estrutura de Controle de Fluxo - FOR

- FOR – é um laço de controle de fluxo com a seguinte **sintaxe**:

```
FOR nome IN [ REVERSE ] expressão .. expressão LOOP
    instruções
END LOOP;
```

- Em que:
 - Nome – é uma variável do tipo inteira é existe somente dentro do laço.
 - Expressão – define os valores inicial e final do laço

```
FOR i IN [ REVERSE ] 1 .. 10 LOOP
    RAISE NOTICE 'i == %', i;
END LOOP;
```


Instruções

- A PL/pgSQL tem algumas instruções básicas:
 - ATRIBUIÇÃO
 - SELECT INTO
 - FOUND
 - PERFORM

Instruções - Atribuição

- Atribui a uma variável ou campo um valor, resultado de uma expressão, de uma variável ou de uma constante.

- **SINTAXE:**

<identificador> := <expressão>

- **EXEMPLO:**

resultado := ((valor_a + valor_b)*50)/7;

Instruções – SELECT INTO

- SELECT INTO – os resultados de um comando SELECT podem ser atribuídos a uma variável, mas somente os resultados de um único valor.

- SINTAXE:

```
SELECT INTO <destino> <expressões_de_seleção>  
FROM...;
```

- Versões mais recentes permitem:

```
<destino> = SELECT <expressões_de_seleção>  
FROM...;
```

Instruções – SELECT INTO (2)

- **Exemplo**: obter a quantidade de funcionarios

```
CREATE OR REPLACE FUNCTION fn_qtdFuncionario() RETURNS int AS $$  
DECLARE  
    qtd integer := 0;  
BEGIN  
    SELECT INTO qtd count(*) FROM funcionario;  
  
    return qtd;  
  
END;  
$$ LANGUAGE plpgsql;
```

- Utilizando a função:

```
select fn_qtdFuncionario();
```

Instruções básicas - FOUND

- FOUND – é uma variável utilizada para determinar se um comando foi executado.
- Recebe TRUE se o(s) comando (s) de:
 - SELECT retornar um registro.
 - PERFORM retornar um registro.
 - UPDATE, INSERT ou DELETE afetar pelo menos um registro.

Instruções básicas - PERFORM (1)

- **PERFORM** – as vezes precisamos executar uma consulta, mas não queremos os resultados retornados.

- **SINTAXE:**

PERFORM <comando>;

Instruções básicas - PERFORM (2)

● Exemplo:

```
CREATE OR REPLACE FUNCTION fn_perform(nome varchar) RETURNS bool AS $$  
BEGIN  
    PERFORM ( SELECT pnome  
                FROM funcionario WHERE pnome=nome);  
  
    IF NOT FOUND THEN  
        return false;  
    ELSE  
        return true;  
    END IF;  
  
END;  
$$LANGUAGE 'plpgsql';
```

● Utilizando a função:

```
select fn_perform('Fulano');
```

Exercícios

Considere o BD Empresa e crie as funções e classifique-as em “Função” ou “Stored Procedure” :

- 1) Para calcular a idade do funcionário.
- 2) A) Para calcular a quantidade de funcionários que cada depto possui e atribuir ao campo qtd_funcionarios da tabela departamento. B) Para calcular a quantidade de funcionários que cada depto possui e atribuir ao campo qtd_funcionarios da tabela departamento.
- 3) Para converter o salário do funcionário em dólares. Passe por parâmetro o valor da cotação do dólar.
- 4) Para listar os aniversariantes do mês de julho.
- 5) Para listar os aniversariantes do mês atual.

Operadores e Funções para String

- **LIKE** para comparar duas strings p1 e p2 (p1 LIKE p2)
- O **ILIKE** é case INsensitive e o **LIKE** case sensitive.
- Ex: `SELECT * FROM FUnctionários`
`WHERE pnome LIKE 'M%';`
- Caracteres especiais: `'%'` (percentual) e `'_'` (underscore):
 - % similar a * -- qualquer combinação de caracteres
 - _ similar a ? (de arquivos no DOS) -- qualquer caractere
- Ex: ... `LIKE '[4-6]_6%'`
 - Seleciona strings em que
 - o primeiro caractere é de 4 a 6,
 - o segundo qualquer dígito ,
 - o terceiro sendo 6 e
 - os demais quaisquer

Funções para String

- **Length - retorna a quantidade de caracteres da string**
 - Ex: `SELECT CHAR_LENGTH('UNIFOR');` - -Retorna 6 ou
`SELECT LENGTH('Database');` - - Retorna 8
- **LOWER - Converter para minúsculas**
 - Ex: `SELECT LOWER('UNIFOR');` -- unifor
- **UPPER - Converter para maiúsculas**
 - Ex: `SELECT UPPER('universidade');` -- UNIVERSIDADE
- **POSITION – retorna a posição de caractere**
 - Ex: `SELECT POSITION ('@' IN 'ribafs@gmail.com');` -- Retorna 7
OU `SELECT STRPOS('Ribamar' , 'mar');` - - Retorna 5
- **INITCAP - Iniciais Maiúsculas**
 - Ex: `INITCAP(text)` - `INITCAP ('olá mundo')` - - Olá Mundo

Funções para String

● Remover Espaços em Branco

● TRIM ([leading | trailing | both] [characters] from string)

Remove caracteres da direita e da esquerda.

- Ex: trim (both 'b' from 'babacatebbbb'); - - abacate

● RTRIM (string text, chars text) - Remove os caracteres chars da direita (default é espaço)

- Ex: rtrim('removarrrr', 'r') - - remova

● LTRIM - (string text, chars text) - Remove os caracteres chars da esquerda

- ltrim('abssssremova', 'abs') - - remova

● Substring

● SUBSTRING(string [FROM inteiro] [FOR inteiro])

● SELECT SUBSTRING ('Ribamar FS' FROM 9 FOR 10); - - Retorna FS

● SUBSTRING(string FROM padrão);

● SELECT SUBSTRING ('PostgreSQL' FROM '.....'); - - Retorna Postgre

● SELECT SUBSTRING ('PostgreSQL' FROM '...\$'); - -Retorna SQL

● Primeiros e últimos ...\$

● SUBSTR ('string', inicio, quantidade);

- Ex: SELECT SUBSTR ('Ribamar', 4, 3); - - Retorna mar

Funções para String

- **Substituir todos os caracteres semelhantes**
 - `SELECT TRANSLATE(string, velho, novo);`
 - `SELECT TRANSLATE('Brasil', 'il', 'ão');` - - Retorna Brasília
 - `SELECT TRANSLATE('Brasileiro', 'eiro', 'eira');`
- **Calcular MD5 de String**
 - `SELECT MD5('ribafs');` - - Retorna 53cd5b2af18063bea8ddc804b21341d1
- **Repetir uma string n vezes**
 - `SELECT REPEAT('SQL-', 3);` - - Retorna SQL-SQL-SQL-
- **Sobrescrever substring em string**
 - `SELECT REPLACE ('Postgresql', 'sql', 'SQL');` - - Retorna PostgreSQL
- **Dividir Cadeia de Caracteres com Delimitador**
 - `SELECT SPLIT_PART('PostgreSQL', 'gre', 2);` - -Retorna SQL
 - `SELECT SPLIT_PART('PostgreSQL', 'gre', 1);` - -Retorna Post

Funções para String

● Like e %

SELECT * FROM frientds WHERE lastname LIKE 'M%';

O ILIKE é case INsensitive e o LIKE case sensitive.

~~ equivale ao LIKE

~~* equivale ao ILIKE

!~~ equivale ao NOT LIKE

!~~* equivale equivale ao NOT ILIKE

% similar a *

_ similar a ? (de arquivos no DOS)

... LIKE '[4-6]_6%' -- Pegar o primeiro sendo de 4 a 6,

- o segundo qualquer dígito, o terceiro sendo 6 e os demais quaisquer

Funções para String

Correspondência com um Padrão

- O PostgreSQL disponibiliza três abordagens distintas para correspondência com padrão: o operador LIKE tradicional do SQL; o operador mais recente SIMILAR TO (adicionado ao SQL:1999); e as expressões regulares no estilo POSIX. Além disso, também está disponível a função de correspondência com padrão substring, que utiliza expressões regulares tanto no estilo SIMILAR TO quanto no estilo POSIX.

```
SELECT substring('XY1234Z', 'Y*([0-9]{1,3})'); - Resultado: 123
```

SIMILAR TO

O operador SIMILAR TO retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é muito semelhante ao LIKE, exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL.

```
'abc' SIMILAR TO 'abc' verdade
```

```
'abc' SIMILAR TO 'a' falso
```

```
'abc' SIMILAR TO '%(b|d)%' verdade
```

```
'abc' SIMILAR TO '(b|c)%' falso
```

```
SELECT 'abc' SIMILAR TO '%(b|d)%'; -- Procura b ou d em 'abc' e no caso retorna TRUE
```

Funções para String e Referências

● REGEXP

SELECT 'abc' ~ '.*ab.*';

~ distingue a de A

~* não distingue a de A

!~ distingue expressões distingue a de A

!~* distingue expressões não distingue a de A

'abc' ~ 'abc' -- TRUE

'abc' ~ '^a' -- TRUE

'abc' ~ '(b|j)' -- TRUE

'abc' ~ '^(b|c)' -- FALSE

● Referências:

pgdocptbr.sourceforge.net/pg80/functions-string.html

pt.wikibooks.org/wiki/PostgreSQL_Pr%C3%A1tico/Fun%C3%A7%C3%B5es_Internas/Strings

Funções para Data e horas

Operações com datas:

- timestamp '2001-09-28 01:00' + interval '23 hours' -> timestamp '2001-09-29 00:00'
- date '2001-09-28' + interval '1 hour' -> timestamp '2001-09-28 01:00'
- date '01/01/2006' – date '31/01/2006'
- time '01:00' + interval '3 hours'time -> '04:00'
- interval '2 hours' - time '05:00' -> time '03:00:00'

Função age (retorna Interval) - **Diferença entre datas**

- age(timestamp)interval (**Subtrai de hoje**)
- age(timestamp '1957-06-13') -> 43 years 8 mons 3 days
- age(timestamp, timestamp)interval Subtrai os argumentos
- age('2001-04-10', timestamp '1957-06-13') -> 43 years 9 mons 27 days

Função extract (retorna double) Extrai parte da data: **ano, mês, dia, hora, minuto, segundo.**

- select extract(year from age('2001-04-10', timestamp '1957-06-13'))
- select extract(month from age('2001-04-10', timestamp '1957-06-13'))
- select extract(day from age('2001-04-10', timestamp '1957-06-13'))

Funções para Data e horas

Data e Hora atuais (retornam data ou hora)

- `SELECT CURRENT_DATE;`
- `SELECT CURRENT_TIME;`
- `SELECT CURRENT_TIME(0);`
- `SELECT CURRENT_TIMESTAMP;`
- `SELECT CURRENT_TIMESTAMP(0);`

Somar dias e horas a uma data:

- `SELECT CAST('06/04/2006' AS DATE) + INTERVAL '27 DAYS' AS Data;`

Função now (retorna timestamp with zone)

- `now()` - Data e hora corrente (timestamp with zone);
- Não usar em campos somente **timestamp**.

Funções para Data e horas

Função date_part (retorna double)

- `SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');`
- Resultado: 16 (day é uma string, diferente de extract)

Obtendo o dia da data atual:

- `SELECT DATE_PART('DAY', CURRENT_TIMESTAMP) AS dia;`

Obtendo o mês da data atual:

- `SELECT DATE_PART('MONTH', CURRENT_TIMESTAMP) AS mes;`

Obtendo o ano da data atual:

- `SELECT DATE_PART('YEAR', CURRENT_TIMESTAMP) AS ano;`

Funções para Data e horas

Função date_trunc (retorna timestamp)

- `SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');`
- Retorna 2001-02-16 00:00:00

Convertendo (CAST)

- `select to_date('1983-07-18', 'YYYY-MM-DD')`
- `select to_date('19830718', 'YYYYMMDD')`

Função timeofday (retorna texto)

- `select timeofday() -> Fri Feb 24 10:07:32.000126 2006 BRT`

Funções para Data e horas

Interval

- `interval [(p)]`
- `to_char(interval '15h 2m 12s', 'HH24:MI:SS')`
- `date '2001-09-28' + interval '1 hour'`
- `interval '1 day' + interval '1 hour'`
- `interval '1 day' - interval '1 hour'`
- `900 * interval '1 second'`
- Interval trabalha com as unidades: second, minute, hour, day, week, month, year, decade, century, millenium ou abreviaturas ou plurais destas unidades.
- Se informado sem unidades '13 10:38:14' será devidamente interpretado '13 days 10 hours 38 minutes 14 seconds'.

CURRENT_DATE - INTERVAL '1' day;

- `TO_TIMESTAMP('2006-01-05 17:56:03', 'YYYY-MM-DD HH24:MI:SS')`

Tipos de Dados: Geométricos e de Rede

- **Tipos Geométricos:**
- CREATE TABLE geometricos(ponto POINT, segmento LSEG, retangulo BOX, poligono POLYGON, circulo CIRCLE);
- ponto (0,0),
- segmento de (0,0) até (0,1),
- retângulo (base inferior (0,0) até (1,0) e base superior (0,1) até (1,1)) e
- círculo com centro em (1,1) e raio 1.
- INSERT INTO geometricos VALUES ('(0,0)', '((0,0),(0,1))', '((0,0),(0,1))', '((0,0),(0,1),(1,1),(1,0))', '((1,1),1)');

Tipos de Dados para Rede:

- Para tratar especificamente de redes o PostgreSQL tem os tipos de dados cidr, inet e macaddr.
- cidr – para redes IPV4 e IPV6
- inet – para redes e hosts IPV4 e IPV6
- macaddr – endereços MAC de placas de rede
- Assim como tipos data, tipos de rede devem ser preferidos ao invés de usar tipos texto para guardar IPs, Máscaras ou endereços MAC.



Usando Registros de Tabelas

Registros

- Variáveis de tipo de dados composto representados pelos registros do BD também podem ser manipulado nas funções.
- As variáveis locais podem ser do tipo de um campo de uma tabela, de um registro de uma tabela, ou de um formato indefinido.
- Veremos:
 - TYPE
 - ROWTYPE
 - RECORD

TYPE

- *Copying Types*: permite que uma variável assuma o tipo de dado de uma coluna de uma tabela do Banco de Dados.

- **SINTAXE**:

```
<nome_variavel> <nome_tabela>.<nome_campo>%TYPE;
```

- **EXEMPLO**: Suponha que desejamos trabalhar com variáveis do mesmo tipo de dado do campo num_depto da tabela funcionário

```
Var_sal funcionario.salario%TYPE;
```

- Desta maneira, a variável var_sal já assume o tipo de dado do campo salario da tabela funcionário

ROWTYPE (1)

- Row Type: permite que uma variável seja declarada como sendo do tipo de dado composto em que é possível armazenar todo o conteúdo de um registro selecionado por SELECT.
- SINATXE para Declaração :

```
<nome_variável> <nome_tabela>%ROWTYPE;
```
- Exemplo:

```
varFunc funcionario%ROWTYPE;
```
- Foi declarado uma variável de nome “varFunc” que possui os mesmos campos que um registro da tabela funcionario.

ROWTYPE (2)

- Para manipular os dados de variável declarada ROWTYPE
- **SINATXE** :
`<nome_variável>. <nome_campo>`
- Cada campo pode ser acessado usando a notação de 'ponto'.
- **Exemplo**:
`SELECT INTO varFunc * FROM Funcionario limit 1;
varFunc.salario =salario *1.1;`
- O conteúdo retornado pelo SELECT deve ser compatível com o ROWTYPE.

ROWTYPE (3)

```
CREATE FUNCTION CapturaRegistro() RETURNS RECORDS AS $$  
DECLARE  
    --Declara uma variável do tipo da linha da tabela em questão  
    RegX Projeto%ROWTYPE;  
BEGIN  
    --Captura uma linha qualquer na tabela especificada  
    SELECT * INTO RegX FROM Projeto WHERE projnumero = 100;  
  
    --Exibe na tela um dos campos da linha retornada  
    RAISE NOTICE ' O campo nome é %', RegX.proj_nome;  
  
    --Retorna a linha completa  
    RETURN RegX ;  
END;  
$$ LANGUAGE plpgsql;
```

RECORD (1)

- Record Type: é semelhante ao Row Type, ou seja, é um tipo composto de variável que serve para armazenar o valor de um registro inteiro, a diferença é que Record Type não tem uma estrutura predefinida.

- **SINTAXE:**

```
<nome_variável> RECORD;
```

- **EXEMPLO:**

```
varProjeto RECORD;
```

RECORD (2)

- Exemplo: Criar uma função para retornar os dados de um projeto dado seu número de identificação.

```
CREATE OR REPLACE FUNCTION fn_record( nproj integer)
  RETURNS record AS $$
DECLARE
  reg_teste RECORD;
BEGIN
  SELECT * INTO reg_teste FROM projeto WHERE projnumero = nproj;
  return reg_teste;
END;
$$LANGUAGE 'plpgsql';
```

- Utilizando a função

```
select fn_record(100);
```

FOR RECORD (1)

- Utilizando um tipo diferente de laço FOR, é possível interagir através do resultado de uma consulta e manipular os dados.
- **SINTAXE:**
FOR *registro_ou_linha* **IN** *comando* **LOOP**
 <conjunto de *instruções*>
END LOOP;
- Cada linha de resultado do *comando* (que deve ser um SELECT) é atribuída, sucessivamente, à variável registro ou linha, e o corpo do laço é executado uma vez para cada linha.

FOR RECORD (2)

- Exemplo: Mostrar o nome de todos os funcionários do departamento número 10;

```
CREATE OR REPLACE FUNCTION fn_looprecord() RETURNS void AS $$  
DECLARE  
    i RECORD;  
BEGIN  
    FOR i IN SELECT * FROM funcionario WHERE num_depto =10 LOOP  
        RAISE NOTICE 'Numero do funcionario é: %', i.numf;  
        RAISE NOTICE 'Nome do funcionario é: %', i.nome;  
    END LOOP;  
  
    RETURN;  
END;  
$$LANGUAGE 'plpgsql'
```

Outros Recursos – Mensagens

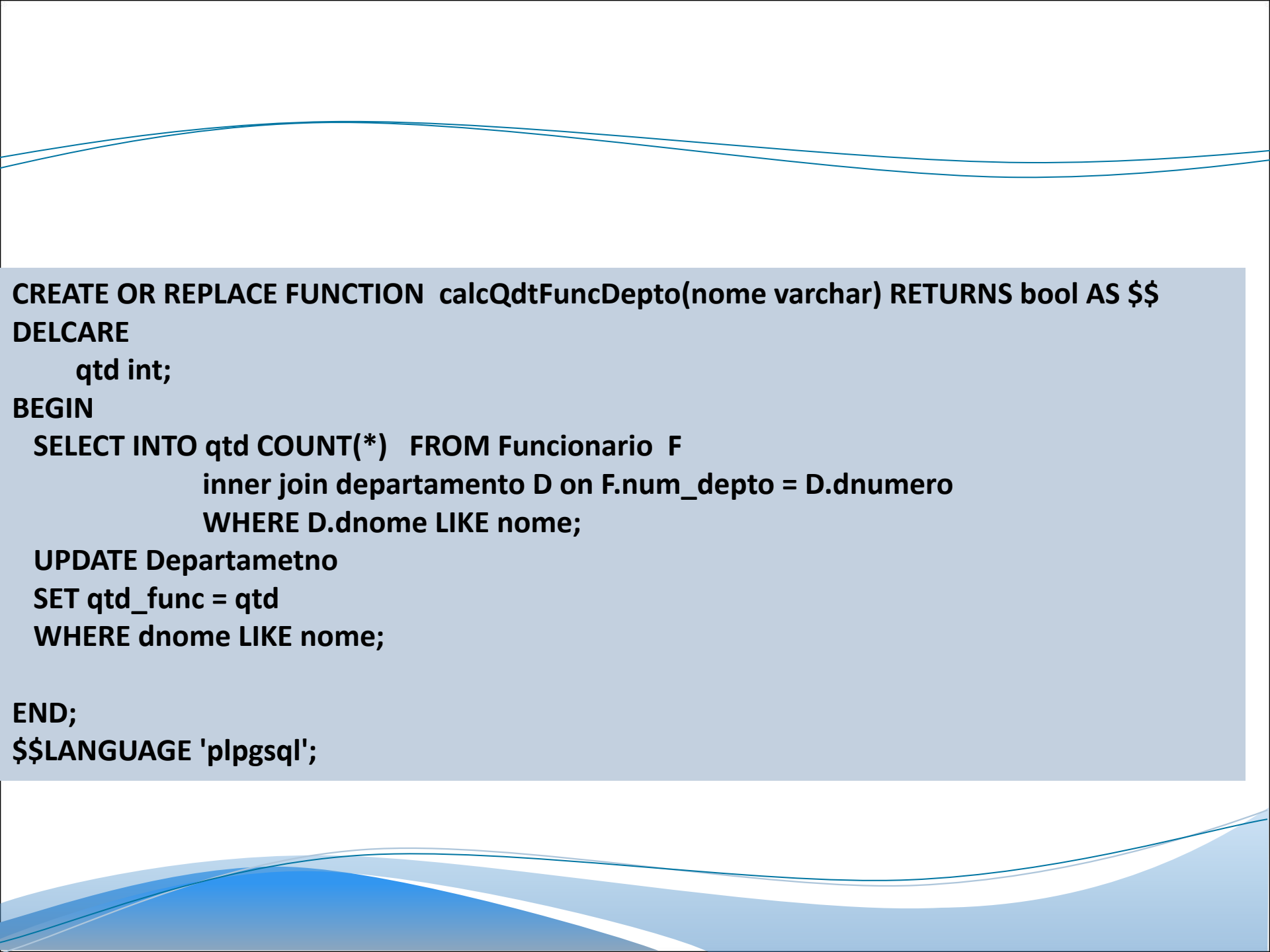
- Mensagens e erros – para relatar alguns erros ou mensagens para o usuário a PL/pgSQL utiliza o RAISE.

RAISE *level* 'mensagem %', variavel

- Onde o *level* pode ser:
 - DEBUG.
 - LOG.
 - INFO.
 - NOTICE.
 - WARNING.
 - EXCEPTION.

Exercícios - Continuação

- 6) Faça uma função para calcular quantos empregados subordinados tem um funcionário. O nome do funcionário supervisor é passado por parâmetro.
- 7) Faça uma função para calcular quantas horas semanais cada funcionário trabalha na empresa. Devem ser considerados todos os projetos que ele participa.
- 8) Crie um campo valor hora na tabela Funcionário e faça uma função para calcular e atualizar, qual o valor médio da hora trabalhada por cada funcionário.
- 9) Para cada departamento apresente: o número e nome do departamento, cada projeto que ele controla e cada funcionário que participa do projeto e respectivas horas de trabalho.



```
CREATE OR REPLACE FUNCTION calcQdtFuncDepto(nome varchar) RETURNS bool AS $$  
DECLARE  
    qtd int;  
BEGIN  
    SELECT INTO qtd COUNT(*) FROM Funcionario F  
        inner join departamento D on F.num_depto = D.dnumero  
        WHERE D.dnome LIKE nome;  
    UPDATE Departametno  
    SET qtd_func = qtd  
    WHERE dnome LIKE nome;  
END;  
$$LANGUAGE 'plpgsql';
```