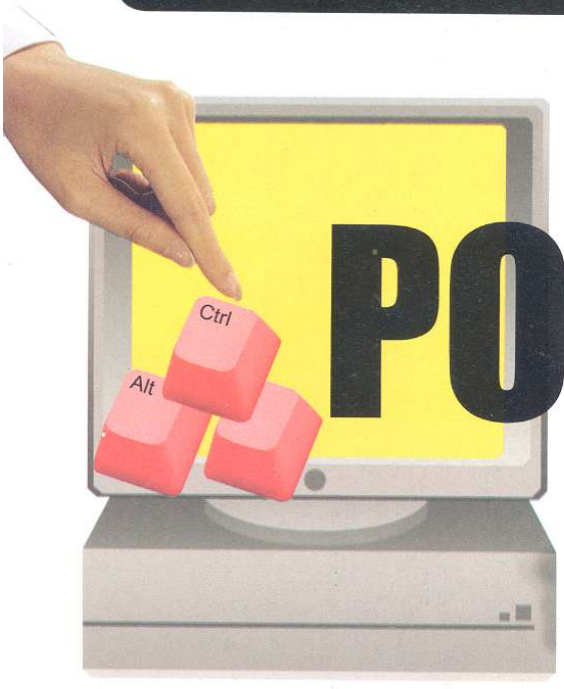


Funções SQL no POSTGRESQL

Aprenda a criar as instruções básicas para seu banco de dados

Pablo Dall' Oglio
pablo@dalloglio.net

Um banco de dados não serve para nada se algumas funções não forem criadas corretamente. As funções de banco de dados são necessárias para definir parâmetros e executar pesquisas e conversões, de acordo com a necessidade do usuário, além, é claro, de armazenar informações.

As funções mais comuns são as 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 de tipos de dados, funções de formatação, geográficas, entre outras. Mas além destas funções, pode-se criar uma infinidade de outras, definidas pelo usuário, permitindo encapsular códigos requeridos para tarefas comuns. Tais funções podem ser criadas em linguagem C, SQL, PL/SQL, PL/TCL, PL/PERL e até PL/PHP. Neste tutorial, você verá como criar funções definidas pelo usuário em linguagem SQL para auxiliar nas tarefas simples do dia-a-dia.

A primeira a ser criada é uma função para realizar a conversão de quilômetros para milhas (*km2mi*), realizando uma operação aritmética por meio de instruções em SQL. Para criação da função, será utilizado o comando *create function* seguido do nome da função, os tipos dos parâmetros (entre parentesis), o tipo de retorno, a ação tomada (que neste exemplo é uma cláusula SQL) e o tipo de linguagem (SQL). Os parâmetros são representados exatamente na ordem 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 utilizar a função, basta executar o comando *select*. No exemplo a seguir, converteremos "100", de quilômetros para milhas:

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

TABELAS

Os exemplos a seguir utilizarão este pequeno banco de dados proposto e que contém 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 o exemplo a seguir:

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

cod	nome	cidade	idade
1	Mauricio de Castro	Lajeado	26
2	Cesar Brod	Lajeado	38
10	Nasair da Silva	Lajeado	20
4	Joao Alex Fritsch	Lajeado	29
5	Daniel Afonso Heisler	Santa Clara	23

Já a tabela *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:

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

JOINS

Uma grande utilidade das funções do Banco de Dados é auxiliar na obtenção de dados não vinculados à tabela principal. No exemplo a seguir, estamos buscando diversas informações do banco de dados provenientes do cruzamento das tabelas *Clientes, Compras e Produtos*. Tais informações são o código do cliente, seu nome, a quantidade comprada, o preço pago, o código do produto e a descrição do produto, sendo que o nome do cliente está na tabela de clientes e a descrição do produto está na tabela de produtos. Veja o exemplo:

```
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;
```

cod	nome	qtd	preco	cod	descrição
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	

A consulta acima, 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 tal, serão criadas duas funções, *get_cliente* (cujo papel é buscar o nome do cliente na tabela de clientes por meio do código) e *get_produto* (cujo papel é buscar a descrição do produto na tabela de produtos pelo código). Abaixo, veja como elas estão sendo criadas:

```
samples=# create function get_cliente (int) returns varchar as
'select nome from clientes where codigo = $1' language 'SQL';
samples=# 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 interação, buscará na tabela auxiliar correspondente (*Clientes ou Produtos*) os dados necessários, por meio das funções criadas acima. Veja abaixo como fica o *Select*, utilizando as funções *get_cliente()* e *get_produto()*.

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

ref_cliente	get_cliente	qtd	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

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 não acontece com o *join* natural, que efetua o cruzamento das tabelas, pela a lógica de matrizes, a não ser que se utilize outros *join*.

Logo, se a informação de um conjunto não tiver um correspondente no outro grupo de dados, simplesmente também não estará no resultado final. Outra grande vantagem são ganhos de performance, percebidos ao se manipular grandes quantidades de dados provenientes de tabelas distintas.

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

Por fim, temos a tabela de compras, que armazena as compras dos clientes. Nela, temos informações como código do cliente, código do produto, quantidade comprada, data da compra e preço pago.

Veja um exemplo:

```
samples=# 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	0	2003-10-31	2.4
7	1	7	2003-10-31	3.3
9	1	3	2003-10-31	3.3

CASE

Na criação de funções, é possível combinar diversos recursos, como o *Case*. Com ele, podemos realizar consultas e retornar valores condicionais. No exemplo demonstrado a seguir, será construída 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 backslashes (\), já que as aspas são utilizadas ao redor da expressão SQL.

```
samples=# 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';
```

A seguir, será demonstrada a utilização da função *categoria()* em um *Select* ao retornar o *Código, Nome, Idade e Categoria da tabela de clientes*.

```
samples=# 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

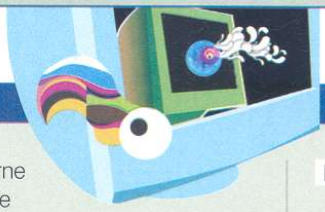
Abaixo, será criada uma pequena função em SQL para retornar o *Signo* de uma pessoa (*get_signo*), passando como parâmetro a data de nascimento. Para tal, antes será necessário criar uma outra função (*get_numdate*), que converte uma data qualquer *'1978-04-12'* em um dado numérico (*0412*), formado pelo mês e dia de nascimento.

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

Veja exemplos de uso:

```
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
```

RETORNOS MÚLTIPLOS



Embora a maioria das funções retorne apenas um valor, pode-se fazer com que apresente múltiplos resultados com a cláusula *SETOF*. As funções podem também realizar operações como *Inserts, Updates, Deletes*, bem como múltiplas pesquisas delimitadas por ";".

A seguir, será criada uma função para retornar todos os clientes 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';
```

Foi utilizada a função como comando *Select*. Veja o exemplo:

```
select codigo, nome from  menores();
codigo  | nome
--------+-----------------------------------
  10    | Nasair da Silva
  11    | Jamiel Spezia
  12    | Henrique Gravina
  13    | William Prigol Lopes
```

FUNÇÕES PL/PGSQL

PL/SQL ou Procedural Language/Structured Query Language é uma extensão da SQL que combina a flexibilidade da SQL com as funcionalidades de uma linguagem de programação estruturada. Além de permitir os comandos SQL com operações envolvendo desde cursores e controle de transações, ainda permite construções como blocos condicionais e procedures. Para iniciar a utilização da linguagem PL/PGSQL, que nada mais é do que a implementação do PostgreSQL para esta especificação, será mostrada a estrutura de uma função em PL/PGSQL:

DECLARE – Declaração de variáveis e tipificação.

BEGIN – Seção que contém o código executável, comandos SQL e procedimentos.

END; – Finalização do bloco de comandos. Para iniciar, vamos construir uma função que irá indicar se um número é primo ou não. Neste caso, há exemplos de estruturas de controle *IF* e *FOR*, além do comando *RAISE EXCEPTION,* que retorna uma mensagem de erro, e do comando *RAISE NOTICE,* que, juntamente com o resultado, exibe uma mensagem de notícia.

```
create function primo(integer) returns boolean as '
DECLARE
nCOUNT integer;
iPRIMO ALIAS FOR $1;
BEGIN
```

Só números inteiros naturais podem ser primos

```
IF iPRIMO < 0 THEN
RAISE EXCEPTION "Numero negativo";
END IF;
```

Ignora 1 por convenção

```
IF iPRIMO = 0 or iPRIMO = 1 THEN
RAISE NOTICE "Não primo, por convenção";
return false;
END IF;
```

Percorre todo intervalo

```
FOR nCOUNT IN 2..iPRIMO-1 LOOP
```

Se for divisível, aborta

```
IF mod(iPRIMO, nCOUNT) = 0 THEN
return false;
END IF;
END LOOP;
return true;
END;'
language 'plpgsql';
```

Resultado:

```
samples=# select primo(4);
primo
f
(1 row)
samples=# select primo(5);
primo
t
(1 row)
samples=# select primo(1);
NOTICE: Não primo, por convenção
primo
f
(1 row)
samples=# select primo(0);
NOTICE: Não primo, por convenção
primo
f
(1 row)
samples=# select primo(-1);
ERROR: Numero negativo
samples=# select primo(-1234);
ERROR: Numero negativo
```

Caso ocorra a seguinte mensagem de erro, indicando que a linguagem *plpgsql* não está presente, tendo a biblioteca *plpgsql.so*, é necessário utilizar o comando *createlang* para disponibilizá-la na base de dados.

```
ERROR: language "plpgsql" does not exist
createlang plpgsql <banco de dados>
# createlang plpgsql samples
```

No exemplo seguinte, o objetivo é construir uma função que insira uma nota fiscal (*na tabela notas*) e seus itens (*na tabela itens*). Para tal, será criado um registro na tabela *notas* e o total os itens na tabela correspondente. A função recebe o número da nota e a quantidade de itens. Neste exemplo, será mostrado um caso de utilização de comandos SQL dentro de funções PL/PGSQL e também da estrutura de controle *WHILE*.

```
create table notas (numero integer);
create table itens (numero integer, item integer);
create function insere(integer, integer) returns void as '
DECLARE
iCOUNT integer;
iNUMERO ALIAS FOR $1;
iQUANTIDADE ALIAS FOR $2;
BEGIN
```

Insere na tabela notas

```
insert into notas values (iNUMERO);
iCOUNT := 1;
```

Loop para quantidade de itens

```
WHILE iCOUNT <= iQUANTIDADE LOOP
```

Insere na tabela itens

```
insert into itens values(iNUMERO, iCOUNT);
iCOUNT := iCOUNT +1;
END LOOP;return null;
END;'
language 'plpgsql';
samples=# select insere(1,3);
```

FUNÇÕES PL/PGSQL

```
samples=# select insere(2,4);
samples=# select * from notas;
numero
     1
     2
samples=# select * from itens;
numero | item
     1 | 1
     1 | 2
     1 | 3
     2 | 1
     2 | 2
     2 | 3
     2 | 4
```

Agora, será criada uma função para exibir números por extenso. Antes disso, será montada uma função auxiliar que recebe uma string de três números e os imprime. Essa função, chamada de *tres_extenso*, será utilizada pela função principal, chamada de *extenso()*, conforme será mostrado mais adiante.

```
create function tres_extenso(varchar) returns
VARCHAR as '
DECLARE
aUNIDADE varchar[];
aDEZENA varchar[];
aCENTENA varchar[];
aEXCESSAO varchar[];
iPOSICAO1 integer;
iPOSICAO2 integer;
iPOSICAO3 integer;
vCENTENA varchar;
vDEZENA varchar;
vUNIDADE varchar;
vRESULTADO varchar;
vVALOR ALIAS FOR $1;
BEGIN
```

Declaração de vetores para Unidades, Dezenas, Centenas e de 10 a 19

```
aUNIDADE := "{,UM ,DOIS ,TRES ,QUATRO
,CINCO ,SEIS ,SETE ,OITO ,NOVE }";
aDEZENA := "{ , ,VINTE E,TRINTA
E,QUARENTA E,CINQUENTA E,SESSENTA
E,SETENTA E,OITENTA E,NOVENTA E}";
aCENTENA := "{ ,CENTO E,DUZENTOS
E,TREZENTOS E,QUATROCENTOS
E,QUINHENTOS E,SEISCENTOS
E,SETECENTOS E, OITOCENTOS
E,NOVECENTOS E}";
aEXCESSAO := "{DEZ ,ONZE ,DOZE ,TREZE
,QUATORZE ,QUINZE ,DESESSEIS
,DESESSETE ,DEZOITO ,DESENOVE }";
```

Extrai as posições de centena, dezena e unidade

```
iPOSICAO1 := substr(vVALOR,1,1);
iPOSICAO2 := substr(vVALOR,2,1);
iPOSICAO3 := substr(vVALOR,3,1);
```

Busca nos vetores as palavras correspondentes

```
vCENTENA := aCENTENA[iPOSICAO1 +1];
vDEZENA := " " || aDEZENA[iPOSICAO2 +1];
vUNIDADE := " " || aUNIDADE[iPOSICAO3 +1];
```

Trata a exceção: 100

```
IF substr(vVALOR,1,3) = "100" THEN
vCENTENA := "CEM "; END IF;
```

Trata as exceções de 10 a 19

```
IF substr(vVALOR,2,1) = "1" THEN
vDEZENA := " " || aEXCESSAO[iPOSICAO3 +1];
vUNIDADE := ""; END IF;
```

Monta a string de resultado

```
vRESULTADO := vCENTENA || vDEZENA ||
vUNIDADE;
vRESULTADO := trim(vRESULTADO);
```

Verifica se sobrou um "E" no final

```
IF substr(vRESULTADO,length(vRESULTADO)-1,2) = " E" THEN
vRESULTADO := substr(vRESULTADO, 1,
length(vRESULTADO)-1); END IF;
```

Retorna o resultado

```
return vRESULTADO; END; ' language plpgsql;
```

Exemplos de uso:

```
samples=# select tres_extenso('001');
tres_extenso
UM
samples=# select tres_extenso('004');
tres_extenso
QUATRO
samples=# select tres_extenso('012');
tres_extenso
DOZE
samples=# select tres_extenso('016');
tres_extenso
DESESSEIS
samples=# select tres_extenso('020');
tres_extenso
VINTE
samples=# select tres_extenso('044');
tres_extenso
QUARENTA E QUATRO
samples=# select tres_extenso('100');
tres_extenso
CEM
samples=# select tres_extenso('123');
tres_extenso
CENTO E VINTE E TRES
samples=# select tres_extenso('200');
tres_extenso
DUZENTOS
samples=# select tres_extenso('999');
tres_extenso
NOVECENTOS E NOVENTA E NOVE
```

Agora, após a função auxiliar, será criada a função *extenso()*, que recebe um valor do tipo *numeric* e retorna o número por extenso.


```
create function extenso(numeric) returns text as '
DECLARE
vMOEDA_SIN varchar;
vMOEDA_PLU varchar;
vMILHAO varchar;
vMILHAR varchar;
vUNIDADE varchar;
vCENTAVO varchar;
vCOMPL_MILHAO varchar;
vCOMPL_MILHAR varchar;
vCOMPL_UNIDADE varchar;
vRESULTADO text;
nVALOR ALIAS FOR $1;
vVALOR varchar;
BEGIN
```

Moeda corrente no singular e plural

```
vMOEDA_SIN := " REAL";
vMOEDA_PLU := " REAIS";
```

Formata o valor de acordo com a máscara 999999999.99

```
vVALOR := replace(substr(to_char(nVALOR,
"999999999D00"), 2), " ", "0");
```

Usa a função tres_extenso para obter quantos milhões

```
vMILHAO :=  tres_extenso(substr(vVALOR,1,3));
IF (substr(vVALOR,1,3)::integer > 1) THEN
vMILHAO := vMILHAO || " MILHOES";
ELSE IF (substr(vVALOR,1,3)::integer = 1) THEN
vMILHAO := vMILHAO || " MILHAO"; END IF; END IF;
```

Usa a função tres_extenso para obter quantos mil

```
vMILHAR :=  tres_extenso(substr(vVALOR,4,3));
IF (substr(vVALOR,4,3)::integer > 0) THEN
vMILHAR := vMILHAR || " MIL"; END IF;
```

Usa a função tres_extenso para obter quantas unidades

```
vUNIDADE := tres_extenso(substr(vVALOR,7,3));
IF (substr(vVALOR,7,3)::integer = 1) THEN
vUNIDADE := vUNIDADE || vMOEDA_SIN; ELSE
vUNIDADE := vUNIDADE || vMOEDA_PLU; END IF;
```

Usa a função tres_extenso para obter quantos centavos

```
vCENTAVO := tres_extenso("0" || substr(vVALOR,11,2));
IF (substr(vVALOR,11,2)::integer > 1) THEN
vCENTAVO := vCENTAVO || " CENTAVOS";
ELSE IF (substr(vVALOR,11,2)::integer = 1) THEN
vCENTAVO := vCENTAVO || " CENTAVO"; END IF; END IF;
```

Verifica a necessidade de "," após o milhão

```
IF length(trim(vMILHAO))::integer<>0 and
length(trim(vMILHAR))::integer<>0 THEN
vCOMPL_MILHAO := ", "; ELSE vCOMPL_MILHAO := "";
END IF;
```

Verifica a necessidade de "," após o mil

```
IF length(trim(vMILHAR))::integer<>0 and
length(trim(vUNIDADE))::integer<>0 THEN
vCOMPL_MILHAR := ", "; ELSE
vCOMPL_MILHAR := ""; END IF;
```

Verifica a necessidade de "," após as unidades

```
IF length(trim(vUNIDADE))::integer<>0 and
length(trim(vCENTAVO))::integer<>0 THEN
vCOMPL_UNIDADE := ", "; ELSE
vCOMPL_UNIDADE := ""; END IF;
vRESULTADO:= vMILHAO || vCOMPL_MILHAO || vMILHAR
|| vCOMPL_MILHAR ||vUNIDADE || vCOMPL_UNIDADE ||
vCENTAVO;
```

Retorna o resultado

```
return vRESULTADO; END;
' language plpgsql;
```

Exemplos de uso:

```
samples=# select extenso(1.01);
extenso
UM REAL, UM CENTAVO
samples=# select extenso(12.05);
extenso
DOZE REAIS, CINCO CENTAVOS
samples=# select extenso(44.12);
extenso
QUARENTA E QUATRO REAIS, DOZE CENTAVOS
samples=# select extenso(99.24);
extenso
NOVENTA E NOVE REAIS, VINTE E QUATRO CENTAVOS
samples=# select extenso(100.50);
extenso
CEM REAIS, CINQUENTA CENTAVOS
samples=# select extenso(123.90);
extenso
CENTO E VINTE E TRES REAIS, NOVENTA CENTAVOS
samples=# select extenso(200.16);
extenso
DUZENTOS REAIS, DESESSEIS CENTAVOS
samples=# select extenso(999.00);
extenso
NOVECENTOS E NOVENTA E NOVE REAIS
samples=# select extenso(999020020.00);
extenso
NOVECENTOS E NOVENTA E NOVE
MILHOES, VINTE MIL, VINTE REAIS
samples=# select extenso(123456345.34);
extenso
CENTO E VINTE E TRES MILHOES,
QUATROCENTOS E CINQUENTA E SEIS MIL,
TREZENTOS E QUARENTA E CINCO REAIS,
TRINTA E QUATRO CENTAVOS
```

Referências – www.dbexperts.net o Momjian, Bruce. PostgreSQL - Introduction and Concepts

PCM