Facilitando a vida do DBAdmin

Funções no PostgreSQL

por Alex Gott-Cumbers: www.sxc.hu

Neste pequeno tutorial, veremos de forma sucinta como criar funções definidas pelo usuário em linguagem SQL para auxiliar na execução de tarefas simples de retorno de dados.

POR PABLO DALL'OGLIO

As funções de banco de dados mais comuns são as funções de agregação `sum()`, `count()`, `avg()`, `min()` e `max()`, utilizadas para operações simples como soma, contagem, média, mínimo e máximo. O PostgreSQL possui também uma gama de funções diversas para operações aritméticas, conversões entre tipos de dados, funções de formatação e geográficas, entre outras. Mas além dessas funções pode-se criar uma infinidade de outras, definidas pelo usuário, permitindo encapsular código requerido para tarefas comuns. Tais funções podem ser criadas em linguagem C, SQL, PL/PGSQL, PL/TCL, PL/PERL e até PHP (com suporte experimental).

A primeira função que iremos criar é chamada *km2mi*, usada para realizar a conversão de distâncias expressas em quilômetros para milhas por meio de uma operação aritmética através de instruções SQL. Para a criar a função, usamos o comando `create function` seguido do nome da função, o tipo dos parâmetros (entre parênteses), o tipo de retorno, a ação tomada (no exemplo uma cláusula SQL) e o tipo de linguagem (SQL). Os parâmetros são representados na ordem em que são passados para a função, como (`$1`, `$2`, `$3`, ...).

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

Para usar a função, basta utilizar o comando `select`. No exemplo a seguir, convertemos cem quilômetros para milhas:

```
samples=# select km2mi (100) as milhas;
milhas: 60
```

Tabela 1: Clientes

```
# select codigo, nome, cidade, idade from clientes limit 5;
```

codigo	nome	cidade	idade
1	Mauricio de Castro	Lajeado	6
2	Cesar Brod	Lajeado	38
10	Nasair da Silva	Lajeado	20
4	Joao Alex Fritsch	Lajeado	29
5	Daniel Afonso Heisler	Santa Clara	23

Tabelas

Os exemplos dados utilizarão um pequeno banco de dados contendo três tabelas: *clientes*, *produtos* e *compras*. A tabela *clientes* possui a estrutura básica de um cadastro de clientes, com informações como código, nome, telefone, rua, cidade e idade. Veja um exemplo na **tabela 1**.

Já a tabela de produtos possui as informações do cadastro de mercadorias, como código, descrição, unidade, estoque atual, valor de compra e valor de venda do produto. Veja um exemplo na **tabela 2.**

Por fim, temos a tabela de compras, que armazena as compras dos clientes. Ali há informações como código do cliente, código do produto, quantidade comprada, data da compra e o preço pago. Veja um exemplo na **tabela 3**.

Retornos múltiplos

Embora a maioria das funções retorne apenas um valor, é possível retornar múltiplos valores através da cláusula `SETOF`. As funções podem também realizar operações como `Insert`, `Update` e `Delete`, bem como múltiplas pesquisas delimi-

tadas por ;. A seguir, criaremos uma função para retornar os nomes de todos os clientes que são menores de idade. Veja que o retorno da função é um conjunto do tipo "clientes".

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

Utilizamos a função através do comando `select`. Veja o exemplo da chamada e seu resultado na **tabela 4**.

Joins

Uma grande utilidade das funções do banco de dados é auxiliar na obtenção de informações não vinculadas à tabela principal. No exemplo a seguir, estamos buscando, no banco de dados, diversas informações provenientes do cruzamento das tabelas clientes, compras e produtos. Essas informações são o código do cliente, seu nome, a quantidade de produto comprada e o preço pago, além do código e da descrição do produto, sendo que o nome do cliente está na tabela de clientes e a descrição do produto está na tabela de produtos. A consulta e seu resultado são mostrados na **tabela 5**.

Essa consulta, que necessita do cruzamento de três tabelas (clientes, compras e produtos), poderia ser substituída por um `select` apenas sobre a tabela que contém os dados, incluíndo as chaves estrangeiras (tabela de compras). Para isso, criaremos duas funções, `get_cliente`, cujo papel é buscar o nome do cliente na tabela de clientes através do código, e `get_produto`, cujo papel é buscar a descrição do produto na tabela de produtos, também através do código. Vamos criá-las assim :

Tabela 2: Produtos

```
# select codigo, descricao, unidade, estoque from ➥
produtos limit 5;
```

codigo	descricao	unidade	estoque
2	Sabao	LT	50
3	Refrigerante	LT	900
4	Bala Flopi	PC	1400
5	Sorvete	LT	400
6	Suco em po	PC	200

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

Dessa forma, o `select` irá trazer os registros da tabela principal e, para cada iteração, buscará na tabela auxiliar correspondente (clientes ou produtos) os dados necessários, através das funções criadas acima. Veja na **tabela 6** como fica a consulta, utilizando as funções `get_cliente()` e `get_produto()`.

Uma das vantagens de se utilizar funções para buscar as informações nas tabelas auxiliares é que, mesmo quando o registro não existir na tabela auxiliar (cliente ou produto), os dados da tabela principal (compras) irão ser exibidos. O mesmo

Tabela 3: Compras

```
# select * from compras limit 5;
```

ref_cliente	ref_produto	quantidade	data	preco
1	1	2	2003-10-31	2.4
3	1	42	2003-10-31	2.4
5	1	7	2003-10-31	2.4
7	1	7	2003-10-31	2.3
9	1	3	2003-10-31	2.3

Tabela 4: Funções *menores ()*

```
# select codigo, nome from menores();
```

codigo	nome
10	Nasair da Silva
11	Jamiel Spezia
12	Henrique Gravina
13	William Prigol Lopes

Tabela 5: Busca de informações através do cruzamento

```
# select c.codigo, c.nome, m.quantidade, m.preco, p.codigo, p.descricao from clientes c, compras m, produtos p where ➥
c.codigo=m.ref_cliente and p.codigo=m.ref_produto;
```

codigo	nome	quantidade	preco	codigo	descricao
1	Mauricio de Castro	2	2.4	1	Chocolate
5	Daniel Afonso Heisler	7	2.4	1	Chocolate
7	Vilson Cristiano Gartner	7	2.3	1	Chocolate
9	Alexandre Schmidt	3	2.3	1	Chocolate
3	Pablo DallOglio	42	2.4	1	Chocolate
1	Mauricio de Castro	2	1.4	2	Sabao
2	Cesar Brod	2	1.4	2	Sabao
10	Nasair da Silva	9	1.5	2	Sabao
4	Joao Alex Fritsch	5	1.4	2	Sabao
6	Paulo Roberto Mallmann	4	1.4	2	Sabao

Tabela 6: Consulta utilizando as funções *get_cliente()* e *get_produto()*

```
# select ref_cliente, get_cliente(ref_cliente), quantidade, preco, ref_produto, get_produto(ref_produto) from compras;
```

ref_cliente	get_cliente	quantidade	preco	ref_produto	get_produto
1	Mauricio de Castro	2	2.4	1	Chocolate
5	Daniel Afonso Heisler	7	2.4	1	Chocolate
7	Vilson Cristiano Gartner	7	2.3	1	Chocolate
9	Alexandre Schmidt	3	2.3	1	Chocolate
3	Pablo DallOglio	42	2.4	1	Chocolate
1	Mauricio de Castro	2	1.4	2	Sabao
2	Cesar Brod	2	1.4	2	Sabao
10	Nasair da Silva	9	1.5	2	Sabao
4	Joao Alex Fritsch	5	1.4	2	Sabao
6	Paulo Roberto Mallmann	4	1.4	2	Sabao

Listagem 1: Função *get_signo()*

```
01 create function get_numdate (date) returns integer as 'select (substr( ↵
   $1 , 6,2) || substr( $1 , 9,2))::integer' language 'SQL';
02 create function get_signo (int) returns varchar as
03 'select
04   case
05         when $1 <=0120 then \'capricornio\'
06         when $1 >=0121 and $1 <=0219 then \'aquario\'
07         when $1 >=0220 and $1 <=0320 then \'peixes\'
08         when $1 >=0321 and $1 <=0420 then \'aries\'
09         when $1 >=0421 and $1 <=0520 then \'touro\'
10         when $1 >=0521 and $1 <=0620 then \'gemeos\'
11         when $1 >=0621 and $1 <=0722 then \'cancer\'
12         when $1 >=0723 and $1 <=0822 then \'leao\'
13         when $1 >=0823 and $1 <=0922 then \'virgem\'
14         when $1 >=0923 and $1 <=1022 then \'libra\'
15         when $1 >=1023 and $1 <=1122 then \'escorpiao\'
16         when $1 >=1123 and $1 <=1222 then \'sagitario\'
17         when $1 >=1223 then \'capricornio\'
18   end as signo' language 'SQL'
```

Tabela 7: Usando a função *categoria()*

```
# select codigo, nome, idade, categoria(codigo) from clientes;
```

codigo	nome	idade	categoria
1	Mauricio de Castro	26	b
2	Cesar Brod	38	c
4	Joao Alex Fritsch	29	b
5	Daniel Afonso Heisler	23	b
6	Paulo Roberto Mallmann	23	b
7	Vilson Cristiano Gartner	30	c
9	Alexandre Schmidt	25	b
3	Pablo DallOglio	23	b
13	William Prigol Lopes	16	a
14	Viviane Berner	27	b
15	Marcia Cantu	31	c
16	Joice Kafer	21	b
11	Jamiel Spezia	17	a
12	Henrique Gravina	17	a
10	Nasair da Silva	17	a

não acontece se utilizarmos o `join` natural, que faz o cruzamento das tabelas utilizando a lógica de matrizes. Logo, se o dado de um conjunto não tiver correspondente no outro conjunto, simplesmente não aparecerá no conjunto final. Outra grande vantagem são ganhos de desempenho, percebidos ao se manipular grandes quantidades de dados provindos de tabelas distintas.

Case

Na criação de funções podemos combinar muitos recursos. Um deles é o `case`. Através dele podemos realizar consultas e retornar valores condicionais. No exemplo demonstrado a seguir, construiremos uma função chamada `categoria()`. O papel da função categoria é retornar `a` para pessoas com idade menor que 20 anos, `b` para pessoas entre 20 e 30 anos e `c` para pessoas com mais de 30 anos de idade. O único parâmetro recebido pela função é o código da pessoa. Para retornar o resultado é necessário utilizar a barra invertida (`\`), uma vez que as aspas já são usadas ao redor da expressão SQL:

```
# 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';
```

Na **tabela 7**, demonstramos a utilização da função `categoria()` em uma consulta que retornava o código, nome, idade e categoria de um registro da tabela de clientes.

Na **listagem 1**, criamos uma pequena função em SQL para retornar o signo de uma pessoa (`get_signo`), passando como parâmetro a data de nascimento. Para tal, antes precisamos criar uma outra função (`get_numdate`), que converte uma data qualquer como *1978-04-12* em um dado numérico *0412*, formado pelo

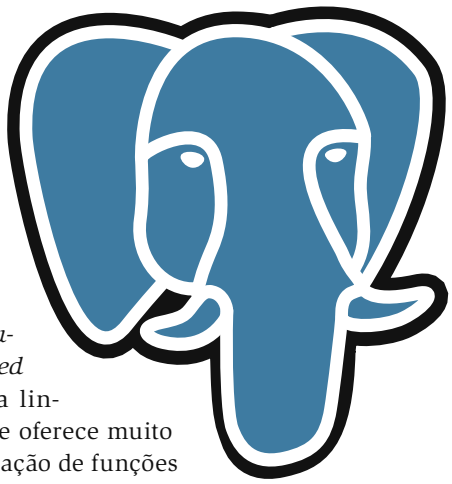
mês e pelo dia. Para tal, utilizaremos a função `date_part()` do PostgreSQL que retorna uma parte específica (ano, mês, dia) da data, para facilitar a compreensão do código. Veja exemplos de uso desta função:

```
# 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
```



Neste artigo você viu de forma sucinta como é fácil lidar com funções SQL em geral, com exemplos específicos do PostgreSQL. Num futuro artigo veremos como trabalhar com funções em PL/PGSQL (*Procedural Language/Structured Query Language*), uma linguagem estruturada que oferece muito mais recursos para a criação de funções que realizem tarefas específicas do lado do banco de dados. ■

Sobre o autor

Pablo Dall'Oglio (pablo@php.net) trabalha com programação e análise de sistemas desde 1995. É criador dos projetos Agata Report (www.agata.org.br) e Tulip (tulip.solis.coop.br) e autor do primeiro livro exclusivo sobre PHP-GTK (www.php-gtk.org.br) no mundo, publicado pela editora Novatec. Atualmente trabalha como desenvolvedor e consultor de TI, realizando prospecções, análise e implantação de sistemas para gestão acadêmica e para gestão de acervos bibliográficos pela cooperativa de soluções livres SOLIS (www.solis.org.br).

Informações

[1] DB Experts, http://www.dbexperts.net

[2] Página oficial do PostgreSQL: http://www.postgresql.org

[3] PL/PGSQL: http://www.postgresql.org/docs/8.0/interactive/plpgsql.html

[4] Otimizando o desempenho do PostgreSQL: http://www.varlena.com/varlena/GeneralBits/Tidbits/perf.html

[5] Histórico e descrição: http://en.wikipedia.org/wiki/PostgreSQL

[6] SQL: http://en.wikipedia.org/wiki/Sql

[7] Momjian, Bruce. PostgreSQL - Introduction and Concepts.