Recherche plein texte avec PostgreSQL 8.3

Cet article, écrit par Guillaume Lelarge, a été publié dans le magazine GNU/Linux Magazine France, numéro 111 (Décembre 2008). Il est disponible maintenant sous licence Creative Commons.

Un peu de théorie

PostgreSQL, comme tout moteur de bases de données, sait faire de la recherche sur des champs de type texte. Cela passe principalement par les opérateurs habituels en SQL, à savoir LIKE et ILIKE, mais cela peut aussi passer par les opérateurs ~ et ~* pour des recherches sur des expressions rationnelles. Cependant, même les recherches sur des expressions rationnelles ne sont pas suffisantes pour subvenir aux besoins des systèmes d'informations modernes. Ces derniers ont maintenant besoin de recherches basées sur la linguistique. Une recherche du mot « présentation » devrait pouvoir aussi ramener les documents qui contiennent les mots « présenter » et « présentateur ». Cette recherche ramène donc beaucoup plus de documents qu'une recherche stricte. C'est pour cela qu'il est aussi bon de pouvoir donner un score aux documents trouvés suivant la similarité des termes qu'ils contiennent avec les termes de la recherche. Enfin, proposer un extrait du document où se trouvent les termes recherchés devient là-aussi indispensable. Ce type de recherche est appelé de la recherche plein texte. L'acronyme FTS, « Full Text Search » en anglais, est souvent utilisé.

Depuis la version 7.4, PostgreSQL dispose d'un module de recherche plein texte nommé Tsearch2. Cependant, ce module ne fait pas partie du cœur de l'application. Il est proposé en tant que module « contrib ». Son installation se fait en plus et nécessite l'exécution d'un script SQL pour ajouter les objets (tables, types, fonctions, opérateurs) nécessaires à son bon fonctionnement. La version 8.3 fait un bon en avant en proposant une variante améliorée de ce module directement au cœur de PostgreSQL. Les objets sont disponibles par défaut et leur utilisation peut se faire directement, sans installation supplémentaire.

La recherche plein texte se fait sur des documents. Un document peut être n'importe quel type de champ texte : char, varchar ou text. Lors d'une recherche, le document passe par un analyseur qui le transforme en liste de jetons. Un jeton n'est qu'un fragment du document. Chaque jeton fait partie d'une classe : nombre, mot, mot composé, adresse mail, URL, etc. Tous les jetons passent, un par un, par le dictionnaire qui a deux actions sur eux : vérifier que le jeton n'est pas un terme courant (nous ne voulons pas indexer les pronoms ou la liaison « et » qui sont bien trop fréquents en français pour être significatifs), et normaliser le jeton (c'est-à-dire le mettre en minuscule, supprimer le suffixe pour ne récupérer que la racine, etc). En sortie, nous obtenons une liste de lexemes. Ces derniers sont les termes qu'il sera possible de rechercher et d'indexer.

Traiter chaque document demande du temps. Lors d'une recherche, ce temps est précieux. Pouvoir stocker les lexemes générés permet de gagner du temps. La recherche plein texte introduit un nouveau type pour le stockage des lexemes : tsvector. Il est possible d'ajouter une colonne à la table contenant un champ de type text où des recherches plein texte ont lieu fréquemment. Il est aussi possible de créer une table où les lexemes seront stockés avec un identifiant permettant de revenir au document d'origine. Pour gagner encore plus en performances, l'indexation de ce type de données est possible. Il existe deux algorithmes particulièrement efficaces. GiST permet une indexation simple et rapide alors que GIN est beaucoup plus performant lors de la recherche au prix d'une indexation plus lente et d'un index beaucoup plus gros. Parfois, il est intéressant de coupler les deux : avoir un index GiST sur tous les documents, et un index GIN sur les documents les plus recherchés (par exemple, les derniers documents disponibles).

Évidemment, l'étape de normalisation est configurable. Suivant l'utilisation, les termes courants, les synonymes, le thésaurus seront différents. Tout ceci est donc modifiable. Toute la théorie étant passé (très rapidement) en revue, passons maintenant à la pratique.

Mise en pratique simple

Commençons par créer une base de données, ajoutons une table avec un champ de type text, et insérons quelques données dans cette table :

```
guillaume@laptop$ createdb glmf
psql guillaume@laptop$ psql glmf
Bienvenue dans psql 8.3.4, l'interface interactive de PostgreSQL.
[... intro coupée ...]
glmf=# CREATE TABLE contenu (id serial, texte text);
NOTICE:  CREATE TABLE créera des séquences implicites « contenu_id_seq » pour la colonne serial « contenu.id »
CREATE TABLE
glmf=# INSERT INTO contenu (texte) VALUES ('PostgreSQL est un excellent moteur de bases de données');
INSERT 0 1
glmf=# INSERT INTO contenu (texte) VALUES ('Pour stocker vos bases de données, pensez PostgreSQL');
INSERT 0 1
glmf=# INSERT INTO contenu (texte) VALUES ('Vous avez besoin d''une excellente façon de stocker vos bases de données ? utilisez pgsql');
INSERT 0 1
```

Pour rechercher du texte, l'opérateur LIKE est le plus connu :

```
glmf=# SELECT * FROM contenu WHERE texte LIKE '%excellente%';
 id |                                texte
----+-----------------------------------------------------------------------------
  3 | Vous avez besoin d'une excellente façon de stocker vos bases de données ? utilisez pgsql
(1 ligne)
```

Utilisons maintenant la recherche plein texte. Comme indiqué dans la partie théorique, nous devons tout d'abord transformer les documents en liste de lexemes. Pour cela, le plus simple est d'utiliser la fonction to_tsvector :

```
glmf=# \x
Affichage étendu activé.
glmf=# SELECT texte, to_tsvector(texte) FROM contenu;
-[ RECORD 1 ]----------------------------------------------------------------------------------------------
texte       | PostgreSQL est un excellent moteur de bases de données
to_tsvector | 'bas':7 'don':9 'moteur':5 'excellent':4 'postgresql':1
-[ RECORD 2 ]----------------------------------------------------------------------------------------------
texte       | Pour stocker vos bases de données, pensez PostgreSQL
to_tsvector | 'bas':4 'don':6 'pens':7 'stock':2 'postgresql':8
-[ RECORD 3 ]----------------------------------------------------------------------------------------------
texte       | Vous avez besoin d'une excellente façon de stocker vos bases de données ? utilisez pgsql
to_tsvector | 'bas':11 'don':13 'pgsql':15 'stock':9 'besoin':3 'façon':7 'utilis':14 'excellent':6
```

Le résultat de la fonction to_tsvector est un ensemble de termes qui ont été analysés et convertis. Ils sont tous en minuscule. Seule la racine du mot est conservée. Par exemple, « bas » pour « bases », « stock » pour « stocker », « pens » pour « pense

```
glmf=# \x
Affichage étendu désactivé.
glmf=# SELECT token, description, dictionary, lexemes
glmf-# FROM ts_debug('PostgreSQL est un excellent moteur de bases de données, disponible en version 8.3.4 sur http://www.postgresql.org');
glmf=# SELECT token, description, dictionary, lexemes
FROM ts_debug('PostgreSQL est un excellent moteur de bases de données, disponible en version 8.3.4 sur http://www.postgresql.org');
      token       |    description    | dictionary |      lexemes
------------------+-------------------+------------+--------------------
 PostgreSQL       | Word, all ASCII   | french_stem | {postgresql}
                  | Space symbols     |            |
 est              | Word, all ASCII   | french_stem | {}
                  | Space symbols     |            |
 un               | Word, all ASCII   | french_stem | {}
                  | Space symbols     |            |
 excellent        | Word, all ASCII   | french_stem | {excellent}
                  | Space symbols     |            |
 moteur           | Word, all ASCII   | french_stem | {moteur}
                  | Space symbols     |            |
 de               | Word, all ASCII   | french_stem | {}
                  | Space symbols     |            |
 bases            | Word, all ASCII   | french_stem | {bas}
                  | Space symbols     |            |
 de               | Word, all ASCII   | french_stem | {}
                  | Space symbols     |            |
 données          | Word, all letters | french_stem | {don}
 ,                | Space symbols     |            |
 disponible       | Word, all ASCII   | french_stem | {disponibl}
                  | Space symbols     |            |
 en               | Word, all ASCII   | french_stem | {}
                  | Space symbols     |            |
```

```
 version          | Word, all ASCII   | french_stem | {version}
                  | Space symbols     |             |
 8.3.4            | Version number    | simple      | {8.3.4}
                  | Space symbols     |             |
 sur              | Word, all ASCII   | french_stem | {}
                  | Space symbols     |             |
 http://          | Protocol head     |             |
 www.postgresql.org | Host            | simple      | {www.postgresql.org}
(30 lignes)
```

On s'aperçoit ainsi que la phrase est découpée en mots, que chaque mot est analysé, qu'un dictionnaire peut le reconnaître et renvoyer la racine du mot. Le dictionnaire utilisé le plus fréquemment dans cet exemple est le dictionnaire de français, mais on voit qu'il existe aussi un dictionnaire pour les numéros de version, un autre pour les en-têtes de protocole, encore un autre pour les hôtes, etc.

Pour effectuer une recherche, nous devons utiliser un opérateur spécial, @@, qui associe une valeur de type tsvector à une valeur de type tsquery. Ce dernier type est une recherche préalablement formatée. Le formatage s'obtient grâce à la fonction to_tsquery. Par exemple, si nous voulons rechercher « excellente » dans tous les documents, cela nous donne cette requête :

```
glmf=# SELECT * FROM contenu WHERE to_tsvector(texte) @@ to_tsquery('excellente');
 id |                          texte
----+---------------------------------------------------------------------------------------------------
  1 | PostgreSQL est un excellent moteur de bases de données
  3 | Vous avez besoin d'une excellente façon de stocker vos bases de données ? utilisez pgsql
(2 lignes)
```

Cette recherche a récupéré deux lignes dont une qui ne contient pas le mot « excellente ». Étonnant, non ? Non, pas du tout. En effet, la racine du mot « excellente » pour la recherche plein texte est « excellent ». Du coup, la recherche se fait sur cette racine. Et elle renvoit bien ces deux documents car ces deux documents contiennent chacun un mot dont la racine est « excellent ».

Le problème principal de cette requête est que la fonction to_tsvector sera exécutée pour chaque ligne de la table. Cela peut prendre beaucoup de temps, notamment si le champ texte contient beaucoup de mots ou si la table contient un grand nombre de lignes. Testons ce dernier cas :

```
glmf=# INSERT INTO contenu (texte) SELECT 'un grand texte ici pour avoir plein de mots et plein de lignes... si si... beaucoup beaucoup beaucoup' FROM generate_series(1, 1000000);
glmf=# \timing
Chronométrage activé.
glmf=# SELECT id FROM contenu WHERE to_tsvector(texte) @@ to_tsquery('excellente');
 id
----
  1
  3
(2 lignes)

Temps : 87749,739 ms
glmf=# SELECT id FROM contenu WHERE texte LIKE '%excellente%';
 id
----
  3
(1 ligne)

Temps : 924,934 ms
```

La différence est frappante : 87 secondes en recherche plein texte, moins d'une seconde avec un LIKE. Le résultat n'est pas identique mais même en ne recherchant que la racine du mot dans la requête LIKE, on obtient sensiblement le même temps d'exécution, et du coup la même différence :

```
glmf=# SELECT id FROM contenu WHERE texte LIKE '%excellent%';
 id
----
  1
  3
(2 lignes)

Temps : 903,873 ms
```

Le seul moyen de corriger cela, c'est de stocker le résultat de la fonction to_tsvector. Peu importe la façon : une nouvelle colonne dans la même table, une colonne dans une table spécifique. L'important est que le temps d'exécution de la fonction ne soit plus à comptabiliser dans la durée d'exécution de la requête. Nous allons donc ajouter une colonne dans la table contenu, et y stocker le résultat de l'exécution de la fonction to_tsvector :

```
glmf=# ALTER TABLE contenu ADD COLUMN texte_vectorise tsvector;
ALTER TABLE
Temps : 12,839 ms
glmf=# UPDATE contenu SET texte_vectorise=to_tsvector(texte);
UPDATE 1000003
Temps : 147229,088 ms
glmf=# SELECT id FROM contenu WHERE texte_vectorise @@ to_tsquery('excellente');
 id
----
  1
  3
(2 lignes)

Temps : 8746,713 ms
```

C'est meilleur, mais c'est encore huit fois plus lent que l'opérateur LIKE. Pour accélérer les performances, le dernier moyen est d'utiliser un index.

```
glmf=# CREATE INDEX idx_vecteur ON contenu USING gin (texte_vectorise);
CREATE INDEX
Temps : 32987,735 ms
glmf=# SELECT id FROM contenu WHERE texte_vectorise @@ to_tsquery('excellente');
 id
----
  1
  3
(2 lignes)

Temps : 41,268 ms
```

41 millisecondes, soit 20 fois plus rapide que le LIKE. Le fait de ne pas utiliser d'index avec LIKE n'est pas injuste. Nous pouvons créer un index sur la colonne texte, mais le fait d'utiliser le joker % sur chaque côté du terme à rechercher invalide immédiatement tous les index disponibles lors d'une recherche avec LIKE.

Le problème que cela pose maintenant est double. Tout d'abord, nous devons nous assurer de la mise à jour de la colonne texte_vectorise à chaque fois que la colonne texte est modifiée. De la même façon, si une nouvelle ligne est insérée, la colonne texte_vectorise doit automatiquement prendre en compte la valeur de la colonne texte. Le mieux dans ce cadre est l'utilisation d'un trigger. Il serait assez simple d'écrire sa propre fonction trigger, mais il se trouve que PostgreSQL nous en propose une par défaut, tsvector_update_trigger. Voyons ce qu'il se passe lors d'une insertion sans le trigger :

```
glmf=# \timing
Chronométrage désactivé.
glmf=# INSERT INTO contenu (texte) VALUES ('un grand texte ici pour avoir plein de mots et plein de lignes... si si... beaucoup beaucoup beaucoup') RETURNING id, texte_vectorise;
   id    | texte_vectorise
---------+-----------------
 1000004 |
(1 ligne)

INSERT 0 1
```

Sans trigger, texte_vectorise est vide (NULL). Ajoutons le trigger et insérons une nouvelle ligne :

```
glmf=# CREATE TRIGGER tsvectorupdate BEFORE INSERT OR UPDATE
glmf-# ON contenu FOR EACH ROW EXECUTE PROCEDURE
glmf-# tsvector_update_trigger(texte_vectorise, 'pg_catalog.english', texte);
CREATE TRIGGER
Temps : 141,013 ms
glmf=# INSERT INTO contenu (texte) VALUES ('un grand texte ici pour avoir plein de mots et plein de lignes... si si... beaucoup beaucoup beaucoup') RETURNING id, texte_vectorise;
   id    |                                                                                                                                                texte_vectorise
---------+-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
 1000005 | 'de':8,12 'et':10 'si':14,15 'un':1 'ici':4 'mot':9 'lign':13 'pour':5 'text':3 'avoir':6 'grand':2 'plein':7,11 'beaucoup':16,17,18
(1 ligne)

INSERT 0 1
```

Cette fois, la colonne est renseignée.

Autre problème : on se rend compte facilement qu'une recherche plein texte va souvent ramener bien plus de document qu'une recherche LIKE. Tout comme rechercher un terme dans Google renvoie un nombre très important de résultats, mais le gros intérêt de Google est qu'il peut classer les résultats d'après un score. Cette fonctionnalité de score est aussi disponible dans la recherche plein texte. La fonction s'appelle ts_rank et demande deux arguments, les lexemes du document et la recherche formatée :

```
glmf=# SELECT id, ts_rank(texte_vectorise, to_tsquery('excellente')) FROM contenu WHERE texte_vectorise @@ to_tsquery('excellente');
 id |  ts_rank
----+-----------
  1 | 0.0607927
  3 | 0.0607927
(2 lignes)
```

Pas de chance, dans notre exemple, les deux lignes ont le même score. Les mots sont identiques, ils sont présents autant de fois dans une chaîne que

dans l'autre. Donc, un même score. Le fait que dans un cas il s'agit réellement du mot « excellent » et dans l'autre cas du mot « excellente » ne change rien car la comparaison, au niveau de la recherche plein texte, ne se fait pas avec les mots du document mais avec les racines trouvées.

Il serait intéressant de pouvoir départager un mot qui se trouve dans une partie importante d'un texte comme un titre par exemple et ceux disponibles dans le reste. C'est tout à fait possible grâce à la notion de poids. Quatre poids sont disponibles, notés de A à D. Pour l'instant, nous avions vu qu'un document était un champ mais il est possible d'assembler plusieurs champs pour créer un document. Avec les poids, il est même possible d'affecter des poids différents (avec une limite de quatre) à chacun des champs.

Commençons par supprimer le million de lignes contenant le même texte. Nous ne faisons plus de tests de performance, autant ne pas perdre de temps à les mettre à jour :

```
glmf=# DELETE FROM contenu WHERE texte='un grand texte ici pour avoir plein de mots et plein de lignes... si si... beaucoup beaucoup beaucoup';
DELETE 1000002
```

Supprimons aussi le trigger :

```
glmf=# drop trigger tsvectorupdate
glmf-# on contenu;
DROP TRIGGER
```

Ajoutons une colonne titre de type text à notre table contenu :

```
glmf=# ALTER TABLE contenu ADD COLUMN titre text;
ALTER TABLE
glmf=# UPDATE contenu SET titre='';
UPDATE 3
glmf=# UPDATE contenu SET titre='De l''excellence de PostgreSQL pour le stockage de vos données'
glmf-# WHERE id=1;
UPDATE 1
glmf=# UPDATE contenu SET titre='PostgreSQL, la base qu''il vous faut' WHERE id=3;
UPDATE 1
```

Il faut maintenant que la colonne texte_vectorise prenne en compte les deux colonnes, en précisant un poids important pour la colonne titre et un poids inférieur pour la colonne texte :

```
gmlf=# UPDATE contenu SET texte_vectorise=setweight(to_tsvector(coalesce(titre,'')), 'A') || setweight(to_tsvector(coalesce(texte,'')), 'B');
UPDATE 3
```

Maintenant, vu que le titre de la ligne 1 contient le texte recherché contrairement à la ligne 3, la ligne 1 devrait obtenir un meilleur score que la ligne 3. Vérifions cela :

```
glmf=# SELECT id, ts_rank(texte_vectorise, to_tsquery('excellente')) FROM contenu WHERE texte_vectorise @@ to_tsquery('excellente');
 id | ts_rank
----+----------
  1 |  0.66872
  3 | 0.243171
(2 lignes)
```

C'est exactement cela. Retournons voir le contenu complet de la table, et notamment celui de la colonne texte_vectorise :

```
glmf=# \x
Affichage étendu activé.
glmf=# SELECT * FROM contenu;
-[ RECORD 1 ]---+------------------------------------------------------------------------------------------------------------------------
id              | 2
texte           | Pour stocker vos bases de données, pensez PostgreSQL
texte_vectorise | 'bas':4B 'don':6B 'pens':7B 'stock':2B 'postgresql':8B
titre           |
-[ RECORD 2 ]---+------------------------------------------------------------------------------------------------------------------------
id              | 1
texte           | PostgreSQL est un excellent moteur de bases de données
texte_vectorise | 'bas':18B 'don':11A,20B 'moteur':16B 'stockag':8A 'excellent':3A,15B 'postgresql':5A,12B
titre           | De l'excellence de PostgreSQL pour le stockage de vos données
-[ RECORD 3 ]---+------------------------------------------------------------------------------------------------------------------------
id              | 3
texte           | Vous avez besoin d'une excellente façon de stocker vos bases de données ? utilisez pgsql
texte_vectorise | 'bas':3A,18B 'don':20B 'faut':7A 'pgsql':22B 'stock':16B 'besoin':10B 'façon':14B 'utilis':21B 'excellent':13B 'postgresql':1A
titre           | PostgreSQL, la base qu'il vous faut
```

Nous avons vu plus haut que le nombre situé après la racine du mot indiquait sa position dans le document. C'est toujours vrai, mais nous remarquons en plus que le poids est aussi précisé. Par exemple, pour la ligne avec id à 1, le lexeme « excellent » est présent deux fois : une première fois en position 3 pour le champ de poids A et une deuxième fois en position 15 pour le champ de poids B. Attention à bien noter que la position indiquée fait référence à la position dans le champ « virtuel » né de l'agrégation des deux champs.

Et si on souhaitait maintenant récupérer en plus un extrait du document montrant les termes recherchés ? La fonction ts_headline est là pour ça. Elle prend en argument le document et la recherche au format tsquery. Voici un exemple d'utilisation :

```
glmf=# \x
Affichage étendu désactivé.
glmf=# SELECT ts_headline(titre || texte, to_tsquery('excellente')) FROM contenu WHERE texte_vectorise @@ to_tsquery('excellente');
                                          ts_headline
-----------------------------------------------------------------------------------------------------------
 <b>excellence</b> de PostgreSQL pour le stockage de vos donnéesPostgreSQL est un <b>excellent</b> moteur de bases
 avez besoin d'une <b>excellente</b> façon de stocker vos bases de données ? utilisez pgsql
(2 lignes)
```

Le résultat est immédiat, le retour est une chaîne de caractères dont les termes de la recherche sont encadrés par des balises HTML de gras (et). Ces balises sont modifiables avec les options StartSel et StopSel. D'autres options sont disponibles : MinWords et MaxWords sont le nombre minimum et maximum de mots dans le résumé renvoyé par la fonction ts_headline. Il existe aussi HighlightAll pour surligner tout le résumé et non pas seulement les mots de la recherche apparaissant dans le résumé. ShortWords permet de supprimer les mots de cette longueur ou plus petits au début et à la fin du résumé. Les options voulues doivent être indiquées dans une chaîne donnée en troisième argument de la fonction, de cette façon :

```
glmf=# SELECT ts_headline(titre || texte, to_tsquery('excellente'),
glmf-# 'StartSel=|, StopSel=|, ShortWords=5')
glmf-# FROM contenu
glmf-# WHERE texte_vectorise @@ to_tsquery('excellente');
                                          ts_headline
-----------------------------------------------------------------------------------------------------------
 |excellence| de PostgreSQL pour le stockage de vos donnéesPostgreSQL est un |excellent| moteur de bases de données
 fautVous avez besoin d'une |excellente| façon de stocker vos bases de données ? utilisez pgsql
(2 lignes)
```

Pour l'instant, nous n'avons fait que des recherches simples avec un seul mot. Un moteur de recherche permet habituellement de chercher plusieurs mots. La recherche plein texte le permet. Chaque mot doit être séparé par un opérateur binaire : & pour le ET logique, | pour le OU logique. Vous pouvez aussi utiliser la négation avec l'opérateur !. L'opérateur est obligatoire. Si vous indiquez deux mots l'un à la suite de l'autre, la recherche plein texte ne peut pas savoir si vous voulez trouver la racine de l'un et la racine de l'autre, ou si vous voulez trouver la racine de l'un ou la racine de l'autre. Voici quelques exemples :

```
glmf=# SELECT ts_headline(titre || texte, to_tsquery('excellente')) FROM contenu
glmf-# WHERE texte_vectorise @@ to_tsquery('excellente & moteur');
                                          ts_headline
-----------------------------------------------------------------------------------------------------------
 <b>excellence</b> de PostgreSQL pour le stockage de vos donnéesPostgreSQL est un <b>excellent</b> moteur de bases
(1 ligne)
```

Si vous voulez chercher une expression complète, vous devez tout d'abord faire la recherche plein texte et ajouter une recherche LIKE dans ce premire résultat. Vous bénéficierez de la rapidité du premier pour aboutir à un ensemble de lignes bien moindres, sur lequel vous pourrez exécuter le filtre LIKE avec moins de problème de performances.

Allons plus loin

Nous n'avons vu pour l'instant que les fonctionnalités de base de la recherche plein texte. Il nous reste encore toute la partie de configuration. Par défaut, la recherche plein texte utilise la configuration indiquée par le paramètre default_text_search_config :

```
glmf=# SHOW default_text_search_config ;
 default_text_search_config
----------------------------
 pg_catalog.french
(1 ligne)
```

En effet, il est possible de créer plusieurs configurations de recherche plein texte. Le catalogue système pg_ts_config indique toutes les configurations disponibles. Par défaut, il en existe 16, une « simple » et 15 dépendantes de la langue. Chaque configuration établit une correspondance entre un type de jeton et le dictionnaire à utiliser pour ce type de jeton. Elle précise aussi l'ordre dans lequel les types de jeton sont vérifiés. Il existe 23 types de jeton, allant du mot simple, à l'entier, en passant par les adresses mails, les URL, les noms de fichier, etc. Par exemple, la configuration french utilise ceci :

```
glmf=# \dF+ french
Configuration « pg_catalog.french » de la recherche de texte
Analyseur : « pg_catalog.default »
      Jeton      | Dictionnaires
-----------------+---------------
 asciihword      | french_stem
 asciiword       | french_stem
```

```
 email           | simple
 file            | simple
 float           | simple
 host            | simple
 hword           | french_stem
 hword_asciipart | french_stem
 hword_numpart   | simple
 hword_part      | french_stem
 int             | simple
 numhword        | simple
 numword         | simple
 sfloat          | simple
 uint            | simple
 url             | simple
 url_path        | simple
 version         | simple
 word            | french_stem
```

Le dictionnaire french_stem est utilisé pour les types asciihword, asciiword, hword, hword_asciipart, hword_part, word (en bref, tout ce qui correspond à un mot de la langue française) et le dictionne simple est utilisé pour tout le reste (donc les nombres, entiers ou flottants, les fichiers, les URL, les versions, etc.)

Quel est l'intérêt de savoir tout cela ? L'intérêt est simple quand on sait qu'il nous est possible de créer notre propre configuration, de définir nos mots courants, de créer un dictionnaire (normal, des synonymes ou un thésaurus), etc. Ce que nous allons faire maintenant.

Nous allons créer une configuration à partir de rien :

```
glmf=# CREATE TEXT SEARCH CONFIGURATION cfg (parser=default);
CREATE TEXT SEARCH CONFIGURATION
```

Changeons de configuration par défaut et lançons une recherche :

```
glmf=# SET default_text_search_config TO cfg;
SET
glmf=# SELECT token, description, dictionary, lexemes FROM ts_debug('PostgreSQL est un excellent moteur de bases de données, disponible en version 8.3.4 sur http://www.postgresql.org') WHERE lexemes IS NOT NULL;
(0 lignes)
```

Aucun lexeme n'est renvoyé. Cela paraît logique car nous n'avons établi aucune correspondance entre type de jeton et dictionnaire. Commençons par exemple avec le dictionnaire simple pour le type de jeton « version » :

```
glmf=# ALTER TEXT SEARCH CONFIGURATION cfg ADD MAPPING FOR version WITH simple;
ALTER TEXT SEARCH CONFIGURATION
glmf=# SELECT token, description, dictionary, lexemes FROM ts_debug('PostgreSQL est un excellent moteur de bases de données, disponible en version 8.3.4 sur http://www.postgresql.org') WHERE lexemes IS NOT NULL;
    token     |    description    | dictionary | lexemes
--------------+-------------------+------------+---------
 8.3.4        | Version number    | simple     | {8.3.4}
(1 ligne)
```

La recherche fonctionne et nous renvoie un numéro de version. Pour obtenir une recherche sur les mots, il nous faut ajouter un dictionnaire pour le type de jeton asciiword :

```
glmf=# ALTER TEXT SEARCH CONFIGURATION cfg ADD MAPPING FOR asciiword WITH french_stem;
ALTER TEXT SEARCH CONFIGURATION
glmf=# SELECT token, description, dictionary, lexemes FROM ts_debug('PostgreSQL est un excellent moteur de bases de données, disponible en version 8.3.4 sur http://www.postgresql.org') WHERE lexemes IS NOT NULL;
   token   |   description   | dictionary |   lexemes
-----------+-----------------+------------+-------------
 PostgreSQL  | Word, all ASCII | french_stem | {postgresql}
 est         | Word, all ASCII | french_stem | {}
 un          | Word, all ASCII | french_stem | {}
 excellent   | Word, all ASCII | french_stem | {excellent}
 moteur      | Word, all ASCII | french_stem | {moteur}
 de          | Word, all ASCII | french_stem | {}
 bases       | Word, all ASCII | french_stem | {bas}
 de          | Word, all ASCII | french_stem | {}
 disponible  | Word, all ASCII | french_stem | {disponibl}
 en          | Word, all ASCII | french_stem | {}
 version     | Word, all ASCII | french_stem | {version}
 8.3.4       | Version number  | simple      | {8.3.4}
 sur         | Word, all ASCII | french_stem | {}
(13 lignes)
```

Parfait. Et si nous voulions ajouter notre propre dictionnaire ? C'est pratiquement aussi simple. Déjà, commençons par regarder les modèles de dictionnaires disponibles. La liste se trouve dans le catalogue système pg_ts_template :

```
glmf=# \qglmf=# SELECT * FROM pg_ts_template;
 tmplname  | tmplnamespace |    tmplinit    |    tmpllexize
-----------+---------------+----------------+-----------------
 simple    |            11 | dsimple_init   | dsimple_lexize
 synonym   |            11 | dsynonym_init  | dsynonym_lexize
 ispell    |            11 | dispell_init   | dispell_lexize
 thesaurus |            11 | thesaurus_init | thesaurus_lexize
 snowball  |            11 | dsnowball_init | dsnowball_lexize
(5 lignes)
```

Autrement dit, il est possible de créer un dictionnaire simple, un dictionnaire des synonymes, un dictionnaire ispell, un thésaurus et enfin un dictionnaire snowball. Pour créer un dictionnaire des synonymes, nous devons tout d'abord créer un fichier, de suffixe .syn dans un répertoire spécifique de PostgreSQL. Ce répertoire s'appelle tsearch_data et se trouve dans un répertoire dont on peut connaître l'emplacement en utilisant l'outil pg_config :

```
glmf=# \q
guillaume@laptop$ pg_config --sharedir
/opt/postgresql-8.3/share
```

Commençons par créer le fichier des synonymes. La syntaxe est simple : le terme à gauche sera remplacé par son synonyme à droite. Nous allons donc indiquer dans ce fichier que les mots postgresql, postgres et pgsql sont des synonymes de pg.

```
guillaume@laptop$ cat > /opt/postgresql-8.3/share/tsearch_data/mes_synonymes.syn <<_fin_
> postgresql pg
> postgres pg
> pgsql pg
> _fin_
```

Maintenant, nous allons créer le dictionnaire en précisant le nom du fichier (sans le chemin et sans l'extension) avec l'option SYNONYMS :

```
guillaume@laptop$ psql glmf
Bienvenue dans psql 8.3.4, l'interface interactive de PostgreSQL.
[... coupé ...]
glmf=# CREATE TEXT SEARCH DICTIONARY mon_dict_synonyme (TEMPLATE = synonym, SYNONYMS = mes_synonymes);
CREATE TEXT SEARCH DICTIONARY
```

Et nous ajoutons ce dictionnaire, avant french_stem, dans la correspondance avec le type de jeton asciiword :

```
glmf=# ALTER TEXT SEARCH CONFIGURATION cfg ALTER MAPPING FOR asciiword WITH mon_dict_synonyme, french_stem;
ALTER TEXT SEARCH CONFIGURATION
glmf=# SET default_text_search_config TO cfg;
SET
glmf=# SELECT token, description, dictionary, lexemes FROM ts_debug('PostgreSQL est un excellent moteur de bases de données, disponible en version 8.3.4 sur http://www.postgresql.org ') WHERE lexemes IS NOT NULL;
   token   |   description   |    dictionary    |   lexemes
-----------+-----------------+------------------+-------------
 PostgreSQL  | Word, all ASCII | mon_dict_synonyme | {pg}
 est         | Word, all ASCII | french_stem       | {}
 un          | Word, all ASCII | french_stem       | {}
 excellent   | Word, all ASCII | french_stem       | {excellent}
 moteur      | Word, all ASCII | french_stem       | {moteur}
 de          | Word, all ASCII | french_stem       | {}
 bases       | Word, all ASCII | french_stem       | {bas}
 de          | Word, all ASCII | french_stem       | {}
 disponible  | Word, all ASCII | french_stem       | {disposnibl}
 en          | Word, all ASCII | french_stem       | {}
 version     | Word, all ASCII | french_stem       | {version}
 8.3.4       | Version number  | simple            | {8.3.4}
 sur         | Word, all ASCII | french_stem       | {}
(13 lignes)
```

La ligne 1 indique bien maintenant que le lexeme de PostgreSQL est pg et que le dictionnaire utilisé pour ce lexeme est bien notre nouveau dictionnaire, à savoir mon_dict_synonyme.

Ajoutons un nouveau dictionnaire, ispell cette fois-ci, et ajoutons-y un fichier personnalisé de termes courants. Commençons par récupérer un dictionnaire ispell français. Le site officiel de Tsearch2 en propose un :

```
glmf=# \q
guillaume@laptop$ cd /opt/postgresql-8.3/share/tsearch_data
guillaume@laptop$ wget -q http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/dicts/ispell/ispell_utf8_french.tar.gz
guillaume@laptop$ tar xvfz ispell_utf8_french.tar.gz
ispell_utf8_french/
ispell_utf8_french/french_utf8-stop-ispell.txt
ispell_utf8_french/french_utf8.aff
ispell_utf8_french/french_utf8.dict
```

Le premier fichier correspond au fichier des termes courants, le deuxième contient les affixes et le dernier est le dictionnaire.

Déplaçons les fichiers dans le bon répertoire et renommons les fichiers pour qu'ils correspondent à la configuration du modèle :

```
guillaume@laptop$ mv ispell_utf8_french/* .
guillaume@laptop$ mv french_utf8.aff french_utf8.affix
guillaume@laptop$ mv french_utf8-stop-ispell.txt french_utf8_stop_ispell.stop
Maintenant, créons le nouveau dictionnaire :
guillaume@laptop$ psql glmf
Bienvenue dans psql 8.3.4, l'interface interactive de PostgreSQL.
[... coupé ...]
glmf=# CREATE TEXT SEARCH DICTIONARY mon_ispell(
glmf-# template = ispell,
glmf-# DictFile = french_utf8,
glmf-# AffFile = french_utf8,
glmf-# StopWords = french_utf8_stop_ispell
glmf-# );
CREATE TEXT SEARCH DICTIONARY
```

Remplaçons le dictionnaire snowball french_stem par notre nouveau dictionnaire ispell :

```
glmf=# ALTER TEXT SEARCH CONFIGURATION cfg ALTER MAPPING FOR asciiword WITH mon_dict_synonyme, mon_ispell;
ALTER TEXT SEARCH CONFIGURATION
glmf=# SET default_text_search_config TO cfg;
SET
glmf=# SELECT token, description, dictionary, lexemes FROM ts_debug('PostgreSQL est un excellent moteur de bases de données, disposnible en version 8.3.4 sur http://www.postgresql.org ') WHERE lexemes IS NOT NULL;
   token    |   description   |    dictionary    |          lexemes
------------+-----------------+------------------+---------------------------
 PostgreSQL | Word, all ASCII | mon_dict_synonyme | {pg}
 est        | Word, all ASCII | mon_ispell       | {}
 un         | Word, all ASCII | mon_ispell       | {}
 excellent  | Word, all ASCII | mon_ispell       | {excellente,excelle,celer}
 moteur     | Word, all ASCII | mon_ispell       | {motrice}
 de         | Word, all ASCII | mon_ispell       | {}
 bases      | Word, all ASCII | mon_ispell       | {base}
 de         | Word, all ASCII | mon_ispell       | {}
 en         | Word, all ASCII | mon_ispell       | {}
 version    | Word, all ASCII | mon_ispell       | {version}
 8.3.4      | Version number  | simple           | {8.3.4}
 sur        | Word, all ASCII | mon_ispell       | {sure}
(12 lignes)
```

Nous voyons rapidement que les lexemes trouvés par ispell sont bien différents de ceux trouvés par snowball.

Ajoutons maintenant un terme courant à notre dernier dictionnaire. Par exemple, « version » :

```
glmf=# \q
guillaume@laptop$ cat >> french_utf8_stop_ispell.stop <<_fin_
> version
> _fin_
guillaume@laptop$ psql glmf
Bienvenue dans psql 8.3.4, l'interface interactive de PostgreSQL.
[... coupé ...]
glmf=# SET default_text_search_config TO cfg;
SET
glmf=# SELECT token, description, dictionary, lexemes FROM ts_debug('PostgreSQL est un excellent moteur de bases de données, disposnible en version 8.3.4 sur http://www.postgresql.org ') WHERE lexemes is not null and token='version';
  token  |   description   | dictionary | lexemes
---------+-----------------+------------+---------
 version | Word, all ASCII | mon_ispell | {}
(1 ligne)
```

Et voilà !

Un outil supplémentaire très intéressant : pg_trgm

La recherche plein texte est particulièrement intéressante dans le cas d'une base documentaire. Malheureusement, il peut arriver qu'un utilisateur saisisse mal sa recherche. Il est intéressant dans ce cas de lui proposer des alternatives. Qui n'a jamais oublié une lettre dans un mot, voire inverser deux lettres d'un même mot ? La recherche plein texte ne fournit pas de solution directe à ce problème. Par contre, un module contrib va nous aider.

pg_trgm cherche à déterminer la similarité d'un texte en se basant sur un trigramme (d'où son nom). Pour l'intégrer à la recherche plein texte, il nous faut avoir une liste complète des mots composant les documents. La fonction de recherche de similarité est ensuite exécutée sur ce dictionnaire de mots.

Commençons déjà par installer les objets de ce module contrib. Vous devez trouver le fichier pg_trgm.sql. Normalement, il est installé au même endroit que tous les modules contrib. Une fois trouvé, vous devez exécuter ce script SQL dans votre base :

```
guillaume@laptop$ psql -f /opt/postgresql-8.3/share/contrib/pg_trgm.sql glmf
SET
CREATE FUNCTION
CREATE FUNCTION
[... coupé ...]
CREATE FUNCTION
CREATE OPERATOR CLASS
```

Ensuite, nous allons créer notre dictionnaire de mots. Pour cela, nous appelons une fonction ts_stat, une fonction qui fournit des statistiques sur les mots d'un vecteur :

```
glmf=# CREATE TABLE mots AS
glmf-# SELECT word AS mot FROM ts_stat('SELECT to_tsvector(''simple'', texte) FROM contenu');
SELECT
```

Ajoutons un index GiN pour gagner en performance :

```
glmf=# CREATE INDEX mots_idx ON mots USING gin(mot gin_trgm_ops);
CREATE INDEX
```

Cherchons tous les mots similaires à « excllente » :

```
glmf=# SELECT mot, similarity(mot, 'excllente') AS sml FROM mots
glmf-# WHERE mot % 'excllente'
glmf-# ORDER BY sml DESC, mot;
    mot    |    sml
-----------+----------
 excellente | 0.615385
 excellent  | 0.428571
(2 lignes)
```

Inversons deux lettres dans PostgreSQL :

```
glmf=# SELECT mot, similarity(mot, 'Postgersql') AS sml FROM mots
glmf-# WHERE mot % 'Postgersql'
glmf-# ORDER BY sml DESC, mot;
    mot    |    sml
-----------+----------
 postgresql | 0.466667
(1 ligne)
```

Excellent, non ?

Remarquez aussi que c'est le seul moyen actuel de faire une recherche sans accent :

```
glmf=# SELECT mot, similarity(mot, 'facon') AS sml FROM mots WHERE mot % 'facon'
ORDER BY sml DESC, mot;
    mot    |    sml
-----------+----------
 façon      | 0.466667
(1 ligne)
```

Conclusion

La recherche plein texte est une fonctionnalité qui a un grand potentiel. La 8.4 va améliorer un certain nombre de points, comme le support du multi-colonnes pour les index GIN et l'amélioration de l'analyse sur les index GIN et GiST.

Mais le plus beau à venir se verra du côté des applications qui chercheront à en tirer parti.