14) Funções em SQL e em PL/pgSQL, Gatilhos e Regras

Funções em SQL no PostgreSQL

Funções em SQL no PostgreSQL executam rotinas com diversos comandos em SQL em seu interior e retornará o resultado da última consulta da lista.

Uma função em SQL também pode retornar um consulto, quando especificamos o retorno da função como sendo do tipo SETOF. Neste caso todos os registros da última consulta serão retornados.

Exemplos simples:

Criando:

```
CREATE FUNCTION dois() RETURNS integer AS '
   SELECT 2;
' LANGUAGE SQL;

Executando:
SELECT dois();

Excluindo empregados com salário negativo:
CREATE FUNCTION limpar_emp() RETURNS void AS '
    DELETE FROM empregados
        WHERE salario < 0;
' LANGUAGE SQL;

SELECT limpar_emp();
```

Função Passando Parâmetros:

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

SELECT adicao(1, 2) AS resposta;
```

Observe que psraâmetros são usados como: $1, $2, etc.

Outro exemplo (debitando valor em uma conta):

```
CREATE FUNCTION debitar (integer, numeric) RETURNS integer AS $$
    UPDATE banco
        SET saldo = saldo - $2
        WHERE conta = $1;
    SELECT 1;
$$ LANGUAGE SQL;
```

Exemplo com retorno mais interessante:

```
CREATE FUNCTION debitar2 (integer, numeric) RETURNS numeric AS $$
    UPDATE banco
        SET saldo = saldo - $2
        WHERE conta = $1;
    SELECT saldo FROM banco WHERE conta = $1;
$$ LANGUAGE SQL;
```

Exemplo com tipos compostos

```
CREATE TABLE empregados (
    nome        text,
    salario      numeric,
    idade         integer,
    cubiculo     point
);

insert into empregados values('Ribamar FS', 3856.45, 51, '(2,1)');

CREATE FUNCTION dobrar_salario(empregado) RETURNS numeric AS $$
    SELECT $1.salario * 2 AS salario;
$$ LANGUAGE SQL;

SELECT nome, dobrar_salario(empregados.*) AS sonho
    FROM empregados
    WHERE empregados.cubiculo ~= point '(2,1)';
```

Retornando um tipo composto:

```
CREATE FUNCTION novo_empregado() RETURNS empregados AS $$
    SELECT text 'Brito Cunha' AS nome,
        1000.0 AS salario,
        25 AS idade,
        point '(2,2)' AS cubiculo;
$$ LANGUAGE SQL;
```

Definindo de outra maneira:

```
CREATE FUNCTION novo_empregado() RETURNS empregados AS $$
    SELECT ROW('Brito Cunha', 1000.0, 25, '(2,2)')::empregados;
$$ LANGUAGE SQL;
```

select novo_empregado();
select (novo_empregado()).nome

```
CREATE FUNCTION recebe_nome(empregados) RETURNS text AS $$
    SELECT $1.nome;
$$ LANGUAGE SQL;
```

Usando uma outra função como parâmetro:

```
SELECT recebe_nome(novo_empregado());
```

Usando parâmetros de saída:

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

SELECT adicionar(3,7);


CREATE FUNCTION soma_n_produtos (x int, y int, OUT soma int, OUT produtos int)
AS 'SELECT $1 + $2, $1 * $2'
LANGUAGE SQL;
```

```
SELECT * FROM soma_n_produtos(11,42);
```

Criando Tipo Denifido pelo Usuário

```
CREATE TYPE soma_produtos AS (soma int, produto int);

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

select soma_n_produtos (4,5);

Tabela como fonte de Funções

```
CREATE TABLE tab (id int, subid int, nome text);
INSERT INTO tab VALUES (1, 1, 'Joe');
INSERT INTO tab VALUES (1, 2, 'Ed');
INSERT INTO tab VALUES (2, 1, 'Mary');

CREATE FUNCTION recebe_tab(int) RETURNS tab AS $$
    SELECT * FROM tab WHERE id = $1;
$$ LANGUAGE SQL;

SELECT *, upper(nome) FROM recebe_tab(1) AS tab1;
```

Observe o retorno:
1 1 Joe JOE

Primeiro retornam todos os campos da tab (*) e depois retorna o nome em maiúsculas.

Função Retornando Conjunto

```
CREATE FUNCTION recebe_tabela(int) RETURNS SETOF tabela1 AS $$
    SELECT * FROM tabela1 WHERE id = $1;
$$ LANGUAGE SQL;

SELECT * FROM recebe_tabela(1) AS tab1;
```

Retornando múltiplos registros:

```
CREATE FUNCTION soma_n_produtos_com_tab (x int, OUT soma int, OUT produtos int)
RETURNS SETOF record AS $$
    SELECT x + tab.y, x * tab.y FROM tab;
$$ LANGUAGE SQL;
```

Precisamos indicar o retorno com RETURNS SETOF record para que sejam retornados vários registros.

Exemplo que retorna conjunto através de select:

```
CREATE FUNCTION listchildren(text) RETURNS SETOF text AS $$
    SELECT name FROM nodes WHERE parent = $1
$$ LANGUAGE SQL;
```

```
SELECT * FROM nodes;
```

Funções Polimórficas

```
CREATE FUNCTION make_array(anyelement, anyelement) RETURNS anyarray AS $$
    SELECT ARRAY[$1, $2];
$$ LANGUAGE SQL;

SELECT make_array(1, 2) AS intarray, make_array('a'::text, 'b') AS textarray;

CREATE FUNCTION is_greater(anyelement, anyelement) RETURNS boolean AS $$
    SELECT $1 > $2;
$$ LANGUAGE SQL;

SELECT is_greater(1, 2);

CREATE FUNCTION dup (f1 anyelement, OUT f2 anyelement, OUT f3 anyarray)
AS 'select $1, array[$1,$1]' LANGUAGE sql;

SELECT * FROM dup(22);
```

Retornando sempre a última consulta (independente do parâmetro passado):

```
CREATE OR REPLACE FUNCTION somar(integer) RETURNS BIGINT AS $$
   SELECT sum(codigo) as x from cliente where codigo <$1;
   SELECT sum(codigo) as x from cliente where codigo < 2;
$$ LANGUAGE SQL;

SELECT somar(3) AS resposta;
```

Exemplo com CASE

```
create function categoria(int) returns char as
'
select
    case when idade<20 then \'a\'
        when idade >=20 and idade<30 then \'b\'
        when idade>=30 then \'c\'
    end as categoria
    from clientes where codigo = $1
'
language 'sql';

create function get_numdate (date) returns integer as
'
select (substr($1 , 6,2) || substr( $1 , 9,2))::integer;
'
 language 'SQL';

create function get_cliente (int) returns varchar as
```

```
'
select nome from clientes where codigo = $1;
'
language 'SQL';


create function get_cliente (int) returns varchar as
'
select nome from clientes where codigo = $1;
'
language 'SQL';

create function menores() returns setof clientes as
'
select * from clientes where idade < 18;
'
language 'SQL';

create function km2mi (float) returns float as
'
select $1 * 0.6;
'
language 'SQL';

create function get_signo (int) returns varchar as
'
select
    case
        when $1 <=0120 then \'capricornio\'
        when $1 >=0121 and $1 <=0219 then \'aquario\'
        when $1 >=0220 and $1 <=0320 then \'peixes\'
        when $1 >=0321 and $1 <=0420 then \'aries\'
        when $1 >=0421 and $1 <=0520 then \'touro\'
        when $1 >=0521 and $1 <=0620 then \'gemeos\'
        when $1 >=0621 and $1 <=0722 then \'cancer\'
        when $1 >=0723 and $1 <=0822 then \'leao\'
        when $1 >=0823 and $1 <=0922 then \'virgem\'
        when $1 >=0923 and $1 <=1022 then \'libra\'
        when $1 >=1023 and $1 <=1122 then \'escorpiao\'
        when $1 >=1123 and $1 <=1222 then \'sagitario\'
        when $1 >=1223 then \'capricornio\'
    end as signo
'
language 'SQL';
```

Consulta usando duas funções:

```
select ref_cliente, get_cliente(ref_cliente),
```

```
quantidade, preco, ref_produto, get_produto(ref_produto)
from compras;
```

Alguns Exemplos de uso das funções:
select get_numdate('14/04/1985');
get_numdate: 414

select get_signo(get_numdate('14/04/1985'));
get_signo: aries

select get_numdate('24/09/1980');
get_numdate: 924

select get_signo(get_numdate('24/09/1980'));
get_signo: libra

Mais Detalhes em:
http://pgdocptbr.sourceforge.net/pg80/xfunc-sql.html
http://www.postgresql.org/docs/current/static/xfunc-sql.html
http://www.linux-magazine.com.br/images/uploads/pdf_aberto/LM07_postgresql.pdf

Stored Procedures no PostgreSQL
(Usando funções em Pl/PgSQL)

No PostgreSQL todos os procedimentos armazenados (stored procedures) são funções, apenas elas usam linguagens de programação procedurais como PL/pgSQL, java, php, ruby, tcl, phyton, perl, etc.

A linguagem SQL é a que o PostgreSQL (e a maioria dos SGBDs relacionais) utiliza como linguagem de comandos. É portável e fácil de ser aprendida. Entretanto, todas as declarações SQL devem ser executadas individualmente pelo servidor de banco de dados.

Isto significa que o aplicativo cliente deve enviar o comando para o servidor de banco de dados, aguardar que seja processado, receber os resultados, realizar algum processamento, e enviar o próximo comando para o servidor. Tudo isto envolve comunicação entre processos e pode, também, envolver tráfego na rede se o cliente não estiver na mesma máquina onde se encontra o servidor de banco de dados.

Executar vários comandos de uma vez:

Usando a linguagem PL/pgSQL pode ser agrupado um bloco de processamento e uma série de comandos *dentro* do servidor de banco de dados, juntando o poder da linguagem procedural com a facilidade de uso da linguagem SQL, e economizando muito tempo, porque não há necessidade da sobrecarga de comunicação entre o cliente e o servidor. Isto pode aumentar o desempenho consideravelmente.

Vantagens no uso da linguagem procedural PL/pgSQL:
- Com ela podemos criar funções e triggers procedurais;
- Adicionar estruturas de controle para a linguagem SQL;
- Executar cálculos complexos;
- Herdar todos os tipos definidos pelo usuário, funções e operadores
- Pode ser definida para ser confiável para o servidor
- É fácil de usar

Nas funções em PL/pgSQL também podemos usar todos os tipos de dados, funções e operadores do SQL.

Argumentos Suportados e Tipos de Dados de Retorno

Funções escritas em PL/pgSQL podem aceitar como argumento qualquer tipo de dados escalar ou array suportado pelo servidor e podem retornar como resultado qualquer desses tipos.

Também podem aceitar ou retornar qualquer tipo composto (tipo row) especificado pelo nome.

Também podemos declarar uma função em PL/pgSQL retornando record, que permite que o resultado seja um registro cujos campos sejam determinados especificando-se na consulta.

Funções em PL/pgSQL também podem ser declaradas para aceitar e retornar os tipos polimórficos `anyelement`, `anyarray`, `anynonarray` e `anyenum`.

Também podem retornar um conjunto (uma tabela) de qualquer tipo de dados que pode retornar como uma instância simples. A função gera sua saída executando RETURN NEXT para cada elemento desejado do conjunto resultante ou usando RETURN QUERY para a saída resultante da avaliação da consulta.

Uma função em PL/pgSQL também, finalmente, pode pode ser declarada para retornar void, caso o retorno não tenha valor útil.

As funções em PL/pgSQL também podem ser declaradas com parâmetros de saída no lugar de uma especificação explícita do tipo de retorno. Isso não adiciona nenhuma capacidade fundamental para a linguagem, mas isso é conveniente, especialmente para retornar múltiplos valores.

A PL/pgSQL é uma linguagem estruturada em blocos.
O texto completo da definição de uma função precisa ser um bloco. Um bloco é definido como:

```
[ <<rótulo>> ]
[ DECLARE
    declaração de variáveis ];
BEGIN
    Instruções;
END [ rótulo ];
```

Delimitador de blocos

Cada DECLARE e cada BEGIN precisam terminar com ponto e vírgula.
O rótulo só é necessário se pretendemos usá-lo em um comando EXIT ou para qualificar os nomes das variáveis declaradas.

Todas as palavras-chaves são case-insensitivas.

A função **sempre** tem que retornar um valor.
Podemos retornar qualquer tipo de dados normal como boolean, text, varchar, integer, double, date, time, void, etc.

Comentários:

```
-- Comentário para uma linha
/* Comentário para
múltiplas
linhas
*/
```

Exemplo Simples:

```
CREATE or replace FUNCTION f_ola_mundo() RETURNS varchar AS $$
DECLARE
   ola varchar := 'Olá';
BEGIN
```

```
   PERFORM ola;
   RETURN ola;
END;
$$ LANGUAGE PLpgSQL;
```

Executando:
select f_ola_mundo()

Obs.: Uma única função pode ter até 16 parametros.

```
Create function numero(num1 text) returns integer as
$$
Declare
	resultado integer;
Begin
	resultado := num1;
	return resultado;
End;
$$ language 'plpgsql';

Create function texto(texto1 text, texto2 text) returns char as
$$
Declare
	resultado text;
Begin
	resultado := texto1 || texto2; --- || é o caracter para concatenação.
	return resultado;
End;
$$ language 'plpgsql';
```

Função que recebe uma string e retorna seu comprimento:

```
create function calc_comprim(text) returns int4 as
'
    declare
	textoentrada alias for $1;
	resultado int4;
    begin
	resultado := (select length(textoentrada));
	return resultado;
    end;
'
language 'plpgsql';
```

Chamando a função:
select calc_comprim('Ribamar');

Excluir função:

drop calc_comprim(text);

Exemplo mais elaborado

```
CREATE FUNCTION f_escopo() RETURNS integer AS $$
DECLARE
    quantidade integer := 30;
BEGIN
    RAISE NOTICE 'Quantidade aqui vale %', quantidade;  -- Imprime 30
    quantidade := 50;
    --
    -- Criar um sub-bloco
    --
    DECLARE
        quantidade integer := 80;
    BEGIN
        RAISE NOTICE 'Quantidade aqui vale %', quantidade;  -- Imprime 80
        RAISE NOTICE 'Quantidade aqui vale %', tabelaext.quantidade;  -- Imprime
50
    END;
    RAISE NOTICE 'Quantidade aqui vale %', quantidade;  -- Imprime 50
    RETURN quantidade;
END;
$$ LANGUAGE PLpgSQL;
```

Declaração de Variáveis

Todas as variáveis usadas em um bloco precisam ser declaras na seção DECLARE do bloco. A única exceção é para variáveis de iteração do loop for.

Variáveis em PL/pgSQL podem ter qualquer tipo de dados do SQL, como integer, varchar, char, etc.

Alguns exemplos de declaração de variáveis:

```
user_id integer;
quantity numeric(5);
url varchar;
myrow tablename%ROWTYPE;
myfield tablename.columnname%TYPE;
arow RECORD;
```

Sintaxe geral de declaração de variável:

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

Atribuição de Variáveis (com :=):

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

create function sec_func(int4,int4,int8,text,varchar) returns int4 as'
declare
myint constant integer := 5;

```
mystring char default "T";
firstint alias for $1;
secondint alias for $2;
third alias for $3;
fourth alias for $4;
fifth alias for $5;
ret_val int4;

begin

select into ret_val employee_id from masters where code_id = firstint and dept_id = secondint;
return ret_value;
end;
'language 'plpgsql';
```

A função acima precisa ser chamada assim:
select sec_func(3,4,cast(5 as int8),cast('trial text' as text),'some text');

Números passados como parâmetros tem valor default int4. Então precisamos fazer um cast para int8 ou bigint.

Alias para Parâmetros de Funções
Os parâmetros passados para as funções são nomeados como $1, $2, ... Para facilitar a leitura podemos criar alias para os parâmetros. Recomenda-se criar junto ao parâmetro na criação da função.

Na criação da função:

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

Na seção Declare:

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

No primeiro caso `subtotal` pode ser referenciada como `sales_tax.subtotal mas no segundo caso não pode.`

```
CREATE FUNCTION instr(varchar, integer) RETURNS integer AS $$
DECLARE
    v_string ALIAS FOR $1;
    index ALIAS FOR $2;
BEGIN
    -- some computations using v_string and index here
END;
```

```
$$ LANGUAGE plpgsql;


CREATE FUNCTION concat_selected_fields(in_t sometablename) RETURNS text AS $$
BEGIN
    RETURN in_t.f1 || in_t.f3 || in_t.f5 || in_t.f7;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION sales_tax(subtotal real, OUT tax real) AS $$
BEGIN
    tax := subtotal * 0.06;
END;
$$ LANGUAGE plpgsql;


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;
```

Quando um tipo de retorno de uma função é declarado como tipo polimórfico, um parâmetro especial $0 é criado. Este é o tipo de dados de retorno da 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;

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

Copiando Tipos
variavel%TYPE
%TYPE contém o tipo de dados de uma variável ou campo de tabela.

Para declarar uma variável chamada user_id com o mesmo tipo de dados de `users.user_id`:

```
user_id users.user_id%TYPE;
```

Usando %TYPE não precisamos conhecer o tipo de dados.

Tipos de Registros
nome nome_da_tabela%ROWTYPE;
```
nome nome_do_tipo_composto;
```

Uma variável do tipo composto é chamada de variável row (linha). Este tipo de variável pode armazenar toda uma linha de resultado de um comando SELECT ou FOR, desde que o conjunto de colunas do comando corresponda ao tipo declarado para a variável. Os campos individuais do valor linha são acessados utilizando a notação usual de ponto como, por exemplo, variável_linha.campo. Uma variável-linha pode ser declarada como tendo o mesmo tipo de dado das linhas de uma tabela ou de uma visão existente, utilizando a notação nome_da_tabela%ROWTYPE; ou pode ser declarada especificando o nome de um tipo composto (Uma vez que todas as tabelas possuem um tipo composto associado, que possui o mesmo nome da tabela, na verdade não faz diferença para o PostgreSQL se %ROWTYPE é escrito ou não, mas a forma contendo %ROWTYPE é mais portável).

Os parâmetros das funções podem ser de tipo composto (linhas completas da tabela). Neste caso, o identificador correspondente $n será uma variável linha, e os campos poderão ser selecionados a partir deste identificador como, por exemplo, $1.id_usuario.

Somente podem ser acessadas na variável tipo-linha as colunas definidas pelo usuário presentes na linha da tabela, a coluna OID e as outras colunas do sistema não podem ser acessadas por esta variável (porque a linha pode ser de uma visão). Os campos do tipo-linha herdam o tamanho do campo da tabela, ou a precisão no caso de tipos de dado como char(n).

create function third_func(text) returns varchar as'
declare
fir_text alias for $1;
sec_text mytable.last_name%type;
--here in the line above will assign the variable sec_text the datatype of
--of table mytable and column last_name.
begin

--some code here
end;
'language 'plpgsql';

```
%ROWTYPE representa a estrutura de uma tabela.
```

Abaixo está mostrado um exemplo de utilização de tipo composto:

```
CREATE FUNCTION mesclar_campos(t_linha nome_da_tabela) RETURNS text AS $$
DECLARE
    t2_linha nome_tabela2%ROWTYPE;
BEGIN
    SELECT * INTO t2_linha FROM nome_tabela2 WHERE ... ;
    RETURN t_linha.f1 || t2_linha.f3 || t_linha.f5 || t2_linha.f7;
END;
$$ LANGUAGE plpgsql;

SELECT mesclar_campos(t.*) FROM nome_da_tabela t WHERE ... ;
```

```
CREATE FUNCTION populate() RETURNS integer AS $$
DECLARE
    -- declarações
BEGIN
    PERFORM minha_funcao();
END;
$$ LANGUAGE plpgsql;
```

create function third(int4) returns varchar as'

declare

myvar alias for $1;
mysecvar mytable%rowtype;
mythirdvar varchar;

begin

select into mysecvar * from mytable where code_id = myvar;
--now mysecvar is a recordset
mythirdvar := mysecvar.first_name|| ' '|| mysecvar.last_name;
--|| is the concatenation symbol
--first_name and last_name are columns in the table mytable
return mythirdvar;
end;
'language 'plpgsql';

create function mess() returns varchar as'
declare
myret :="done";
begin
raise notice "hello there";
raise debug "this is the debug message";
raise exception "this is the exception message";

return myret;
end;
'language 'plpgsql';

Chamar com:
select mess();

Função dentro de Função

Podemos chamar uma função de dentro de outro sem retornar um valor.

create function test(int4,int4,int4) returns int4 as'
declare
first alias for $1;

```
sec alias for $2;
third alias for $3;

perform another_funct(first,sec);
return (first + sec);

end;
'language 'plpgsq';
```

Se a função acima for executada fará referência ao OID da minha_funcao() no plano de execução gerado para a instrução PERFORM. Mais tarde, se a função minha_funcao() for removida e recriada, então populate() não vai mais conseguir encontrar minha_funcao(). Por isso é necessário recriar populate(), ou pelo menos começar uma nova sessão de banco de dados para que a função seja compilada novamente. Outra forma de evitar este problema é utilizar CREATE OR REPLACE FUNCTION ao atualizar a definição de minha_funcao (quando a função é "substituída" o OID não muda).

Tipos registro

```
nome RECORD;
```

As variáveis registro são semelhantes às variáveis tipo-linha, mas não possuem uma estrutura pré-definida. Assumem a estrutura da linha para a qual são atribuídas pelo comando SELECT ou FOR. A subestrutura da variável registro pode mudar toda vez que é usada em uma atribuição. Como conseqüência, antes de ser utilizada em uma atribuição a variável registro não possui subestrutura, e qualquer tentativa de acessar um de seus campos produz um erro em tempo de execução.

Deve ser observado que RECORD não é um tipo de dado real, mas somente um guardador de lugar. Deve-se ter em mente, também, que declarar uma função do PL/pgSQL como retornando o tipo record não é exatamente o mesmo conceito de variável registro, embora a função possa utilizar uma variável registro para armazenar seu resultado. Nos dois casos a verdadeira estrutura da linha é desconhecida quando a função é escrita, mas na função que retorna o tipo record a estrutura verdadeira é determinada quando o comando que faz a chamada é analisado, enquanto uma variável registro pode mudar a sua estrutura de linha em tempo de execução.

RENAME

```
RENAME nome_antigo TO novo_nome;
```

O nome de uma variável, registro ou linha pode ser mudado através da instrução RENAME. A utilidade principal é quando NEW ou OLD devem ser referenciados por outro nome dentro da função de gatilho. Consulte também ALIAS.

Exemplos:

```
RENAME id TO id_usuario;
RENAME esta_variável TO aquela_variável;
```

> **Nota:** RENAME parece estar com problemas desde o PostgreSQL 7.3. A correção possui baixa prioridade, porque o ALIAS cobre a maior parte dos usos práticos do RENAME.

Expressões

As duas funções abaixo são diferentes:

```
CREATE FUNCTION logfunc1(logtxt text) RETURNS timestamp AS $$
    BEGIN
        INSERT INTO logtable VALUES (logtxt, 'now');
        RETURN 'now';
    END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION logfunc2(logtxt text) RETURNS timestamp AS $$
    DECLARE
        curtime timestamp;
    BEGIN
        curtime := 'now';
        INSERT INTO logtable VALUES (logtxt, curtime);
        RETURN curtime;
    END;
$$ LANGUAGE plpgsql;
```

Instruções básicas

Esta seção e as seguintes descrevem todos os tipos de instruções compreendidas explicitamente pela PL/pgSQL. Tudo que não é reconhecido como um destes tipos de instrução é assumido como sendo um comando SQL, e enviado para ser executado pela máquina de banco de dados principal (após a substituição das variáveis do PL/pgSQL na instrução). Desta maneira, por exemplo, os comandos SQL INSERT, UPDATE e DELETE podem ser considerados como sendo instruções da linguagem PL/pgSQL, mas não são listados aqui.

Atribuições

A atribuição de um valor a uma variável, ou a um campo de linha ou de registro, é escrita da seguinte maneira:

```
identificador := expressão;
```

Conforme explicado anteriormente, a expressão nesta instrução é avaliada através de um comando SELECT do SQL enviado para a máquina de banco de dados principal. A expressão deve produzir um único valor.

Se o tipo de dado do resultado da expressão não corresponder ao tipo de dado da variável, ou se a variável possuir um tipo/precisão específico (como char(20)), o valor do resultado será convertido implicitamente pelo interpretador do PL/pgSQL, utilizando a função de saída do tipo do resultado e a função de entrada do tipo da variável. Deve ser observado que este procedimento pode ocasionar erros em tempo de execução gerados pela função de entrada, se a forma cadeia de caracteres do valor do resultado não puder ser aceita pela função de entrada.

Exemplos:

```
id_usuario := 20;
taxa := subtotal * 0.06;
```

SELECT INTO

O resultado de um comando SELECT que retorna várias colunas (mas apenas uma linha) pode ser atribuído a uma variável registro, **a uma variável tipo-linha, ou a uma lista de variáveis escalares**. É feito através de

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

Exemplo:

SELECT INTO x SUM(quantidade) AS total FROM produtos;

Observe que a variável x retornará o valor da consulta SQL.
```

onde **destino** pode ser uma variável registro, uma variável linha, ou uma lista separada por vírgulas de variáveis simples e campos de registro/linha. A expressões_de_seleção e o restante do comando são os mesmos que no SQL comum.

Deve ser observado que é bem diferente da interpretação normal de SELECT INTO feita pelo PostgreSQL, onde o destino de INTO é uma nova tabela criada. Se for desejado criar uma tabela dentro de uma função PL/pgSQL a partir do resultado do SELECT, deve ser utilizada a sintaxe CREATE TABLE ... AS SELECT.

Se for utilizado como destino uma linha ou uma lista de variáveis, os valores selecionados devem corresponder exatamente à estrutura do destino, senão ocorre um erro em tempo de execução. Quando o destino é uma variável registro, esta se autoconfigura automaticamente para o tipo linha das colunas do resultado da consulta.

Exceto pela cláusula INTO, a instrução SELECT é idêntica ao comando SELECT normal do SQL, podendo utilizar todos os seus recursos.

A cláusula INTO pode aparecer em praticamente todos os lugares na instrução SELECT. Habitualmente é escrita logo após o SELECT, conforme mostrado acima, ou logo antes do FROM — ou seja, logo antes ou logo após a lista de expressões_de_seleção.

Se a consulta não retornar nenhuma linha, são atribuídos valores nulos aos destinos. Se a consulta retornar várias linhas, a primeira linha é atribuída aos destinos e as demais são desprezadas; deve ser observado que "a primeira linha" não é bem definida a não ser que seja utilizado ORDER BY.

A variável especial FOUND pode ser verificada imediatamente após a instrução SELECT INTO para determinar se a atribuição foi bem-sucedida, ou seja, foi retornada pelo menos uma linha pela consulta. (consulte a Seção 35.6.6). Por exemplo:

```
SELECT INTO meu_registro * FROM emp WHERE nome_emp = meu_nome;
IF NOT FOUND THEN
    RAISE EXCEPTION ''não foi encontrado o empregado %!'', meu_nome;
END IF;
```

Para testar se o resultado do registro/linha é nulo, pode ser utilizada a condição IS NULL. Entretanto, não existe maneira de saber se foram desprezadas linhas adicionais. A seguir está mostrado um exemplo que trata o caso onde não foi retornada nenhuma linha:

```
DECLARE
    registro_usuario RECORD;
BEGIN
    SELECT INTO registro_usuario * FROM usuarios WHERE id_usuario=3;
```

```
    IF registro_usuario.pagina_web IS NULL THEN
        -- o usuario não informou a página na web, retornar "http://"
        RETURN ''http://'';
    END IF;
END;
```

Exemplo Prático

Quero pegar o retorno de duas consulta, somar e receber este resultado no retorno.

```
CREATE OR REPLACE FUNCTION somar(quant integer) RETURNS BIGINT AS $$
declare
    x bigint;
    y bigint;
begin
   SELECT INTO x sum(codigo) from cliente where codigo <= quant;
IF NOT FOUND THEN
   -- A frase abaixo precisa ser delimitada por duas aspas simples,
   -- caso o delimitador da função seja aspas simples
   RAISE EXCEPTION 'Não foi encontrado o código %!', codigo;
END IF;
   SELECT INTO y sum(codigo) from cliente where codigo < quant;
IF NOT FOUND THEN
   RAISE EXCEPTION 'Não foi encontrado o código %!', codigo;
END IF;
   --z := cast(x as integer) + cast(y as integer);
   RETURN x+y;
end;
$$ LANGUAGE PLPGSQL;

SELECT somar();
```

Execução de expressão ou de consulta sem resultado

Instrução PERFORM

Algumas vezes se deseja **avaliar uma expressão ou comando e desprezar o resultado** (normalmente quando está sendo chamada uma função que produz efeitos colaterais, mas não possui nenhum valor de resultado útil). **Para se fazer isto no PL/pgSQL é utilizada a instrução PERFORM**:

```
PERFORM comando;
```

Esta instrução executa o comando e despreza o resultado. **A instrução deve ser escrita da mesma maneira que se escreve um comando SELECT do SQL, mas com a palavra chave inicial SELECT substituída por PERFORM**. As variáveis da linguagem PL/pgSQL são substituídas no comando da maneira usual. Além disso, a variável especial FOUND é definida como

verdade se a instrução produzir pelo menos uma linha, ou falso se não produzir nenhuma linha.

> **Nota:** Poderia se esperar que SELECT sem a cláusula INTO produzisse o mesmo resultado, mas atualmente a única forma aceita para isto ser feito é através do PERFORM.

Exemplo:

```
PERFORM create_mv('cs_session_page_requests_mv', my_query);
```

Não fazer nada

Algumas vezes uma instrução guardadora de lugar que não faz nada é útil. Por exemplo, pode indicar que uma ramificação da cadeia if/then/else está deliberadamente vazia. Para esta finalidade deve ser utilizada a instrução NULL:

```
NULL;
```

Por exemplo, os dois fragmentos de código a seguir são equivalentes:

```
BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        NULL;  -- ignorar o erro
END;

BEGIN
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN  -- ignorar o erro
END;
```

Qual dos dois escolher é uma questão de gosto.

> **Nota:** Na linguagem PL/SQL do Oracle não é permitida instrução vazia e, portanto, a instrução NULL é *requerida* em situações como esta. Mas a linguagem PL/pgSQL permite que simplesmente não se escreva nada.

Execução de comandos dinâmicos

As vezes é necessário gerar comandos dinâmicos dentro da função PL/pgSQL, ou seja, comandos que envolvem tabelas diferentes ou tipos de dado diferentes cada vez que são executados. A tentativa normal do PL/pgSQL de colocar planos para os comandos no *cache* não funciona neste cenário. A instrução EXECUTE é fornecida para tratar este tipo de problema:

```
EXECUTE cadeia_de_caracteres_do_comando;
```

onde cadeia_de_caracteres_do_comando é uma expressão que produz uma cadeia de caracteres (do tipo text) contendo o comando a ser executado. **A cadeia de caracteres é enviada literalmente para a máquina SQL.**

Em particular, deve-se observar que não é feita a substituição das variáveis do PL/pgSQL na cadeia de caracteres do comando. Os valores das variáveis devem ser inseridos na cadeia de caracteres do

comando quando esta é construída.

Diferentemente de todos os outros comandos do PL/pgSQL, o comando executado pela instrução EXECUTE não é preparado e salvo apenas uma vez por todo o tempo de duração da sessão. Em vez disso, o comando é preparado cada vez que a instrução é executada. A cadeia de caracteres do comando pode ser criada dinamicamente dentro da função para realizar ações em tabelas e colunas diferentes.

Os resultados dos comandos SELECT são desprezados pelo EXECUTE e, atualmente, o SELECT INTO não é suportado pelo EXECUTE. Portanto não há maneira de extrair o resultado de um comando SELECT criado dinamicamente utilizando o comando EXECUTE puro. Entretanto, **há duas outras maneiras disto ser feito: uma é utilizando o laço FOR-IN-EXECUTE** descrito na Seção 35.7.4, **e a outra é utilizando um cursor com OPEN-FOR-EXECUTE**, conforme descrito na Seção 35.8.2.

Quando se trabalha com comandos dinâmicos, muitas vezes é necessário tratar o escape dos apóstrofos. O método recomendado para delimitar texto fixo no corpo da função é utilizar o cifrão (Caso exista código legado que não utiliza a delimitação por cifrão por favor consulte a visão geral na Seção 35.2.1, que pode ajudar a reduzir o esforço para converter este código em um esquema mais razoável).

Os valores dinâmicos a serem inseridos nos comandos construídos requerem um tratamento especial, uma vez que estes também podem conter apóstrofos ou aspas. Um exemplo (**assumindo que está sendo utilizada a delimitação por cifrão para a função como um todo e, portanto, os apóstrofos não precisam ser duplicados**) é:

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(nome_da_coluna)
        || ' = '
        || quote_literal(novo_valor)
        || ' WHERE key = '
        || quote_literal(valor_chave);
```

Este exemplo mostra o uso das funções quote_ident(text) e quote_literal(text). **Por motivo de segurança, as variáveis contendo identificadores de coluna e de tabela devem ser passadas para a função quote_ident**. As variáveis contendo valores que devem se tornar literais cadeia de caracteres no comando construído devem ser passadas para função quote_literal. Estas duas funções executam os passos apropriados para retornar o texto de entrada envolto por aspas ou apóstrofos, respectivamente, com todos os caracteres especiais presentes devidamente colocados em seqüências de escape.

Deve ser observado que **a delimitação por cifrão somente é útil para delimitar texto fixo**. Seria uma péssima idéia tentar codificar o exemplo acima na forma

```
EXECUTE 'UPDATE tbl SET '
        || quote_ident(nome_da_coluna)
        || ' = $$'
        || novo_valor
        || '$$ WHERE key = '
        || quote_literal(valor_chave);
```

porque não funcionaria se o conteúdo de novo_valor tivesse $$. A mesma objeção se aplica a qualquer outra delimitação por cifrão escolhida. Portanto, **para delimitar texto que não é previamente conhecido *deve* ser utilizada a função quote_literal**.

Pode ser visto no Exemplo 35-8, onde é construído e executado um comando CREATE FUNCTION para definir uma nova função, um caso muito maior de comando dinâmico e EXECUTE.

Obtenção do status do resultado

Existem diversas maneiras de determinar o efeito de um comando. O primeiro método é utilizar o comando **GET DIAGNOSTICS**, que possui a forma:

```
GET DIAGNOSTICS variável = item [ , ... ] ;
```

Este comando permite obter os indicadores de status do sistema. Cada item é uma palavra chave que identifica o valor de estado a ser atribuído a variável especificada (que deve ser do tipo de dado correto para poder receber o valor). Os itens de status disponíveis atualmente são ROW_COUNT, o número de linhas processadas pelo último comando SQL enviado para a máquina SQL, e RESULT_OID, o OID da última linha inserida pelo comando SQL mais recente. Deve ser observado que RESULT_OID só tem utilidade após um comando INSERT.

Exemplo:

```
GET DIAGNOSTICS variável_inteira = ROW_COUNT;
create or replace function diag() returns integer as '
declare
variavel_inteira integer;
begin
GET DIAGNOSTICS variavel_inteira = ROW_COUNT;
return variavel_inteira;
end;
' language plpgsql;
```

O segundo método para determinar os efeitos de um comando é verificar a variável especial FOUND, que é do tipo boolean. A variável FOUND é iniciada como falso dentro de cada chamada de função PL/pgSQL. É definida por cada um dos seguintes tipos de instrução:

- A instrução SELECT INTO define FOUND como verdade quando retorna uma linha, e como falso quando não retorna nenhuma linha.
- A instrução PERFORM define FOUND como verdade quando produz (e despreza) uma linha, e como falso quando não produz nenhuma linha.
- As instruções UPDATE, INSERT e DELETE definem FOUND como verdade quando pelo menos uma linha é afetada, e como falso quando nenhuma linha é afetada.
- A instrução FETCH define FOUND como verdade quando retorna uma linha, e como falso quando não retorna nenhuma linha.
- A instrução FOR define FOUND como verdade quando interage uma ou mais vezes, senão define como falso. Isto se aplica a todas três variantes da instrução FOR (laços FOR inteiros, laços FOR em conjuntos de registros, e laços FOR em conjuntos de registros dinâmicos). A variável FOUND é definida desta maneira ao sair do laço FOR: dentro da execução do laço a variável FOUND não é modificada pela instrução FOR, embora possa ser modificada pela execução de outras instruções dentro do corpo do laço.

FOUND é uma variável local dentro de cada função PL/pgSQL; qualquer mudança feita na mesma afeta somente a função corrente.

Estruturas de controle

As estruturas de controle provavelmente são a parte mais útil (e mais importante) da linguagem PL/pgSQL. Com as estruturas de controle do PL/pgSQL os dados do PostgreSQL podem ser manipulados de uma forma muita flexível e poderosa.

Retorno de uma função

Estão disponíveis dois comandos que permitem retornar dados de uma função: RETURN e RETURN NEXT.

RETURN

```
RETURN expressão;
```

O comando RETURN com uma expressão termina a função e retorna o valor da expressão para quem chama. Esta forma é utilizada pelas funções do PL/pgSQL que não retornam conjunto.

Qualquer expressão pode ser utilizada para retornar um tipo escalar. O resultado da expressão é automaticamente convertido no tipo de retorno da função conforme descrito nas atribuições. Para retornar um valor composto (linha), deve ser escrita uma variável registro ou linha como a expressão.

O valor retornado pela função não pode ser deixado indefinido. Se o controle atingir o final do bloco de nível mais alto da função sem atingir uma instrução RETURN, ocorrerá um erro em tempo de execução.

Se a função for declarada como retornando void, ainda assim deve ser especificada uma instrução RETURN; mas neste caso a expressão após o comando RETURN é opcional, sendo ignorada caso esteja presente.

RETURN NEXT

```
RETURN NEXT expressão;
```

Quando uma função PL/pgSQL é declarada como retornando SETOF algum_tipo, o procedimento a ser seguido é um pouco diferente. Neste caso, os itens individuais a serem retornados são especificados em comandos RETURN NEXT, e um comando RETURN final, sem nenhum argumento, é utilizado para indicar que a função chegou ao fim de sua execução. O comando RETURN NEXT pode ser utilizado tanto com tipos de dado escalares quanto compostos; no último caso toda uma "tabela" de resultados é retornada.

As funções que utilizam RETURN NEXT devem ser chamadas da seguinte maneira:

```
SELECT * FROM alguma_função();
```

Ou seja, a função deve ser utilizada como uma fonte de tabela na cláusula FROM.

Na verdade, o comando RETURN NEXT não faz o controle sair da função: simplesmente salva o valor da expressão. Em seguida, a execução continua na próxima instrução da função PL/pgSQL. O

conjunto de resultados é construído se executando comandos RETURN NEXT sucessivos. O RETURN final, que não deve possuir argumentos, faz o controle sair da função.

> **Nota:** A implementação atual de RETURN NEXT para o PL/pgSQL armazena todo o conjunto de resultados antes de retornar da função, conforme foi mostrado acima. Isto significa que, se a função PL/pgSQL produzir um conjunto de resultados muito grande, o desempenho será ruim: os dados serão escritos em disco para evitar exaurir a memória, mas a função não retornará antes que todo o conjunto de resultados tenha sido gerado. Uma versão futura do PL/pgSQL deverá permitir aos usuários definirem funções que retornam conjuntos que não tenham esta limitação. Atualmente, o ponto onde os dados começam a ser escritos em disco é controlado pela variável de configuração work_mem. Os administradores que possuem memória suficiente para armazenar conjuntos de resultados maiores, devem considerar o aumento deste parâmetro.

Condicionais

As instruções IF permitem executar os comandos com base em certas condições. A linguagem PL/pgSQL possui cinco formas de IF:

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

IF-THEN

```
IF expressão_booleana THEN
    instruções
END IF;
```

As instruções IF-THEN são a forma mais simples de IF. As instruções entre o THEN e o END IF são executadas se a condição for verdade. Senão, são saltadas.

Exemplo:

```
IF v_id_usuario <> 0 THEN
    UPDATE usuarios SET email = v_email WHERE id_usuario = v_id_usuario;
END IF;
```

Outro exemplo

```
CREATE FUNCTION calclonger(text,text) RETURNS int4 AS

'
    DECLARE
```

```
        in_one  ALIAS FOR $1;
        in_two  ALIAS FOR $2;
        len_one int4;
        len_two int4;
        result  int4;
    BEGIN
        len_one := (SELECT LENGTH(in_one));
        len_two := (SELECT LENGTH(in_two));

        IF     len_one > len_two THEN
            RETURN len_one;
        ELSE
            RETURN len_two;
        END IF;
    END;
' LANGUAGE 'plpgsql';

IF     len_one > 20 AND len_one < 40 THEN
    RETURN len_one;
ELSE
    RETURN len_two;
END IF;
```

IF-THEN-ELSE

```
IF expressão_booleana THEN
    instruções
ELSE
    instruções
END IF;
```

As instruções IF-THEN-ELSE ampliam o IF-THEN permitindo especificar um conjunto alternativo de instruções a serem executadas se a condição for avaliada como falsa.

Exemplos:

```
IF id_pais IS NULL OR id_pais = ''
THEN
    RETURN nome_completo;
ELSE
    RETURN hp_true_filename(id_pais) || '/' || nome_completo;
END IF;

IF v_contador > 0 THEN
    INSERT INTO contador_de_usuários (contador) VALUES (v_contador);
    RETURN 't';
ELSE
    RETURN 'f';
END IF;
```

IF-THEN-ELSE IF

As instruções IF podem ser aninhadas, como no seguinte exemplo:

```
IF linha_demo.sexo = 'm' THEN
    sexo_extenso := 'masculino';
ELSE
    IF linha_demo.sexo = 'f' THEN
        sexo_extenso := 'feminino';
    END IF;
END IF;
```

Na verdade, quando esta forma é utilizada uma instrução IF está sendo aninhada dentro da parte ELSE da instrução IF externa. Portanto, há necessidade de uma instrução END IF para cada IF aninhado, mais um para o IF-ELSE pai. Embora funcione, cresce de forma tediosa quando existem muitas alternativas a serem verificadas. Por isso existe a próxima forma.

IF-THEN-ELSIF-ELSE

```
IF expressão_booleana THEN
    instruções
[ ELSIF expressão_booleana THEN
    instruções
[ ELSIF expressão_booleana THEN
    instruções
    ...]]
[ ELSE
    instruções ]
END IF;
```

A instrução IF-THEN-ELSIF-ELSE fornece um método mais conveniente para verificar muitas alternativas em uma instrução. Formalmente equivale aos comandos IF-THEN-ELSE-IF-THEN aninhados, mas somente necessita de um END IF.

Abaixo segue um exemplo:

```
IF numero = 0 THEN
    resultado := 'zero';
ELSIF numero > 0 THEN
    resultado := 'positivo';
ELSIF numero < 0 THEN
    resultado := 'negativo';
ELSE
    -- hmm, a única outra possibilidade é que o número seja nulo
    resultado := 'NULL';
END IF;
```

IF-THEN-ELSEIF-ELSE

ELSEIF é um aliás para ELSIF.

Laços simples

Com as instruções LOOP, EXIT, WHILE e FOR pode-se fazer uma função PL/pgSQL repetir uma série de comandos.

LOOP

```
[<<rótulo>>]
LOOP
    instruções
```

```
END LOOP;
```

A instrução LOOP define um laço incondicional, repetido indefinidamente até ser terminado por uma instrução EXIT ou RETURN. Nos laços aninhados pode ser utilizado um rótulo opcional na instrução EXIT para especificar o nível de aninhamento que deve ser terminado.

EXIT

```
EXIT [ rótulo ] [ WHEN expressão ];
```

Se não for especificado nenhum rótulo, o laço mais interno será terminado, e a instrução após o END LOOP será executada a seguir. Se o rótulo for fornecido, deverá ser o rótulo do nível corrente, ou o rótulo de algum nível externo ao laço ou bloco aninhado. Nesse momento o laço ou bloco especificado será terminado, e o controle continuará na instrução após o END correspondente ao laço ou bloco.

Quando WHEN está presente, a saída do laço ocorre somente se a condição especificada for verdadeira, senão o controle passa para a instrução após o EXIT.

Pode ser utilizado EXIT para causar uma saída prematura de qualquer tipo de laço; não está limitado aos laços incondicionais.

Exemplos:

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

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

BEGIN
    -- algum processamento
    IF estoque > 100000 THEN
        EXIT;  -- causa a saída do bloco BEGIN
    END IF;
END;
```

WHILE

```
[<<rótulo>>]
WHILE expressão LOOP
    instruções
END LOOP;
```

A instrução WHILE repete uma seqüência de instruções enquanto a expressão de condição for avaliada como verdade. A condição é verificada logo antes de cada entrada no corpo do laço.

Por exemplo:

```
WHILE quantia_devida > 0 AND saldo_do_certificado_de_bonus > 0 LOOP
    -- algum processamento
END LOOP;

WHILE NOT expressão_booleana LOOP
    -- algum processamento
END LOOP;
```

```
CREATE FUNCTION countc (text, text) RETURNS int4 AS '
    DECLARE
        intext ALIAS FOR $1;
        inchar ALIAS FOR $2;
        len    int4;
        result  int4;
        i      int4;
        tmp    char;
    BEGIN
        len    := length(intext);
        i      := 1;
        result  := 0;
        WHILE   i<= len LOOP
            tmp := substr(intext, i, 1);
            IF     tmp = inchar THEN
                result := result + 1;
            END IF;
            i:= i+1;
        END LOOP;
        RETURN result;
    END;
' LANGUAGE 'plpgsql';
```

FOR (variação inteira)

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

Esta forma do FOR cria um laço que interage num intervalo de valores inteiros. A variável nome é definida automaticamente como sendo do tipo integer, e somente existe dentro do laço. As duas expressões que fornecem o limite inferior e superior do intervalo são avaliadas somente uma vez, ao entrar no laço. Normalmente o passo da interação é 1, mas quando REVERSE é especificado se torna -1.

Alguns exemplos de laços FOR inteiros:

```
FOR i IN 1..10 LOOP
    -- algum processamento
```

```
    RAISE NOTICE 'i é %', i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- algum processamento
END LOOP;
```

Se o limite inferior for maior do que o limite superior (ou menor, no caso do REVERSE), o corpo do laço não é executado nenhuma vez. Nenhum erro é gerado.

Laço através do resultado da consulta

Utilizando um tipo diferente de laço FOR, é possível interagir através do resultado de uma consulta e manipular os dados. A sintaxe é:

```
[<<rótulo>>]
FOR registro_ou_linha IN comando LOOP
    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. Abaixo segue um exemplo:

```
CREATE FUNCTION cs_refresh_mviews() RETURNS integer AS $$
DECLARE
    mviews RECORD;
BEGIN
    PERFORM cs_log('Atualização das visões materializadas...');

    FOR mviews IN SELECT * FROM cs_materialized_views ORDER BY sort_key LOOP

        -- Agora "mviews" possui um registro de cs_materialized_views

        PERFORM cs_log('Atualizando a visão materializada ' ||
quote_ident(mviews.mv_name) || ' ...');
        EXECUTE 'TRUNCATE TABLE ' || quote_ident(mviews.mv_name);
        EXECUTE 'INSERT INTO ' || quote_ident(mviews.mv_name) || ' ' ||
mviews.mv_query;
    END LOOP;

    PERFORM cs_log('Fim da atualização das visões materializadas.');
    RETURN 1;
END;
$$ LANGUAGE plpgsql;
```

Se o laço for terminado por uma instrução EXIT, o último valor de linha atribuído ainda é acessível após o laço.

A instrução FOR-IN-EXECUTE é outra forma de interagir sobre linhas:

```
[<<rótulo>>]
FOR registro_ou_linha IN EXECUTE texto_da_expressão LOOP
    instruções
END LOOP;
```

Esta forma é semelhante à anterior, exceto que o código fonte da instrução SELECT é especificado como uma expressão cadeia de caracteres, que é avaliada e replanejada a cada entrada no laço FOR. Isto permite ao programador escolher entre a velocidade da consulta pré-planejada e a flexibilidade da consulta dinâmica, da mesma maneira que na instrução EXECUTE pura.

> **Nota:** Atualmente o analisador da linguagem PL/pgSQL faz distinção entre os dois tipos de laços FOR (inteiro e resultado de consulta), verificando se aparece .. fora de parênteses entre IN e LOOP. Se não for encontrado .., então o laço é assumido como sendo um laço sobre linhas. Se .. for escrito de forma errada, pode causar uma reclamação informando que "a variável do laço, para laço sobre linhas, deve ser uma variável registro ou linha", em vez de um simples erro de sintaxe como poderia se esperar.

```
CREATE FUNCTION countc(text, text, int4, int4) RETURNS int4 AS '
    DECLARE
        intext         ALIAS FOR $1;
        inchar         ALIAS FOR $2;
        startspos      ALIAS FOR $3;
        eendpos        ALIAS FOR $4;
        tmp            text;
        i              int4;
        len            int4;
        result         int4;
    BEGIN
        result = 0;
        len := LENGTH(intext);
        FOR i IN startpos..endpos LOOP
            tmp := substr(intext, i, 1);
            IF    tmp = inchar THEN
                  result := result + 1;
            END IF;
        END LOOP;
        RETURN result;
    END;
' LANGUAGE 'plpgsql';
```

Escrever a função anterior sem o FOR, com LOOP/EXIT

```
CREATE FUNCTION countc(text, text, int4, int4) RETURNS int4 AS '
    DECLARE
        intext     ALIAS FOR $1;
        inchar     ALIAS FOR $2;
        startpos   ALIAS FOR $3;
        endpos     ALIAS FOR $4;
        i          int4;
        tmp        text;
        len        int4;
        result     int4;
```

```
    BEGIN
        result = 0;
        i := startpos;
        len := LENGTH(intext);
        LOOP
            IF     i <= endpos AND i <= len THEN
                tmp := substr(intext, i, 1);
                IF     tmp = inchar THEN
                    result := result + 1;
                END IF;
                i := i + 1;
            ELSE
                EXIT;
            END IF;
        END LOOP;
        RETURN result;
    END;
' LANGUAGE 'plpgsql';
```

Sobrecarga de Funções

```
CREATE TABLE employees(id serial, name varchar(50), room int4, salary int4);
INSERT INTO employees (name, room, salary) VALUES ('Paul', 1, 3000);
INSERT INTO employees (name, room, salary) VALUES ('Josef', 1, 2945);
INSERT INTO employees (name, room, salary) VALUES ('Linda', 2, 3276);
INSERT INTO employees (name, room, salary) VALUES ('Carla', 1, 1200);
INSERT INTO employees (name, room, salary) VALUES ('Hillary', 2, 4210);
INSERT INTO employees (name, room, salary) VALUES ('Alice', 3, 1982);
INSERT INTO employees (name, room, salary) VALUES ('Hugo', 4, 1982);

CREATE FUNCTION insertupdate(text, int4) RETURNS bool AS '
    DECLARE
        intext  ALIAS FOR $1;
        newsal  ALIAS FOR $2;
        checkit record;
    BEGIN
        SELECT INTO checkit * FROM employees
            WHERE name=intext;
        IF NOT FOUND THEN
            INSERT INTO employees(name, room, salary)
            VALUES(intext,"1",newsal);
            RETURN "t";
        ELSE
            UPDATE employees SET
                salary=newsal, room=checkit.room
            WHERE name=intext;
            RETURN "f";
        END IF;
```

```
        RETURN "t";
    END;
' LANGUAGE 'plpgsql';

SELECT insertupdate('Alf',700);

SELECT * FROM employees WHERE name='Alf';

SELECT insertupdate('Alf',1250);

SELECT * FROM employees WHERE name='Alf';
```

Trabalhando com SELECT e LOOP

```
CREATE FUNCTION countsel(text) RETURNS int4 AS '
    DECLARE
        inchar  ALIAS FOR $1;
        colval  record;
        tmp     text;
        result  int4;
    BEGIN
        result = 0;
        FOR colval IN SELECT name FROM employees LOOP
            tmp := substr(colval.name, 1, 1);
            IF      tmp = inchar THEN
                result := result + 1;
            END IF;
        END LOOP;
    RETURN result;
    END;
' LANGUAGE 'plpgsql';
```

Captura de erros

Por padrão, qualquer erro que ocorra em uma função PL/pgSQL interrompe a execução da função, e também da transação envoltória. É possível capturar e se recuperar de erros utilizando um bloco BEGIN com a cláusula EXCEPTION. A sintaxe é uma extensão da sintaxe normal do bloco BEGIN:

```
[ <<rótulo>> ]
[ DECLARE
    declarações ]
BEGIN
    instruções
EXCEPTION
    WHEN condição [ OR condição ... ] THEN
        instruções_do_tratador
    [ WHEN condição [ OR condição ... ] THEN
          instruções_do_tratador
```

```
      ... ]
END;
```

Caso não ocorra nenhum erro, esta forma do bloco simplesmente executa todas as instruções, e depois o controle passa para a instrução seguinte ao END. Mas se acontecer algum erro dentro de instruções, o processamento das instruções é abandonado e o controle passa para a lista de EXCEPTION. É feita a procura na lista da primeira condição correspondendo ao erro encontrado. Se for encontrada uma correspondência, as instruções_do_tratador correspondentes são executadas, e o controle passa para a instrução seguinte ao END. Se não for encontrada nenhuma correspondência, o erro se propaga para fora como se a cláusula EXCEPTION não existisse: o erro pode ser capturado por um bloco envoltório contendo EXCEPTION e, se não houver nenhum, o processamento da função é interrompido.

O nome da condição pode ser qualquer um dos mostrados no Apêndice A. Um nome de categoria corresponde a qualquer erro desta categoria. O nome de condição especial OTHERS corresponde a qualquer erro, exceto QUERY_CANCELED (É possível, mas geralmente não aconselhável, capturar QUERY_CANCELED por nome). Não há diferença entre letras maiúsculas e minúsculas nos nomes das condições.

Caso ocorra um novo erro dentro das instruções_do_tratador selecionadas, este não poderá ser capturado por esta cláusula EXCEPTION, mas é propagado para fora. Uma cláusula EXCEPTION envoltória pode capturá-lo.

Quando um erro é capturado pela cláusula EXCEPTION, as variáveis locais da função PL/pgSQL permanecem como estavam quando o erro ocorreu, mas todas as modificações no estado persistente do banco de dados dentro do bloco são desfeitas. Como exemplo, consideremos este fragmento de código:

```
INSERT INTO minha_tabela(nome, sobrenome) VALUES('Tom', 'Jones');
BEGIN
    UPDATE minha_tabela SET nome = 'Joe' WHERE sobrenome = 'Jones';
    x := x + 1;
    y := x / 0;
EXCEPTION
    WHEN division_by_zero THEN
        RAISE NOTICE 'capturado division_by_zero';
        RETURN x;
END;
```

Quando o controle chegar à atribuição de y, vai falhar com um erro de division_by_zero. Este erro será capturado pela cláusula EXCEPTION. O valor retornado na instrução RETURN será o valor de x incrementado, mas os efeitos do comando UPDATE foram desfeitos. Entretanto, o comando INSERT que precede o bloco não é desfeito e, portanto, o resultado final no banco de dados é Tom Jones e não Joe Jones.

> **Dica:** Custa significativamente mais entrar e sair de um bloco que contém a cláusula EXCEPTION que de um bloco que não contém esta cláusula. Portanto, a cláusula EXCEPTION só deve ser utilizada quando for necessária.

```
CREATE FUNCTION calcsum(int4, int4) RETURNS int4 AS '
    DECLARE
        lower   ALIAS FOR $1;
        higher  ALIAS FOR $2;
```

```
        lowres  int4;
        lowtmp  int4;
        highres int4;
        result  int4;
    BEGIN
        IF (lower < 1) OR (higher < 1) THEN
        RAISE EXCEPTION "both param. have to be > 0";
        ELSE
            IF     (lower <= higher) THEN
                lowtmp  := lower - 1;
                lowres  := (lowtmp+1)*lowtmp/2;
                highres := (higher+1)*higher/2;
                result  := highres-lowres;
            ELSE
                RAISE EXCEPTION "The first value (%) has to be higher than the second
value (%)", higher, lower;

            END IF;
        END IF;
        RETURN result;
    END;
' LANGUAGE 'plpgsql';
```

Cursores
Em vez de executar toda a consulta de uma vez, é possível definir um *cursor* encapsulando a consulta e, depois, ler umas poucas linhas do resultado da consulta de cada vez. Um dos motivos de se fazer desta maneira, é para evitar o uso excessivo de memória quando o resultado contém muitas linhas (Entretanto, normalmente não há necessidade dos usuários da linguagem PL/pgSQL se preocuparem com isto, uma vez que os laços FOR utilizam internamente um cursor para evitar problemas de memória, automaticamente). Uma utilização mais interessante é retornar a referência a um cursor criado pela função, permitindo a quem chamou ler as linhas. Esta forma proporciona uma maneira eficiente para a função retornar conjuntos grandes de linhas.

Declaração de variável cursor

Todos os acessos aos cursores na linguagem PL/pgSQL são feitos através de variáveis cursor, que sempre são do tipo de dado especial refcursor. Uma forma de criar uma variável cursor é simplesmente declará-la como sendo do tipo refcursor. Outra forma é utilizar a sintaxe de declaração de cursor, cuja forma geral é:

```
nome CURSOR [ ( argumentos ) ] FOR comando ;
```

(O FOR pode ser substituído por IS para ficar compatível com o Oracle). Os argumentos, quando especificados, são uma lista separada por vírgulas de pares nome tipo_de_dado. Esta lista define nomes a serem substituídos por valores de parâmetros na consulta. Os valores verdadeiros que substituirão estes nomes são especificados posteriormente, quando o cursor for aberto.

Alguns exemplos:

```
DECLARE
    curs1 refcursor;
    curs2 CURSOR FOR SELECT * FROM tenk1;
    curs3 CURSOR (chave integer) IS SELECT * FROM tenk1 WHERE unico1 = chave;
```

Todas estas três variáveis possuem o tipo de dado refcursor, mas a primeira pode ser utilizada em qualquer consulta, enquanto a segunda possui uma consulta totalmente especificada *ligada* à mesma, e a terceira possui uma consulta parametrizada ligada à mesma (O parâmetro chave será substituído por um valor inteiro quando o cursor for aberto). A variável curs1 é dita como *desligada* (unbound), uma vez quer não está ligada a uma determinada consulta.

Abertura de cursor

Antes do cursor poder ser utilizado para trazer linhas, este deve ser *aberto* (É a ação equivalente ao comando SQL DECLARE CURSOR). A linguagem PL/pgSQL possui três formas para a instrução OPEN, duas das quais utilizam variáveis cursor desligadas, enquanto a terceira utiliza uma variável cursor ligada.

OPEN FOR SELECT

```
OPEN cursor_desligado FOR SELECT ...;
```

A variável cursor é aberta e recebe a consulta especificada para executar. O cursor não pode estar aberto, e deve ter sido declarado como um cursor desligado, ou seja, simplesmente como uma variável do tipo refcursor. O comando SELECT é tratado da mesma maneira que nas outras instruções SELECT da linguagem PL/pgSQL: Os nomes das variáveis da linguagem PL/pgSQL são substituídos, e o plano de execução é colocado no *cache* para uma possível reutilização.

Exemplo:

```
OPEN curs1 FOR SELECT * FROM foo WHERE chave = minha_chave;
```

OPEN FOR EXECUTE

```
OPEN cursor_desligado FOR EXECUTE cadeia_de_caracteres_da_consulta;
```

A variável cursor é aberta e recebe a consulta especificada para executar. O cursor não pode estar aberto, e deve ter sido declarado como um cursor desligado, ou seja, simplesmente como uma variável do tipo refcursor. A consulta é especificada como uma expressão cadeia de caracteres da mesma maneira que no comando EXECUTE. Como habitual, esta forma provê flexibilidade e, portanto, a consulta pode variar entre execuções.

Exemplo:

```
OPEN curs1 FOR EXECUTE 'SELECT * FROM ' || quote_ident($1);
```

Abertura de cursor ligado

```
OPEN cursor_ligado [ ( valores_dos_argumentos ) ];
```

Esta forma do OPEN é utilizada para abrir uma variável cursor cuja consulta foi ligada à mesma ao ser declarada. O cursor não pode estar aberto. Deve estar presente uma lista de expressões com os valores reais dos argumentos se, e somente se, o cursor for declarado como recebendo argumentos. Estes valores são substituídos na consulta. O plano de comando do cursor ligado é sempre considerado como passível de ser colocado no *cache*; neste caso não há forma EXECUTE equivalente.

Exemplos:

```
OPEN curs2;
OPEN curs3(42);
```

Utilização de cursores

Uma vez que o cursor tenha sido aberto, este pode ser manipulado pelas instruções descritas a seguir.

Para começar, não há necessidade destas manipulações estarem na mesma função que abriu o cursor. Pode ser retornado pela função um valor refcursor, e deixar por conta de quem chamou operar o cursor (Internamente, o valor de refcursor é simplesmente uma cadeia de caracteres com o nome do tão falado portal que contém a consulta ativa para o cursor. Este nome pode ser passado, atribuído a outras variáveis refcursor, e por aí em diante, sem perturbar o portal).

Todos os portais são fechados implicitamente ao término da transação. Portanto, o valor de refcursor pode ser utilizado para fazer referência a um cursor aberto até o fim da transação.

FETCH

```
FETCH cursor INTO destino;
```

A instrução FETCH coloca a próxima linha do cursor no destino, que pode ser uma variável linha, uma variável registro, ou uma lista separada por vírgulas de variáveis simples, da mesma maneira que no SELECT INTO. Como no SELECT INTO, pode ser verificada a variável especial FOUND para ver se foi obtida uma linha, ou não.

Exemplos:

```
FETCH curs1 INTO variável_linha;
FETCH curs2 INTO foo, bar, baz;
```

CLOSE

```
CLOSE cursor;
```

A instrução CLOSE fecha o portal subjacente ao cursor aberto. Pode ser utilizada para liberar recursos antes do fim da transação, ou para liberar a variável cursor para que esta possa ser aberta novamente.

Exemplo:

```
CLOSE curs1;
```

Retornar cursor

As funções PL/pgSQL podem retornar cursores para quem fez a chamada. É útil para retornar várias linhas ou colunas, especialmente em conjuntos de resultados muito grandes. Para ser feito, a função abre o cursor e retorna o nome do cursor para quem chamou (ou simplesmente abre o cursor utilizando o nome do portal especificado por, ou de outra forma conhecido por, quem chamou). Quem chamou poderá então ler as linhas usando o cursor. O cursor pode ser fechado por quem chamou, ou será fechado automaticamente ao término da transação.

O nome do portal utilizado para o cursor pode ser especificado pelo programador ou gerado automaticamente. Para especificar o nome do portal deve-se, simplesmente, atribuir uma cadeia de caracteres à variável refcursor antes de abri-la. O valor cadeia de caracteres da variável refcursor será utilizado pelo OPEN como o nome do portal subjacente. Entretanto, quando a variável refcursor é nula, o OPEN gera automaticamente um nome que não conflita com nenhum portal existente, e atribui este nome à variável refcursor.

> **Nota:** Uma variável cursor ligada é inicializada com o valor cadeia de caracteres que representa o seu nome e, portanto, o nome do portal é o mesmo da variável cursor, a menos que o programador mude este nome fazendo uma atribuição antes de abrir o cursor. Porém, uma variável cursor desligada tem inicialmente o valor nulo por padrão e, portanto, recebe um nome único gerado automaticamente, a menos que este seja mudado.

O exemplo a seguir mostra uma maneira de fornecer o nome do cursor por quem chama:

```
CREATE TABLE teste (col text);
INSERT INTO teste VALUES ('123');

CREATE FUNCTION reffunc(refcursor) RETURNS refcursor AS '
BEGIN
    OPEN $1 FOR SELECT col FROM teste;
    RETURN $1;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc('funccursor');

  reffunc
------------
 funccursor
(1 linha)

FETCH ALL IN funccursor;

 col
-----
 123
(1 linha)

COMMIT;
```

O exemplo a seguir usa a geração automática de nome de cursor:

```
CREATE FUNCTION reffunc2() RETURNS refcursor AS '
DECLARE
```

```
    ref refcursor;
BEGIN
    OPEN ref FOR SELECT col FROM teste;
    RETURN ref;
END;
' LANGUAGE plpgsql;

BEGIN;
SELECT reffunc2();

      reffunc2
--------------------
 <unnamed portal 1>
(1 linha)

FETCH ALL IN "<unnamed cursor 1>";

 col
-----
 123
(1 linha)

COMMIT;
```

Os exemplos a seguir mostram uma maneira de retornar vários cursores de uma única função:

```
CREATE FUNCTION minha_funcao(refcursor, refcursor) RETURNS SETOF refcursor AS $$
BEGIN
    OPEN $1 FOR SELECT * FROM tabela_1;
    RETURN NEXT $1;
    OPEN $2 FOR SELECT * FROM tabela_2;
    RETURN NEXT $2;
    RETURN;
END;
$$ LANGUAGE plpgsql;

-- é necessário estar em uma transação para poder usar cursor
BEGIN;

SELECT * FROM minha_funcao('a', 'b');

FETCH ALL FROM a;
FETCH ALL FROM b;
COMMIT;
```

Erros e mensagens

A instrução RAISE é utilizada para gerar mensagens informativas e causar erros.

```
RAISE nível 'formato' [, variável [, ...]];
```

Os níveis possíveis são DEBUG, LOG, INFO, NOTICE, WARNING, e EXCEPTION. O nível EXCEPTION causa um erro (que normalmente interrompe a transação corrente); os outros níveis apenas geram mensagens com diferentes níveis de prioridade. Se as mensagens de uma determinada

prioridade são informadas ao cliente, escritas no *log* do servidor, ou as duas coisas, é controlado pelas variáveis de configuração log_min_messages e client_min_messages. Para obter informações adicionais deve ser consultada a Seção 16.4.

Dentro da cadeia de caracteres de formatação, o caractere % é substituído pela representação na forma de cadeia de caracteres do próximo argumento opcional. Deve ser escrito %% para produzir um % literal. Deve ser observado que atualmente os argumentos opcionais devem ser variáveis simples, e não expressões, e o formato deve ser um literal cadeia de caracteres simples.

Neste exemplo o valor de v_job_id substitui o caractere % na cadeia de caracteres:

```
RAISE NOTICE 'Chamando cs_create_job(%)', v_job_id;
```

Este exemplo interrompe a transação com a mensagem de erro fornecida:

```
RAISE EXCEPTION 'ID inexistente --> %', id_usuario;
```

Atualmente RAISE EXCEPTION sempre gera o mesmo código SQLSTATE, P0001, não importando a mensagem com a qual seja chamado. É possível capturar esta exceção com EXCEPTION ... WHEN RAISE_EXCEPTION THEN ..., mas não há como diferenciar um RAISE de outro.

```
CREATE FUNCTION checksal(text) RETURNS int4 AS '
      DECLARE
            inname  ALIAS FOR $1;
            sal     employees%ROWTYPE;
            myval  employees.salary%TYPE;

      BEGIN
            SELECT INTO myval salary
                  FROM employees WHERE name=inname;
            RETURN myval;
      END;
' LANGUAGE 'plpgsql';

SELECT checksal('Paul');

select 5/2;

select timestamp(5000000/2);
```

Tratamento dos apóstrofos
O código da função PL/pgSQL é especificado no comando CREATE FUNCTION como um literal cadeia de caracteres. Se o literal cadeia de caracteres for escrito da maneira usual, que é entre apóstrofos ('), então os apóstrofos dentro do corpo da função devem ser duplicados; da mesma maneira, as contrabarras dentro do corpo da função (\) devem ser duplicadas. Duplicar os apóstrofos é no mínimo entediante, e nos casos mais complicados pode tornar o código difícil de ser compreendido, porque pode-se chegar facilmente a uma situação onde são necessários seis ou mais apóstrofos adjacentes. Por isso, recomenda-se que o corpo da função seja escrito em um literal cadeia de caracteres delimitado por "cifrão" (consulte a Seção 4.1.2.2) em vez de delimitado por apóstrofos. Na abordagem delimitada por cifrão os apóstrofos nunca são duplicados e, em vez disso, toma-se o cuidado de escolher uma marca diferente para cada nível de aninhamento necessário. Por exemplo, o comando CREATE FUNCTION pode ser escrito da seguinte maneira:

```
CREATE OR REPLACE FUNCTION funcao_teste(integer) RETURNS integer AS $PROC$
        ....
$PROC$ LANGUAGE plpgsql;
```

No corpo da função podem ser utilizados apóstrofos para delimitar cadeias de caracteres simples nos comandos SQL, e $$ para delimitar fragmentos de comandos SQL montados como cadeia de caracteres. Se for necessário delimitar um texto contendo $$, deve ser utilizado Q, e assim por diante.

O quadro abaixo mostra o que deve ser feito para escrever o corpo da função entre apóstrofos (sem uso da delimitação por cifrão). Pode ser útil para tornar códigos anteriores à delimitação por cifrão mais fácil de serem compreendidos.

1 apóstrofo

para começar e terminar o corpo da função como, por exemplo:

```
CREATE FUNCTION foo() RETURNS integer AS '
          ....
' LANGUAGE plpgsql;
```

Em todas as ocorrências dentro do corpo da função os apóstrofos *devem* aparecer em pares.

2 apóstrofos

Para literais cadeia de caracteres dentro do corpo da função como, por exemplo:

```
a_output := ''Blah'';
SELECT * FROM users WHERE f_nome=''foobar'';
```

Na abordagem delimitada por cifrão seria escrito apenas

```
a_output := 'Blah';
SELECT * FROM users WHERE f_nome='foobar';
```

que é exatamente o código visto pelo analisador do PL/pgSQL nos dois casos.

4 apóstrofos

Quando é necessário colocar um apóstrofo em uma constante cadeia de caracteres dentro do corpo da função como, por exemplo:

```
a_output := a_output || '' AND nome LIKE ''''foobar'''' AND xyz''
```

O verdadeiro valor anexado a a_output seria: AND nome LIKE 'foobar' AND xyz.

Na abordagem delimitada por cifrão seria escrito

```
a_output := a_output || $$ AND nome LIKE 'foobar' AND xyz$$
```

tendo-se o cuidado de que todos os delimitadores por cifrão envolvendo este comando não sejam apenas $$.

6 apóstrofos

Quando o apóstrofo na cadeia de caracteres dentro do corpo da função está adjacente ao final da constante cadeia de caracteres como, por exemplo:

```
a_output := a_output || '' AND nome LIKE ''''foobar''''''
```

O valor anexado à a_output seria: AND nome LIKE 'foobar'.

Na abordagem delimitada por cifrão se tornaria

```
a_output := a_output || $$ AND nome LIKE 'foobar'$$
```

10 apóstrofos

Quando é necessário colocar dois apóstrofos em uma constante cadeia de caracteres (que necessita de 8 apóstrofos), e estes dois apóstrofos estão adjacentes ao final da constante cadeia de caracteres (mais 2 apóstrofos). Normalmente isto só é necessário quando são escritas funções que geram outras funções como no Exemplo 35-8. Por exemplo:

```
a_output := a_output || '' if v_'' ||
    referrer_keys.kind || '' like ''''''''''
    || referrer_keys.key_string || ''''''''''
    then return ''''''  || referrer_keys.referrer_type
    || ''''''; end if;'';
```

O valor de a_output seria então:

```
if v_... like ''...'' then return ''...''; end if;
```

Na abordagem delimitada por cifrão se tornaria

```
a_output := a_output || $$ if v_$$ || referrer_keys.kind || $$ like '$$
    || referrer_keys.key_string || $$'
    then return '$$  || referrer_keys.referrer_type
    || $$'; end if;$$;
```

onde se assume que só é necessário colocar um único apóstrofo em a_output, porque este será delimitado novamente antes de ser utilizado.

Uma outra abordagem é fazer o escape dos apóstrofos no corpo da função utilizando a contrabarra em vez de duplicá-los. Desta forma é escrito \'\' no lugar de ''. Alguns acham esta forma mais fácil, porém outros não concordam.

Sobrecarga de função.

O conceito de sobrecarga de função na PL/pgSQL é o mesmo encontrado nas linguagens de programação orientadas a objeto. Abaixo um exemplo bem simples que ilustra esse conceito.

Imagine uma função soma.

```
Create function soma(num1 integer, num2 integer) returns integer as
$$
Declare
	resultado integer := 0;
Begin
	resultado := num1 + num2;
	return resultado;
End;
$$ language ‘plpgsql’;
```

Essa é uma função soma que recebe dois números inteiros. Mas e se o usuário passar dois dados do tipo caracter ao invés de números, qual seria o comportamento da minha função? Ai é que entra o conceito de sobrecarga de função. O PostgreSQL me permite ter uma função com o mesmo nome, mas com uma assinatura diferente. A assinatura é formada pelo nome + os parâmetros recebidos pela função. Por isso é que para remover uma função nós temos que utilizar o comando drop function + a assinatura da função e não somente o nome da função.

Para resolvermos o problema acima basta criarmos uma função soma da seguinte forma:

```
Create function soma(num1 text, num2 text) returns char as
$$
Declare
	resultado text;
Begin
	resultado := num1 || num2; --- || é o caracter para concatenação.
	return resultado;
End;
$$ language 'plpgsql';
```

Dessa forma, teríamos duas funções “soma”, uma que recebe dois inteiros e devolve o resultado da soma dos dois e outra que recebe dois caracteres e retorna a concatenação desses caracteres. O usuário só precisa saber que para concatenar caracteres ou somar valores basta chamar uma função soma.

Baseado no tipo dos parâmetros passados, o PostgreSQL se encarrega de definir qual a função será utilizada.

Observações: *Nas funções acima eu poderia ter suprimido a variável resultado e retornado os valores diretamente chamando "return num1 + num2;". A intenção foi a de reforçar o conceito da declaração de variável.*

O exemplo é bem simples o que quero mostrar é o conceito.

Exemplo prático de uso de funções plpgsql para validar CPF e CNPJ:

```
-- ************************************************
-- Função: f_cnpjcpf
-- Objetivo:
--   Validar o número do documento especificado
--   (CNPJ ou CPF) ou não (livre)
-- Argumentos:
--   Pessoa [Jurídica(0),Física(1) ou
--   Livre(2)] (integer), Número com dígitos
--   verificadores e sem pontuação (bpchar)
-- Retorno:
--   -1: Tipo de Documento invalido.
--   -2: Caracter inválido no numero do documento.
--   -3: Numero do Documento invalido.
--   1: OK (smallint)
-- ************************************************
--
CREATE OR REPLACE FUNCTION f_cnpjcpf (integer,bpchar)
RETURNS integer
AS '
DECLARE

-- Argumentos
-- Tipo de verificacao : 0 (PJ), 1 (PF) e 2 (Livre)
  pTipo ALIAS FOR $1;
-- Numero do documento
  pNumero ALIAS FOR $2;

-- Variaveis
  i INT4; -- Contador
  iProd INT4; -- Somatório
  iMult INT4; -- Fator
  iDigito INT4; -- Digito verificador calculado
  sNumero VARCHAR(20); -- numero do docto completo

BEGIN

-- verifica Argumentos validos
  IF (pTipo < 0) OR (pTipo > 2) THEN
    RETURN -1;
  END IF;
```

```
-- se for Livre, nao eh necessario a verificacao
  IF pTipo = 2 THEN
    RETURN 1;
  END IF;

  sNumero := trim(pNumero);
  FOR i IN 1..char_length(sNumero) LOOP
    IF position(substring(sNumero, i, 1) in "1234567890") = 0 THEN
      RETURN -2;
    END IF;
  END LOOP;
  sNumero := "";

-- ******************************************************
-- Verifica a validade do CNPJ
-- ******************************************************

  IF (char_length(trim(pNumero)) = 14) AND (pTipo = 0) THEN

-- primeiro digito
    sNumero := substring(pNumero from 1 for 12);
    iMult := 2;
    iProd := 0;

    FOR i IN REVERSE 12..1 LOOP
      iProd := iProd + to_number(substring(sNumero from i for 1),"9") * iMult;
      IF iMult = 9 THEN
        iMult := 2;
      ELSE
        iMult := iMult + 1;
      END IF;
    END LOOP;

    iDigito := 11 - (iProd % 11);
    IF iDigito >= 10 THEN
      iDigito := 0;
    END IF;

    sNumero := substring(pNumero from 1 for 12) || trim(to_char(iDigito,"9")) || "0";

-- segundo digito
    iMult := 2;
    iProd := 0;

    FOR i IN REVERSE 13..1 LOOP
      iProd := iProd + to_number(substring(sNumero from i for 1),"9") * iMult;
      IF iMult = 9 THEN
        iMult := 2;
```

```
        ELSE
          iMult := iMult + 1;
        END IF;
      END LOOP;

      iDigito := 11 - (iProd % 11);
      IF iDigito >= 10 THEN
        iDigito := 0;
      END IF;

      sNumero := substring(sNumero from 1 for 13) || trim(to_char(iDigito,"9"));
    END IF;

-- ********************************************************
-- Verifica a validade do CPF
-- ********************************************************

    IF (char_length(trim(pNumero)) = 11) AND (pTipo = 1) THEN

-- primeiro digito
      iDigito := 0;
      iProd := 0;
      sNumero := substring(pNumero from 1 for 9);

      FOR i IN 1..9 LOOP
        iProd := iProd + (to_number(substring(sNumero from i for 1),"9") * (11 - i));
      END LOOP;
      iDigito := 11 - (iProd % 11);
      IF (iDigito) >= 10 THEN
        iDigito := 0;
      END IF;
      sNumero := substring(pNumero from 1 for 9) || trim(to_char(iDigito,"9")) || "0";

-- segundo digito
      iProd := 0;
      FOR i IN 1..10 LOOP
        iProd := iProd + (to_number(substring(sNumero from i for 1),"9") * (12 - i));
      END LOOP;
      iDigito := 11 - (iProd % 11);
      IF (iDigito) >= 10 THEN
        iDigito := 0;
      END IF;
      sNumero := substring(sNumero from 1 for 10) || trim(to_char(iDigito,"9"));

    END IF;

-- faz a verificacao do digito verificador calculado
    IF pNumero = sNumero::bpchar THEN
```

```
    RETURN 1;
  ELSE
    RETURN -3;
  END IF;
END;
' LANGUAGE 'plpgsql';
```

Rode este script no seu banco, caso retorne um erro *"ERROR: language 'plpgsql' does not exist"* ou qualquer coisa parecida, é preciso dizer ao banco que esta base deve aceitar funções escritas em plpgsql. O PostgreSQL tem diversas linguagens PL, em uma das minhas colunas, menciona a maioria delas, dêem um olhada.

Bom, para habilitar a base a aceitar o plpgsql execute o comando no prompt bash ($):

```
createlang -U postgres pgplsql nomedabase
```

Em seguida, rode o script novamente.
Para executá-lo, digite no prompt da base (=#):

```
SELECT f_cnpjcpf( 1, '12312312345' );
```

Neste caso retorna um erro (-3) definido com documento inválido na função.

```
SELECT f_cnpjcpf( 2, '12312312345' );
```

Neste caso retorna (1) que significa que a operação foi bem sucedida! Porquê?!
Lembre-se, o argumento Pessoa tipo 2 não faz a validação do documento digitado.

Você também pode utilizar a função **f_cnpjcpf** na validação de um campo, por exemplo:

```
CREATE TABLE cadastro (
  nome      VARCHAR(50) NOT NULL,
  tipopessoa INT2 NOT NULL
          CHECK (tipopessoa IN (0,1)),
  cpfcnpj   CHAR(20) NOT NULL
          CHECK (f_cnpjcpf(tipopessoa, cpfcnpj)=1)
);
```

Ao tentar inserir um registro com um número de cpf ou cnpj inválido, volta um erro retornado pela *Check Constraint* responsável pela validação. Tente:

```
INSERT INTO cadastro (nome, tipopessoa, cpfcnpj)
VALUES ( 'Juliano S. Ignacio', 1, '12312312345');
```

Coloque o seu nome e cpf e verá que o registro será inserido.

Outra aplicação é na validação de cnpj (por exemplo) de uma tabela importada de uma origem qualquer:

```
SELECT * FROM nomedatabela
```

WHERE **f_cnpjcpf**(0, campocnpjimportado) < 1;

Dessa maneira, irá selecionar todos os registros onde o número do cnpj estiver errado.

Observe que as atribuições são com := e que o IF usa apenas = para seus testes.

Mais exemplos (do livro PostgreSQL a Comprehensive Guide)

```
drop table customers;
drop table tapes;
drop table rentals;

CREATE TABLE "customers" (
    "customer_id" integer unique not null,
    "customer_name" character varying(50) not null,
    "phone" character(8) null,
    "birth_date" date null,
    "balance" decimal(7,2)
);


CREATE TABLE "tapes" (
    "tape_id" character(8) not null,
    "title" character varying(80) not null,
    "duration" interval
);


CREATE TABLE "rentals" (
    "tape_id" character(8) not null,
    "rental_date" date not null,
    "customer_id" integer not null
);


INSERT INTO customers VALUES (3, 'Panky, Henry', '555-1221', '1968-01-21', 0.00);
INSERT INTO customers VALUES (1, 'Jones, Henry', '555-1212', '1970-10-10', 0.00);
INSERT INTO customers VALUES (4, 'Wonderland, Alice N.', '555-1122', '1969-03-05', 3.00);
INSERT INTO customers VALUES (2, 'Rubin, William', '555-2211', '1972-07-10', 15.00);

INSERT INTO tapes VALUES ('AB-12345', 'The Godfather');
INSERT INTO tapes VALUES ('AB-67472', 'The Godfather');
INSERT INTO tapes VALUES ('MC-68873', 'Casablanca');
INSERT INTO tapes VALUES ('OW-41221', 'Citizen Kane');
INSERT INTO tapes VALUES ('AH-54706', 'Rear Window');

INSERT INTO rentals VALUES ('AB-12345', '2001-11-25', 1);
```

```
INSERT INTO rentals VALUES ('AB-67472', '2001-11-25', 3);
INSERT INTO rentals VALUES ('OW-41221', '2001-11-25', 1);
INSERT INTO rentals VALUES ('MC-68873', '2001-11-20', 3);

\echo '****************************************'
\echo 'You may see a few error messages:'
\echo '  table customers does not exist'
\echo '  table tapes does not exist'
\echo '  table rentals does not exist'
\echo '  CREATE TABLE / UNIQUE will create implicit index customers_id_key for table
customers'
\echo ''
\echo 'This is normal and you can ignore those messages'
\echo '****************************************'

--  exchange_index.sql
--
CREATE OR REPLACE FUNCTION get_exchange( CHARACTER )
  RETURNS CHARACTER AS '

  DECLARE
    result      CHARACTER(3);
  BEGIN

     result := SUBSTR( $1, 1, 3 );

    return( result );
  END;
' LANGUAGE 'plpgsql' WITH ( ISCACHABLE );


-- File: dirtree.sql

CREATE OR REPLACE FUNCTION dirtree( TEXT ) RETURNS SETOF _fileinfo AS $$
DECLARE
  file  _fileinfo%rowtype;
  child _fileinfo%rowtype;
BEGIN

  FOR file IN SELECT * FROM fileinfo( $1 ) LOOP
    IF file.filename != '.' and file.filename != '..' THEN
      file.filename = $1 || '/' || file.filename;

      IF file.filetype = 'd' THEN
        FOR child in SELECT * FROM dirtree( file.filename ) LOOP
          RETURN NEXT child;
    END LOOP;
      END IF;
```

```
    RETURN NEXT file;
   END IF;
  END LOOP;

  RETURN;

END
$$ LANGUAGE 'PLPGSQL';

-------------------------------------------------------------------------------
-- my_factorial
-------------------------------------------------------------------------------

CREATE OR REPLACE FUNCTION my_factorial( value INTEGER ) RETURNS INTEGER AS
$$
  DECLARE
    arg INTEGER;
  BEGIN

    arg := value;

    IF arg IS NULL OR arg < 0 THEN
        RAISE NOTICE 'Invalid Number';
        RETURN NULL;
    ELSE
     IF arg = 1 THEN
         RETURN 1;
       ELSE
      DECLARE
          next_value INTEGER;
        BEGIN
       next_value := my_factorial(arg - 1) * arg;
          RETURN next_value;
        END;
      END IF;
    END IF;
  END;
$$ LANGUAGE 'plpgsql';

-------------------------------------------------------------------------------
-- my_factorial
-------------------------------------------------------------------------------

CREATE FUNCTION my_factorial( value INTEGER ) RETURNS INTEGER AS $$
  DECLARE
    arg INTEGER;
  BEGIN
```

```
   arg := value;

   IF arg IS NULL OR arg < 0 THEN
     BEGIN
       RAISE NOTICE 'Invalid Number';
       RETURN NULL;
     END;
   ELSE
     IF arg = 1 THEN
       BEGIN
         RETURN 1;
       END;
     ELSE
       DECLARE
         next_value INTEGER;
       BEGIN
         next_value := my_factorial(arg - 1) * arg;
         RETURN next_value;
       END;
     END IF;
   END IF;
  END;
$$ LANGUAGE 'plpgsql';

-------------------------------------------------------------------------------
-- compute_due_date
-------------------------------------------------------------------------------

CREATE FUNCTION compute_due_date( DATE ) RETURNS DATE AS $$
  DECLARE
    due_date     DATE;
    rental_period   INTERVAL := '7 days';

  BEGIN

    due_date := $1 + rental_period;

    RETURN( due_date );

  END;
$$ LANGUAGE 'plpgsql';

-------------------------------------------------------------------------------
-- compute_due_date
-------------------------------------------------------------------------------

CREATE FUNCTION compute_due_date( DATE, INTERVAL ) RETURNS DATE AS $$
  DECLARE
```

```
   rental_date   ALIAS FOR $1;
   rental_period ALIAS FOR $2;
  BEGIN

   RETURN( rental_date + rental_period );

  END;
$$ LANGUAGE 'plpgsql';

-------------------------------------------------------------------------------
-- getBalances
-------------------------------------------------------------------------------

CREATE OR REPLACE FUNCTION getBalances( id INTEGER ) RETURNS SETOF NUMERIC
AS $$
  DECLARE
   customer   customers%ROWTYPE;
  BEGIN

   SELECT * FROM customers INTO customer WHERE customer_id = id;

   FOR month IN 1..12 LOOP

     IF customer.monthly_balances[month] IS NOT NULL THEN
       RETURN NEXT customer.monthly_balances[month];
     END IF;

   END LOOP;

   RETURN;

  END;
$$ LANGUAGE 'plpgsql';

-------------------------------------------------------------------------------
-- max
-------------------------------------------------------------------------------

CREATE OR REPLACE FUNCTION max( arg1 ANYELEMENT, arg2 ANYELEMENT )
RETURNS ANYELEMENT AS $$
  BEGIN

   IF( arg1 > arg2 ) THEN
     RETURN( arg1 );
   ELSE
     RETURN( arg2 );
   END IF;
```

```
  END;
$$ LANGUAGE 'plpgsql';

-------------------------------------------------------------------------------
-- firstSmaller
-------------------------------------------------------------------------------

CREATE OR REPLACE FUNCTION firstSmaller( arg1 ANYELEMENT, arg2 ANYARRAY )
RETURNS ANYELEMENT AS $$
  BEGIN

    FOR i IN array_lower( arg2, 1 ) .. array_upper( arg2, 1 ) LOOP

      IF arg2[i] < arg1 THEN
        RETURN( arg2[i] );
      END IF;

    END LOOP;

    RETURN NULL;

  END;
$$ LANGUAGE 'plpgsql';


-------------------------------------------------------------------------------
-- sum
-------------------------------------------------------------------------------
CREATE OR REPLACE FUNCTION sum( arg1 ANYARRAY ) RETURNS ANYELEMENT AS $
$
  DECLARE
    result ALIAS FOR $0;
  BEGIN

    result := 0;

    FOR i IN array_lower( arg1, 1 ) .. array_upper( arg1, 1 ) LOOP

      IF arg1[i] IS NOT NULL THEN
        result := result + arg1[i];
      END IF;

    END LOOP;

    RETURN( result );

  END;
```

```
$$ LANGUAGE 'plpgsql';
```

Código fonte em:
http://www.conjectrix.com/pgbook/index.html

Mais exemplos:

```
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';
```

Função para remover registros de uma tabela:

```
CREATE OR REPLACE FUNCTION exclui_cliente(pid_cliente int4) RETURNS int4
AS
$body$
DECLARE
   vLinhas int4 DEFAULT 0;
BEGIN
   DELETE FROM clientes WHERE id_cliente = pid_cliente;
   GET DIAGNOSTICS vLinhas = ROW_COUNT;
   RETURN vLinhas;
END;
```

```
$body$ LANGUAGE 'plpgsql';
```

Receber como parâmetro o identificador de um cliente e devolver o seu volume de compras médio:

```
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';
```

Exemplo com cursores:

```
CREATE OR REPLACE FUNCTION media_compras() RETURNS VOID AS
$body$
DECLARE
   linhaCliente clientes%ROWTYPE;
   mediaCompras numeric(9,2);
   totalCompras numeric(9,2);
   periodo int4;
BEGIN
   FOR linhaCliente IN SELECT * FROM clientes LOOP
      periodo := (current_date linhaCliente.data_inclusao);
      SELECT SUM(valor_total) INTO totalCompras
      FROM pedidos WHERE id_cliente = pid_cliente;
      mediaCompras := totalCompras / periodo;
      UPDATE clientes SET media_compras = mediaCompras
      WHERE id_cliente = pid_cliente;
   END LOOP;
   RETURN;
END;
```

```
$body$ LANGUAGE 'plpgsql';
```

Função baseada na exclui_cliente mas genérica, para excluir em qualquer tabela:

```
CREATE OR REPLACE FUNCTION exclui_registro(nome_tabela text, nome_chave
text, id int4) RETURNS int4 AS
$body$
DECLARE
   vLinhas int4 DEFAULT 0;
BEGIN
   EXECUTE 'DELETE FROM ' || nome_tabela || '
      WHERE ' || nome_chave || ' = ' || id;
   GET DIAGNOSTICS vLinhas = ROW_COUNT;
   RETURN vLinhas;
END;
$body$ LANGUAGE 'plpgsql';
```

Outro exemplo de uso da função de exclusão mas agora controlando o uso malicioso de exclusão de uma tabela inteira (excluindo apenas um registro):

```
CREATE OR REPLACE FUNCTION exclui_registro(nome_tabela text, nome_chave
text, id int4) RETURNS int4 AS
$body$
DECLARE
   vLinhas int4 DEFAULT 0;
BEGIN
   EXECUTE 'DELETE FROM ' || nome_tabela || '
      WHERE ' || nome_chave || ' = ' || id;
   GET DIAGNOSTICS vLinhas = ROW_COUNT;
   IF vLinhas > 1 THEN
      RAISE EXCEPTION 'A exclusão de mais de uma linha não é permitida.';
   END IF;
   RETURN vLinhas;
END;
$body$ LANGUAGE 'plpgsql' SECURITY DEFINER;
```

Exemplo com Arrays:

```
CREATE OR REPLACE FUNCTION sp_teste(teste_array integer[]) returns integer[] as
$$
begin

return teste_array;
end;
$$
language plpgsql;
```

select sp_teste('{123}');

SELECT sp_teste(ARRAY[1,2]);
OU
SELECT sp_teste('{1,2}');

-Leo

Tente:
SELECT sp_teste('{1234, 4321}'::int[]);

Osvaldo

SELECT sp_teste('{1234,4321}'::integer[]);
ou
SELECT sp_teste('{1234,4321}');

Como posso retornar diversos campos em uma store procedure?

Tenho uma SP que funciona como uma função para outras SP's, e o processamento dessa SP resulta em 4 resultados que a SP que tiver chamado essa terá que utilizar. Pode ser parametros out ou return array? como funciona isso em Postgres?
Se alguem tiver algum link para indicar também ajuda.

Rúben Lício Reis

```
CREATE FUNCTION r(c1 out text,c2 out text)
LANGUAGE plpgsql;
AS $c$
BEGIN
SELECT 'teste1' AS foo,'teste2' AS bar
   INTO $1,$2;
END;
$c$;
SELECT * FROM r();
```

Leo

Validação de CPF com Pl/ PgSQL

A partir de hoje passamos a divulgar algoritmos de funções e consultas que sejam de utilidade pública. A validação de CPF com Pl/ PgSQL foi escolhida em primeiro lugar por ser um algoritmo simples mas bastante útil (além disto, procurei em vários sites e não encontrei um exemplo em PL/ PGSQL).

O CPF é utilizado por muitos sistemas brasileiros como identificação dos indivíduos. Validar o CPF é fazer a verificação dos dois últimos dígitos que são gerados a partir dos nove primeiros. O código

abaixo foi uma tradução mais ou menos literal do código em javascript deste site.

Talvez possa ser feita otimização ou melhoria neste algoritmo, mas a idéia é que vocês o melhorem e atualizem neste site. Estejam à vontade para utilizar e compartilhar este código.

```
CREATE OR REPLACE FUNCTION CPF_Validar(par_cpf varchar(11)) RETURNS integer AS $$
-- ROTINA DE VALIDAÇÃO DE CPF
-- Conversão para o PL/ PGSQL: Cláudio Leopoldino - http://postgresqlbr.blogspot.com/
-- Algoritmo original: http://webmasters.neting.com/msg07743.html
-- Retorna 1 para CPF correto.
DECLARE
x real;
y real; --Variável temporária
soma integer;
dig1 integer; --Primeiro dígito do CPF
dig2 integer; --Segundo dígito do CPF
len integer; -- Tamanho do CPF
contloop integer; --Contador para loop
val_par_cpf varchar(11); --Valor do parâmetro
BEGIN
-- Teste do tamanho da string de entrada
IF char_length(par_cpf) = 11 THEN
ELSE
RAISE NOTICE 'Formato inválido: %',$1;
RETURN 0;
END IF;
-- Inicialização
x := 0;
soma := 0;
dig1 := 0;
dig2 := 0;
contloop := 0;
val_par_cpf := $1; --Atribuição do parâmetro a uma variável interna
len := char_length(val_par_cpf);
x := len -1;
--Loop de multiplicação - dígito 1
contloop :=1;
WHILE contloop <= (len -2) LOOP
y := CAST(substring(val_par_cpf from contloop for 1) AS NUMERIC);
soma := soma + ( y * x);
x := x - 1;
contloop := contloop +1;
END LOOP;
dig1 := 11 - CAST((soma % 11) AS INTEGER);
if (dig1 = 10) THEN dig1 :=0 ; END IF;
if (dig1 = 11) THEN dig1 :=0 ; END IF;

-- Dígito 2
```

```
x := 11; soma :=0;
contloop :=1;
WHILE contloop <= (len -1) LOOP
soma := soma + CAST((substring(val_par_cpf FROM contloop FOR 1)) AS REAL) * x;
x := x - 1;
contloop := contloop +1;
END LOOP;
dig2 := 11 - CAST ((soma % 11) AS INTEGER);
IF (dig2 = 10) THEN dig2 := 0; END IF;
IF (dig2 = 11) THEN dig2 := 0; END IF;
--Teste do CPF
IF ((dig1 || '' || dig2) = substring(val_par_cpf FROM len-1 FOR 2)) THEN
RETURN 1;
ELSE
RAISE NOTICE 'DV do CPF Inválido: %',$1;
RETURN 0;
END IF;
END;
$$ LANGUAGE PLPGSQL;
```

Fonte: http://postgresqlbr.blogspot.com/2008/06/validao-de-cpf-com-pl-pgsql.html

Mais informações:

http://www.postgresql.org/docs/current/static/PL/pgSQL.html
http://pgdocptbr.sourceforge.net/pg80/plpgsql.html
http://www.devmedia.com.br/articles/viewcomp.asp?comp=6906
http://www.tek-tips.com/faqs.cfm?fid=3463
http://imasters.uol.com.br/artigo/1308/stored_procedures_triggers_functions
http://postgresql.org.br/Documenta%C3%A7%C3%A3o?action=AttachFile&do=get&target=procedures.pdf

Gatilhos

Sumário

Este capítulo descreve como escrever funções de gatilho. As funções de gatilho podem ser escritas na linguagem C, ou em uma das várias linguagens procedurais disponíveis. No momento não é possível escrever funções de gatilho na linguagem SQL.

Visão geral do comportamento dos gatilhos

O gatilho pode ser definido para executar antes ou depois de uma operação de INSERT, UPDATE ou DELETE, tanto uma vez para cada linha modificada quanto uma vez por instrução SQL. Quando ocorre o evento do gatilho, a função de gatilho é chamada no momento apropriado para tratar o evento.

A função de gatilho deve ser definida antes do gatilho ser criado. A função de gatilho deve ser declarada como uma função que não recebe argumentos e que retorna o tipo trigger (A função de gatilho recebe sua entrada através de estruturas TriggerData passadas especialmente para estas funções, e não na forma comum de argumentos de função).

Tendo sido criada a função de gatilho adequada, o gatilho é estabelecido através do comando *CREATE TRIGGER*. A mesma função de gatilho pode ser utilizada por vários gatilhos.

Existem dois tipos de gatilhos: gatilhos-por-linha e gatilhos-por-instrução. Em um gatilho-por-linha, a função é chamada uma vez para cada linha afetada pela instrução que disparou o gatilho. Em contraste, um gatilho-por-instrução é chamado somente uma vez quando a instrução apropriada é executada, a despeito do número de linhas afetadas pela instrução. Em particular, uma instrução que não afeta nenhuma linha ainda assim resulta na execução dos gatilhos-por-instrução aplicáveis. Este dois tipos de gatilho são algumas vezes chamados de "gatilhos no nível-de-linha" e "gatilhos no nível-de-instrução", respectivamente.

Os gatilhos no nível-de-instrução "BEFORE" (antes) naturalmente disparam antes da instrução começar a fazer alguma coisa, enquanto os gatilhos no nível-de-instrução "AFTER" (após) disparam bem no final da instrução. Os gatilhos no nível-de-linha "BEFORE" (antes) disparam logo antes da operação em uma determinada linha, enquanto os gatilhos no nível-de-linha "AFTER" (após) disparam no fim da instrução (mas antes dos gatilhos no nível-de-instrução "AFTER").

As funções de gatilho chamadas por gatilhos-por-instrução devem sempre retornar NULL. As funções de gatilho chamadas por gatilhos-por-linha podem retornar uma linha da tabela (um valor do tipo HeapTuple) para o executor da chamada, se assim o decidirem. Os gatilhos no nível-de-linha disparados antes de uma operação possuem as seguintes escolhas:

- Podem retornar NULL para saltar a operação para a linha corrente. Isto instrui ao executor a não realizar a operação no nível-de-linha que chamou o gatilho (a inserção ou a modificação de uma determinada linha da tabela).

- Para os gatilhos de INSERT e UPDATE, no nível-de-linha apenas, a linha retornada se torna a linha que será inserida ou que substituirá a linha sendo atualizada. Isto permite à função de gatilho modificar a linha sendo inserida ou atualizada.

Um gatilho no nível-de-linha, que não pretenda causar nenhum destes comportamentos, deve ter o cuidado de retornar como resultado a mesma linha que recebeu (ou seja, a linha NEW para os gatilhos de INSERT e UPDATE, e a linha OLD para os gatilhos de DELETE).

O valor retornado é ignorado nos gatilhos no nível-de-linha disparados após a operação e, portanto, podem muito bem retornar NULL.

Se for definido mais de um gatilho para o mesmo evento na mesma relação, os gatilhos são disparados pela ordem alfabética de seus nomes. No caso dos gatilhos para antes, a linha possivelmente modificada retornada por cada gatilho se torna a entrada do próximo gatilho. Se algum dos gatilhos para antes retornar NULL, a operação é abandonada e os gatilhos seguintes não são disparados.

Tipicamente, os gatilhos no nível-de-linha que disparam antes são utilizados para verificar ou modificar os dados que serão inseridos ou atualizados. Por exemplo, um gatilho que dispara antes pode ser utilizado para inserir a hora corrente em uma coluna do tipo timestamp, ou para verificar se dois elementos da linha são consistentes. Os gatilhos no nível-de-linha que disparam depois fazem mais sentido para propagar as atualizações para outras tabelas, ou fazer verificação de consistência com relação a outras tabelas. O motivo desta divisão de trabalho é porque um gatilho que dispara depois pode ter certeza de estar vendo o valor final da linha, enquanto um gatilho que dispara antes não pode ter esta certeza; podem haver outros gatilhos que disparam antes disparando após o mesmo. Se não houver nenhum motivo específico para fazer um gatilho disparar antes ou depois, o gatilho para antes é mais eficiente, uma vez que a informação sobre a operação não precisa ser salva até o fim da instrução.

Se a função de gatilho executar comandos SQL, então estes comandos podem disparar gatilhos novamente. Isto é conhecido como cascatear gatilhos. Não existe limitação direta do número de níveis de cascateamento. É possível que o cascateamento cause chamadas recursivas do mesmo gatilho; por exemplo, um gatilho para INSERT pode executar um comando que insere uma linha adicional na mesma tabela, fazendo com que o gatilho para INSERT seja disparado novamente. É responsabilidade do programador do gatilho evitar recursões infinitas nestes casos.

Ao se definir um gatilho, podem ser especificados argumentos para o mesmo. A finalidade de se incluir argumentos na definição do gatilho é permitir que gatilhos diferentes com requisitos semelhantes chamem a mesma função. Por exemplo, pode existir uma função de gatilho generalizada que recebe como argumentos dois nomes de colunas e coloca o usuário corrente em uma e a data corrente em outra. Escrita de maneira apropriada, esta função de gatilho se torna independente da tabela específica para a qual está sendo utilizada. Portanto, a mesma função pode ser utilizada para eventos de INSERT em qualquer tabela com colunas apropriadas, para acompanhar automaticamente a criação de registros na tabela de transação, por exemplo. Também pode ser utilizada para acompanhar os eventos de última atualização, se for definida em um gatilho de UPDATE.

Cada linguagem de programação que suporta gatilhos possui o seu próprio método para tornar os dados de entrada do gatilho disponíveis para a função de gatilho. Estes dados de entrada incluem o tipo de evento do gatilho (ou seja, INSERT ou UPDATE), assim como os argumentos listados em CREATE TRIGGER. Para um gatilho no nível-de-linha, os dados de entrada também incluem as linhas NEW para os gatilhos de INSERT e UPDATE, e/ou a linha OLD para os gatilhos de UPDATE e DELETE. Atualmente não há maneira de examinar individualmente as linhas

modificadas pela instrução nos gatilhos no nível-de-instrução.

Visibilidade das mudanças nos dados

Se forem executados comandos SQL na função de gatilho, e estes comandos acessarem a tabela para a qual o gatilho se destina, então deve-se estar ciente das regras de visibilidade dos dados, porque estas determinam se estes comandos SQL enxergam as mudanças nos dados para os quais o gatilho foi disparado. Em resumo:

- Os gatilhos no nível-de-instrução seguem regras simples de visibilidade: nenhuma das modificações feitas pela instrução é enxergada pelos gatilhos no nível-de-instrução chamados antes da instrução, enquanto todas as modificações são enxergadas pelos gatilhos no nível-de-instrução que disparam depois da instrução.
- As modificações nos dados (inserção, atualização e exclusão) causadoras do disparo do gatilho, naturalmente *não* são enxergadas pelos comandos SQL executados em um gatilho no nível-de-linha que dispara antes, porque ainda não ocorreram.
- Entretanto, os comandos SQL executados em um gatilho no nível-de-linha para antes *enxergam* os efeitos das modificações nos dados das linhas processadas anteriormente no mesmo comando externo. Isto requer cautela, uma vez que a ordem destes eventos de modificação geralmente não é previsível; um comando SQL que afeta várias linhas pode atuar sobre as linhas em qualquer ordem.
- Quando é disparado um gatilho no nível-de-linha para depois, todas as modificações nos dados feitas pelo comando externo já estão completas, sendo enxergadas pela função de gatilho chamada.

Podem ser encontradas informações adicionais sobre as regras de visibilidade dos dados na Seção 40.4. O exemplo na Seção 32.4 contém uma demonstração destas regras.

Até a versão atual não existe como criar funções de gatilho na linguagem SQL.

Uma função de gatilho pode ser criada para executar antes (BEFORE) ou após (AFTER) as consultas INSERT, UPDATE OU DELETE, uma vez para cada registro (linha) modificado ou por instrução SQL. Logo que ocorre um desses eventos do gatilho a função do gatilho é disparada automaticamente para tratar o evento.

A função de gatilho deve ser declarada como uma função que não recebe argumentos e que retorna o tipo TRIGGER.

Após criar a função de gatilho, estabelecemos o gatilho pelo comando CREATE TRIGGER. Uma função de gatilho pode ser utilizada por vários gatilhos.

As funções de gatilho chamadas por gatilhos-por-instrução devem sempre retornar NULL.

As funções de gatilho chamadas por gatilhos-por-linha podem retornar uma linha da tabela (um valor do tipo HeapTuple) para o executor da chamada, se assim o decidirem.

Sintaxe:
CREATE TRIGGER nome { BEFORE | AFTER } { evento [OR ...] }

```
ON tabela [ FOR [ EACH ] { ROW | STATEMENT } ]
EXECUTE PROCEDURE nome_da_função ( argumentos )
```

Triggers em PostgreSQL sempre executam uma função que retorna TRIGGER.

O gatilho fica associado à tabela especificada e executa a função especificada nome_da_função quando determinados eventos ocorrerem.

O gatilho pode ser especificado para disparar antes de tentar realizar a operação na linha (antes das restrições serem verificadas e o comando INSERT, UPDATE ou DELETE ser tentado), ou após a operação estar completa (após as restrições serem verificadas e o INSERT, UPDATE ou DELETE ter completado).

evento
Um entre INSERT, UPDATE ou DELETE; especifica o evento que dispara o gatilho. Vários eventos podem ser especificados utilizando OR.

Exemplos:

```
CREATE TABLE empregados(
  codigo int4 NOT NULL,
  nome varchar,
  salario int4,
  departamento_cod int4,
  ultima_data timestamp,
  ultimo_usuario varchar(50),
  CONSTRAINT empregados_pkey PRIMARY KEY (codigo) )


CREATE FUNCTION empregados_gatilho() RETURNS trigger AS $empregados_gatilho$
    BEGIN
        -- Verificar se foi fornecido o nome e o salário do empregado
        IF NEW.nome IS NULL THEN
            RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
        END IF;
        IF NEW.salario IS NULL THEN
            RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome;
        END IF;

        -- Quem paga para trabalhar?
        IF NEW.salario < 0 THEN
            RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome;
        END IF;

        -- Registrar quem alterou a folha de pagamento e quando
        NEW.ultima_data := 'now';
        NEW.ultimo_usuario := current_user;
        RETURN NEW;
    END;
```

```
$empregados_gatilho$ LANGUAGE plpgsql;

CREATE TRIGGER empregados_gatilho BEFORE INSERT OR UPDATE ON empregados
   FOR EACH ROW EXECUTE PROCEDURE empregados_gatilho();

INSERT INTO empregados (codigo,nome, salario) VALUES (5,'João',1000);
INSERT INTO empregados (codigo,nome, salario) VALUES (6,'José',1500);
INSERT INTO empregados (codigo,nome, salario) VALUES (7,'Maria',2500);

SELECT * FROM empregados;

INSERT INTO empregados (codigo,nome, salario) VALUES (5,NULL,1000);
```

NEW – Para INSERT e UPDATE
OLD – Para DELETE

```
CREATE TABLE empregados (
   nome  varchar NOT NULL,
   salario     integer
);

CREATE TABLE empregados_audit(
   operacao    char(1)   NOT NULL,
   usuario     varchar      NOT NULL,
   data        timestamp NOT NULL,
   nome    varchar      NOT NULL,
   salario     integer
);


CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS
$emp_audit$
   BEGIN
      --
      -- Cria uma linha na tabela emp_audit para refletir a operação
      -- realizada na tabela emp. Utiliza a variável especial TG_OP
      -- para descobrir a operação sendo realizada.
      --
      IF (TG_OP = 'DELETE') THEN
         INSERT INTO emp_audit SELECT 'E', user, now(), OLD.*;
         RETURN OLD;
      ELSIF (TG_OP = 'UPDATE') THEN
         INSERT INTO emp_audit SELECT 'A', user, now(), NEW.*;
         RETURN NEW;
      ELSIF (TG_OP = 'INSERT') THEN
         INSERT INTO emp_audit SELECT 'I', user, now(), NEW.*;
         RETURN NEW;
      END IF;
```

```
      RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
   END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON empregados
   FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

INSERT INTO empregados (nome, salario) VALUES ('João',1000);
INSERT INTO empregados (nome, salario) VALUES ('José',1500);
INSERT INTO empregados (nome, salario) VALUES ('Maria',250);
UPDATE empregados SET salario = 2500 WHERE nome = 'Maria';
DELETE FROM empregados WHERE nome = 'João';

SELECT * FROM empregados;

SELECT * FROM empregados_audit;
```

Outro exemplo:

```
CREATE TABLE empregados (
   codigo         serial  PRIMARY KEY,
   nome    varchar    NOT NULL,
   salario     integer
);

CREATE TABLE empregados_audit(
   usuario        varchar      NOT NULL,
   data           timestamp NOT NULL,
   id             integer   NOT NULL,
   coluna         text      NOT NULL,
   valor_antigo    text      NOT NULL,
   valor_novo      text      NOT NULL
);

CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS
$emp_audit$
   BEGIN
      --
      -- Não permitir atualizar a chave primária
      --
      IF (NEW.codigo <> OLD.codigo) THEN
         RAISE EXCEPTION 'Não é permitido atualizar o campo codigo';
      END IF;
      --
      -- Inserir linhas na tabela emp_audit para refletir as alterações
      -- realizada na tabela emp.
      --
      IF (NEW.nome <> OLD.nome) THEN
```

```
        INSERT INTO emp_audit SELECT current_user, current_timestamp,
                NEW.id, 'nome', OLD.nome, NEW.nome;
      END IF;
      IF (NEW.salario <> OLD.salario) THEN
        INSERT INTO emp_audit SELECT current_user, current_timestamp,
                NEW.codigo, 'salario', OLD.salario, NEW.salario;
      END IF;
      RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
   END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
AFTER UPDATE ON empregados
   FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

INSERT INTO empregados (nome, salario) VALUES ('João',1000);
INSERT INTO empregados (nome, salario) VALUES ('José',1500);
INSERT INTO empregados (nome, salario) VALUES ('Maria',2500);
UPDATE empregados SET salario = 2500 WHERE id = 2;
UPDATE empregados SET nome = 'Maria Cecília' WHERE id = 3;
UPDATE empregados SET codigo=100 WHERE codigo=1;
ERRO:  Não é permitido atualizar o campo codigo
SELECT * FROM empregados;

SELECT * FROM empregados_audit;
```

Crie a mesma função que insira o nome da empresa e o nome do cliente retornando o id de ambos

```
create or replace function empresa_cliente_id(varchar,varchar) returns _int4 as
'
declare
   nempresa alias for $1;
   ncliente alias for $2;
   empresaid integer;
   clienteid integer;
begin
   insert into empresas(nome) values(nempresa);
   insert into clientes(fkempresa,nome)   values (currval (''empresas_id_seq''), ncliente);
   empresaid := currval(''empresas_id_seq'');
   clienteid := currval(''clientes_id_seq'');

   return ''{''|| empresaid ||'',''|| clienteid ||''}'';
end;
'
language 'plpgsql';
```

Crie uma função onde passamos como parâmetro o id do cliente e seja retornado o seu

```
nome

create or replace function id_nome_cliente(integer) returns text as
'
declare
    r record;
begin
    select into r * from clientes where id = $1;
    if not found then
       raise exception "Cliente não existente !";
    end if;
    return r.nome;
end;
'
language 'plpgsql';
```

Crie uma função que retorne os nome de toda a tabela clientes concatenados em um só campo

```
create or replace function clientes_nomes() returns text as
'
declare
      x text;
      r record;
begin
    x:="Inicio";
    for r in select * from clientes order by id loop
       x:= x||" : "||r.nome;
    end loop;
    return x||" : fim";
end;
'
language 'plpgsql';
```

Gatilhos escritos em PL/pgSQL

A linguagem PL/pgSQL pode ser utilizada para definir procedimentos de gatilho. O procedimento de gatilho é criado pelo comando CREATE FUNCTION, declarando o procedimento como uma função sem argumentos e que retorna o tipo trigger. Deve ser observado que a função deve ser declarada sem argumentos, mesmo que espere receber os argumentos especificados no comando CREATE TRIGGER — os argumentos do gatilho são passados através de TG_ARGV, conforme descrito abaixo.

Quando uma função escrita em PL/pgSQL é chamada como um gatilho, diversas variáveis especiais são criadas automaticamente no bloco de nível mais alto. São estas:

NEW

Tipo de dado RECORD; variável contendo a nova linha do banco de dados, para as operações de INSERT/UPDATE nos gatilhos no nível de linha. O valor desta variável é NULL nos gatilhos no nível de instrução.

OLD

Tipo de dado RECORD; variável contendo a antiga linha do banco de dados, para as operações de UPDATE/DELETE nos gatilhos no nível de linha. O valor desta variável é NULL nos gatilhos no nível de instrução.

TG_NAME

Tipo de dado name; variável contendo o nome do gatilho disparado.

TG_WHEN

Tipo de dado text; uma cadeia de caracteres contendo BEFORE ou AFTER, dependendo da definição do gatilho.

TG_LEVEL

Tipo de dado text; uma cadeia de caracteres contendo ROW ou STATEMENT, dependendo da definição do gatilho.

TG_OP

Tipo de dado text; uma cadeia de caracteres contendo INSERT, UPDATE, ou DELETE, informando para qual operação o gatilho foi disparado.

TG_RELID

Tipo de dado oid; o ID de objeto da tabela que causou o disparo do gatilho.

TG_RELNAME

Tipo de dado name; o nome da tabela que causou o disparo do gatilho.

TG_NARGS

Tipo de dado integer; o número de argumentos fornecidos ao procedimento de gatilho na instrução CREATE TRIGGER.

TG_ARGV[]

Tipo de dado matriz de text; os argumentos da instrução CREATE TRIGGER. O contador do índice começa por 0. Índices inválidos (menor que 0 ou maior ou igual a tg_nargs) resultam em um valor nulo.

Uma função de gatilho deve retornar nulo, ou um valor registro/linha possuindo a mesma estrutura da tabela para a qual o gatilho foi disparado.

Os gatilhos no nível de linha disparados BEFORE (antes) podem retornar nulo, para sinalizar ao gerenciador do gatilho para pular o restante da operação para esta linha (ou seja, os gatilhos posteriores não serão disparados, e não ocorrerá o INSERT/UPDATE/DELETE para esta linha. Se for retornado um valor diferente de nulo, então a operação prossegue com este valor de linha. Retornar um valor de linha diferente do valor original de NEW altera a linha que será inserida ou atualizada (mas não tem efeito direto no caso do DELETE). Para alterar a linha a ser armazenada, é possível substituir valores individuais diretamente em NEW e retornar o NEW modificado, ou construir um novo registro/linha completo a ser retornado.

O valor retornado por um gatilho BEFORE ou AFTER no nível de instrução, ou por um gatilho AFTER no nível de linha, é sempre ignorado; pode muito bem ser nulo. Entretanto, qualquer um destes tipos de gatilho pode interromper toda a operação gerando um erro.

O Exemplo 35-1 mostra um exemplo de procedimento de gatilho escrito em PL/pgSQL.

Exemplo 35-1. Procedimento de gatilho PL/pgSQL

O gatilho deste exemplo garante que quando é inserida ou atualizada uma linha na tabela, fica sempre registrado nesta linha o usuário que efetuou a inserção ou a atualização, e quando isto ocorreu. Além disso, o gatilho verifica se é fornecido o nome do empregado, e se o valor do salário é um número positivo.

```
CREATE TABLE emp (
    nome_emp       text,
    salario        integer,
    ultima_data    timestamp,
    ultimo_usuario text
);

CREATE FUNCTION emp_gatilho() RETURNS trigger AS $emp_gatilho$
    BEGIN
        -- Verificar se foi fornecido o nome e o salário do empregado
        IF NEW.nome_emp IS NULL THEN
            RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
        END IF;
        IF NEW.salario IS NULL THEN
            RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome_emp;
        END IF;

        -- Quem paga para trabalhar?
        IF NEW.salario < 0 THEN
            RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome_emp;
        END IF;

        -- Registrar quem alterou a folha de pagamento e quando
        NEW.ultima_data := 'now';
        NEW.ultimo_usuario := current_user;
        RETURN NEW;
    END;
$emp_gatilho$ LANGUAGE plpgsql;

CREATE TRIGGER emp_gatilho BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_gatilho();
```

```
INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',2500);

SELECT * FROM emp;

 nome_emp | salario |        ultima_data         | ultimo_usuario
----------+---------+----------------------------+----------------
 João     |    1000 | 2005-11-25 07:07:50.59532  | folha
 José     |    1500 | 2005-11-25 07:07:50.691905 | folha
 Maria    |    2500 | 2005-11-25 07:07:50.694995 | folha
(3 linhas)
```

Exemplo 35-2. Procedimento de gatilho PL/pgSQL para registrar inserção e atualização

O gatilho deste exemplo garante que quando é inserida ou atualizada uma linha na tabela, fica sempre registrado nesta linha o usuário que efetuou a inserção ou a atualização, e quando isto ocorreu. Porém, diferentemente do gatilho anterior, a criação e a atualização da linha são registradas em colunas diferentes. Além disso, o gatilho verifica se é fornecido o nome do empregado, e se o valor do salário é um número positivo. [1]

```
CREATE TABLE emp (
    nome_emp       text,
    salario        integer,
    usu_cria       text,        -- Usuário que criou a linha
    data_cria      timestamp,   -- Data da criação da linha
    usu_atu        text,        -- Usuário que fez a atualização
    data_atu       timestamp    -- Data da atualização
);

CREATE FUNCTION emp_gatilho() RETURNS trigger AS $emp_gatilho$
    BEGIN
        -- Verificar se foi fornecido o nome do empregado
        IF NEW.nome_emp IS NULL THEN
            RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
        END IF;
        IF NEW.salario IS NULL THEN
            RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome_emp;
        END IF;

        -- Quem paga para trabalhar?
        IF NEW.salario < 0 THEN
            RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome_emp;
        END IF;

        -- Registrar quem criou a linha e quando
        IF (TG_OP = 'INSERT') THEN
            NEW.data_cria := current_timestamp;
            NEW.usu_cria  := current_user;
        -- Registrar quem alterou a linha e quando
        ELSIF (TG_OP = 'UPDATE') THEN
            NEW.data_atu := current_timestamp;
            NEW.usu_atu  := current_user;
        END IF;
        RETURN NEW;
    END;
$emp_gatilho$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_gatilho BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_gatilho();

INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',250);
UPDATE emp SET salario = 2500 WHERE nome_emp = 'Maria';

SELECT * FROM emp;

 nome_emp | salario | usu_cria |         data_cria          | usu_atu |
data_atu
----------+---------+----------+----------------------------+---------
+----------------------------
 João     |    1000 | folha    | 2005-11-25 08:11:40.63868  |         |
 José     |    1500 | folha    | 2005-11-25 08:11:40.674356 |         |
 Maria    |    2500 | folha    | 2005-11-25 08:11:40.679592 | folha   |
2005-11-25 08:11:40.682394
(3 linhas)
```

Uma outra maneira de registrar as modificações na tabela envolve a criação de uma nova tabela contendo uma linha para cada inserção, atualização ou exclusão que ocorra. Esta abordagem pode ser considerada como uma auditoria das mudanças na tabela. O Exemplo 35-3 mostra um procedimento de gatilho de auditoria em PL/pgSQL.

Exemplo 35-3. Procedimento de gatilho PL/pgSQL para auditoria

Este gatilho garante que todas as inserções, atualizações e exclusões de linha na tabela emp são registradas na tabela emp_audit, para permitir auditar as operações efetuadas na tabela emp. O nome de usuário e a hora corrente são gravadas na linha, junto com o tipo de operação que foi realizada.

```
CREATE TABLE emp (
    nome_emp    text NOT NULL,
    salario     integer
);

CREATE TABLE emp_audit(
    operacao    char(1)   NOT NULL,
    usuario     text      NOT NULL,
    data        timestamp NOT NULL,
    nome_emp    text      NOT NULL,
    salario     integer
);

CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Cria uma linha na tabela emp_audit para refletir a operação
        -- realizada na tabela emp. Utiliza a variável especial TG_OP
        -- para descobrir a operação sendo realizada.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'E', user, now(), OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'A', user, now(), NEW.*;
            RETURN NEW;
```

```
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', user, now(), NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho
AFTER
    END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',250);
UPDATE emp SET salario = 2500 WHERE nome_emp = 'Maria';
DELETE FROM emp WHERE nome_emp = 'João';

SELECT * FROM emp;

 nome_emp | salario
----------+---------
 José     |    1500
 Maria    |    2500
(2 linhas)

SELECT * FROM emp_audit;

 operacao | usuario |            data             | nome_emp | salario
----------+---------+-----------------------------+----------+---------
 I        | folha   | 2005-11-25 09:06:03.008735 | João     |    1000
 I        | folha   | 2005-11-25 09:06:03.014245 | José     |    1500
 I        | folha   | 2005-11-25 09:06:03.049443 | Maria    |     250
 A        | folha   | 2005-11-25 09:06:03.052972 | Maria    |    2500
 E        | folha   | 2005-11-25 09:06:03.056774 | João     |    1000
(5 linhas)
```

Exemplo 35-4. Procedimento de gatilho PL/pgSQL para auditoria no nível de coluna

Este gatilho registra todas as atualizações realizadas nas colunas nome_emp e salario da tabela emp na tabela emp_audit (isto é, as colunas são auditadas). O nome de usuário e a hora corrente são registrados junto com a chave da linha (id) e a informação atualizada. Não é permitido atualizar a chave da linha. Este exemplo difere do anterior pela auditoria ser no nível de coluna, e não no nível de linha. [2]

```
CREATE TABLE emp (
    id          serial  PRIMARY KEY,
    nome_emp    text    NOT NULL,
    salario     integer
);

CREATE TABLE emp_audit(
    usuario         text      NOT NULL,
    data            timestamp NOT NULL,
    id              integer   NOT NULL,
    coluna          text      NOT NULL,
    valor_antigo    text      NOT NULL,
```

```
    valor_novo      text      NOT NULL
);

CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Não permitir atualizar a chave primária
        --
        IF (NEW.id <> OLD.id) THEN
            RAISE EXCEPTION 'Não é permitido atualizar o campo ID';
        END IF;
        --
        -- Inserir linhas na tabela emp_audit para refletir as alterações
        -- realizada na tabela emp.
        --
        IF (NEW.nome_emp <> OLD.nome_emp) THEN
           INSERT INTO emp_audit SELECT current_user, current_timestamp,
                       NEW.id, 'nome_emp', OLD.nome_emp, NEW.nome_emp;
        END IF;
        IF (NEW.salario <> OLD.salario) THEN
           INSERT INTO emp_audit SELECT current_user, current_timestamp,
                       NEW.id, 'salario', OLD.salario, NEW.salario;
        END IF;
        RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho
AFTER
    END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
AFTER UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

INSERT INTO emp (nome_emp, salario) VALUES ('João',1000);
INSERT INTO emp (nome_emp, salario) VALUES ('José',1500);
INSERT INTO emp (nome_emp, salario) VALUES ('Maria',2500);
UPDATE emp SET salario = 2500 WHERE id = 2;
UPDATE emp SET nome_emp = 'Maria Cecília' WHERE id = 3;
UPDATE emp SET id=100 WHERE id=1;
ERRO:  Não é permitido atualizar o campo ID

SELECT * FROM emp;

 id |   nome_emp    | salario
----+---------------+---------
  1 | João          |    1000
  2 | José          |    2500
  3 | Maria Cecília |    2500
(3 linhas)

SELECT * FROM emp_audit;

 usuario |            data            | id |  coluna  | valor_antigo |
valor_novo
---------+----------------------------+----+----------+--------------
+---------------
 folha   | 2005-11-25 12:21:08.493268 |  2 | salario  | 1500         | 2500
 folha   | 2005-11-25 12:21:08.49822  |  3 | nome_emp | Maria        | Maria
Cecília
(2 linhas)
```

Uma das utilizações de gatilho é para manter uma tabela contendo o sumário de outra tabela. O sumário produzido pode ser utilizado no lugar da tabela original em diversas consultas — geralmente com um tempo de execução bem menor. Esta técnica é muito utilizada em Armazém de Dados (Data Warehousing), onde as tabelas dos dados medidos ou observados (chamadas de tabelas fato) podem ser muito grandes. O Exemplo 35-5 mostra um procedimento de gatilho em PL/pgSQL para manter uma tabela de sumário de uma tabela fato em um armazém de dados.

Exemplo 35-5. Procedimento de gatilho PL/pgSQL para manter uma tabela sumário

O esquema que está detalhado a seguir é parcialmente baseado no exemplo *Grocery Store* do livro *The Data Warehouse Toolkit* de Ralph Kimball.

```
--
-- Main tables - time dimension and sales fact.
--
CREATE TABLE time_dimension (
    time_key                    integer NOT NULL,
    day_of_week                 integer NOT NULL,
    day_of_month                integer NOT NULL,
    month                       integer NOT NULL,
    quarter                     integer NOT NULL,
    year                        integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
    time_key                    integer NOT NULL,
    product_key                 integer NOT NULL,
    store_key                   integer NOT NULL,
    amount_sold                 numeric(12,2) NOT NULL,
    units_sold                  integer NOT NULL,
    amount_cost                 numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
    time_key                    integer NOT NULL,
    amount_sold                 numeric(15,2) NOT NULL,
    units_sold                  numeric(12) NOT NULL,
    amount_cost                 numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON sales_summary_bytime(time_key);

--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER AS
$maint_sales_summary_bytime$
    DECLARE
        delta_time_key          integer;
        delta_amount_sold       numeric(15,2);
        delta_units_sold        numeric(12);
        delta_amount_cost       numeric(15,2);
    BEGIN
```

```
        -- Work out the increment/decrement amount(s).
        IF (TG_OP = 'DELETE') THEN

            delta_time_key = OLD.time_key;
            delta_amount_sold = -1 * OLD.amount_sold;
            delta_units_sold = -1 * OLD.units_sold;
            delta_amount_cost = -1 * OLD.amount_cost;

        ELSIF (TG_OP = 'UPDATE') THEN

            -- forbid updates that change the time_key -
            -- (probably not too onerous, as DELETE + INSERT is how most
            -- changes will be made).
            IF ( OLD.time_key != NEW.time_key) THEN
                RAISE EXCEPTION 'Update of time_key : % -> % not allowed',
OLD.time_key, NEW.time_key;
            END IF;

            delta_time_key = OLD.time_key;
            delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
            delta_units_sold = NEW.units_sold - OLD.units_sold;
            delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

        ELSIF (TG_OP = 'INSERT') THEN

            delta_time_key = NEW.time_key;
            delta_amount_sold = NEW.amount_sold;
            delta_units_sold = NEW.units_sold;
            delta_amount_cost = NEW.amount_cost;

        END IF;


        -- Update the summary row with the new values.
        UPDATE sales_summary_bytime
            SET amount_sold = amount_sold + delta_amount_sold,
                units_sold = units_sold + delta_units_sold,
                amount_cost = amount_cost + delta_amount_cost
            WHERE time_key = delta_time_key;


        -- There might have been no row with this time_key (e.g new data!).
        IF (NOT FOUND) THEN
            BEGIN
                INSERT INTO sales_summary_bytime (
                            time_key,
                            amount_sold,
                            units_sold,
                            amount_cost)
                    VALUES (
                            delta_time_key,
                            delta_amount_sold,
                            delta_units_sold,
                            delta_amount_cost
                           );
            EXCEPTION
                --
                -- Catch race condition when two transactions are adding data
                -- for a new time_key.
```

```
                --
                WHEN UNIQUE_VIOLATION THEN
                    UPDATE sales_summary_bytime
                        SET amount_sold = amount_sold + delta_amount_sold,
                            units_sold = units_sold + delta_units_sold,
                            amount_cost = amount_cost + delta_amount_cost
                        WHERE time_key = delta_time_key;

            END;
        END IF;
        RETURN NULL;

    END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
    FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();
```

Exemplo 35-6. Procedimento de gatilho para controlar sobreposição de datas

O gatilho deste exemplo verifica se o compromiso sendo agendado ou modificado se sobrepõe a outro compromisso já agendado. Se houver sobreposição, emite mensagem de erro e não permite a operação. [3]

Abaixo está mostrado o script utilizado para criar a tabela, a função de gatilho e os gatilhos de inserção e atualização.

```
CREATE TABLE agendamentos (
    id          SERIAL PRIMARY KEY,
    nome        TEXT,
    evento      TEXT,
    data_inicio TIMESTAMP,
    data_fim    TIMESTAMP
);

CREATE FUNCTION fun_verifica_agendamentos() RETURNS "trigger" AS
$fun_verifica_agendamentos$
    BEGIN
        /* Verificar se a data de início é maior que a data de fim */
        IF NEW.data_inicio > NEW.data_fim THEN
           RAISE EXCEPTION 'A data de início não pode ser maior que a data de
fim';
        END IF;
        /* Verificar se há sobreposição com agendamentos existentes */
        IF EXISTS (
            SELECT 1
                FROM agendamentos
                WHERE nome = NEW.nome
                  AND ((data_inicio, data_fim) OVERLAPS
                       (NEW.data_inicio, NEW.data_fim))
        )
        THEN
            RAISE EXCEPTION 'impossível agendar - existe outro compromisso';
        END IF;
        RETURN NEW;
    END;
$fun_verifica_agendamentos$ LANGUAGE plpgsql;
```

```
COMMENT ON FUNCTION fun_verifica_agendamentos() IS
    'Verifica se o agendamento é possível';

CREATE TRIGGER trg_agendamentos_ins
    BEFORE INSERT ON agendamentos
    FOR EACH ROW
    EXECUTE PROCEDURE fun_verifica_agendamentos();

CREATE TRIGGER trg_agendamentos_upd
    BEFORE UPDATE ON agendamentos
    FOR EACH ROW
    EXECUTE PROCEDURE fun_verifica_agendamentos();
```

Abaixo está mostrado um exemplo de utilização do gatilho. Deve ser observado que os intervalos ('2005-08-23 14:00:00', '2005-08-23 15:00:00') e ('2005-08-23 15:00:00', '2005-08-23 16:00:00') não se sobrepõem, uma vez que o primeiro intervalo termina às quinze horas, enquanto o segundo intervalo inicia às quinze horas, estando, portanto, o segundo intervalo imediatamente após o primeiro.

```
=> INSERT INTO agendamentos VALUES
(DEFAULT,'Joana','Congresso','2005-08-23','2005-08-24');
=> INSERT INTO agendamentos VALUES
(DEFAULT,'Joana','Viagem','2005-08-24','2005-08-26');
=> INSERT INTO agendamentos VALUES
(DEFAULT,'Joana','Palestra','2005-08-23','2005-08-26');
ERRO:  impossível agendar - existe outro compromisso
=> INSERT INTO agendamentos VALUES (DEFAULT,'Maria','Cabeleireiro','2005-08-23
14:00:00','2005-08-23 15:00:00');
=> INSERT INTO agendamentos VALUES (DEFAULT,'Maria','Manicure','2005-08-23
15:00:00','2005-08-23 16:00:00');
=> INSERT INTO agendamentos VALUES (DEFAULT,'Maria','Médico','2005-08-23
14:30:00','2005-08-23 15:00:00');
ERRO:  impossível agendar - existe outro compromisso
=> UPDATE agendamentos SET data_inicio='2005-08-24' WHERE id=2;
ERRO:  impossível agendar - existe outro compromisso
=> SELECT * FROM agendamentos;

 id | nome  |    evento    |     data_inicio     |      data_fim
----+-------+--------------+---------------------+---------------------
  1 | Joana | Congresso    | 2005-08-23 00:00:00 | 2005-08-24 00:00:00
  2 | Joana | Viagem       | 2005-08-24 00:00:00 | 2005-08-26 00:00:00
  4 | Maria | Cabeleireiro | 2005-08-23 14:00:00 | 2005-08-23 15:00:00
  5 | Maria | Manicure     | 2005-08-23 15:00:00 | 2005-08-23 16:00:00
(4 linhas)
```

Notas

[1] Exemplo escrito pelo tradutor, não fazendo parte do manual original.

[2] Exemplo escrito pelo tradutor, não fazendo parte do manual original.

[3] Exemplo escrito pelo tradutor, não fazendo parte do manual original, baseado em exemplo da lista de discussão pgsql-sql.

Exemplo do artigo do Kaui Aires em:
http://imasters.uol.com.br/artigo/5033/postgresql/auditoria_em_banco_de_dados_postgresql/imprimir/

```
/*
    OBJETIVO:    INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE BÁSICA DO SISTEMA.
    AUTOR:       KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS
     CONTATO: KAUIAIRES@GMAIL.COM
    CRIAÇÃO:     20/10/2006
    ATUALIZAÇÃO: 23/10/2006
    BANCO: POSTGRESQL

*/


--  PRÉ-REQUISITO:
--  --------------
--  Caso a linguagem plpgsql ainda não tenha sido definida em seu banco de  dados, DEVE ENTAO CONTINUAR COM TODO ESTE SCRIPT.
--  O terminal interativo do PostgreSQL (psql) poderá ser utilizado para essa finalidade, lembre-se que o usuário deverá possuir privilégio de "administrador" do banco de dados.
--  A função create_trigger(), assim como todas as demais funções criadas pela mesma, são escrita nessa "linguagem procedural".
--  A infra-estrutura é criada no esquema (espaço de tabela) corrente.

-- CRIA A LINGUAGEM PL/pgSQL
CREATE TRUSTED PROCEDURAL LANGUAGE plpgsql
    HANDLER plpgsql_call_handler;
COMMIT;

-- CRIA O USUARIO PARA AUDITORIA NO POSTGRES

-- Role: "auditoria_banco"

-- DROP ROLE auditoria_banco;

CREATE ROLE auditoria_banco LOGIN
  ENCRYPTED PASSWORD 'md58662fb9d00ee6acc21190bfce1f93fda'
  NOSUPERUSER INHERIT CREATEDB NOCREATEROLE;

-- OBS.: ESTE PASSWORD É:  "auditoria_banco" SEM ASPAS
COMMIT;

-- CRIA O SCHEMA NO BANCO DE AUDITORIA

CREATE SCHEMA auditoria_banco
  AUTHORIZATION auditoria_banco;
```

```
COMMENT ON SCHEMA auditoria_banco IS 'SCHEMA DE AUTIDORIA DO BANCO DE
DADOS POSTGRESQL - POR KAUI AIRES - KAUIAIRES@GMAIL.COM';
COMMIT;


-- REGISTRA A FUNÇÃO TRATADORA DE CHAMADAS
CREATE OR REPLACE FUNCTION "auditoria_banco"."plpgsql_call_handler"() RETURNS
language_handler
   AS '$libdir/plpgsql'
LANGUAGE C;

COMMENT ON FUNCTION "auditoria_banco"."plpgsql_call_handler"()
   IS 'Registra o tratador de chamadas associado linguagem plpgsql';
COMMIT;
ALTER FUNCTION "auditoria_banco"."plpgsql_call_handler"() OWNER TO auditoria_banco;
GRANT ALL ON SCHEMA auditoria_banco TO auditoria_banco WITH GRANT OPTION;
GRANT USAGE ON SCHEMA auditoria_banco TO public;
COMMIT;


-- FIM ESTRUTURA BÁSICA



/*
   OBJETIVO:    INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE
BÁSICA DO SISTEMA.
   AUTOR:       KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS
    CONTATO: KAUIAIRES@GMAIL.COM
   CRIAÇÃO:     20/10/2006
   ATUALIZAÇÃO: 23/10/2006
   BANCO: POSTGRESQL

*/


create table "auditoria_banco"."tab_dado_auditado"
(
   "nome_usuario_banco" varchar(200) not null,
   "data_hora_acesso" timestamp not null,
   "nome_tabela" varchar(200) not null,
   "operacao_tabela" varchar(20) not null,
   "ip_maquina_acesso" cidr not null,
   "sessao_usuario" varchar(255) not null
) with oids;


/* create indexes */
```

```
create index "idx_tab_dado_auditado_01" on "auditoria_banco"."tab_dado_auditado" using btree
("operacao_tabela");
create index "idx_tab_dado_auditado_02" on "auditoria_banco"."tab_dado_auditado" using btree
("ip_maquina_acesso");



/* create role permissions */
/* role permissions on tables */
grant select on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";
grant update on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";
grant delete on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";
grant insert on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";
grant references on "auditoria_banco"."tab_dado_auditado" to "auditoria_banco";

/* role permissions on views */

/* role permissions on procedures */



/* create comment on tables */
comment on table "auditoria_banco"."tab_dado_auditado" is 'tabela contendo os dados auditados';


/* create comment on columns */
comment on column "auditoria_banco"."tab_dado_auditado"."nome_usuario_banco" is 'nome do
usario do banco de dados que executou tal operacao';
comment on column "auditoria_banco"."tab_dado_auditado"."data_hora_acesso" is 'data de acesso';
comment on column "auditoria_banco"."tab_dado_auditado"."nome_tabela" is 'nome da tabela em
que foi realizado a alteracao';
comment on column "auditoria_banco"."tab_dado_auditado"."operacao_tabela" is 'operacoes
realizadas em tabelas tipo insert, update ou delete';
comment on column "auditoria_banco"."tab_dado_auditado"."ip_maquina_acesso" is 'ip maquina
acesso sistema';
comment on column "auditoria_banco"."tab_dado_auditado"."sessao_usuario" is 'sessao do
usuario';


/* create comment on domains and types */


/* create comment on indexes */
comment on index "auditoria_banco"."idx_tab_dado_auditado_01" is null;
comment on index "auditoria_banco"."idx_tab_dado_auditado_02" is null;

commit;
ALTER TABLE "auditoria_banco"."tab_dado_auditado" OWNER TO auditoria_banco;
```

```
GRANT INSERT ON TABLE auditoria_banco.tab_dado_auditado TO public;



/*
    OBJETIVO:    INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE BÁSICA DO SISTEMA.
    AUTOR:       KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS
     CONTATO: KAUIAIRES@GMAIL.COM
    CRIAÇÃO:     20/10/2006
    ATUALIZAÇÃO: 23/10/2006
    BANCO: POSTGRESQL

*/

-- SOMENTE PARA TESTE VAMOS CRIAR UMA TABELA "EMPREGADO" E CONFERIR NOSSA AUDITORIA

CREATE TABLE "public"."empregado" (
   "nome_empregado"    VARCHAR(150) NOT NULL,
   "salario"     integer
) with oids;

--FEITO ISTO VAMOS AO GATILHO



/*
    OBJETIVO:    INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE BÁSICA DO SISTEMA.
    AUTOR:       KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS
     CONTATO: KAUIAIRES@GMAIL.COM
    CRIAÇÃO:     20/10/2006
    ATUALIZAÇÃO: 23/10/2006
    BANCO: POSTGRESQL

*/
--Procedimento de gatilho PL/pgSQL para auditoria

-- Este gatilho garante que todas as inserções, atualizações e exclusões de linha na tabela e são registradas na tabela
-- tab_dado_auditado, para permitir auditar as operações efetuadas em uma tabela qualquer.
-- atualizar para o seu uso...
-- PARA O USO EM OUTRAS TABELAS  TROCAR ONDE TEM O PALAVRA "EMPREGADO"  SUGIRO TROCAR PELO NOME DA TABELA QUE SERÁ AUDITADA
    PARA FACILITAR DEPOIS A VISUALIZAÇAO

CREATE OR REPLACE FUNCTION auditoria_banco.processa_empregado_audit() RETURNS TRIGGER AS $empregado_audit$ -- AQUI TAMBÉM ALTERAR O NOME EMPREGADO
```

```
PARA O NOME CORRETO DA SUA FUNÇAO
   DECLARE
   -- PARAMETROS DE VERIFICAÇÃO E VARIAVEIS
    iQtd             integer;      -- valor de retorno
    auditar_delete integer = 1;       -- se voce deseja auditar DELETE = 1 se não deseja = 0
    auditar_update    integer = 1;       -- se voce deseja auditar UPDATE = 1 se não deseja = 0
    auditar_insert  integer = 1;       -- se voce deseja auditar INSERT = 1 se não deseja = 0
    SchemaName      text = 'public'; -- Nome do Schema que está a tabela  que Será Auditada
    TabName          text = 'empregado';   -- Nome da Tabela que Será Auditada

    BEGIN

        --
        -- VERIFICA A EXISTÊNCIA DA DEFINIÇÃO DA LINGUAGEM
        --
        SELECT count(*) FROM pg_catalog.pg_language  INTO iQtd
            WHERE lanname = 'plpgsql';
            IF iQtd = 0 THEN
                RAISE EXCEPTION 'NA AUDITORIA - A linguagem PLPGSQL ainda não foi
definida.';
            END IF;
       --
        -- VERIFICA A EXISTÊNCIA DA TABELA TAB_DADO_AUDITADO
        --
        SELECT count(*) FROM pg_catalog.pg_tables  INTO iQtd
            WHERE tablename = 'tab_dado_auditado';
            IF iQtd = 0 THEN
                RAISE EXCEPTION 'NA AUDITORIA - A tabela tab_dado_auditado não exite.';
            END IF;
        --
        -- VERIFICA SE PARÂMETROS SÃO DIFERNTES
        --
        IF SchemaName = TabName THEN
           RAISE EXCEPTION 'NA AUDITORIA - Deverá ser especificados parâmetros diferentes.';
        END IF;
        --
        -- VERIFICA A EXISTÊNCIA DA TABELA A SER AUDITADA
        --
       IF SchemaName IS NULL THEN
          SELECT count(*) FROM pg_tables INTO iQtd
          WHERE tablename = TabName;
       ELSE
          SELECT count(*) FROM pg_tables INTO iQtd
          WHERE schemaname = SchemaName
          AND tablename = TabName;
       END IF;
       IF iQtd = 0 THEN
            RAISE EXCEPTION 'NA AUDITORIA - A tabela % não exite.', SchemaName ||
```

```
TabName;
        END IF;
        --
        -- SE PASSAR POR TODOS OS TESTES ACIMA CRIA NOSSO GATILHO PARA
MONITORAR
        --
        --
        IF (TG_OP = 'DELETE') and auditar_delete = 1 THEN

            INSERT INTO auditoria_banco.tab_dado_auditado
                SELECT
                user, -- nome_usuario_banco
                now(), -- data_hora_acesso
                SchemaName || '.' || TabName, -- nome_tabela
                'DELETE', -- operacao_tabela
                inet_client_addr(), -- ip_maquina_acesso
                session_user; -- sessao_usuario
          RETURN OLD;
       ELSIF (TG_OP = 'UPDATE') and auditar_update = 1 THEN

            INSERT INTO auditoria_banco.tab_dado_auditado
                SELECT
                user, -- nome_usuario_banco
                now(), -- data_hora_acesso
                SchemaName || '.' || TabName, -- nome_tabela
                'UPDATE', -- operacao_tabela
                inet_client_addr(), -- ip_maquina_acesso
                session_user; -- sessao_usuario
          RETURN NEW;
       ELSIF (TG_OP = 'INSERT') and auditar_insert = 1 THEN

          INSERT INTO auditoria_banco.tab_dado_auditado
                SELECT
                user, -- nome_usuario_banco
                now(), -- data_hora_acesso
                SchemaName || '.' || TabName, -- nome_tabela
                'INSERT', -- operacao_tabela
                inet_client_addr(), -- ip_maquina_acesso
                session_user; -- sessao_usuario
          RETURN NEW;
       END IF;
       RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
    END;
$empregado_audit$ language plpgsql; -- AQUI TAMBÉM ALTERAR O NOME EMPREGADO
PARA O NOME CORRETO DA SUA FUNÇAO

CREATE TRIGGER empregado_audit -- AQUI TAMBÉM ALTERAR O NOME EMPREGADO
PARA O NOME CORRETO DA SUA FUNÇAO
```

```
AFTER INSERT OR UPDATE OR DELETE ON public.empregado -- COLOQUE AQUI
TAMBÉM O NOME DO SCHEMA E A TABELA QUE SERÁ AUDITADA SEPARADO
SOMENTE PELO PONTO
   FOR EACH ROW EXECUTE PROCEDURE auditoria_banco.processa_empregado_audit(); --
NOME DA FUNÇÃO A EXECUTAR NO LUGAR DO EMPREGADO COLOCAR O NOME DA
FUN ÇÃO
COMMIT;
ALTER FUNCTION auditoria_banco.processa_empregado_audit() OWNER TO auditoria_banco;
   -- AQUI TAMBÉM ALTERAR O NOME EMPREGADO PARA O NOME CORRETO DA SUA
FUNÇAO

-- FIM


/*
   OBJETIVO:    INSTALAR A LINGUAGEM PL/pgSQL NO BANCO DE DADOS E PARTE
BÁSICA DO SISTEMA.
   AUTOR:       KAUI AIRES OLIVEIRA - ARQUITETO DE BANCO DE DADOS
    CONTATO: KAUIAIRES@GMAIL.COM
   CRIAÇÃO:     20/10/2006
   ATUALIZAÇÃO: 23/10/2006
   BANCO: POSTGRESQL

*/
-- PARA TESTARMOS COMO FICOU NOSSA AUDITORIA FAREMOS ALGUNS TESTES
EXECUTE ESTE SCRIPT

INSERT INTO public.empregado (nome_empregado, salario) VALUES ('João',1000);
INSERT INTO public.empregado (nome_empregado, salario) VALUES ('José',1500);
INSERT INTO public.empregado (nome_empregado, salario) VALUES ('Maria',250);
UPDATE public.empregado SET salario = 2500 WHERE nome_empregado = 'Maria';
DELETE FROM public.empregado WHERE nome_empregado = 'João';

-- DEPOIS VEJA COMO FICOU NOSSA TABELA EMPREGADO

SELECT * FROM public.empregado;

 nome_empregado     salario
 -----------------  ----------
 José               1500
 Maria               2500

 2 record(s) selected [Fetch MetaData: 0/ms] [Fetch Data: 32/ms]



-- E A SUA TABELA DE AUDITORIA DEVE TER FICADO ASSIM
```

SELECT * FROM auditoria_banco.tab_dado_auditado;

nome_usuario_banco	data_hora_acesso	nome_tabela	operacao_tabela	ip_maquina_acesso	sessao_usuario
kaui_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200	kaui_aires
kaui_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200	kaui_aires
kaui_aires	24/10/2006 15:34	public.empregado	INSERT	172.25.0.200	kaui_aires
kaui_aires	24/10/2006 15:34	public.empregado	UPDATE	172.25.0.200	kaui_aires
kaui_aires	24/10/2006 15:34	public.empregado	DELETE	172.25.0.200	kaui_aires

5 record(s) selected [Fetch MetaData: 16/ms] [Fetch Data: 16/ms]

Mais um exemplo de monitoração com triggers:

De: Hesley Py

Neste artigo vamos criar um sistema de "log" no PostgreSQL. O sistema deve armazenar em uma tabela log todas as alterações (inserção, atualização e deleção), a data em que ocorreram e o autor da alteração na tabela "alterada".

Para o nosso exercício precisamos criar as seguintes tabelas no banco:
Alterada (cod serial primary key, valor varchar(50))

```
create table alterada(
        cod serial primary key,
        valor varchar(50)
);
```

Log (cod serial primary key, data date, autor varchar(20), alteracao varchar(6))

```
create table log(
      cod serial primary key,
      data date,
      autor varchar(20),
      alteracao varchar(6)
);
```

Como veremos a seguir, como um procedimento armazenado comum, um procedimento de gatilho no PostgreSQL também é tratado como uma função. A diferença está no tipo de dados que essa função deve retornar, o tipo especial trigger.

Primeiro vamos criar o nosso procedimento armazenado (function) que será executado quando o evento ocorrer. A sintaxe é a mesma já vista nos artigos anteriores, a direrença, como já dito, é o tipo de retorno trigger. Para ver mais detalhes execute o comanto \h create function no psql. A linguagem utilizada será a PL/pgSQL.

Nossa função de gatilho não deve receber nenhum parâmetro e deve retornar o tipo trigger.

```
create function gera_log() returns trigger as
$$
Begin
      insert into log (data, autor, alteracao) values (now(), user, TG_OP);
      return new;
end;
$$ language 'plpgsql';
```

Só pra lembrar, para criarmos uma função com a linguagem plpgsql, essa linguagem deve ter sido instalada em nosso banco através do comando "create language".

Quando uma função que retorna o tipo trigger é executada, o PostgreSQL cria algumas variáveis especiais na memória. Essas variáveis podem ser usadas no corpo das nossas funções. Na função acima estamos usando uma dessas variáveis, a TG_OP que retorna a operação que foi realizada (insert, update ou delete). Existem algumas outras variáveis muito

úteis como a new e a old. Minha intenção é abordá-las em um próximo artigo.

Para completar nosso exemplo, utilizamos as funções now e user para retornar data e hora da operação e o usuário logado respectivamente.

Agora precisamos criar o gatilho propriamente dito que liga a função à um evento ocorrido em uma tabela.

create trigger tr_gera_log after insert or update or delete on alterada
for each row execute procedure gera_log();

O comando acima cria um gatilho chamado tr_gera_log que deve ser executado após (after) as operações de insert, update ou delete na tabela "alterada". Para cada linha alterada deve ser executado o procedimento ou a função gera_log().

Para mais detalhes executar \h create trigger no psql.

Inserir registros na tabela e observar os resultados.

Mais Detalhes em:
http://www.postgresql.org/docs/current/static/plpgsql-trigger.html
http://pgdocptbr.sourceforge.net/pg80/plpgsql-trigger.html
http://imasters.uol.com.br/artigo/5033/postgresql/auditoria_em_banco_de_dados_postgresql/imprimir/
http://conteudo.imasters.com.br/5033/01.rar (arquivo exemplo)
http://www.devmedia.com.br/articles/viewcomp.asp?comp=7032

```
CREATE TRIGGER name {  BEFORE | AFTER }  {  event [OR ...] }
   ON table FOR EACH {  ROW | STATEMENT }
   EXECUTE PROCEDURE func ( arguments )

CREATE FUNCTION timetrigger() RETURNS opaque AS '
      BEGIN
           UPDATE shop SET sold="now" WHERE sold IS NULL;
           RETURN NEW;
      END;
' LANGUAGE 'plpgsql';

CREATE TRIGGER inserttime AFTER INSERT
      ON shop FOR EACH ROW EXECUTE
PROCEDURE timetrigger();

INSERT INTO shop (prodname,price,amount)
      VALUES('Knuth"s Biography',39,1);

SELECT * FROM shop ;
```

Variáveis criadas automaticamente:

NEW holds the new database row on INSERT/UPDATE statements on row-level triggers. The

datatype of NEW is record.

OLD contains the old database row on UPDATE/DELETE operations on row-level triggers. The datatype is record.

TG_NAME contains the name of the trigger that is actually fired (datatype name).

TG_WHEN contains either AFTER or BEFORE (see the syntax overview of triggers shown earlier).

TG_LEVEL tells whether the trigger is a ROW or a STATEMENT trigger (datatype text).

TG_OP tells which operation the current trigger has been fired for (INSERT, UPDATE, or DELETE; datatype text).

TG_RELID contains the object ID (datatype oid), and TG_RELNAME (datatype name) contains the name of the table the trigger is fired for.

TG_RELNAME contains the name of the table that caused the trigger invocation. The datatype of the return value is name.

TG_ARGV[] contains the arguments from the CREATE TRIGGER statement in an array of text.

TG_NARGS is used to store the number of arguments passed to the trigger in the CREATE TRIGGER statement. The arguments themselves are stored in an array called TG_ARGV[] (datatype array of text). TG_ARGV[] is indexed starting with 0.

RULES

O comando CREATE RULE cria uma regra aplicada à tabela ou view especificada.

Uma regra faz com que comandos adicionais sejam executados quando um determinado comando é executado em uma determinada tabela.

É importante perceber que a regra é, na realidade, um mecanismo de transformação de comando, ou uma macro de comando.

É possível criar a ilusão de uma view atualizável definindo regras ON INSERT, ON UPDATE e ON DELETE, ou qualquer subconjunto destas que seja suficiente para as finalidades desejadas, para substituir as ações de atualização na visão por atualizações apropriadas em outras tabelas.

Existe algo a ser lembrado quando se tenta utilizar rules condicionais para atualização de visões: é obrigatório haver uma rule incondicional INSTEAD para cada ação que se deseja permitir na visão. Se a rule for condicional, ou não for INSTEAD, então o sistema continuará a rejeitar as tentativas de realizar a ação de atualização, porque acha que poderá acabar tentando realizar a ação sobre a tabela fictícia da visão em alguns casos.

```
banco=# \h create rule
Comando:     CREATE RULE
Descrição:   define uma nova regra de reescrita
Sintaxe:
CREATE [ OR REPLACE ] RULE nome AS ON evento
    TO tabela [ WHERE condição ]
    DO [ ALSO | INSTEAD ] { NOTHING | comando | ( comando ; comando ... ) }
```

O comando CREATE RULE cria uma rule aplicada à tabela ou visão especificada.

evento
Evento é um entre SELECT, INSERT, UPDATE e DELETE.

condição
Qualquer expressão condicional SQL (retornando boolean). A expressão condicional não pode fazer referência a nenhuma tabela, exceto NEW e OLD, e não pode conter funções de agregação.

INSTEAD
INSTEAD indica que os comandos devem ser executados em vez dos (instead of) comandos originais.

ALSO
ALSO indica que os comandos devem ser executados adicionalmente aos comandos originais.

Se não for especificado nem ALSO nem INSTEAD, ALSO é o padrão.

comando
O comando ou comandos que compõem a ação da regra. Os comandos válidos são SELECT, INSERT, UPDATE, DELETE e NOTIFY.

Dentro da condição e do comando, os nomes especiais de tabela NEW e OLD podem ser usados para fazer referência aos valores na tabela referenciada. O NEW é válido nas regras ON INSERT e ON UPDATE, para fazer referência à nova linha sendo inserida ou atualizada. O OLD é válido nas regras ON UPDATE e ON DELETE, para fazer referência à linha existente sendo atualizada ou excluída.

Obs.: É necessário possuir o privilégio RULE na tabela para poder definir uma regra para a mesma.

Exemplos:
CREATE RULE me_notifique AS ON UPDATE TO datas DO ALSO NOTIFY datas;

CREATE RULE r1 AS ON INSERT TO TBL1 DO
(INSERT INTO TBL2 VALUES (new.i); NOTIFY TBL2);

CREATE RULE "_RETURN" AS ON SELECT TO minha_visão DO INSTEAD
SELECT * FROM minha_tabela; -- Ao invés de selecionar da visão seleciona da tabela.

Exemplo de uso prático de Rule

Existe uma tabela `shoelace_data que queremos monitorar e guardar os relatórios de alterações do camposl_avail na tabela shoelace_log. Para isso criaremos uma rule que gravará um registro na tabela shoelace_log sempre que o campo sl_avail for alterado em shoelace_data (vide estrutura abaixo).`

Say we want to trace changes to the `sl_avail` column in the `shoelace_data` relation. So we set up a log table and a rule that conditionally writes a log entry when an `UPDATE` is performed on `shoelace_data`.

```
CREATE TABLE shoelace_data (
    sl_name    text,          -- primary key
    sl_avail   integer,       -- available number of pairs
    sl_color   text,          -- shoelace color
    sl_len     real,          -- shoelace length
    sl_unit    text           -- length unit
);

CREATE TABLE shoelace_log (
    sl_name    text,          -- shoelace changed
    sl_avail   integer,       -- new available value
    log_who    text,          -- who did it
    log_when   timestamp      -- when
```

```
);

CREATE RULE log_shoelace AS ON UPDATE TO shoelace_data
    WHERE NEW.sl_avail <> OLD.sl_avail
    DO INSERT INTO shoelace_log VALUES (
                                    NEW.sl_name,
                                    NEW.sl_avail,
                                    current_user,
                                    current_timestamp
                                );
```

Vejamos o conteúdo da tabela shoelace_log:
SELECT * FROM shoelace_log;

Atualizemos a tabela shoelace_data alterando o campo sl_avail:
UPDATE shoelace_data SET sl_avail = 6 WHERE sl_name = 'sl7';

Vejamos novamente o que ocorreu na tabela shoelace_log:
SELECT * FROM shoelace_log;

```
 sl_name | sl_avail | log_who | log_when
---------+----------+---------+----------------------------------
 sl7     |        6 | Al      | Tue Oct 20 16:14:45 1998 MET DST
(1 row)
```

Veja que agora ela armazena as informações que indicamos.

Crie um exemplo similar para uso de rules.

Derrubar usuário Conectado no PostgreSQL
No Linux e no Windows

No Linux

Para retornar o PID dos usuários e suas respectivas consultas execute aconsulta:

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
     pg_stat_get_backend_activity(s.backendid) AS current_query
   FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Cuidado, evite usar o comando "kill -9 PID" para derrubar um usuário do PostgreSQL.

"-Nunca- faça isso. Se você executar um 'kill -9 XXXX' você irá derrubar a conexão problemática e todas as outras ativas no momento. Isso porque todas as conexões compartilhar uma área de memória comum (aka shared buffers) e, se você a corrompe o PostgreSQL tem que garantir que os dados ali estejam consistentes.
Se você quer cancelar uma consulta demorada utilize pg_cancel_backend()
[1]. Existia uma função pg_terminate_backend() (não me recordo se ele chegou a estar em alguma versão do 8.0) mas ela se tornou obsoleta por motivos de confiabilidade. Esta mesma função utiliza 'kill -TERM XXXX' para derrubar os processos do postgres. Esta seria a maneira "correta" de terminar os processos postgres mas ...
O que aconselho fazer é:

(i) pg_cancel_backend(pid)

(ii) kill -TERM pid

ou

(i) pg_cancel_backend(pid)
(ii) pg_ctl kill TERM pid

[1] http://www.postgresql.org/docs/8.3/static/functions-admin.html"

Dica do Euler na lista pgbr-geral

No Windows:

Baixar o utilitário kill for windows de http://www.mattkruse.com/utilities/kill-tlist.zip
Descompactar e copiar o KILL.EXE para c:\windows

Teclar Ctrl+Alt+Del
Clicar em Exibir
Selecionar Colunas
Selecionar PID (Identificador de processos)

Abrir um terminal e executar o psql como superusuário:
psql -U postgres

Abrir um segundo terminal com um usuário comun, que tenha privilégio de login:

psql -U dba1 clientes

```
SELECT pg_stat_get_backend_pid(s.backendid) AS pid,
      pg_stat_get_backend_activity(s.backendid) AS current_query
   FROM (SELECT pg_stat_get_backend_idset() AS backendid) AS s;
```

Anotar o pid.

No primeiro terminal, como superusuário, sem sair do psql, executar:

\!kill -f pid

Derrubará o usuário dba1 somente.