Funções em PostgreSQL

traduzido e adaptado do manual oficial do PostgreSQL

O SGBD PostgreSQL permite que sejam criadas funções pré-definidas pelo usuário sendo que as mesmas são armazenadas como objetos do SGBD e podem ser executadas diretamente no BD sem a necessidade da criação de uma aplicação externa. As funções criadas e armazenadas no banco são chamadas, historicamente, de *stored procedures* (procedimentos armazenados). Esse termo é herança do SGBD Oracle. Em PostgreSQL, como na linguagem C, não existem procedimentos, tudo é função. Um procedimento é um tipo especial de função que retorna vazio (*void*).

Um diferencial do PostgreSQL em relação aos outros SGBDs é a possibilidade de criar funções em sua linguagem padrão (plspgql – **p**rocedure **l**anguage **p**ost**g**res **sql**), pode-se criar funções em phyton, C, entre outros.

A estrutura do PL/pgSQL

PL/pgSQL é um linguagem estruturada por blocos. A definição de uma função tem que ser um bloco.
Um bloco é definido como:

```
[ <<rotulo>> ]
[ DECLARE
   declarações ]
BEGIN
   comandos
END [ rotulo ];
```

Cada declaração e cada comando dentro de um bloco termina com um ponto e vírgula. Um bloco que aparece dentro de outro deve ter um ponto e vírgula após o END, como apresentado acima. Contudo, o último END (o END mais externo) não necessita de ponto e vírgula.
Um rótulo é necessário apenas se queira-se identificar o bloco para uso do comando EXIT, ou qualificar os nomes de variáveis declaradas no bloco. Se um rótulo é dado após o comando END, esse rótulo deve coincidir com o rótulo dado no início do bloco.

Todas as palavras não são sensíveis ao caso. Identificadores são implicitamente convertidos para minúsculos, exceto se forem envolvidos por aspas.

Existem duas formas de comentar em PL/pgSQL. Um hífen duplo (--) inicia um comentário que ocupa apenas uma linha. Um /* inicia um bloco de comentário que vai até a ocorrência de */.

Qualquer comando na seção de comandos de um bloco pode ser um subbloco. Sublocos podem ser utilizados para agrupamentos lógicos ou para localizar variáveis dentro de um pequeno grupo de comandos. Variáveis declaradas em um subbloco são locais ao subbloco. Caso um variável local a um subbloco tenha o mesmo nome de uma variável mais externa, essa pode ser acessada qualificando a variável com o nome do bloco externo.
Exemplo:

```
CREATE FUNCTION funcaoqquer() RETURNS integer AS $$
<< blocoexterno >>
DECLARE
  qtdade integer := 30;
BEGIN
```

```
  RAISE NOTICE 'Quantidade aqui eh %', qtdade;  -- Imprime 30
  qtdade := 50;
  --
  -- Cria um subbloco
  --
  DECLARE
    qtdade integer := 80;
  BEGIN
    RAISE NOTICE ' Quantidade aqui eh %', qtdade;  -- Imprime 80
    RAISE NOTICE 'Quantidade externa eh  %', blocoexterno.qtdade;  -- Imprime 50
  END;
 RAISE NOTICE 'Quantidade aqui eh  %', qtdade;  -- Imprime 50
 RETURN qtdade;
END;
$$ LANGUAGE plpgsql;
```

Nota: existe um bloco mais externo que leva o nome da função criada.

Uma função pode ser testada chamando-a através do comando SELECT vazio.

```
Select funcaoqquer();
```

Declarações

Todas as variáveis utilizadas em um bloco devem ser declaradas na seção especificada para tal (DECLARE) – a única exceção são as variáveis de laço de um FOR de inteiros que automática declarada com uma variável do tipo inteiro, bem como laços cursores onde as variáveis são declaradas automaticamente como registros.
As variáveis PL/pgSQL podem ser de qualquer tipo dos dados SQL, tais como integer, varchar, e char.
Algumas declarações válidas:

```
user_id integer;
qtdade numeric(5);
url varchar;
minhatupla tablename%ROWTYPE;
meucamapo tablename.columnname%TYPE;
seta RECORD;
```

A sintaxe genérica para a declaração de variáveis é:

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

A cláusula DEFAULT, se utilizada, especifica o valor inicial associado à variável quando o bloco é acessado. Se DEFAULT não é especificado então a variável é inicializada com NULL. A opção CONSTANT protege a variável (que torna-se uma constante) de ser alterada durante sua existência no blobo. Se NOT NULL é especificado, se a variável for associada ao valor NULL, ocorre um erro em tempo de execução. Todas as variáveis declaradas como NOT NULL devem ter um valor não nulo como DEFAULT.

O valor default dado a uma variável é associado a mesma toda as vezes que bloco for executado (não apenas na primeira vez).

Exemplos:

```
qtdade integer DEFAULT 32;
url varchar := 'http://mysite.com';
user_id CONSTANT integer := 10;
```

Apelidos para os parâmetros

Parâmetros passados para as funções são nomeados com os identificadores $1, $2, etc. (conforme a posição dos mesmos). Opcionalmente, apelidos podem ser declarados para o nome do parâmetro $n para aumentar a redigibilidade. Ou o apelido ou o valor posicional pode ser utilizado para referenciar o valor do parâmetro.

Existem duas maneiras para se criar um apelido. O jeito preferido é dar um nome ao parâmetro no momento de criar a função, por exemplo:

```
CREATE FUNCTION taxa_venda(subtotal real) RETURNS real AS $$
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

A outra maneira, que era a única nas versões anteriores a 8.0, é explicitar a declaração de um apelido através da sintaxe:
apelido ALIAS FOR $n;

O mesmo exemplo utilizando a forma de apelido declarado:

```
CREATE FUNCTION taxa_venda(real) RETURNS real AS $$
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Esses dois exemplos não são estritamente equivalentes. No primeiro caso, subtotal poderia ser referenciado como taxa_venda.subtotal, já no segundo caso, não poderia pois o rótulo default (conforme apresentado acima) pode ser utilizado por variáveis especiais e pelos parâmetros.

Outro exemplo:

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- alguns cálculos utilizando v_string e index
END;
$$ LANGUAGE plpgsql;
```

Quando um parâmetro for declarado como de saída (utilizando OUT) funciona como uma variável que é inicializada com NULL e deve ser associada a um valor durante a execução de uma função. Esse tipo de parâmetro retorna automaticamente um valor para o chamador.

```
CREATE FUNCTION taxa_venda(subtotal real, OUT taxa real) AS $$
BEGIN
    taxa := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;
```

Perceba que o retornmo REAL foi omitido – poderia ser colocado mas seria redundante. Parâmetros do tipo OUT são úteis quando se deseja retornar múltiplos valores. Um exemplo simples:

```
CREATE FUNCTION soma_mult(x int, y int, OUT soma int, OUT mult int) AS $$
BEGIN
    soma := x + y;
    mult := x * y;
END;
$$ LANGUAGE plpgsql;
```

Quando RETURNS é omitido não pode existir o comando return no corpo da função.

Existem uma série de tipos que podem ser retornados, por exemplo, RETURNS TABLE retorna uma tabela (resultado de uma consulta):

```
CREATE FUNCTION vendas_ext (p_itemnr int) RETURNS TABLE(qtdade int, total
numeric) AS $$
BEGIN
  RETURN QUERY SELECT quantidade, quantidade * preco FROM vendas WHERE itemnr = p_itemnr;
END;
$$ LANGUAGE plpgsql;
```

Este tipo de retorno é equivalente ao uso de parâmetros do tipo OUT ou especificar RETURNS SETOF algum tipo.

Quando o tipo de retorno de uma função PL/pgSQL é declarado como polimórfico (anyelement, anyarray, anynonarray, ou anyenum), um parâmetro especial $0 é criado. O tipo de dado do mesmo é o retornado pela função, deduzido a partir dos tipos de entrada.

$0 é inicializado como null e pode ser modificado pelo função:

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

O mesmo efeito pode ser conseguido declarando um ou mais parâmetros do tipo OUT como polimórficos. Neste caso, o parâmetro especial $0 não precisa ser utilizado; os próprios parâmetros OUT servem para o propósito:

```
CREATE FUNCTION soma_3_valores(v1 anyelement, v2 anyelement, v3 anyelement,
                               OUT soma anyelement)
AS $$
BEGIN
    soma := v1 + v2 + v3;
END;
$$ LANGUAGE plpgsql;
```

Tipos de Cópia

A construção Nomevariável%TYPE oferece um tipo de dado copiado de uma outra variável ou coluna de tabela. Pode-se utilizar o recurso para declarar variáveis que armazenarão valores da base de dados. Imagine que uma tabela usuario tenha o atributo user_id e queremos criar uma variável do mesmo tipo de user_id, a construção seria:

user_id users.user_id%TYPE;

Utilizando %TYPE não é necessário conhecer o tipo de dado que se está referenciando e, mais importante, se o tipo de dado do item referenciado muda, a variável que referencia o tipo muda também sem a necessidade de alterar a função.

%TYPE is particularly valuable in polymorphic functions, since the data types needed for internal variables can change from one call to the next. Appropriate variables can be created by applying %TYPE to the function's arguments or result placeholders.

Tipos Linhas

nome nome_tabeka%ROWTYPE;

Este tipo de variável é chamado de variável de linha. Tal variável pode armazenar toda a linha de um SELECT. Para acessar os atributos, utilize a notação de ponto. Por exemplo, variavel.atributo.

Parâmetros em funções podem ser do tipo linha também. Neste caso, a posição correspondente do parâmetro $n pode ser utilizado com um variável de linha, por exemplo, $1.user_id.

Apenas as colunas definidas pelo usuário em um tabela podem ser acessadas com este tipo de variável. Pseudo-colunas (como por exemplo OID) não são acessíveis.

O exemplo abaixo utiliza esta definição. Imagine que as tabelas usuario e log existam e que os atributos citados existam.

```
CREATE FUNCTION merge_fields(t_row usuario) RETURNS text AS $$
DECLARE
    t2_row log%ROWTYPE;
BEGIN
    SELECT * INTO t2_row FROM log WHERE <condicao> ;
    RETURN t_row.id || t2_row.hora || t_row.nome || t2_row.ope;
END;
$$ LANGUAGE plpgsql;
```

Utilização:

```
SELECT merge_fields(t.*) FROM usuario t WHERE <condicao>;
```

Funções SQL

Funções SQL executam uma lista arbitrária de comandos SQL, retornando o resultado da última consulta na lista. No caso simples (não utilizando conjunto), a primeira linha da última consulta será o valor de retorno (essa linha não pode ser definida, exceto se utilizarmos o ORDER BY). Se a última consulta não retornar linha alguma, um valor null será retornado.

Também podemos declarar o retorno de uma função SQL como conjunto, colocando no retorno SETOF <algumTipo> ou RETURNS TABLE(colunas). Neste caso, todas as linhas da última

consulta são retornadas.

O corpo de uma função SQL é uma lista de comandos SQL separada por ponto e vírugla. O ponto e vírgula após o último comando é opcional. Exceto se a função tenha sido declarada com VOID, o último comando tem que ser um SELECT, ou um INSERT, UPDATE, ou DELETE que tenha uma cláusula RETURNING.

Exemplos:

```
CREATE FUNCTION clean_emp() RETURNS void AS '
    DELETE FROM emp
        WHERE salary < 0;
' LANGUAGE SQL;

SELECT clean_emp();

 clean_emp
-----------

(1 row)
```

Pode-se colocar o comando abaixo para atualizar uma tabela (minhatabela) com um valor de parâmetro (o primeiro):
`INSERT INTO mytable VALUES ($1);`
Mas o parâmetro não pode ser utilizado para substituir nome de atributo/coluna. Assim, o comando abaixo não funcionaria:
`INSERT INTO $1 VALUES (42);`

Funções com retorno:

```
CREATE FUNCTION um() RETURNS integer AS $$
    SELECT 1 AS result;
$$ LANGUAGE SQL;

-- Sintaxe alternativa: função como string
CREATE FUNCTION um() RETURNS integer AS '
    SELECT 1 AS result;
' LANGUAGE SQL;

SELECT um();

 um
-----
   1
```

Utilizando parâmetros sem nome:

```
CREATE FUNCTION adiciona(integer, integer) RETURNS integer AS $$
    SELECT $1 + $2;
$$ LANGUAGE SQL;

SELECT adiciona(1, 2) AS answer;

 answer
--------
      3
```

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS integer AS $$
    UPDATE bank
        SET balance = balance - $2
        WHERE accountno = $1;
    SELECT 1;
$$ LANGUAGE SQL;
```

Utilização:

```
SELECT tf1(17, 100.0);
```

A função melhorada:

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - $2
        WHERE accountno = $1;
    SELECT balance FROM bank WHERE accountno = $1;
$$ LANGUAGE SQL;
```

Utilizando RETURNING:

```
CREATE FUNCTION tf1 (integer, numeric) RETURNS numeric AS $$
    UPDATE bank
        SET balance = balance - $2
        WHERE accountno = $1
    RETURNING balance;
$$ LANGUAGE SQL;
```

Tipos Compostos

Utilizando tipos tabela e outros em funções SQL.

```
CREATE TABLE emp (
    name        text,
    salary      numeric,
    age         integer,
    cubicle     point
);

CREATE FUNCTION double_salary(emp) RETURNS numeric AS $$
    SELECT $1.salary * 2 AS salary;
$$ LANGUAGE SQL;

SELECT name, double_salary(emp.*) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';

 name | dream
------+-------
 Bill |  8400
```

Perceba que a sintaxe $1.salary seleciona um campo do parâmetro linha passado. Também perceba que o comando SELECT utiliza * para selecionar todos os campos da linha corrente da tabela que faz parte do parâmetro. Podemos referenciar a tabela da seguinte forma:

```
SELECT name, double_salary(emp) AS dream
    FROM emp
    WHERE emp.cubicle ~= point '(2,1)';
```

Mais exemplos:

```
CREATE FUNCTION getname(emp) RETURNS text AS $$
    SELECT $1.name;
$$ LANGUAGE SQL;

SELECT getname(new_emp());
 getname
---------
 None
(1 row)
```

Funções SQL com parâmetros do tipo OUT (saída)

```
CREATE FUNCTION add_em (IN x int, IN y int, OUT sum int)
AS 'SELECT $1 + $2'
LANGUAGE SQL;

SELECT add_em(3,7);
 add_em
--------
     10
(1 row)


CREATE FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int)
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;

 SELECT * FROM sum_n_product(11,42);
 sum | product
-----+---------
  53 |     462
(1 row)
```

Pode-se criar tipos do usuário:

```
CREATE TYPE sum_prod AS (sum int, product int);

CREATE FUNCTION sum_n_product (int, int) RETURNS sum_prod
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

Parâmetros da saída não são considerados como parte da assinatura da função, assim, para excluir a função sum_n_product do banco podemos executar:

```
DROP FUNCTION sum_n_product (x int, y int, OUT sum int, OUT product int);
```

ou

```
DROP FUNCTION sum_n_product (int, int);
```

Funções como Tabelas

```
CREATE TABLE foo (fooid int, foosubid int, fooname text);
INSERT INTO foo VALUES (1, 1, 'Joe');
INSERT INTO foo VALUES (1, 2, 'Ed');
INSERT INTO foo VALUES (2, 1, 'Mary');

CREATE FUNCTION getfoo(int) RETURNS foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT *, upper(fooname) FROM getfoo(1) AS t1;

 fooid | foosubid | fooname | upper
-------+----------+---------+-------
     1 |        1 | Joe     | JOE
(1 row)
```

Retornando Conjuntos

```
CREATE FUNCTION getfoo(int) RETURNS SETOF foo AS $$
    SELECT * FROM foo WHERE fooid = $1;
$$ LANGUAGE SQL;

SELECT * FROM getfoo(1) AS t1;

Then we would get:

 fooid | foosubid | fooname
-------+----------+---------
     1 |        1 | Joe
     1 |        2 | Ed
(2 rows)
```

Classificação das Funções

As funções podem ser `VOLATILE`, `STABLE`, ou `IMMUTABLE`.
`VOLATILE` é o tipo padrão, caso não for especificado

- Uma função `VOLATILE` pode fazer qualqquer coisa, incluindo modificar o banco de dados. Pode retornar resultados diferentes para chamdas sucessivas com os mesmos parâmetros. A cada chamada, o processador de funções reavalia a mesma.
- Uma função `STABLE` não pode modificar o banco de dados e garante retornar os mesmos resultados com os mesmos argumentos para todas as linhas em um comando simples. Esta categoria permite ao otimizador otimizar múltiplas chamadas como um só, não reavaliando a função a cada chamada.
- Uma função `IMMUTABLE` não pode alterar o banco de dados e garante retornar o mesmo resultado sempre. Esta categoria permite ao otimizador pré-avaliar a função quando uma consulta possui argumentos constantes. Por exemplo, `SELECT ... WHERE x = 2 + 2` pode ser simplificado para `SELECT ... WHERE x = 4`.

Essa classificação de funções permitem otimizar a chamada das mesmas. Dependendo do objetivo da função, pode-se colocar o tipo mais restrito, IMMUTABLE. Neste caso, a execução da função poderá ser otimizada de forma melhor.
Porém, a maioria das funções se enquadram como VOLATILE, pois a maioria das funções SQL precisam retornar valores diferentes a cada chamada.