Indexação de Textos com o Tsearch 2

Busca bonita e veloz

Gaston THAUVIN – www.sxc.hu

Vamos realizar buscas em um portal jornalístico fictício para verificar a necessidade da indexação de textos e de outros recursos oferecidos pelo módulo Tsearch2 do PostgreSQL.

por Nabucodonosor Coutinho Costa

Se você já tem o ambiente de desenvolvimento LAMP instalado e configurado, e tem também instalado o *PostgreSQL* em sua máquina, só nos falta configurar o *PHP* para carregar a extensão *php_psql.so*.

Se você instalou o PHP a partir do código-fonte, é necessário rodar novamente o `configure` com a opção `--with-pgsql` e informar o caminho para a instalação do PostgreSQL, ou instalar pelo gerenciador de pacotes da sua distribuição. Em seguida, você deve editar seu arquivo `php.ini`, que se localiza em `/etc` ou `/usr/local/etc`, dependendo da forma de instalação usada, e descomentar a linha:

```
extension=php_pgsql.so
```

Com isso, seu ambiente de desenvolvimento estará pronto. Agora, é só reiniciar o Apache e verificar se a extensão foi carregada. Para isso, criaremos a boa e velha página de informações com o seguinte código:

```
<?php
phpinfo();
?>
```

Salve essa página com o nome `teste.php` dentro do diretório que está sendo servido pelo Apache. Se você se esqueceu, ou não sabe qual é esse diretório, veja a diretiva *DocumentRoot* em seu `/etc/httpd.conf`. A partir de agora, trataremos essa pasta apenas como *DocumentRoot*.

Agora vamos acessar nossa página de teste pelo navegador, e conferir se o PHP carregou o módulo de acesso ao PostgreSQL. Abra seu navegador e acesse `http://localhost/teste.php`, e verifique as informações sobre o módulo, que devem ser semelhantes à **figura 1**.

Nossa base de dados

Vamos agora criar uma base de dados chamada `linux_magazine`, na qual criaremos a nossa tabela de notícias.

```
(coutinho@coitada - ~ @21:26:10)
0: psql -U postgres
Welcome to psql 8.1.4, the PostgreSQL
➥ interactive terminal.

Type: \copyright for distribution terms
\h for help with SQL commands
\? for help with psql commands
\g or terminate with semicolon
➥ to execute query
\q to quit

postgres=# create database linux_magazine;
CREATE DATABASE
postgres=#

Conectando ao banco de dados
➥ linux_magazine:

\c linux_magazine;
You are now connected to database
➥ "linux_magazine".
linux_magazine=#
```

Agora vamos criar a tabela de notícias executando a seguinte definição SQL:

```
linux_magazine=# CREATE TABLE news (
linux_magazine(# id serial primary key,
linux_magazine(# title varchar,
linux_magazine(# content text
linux_magazine(# );
CREATE TABLE
linux_magazine=#
```

Nossa tabela fica assim:

```
linux_magazine=# \d news
Table "public.news"
Column | Type | Modifiers
---------+------------------+------------
id | integer | not null default
➥ nextval('news_id_seq'::regclass)
title | character varying |
content | text |
Indexes:
"news_pkey" PRIMARY KEY, btree (id)
```

Para os marinheiros de primeira viagem no PostgreSQL, o primeiro campo da nossa tabela é do tipo inteiro e será numerado de forma seqüencial e automática pelo PostgreSQL.

Precisamos agora inserir alguns dados em nossa tabela, para depois simularmos nosso sistema de busca.

```
INSERT INTO news (title, content) VALUES (
'Python e PostgreSQL',
'É possível escrever funções em python
➥ dentro do PostgreSQL'
);
```

```
INSERT INTO news (title, content) VALUES (
'Ruby',
'Após o "Ruby on Rails" a linguagem Ruby
➥ vem se popularizando'
);

INSERT INTO news (title, content) VALUES (
'O elefante que fala sua língua',
'No postgreSQL é possível desenvolver
➥ funções em c, python, java, ruby, php,
➥ perl, tcl e em muitas outras'
);
```

Com isso, temos nossa tabela suficientemente populada para começarmos a brincadeira.

A página de busca

Antes de iniciarmos, é importante saber que o código HTML no **exemplo 1** não é uma apologia ao esquecimento das boas práticas de programação; ele apenas tem o objetivo de ser o mais enxuto e compreensível possível. Salve esse arquivo com o nome `index.php` na pasta DocumentRoot. Para testar nossa busca, abra seu navegador e acesse `http://localhost/`.

Milagres do Tsearch2

Ainda podemos melhorar a velocidade da nossa busca. Com o módulo *Tsearch2* ([1]), podemos indexar os textos de nossas notícias para que sejam encontrados mais rapidamente pelo motor de busca do PostgreSQL. Outras vantagens do uso do Tsearch2 são o uso de funções que ajudam a fazer a classificação de relevância dos resultados encontrados, de acordo com o que foi pesquisado.

Outra função muito interessante é a `headline`, que mostra um trecho do texto onde a palavra pesquisada foi encontrada, e ainda destaca-a em negrito dentro do trecho de amostra (**figura 2**), como acontece em sites de busca como o Google [2].

Instalação e configuração

Se você não instalou o PostgreSQL a partir do código-fonte, deve procurar o módulo Tsearch2 adequado para o tipo de instalação da sua distribuição. Se você gosta de obter o máximo em performance e ter flexibilidade para instalar novos módulos, patches etc, você provavelmente instalou a partir dos fontes. Nesse caso, existe uma pasta `contrib` junto com os fontes do PostgreSQL.

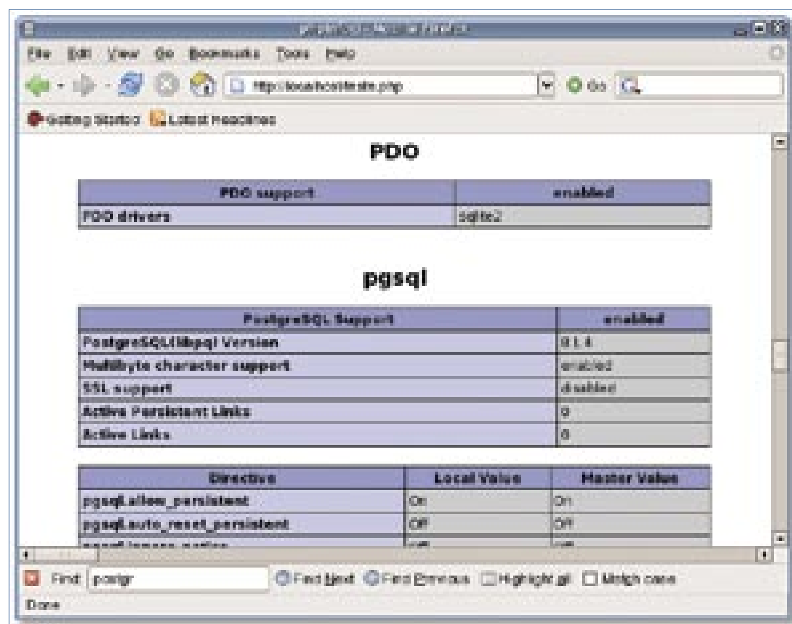

Figura 1 Página de teste de acesso ao *PostgreSQL* pelo *PHP*.

Exemplo 1: PHP com PostgreSQL

```
01 index.php
02
03 <html>
04
05 <head>
06 <title>Utilizando PHP com PostgreSQL</title>
07 </head>
08
09 <body>
10
11 <h1>Testando uma busca no banco de dados</h1>
12
13 <form method="post">
14 Buscar: <input type="text" name="q"> <input type="submit" value="Buscar">
15 </form>
16
17 <?php
18
19 if ($_POST["q"]) {
20
21 echo "<br/><h2>Resultado da busca</h2>";
22
23 $conn = pg_connect("host=localhost dbname=linux_magazine user=postgres");
24 $sql = "select * from news where content ilike '%" . $_POST["q"] . "%'";
25 $dados = pg_query($sql);
26
27 for ($i=0;$i<pg_num_rows($dados);$i++)
28 {
29 echo pg_result($dados,$i,"content");
30 echo "<hr/>";
31 }
32
33 $conn = null;
34 }
35
36 ?>
37
38 </body>
39 </html>
```

Na pasta contrib, temos vários módulos não oficiais, frutos de contribuições voluntárias. Temos desde utilitários para benchmarks até módulos para migração de dados. É lá que iremos encontrar uma pasta chamada tsearch2.

Para instalar o Tsearch2, é só acessar a pasta, compilar e instalar:

```
cd /usr/src/postgresql-8.1.4/contrib/
➥tsearch2
make
su
make install
```

Depois disso, o Tsearch2 já estará instalado na sua máquina.

O Tsearch é basicamente uma biblioteca de funções escritas em C e compiladas como *Shared Object* (.so). Quando mandamos instalar o Tsearch2, essa biblioteca foi colocada dentro da pasta lib abaixo da sua instalação do PostgreSQL. No meu caso, essa pasta é /usr/local/pgsql/lib. Com isso, as funções da biblioteca do Tsearch ficam disponíveis para o uso no PostgreSQL. No entanto, ainda é necessário que essas funções sejam registradas no banco de dados onde pretendemos usar suas funcionalidades.

O Tsearch também é composto por alguns componentes personalizados comuns de bancos de dados, como tabelas, tipos e operadores.

Para registrar as funções em seu banco de dados e criar os demais objetos, o Tsearch2 coloca um arquivo com extensão .sql dentro da pasta share/contrib da instalação do PostgreSQL.

```
# ls /usr/local/pgsql/share/contrib/
english.stop russian.stop tsearch2.sql
➥ untsearch2.sql
```

Para instalar o Tsearch2 no banco de dados linux_magazine que criamos, usaremos o arquivo .sql fornecido pelo Tsearch2 para registrar as funções e criar todos os objetos necessários para o funcionamento da biblioteca.

```
linux_magazine=# \i /usr/local/
➥pgsql/share/contrib/tsearch2.sql
```

Verifiquemos se deu certo...

```
linux_magazine=# \d
List of relations
Schema | Name | Type | Owner
--------+-------------+----------+--------
public | news | table | postgres
public | news_id_seq | sequence | postgres
public | pg_ts_cfg | table | postgres
public | pg_ts_cfgmap | table | postgres
public | pg_ts_dict | table | postgres
public | pg_ts_parser | table | postgres
(6 rows)
```

Vemos na lista quatro tabelas com o prefixo pg_ts_. Essas são as tabelas de configuração do Tsearch2. Também podemos usar os comandos \dT, \do e \df para listar também os tipos, operadores e funções respectivamente, criados pelo arquivo tsearch2.sql.

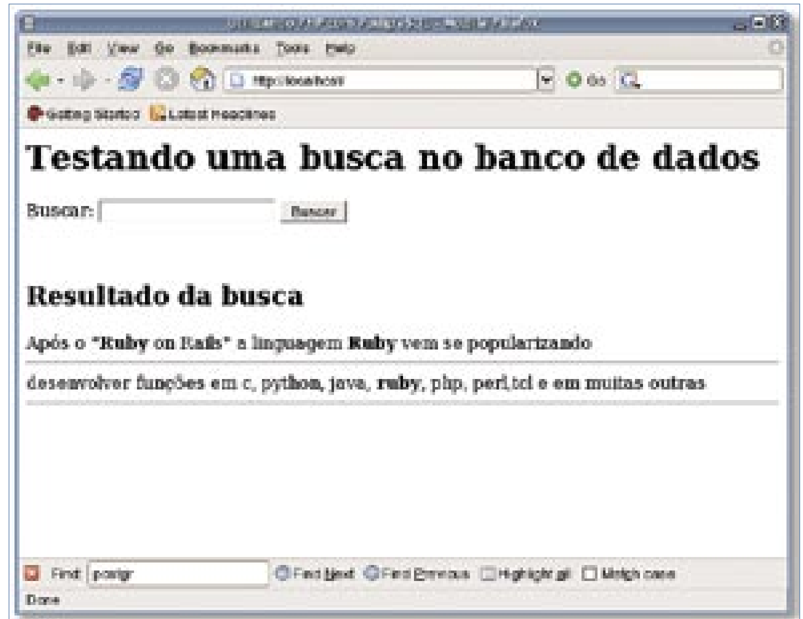

Figura 2 Resultado mais nítido com o destaque dos trechos de resultado.

Exemplo 2: Função para popular a tabela

```
01 create or replace function popula_texto(int4) returns int4 as $$
02 declare
03 reg record;
04 qtde alias for $1;
05 i int4;
06 begin
07
08 i := 1;
09 for reg in select * from news limit 1 offst 0 LOOP
10 while i <= qtde LOOP
11 insert into news (title,content,vectors)
12 values (reg.title,reg.content, reg.vectors);
13 raise notice 'linha %', i;
14 i := i + 1;
15 end loop;
16 end loop;
17
18 return i - 1;
19
20 end;
21 $$ language 'plpgsql';
```

Quadro 1: Criando um campo para a indexação

```
linux_magazine=# alter table news add column vectors tsvector; ALTER TABLE
linux_magazine=# create index my_full_text_index on news using gist(vectors); CREATE INDEX
```

Testando, configurando e testando

Para ver se o Tsearch2 está funcionando corretamente no banco de dados, podemos tentar usar uma de suas funções e ver se ela retorna algo que não seja um erro:

```
linux_magazine=# select
➥ to_tsquery('teste');
ERROR: could not find tsearch config
➥ by locale
```

Isso pode não funcionar, como foi meu caso. A maioria dos usuários terá que configurar o *local e* padrão do Tsearch2 **antes** de usá-lo, e ele deve ser o mesmo usado pelo banco de dados. O comando `show` exibe o local e padrão:

```
linux_magazine=# show lc_ctype;
lc_ctype
----------
en_US
(1 row)
```

Se o do Tsearch2 não for o mesmo, podemos corrigi-lo com:

```
linux_magazine=# update pg_ts_cfg set
➥ locale = 'en_US' where ts_name =
➥ 'default';
```

E testá-lo novamente:

```
linux_magazine=#
➥ select to_tsquery('teste');
to_tsquery
-----------
'test'
(1 row)
```

Agora o retorno não foi um erro. Isso quer dizer que o Tsearch2 está instalado e configurado corretamente.

Índices full text

A primeira grande vantagem do uso do Tsearch2 é que ele traz uma funcionalidade que não está presente no kernel do PostgreSQL: um poderoso sistema de indexação de textos.

Não é o objetivo deste artigo dar uma aula sobre o funcionamento de índices no PostgreSQL, mas vou explicar como funciona a indexação do Tsearch2. Primeiramente, criaremos um campo extra do tipo `ts_vector`, que é um dos tipos criados pelo `tsearch`. Esse campo irá guardar as palavras-chave do nosso texto em forma de vetor, que é uma estrutura rapidamente legível pelo PostgreSQL. Nosso índice trabalhará em cima desse novo campo, e não do campo com o texto real, o que nos economiza mais tempo ainda durante as buscas. O **quadro 1** demonstra a criação do campo para indexação.

Com as duas linhas acima, adicionamos um campo chamado `vectors` à tabela `news`, e em seguida criamos um índice `gist` para esse campo. A nova estrutura de nossa tabela agora é essa:

```
linux_magazine=# \d news
Table "public.news"
Column | Type | Modifiers
---------+------------------+------------
id | integer | not null default
➥ nextval('news_id_seq'::regclass)
title | character varying |
content | text |
vectors | tsvector |
Indexes:
"news_pkey" PRIMARY KEY, btree (id)
"my_full_text_index" gist (vectors)
```

Agora, vamos fazer o que seria nosso trabalho de indexação dessa tabela, depois de o Tsearch2 pegar as palavras-chave do campo `content` e colocar no campo `vectors`.

A função `to_tsvector()` do Tsearch2 é usada para converter cadeias de caracteres para o tipo `tsvector`. Vejamos:

```
linux_magazine=# select to_tsvector('meu
➥ nome eh coutinho');
to_tsvector
-------------------------------------
'eh':3 'meu':1 'nome':2 'coutinho':4
```

A função `to_tsvector()` cria um *hash* com as palavras do texto. É assim que pegaremos as palavras do campo `content` para colocá-las no campo `vectors`, que é o campo que indexamos.

```
linux_magazine=# update news set
➥ vectors=to_tsvector(content); UPDATE 3
```

Análise de desempenho

Para ver se isso realmente funciona, o **exemplo 2** mostra uma pequena função em *plpgsql* para popular a tabela automaticamente.

Agora, tenho mais de 700 registros em minha tabela `news`. Para quem desejar usar essa função, é necessário instalar a plpgsql no banco com o comando `linux_magazine=# \!createlang -U postgres plpgsql`.

Em seguida, chame a função, passando como argumento o número de registros a serem gerados. No exemplo abaixo, são utilizados dez registros:

```
linux_magazine=# select *
➥ from popula_texto(10);
NOTICE: linha 1
NOTICE: linha 2
NOTICE: linha 3
NOTICE: linha 4
NOTICE: linha 5
NOTICE: linha 6
NOTICE: linha 7
NOTICE: linha 8
NOTICE: linha 9
NOTICE: linha 10
popula_texto
-------------
10
```

Após atingir algo em torno de 500 registros na sua tabela, é possível ver a diferença entre fazer uma busca comum com `like` e uma utilizando o nosso índice:

```
linux_magazine=# select count(id) from
➥ news;
```

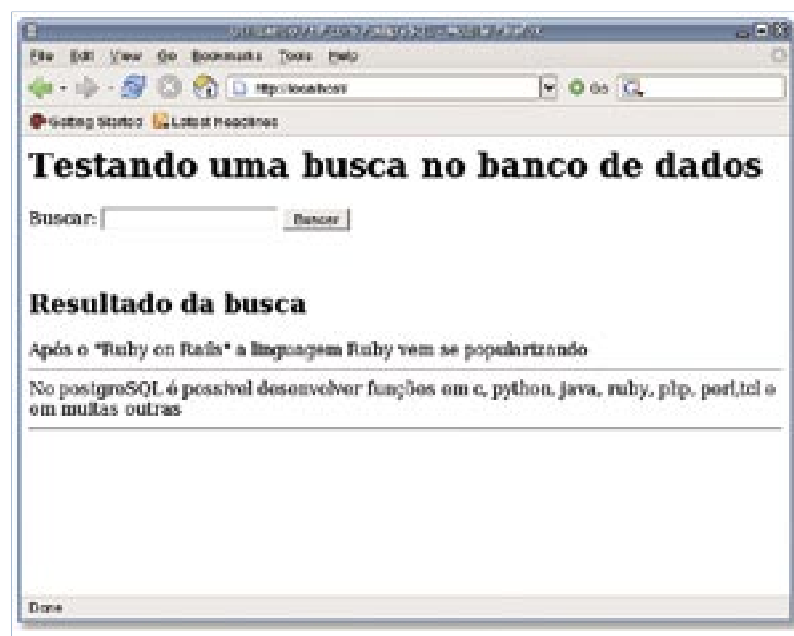

Figura 3 Resultado da busca por "python ruby" em nosso sistema.

```
count
-------
1376
(1 row)
```

Nesse exemplo, a tabela `news` tinha 1376 registros. Vamos ver o resultado do comando `explain` para uma busca comum com `like`:

```
linux_magazine=# explain select * from
➥ news where content like '%postgresql%';
QUERY PLAN
----------------------------------------
Seq Scan on news (cost=0.00..19.71 rows=1
➥ width=100)
Filter: (content ~~ '%postgresql%'::text)
(2 rows)
```

Agora, a saída do comando `explain` para uma consulta usando a indexação:

```
linux_magazine=# explain select * from
➥ news where vectors @@
➥ to_tsquery('postgresql');
QUERY PLAN
----------------------------------------
Index Scan using my_full_text_index on
➥ news (cost=0.00..4.91 rows=1 width=100)
Index Cond: (vectors @@
➥ '''postgresql'''::tsquery)
Filter: (vectors @@
➥ '''postgresql'''::tsquery)
(3 rows)
```

Vejam que a consulta indexada, visivelmente mais rápida, não usou o operador `like` com o campo `content`, e sim o operador `@@`, para verificar a ocorrência da palavra-chave `php` no campo indexado `vectors`. Podemos ver ainda, de acordo com a segunda linha do `explain`, que a condição de busca foi indexada.

Exemplo 3: Nossa página de busca com visual melhorado

```
01 <head>
02 <title>Utilizando PHP com PostgreSQL</title>
03 </head>
04
05 <body>
06
07 <h1>Testando uma busca no banco de dados</h1>
08
09 <form method="post">
10 Buscar: <input type="text" name="q"> <input type="submit" value="Buscar">
11 </form>
12
13 <?php
14
15 if ($_POST["q"]) {
16
17 echo "<br/><h2>Resultado da busca</h2>";
18
19 $conn = pg_connect("host=localhost dbname=linux_magazine user=postgres");
20
21 $sql = "select";
22 $sql += "title,";
23 $sql += "headline(content,to_tsquery('" . $_POST["q"] . "') as content," ;
24 $sql += "rank(vectors, to_tsquery('" . $_POST["q"] . "')) as rank";
25 $sql += "from news";
26 $sql += "where content ilike '%" . $_POST["q"] . "%'";
27 $sql += "order by rank desc";
28
29
30 $dados = pg_query($sql);
31
32
33 for ($i=0;$i<pg_num_rows($dados);$i++)
34 {
35 echo "<p><b>" . pg_result($dados,$i,"title") . "</b><br/>";
36 echo pg_result($dados,$i,"content");
37 echo "</p>";
38 }
39
40 $conn = null;
41 }
42
43 ?>
44
45 </body>
46 </html>
```

A função stat

Com o uso da função `stat(text)`, é possível obter estatísticas das palavras por número de documentos ou por número de ocorrências por documento. Vejamos como listar as três palavras mais presentes em números de documentos:

```
linux_magazine=# select * from
➥ stat('select vectors from news')
➥ order by ndoc desc limit 3;
word | ndoc | nentry
-----------+------+--------
postgresql | 35 | 53
python | 34 | 34
php | 5 | 296
(3 rows)
```

Segundo a saída do comando acima, a palavra postgresql aparece em 35 documentos (registros), totalizando 53 ocorrências distribuídas nesses 35 documentos.

A função rank

A função `rank` gera um número que representa a classificação do registro na nossa consulta. Por exemplo, se pesquisarmos a palavra `postgresql`, e um registro tiver apenas uma ocorrência dessa palavra, ele recebe um valor *x* de classificação. Se outro registro possuir uma quantidade maior de ocorrências de `postgresql`, esse outro registro recebe uma classificação maior.

Com esse recurso, podemos implementar uma funcionalidade semelhante à utilizada pelo Google, ordenando os registros pela classificação e facilitando a vida do usuário.

Vamos ver agora um exemplo da função de classificação `rank`:

```
select
title,
rank(vectors, to_tsquery('ruby'))
➥ as rank
from news
where vectors
➥ @@ to_tsquery('ruby');

title | rank
------------------------------+---------
Ruby | 0.0759909
O elefante que fala sua língua | 0.0607927
PostgreSQL README | 0.0607927
(3 rows)
```

Veja que passamos para a função `rank` dois argumentos. O primeiro foi o campo `vectors`, e o segundo foi nossa consulta, a mesma que usamos no `where`, e a mágica foi feita; os registros vieram ordenados

pela classificação, e a notícia com o título *Ruby* veio em primeiro, pois sua classificação é maior que a das demais.

A função headline

A função `headline` é bem interessante, e seu resultado sozinho já tornaria o Tsearch2 um produto bastante atraente. Ela traz um trecho do texto contendo a palavra encontrada em cada documento. Esse recurso já é bem conhecido por qualquer pessoa que utilize sites de busca na Internet.

Vamos a um pequeno exemplo:

```
select
title,
headline(content,to_tsquery('ruby')),
rank(vectors, to_tsquery('ruby')) as rank
from news
where vectors @@ to_tsquery('ruby')
order by rank;
```

Essa consulta vai fazer a mesma coisa que a anterior, e vai aplicar a função `headline` no campo `content`. Testando-a diretamente no psql, o resultado vem um tanto feio, mas vamos levá-la para dentro de nossa página PHP com o formulário de busca para vermos como o resultado fica extremamente agradável.

Melhorando a busca

Vamos agora melhorar nossa feiosa página de busca. Para isso, vamos alterar o seguinte trecho de código para implementar a consulta com o Tsearch2. Nossa página deve ficar assim:

Agora vamos acessar nossa página de busca novamente e, ao pesquisar por alguma palavra, vamos ver a diferença na renderização da página, principalmente o trecho do texto contendo em negrito a palavra que procuramos. Como já tínhamos implementado antes, nossa busca continua classificada.

Mais fácil

Podemos ainda escrever uma pequena função para facilitar de vez a vida do programador na hora de escrever as consultas SQL para o Tsearch2. Mas antes de criá-la, criaremos nosso tipo de dados personalizado, pois nossa função irá retornar dados nesse formato.

```
CREATE TYPE tp_search as
(
id int4,
title varchar,
content text,
rank float4
);
```

Agora vamos fazer a função que executará aquela busca e nos retornará os dados no formato especificado pelo tipo `tp_search`:

```
CREATE OR REPLACE FUNCTION
➥ my_search(varchar)
RETURNS SETOF tp_search AS
$$
SELECT
id,
title,
headline(content,my_query) as content,
rank(vectors,my_query) as rank
FROM
news,
to_tsquery($1) AS my_query
WHERE
vectors @@ my_query
ORDER BY
rank DESC;
$$ LANGUAGE 'SQL';
```

Criada a nossa função, podemos usar um SQL mais simples para fazer nossa busca com o Tsearch2:

```
select * from my_search('ruby');
```

Fazendo a substituição no HTML de nossa página, teremos uma redução considerável no nosso código, bem no trecho que conecta ao banco de dados e faz a pesquisa:

```
...
$conn = pg_connect("host=localhost
➥ dbname=linux_magazine user=postgres");
$sql = "select * from my_search('" .
➥ $_POST["q"] . "')";
 $dados = pg_query($sql);
...
```

...E além

O Tsearch2 pode ir muito além. Nós ainda não fomos muito longe, já que nossa busca nem sequer aceita que digitemos mais de uma palavra para ser pesquisada. A **figura 3** mostra o que acontece se você mandar pesquisar por mais de uma palavra em sua página de busca. Nesse caso, eu digitei `python ruby` na caixa de texto de pesquisa.

Para fazermos pesquisas mais complexas, envolvendo, por exemplo, mais de uma palavra, precisaremos fazer uso dos operadores do Tsearch2. Para informar a ele se desejamos, por exemplo, resultados com uma das palavras ou com as duas palavras.

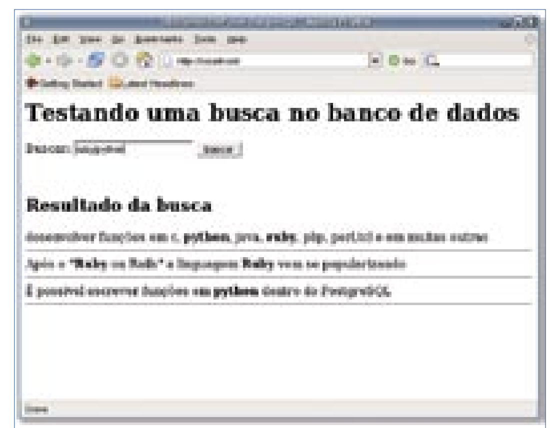

Figura 4 Uso do operador booleano `OR`.

Quando digitei a frase `python ruby` na minha página de busca, o SQL enviado ao PostgreSQL foi o seguinte:

```
select * from my_search('python ruby');
```

Vamos relembrar o que nossa função faz na realidade. Ela pega a cadeia de caracteres passada como parâmetro e conta ao SQL qual final será executado no banco de dados.

Vejamos o SQL montado por nossa função:

```
SELECT
id,
title,
headline(content,my_query) as content,
rank(vectors,my_query) as rank
FROM
news,
to_tsquery($1) AS my_query
WHERE
vectors @@ my_query
ORDER BY
rank DESC;
```

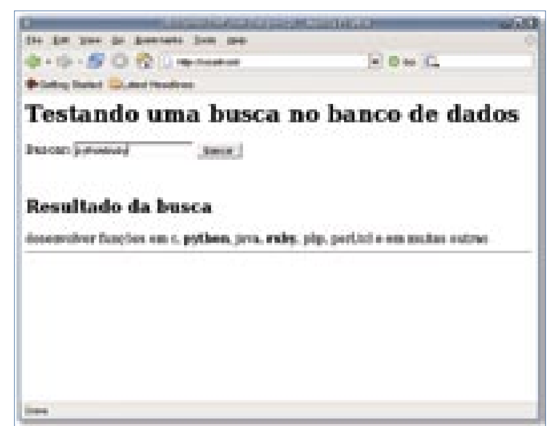

Figura 5 Resultado de uma busca com o uso do operador booleano `AND`.

No trecho `to_tsquery($1)`, a variável `$1` é substituída pela cadeia de caracteres que enviamos. No meu caso, esses comandos SQL foram enviados ao banco dados da seguinte forma:

```
SELECT
id,
title,
headline(content,my_query) as content,
rank(vectors,my_query) as rank
FROM
news,
to_tsquery('python ruby') AS my_query
WHERE
vectors @@ my_query
ORDER BY
rank DESC;
```

O Tsearch2 gera um erro com essa busca, pois o parâmetro passado não está no formato esperado.

Operadores no Tsearch

Para enviarmos consultas complexas ao Tsearch2, é necessário fazer uso de seus "operadores especiais", os quais usamos dentro do parâmetro que passamos a ele. Por exemplo, ao digitar `python ruby` na caixa de texto de pesquisa, eu deveria ter levado em conta se eu desejava que o resultado contivesse **ambas** as palavras ou qualquer uma delas.

Vamos supor que queiramos pesquisar textos que contenham qualquer uma das palavras que digitei. Para isso, usaremos o operador `OR` do Tsearch2, que é representado pelo caractere `|` (barra vertical, ou *pipe*). Para procurar textos com a palavra *Ruby* ou com a palavra *Python*, teríamos que digitar `ruby|python` (**figura 4**).

Observe que com essa consulta eu obtive registros que continham a palavra *Ruby* **ou** a palavra *Python*.

A **figura 5** ilustra a situação do operador booleano `AND` do Tsearch2. Nesse caso, para realizar a consulta, digitei `python&ruby`. Com isso, obtive apenas os registros que contêm a palavra *Python* **e** a palavra *Ruby*.

Operadores na página

Vamos mexer novamente em nossa página HTML, dessa vez para inserir no formulário de busca controles através dos quais o usuário possa determinar o tipo de busca que está fazendo, e para que assim nosso sistema de busca possa saber qual operador do Tsearch2 deve ser usado. Nosso formulário ficará assim:

```
...
<form method="post">
Buscar: <input type="text" name="q">
<input type="submit" value="Buscar"><br/>

<input type="radio" name="op"
➥ value="or">Uma ou outra
<input type="radio" name="op"
➥ value="and">Todas as palavras
</form>
...
```

O trecho que foi adicionado ao formulário acrescenta à página dois botões de rádio, que permitem ao usuário definir o tipo de operação booleana a ser usada na consulta.

Agora precisamos alterar o código PHP para substituir os espaços em branco pelo operador correto do Tsearch2 na cadeia de caracteres a ser pesquisada:

```
...
$conn = pg_connect("host=localhost
➥ dbname=linux_magazine user=postgres");

if ($_POST["op"] == "or") {
$query = str_replace(' ','|',$_POST["q"]);
} else {
$query = str_replace(' ','&',$_POST["q"]);
}

$sql = "select * from my_search('" .
➥ $query . "')";
$dados = pg_query($sql);
...
```

O resultado final de nossa página seria igual ao da **figura 6**.

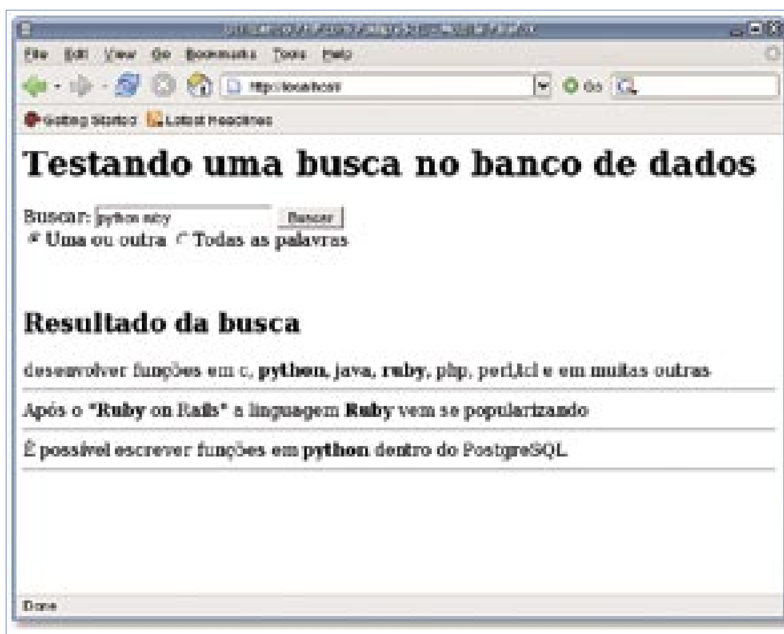

Figura 6 Nossa página de busca, com botões de rádio que permitem definir o tipo de busca.

Conclusão

Esse minissistema de busca ilustrou como o Tsearch2 traz, além da indexação de textos, diversos outros recursos, os quais tornam o PostgreSQL ainda mais interessante para o desenvolvimento Web.

Além do que foi apresentado neste artigo, o Tsearch2 possui ainda recursos de dicionário de idiomas e de sinônimos, que podem ser usados para tornar seu sistema de busca ainda mais útil no dia-a-dia, mais interessante e mais parecido com grandes sistemas de busca como Google e Altavista. ■

O autor

Nabucodonosor Coutinho Costa é administrador de bancos de dados PostgreSQL e pode ser contactado pelo email coutinho.php@gmail.com

Mais Informações

[1] Página do Tsearch2: http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/

[2] Google :http://www.google.com