Recherche en plein texte avec PostgreSQL

Stéphane Bortzmeyer
<stephane+blog@bortzmeyer.org>

Première rédaction de cet article le 11 décembre 2008

https://www.bortzmeyer.org/postgresql-recherche-texte.html

Traditionnellement, les SGBD ne traitaient que des données simples et courtes, comme le nom d'un employé ou son salaire. Les recherches portaient donc sur la totalité du champ. Désormais, il est de plus en plus fréquent de voir un SGBD utilisé pour stocker des textes relativement longs (des articles d'un blog, par exemple) et la recherche dans ces textes devient très longue et peu pratique. Elle nécessite l'indexation et des outils de recherche spécialisés. Qu'en est-il sur PostgreSQL ?

Pendant longtemps, le moteur de recherche plein texte de PostgreSQL était tsearch2 <http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/>. Distribué dans le répertoire contrib/ de PostgreSQL, il devait être installé séparément (ou via un paquetage du système d'exploitation considéré, par exemple databases/postgresql82-tsearch2 sur NetBSD). Désormais, depuis la version 8.3 <http://www.postgresql.org/docs/8.3/static/release-8-3.html>, il est intégré complètement à PostgreSQL, et pourvu d'une excellente documentation <http://www.postgresql.org/docs/current/interactive/textsearch.html>, très détaillée.

Le principe est de couper le texte en élements lexicaux, puis en lexèmes, qui sont des versions **normalisées** des éléments lexicaux. Cela se fait avec la fonction to_tsvector :

```
essais=> SELECT to_tsvector('les gros chats mangent les rats maigres');
                        to_tsvector
-----------------------------------------------------------
 'le':1,5 'rat':6 'chat':3 'gros':2 'maigr':7 'mangent':4
(1 row)
```

Ici, elle a analysé la phrase, trouvé six mots (dont un répété) et réduit (normalisé) ceux qu'elles pouvaient. Ce processus dépend de la langue et donne parfois des résultats surprenants (« mangent » n'a pas été normalisé en « manger »).

On cherche ensuite dans ces éléments lexicaux avec la fonction de correspondance, @@. Elle prend comme paramètres un vecteur (tsvector) comme celui ci-dessus et une requête, créée avec to_tsquery :

```
essais=> SELECT to_tsquery('chat|rat');
   to_tsquery
----------------
 'chat' | 'rat'
(1 row)
```

Le langage des requêtes (vous avez sans doute déjà deviné que | veut dire OU) est évidemment documenté <http://www.postgresql.org/docs/current/interactive/datatype-textsearch.html#DATATYPE-TSQUERY>.

En combinant requête, vecteur de mots et opérateur de correspondance, on peut faire une recherche complète :

```
essais=> SELECT to_tsvector('les gros chats mangent les rats maigres')
                        @@ to_tsquery('chat|rat');
 ?column?
----------
 t
(1 row)

essais=> SELECT to_tsvector('on ne parle pas de ces animaux ici')
                        @@ to_tsquery('chat|rat');
 ?column?
----------
 f
(1 row)
```

On a trouvé un résultat dans le premier cas et zéro dans le second.

Bien sûr, taper de telles requêtes à la main est plutôt pénible, on a donc intérêt à créer ses propres fonctions.

L'analyse d'une phrase en mots et la normalisation de ces derniers dépend de la langue. Il y a un paramètre à `to_tsvector` et `to_tsquery` qui indique une configuration ('french' par défaut, sur mon site). Voyons quelles configurations a une installation typique de PostgreSQL 8.3 (ici, le paquetage Debian) :

```
essais=> \dF
              List of text search configurations
   Schema   |    Name    |              Description
------------+------------+---------------------------------------
...
 pg_catalog | finnish    | configuration for finnish language
 pg_catalog | french     | configuration for french language
 pg_catalog | german     | configuration for german language
...

essais=> show default_text_search_config;
 default_text_search_config
----------------------------
 pg_catalog.french
(1 row)
```

Prenons maintenant un exemple réel, un moteur de recherche dans la liste des RFC. On commence par créer la base :

```
-- Un RFC a un numéro, un titre et le corps, du texte brut
CREATE TABLE Rfcs (id SERIAL,
   inserted TIMESTAMP default now(),
   num INTEGER,
   title TEXT,
   body TEXT);

-- Créer son propre type facilite l'utilisation de la fonction search()
CREATE TYPE Results AS (num INTEGER, title TEXT,
                        body TEXT, rank REAL);

-- Les RFC sont en anglais donc on utilise systématiquement la
-- configuration 'english'
CREATE FUNCTION search (TEXT) RETURNS SETOF Results AS
   'SELECT num, title,
           body, ts_rank_cd(to_tsvector(''english'', title || body),
                            to_tsquery(''english'',$1))
      FROM Rfcs
         WHERE to_tsvector(''english'', title || body) @@ to_tsquery(''english'',$1)
         ORDER BY ts_rank_cd(to_tsvector(''english'', title || body),
                             to_tsquery(''english'',$1)) DESC;'
  LANGUAGE SQL;
```

La fonction `search()` appelle une fonction `ts_rank_cd` qui calcule la pertinence d'un texte par rapport à une requête. Il faut aussi remarquer la concaténation des champs (`title || body`) qui indique qu'on cherche dans les deux colonnes. Testons `search()` :

```
essais=> SELECT num,title,rank FROM search('dnssec') ORDER BY rank DESC;
 num  |                        title                        | rank
------+-----------------------------------------------------+------
 4035 | Protocol Modifications for the DNS Security ...     | 10.1
 3130 | Notes from the State-Of-The-Technology: DNSSEC      |  7.4
 4641 | DNSSEC Operational Practices                        |  7.1
...
```

La recherche peut être plus compliquée, par exemple si on cherche les RFC qui parlent de DNSSEC **et** LDAP (notons que les pertinences sont bien plus faibles) :

```
essais=> SELECT num,title,rank FROM search('dnssec & ldap') ORDER BY rank DESC;
 num  |                     title                       |    rank
------+-------------------------------------------------+-------------
 5000 | Internet Official Protocol Standards            |    0.111842
 4238 | Voice Message Routing Service                   |    0.0170405
 4513 | LDAP: Authentication Methods and Security ... |    0.00547112
...
```

Chercher dans la totalité de la base à chaque fois peut être assez long. Les moteurs de recherche emploient typiquement l'indexation pour accélerer la recherche. PostgreSQL dispose de plusieurs types d'index <http://www.postgresql.org/docs/8.3/static/textsearch-indexes.html>. Pour favoriser le temps de recherche (au détriment du temps d'indexation), j'utilise les index GIN :

```
CREATE INDEX rfcs_idx ON Rfcs USING gin(to_tsvector('english', title || body));
```

Avec ces index, les recherches sont bien plus rapides. En effet, sans index, il faut balayer toute la table comme l'indique `EXPLAIN` :

```
essais=> EXPLAIN SELECT id,num FROM Rfcs   WHERE to_tsvector('english', title || body)  @@ to_tsquery('dnsse
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Seq Scan on rfcs  (cost=0.00..186.11 rows=5 width=8)
   Filter: (to_tsvector('english'::regconfig, body) @@ to_tsquery('dnssec'::text))
(2 rows)
```

(`Seq Scan` est "*sequential scan*".) Avec l'index, la recherche se fait d'abord dans l'index :

```
essais=> EXPLAIN SELECT id,num FROM Rfcs   WHERE to_tsvector('english', title || body)  @@ to_tsquery('dnsse
                                 QUERY PLAN
-----------------------------------------------------------------------------------------------
 Bitmap Heap Scan on rfcs  (cost=40.70..57.34 rows=5 width=8)
   Recheck Cond: (to_tsvector('english'::regconfig, body) @@ to_tsquery('dnssec'::text))
   ->  Bitmap Index Scan on rfcs_idx  (cost=0.00..40.70 rows=5 width=0)
         Index Cond: (to_tsvector('english'::regconfig, body) @@ to_tsquery('dnssec'::text))
(4 rows)
```

Le programme qui prend l'ensemble des RFC et les met dans la base est (en ligne sur `https://www.bortzmeyer.org/files/index-rfcs.py`). Un autre exemple pratique est le moteur de recherche de mon blog </search>, documenté dans « Mise en œuvre du moteur de recherche de ce blog <https://www.bortzmeyer.org/moteur-recherche-wsgi.html>.

Sur le même sujet et dans la même langue, on peut lire le très détaillé « Recherche plein texte avec PostgreSQL 8.3 <http://www.dalibo.org/glmf111_recherche_plein_texte_avec_postgresql> ».