

Stored Procedure - PL/PGSQL Functions

Banco de Dados

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Divisão de Computação
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul - IFRS
Câmpus Rio Grande

Introdução

Se há uma função que é usada por muitas aplicações, esta função poderá ser incorporada SGBD.

Com esta abordagem, cada aplicativo não precisará mais de uma cópia da função.

- Sempre que a função é necessária, o cliente pode simplesmente chamá-la.

Introdução

Além disso, as funções instaladas centralmente no servidor podem ser facilmente modificadas.

- Quando uma função é alterada, todos os aplicativos cliente imediatamente podem começar a utilizar a nova versão.

Exemplo de função pré-instalada no SGBD: *upper()*.

Introdução

O PostgreSQL oferece 4 tipos de funções:

- Funções escritas em SQL
- Funções em linguagens de procedimento:
 - PL/pgSQL, PL/Tcl, PL/php, PL/Java, etc
- Funções na linguagem C

Stored Procedure

Funções são criados com o comando **CREATE FUNCTION** e removidas com **DROP FUNCTION**.

CREATE FUNCTION requer a seguinte informação:

- Nome da função
- Número de argumentos da função
- Tipo de dado de cada argumento
- Tipo de retorno da função
- Ação da função
- Linguagem usada pela função

Stored Procedure

Exemplo 1: Convertendo temperatura de *Fahrenheit* para *Celcius*.
Tem-se as seguintes informações para criar a função:

- Nome da função é **ftoc**
- A função tem um argumento do tipo **float**
- A função retorna um **float**
- A ação da função é `SELECT ($1 - 32.0) * 5.0 / 9.0;`
- A linguagem é SQL

Stored Procedure

Exemplo 1: Convertendo temperatura de *Fahrenheit* para *Celcius*.

Criação:

```
CREATE FUNCTION ftoc(float) RETURNS float as  
'SELECT (($1 - 32.0)* 5.0 / 9.0);'  
LANGUAGE 'sql';
```

Chamada:

```
SELECT ftoc(68);  
ftoc  
---  
20  
(1 row)
```

Stored Procedure

- Embora a maioria das funções retornem apenas um valor, funções SQL podem retornar vários valores
- Ações de função também podem conter instruções **INSERT's**, **UPDATE's**, e **DELETE's**, bem como várias consultas separados por ponto e vírgula.

Comentário:

A ação da função *ftoc()* usa **SELECT** para executar um cálculo. Ela não acessa qualquer tabela.

Parâmetros

O \$1 no **SELECT** é automaticamente substituído pelo primeiro argumento da chamada de função. Se um segundo argumento está presente, ele poderia ser representado como \$2.

PL/PGSQL Functions

PL/PGSQL é uma outra linguagem destinada para as funções no SGBD.

- É uma linguagem de programação de verdade.

Enquanto as funções SQL permitem somente substituição de argumento, PL/PGSQL inclui recursos como **variáveis**, **avaliação condicional** e **looping**.

PL/PGSQL não está instalada em cada banco de dados por padrão. Para usá-la no banco de dados, você deve instalá-la.

```
createlang plpgsql -U nomeuser nomebanco
```

A PL/PGSQL é a linguagem de procedimentos armazenados mais utilizada no PostgreSQL, devido ser a mais madura e com mais recursos

PL/PGSQL Functions

Exemplo: Função que concatena textos

```
CREATE FUNCTION soma(text, text) RETURNS char AS
$$
DECLARE
resultado text;
BEGIN
resultado := $1 || $2;
return resultado;
END;
$$ LANGUAGE 'plpgsql';

-- Chamada
select soma('Sidney ', 'Silva');
```

Estrutura

- **DECLARE:** define as variáveis usadas na função.
- **RETURN:** Sai e retorna um valor da função.
- **BEGIN e END:** corpo principal da PL/PGSQL Function.

Atribuição e Declarações

Atribuição

```
variable := expression;
```

Apelidar um Parâmetro

```
name ALIAS FOR $n;
```

Atribuição e Declarações

```
user_id integer;  
quantity numeric(8,2);  
url character varying(60);  
myrow tablename%ROWTYPE;  
myfield tablename.columnname%TYPE;  
arow RECORD;
```

- **myrow**: variável que representa uma linha/tupla referente a tabela (**tablename**)
- **myfield**: variável que tem o mesmo tipo da coluna (**columnname**) da tabela (**tablename**)

Atribuição e Declarações

- Na declaração de variáveis, a expressão %TYPE fornece o tipo de dado de variável já declarada ou da coluna de uma relação
 - Geralmente, é utilizado para declarar variáveis que armazenam valores do banco de dados

Exemplo

```
id_depto      departamento.id_depto%TYPE;  
quantidade    numeric(3);  
quantidade2   quantidade%TYPE;
```

Representa o tipo de dado da variável "quantidade"

Representa o tipo de dado da coluna "id_depto" da relação "departamento"

Atribuição e Declarações

Vetores:

```
CREATE OR REPLACE FUNCTION test(in_array TEXT[]) RETURNS void as
$$
DECLARE
    t TEXT;
BEGIN
    FOREACH t IN ARRAY in_array LOOP
        raise notice 't: %', t;
    END loop;
END;
$$ language plpgsql;

-- Chamada
SELECT test( array['a', 'b', 'c'] );
```

```
locadora=# SELECT test( array['a', 'b', 'c'] );
NOTICE:  t: a
NOTICE:  t: b
NOTICE:  t: c
 test
-----
(1 row)
```

Condicionais

```
-- Validação de um número
IF numero = 0 THEN
    resultado := 'zero';
ELSIF numero > 0 THEN
    resultado := 'positivo';
ELSIF numero < 0 THEN
    resultado := 'negativo';
ELSE
    -- Outra possibilidade é o número nulo
    resultado := 'NULL';
END IF;
```

Figura: Condicionais

For

```
FOR i IN 1..(quantidade*2) LOOP
    -- algum processamento
    RAISE NOTICE 'i é %', i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- algum processamento
    RAISE NOTICE 'i é %', i;
END LOOP;
```

Figura: For

For

```
create function calculamaiorsalario(codigo integer) returns real as $$
declare
    r_funcionario record;
    salario real;
begin
    salario := 0;
    for r_funcionario in select * from funcionario loop
        raise notice 'funcionario: %,%,%', r_funcionario.codigo, r_funcionario.nome, r_funcionario.departamento, r_funcionario.salario;
        if (r_funcionario.departamento = codigo) and (r_funcionario.salario > salario) then
            salario := r_funcionario.salario;
        end if;
    end loop;
    return salario;
end;
$$ language 'plpgsql';
```

Figura: For

WHILE

```
WHILE quantidade > 0 AND quantidade < 1000 LOOP
    -- algum processamento
END LOOP;

WHILE NOT (quantidade <= 0 ) LOOP
    -- algum processamento
END LOOP;
```

Figura: WHILE

LOOP

```
LOOP
    -- algum processamento
    IF contador > 0 THEN
        EXIT;  -- finalização do laço
    END IF;
END LOOP;

LOOP
    -- algum processamento
    EXIT WHEN contador > 0; -- mesmo resultado do anterior
END LOOP;

BEGIN
    -- algum processamento
    IF quantidade > 100000 THEN
        EXIT; -- finalização do bloco
    END IF;
END;
```

Figura: Loop

Exemplo

```
CREATE OR REPLACE FUNCTION lacos(tipo_laco int4) RETURNS VOID AS
$body$
DECLARE
    contador int4 NOT NULL DEFAULT 0;
BEGIN
    IF tipo_laco = 1 THEN
        -- Loop usando WHILE
        WHILE contador < 10 LOOP
            contador := contador + 1;
            RAISE NOTICE 'Contador: %', contador;
        END LOOP;
    ELSIF tipo_laco = 2 THEN
        -- Loop usando LOOP
        LOOP
            contador := contador + 1;
            RAISE NOTICE 'Contador: %', contador;
            EXIT WHEN contador > 9;
        END LOOP;
    ELSE
        -- Loop usando FOR
        FOR contador IN 1..10 LOOP
            RAISE NOTICE 'Contador: %', contador;
        END LOOP;
    END IF;
    RETURN;
END;
$body$ LANGUAGE 'plpgsql';
```

Parâmetros

```
create function eh_par(integer) returns boolean as
$$
begin
    if ($1 % 2 = 0) then
        return true;
    else
        return false;
    end if;
end;
$$
language 'plpgsql'
```

Figura: Método 1 - Utilizar Parâmetros

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$
BEGIN
    RETURN i + 1;
END;
$$ LANGUAGE plpgsql;
```

Figura: Método 2 - Utilizar Parâmetros

Retornar mais de um Parâmetro

```
CREATE FUNCTION sum_n_product(x int, y int, OUT sum int, OUT prod int) AS $$  
BEGIN  
    sum := x + y;  
    prod := x * y;  
END;  
$$ LANGUAGE plpgsql;
```

Figura: Retornar mais de um Parâmetro: Criação OUT

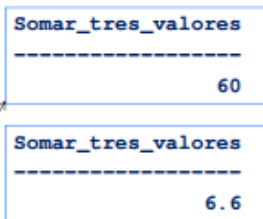
```
postgres=# select sum_n_product(1,2);  
          REATE  
sum_n_product  
-----  
      (3,2)  
(1 row)
```

Figura: Retornar mais de um Parâmetro: Utilização OUT

Parâmetros de Qualquer Tipo

```
CREATE FUNCTION somar_tres_valores(v1 anyelement, v2
anyelement, v3 anyelement) RETURNS anyelement AS $$
DECLARE
    resultado ALIAS FOR $0;
BEGIN
    resultado := v1 + v2 + v3;
    RETURN resultado;
END;
$$ LANGUAGE plpgsql;

-- Ativação da função somar_valores
SELECT somar_tres_valores(10,20,30);
SELECT somar_tres_valores(1.1,2.2,3.3);
```



Somar_tres_valores
60

Somar_tres_valores
6.6

A função somar três valores funciona qualquer tipo de dado que permita o operador +

RAISE

```
RAISE DEBUG 'The value of n is %', n;
```

This results in a log entry in the PostgreSQL log file (on Linux this is often `/usr/local/pgsql/data/postmaster.log`) that reads something like this:

```
DEBUG: The value of n is 4
```

```
create function scope() returns integer AS $$  
DECLARE  
    n integer := 4;  
BEGIN  
    RAISE DEBUG 'n is %', n;  
    return n;  
END;  
$$ language plpgsql;
```

Erros

```
CREATE FUNCTION altera_salario() RETURNS integer AS $$  
BEGIN  
    UPDATE funcionario SET salario = salario * 1.1;  
    -- sub-bloco  
    DECLARE  
        x integer;  
    BEGIN  
        UPDATE funcionario SET salario = 5000;  
        -- geração do erro propositalmente  
        x := 1/0;  
        RETURN 1;  
    EXCEPTION  
        WHEN division_by_zero THEN RAISE NOTICE 'Divisão por zero';  
        RETURN 0;  
    END;  
END;  
$$ LANGUAGE plpgsql;
```

O erro é capturado pela cláusula EXCEPTION do sub-bloco, desfazendo a 2ª alteração do salário. A 1ª alteração não é desfeita.

Erros

DATE_FIELD_OVERFLOW
DIVISION_BY_ZERO
ERROR_IN_ASSIGNMENT
ESCAPE_CHARACTER_CONFLICT
INDICATOR_OVERFLOW
INTERVAL_FIELD_OVERFLOW

O nome de condição especial OTHERS representa qualquer erro.

SELECT INTO

```
CREATE FUNCTION numero_de_empregados
    (depto departamento.nomedeppto%TYPE) RETURNS integer AS $$
DECLARE
    reg RECORD;  contador integer := 0;
BEGIN
    SELECT INTO reg * FROM departamento WHERE nomedeppto = depto;
    IF NOT FOUND THEN
        RAISE EXCEPTION 'Departamento % não encontrado', depto;
    ELSE
        for reg IN SELECT * FROM funcionario
            WHERE id_depto = reg.id_depto ORDER BY nomefunc LOOP
            contador := contador + 1;
            RAISE NOTICE '%) % : %', contador, reg.nomefunc, reg.endereco;
            PERFORM insere_log_2('leitura do funcionario ' || reg.id_func);
        END LOOP;
        RETURN contador;
    END IF;
END;
$$ LANGUAGE plpgsql;
```

A instrução PERFORM executa o comando especificado, desprezando algum resultado, caso haja.

ROWTYPE

```
CREATE FUNCTION encontrar_gereentes (tuplad departamento)
RETURNS text AS $$
DECLARE
    tuplaf funcionario%ROWTYPE;
BEGIN
    SELECT * INTO tuplaf FROM funcionario where
                                tuplad.id_gerente = id_func;
    RETURN tuplad.nomedepcto || '---' || tuplaf.nomefunc;
END;
$$ LANGUAGE plpgsql;

-- Ativação da função encontrar_gereentes
SELECT encontrar_gereentes(t.*) FROM departamento t;
```

Armazena em "tuplaf" a linha do funcionário que é gerente do departamento em questão ("tuplad")

Retorno da função:

Output pane	
Data Output Explain Messages History	
	encontrar_gereentes text
1	Pesquisa---Frank Santos
2	Administracao---Joao Mendes
3	Construcao---Julia Mendes

ROWTYPE

```
CREATE OR REPLACE FUNCTION media_compras(pid_cliente int4) RETURNS numeric AS
$body$
DECLARE
    linhaCliente      clientes%ROWTYPE;
    mediaCompras      numeric(9,2);
    totalCompras      numeric(9,2);
    periodo           int4;
BEGIN
    SELECT * INTO linhaCliente FROM clientes WHERE id_cliente = pid_cliente;
    -- Calcula o periodo em dias que trabalhamos com o cliente subtraindo da data atual a data
    de inclusão do cliente
    periodo := (current_date - linhaCliente.data_inclusao);
    -- Coloca na variável totalCompras o somatório de todos os pedidos do cliente
    SELECT SUM(valor_total) INTO totalCompras FROM pedidos WHERE id_cliente =
pid_cliente;
    -- Faz a divisão e retorna o resultado
    mediaCompras := totalCompras / periodo;
    RETURN mediaCompras;
END;
$body$ LANGUAGE 'plpgsql';
```

RECORD

As variáveis do tipo RECORD, são semelhantes às variáveis ROWTYPE, mas não possuem uma estrutura pré-definida.

- Uma variável RECORD assume a estrutura da linha que lhe é atribuída por meios dos comandos SELECT e FOR.
- Uma atribuição à variável RECORD já preenchida faz com que tal variável assuma a estrutura da nova linha que lhe é atribuída
- Antes de ser utilizada em uma atribuição, a variável RECORD não possui estrutura; assim, qualquer tentativa de acessar um de seus campos produz um erro em tempo de execução.

RECORD

```
CREATE FUNCTION encontrar_departamento(func funcionario.id_func%TYPE)
    RETURNS text AS $$
DECLARE
    reg RECORD; depto funcionario.id_depto%TYPE;
BEGIN
    SELECT INTO reg * FROM funcionario WHERE id_func = func;
    IF NOT FOUND THEN --após SELECT INTO, a variável especial FOUND
        --retorna falso se nenhum registro foi armazenado
        RAISE EXCEPTION 'Empregado % não encontrado', func;
    ELSE
        depto := reg.id_depto;
        SELECT INTO reg * FROM departamento WHERE id_depto = depto;
        RETURN reg.id_depto || '---' || reg.nomedepto;
    END IF;
END;
$$ LANGUAGE plpgsql;

-- Ativação da função encontra_departamento
SELECT encontra_departamento(1);
```

Armazena em "reg" o registro do funcionário de código "func"

Armazena em "reg" o registro do departamento de código "depto"

Exemplos

```
CREATE OR REPLACE FUNCTION increment(i integer) RETURNS integer AS $$  
    BEGIN  
        RETURN i + 1;  
    END;  
$$ LANGUAGE plpgsql;
```

```
CREATE FUNCTION less_than(a text, b text) RETURNS boolean AS $$  
DECLARE  
    local_a text := a;  
    local_b text := b;  
BEGIN  
    RETURN local_a < local_b;  
END;  
$$ LANGUAGE plpgsql;
```

Stored Procedure - PL/PGSQL Functions

Banco de Dados

Prof. Igor Avila Pereira
igor.pereira@riogrande.ifrs.edu.br

Divisão de Computação
Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Sul - IFRS
Câmpus Rio Grande