

Artigo

Invista em você! Saiba como a DevMedia pode ajudar sua carreira.

PostgreSQL Stored Procedures: Funções em Linguagem SQL

Artigo da Revista SQL Magazine.



Anotar



Marcar como concluído

Artigos



Banco de Dados



PostgreSQL Stored Procedures: Funções em Linguagem SQL

Atenção: por essa edição ser muito antiga não há arquivo PDF para download.

Os artigos dessa edição estão disponíveis somente através do formato HTML.



[Clique aqui para ler todos os artigos desta edição](#)

Em muitos SGDBs temos o conceito de *Stored Procedures*, programas desenvolvidos em uma determinada linguagem de *script* e armazenados no servidor, onde serão processados. No PostgreSQL, as *Stored Procedures* são conhecidas com o nome de *Functions*.

Tipos de funções

No **PostgreSQL** podemos ter três tipos de funções:

- **Funções em Linguagem SQL:** As funções em SQL não possuem variáveis e estruturas de comando (*if*, *for* etc). Elas apenas consistem em uma lista de comandos SQL (SELECT, INSERT, DELETE ou UPDATE), devendo retornar, obrigatoriamente, um determinado valor. Assim, o último comando deve ser sempre um SELECT. Essas funções são carregadas juntamente com o serviço do PostgreSQL, não necessitando de nenhuma carga de módulo adicional.
- **Funções de Linguagens Procedurais:** Esse tipo de função utiliza variáveis e estruturas de comandos, além de executar ações SQL. Na versão atual do PostgreSQL temos quatro tipos de linguagens procedurais: PL/PgSQL, PL/Tcl, PL/Perl e PL/Python. A Linguagem PL/PgSQL é a mais utilizada, pois é bem estruturada e fácil de aprender. As linguagens PL/Tcl, PL/Perl e PL/Python têm sintaxe semelhante às linguagens das quais elas herdam sua implementação. Sua utilização será explorada nas próximas edições da SQL.

Magazine.

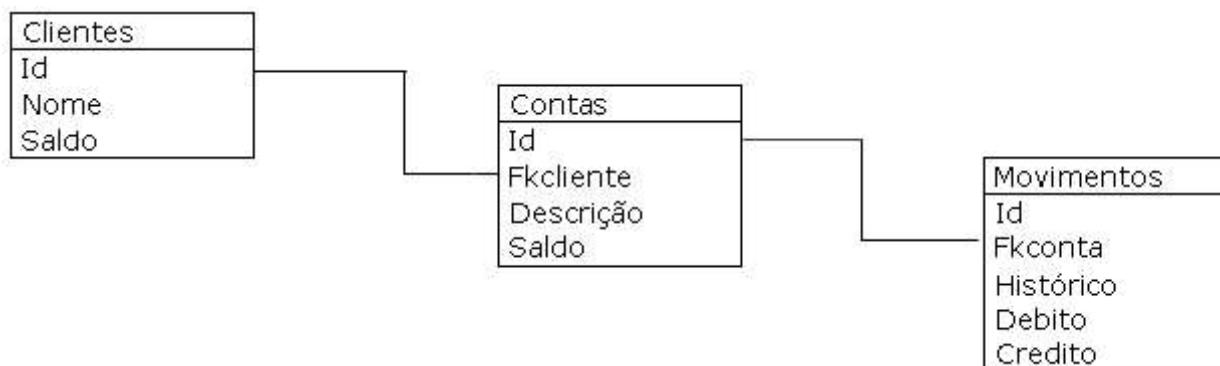
Nota: A linguagem PL/PgSQL é semelhante a linguagem PL/SQL, do Oracle.

- **Funções Externas:** No PostgreSQL podemos utilizar funções desenvolvidas em uma linguagem externa, como C++. A vantagem é que passamos a contar com o poder de uma linguagem de programação completa, possibilitando a implementação de rotinas complexas no banco de dados. As funções devem ser empacotadas em bibliotecas compatilhadas que, por sua vez, devem ser registradas no SGBD.

Este artigo apresenta as funções em SQL. Os demais tipos de função serão temas para as próximas edições da revista.

Desenvolvendo Funções em SQL

Tomaremos como exemplo o modelo de dados da figura a seguir. Os exemplos aqui descritos foram executados no Linux.



Para criação do modelo, execute os procedimentos:

1. Crie um banco de dados chamado *sqlmagazine* com o comando *createdb*:

```
1 | createdb -U postgres sqlmagazine
```

2. Abra o banco de dados, com o comando:

3. No *psql*, crie os objetos do modelo e insira alguns registros nas tabelas. A listagem completa para este procedimento está disponível para download no site deste artigo.

O Comando Create Function

Para funções implementadas em SQL, o comando *create function* tem a seguinte estrutura:

```
1 | CREATE [OR REPLACE] FUNCTION NomeDaFuncao([parâmetro 1, parâmetro 2,  
2 |           RETURNS RetornoTipoDeDados AS '  
3 |               Corpo da Função;  
4 |           '  
5 |           LANGUAGE 'SQL';
```

Onde,

- **CREATE FUNCTION** - Define o nome da função e seus respectivos parâmetros, caso existam. Esses parâmetros são identificados internamente como \$1 (Parâmetro 1), \$2 (Parâmetro 2), \$3 (Parâmetro 3), e assim por diante.
- **RETURNS RetornoTipoDeDados** - Indica o tipo de dado de retorno da função. Uma função pode retornar um tipo simples como *integer*, *varchar* etc. Funções em SQL também podem retornar um conjunto de valores ou uma estrutura composta de várias linhas (*resultset*).
- **Corpo da Função** - Contém a implementação da função e deve estar entre aspas simples.
- **LANGUAGE 'SQL'** - indica que a linguagem utilizada para implementação da função é SQL (se estivéssemos utilizando a linguagem PL/PgSQL, usariámos **LANGUAGE 'PL/pgsql'**).

NOTA: A função é de “propriedade” do usuário que a criou e, para ser acessada por outro usuário do banco, é necessário que este possua o *grant* de EXECUTE na mesma. Esse procedimento pode ser efetuado com o comando GRANT EXECUTE ON *nomedafuncao* TO GROUP *nomedousuario*.

Vejamos um exemplo de criação de uma função SQL. Esta deve ser digitada em um editor de textos e gravada, por exemplo, em */root*, com o nome de *funcao01.sql*. Esta função recebe como parâmetro um único valor do tipo inteiro (identificado como *\$1*), e também retorna um valor do tipo inteiro. O valor de retorno é definido pela execução do último comando da rotina (neste exemplo, o único comando existente):

```
1 CREATE FUNCTION incrementar(INTEGER)
2 RETURNS INTEGER AS '
3     SELECT $1 + 1 ;
4 '
5 LANGUAGE 'SQL' ;
```

Para carregar a função no banco de dados, execute no *psql* o comando: *|i /root/funcao01.sql*. Esse comando carrega e executa um arquivo texto no *psql*. Nesse caso, não é possível digitar a rotina direto no *prompt* do *psql* pois o ‘;’ é o terminador de linha deste aplicativo.

Para executar a função, devemos utilizar o comando *SELECT*. Veja o exemplo:

```
1 sqlmagazine=# SELECT incrementar(10);
2
3     incrementar
4 -----
5         11
6      (1 row)
7
sqlmagazine=#
```

O próximo exemplo retorna o número de contas que um determinado cliente possui. O *id* do cliente deverá ser passado como parâmetro:

```
1 | CREATE FUNCTION ncontas(INTEGER)
2 |   RETURNS INT8 AS '
3 |     SELECT COUNT(*) FROM contas
4 |       WHERE fkcliente = $1;
5 |
6 | LANGUAGE 'SQL';
```

Após carregar a função no banco de dados, utilize o comando SELECT para executá-la:

```
1 | SELECT ncontas(2);
```

O exemplo a seguir insere dados referentes ao cliente e sua conta, retornando o id do cliente inserido:

```
1 | CREATE FUNCTION cliente_contadesc(VARCHAR(30), VARCHAR(30))
2 |   RETURNS INT8 AS '
3 |     INSERT INTO clientes(nome) VALUES($1);
4 |     INSERT INTO contas(fkcliente, descricao)
5 |       VALUES(CURRVAL(''clientes_id_seq''),$2);
6 |     SELECT CURRVAL(''clientes_id_seq'');
7 |
8 | LANGUAGE 'SQL';
```

Observe que o código principal é implementado entre aspas simples e o parâmetro da função currval(), que é um *string*, tem de ser passado entre duas aspas simples. O último comando retorna o id do cliente que acabou de ser inserido.

Após carregar a função no banco de dados, utilize o comando a seguir para

```
1 | SELECT cliente_contadesc('SILVIO', 'SEMANA04');
```

NOTA: Para apagar uma função, utilize o comando: drop function <nome_da_funcao()>;</nome_da_funcao()>

Retornando um conjunto de registros e campos

Em funções SQL podemos retornar um conjunto de linhas; para isso, devemos acrescentar o parâmetro SETOF antes do tipo de dado a ser retornado. Como exemplo, criaremos agora uma função para retornar o id dos clientes com saldo negativo.

```
1 | CREATE FUNCTION quemdeve() RETURNS SETOF INTEGER AS '
2 |     SELECT clientes.id FROM clientes
3 |     INNER JOIN contas ON clientes.id = contas.fkcliente
4 |     INNER JOIN movimentos ON contas.id = movimentos_fkcont
5 |     GROUP BY clientes.id
6 |     HAVING SUM(movimentos.credito - movimentos.debito) < €
7 |
8 | LANGUAGE 'SQL';
```

Utilize o comando SELECT quemdeve() para executar a função. O resultado exibido é o seguinte:

```
1 | quemdeve  -----
2 |          1
3 |          2
4 | (2 rows)
```

Caso seja necessário retornar todas as colunas da tabela *CLIENTES*, o tipo de dados de retorno da função deve referenciar a tabela ou uma *view* que possua

estão disponíveis somente a partir da *Versão 7.3 do PostgreSQL*. Veja um exemplo:

```

1 CREATE FUNCTION devedores()
2     RETURNS SETOF clientes AS '
3         SELECT * FROM clientes WHERE id IN
4             (
5                 SELECT clientes.id FROM clientes
6                     INNER JOIN contas ON clientes.id = contas.fk
7                     INNER JOIN movimentos ON contas.id = movimentos.id
8                     GROUP BY clientes.id
9                     HAVING SUM(movimentos.credito) - movimentos.c
10                );
11
12 LANGUAGE 'SQL';

```

Uma função que retorna um *resultset* deve ser chamada de forma semelhante a qualquer tabela do banco de dados, ou seja, na cláusula *FROM*. Veja um exemplo:

```

1 select id, nome from devedores();   id | nome
2 -----+-----
3       1 | MARIA
4       2 | JOSE
5      (2 rows)

```

O exemplo a seguir retorna os dados dos clientes que possuem o movimento crédito de suas contas maior ou igual ao valor passado como parâmetro:

```

CREATE FUNCTION MaioresClientes(NUMERIC(15,2))
    RETURNS SETOF clientes AS '
        SELECT * FROM clientes WHERE id IN
            (
                SELECT clientes.id FROM clientes
                    INNER JOIN contas ON clientes.id = contas.fk
                    INNER JOIN movimentos ON contas.id = movimentos.id
                    WHERE movimentos.credito >=

```

```
11 |     );
12 | LANGUAGE 'SQL';
```

Nesse caso, se quisermos a listagem dos clientes que movimentaram mais de R\$ 10.000,00, podemos executar o comando:

```
1 | select id, nome from MaioresClientes (10000);
```

Onde se encontram as *functions*?

A tabela de sistema pg_proc armazena todas as *functions* que foram criadas. Para visualizá-las, utilize a seguinte *query*:

```
1 | SELECT proname, prosrc FROM pg_proc WHERE proname = 'quemdeve';
```

NOTA: Todas as tabelas de sistema do PostgreSQL são iniciadas por ‘pg’

Conclusão

Funções em Linguagem SQL constituem-se de *Queries* que são armazenadas no servidor, podendo ou não receber parâmetros. Elas podem retornar valores ou conjuntos de registros.

O uso de funções SQL torna-se interessante devido à simplicidade de sua implementação. Caso o processamento a ser implementado seja mais complexo, a ponto de requerer a utilização de estruturas de controle, condicionais, variáveis etc, a utilização de PL/PGSQL ou até do C é mais recomendada. Na próxima edição abordaremos mais detalhes sobre esse assunto.

em um único artigo. Bons estudos :)

PostgreSQL Stored Procedures: Funções e Triggers - Parte 2



[Clique aqui para ler todos os artigos desta edição](#)

A primeira parte deste artigo definiu o conceito de funções (utilização e tipos) e abordou a criação e o uso de funções do tipo SQL. Este artigo trata de outro tipo de função que pode ser implementada no PostgreSQL, mas conhecida como função de linguagens procedurais. Essas funções, além de executarem comandos SQL, utilizam recursos de variáveis e estruturas de controle fundamentais à qualquer linguagem procedural.

Na versão atual do PostgreSQL, temos 4 tipos de linguagens procedurais:

PL/pgsql, PL/tcl, PL/perl e PL/python. Ao contrário das funções em SQL, é necessário carregar a linguagem de desenvolvimento no banco de dados como um módulo. Esse procedimento também será descrito no decorrer deste artigo.

A sintaxe da PL/pgsql é muito semelhante à da PL/SQL do Oracle. As linguagens

herdam, e são pouco utilizadas. Em função disso, abordaremos a sintaxe de funções no PostgreSQL por meio da linguagem procedural PL/PgSQL. Assim como no artigo anterior, todos os procedimentos foram executados em ambiente Linux.

Processo de criação de Stored Procedures em PL/PgSQL

No PostgreSQL, as linguagens procedurais não funcionam como as funções em SQL, onde podemos criar diretamente uma função e, em seguida, executá-la no banco de dados. Para criarmos uma stored procedure em uma linguagem procedural PL/PgSQL, devemos primeiro carregar no banco de dados o módulo da linguagem a ser utilizada para desenvolver a função. Essa regra se aplica também a outras linguagens procedurais, como PL/Tcl, PL/Perl e PL/Python.

Vamos desenvolver nossas funções em PL/PgSQL no banco `sqlmagazine` criado no artigo anterior. Utilizaremos o comando `createlang` para carregar a linguagem PL/PgSQL nesse banco de dados. No prompt do Linux, execute o comando abaixo:

```
1 | createlang -U postgres plpgsql sqlmagazine
```

Com este comando, carregaremos o módulo da linguagem procedural PL/PgSQL somente no banco `sqlmagazine`. O PostgreSQL possui um banco padrão denominado `template1`. Todos os databases que criamos copiam a estrutura desse banco padrão, portanto, se quisermos que a linguagem PL/PgSQL esteja disponível em todos os bancos no servidor, devemos executar o seguinte comando:

```
1 | createlang -U postgres plpgsql template1
```

Estrutura de uma Função em PL/PGSQL

A **Listagem 1** mostra a sintaxe geral para a criação de uma função em PL/PgSQL:

Listagem 1 - Sintaxe para a criação de funções

```
1 CREATE [OR REPLACE] FUNCTION nome_da_procedure(parametro1,  
2         parametro2, ..., parametroN)  
3             RETURNS [SETOF] tipo_dado_retornado AS '  
4             DECLARE  
5                 variaveis;  
6             BEGIN  
7                 algoritmo_da_procedure;  
8             END;  
9             ' LANGUAGE 'PLPGSQL';
```

Observando a **Listagem 1** temos:

O comando CREATE FUNCTION: cria uma função. Para alterar ou sobrescrever uma função já existente, devemos utilizar CREATE OR REPLACE FUNCTION.

Em nome_da_procedure, temos o nome da procedure a ser criada. Aqui poderíamos mencionar o nome do Schema do qual a função faria parte. Ao omitirmos o nome do Schema, a procedure será criada sobre um Schema padrão denominado public.

Em (parametro1, parametro2,..., parametroN), caso tenham sido declarados, definimos o tipo de dado dos parâmetros de entrada da função. Esses parâmetros serão acessados na implementação da procedure como \$1 (Parâmetro 1), \$2 (Parâmetro 2) , ..., \$N (Parâmetro N). Os tipos de dados dos parâmetros podem ser: base (como int, int2, int4, int8, char, varchar) ou colunas (referenciados como nome_da_tabela.coluna%type). Desse modo, o tipo de dado do parâmetro de entrada será definido e estará vinculado ao tipo de dado do campo indicado.

Tanto a definição dos parâmetros como o acesso a eles funcionam da mesma forma que em funções SQL.

Em tipo_de_dados_retornado mencionamos o tipo de dado de retorno da procedure, que poderá ser base ou coluna. Toda procedure deverá retornar uma informação. O modificador SETOF indica que a procedure retornará um conjunto de linhas em vez de uma só linha.

Na linha 2 usamos o comando declare para a declaração de variáveis. As variáveis poderão ser do tipo alias, base, colunas ou record. Vale a pena destacarmos aqui alguns conceitos referentes à **declaração de variáveis**. Observe os exemplos a seguir:

```
1 | declare
2 |     idvenda ALIAS FOR $1;
3 |     idcomprador INT4;
4 |     idusuario usuarios.id%TYPE;
5 |     r RECORD;
6 |     rec usuarios%ROWTYPE;
```

No exemplo acima, usamos o comando declare com os seguintes valores:

```
1 | idvenda ALIAS FOR $1;
```

Com esta declaração, definimos a variável idvenda como um apelido do parâmetro \$1 passado para a procedure. Usamos esse artifício para poder substituir a string \$1 no algoritmo principal de nossa procedure por um nome de fácil compreensão.

```
1 | idcomprador INT4;
```

Entendendo a função de uma variável declarada: quando é útil usar a declaração

```
1 | idusuario usuarios.id%TYPE;
```

Definimos a variável idusuario de acordo com o tipo de dados da coluna id da tabela usuarios. Dessa forma, teremos um vínculo entre ambos. Com isso, quando o tipo de dados da coluna id na tabela usuarios for alterado, o tipo de dados da variável idusuario será automaticamente alterado também.

```
1 | r RECORD;
```

Aqui definimos que r será uma variável do tipo record . Esse tipo de variável permite trabalhar com os registros e campos de determinada tabela em estruturas de controle.

```
1 | rec usuarios%ROWTYPE;
```

As variáveis %ROWTYPE definem o tipo de dados de uma variável de acordo com o tipo de dados de uma linha inteira de determinada tabela. Definimos a variável rec como um tipo de dados de acordo com toda a linha da tabela usuarios. Dessa forma, teremos um vínculo entre ambos e, quando um tipo de dados de qualquer uma das colunas da tabela usuarios for alterado, os valores de rec serão alterados de acordo.

Após a DECLARE, temos a seção de implementação da procedure, delimitada por BEGIN e END. É importante ressaltar que BEGIN e END não têm nenhum vínculo com os conceitos de transação do PostgreSQL.

Por fim, temos a declaração da linguagem utilizada para criar a procedure. Nesse caso, utilizamos a LANGUAGE ‘PLPGSQL’.

Elementos da Linguagem PL /

PgSQL

1. Estruturas Condicionais

Assim como em outras linguagens, os testes lógicos são efetuados com a estrutura condicional IF-THEN-ELSE. A expressão booleana pode comparar valores de diversos tipos de dados e pode ter expressões select em sua composição. No geral temos:

```
1 | IF expressão-booleana THEN
2 |   comandos
3 | ELSE
4 |   comandos
5 | END IF;
```

Como exemplo, tendo V_COUNT como um número inteiro, temos:

```
1 | IF v_count > 0 THEN
2 |   INSERT INTO users_count(count)
3 |   VALUES(v_count);
4 |   return 'v';
5 | ELSE
6 |   return 'f';
7 | END IF;
```

2. Estruturas de repetição e laços condicionais

Para estruturas que implementam laços ou loops temos o LOOP-END LOOP, FOREND LOOP, WHILE-END LOOP e o que chamamos de QUERY-LOOP. Seguem as sintaxes:

```
1 | LOOP-END LOOP
2 |
~ |   loop
```



```
    | END LOOP  
    ;
```

Para ‘encerrar’ ou sair de um loop, utilizamos o comando exit. Esse comando pode ser utilizando para interromper o loop em todas as estruturas aqui descritas.

FOR-END LOOP

```
1 | FOR variavel IN [ valores ] expressão  
2 |     .. LOOP  
3 |     Comandos  
4 |     END LOOP;
```

O exemplo abaixo faz um loop com a variável i, alterna seu valor entre 1 e 10 e retorna à aplicação cliente o valor da variável no loop por meio da função RAISE NOTICE. Esta função retorna uma informação qualquer ao client (como, por exemplo, um printf do C). Sua sintaxe é RAISENOTICE mensagem, variavel. O conteúdo de variável pode ser utilizado na mensagem e ser substituído automaticamente pelo Postgre quando ele encontrar o símbolo % na mensagem.

Segue o exemplo:

```
1 | FOR i IN 1..10 LOOP  
2 |     Comandos  
3 |  
4 |     RAISE NOTICE 'i vale %',i;  
5 |     END LOOP;
```

```
1 | WHILE expressão LOOP  
2 |     Comandos  
3 |     END LOOP;
```

OUTRA LIGAÇÃO

DEVMEDIA



Você pode implementar um laço condicional diretamente com o resultado de um select, algo como o “**While Not DataSet.Eof Do**” do Delphi. Para isso, é possível utilizar variáveis do tipo Record ou %ROWTYPE. A sintaxe geral dessa estrutura é a seguinte:

```
1 | FOR variavel_record_ou_%rowtype IN select
2 |   LOOP
3 |   Comandos
4 | END LOOP;
```

Segue um exemplo da utilização dessa estrutura de repetição, na qual os dados de uma tabela são inseridos em outra tabela que possui a mesma estrutura:

```
1 | DECLARE
2 |   registro RECORD;
3 | BEGIN
4 |   Comandos;
5 |   FOR registro IN SELECT * FROM TABELA1
6 |     ORDER BY ID LOOP
7 |     INSERT INTO TABELA2(ID, DESCRICAO)
8 |       VALUES (registro.ID, registro.DESCRICAO);
9 |   END LOOP;
10 |  Comandos;
11 | END;
```

3. Executando QUERYS em expressão texto.

O PL/PgSQL possui um comando chamado EXECUTE. Este comando executa uma instrução SQL em formato texto. A sintaxe geral é

```
1 | EXECUTE 'expressão SELECT';
```

A **Listagem 2** mostra a implementação de uma função que executa um select

dinâmico na tabela passada como parâmetro. Neste exemplo, devemos digitar

composição do comando ‘SELECT * FROM’ com o nome da tabela passada como parâmetro).

Listagem 2 - Sintaxe para a criação de uma função que executa um Select dinâmico

```

1 CREATE FUNCTION select_dinamico (text) RETURNS INTEGER AS'
2   DECLARE
3     vtabla alias for $1;
4     vSQL text;
5   BEGIN
6     vSQL := 'SELECT * FROM ' || vtabla;
7     EXECUTE vSQL;
8     RETURN 1;
9   END;
10  ' LANGUAGE 'plpgsql';

```

O EXECUTE também pode ser utilizado nos QUERY-LOOPS exemplificados anteriormente. A sintaxe geral para sua utilização é a seguinte:

```

1 FOR variavel_record_ou_%rowtype IN EXECUTE
2   STRING LOOP
3     Comandos
4   END LOOP;

```

Desenvolvendo em PL/PgSQL

Começaremos pelo desenvolvimento de uma Function simples, na qual passaremos como parâmetro o id de um cliente. Se o cliente existir, será retornado o seu nome; caso contrário, será retornado uma mensagem do tipo “Cliente id X não existe”. Utilizando um editor de textos simples, edite a Function da Listagem 3 e salve-a em /root com o nome de proc01.sql.

```
1 -- function proc01.sql
2 --
3     CREATE OR REPLACE FUNCTION id_nome_cliente( INTEGER ) RETURNS TEX
4     DECLARE
5         r RECORD;
6     BEGIN
7         SELECT INTO r * FROM clientes WHERE id = $1;
8         IF NOT FOUNT THEN
9             RAISE EXCEPTION ''Cliente % não existente !'', $1;
10        END IF;
11        RETURN r.nome;
12    END;
13
14 LANGUAGE 'PLPGSQL';
```

Nota: No PL/PgSQL, os comentários das linhas em SQL iniciam com `--`, e os comentários dos blocos de linhas estarão entre `/* string comentário */`.

A Function criada na Listagem 3 utiliza um record para armazenar o conteúdo do registro de retorno, e o modificador NOT FOUNT retorna um boolean que indica se o último SELECT executado retornou registros. O RAISE EXCEPTION funciona de forma semelhante ao RAISE NOTICE explicado anteriormente, porém gera uma exceção e aborta a execução da Function. Para carregar essa Function no banco de dados, execute os seguintes procedimentos:

1. No prompt do Linux, abra o banco de dados com o seguinte comando:

```
1 | psql -U postgres sqlmagazine
```

2. O banco `sqlmagazine` será aberto. Carregue a Function com o comando `\iname_procedure.sql`. A Listagem 4 mostra o prompt de inicialização do banco e a execução do comando de carga da função criada.

Listagem 4 - Prompt de inicialização e carregamento da função implementada



```
1 Welcome to psql 7.3.3, the PostgreSQL interactive terminal.  
2  
3     Type: \copyright for distribution terms  
4     \h for help with SQL commands  
5     \? for help on internal slash commands  
6     \g or terminate with semicolon to execute query  
7     \q to quit  
8  
9     sqlmagazine=#\i proc01.sql  
10    CREATE FUNCTION  
11    sqlmagazine=#
```

3. Para executar a Function, use o comando `select nome_da_procedure()`. A Listagem 5 mostra a execução da procedure e o resultado de seu processamento para um cliente cadastrado e para um cliente não cadastrado.

Listagem 5 - Execução da procedure criada na Listagem 4

```
1 sqlmagazine=# SELECT id_nome_cliente(2);  
2      id_nome_cliente  
3      -----  
4      JOSE  
5      (1 row)  
6  
7      sqlmagazine=# SELECT id_nome_cliente(2000);  
8      ERROR: Cliente 2000 não existente !  
9      sqlmagazine=#
```

Os procedimentos aqui descritos podem ser utilizados para criação, carga e execução de qualquer Function no PostgreSQL. Para consultar as procedures criadas no seu banco de dados, execute um `SELECT` sobre a tabela de sistema que armazena todas as procedures. Esse select retornará inclusive o corpo da implementação da procedure criada. Consulte a **Listagem 6**.

Listagem 6 - Verificando as procedures criadas no banco de dados



```

1  sqlmagazine=# SELECT proname, prosrc FROM pg_proc WHERE proname = 'i
2
3      proname    |      prosrc
4
5  -----+-----+
6  id_nome_cliente | DECLARE
7  r RECORD;
8  BEGIN
9  SELECT INTO r * FROM clientes WHERE id = $1;
10 IF NOT FOUND THEN
11 RAISE EXCEPTION 'Cliente % não existente !', $1;
12 END IF;
13 RETURN r.nome;
14 END;
15
16 (1 row)
17
18 sqlmagazine=#

```

Triggers

A implementação da Trigger é feita por meio da criação de um objeto que envia uma chamada para uma função, que por sua vez executará a ação. Por exemplo, a **Listagem 7** implementa a função ftr_ins_ teste() que será chamada por uma trigger. Isso permite que as triggers no PostgreSQL sejam implementadas em diversas linguagens de desenvolvimento, como JAVA, PERL, C, TCL etc.

Listagem 7 - Criação de uma Function para implementação de uma Trigger

```

Create table teste(id int4, nome text);
Create table teste2(id_teste int4, nome text);

1  create or replace Function ftr_ins_teste() returns trigger
2  Begin

```

```
9      6      return new;
10     7      end;
11     8      'Language 'plpgsql';
```

Para executar o exemplo da Listagem 7, crie as seguintes tabelas:

Vamos analisar a implementação da Function:

Linha 1: create or replace Function ftr_ins_geuf() returns trigger as'

Observamos que o tipo de retorno aqui é diferente: a Function a ser utilizada numa Trigger tem um tipo de retorno específico, definido como trigger.

Linha 2: Begin Inicialização do código de implementação da Function, mas poderíamos ter um declare exatamente igual a uma função, pois a linguagem é PL/PgSQL.

Linha 3: if new.id is not null then O modificador new possibilita o acesso ao novo valor atribuído às colunas da tabela que disparou a trigger.

Linha 4: insert into teste2(id_teste, nome) values(new.id, new.nome);

Insere o registro da tabela que disparou a trigger na Tabela 2, caso a coluna id não seja nula (condição testada na linha 3).

Linha 6: return new; No caso de Functions que serão utilizadas em triggers de insert, o result recebe sempre o modificador NEW. No caso de triggers de Update, o result pode ser NEW ou OLD (para deletes, apenas OLD).

Nota: os modificadores NEW e OLD são variáveis do tipo RECORD.

Uma vez criada a função PL/PgSQL com os procedimentos citados anteriormente, passaremos então à criação da trigger que chamará a função. Use o comando a seguir:

```
1 | Create trigger tr_ins_teste after insert
2 |   on teste for each row execute procedure
3 |     ftr_ins_teste();
```

Neste comando, temos:

Create trigger tr_ins_teste - o comando create da trigger. A sintaxe genérica é
CREATE TRIGGER NOME_TRIGGER;

after insert - esta é uma parte interessante da criação. Estamos definindo aqui que a ação será realizada **depois de ocorrer o insert na tabela teste**. Os eventos (momentos de execução da função) possíveis para a uma trigger são:

- AFTER INSERT ON TABELA
- BEFORE INSERT ON TABELA
- AFTER UPDATE ON TABELA
- BEFORE UPDATE ON TABELA
- AFTER DELETE ON TABELA
- BEFORE DELETE ON TABELA

on teste - dizemos aqui que a *trigger* está vinculada à tabela teste.

for each row execute procedure ftr_ins_teste() - aqui é feita a chamada da função para a *trigger* que criamos, ftr_ins_teste().

Conclusão

Neste artigo, mostro os elementos de sintaxe da linguagem PL/PgSQL e os procedimentos de criação, alteração e carga no banco de dados de uma Function. O uso dessa linguagem alarga o horizonte do desenvolvedor e disponibiliza recursos e estruturas não encontradas nas funções implementadas em C. Fizemos

série, vamos implementar as funções por meio da linguagem C e explorar seus recursos e utilizações. Até lá.

PARTE III

Veja abaixo a terceira e última parte do artigo - **Agora as partes I, II e III foram compiladas em um único artigo. Bons estudos :)**

Desenvolvimento de funções (Stored Procedures) em PostgreSQL: Funções em C - Parte 3

Atenção: por essa edição ser muito antiga não há arquivo PDF para download. Os artigos dessa edição estão disponíveis somente através do formato HTML.



[Clique aqui para ler todos os artigos desta edição](#)

Os dois primeiros artigos dessa série abordaram os fundamentos para desenvolvimento e utilização de funções no PostgreSQL utilizando SQL e PL/PGSQL. Essas duas linguagens oferecem ao desenvolvedor PostgreSQL recursos para construir sistemas complexos com finalidades diversas. Mas para customizações que exigem um nível maior de complexidade e recursos - não disponíveis na linguagem SQL ou PL/PGSQL – o PostgreSQL permite a utilização do C.

Mas porque utilizar C? Como vimos nos artigos anteriores, o PostgreSQL trabalha

com muitas linguagens TATA TOT DEPT DIVISION etc. Tais bancos matina de

a limitação do PL/PGSQL em nível de programação. Enquanto o PL/PGSQL é uma linguagem interpretada, o C é compilado.

Primeiros passos em desenvolvimento de funções em C

A implementação de funções em C a serem utilizadas pelo PostgreSQL consiste basicamente de três passos:

1. Implementação da função

A *Listagem 1* traz um exemplo clássico de uma função que recebe como argumento um número inteiro e retorna o dobro do valor do número passado.

```
1 | #include
2 |     extern int double_me(int a)
3 | {
4 |     return a*2;
5 | }
```

Listagem 1 – Código fonte C para implementação da função.

2. Compilação da mesma para criação do .SO no Linux

Como nosso ambiente de desenvolvimento é linux, vamos compilar a função para criação do objeto SO. Para tal, salve o arquivo com o nome de *double.c* e compile o mesmo com o seguinte comando a partir do shell do linux:

```
1 | gcc -shared -Wl,-soname, libpgdouble.so.1 -o libpgdouble.so double.c
```

Após a compilação será gerado o objeto *libpgdouble.so* que deve ser copiado para

```
1 | cp sqlmagazine.so /usr/local/pgsql/lib
```

3. Criação da função no banco com a mesma assinatura e tipos de dados equivalentes da função implementada em C

O comando a seguir cria no PostgreSQL a função double_me que contém um “ponteiro” para execução da função implementada e compilada nos passos anteriores. Vamos utilizar o banco de dados sqlmagazine criado nas edições anteriores para executar o comando de criação das funções e selects apresentados nesse artigo.

```
1 | CREATE FUNCTION double_me(int4) RETURNS int4 AS '/usr/local/pgsql/li
```

Observe que: i) o tipo de dados do parâmetro da função no PostgreSQL é int4, que é o tipo de dados equivalente ao int no C e, ii) no corpo da função há apenas uma linha “apontando” para o objeto SO criado no passo 2. O último modificador do comando, LANGUAGE ‘C’, indica que a função será implementada utilizando a linguagem C. A *tabela 1* mostra os tipos de dados em C e seus equivalentes no PostgreSQL bem como os includes que devem ser adicionados ao fonte em C para sua utilização.

Para executar a função implementada basta executar o seguinte comando no banco de dados sqlmagazine, como qualquer outra função no PostgreSQL:

```
1 | SELECT double_me(5);
2 |
3 |     double_me
4 | -----
5 |     10
```

processo do PostgreSQL pode ser parado em virtude da violação de segurança.

Tipo SQL	Tipo no C	Definido em
Abstime	AbsoluteTime	utils/nabstime.h
Boolean	bool	postgres.h
Box	BOX*	utils/geo_decls.h
Bytea	bytea*	postgres.h
"char"	char	C PADRÃO
Character	BpChar*	postgres.h
cid	CommandId	postgres.h
Date	DateADT	utils/date.h
smallint (int2)	int2 or int16	postgres.h
int2vector	int2vector*	postgres.h
integer (int4)	int4 or int32	postgres.h
real (float4)	float4*	postgres.h
double precision (float8)	float8*	postgres.h

Interval	Interval*	utils/timestamp.h
Lseg	LSEG*	utils/geo_decls.h
Name	Name	postgres.h
Oid	Oid	postgres.h
Oidvector	oidvector*	postgres.h
Path	PATH*	utils/geo_decls.h
Point	POINT*	utils/geo_decls.h
Regproc	regproc	postgres.h
Reltime	RelativeTime	utils/nabstime.h
Text	text*	postgres.h
Tid	ItemPointer	storage/itemptr.h
Time	TimeADT	utils/date.h
time with time zone	TimeTzADT	utils/date.h
Timestamp	Timestamp*	utils/timestamp.h
Tinterval	TimeInterval	utils/nabstime.h

Varchar	VarChar*	postgres.h
Xid	TransactionId	postgres.h

Tabela 1 – Tipos de dados equivalentes entre o C e o PostgreSQL

Utilizando recursos do PostgreSQL para implementar funções em C

É possível escrever função na forma tradicional como visto na seção anterior ou usando os recursos do header postgres.h, que inclui os tipos de dados do PostgreSQL, funções para definição de parâmetros, etc, tornando o código mais estável uma vez que sua execução é efetuada no mesmo processo do kernel do banco de dados. O exemplo da listagem 2 mostra uma implementação genérica onde a função recebe dois parâmetros e retorna um string com os mesmos concatenados e separados por um espaço.

```

01 #include "postgres.h"
02 #include
03 #include "fmgr.h"
04
05 PG_FUNCTION_INFO_V1(sql_magazine);
06
07 Datum sql_magazine(PG_FUNCTION_ARGS)
08 {
09     text *arg1 = PG_GETARG_TEXT_P(0);
10     text *arg2 = PG_GETARG_TEXT_P(1);
11
12     int32 new_text_size = VARSIZE(arg1) + VARSIZE(arg2) - VARHDRSZ;
13     text *new_text = (text *) palloc(new_text_size);
14
15     VARATT_SIZEP(new_text) = new_text_size;

```

```

19      VARDATA(arg2), VARSIZE(arg2)-VARHDRSZ);
20
21 PG_RETURN_TEXT_P(new_text);
21 }

```

Listagem 2 – Implementação de funções em C utilizando o header postgres.h

Observando a *listagem 2*, temos:

- **Linha 5:** Esta linha diz ao PostgreSQL que estaremos usando a versão 1 (a mais recente) para implementar funções em C. Entre () está o nome da função.
- **Linha 7:** Esta é uma linha padrão no formato Datum nome_funcao(PG_FUNCTION_ARGS)
- **Linhas 9 e 10:** Aqui são criadas variáveis para manipular os valores passados como parâmetros. *text* é um tipo de dados proveniente do *postgres.h* e *PG_GETARG_TEXT_P(0)* retorna o valor do primeiro parâmetro da função definida no PostgreSQL (CREATE FUNCTION) no formato TEXT. Se esse parâmetro fosse do tipo int32, utilizariammos *PG_GETARG_INT32(1)*. Na maioria das vezes, para acessar o parâmetro da função, utilizaremos a sintaxe *PG_GETARG_TIPODEDADOS*(índice) ou *PG_GETARG_TIPODEDADOS_P*(índice) para tipos text e point.
- **Linha 20:** Nesta linha temos a atribuição do retorno da função de acordo com seu tipo de dados. A função de retorno tem sempre o formato *PG_RETURN_TIPODEDADO(valor)* ou *PG_RETURN_TIPODEDADO_P(valor)* para tipos text e point.

Salve o arquivo da *listagem 2* como *sqlmagazine.c* e compile com o seguinte comando:

```
1 | gcc -fpic -I ../../src/include/ -c sqlmagazine.c
```

Este comando irá compilar e gerar um arquivo “.o”. Agora iremos transformá-lo em .so:

```
1 | gcc -shared -o sqlmagazine.so sqlmagazine.o
```

Copie o arquivo SO gerado para a pasta lib do PostgreSQL com o comando

```
1 | #cp sqlmagazine.so /usr/local/pgsql/lib
```

E por fim crie a função no banco de dados *sqlmagazine* com o seguinte comando:

```
1 | CREATE FUNCTION sql_magazine(text,text) RETURNS text AS '/usr/local/
```

Para verificar o resultado da função criada, execute o comando a seguir:

```
1 | select sql_magazine('SQL ', ' Magazine');
2 |   sql_magazine
3 | -----
4 |   SQL Magazine
5 | (1 row)
```

Como executar um código SQL nas funções implementadas em C

O “acesso” e execução de comandos SQL no PostgreSQL a partir de funções implementadas em C é efetuado através de funções contidas no header *spi.h*. Essas funções possibilitam, além da execução de comandos, a manipulação e acesso aos registros retornados por uma determinada consulta, o controle da quantidade de registros, verificação de erros gerados pela execução dos

comandos contidos anteriormente.

O exemplo da *listagem 3* implementa uma função que, a partir de um valor inteiro passado como parâmetro, busca o registro da tabela *sql_magazine* cujo identificador equivale ao parâmetro passado e retorna a mensagem equivalente ao valor ou uma mensagem de erro do processamento.

```
01 #include "postgres.h"
02 #include
03 #include "fmgr.h"
04 #include "executor/spi.h"
05
06 PG_FUNCTION_INFO_V1(sql_magazine);
07
08 Datum
09 sql_magazine(PG_FUNCTION_ARGS)
10 {
11     text *arg1 = PG_GETARG_TEXT_P(0);
12     text *new_text;
13     char query[150];
14     int ret,proc;
15     int new_text_size = 8024;
16     SPI_connect();
17
18     sprintf(query,"select mensagem from sql_magazine where ano =
19     ret = SPI_exec(query, 0);
20
21     if (ret != SPI_OK_SELECT) {
22         elog(ERROR, "SPI_exec, XIII AI MEU DEUS, ERRO!!!.");
23         PG_RETURN_NULL();
24     }
25
26     proc = SPI_processed;
27
28     if (proc <= 0) {
29         elog(ERROR, "SPI_exec, sem retorno de registros!");
30         PG_RETURN_NULL();
31     }
32     new_text = (text *) palloc(new_text_size);
33 }
```

```
37 SPI_finish();  
38  
39 PG_RETURN_TEXT_P(new_text);  
40  
41 pfree(new_text);  
42 }
```

Listagem 3 – Executando comandos SQL em funções C

Observando a *listagem 3*, temos:

- **Linha 11:** Cria uma variável e inicializa com o valor do parâmetro passado.
- **Linha 16:** “Abre” a conexão com a seção que chamou a execução da função em C.
- **Linha 18:** “Monta” a string com a frase SQL a ser executada e armazena na variável *query*.
- **Linha 19:** Executa a frase SQL através da função *SPI_EXEC* e armazena o retorno deste processamento na variável *ret*. A função *SPI_EXEC* possui dois parâmetros: o primeiro equivale ao comando a ser executado e o segundo a quantidade de registros afetados ou retornados pelo comando - em nosso exemplo utilizamos ‘zero’ pois vamos processar todos os registros retornados.
- **Linha 26:** Armazena na variável *proc* o número de registros processados ou retornados pelo comando executado, através da função *SPI_PROCESSED*.
- **Linhas 34 e 35:** Acessam o valor da primeira coluna *vals[0]* do registro corrente retornado pelo *SPI_EXEC*, através do *SPI_tuptable*.
- **Linha 37:** “Fecha” a conexão aberta na linha 11.

Os comandos a seguir compilam o código da *listagem 3* e movem o objeto C para a pasta lib do PostgreSQL.

```
mv sqlmagazine.so /usr/local/pgsql/lib/
```

Criando a função no PostgreSQL no banco de dados *sqlmagazine*:

```
1 | CREATE FUNCTION sql_magazine(text) RETURNS text AS '/usr/local/pgsql
```

Os comandos a seguir criam a tabela utilizada na listagem 3 e efetuam a inserção de um registro na mesma:

```
1 | CREATE TABLE SQL_MAGAZINE(mensagem text, ano char(4));  
2 |     INSERT INTO SQL_MAGAZINE(mensagem,ano) VALUES('Feliz 2004!!!',
```

Vamos verificar dois possíveis resultados da execução da função: O primeiro com um identificador não cadastrado e o segundo com um identificador cadastrado pela listagem anterior:

```
1 | select sql_magazine('2222');  
2 |     ERROR: SPI_exec, sem retorno de registros!  
3 |  
4 | select sql_magazine('2004');  
5 |     sql_magazine  
6 |-----  
7 |     Feliz 2004!!!  
8 |     (1 row)
```

Conclusão

Esse artigo mostrou de forma introdutória a utilização do C para implementação de funções no PostgreSQL. É claro que sua utilização requer conhecimento e experiência na linguagem mas, o horizonte que a mesma abre ao desenvolvedor



Anotar



Marcar como concluído

Confira outros conteúdos:

SQL

SQL SUM: somando os
valores de uma...

 DEVMEDIA

SQL INNER JOIN

SQL: INNER JOIN

**PARA QUEM QUER SER
PROGRAMADOR DE VERDADE.
VAGAS LIMITADAS**

Em caso de dúvidas chame no whatsapp



Plano Recorrente	
R\$ 89,90 /MÊS	PRIMEIROS 3 MESES
R\$ 49,90 /MÊS	A PARTIR DO 4º MÊS
12 MESES = R\$ 718,80	
Formação FullStack completa	✓
+10mil exercícios gamificados	✓
+50 projetos reais	✓
Suporte online	✓
Pra quem tem pouco limite no cartão	✓
Fidelidade de 12 meses	✓
Matricule-se	

Perguntas Frequentes

Quem somos?

Por que a programação se tornou a profissão mais promissora da atualidade?

Como faço para começar a estudar?

Em quanto tempo de estudo vou me tornar um programador?

Sim, você pode se tornar um programador e não precisa ter diploma de curso superior!

O que eu irei aprender estudando pela DevMedia?

Principais diferenciais da DevMedia

Qual o investimento financeiro que preciso fazer para me tornar um programador?

Como funciona a forma de pagamento da DevMedia?

Nossos casos de sucesso

Leonardo Carlos



Eu sabia pouquíssimas coisas de programação antes de começar a estudar com vocês, fui me especializando em várias áreas e ferramentas que tinham na plataforma, e com essa bagagem **consegui um estágio logo no início do meu primeiro período na faculdade.**

Lucas Rodrigues



Estudo aqui na Dev desde o meio do ano passado! Nesse período a Dev me ajudou a crescer muito aqui no trampo.

Fui o primeiro desenvolvedor contratado pela minha empresa. Hoje eu lidero um time de desenvolvimento!

Heráclito Júnior

Economizei 3 meses para assinar a plataforma e sendo sincero valeu muito a pena, pois a **plataforma é bem intuitiva e muuuuito didática a metodologia de ensino**. Sinto que estou EVOLUINDO a cada dia. Muito obrigado!

Julio Cahlen

Nossa! Plataforma maravilhosa. To amando o curso de desenvolvimento front-end, tinha coisas que eu ainda não tinha visto. **A didática é do jeito que qualquer pessoa consegue aprender**. Sério, to apaixonado, adorando demais.

Joelberth Sena

Adquiri o curso de vocês e logo percebi que são os melhores do Brasil. É um passo a passo incrível. **Só não aprende quem não quer. Foi o melhor investimento da minha vida!**

Felipe Nunes

Foi um dos melhores investimentos que já fiz na vida e tenho aprendido bastante com a plataforma. Vocês estão fazendo parte da minha jornada nesse mundo da programação, **irei assinar meu contrato como programador graças a plataforma**.

Wanderson Oliveira

Comprei a assinatura tem uma semana, aprendi mais do que 4 meses estudando outros cursos. Exercícios práticos que não tem como não aprender, estão de parabéns!

Obrigado DevMedia, nunca presenciei **uma plataforma de ensino tão presente na vida acadêmica de seus alunos**, parabéns!

Eduardo Dorneles



Aprendi React na plataforma da DevMedia há cerca de 1 ano e meio... **Hoje estou há 1 ano empregado** trabalhando 100% com React!

Adauto Junior



Já fiz alguns cursos na área e **nenhum é tão bom quanto o de vocês**. Estou aprendendo muito, muito obrigado por existirem. Estão de parabéns... Espero um dia conseguir um emprego na área.

[Ver todos os casos de sucesso](#)



Por Paulo

Em 2008

Cursos

DEVMEDIA



[Quem Somos](#)[Fale conosco](#)[Plano para Instituição de ensino](#)[Assinatura para empresas](#)[Assine agora](#)

Hospedagem web por Porta 80 Web Hosting.

