Full-Text Search in PostgreSQL

A Gentle Introduction

Oleg Bartunov
Moscow University
oleg@sai.msu.su
Moscow
Russia

Teodor Sigaev
Moscow University
teodor@sigaev.ru
Moscow
Russia
Attention: Patch is under development, so the syntax of SQL commands will changed. Please, wait for the final release!

This document is a gentle introduction to the full-text search in ORDBMS PostgreSQL (version 8.3+). It covers basic features and contains reference of SQL commands, related to the FTS.

Brave and smart can play with the new FTS - patch for the CVS HEAD is available tsearch_core-0.52.gz\(^1\).

---

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

\(^1\) [http://www.sigaev.ru/misc/tsearch_core-0.52.gz](http://www.sigaev.ru/misc/tsearch_core-0.52.gz)
# Table of Contents

1. **FTS Introduction** ............................................................................................................. 1
   1.1. Full Text Search in databases ...................................................................................... 1
       1.1.1. What is a document? ......................................................................................... 1
   1.2. FTS Overview .............................................................................................................. 1
       1.2.1. Tquery and tsvector ....................................................................................... 2
       1.2.2. FTS operator ................................................................................................. 4
   1.3. Basic operations ......................................................................................................... 4
       1.3.1. Obtaining tsvector .......................................................................................... 4
       1.3.2. Obtaining tquery ............................................................................................. 6
       1.3.3. Ranking search results .................................................................................... 6
       1.3.4. Getting results ............................................................................................... 8
       1.3.5. Dictionaries .................................................................................................... 8
       1.3.6. Stop words ..................................................................................................... 9
   1.4. FTS features ............................................................................................................... 10
   1.5. FTS Limitations ......................................................................................................... 10
   1.6. A Brief History of FTS in PostgreSQL .................................................................... 10
       1.6.1. Pre-tsearch ....................................................................................................... 11
       1.6.2. Tsearch v1 ...................................................................................................... 11
       1.6.3. Tsearch v2 ...................................................................................................... 11
       1.6.4. FTS current ..................................................................................................... 12
   1.7. Links .......................................................................................................................... 12
   1.8. FTS Todo .................................................................................................................. 12
   1.9. Acknowledgements .................................................................................................. 13

2. **FTS Operators and Functions** ......................................................................................... 14
   2.1. FTS operator ............................................................................................................ 14
   2.2. Vector Operations .................................................................................................... 15
   2.3. Query Operations ..................................................................................................... 16
       2.3.1. Query rewriting ............................................................................................. 18
       2.3.2. Operators for tquery ....................................................................................... 19
       2.3.3. Index for tquery ............................................................................................. 20
   2.4. Parser functions ....................................................................................................... 20
   2.5. Ranking .................................................................................................................... 21
   2.6. Headline .................................................................................................................. 22
   2.7. Full-text indexes ...................................................................................................... 22
   2.8. Dictionaries ............................................................................................................. 24
       2.8.1. Simple dictionary ......................................................................................... 25
       2.8.2. Ispell dictionary ........................................................................................... 25
       2.8.3. Snowball stemming dictionary ...................................................................... 27
       2.8.4. Synonym dictionary ...................................................................................... 27
       2.8.5. Thesaurus dictionary .................................................................................... 28
           2.8.5.1. Thesaurus configuration ........................................................................... 29
           2.8.5.2. Thesaurus examples ............................................................................... 29
   2.9. FTS Configuration ................................................................................................... 30
   2.10. Debugging ............................................................................................................. 31
   2.11. Psq1 support .......................................................................................................... 32
Chapter 1. FTS Introduction

1.1. Full Text Search in databases

Full-Text Search (FTS) is a search for the documents, which satisfy query and, optionally, return them in some order. Most usual case is to find documents containing all query terms and return them in order of their similarity to the query. Notions of query and similarity are very flexible and depend on specific applications. The simplest search machine considers query as a set of words and similarity - as how frequent are query words in the document.

Ordinary full text search engines operate with collection of documents where document is considered as a "bag of words", i.e., there is a minimal knowledge about the document structure and its metadata. Big search machines make use of sophisticated heuristics to get some metadata, such as title, author(s), modification date, but their knowledge is limited by web site owner policy. But, even if you have a full access to the documents, very often, document itself, as it shown to the visitor, depends on many factors, which makes indexing of such dynamical documents practically impossible and actually, search engines fail here ("The Hidden Web" phenomena). Moreover, modern information systems are all database driven and there is a need in IR (Information Retrieval) style full text search inside database with full conformance to the database principles (ACID). That’s why, many databases have built-in full text search engines, which allow to combine text searching and additional metadata, stored in various tables and available through powerful and standard SQL language.

1.1.1. What is a document?

Document, in usual meaning, is a text file, that one could open, read and modify. Search machines parse text files and store associations of lexemes (words) with their parent document. Later, these associations used to search documents, which contain query words. In databases, notion of document is much complex, it could be any textual attribute or their combination (concatenation), which in turn may be stored in various tables or obtained on-fly. In other words, document looks as it were constructed from different pieces (of various importance) for a moment of indexing and it might be not existed as a whole. For example,

```
SELECT title || ' ' || author || ' ' || abstract || ' ' || body as document
FROM messages
WHERE mid = 12;
```

```
SELECT m.title || ' ' || m.author || ' ' || m.abstract || ' ' || d.body as document
FROM messages m, docs d
WHERE mid = did and mid = 12;
```

Document can be ordinary file, stored in filesystem, but accessible through database. In that case, database used as a storage for full text index and executor for searches. Document processed outside of database using external programs. In any cases, it’s important, that document must be somehow uniquely identified.

Actually, in previous examples we should use coalesce function to prevent document to be NULL if some of its part is NULL.
1.2. FTS Overview

Text search operators in database existed for years. PostgreSQL has ~,~*, LIKE, ILIKE operators for textual datatypes, but they lack many essential properties required for modern information system:

- there is no linguistic support, even in english, regular expressions are not enough - satisfies -> satisfy, for example. You may miss documents, which contains word satisfies, although certainly would love to find them when search for satisfy. It is possible to use OR to search any of them, but it’s boring and ineffective (some words could have several thousands of derivatives).
- they provide no ordering (ranking) of search results, which makes them a bit useless, unless there are only a few documents found.
- they tends to be slow, since they process all documents every time and there is no index support.

The improvements to the FTS came from the idea to preprocess document at index time to save time later, at a search stage. Preprocessing includes:

Parsing document to lexemes. It’s useful to distinguish various kinds of lexemes, for example, digits, words, complex words, email address, since different types of lexemes can be processed different. It’s useless to attempt normalize email address using morphological dictionary of russian language, but looks reasonable to pick out domain name and be able to search for domain name. In principle, actual types of lexemes depend on specific applications, but for plain search it’s desirable to have predefined common types of lexemes.

Applying linguistic rules to normalize lexeme to their infinitive form, so one should not bother entering search word in specific form. Taking into account type of lexeme obtained before provides rich possibilities for normalization.

Store preprocessed document in a way, optimized for searching, for example, represent document as a sorted array of lexemes. Along with lexemes itself it’s desirable to store positional information to use it for proximity ranking, so that document which contains more "dense" region with query words assigned a higher rank than one with query words scattered all over.

PostgreSQL is an extendable database, so it’s natural to introduce a new data types (Section 1.2.1) tsvector for storing preprocessed document and tsquery for textual queries. Also, full-text search operator (FTS) @@ is defined for these data types (Section 1.2.2). FTS operator can be accelerated using indices (Section 2.7).

1.2.1. Tsquery and tsvector

tsvector

tsvector is a data type, which represents document, and optimized for FTS. In simple phrase, tsvector is a sorted list of lexemes, so even without index support full text search should performs better than standard ~, LIKE operators.

```sql
=# select 'a fat cat sat on a mat and ate a fat rat':tsvector;
   tsvector
```
Chapter 1. FTS Introduction

Notice, that space is also lexeme!

Each lexeme, optionally, could have positional information, which used for proximity ranking.

Each position of a lexeme can be labeled by one of 'A','B','C','D', where 'D' is default. These labels can be used to indicate group membership of lexeme with different importance or rank, for example, reflect document structure. Actually, labels are just a way to differentiate lexemes. Actual values will be assigned at search time and used for calculation of document rank. This is very convenient to control and tune search machine.

The order is important if tsvector contains positional information. Of course, using SQL join operator, it is possible to "build" document using different tables.

Tsquery is a data type for textual queries with support of boolean operators - & (AND), | (OR), parenthesis. Tsquery consists of lexemes (optionally labeled by letter[s]) with boolean operators between.

Labels could be used to restrict search region, which allows to develop different search engines using the same full text index.

tsqueries could be concatenated using && (AND-ed) and || (OR-ed) operators.
Chapter 1. FTS Introduction

1.2.2. FTS operator

FTS in PostgreSQL provides operator @@ for the two data types - tsquery and tsvector, which represents, correspondingly, document and query. Also, FTS operator has support of TEXT, VARCHAR data types, which allows to setup simple full-text search, but without ranking support.

\text{tsvector} @@ \text{tsquery} \\
\text{tsquery} @@ \text{tsvector} \\
\text{text|varchar} @@ \text{text|tsquery}

Full text search operator @@ returns TRUE if \text{tsvector} contains \text{tsquery}.

=# select 'cat & rat':: tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;  
?column?  
----------
\text{t}  
=# select 'fat & cow':: tsquery @@ 'a fat cat sat on a mat and ate a fat rat'::tsvector;  
?column?  
----------
\text{f}

1.3. Basic operations

To implement full-text search engine we need some functions to obtain \text{tsvector} from a document and \text{tsquery} from user’s query. Also, we need to return results in some order, i.e., we need a function which compare documents in respect to their relevance to the \text{tsquery}. FTS in PostgreSQL provides support of all of these functions, introduced in this section.

1.3.1. Obtaining tsvector

FTS in PostgreSQL provides function to\_tsvector, which transforms document to \text{tsvector} data type. More details is available in Section 2.2, but for now we consider a simple example.

=# select to\_tsvector('english', 'a fat cat sat on a mat - it ate a fat rats');  
to\_tsvector  
-------------------------------------------
'ate':9 'cat':3 'fat':2,11 'mat':7 'rat':12 'sat':4
In the example above we see, that resulted tsvector does not contains a, on, it, word rats became rat and punctuation sign – was ignored.

to_tsvector function internally calls parser function which breaks document (a fat cat sat on a mat - it ate a fat rats) on words and corresponding type. Default parser recognizes 23 types, see Section 2.4 for details. Each word, depending on its type, comes through a stack of dictionaries (Section 1.3.5). At the end of this step we obtain what we call a lexeme. For example, rats became rat, because one of the dictionaries recognized that word rats is a plural form of rat. Some words are treated as a "stop-word" (Section 1.3.6) and ignored, since they are too frequent and have no informational value. In our example these are a, on, it. Punctuation sign – was also ignored, because it’s type (Space symbols) was forbidden for indexing. All information about the parser, dictionaries and what types of lexemes to index contains in the full-text configuration (Section 2.9). It’s possible to have many configurations and actually, many predefined system configurations are available for different languages. In our example we used default configuration english for english language.

To make things clear, below is an output from ts_debug function (Section 2.10), which show all details of FTS machinery.

```sql
=# select * from ts_debug('english','a fat cat sat on a mat - it ate a fat rats');

<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
<th>Token</th>
<th>Dicts list</th>
<th>Lexized token</th>
</tr>
</thead>
<tbody>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>a</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>fat</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {fat}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>cat</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {cat}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>sat</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {sat}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>on</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>a</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>mat</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {mat}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>it</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>ate</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {ate}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>a</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>fat</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {fat}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>rats</td>
<td>{pg_catalog.en_stem}</td>
<td>pg_catalog.en_stem: {rat}</td>
</tr>
</tbody>
</table>

(24 rows)
```

Function setweight() is used to label tsvector. The typical usage of this is to mark out the different parts of document (say, importance). Later, this can be used for ranking of search results in addition to the
positional information (distance between query terms). If no ranking is required, positional information can be removed from tsvector using strip() function to save some space.

Since to_tsvector(NULL) produces NULL, it is recommended to use coalesce to avoid unexpected results. Here is the safe method of obtaining tsvector of structured document.

```
  test=# update tt set ti=
    test=# setweight( to_tsvector(coalesce(title,")), 'A' ) || ' ' ||
    test=# setweight( to_tsvector(coalesce(keyword,")), 'B' ) || ' ' ||
    test=# setweight( to_tsvector(coalesce(abstract,")), 'C' ) || ' ' ||
    test=# setweight( to_tsvector(coalesce(body,")), 'D' );
```

### 1.3.2. Obtaining tsquery

FTS provides two functions for obtaining tsquery - to_tsquery and plainto_tsquery (Section 2.3.2).

```
  select to_tsquery('english', 'fat & rats');
  to_tsquery
  -----------------
    'fat' & 'rat'
  select plainto_tsquery('english', 'fat rats');
  plainto_tsquery
  -----------------
    'fat' & 'rat'
```

Tsquery data type obtained at search time and the same way as tsvector (Section 1.3.1).

There is a powerful technique to rewrite query online, called Query Rewriting (Section 2.3.1). It allows to manage searches on the assumption of application semantics. Typical usage is a synonym extension or changing query to direct search in the necessary direction. The nice feature of Query Rewriting is that it doesn’t require reindexing in contrast of using thesaurus dictionary (Section 2.8.5). Also, Query Rewriting is table-driven, so it can be configured online.

### 1.3.3. Ranking search results

Ranking of search results is de-facto standard feature of all search engines and PostgreSQL FTS provides two predefined ranking functions, which attempt to produce a measure of how a document is relevant to the query. In spite of that the concept of relevancy is vague and is very application specific, these functions try to take into account lexical, proximity and structural information. Detailed description is available (Section 2.5). Different application may require an additional information to rank, for example, document modification time.

Lexical part of ranking reflects how often are query terms in the document, proximity - how close in document query terms are and structural - in what part of document they occur.
Since longer document has a bigger chance to contain a query, it is reasonable to take into account the document size. FTS provides several options for that.

It is important to notice, that ranking functions does not use any global information, so, it is impossible to produce a fair normalization to 1 or 100%, as sometimes required. However, a simple technique, like \( \frac{\text{rank}}{\text{rank}+1} \) can be applied. Of course, this is just a cosmetic change, i.e., ordering of search results will not changed.

Several examples are shown below. Notice, that second example used normalized rank.

```sql
=# select title, rank_cd('{0.1, 0.2, 0.4, 1.0}',fts, query) as rnk
from apod, to_tsquery('neutrino|(dark & matter)') query
where query @@ fts order by rnk desc limit 10;
```

<table>
<thead>
<tr>
<th>title</th>
<th>rnk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neutrinos in the Sun</td>
<td>3.1</td>
</tr>
<tr>
<td>The Sudbury Neutrino Detector</td>
<td>2.4</td>
</tr>
<tr>
<td>A MACHO View of Galactic Dark Matter</td>
<td>2.01317</td>
</tr>
<tr>
<td>Hot Gas and Dark Matter</td>
<td>1.91171</td>
</tr>
<tr>
<td>The Virgo Cluster: Hot Plasma and Dark Matter</td>
<td>1.90953</td>
</tr>
<tr>
<td>Rafting for Solar Neutrinos</td>
<td>1.9</td>
</tr>
<tr>
<td>NGC 4650A: Strange Galaxy and Dark Matter</td>
<td>1.85774</td>
</tr>
<tr>
<td>Hot Gas and Dark Matter</td>
<td>1.6123</td>
</tr>
<tr>
<td>Ice Fishing for Cosmic Neutrinos</td>
<td>1.6</td>
</tr>
<tr>
<td>Weak Lensing Distorts the Universe</td>
<td>1.85218</td>
</tr>
</tbody>
</table>

```sql
=# select title, rank_cd('{0.1, 0.2, 0.4, 1.0}',fts, query)/(rank_cd('{0.1, 0.2, 0.4, 1.0}',fts, query) + 1) as rnk from apod, to_tsquery('neutrino|(dark & matter)') query where query @@ fts order by rnk desc limit 10;
```

<table>
<thead>
<tr>
<th>title</th>
<th>rnk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neutrinos in the Sun</td>
<td>0.756097569485493</td>
</tr>
<tr>
<td>The Sudbury Neutrino Detector</td>
<td>0.705882361190954</td>
</tr>
<tr>
<td>A MACHO View of Galactic Dark Matter</td>
<td>0.668123210574724</td>
</tr>
<tr>
<td>Hot Gas and Dark Matter</td>
<td>0.65655958650282</td>
</tr>
<tr>
<td>The Virgo Cluster: Hot Plasma and Dark Matter</td>
<td>0.656301290640973</td>
</tr>
<tr>
<td>Rafting for Solar Neutrinos</td>
<td>0.655172410958162</td>
</tr>
<tr>
<td>NGC 4650A: Strange Galaxy and Dark Matter</td>
<td>0.650072921219637</td>
</tr>
<tr>
<td>Hot Gas and Dark Matter</td>
<td>0.617195790024749</td>
</tr>
<tr>
<td>Ice Fishing for Cosmic Neutrinos</td>
<td>0.615384618911517</td>
</tr>
<tr>
<td>Weak Lensing Distorts the Universe</td>
<td>0.450010798361481</td>
</tr>
</tbody>
</table>

First argument in `rank_cd` is an optional parameter, which specifies actual weights for labels D,C,B,A, used in function `setweight`. These default values show that lexemes labeled as A are 10 times important than one with label D.

Ranking could be expensive, since it requires consulting `tsvector` of all found documents, which is IO bound and slow. Unfortunately, it is almost impossible to avoid, since FTS in databases should works without index, moreover, index could be lossy (GiST index, for example), so it requires to check documents to avoid false hits. External search engines doesn’t suffer from this, because ranking information usually contain in the index itself and it is not needed to read documents.

7
Chapter 1. FTS Introduction

1.3.4. Getting results

To present search results it is desirable to show part(s) of documents which somehow identify its context and how it is related to the query. Usually, search engines show fragments of documents with marked search terms. FTS provides function `headline()` (see details in Section 2.6) for this. It uses original document, not `tsvector`, so it is rather slow and should be used with care. Typical mistake is to call `headline()` for all found documents, while usually one need only 10 or so documents to show. SQL subselects help here. Below is an example of that.

```sql
SELECT id,headline(body,q),rank
FROM ( SELECT id,body,q, rank_cd (ti,q) AS rank FROM apod, to_tsquery('stars') q
    WHERE ti @@ q ORDER BY rank DESC LIMIT 10) AS foo;
```

1.3.5. Dictionaries

Dictionary is a *program*, which accepts lexeme(s) on input and returns:

- array of lexeme(s) if input lexeme is known to the dictionary
- void array - dictionary knows lexeme, but it’s stop word.
- NULL - dictionary doesn’t recognized input lexeme

**WARNING:** Data files, used by dictionaries, should be in `server_encoding` to avoid possible problems!

Usually, dictionaries used for normalization of words and allows user to not bother which word form use in query. Also, normalization can reduce a size of `tsvector`. Normalization not always has linguistic meaning and usually depends on application semantics.

Some examples of normalization:

- Linguistic - ispell dictionaries try to reduce input word to its infinitive, stemmer dictionaries remove word ending.
- All URL-s are equivalent to the http server:
  - `http://www.pgsqul.ru/db/mw/index.html`
  - `http://www.pgsqul.ru/db/mw/
  - `http://www.pgsqul.ru/db/../db/mw/index.html`
- Colour names substituted by their hexadecimal values - red,green,blue, magenta -> FF0000, 00FF00, 0000FF, FF00FF
- Cut fractional part to reduce the number of possible numbers, so 3.14159265359, 3.1415926, 3.14 will be the same after normalization, if leave only two numbers after period. See dictionary for integers (Appendix C) for more details.

FTS provides several predefined dictionaries (Section 2.8), available for many languages, and SQL commands to manipulate them online (Part I). Besides this, it is possible to develop custom dictionaries using API, see dictionary for integers Appendix C, for example.
CREATE FULLTEXT MAPPING command ( CREATE FULLTEXT MAPPING ) binds specific type of lexeme and a set of dictionaries to process it. Lexeme come through a stack of dictionaries until some dictionary identify it as a known word or found it is a stop-word. If no dictionary will recognize a lexeme, than it will be discarded and not indexed. A general rule for configuring stack of dictionaries is to place at first place the most narrow, most specific dictionary, then more general dictionary and finish it with very general dictionary, like snowball stemmer or simple, which recognize everything. For example, for astronomy specific search (astro_en configuration) one could bind lword (latin word) with synonym dictionary of astronomical terms, general english dictionary and snowball english stemmer.

```sql
=# CREATE FULLTEXT MAPPING ON astro_en FOR lword WITH astrosyn, en_ispell, en_stem;
```

Function `lexize` can be used to test dictionary, for example:

```sql
=# select lexize('en_stem', 'stars');
lexize
--------
{star}
(1 row)
```

Also, `ts_debug` function (Section 2.10) is very useful.

### 1.3.6. Stop words

Stop words are the words, which are too popular and appear almost in every document and have no discrimination value, so they could be ignored in full-text index. For example, every english text contains word a and it is useless to have it in index. However, stop words does affect to the positions in `tsvector`, which in turn, does affect ranking.

```sql
=# select to_tsvector('english','in the list of stop words');
to_tsvector
-----------------------------
'list':3 'stop':5 'word':6
```

The gaps between positions 1-3 and 3-5 are because of stop words, so ranks, calculated for document with/without stop words, are quite different!

```sql
=# select rank_cd ('{1,1,1,1}', to_tsvector('english','in the list of stop words'), to_tsvector('english','list stop words'));
rank_cd
---------
0.5
```

```sql
postgres=# select rank_cd ('{1,1,1,1}', to_tsvector('english','list stop words'), to_tsvector('english','in the list of stop words'));
rank_cd
---------
1
```
Chapter 1. FTS Introduction

It is up to the specific dictionary, how to treat stop-words. For example, ispell dictionaries first normalized word and then lookups it in the list of stop words, while stemmers first lookups input word in stop words. The reason for such different behaviour is an attempt to decrease a possible noise.

1.4. FTS features

Full text search engine in PostgreSQL is fully integrated into the database core. Its main features are:

- It is mature, more than 5 years of development
- Supports multiple configurations, which could be managed using a set of SQL commands.
- Flexible and rich linguistic support using pluggable user-defined dictionaries with stop words supports. Several predefined templates, including ispell, snowball, Thesaurus and synonym dictionaries, are supplied.
- Full multibyte support, UTF-8 as well
- Sophisticated ranking functions with support of proximity and structure information allow ordering of search results according their similarity to the query.
- Index support with concurrency and recovery support
- Rich query language with query rewriting support

1.5. FTS Limitations

Current implementation of FTS has some limitations.

- Length of lexeme < 2K
- Length of tsvector (lexemes + positions) < 1Mb
- The number of lexemes < 4\(^32\)
- 0< Positional information < 16383
- No more than 256 positions per lexeme
- The number of nodes (lexemes + operations) in tquery < 32768

For comparison, PostgreSQL 8.1 documentation consists of 10441 unique words, total 335420 words and most frequent word 'postgreSQL' mentioned 6127 times in 655 documents.
Another example - PostgreSQL mailing list archive consists of 910989 unique words, total 57,491,343 lexemes in 461020 messages.
Chapter 1. FTS Introduction

1.6. A Brief History of FTS in PostgreSQL

This is a historical notes about full-text search in PostgreSQL by authors of FTS Oleg Bartunov and Teodor Sigaev.

1.6.1. Pre-tsearch

Development of full-text search in PostgreSQL began from OpenFTS\(^1\) in 2000 after realizing that we need a search engine optimized for online updates with access to metadata from the database. This is essential for online news agencies, web portals, digital libraries, etc. Most search engines available at that time utilize an inverted index which is very fast for searching but very slow for online updates. Incremental updates of an inverted index is a complex engineering task while we needed something light, free and with the ability to access metadata from the database. The last requirement was very important because in a real life search application should always consult metadata (topic, permissions, date range, version, etc.).

We extensively use PostgreSQL as a database backend and have no intention to move from it, so the problem was to find a data structure and a fast way to access it. PostgreSQL has rather unique data type for storing sets (think about words) - arrays, but lacks index access to them. During our research we found a paper of Joseph Hellerstein, who introduced an interesting data structure suitable for sets - RD-tree (Russian Doll tree). Further research lead us to the idea to use GiST for implementing RD-tree, but at that time the GiST code was untouched for a long time and contained several bugs. After work on improving GiST for version 7.0.3 of PostgreSQL was done, we were able to implement RD-Tree and use it for index access to arrays of integers. This implementation was ideally suited for small arrays and eliminated complex joins, but was practically useless for indexing large arrays. The next improvement came from an idea to represent a document by a single bit-signature, a so-called superimposed signature (see "Index Structures for Databases Containing Data Items with Set-valued Attributes", 1997, Sven Helmer for details). We developed the contrib/intarray module and used it for full text indexing.

1.6.2. Tsearch v1

It was inconvenient to use integer id’s instead of words, so we introduced a new data type txtidx - a searchable data type (textual) with indexed access. This was a first step of our work on an implementation of a built-in PostgreSQL full-text search engine. Even though tsearch v1 had many features of a search engine it lacked configuration support and relevance ranking. People were encouraged to use OpenFTS, which provided relevance ranking based on positional information and flexible configuration. OpenFTS v:0.34 was the last version based on tsearch v1.

1.6.3. Tsearch v2

People recognized tsearch as a powerful tool for full text searching and insisted on adding ranking support, better configurability, etc. We already thought about moving most of the features of OpenFTS to tsearch, and in the early 2003 we decided to work on a new version of tsearch. We abandoned auxiliary index tables, used by OpenFTS to store positional information, and modified the txtidx type to store them.

\(^1\) http://openfts.sourceforge.net
internally. We added table-driven configuration, support of ispell dictionaries, snowball stemmers and the ability to specify which types of lexemes to index. Now, it’s possible to generate headlines of documents with highlighted search terms. These changes make tsearch user friendly and turn it into a really powerful full text search engine. For consistency, tsearch functions were renamed, `txtidx` type became `tsvector`. To allow users of tsearch v1 smooth upgrade, we named the module as tsearch2. Since version 0.35 OpenFTS uses tsearch2.

PostgreSQL version 8.2 contains a major upgrade of tsearch v2 - multibyte and GIN (A Generalized Inverted Index) support. Multibyte support provides full UTF-8 support and GIN scales tsearch v2 to millions of documents. Both indices (GiST and GiN) are concurrent and recoverable. All these improvements bring out FTS to enterprise level.

### 1.6.4. FTS current

Since PostgreSQL 8.3 release, there is no need to compile and install contrib/tsearch2 module, it’s already installed in your system with PostgreSQL. Most important new features are:

- A set of SQL commands, which controls creation, modification and dropping of FTS objects. This allow to keep dependencies and correct dumping and dropping.
- Many FTS configurations already predefined for different languages with snowball stemmers are available.
- FTS objects now have ownership and namespace support like other postgresql’s objects.
- Current FTS configuration could be defined using GUC variable `tsearch_conf_name`.
- Default FTS configuration is now schema specific.

### 1.7. Links

Tsearch2[^2]

An Official Home page of Tsearch2.

Tsearch Wiki[^3]

Tsearch2 Wiki contains many informations, work in progress.

OpenFTS[^4]

OpenFTS search engine

OpenFTS mailing list[^5]

OpenFTS-general mailing list used for discussion about OpenFTS itself and FTS in PostgreSQL.

[^2]: http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2
[^3]: http://www.sai.msu.su/~megera/wiki/Tsearch2
[^4]: http://openfts.sourceforge.net
[^5]: http://lists.sourceforge.net/lists/listinfo/openfts-general
1.8. FTS Todo

This place reserved for FTS development plan.

1.9. Acknowledgements

The work on developing of FTS in PostgreSQL was supported by several companies and authors are glad to express their gratitude to the University of Mannheim, jfg:networks, Georgia Public Library Service and LibLime Inc., Enterprisedb PostgreSQL Development Fund, Russian Foundation for Basic Research, Rambler Internet Holding.
Chapter 2. FTS Operators and Functions

Vectors and queries both store lexemes, but for different purposes. A tsvector stores the lexemes of the words that are parsed out of a document, and can also remember the position of each word. A tsquery specifies a boolean condition among lexemes.

Any of the following functions with a configuration argument can use either an integer id or textual ts_name to select a configuration; if the option is omitted, then the current configuration is used. For more information on the current configuration, read the next section on Section 2.9.

2.1. FTS operator

TSQUERY @@ TSVECTOR
TSVECTOR @@ TSQUERY

Returns TRUE if TSQUERY contained in TSVECTOR and FALSE otherwise.

```sql
=# select 'cat & rat':: tsquery @@ 'a fat cat sat on a mat and ate a fat rat':tsvector;

?column?
----------

= t

=# select 'fat & cow':: tsquery @@ 'a fat cat sat on a mat and ate a fat rat':tsvector;

?column?
----------

= f
```

TEXT @@ TSQUERY
VARCHAR @@ TSQUERY

Returns TRUE if TSQUERY contained in TEXT/VARCHAR and FALSE otherwise.

```sql
=# select 'a fat cat sat on a mat and ate a fat rat':text @@ 'cat & rat': tsquery;

?column?
----------

= t

=# select 'a fat cat sat on a mat and ate a fat rat':text @@ 'cat & cow': tsquery;

?column?
----------

= f
```

TEXT @@ TEXT
VARCHAR @@ TEXT

Returns TRUE if TEXT contained in TEXT/VARCHAR and FALSE otherwise.

```sql
postgres=# select 'a fat cat sat on a mat and ate a fat rat' @@ 'cat rat';

?column?
----------

= t

postgres=# select 'a fat cat sat on a mat and ate a fat rat' @@ 'cat cow';

?column?
----------
```
Chapter 2. FTS Operators and Functions

2.2. Vector Operations

to_tsvector( [configuration,] document TEXT) RETURNS TSVECTOR

Parses a document into tokens, reduces the tokens to lexemes, and returns a tsvector which lists the lexemes together with their positions in the document in lexicographic order.

strip(vector TSVECTOR) RETURNS TSVECTOR

Return a vector which lists the same lexemes as the given vector, but which lacks any information about where in the document each lexeme appeared. While the returned vector is thus useless for relevance ranking, it will usually be much smaller.

setweight(vector TSVECTOR, letter) RETURNS TSVECTOR

This function returns a copy of the input vector in which every location has been labeled with either the letter ‘A’, ‘B’, or ‘C’, or the default label ‘D’ (which is the default with which new vectors are created, and as such is usually not displayed). These labels are retained when vectors are concatenated, allowing words from different parts of a document to be weighted differently by ranking functions.

vector1 ||| vector2

concat(vector1 TSVECTOR, vector2 TSVECTOR) RETURNS TSVECTOR

Returns a vector which combines the lexemes and position information in the two vectors given as arguments. Position weight labels (described in the previous paragraph) are retained intact during the concatenation. This has at least two uses. First, if some sections of your document need be parsed with different configurations than others, you can parse them separately and concatenate the resulting vectors into one. Second, you can weight words from some sections of you document more heavily than those from others by: parsing the sections into separate vectors; assigning the vectors different position labels with the setweight() function; concatenating them into a single vector; and then providing a weights argument to the rank() function that assigns different weights to positions with different labels.

length(vector TSVECTOR) RETURNS INT4

Returns the number of lexemes stored in the vector.

text::TSVECTOR RETURNS TSVECTOR

Directly casting text to a tsvector allows you to directly inject lexemes into a vector, with whatever positions and position weights you choose to specify. The text should be formatted like the vector would be printed by the output of a SELECT.

tsearch(vector_column_name[, (my_filter_name | text_column_name1) [...] ], text_column_nameN)

tsearch() trigger used to automatically update vector_column_name, my_filter_name is the function name to preprocess text_column_name. There are can be many functions and
text columns specified in `tsearch()` trigger. The following rule used: function applied to all
subsequent text columns until next function occurs. Example, function `dropatsymbol` replaces all
entries of @ sign by space.

```sql
CREATE FUNCTION dropatsymbol(text) RETURNS text
AS 'select replace($1, "@", " ");'
LANGUAGE SQL;

CREATE TRIGGER tsvectorupdate BEFORE UPDATE OR INSERT
ON tblMessages FOR EACH ROW EXECUTE PROCEDURE
  tsearch(tsvector_column,dropatsymbol, strMessage);
```

Here `statinfo` is a type, defined as

```sql
CREATE TYPE statinfo as (word text, ndoc int4, nentry int4);
```

and `sqlquery` is a query, which returns column `tsvector`. This returns statistics (the number of
documents `ndoc` and total number `nentry` of word in the collection) about column vector `tsvector`.
Useful to check how good is your configuration and to find stop-words candidates. For example, find
top 10 most frequent words:

```sql
=# select * from stat('select vector from apod') order by ndoc desc, nentry desc,word limit 10;
```

Optionally, one can specify `weight` to obtain statistics about words with specific `weight`.

```sql
=# select * from stat('select vector from apod','a') order by ndoc desc, nentry desc,word limit 10;
```

TSVECTOR < TSVECTOR
TSVECTOR <= TSVECTOR
TSVECTOR = TSVECTOR
TSVECTOR >= TSVECTOR
TSVECTOR > TSVECTOR

All btree operations defined for `tsvector` type. tsvectors compares with each other using lexico-
graphical order.

### 2.3. Query Operations

`to_tsquery( [configuration,] querytext text) RETURNS TSQUERY`

Accepts `querytext`, which should be a single tokens separated by the boolean operators & and, | or, and ! not, which can be grouped using parenthesis. In other words, `to_tsquery` expects already
parsed text. Each token is reduced to a lexeme using the current or specified configuration. Weight
class can be assigned to each lexeme entry to restrict search region (see `setweight` for explanation),
for example

```
'fat:a & rats'
```

to_tsquery function could accept `text string`. In this case `querytext` should be quoted. This
may be useful, for example, to use with thesaurus dictionary. In example below, thesaurus contains
rule `supernovae stars : sn`.

```sql
=# select to_tsquery("'supernovae stars" & !crab");
```

```sql
to_tsquery
```

16
Without quotes to_tsquery will complain about syntax error.

plainto_tsquery( [configuration,] querytext text) RETURNS TSQUERY

Transforms unformatted text querytext to tsquery. It is the same as to_tsquery, but accepts text and will call parser to break it onto tokens. plainto_tsquery assumes & boolean operator between words and doesn’t recognizes weight classes.

querytree(query TSQUERY) RETURNS text

This returns a query which actually used in searching in index. It could be used to test for an empty query. Select below returns ’T’, which corresponds to empty query, since GIN index doesn’t supports negate query and full index scan is very ineffective.

---

=# select querytree( to_tsquery(‘!defined’) ) ;
querytree
----------
T

plainto_tsquery(text::TSQUERY) RETURNS TSQUERY

Directly casting text to a tsquery allows you to directly inject lexemes into a query, with whatever positions and position weight flags you choose to specify. The text should be formatted like the query would be printed by the output of a SELECT.

numnode(query TSQUERY) RETURNS INTEGER

This returns the number of nodes in query tree. This function could be used to resolve if query is meaningful (returns > 0), or contains only stop-words (returns 0).

---

=# select numnode( plainto_tsquery(‘the any’) ) ;
numnode
-------
0
=# select numnode( plainto_tsquery(‘the table’) ) ;
numnode
-------
1
=# select numnode( plainto_tsquery(‘long table’) ) ;
numnode
-------
3

TSQUERY && TSQUERY RETURNS TSQUERY

Returns AND-ed TSQUERY

TSQUERY || TSQUERY RETURNS TSQUERY

Returns OR-ed TSQUERY
Chapter 2. FTS Operators and Functions

!! TSQUERY RETURNS TSQUERY

negation of TSQUERY

TSQUERY < TSQUERY
TSQUERY <= TSQUERY
TSQUERY = TSQUERY
TSQUERY >= TSQUERY
TSQUERY > TSQUERY

All btree operations defined for tsquery type. tsqueries compares with each other using lexicographical order.

2.3.1. Query rewriting

Query rewriting is a set of functions and operators for tsquery type. It allows to control search at query time without reindexing (opposite to thesaurus), for example, expand search using synonyms (new york, big apple, nyc, gotham) or narrow search directing user to some hot topic. Notice, that rewriting rules can be added online.

rewrite() function changes original query by replacing part of the query by sample string of type tsquery, as it defined by rewrite rule. Arguments of rewrite() function can be column names of type tsquery.

CREATE TABLE aliases (t TSQUERY primary key, s TSQUERY);
INSERT INTO aliases values('a', 'c');

rewrite (query TSQUERY, target TSQUERY, sample TSQUERY) RETURNS TSQUERY

=# select rewrite('a & b'::TSQUERY, 'a'::TSQUERY, 'c'::TSQUERY);
rewrite
-------------
'b' & 'c'

rewrite (ARRAY[query TSQUERY, target TSQUERY, sample TSQUERY]) RETURNS TSQUERY

=# select rewrite(ARRAY['a & b'::TSQUERY, t,s]) from aliases;
rewrite
-------------
'b' & 'c'

rewrite (query TSQUERY,'select target ,sample from test'::text) RETURNS TSQUERY

=# select rewrite('a & b'::TSQUERY, 'select t,s from aliases');
rewrite
-------------
'b' & 'c'

What if there are several variants of rewriting ? For example, query 'a & b' can be rewritten as 'b & c' and 'cc'.

=# select * from aliases;
   t | s
  ---+---
   18
Chapter 2. FTS Operators and Functions

---

| 'a' | 'c' |
| 'x' | 'z' |
| 'a' & 'b' | 'cc' |

This ambiguity can be resolved specifying sort order.

```sql
=# select rewrite('a & b', 'select t,s from aliases order by t desc');
rewrite
-----------
'cc'
```

```sql
=# select rewrite('a & b', 'select t,s from aliases order by t asc');
rewrite
-----------
'b' & 'c'
```

Let's consider real-life astronomical example. We'll expand query `supernovae` using table-driven rewriting rules.

```sql
=# create table aliases (t tsquery primary key, s tsquery);
=# insert into aliases values(to_tsquery('supernovae'), to_tsquery('supernovae|sn'));
=# select rewrite(to_tsquery('supernovae'), 'select * from aliases') && to_tsquery('crab');
?
column?
---------------------------------
('supernova' | 'sn') & 'crab'
```

Notice, that we can change rewriting rule online!

```sql
=# update aliases set s=to_tsquery('supernovae|sn&!nebulae') where t=to_tsquery('supernovae');
=# select rewrite(to_tsquery('supernovae'), 'select * from aliases') && to_tsquery('crab');
?
column?
---------------------------------------------
('supernova' | 'sn' & '!nebula') & 'crab'
```

### 2.3.2. Operators for tsquery

Rewriting can be slow in case of many rewriting rules, since it checks every rule for possible hit. To filter out obvious non-candidate rules there are containment operators for `tsquery` type. In example below, we select only those rules, which might contains in the original query.

```sql
=# select rewrite(ARRAY['a & b::TSQUERY, t,s]) from aliases where 'a&b' @> t;
rewrite
-----------
'b' & 'c'
```

Two operators defined for `tsquery` type:
Chapter 2. FTS Operators and Functions

TSQUERY @> TSQUERY

Returns TRUE if right argument might contained in left argument.

TSQUERY <@ TSQUERY

Returns TRUE if left argument might contained in right argument.

### 2.3.3. Index for tsquery

To speed up operators <@, @> for tsquery one can use GiST index with tsquery_ops opclass.

```sql
create index t_idx on aliases using gist (t tsquery_ops);
```

### 2.4. Parser functions

```sql
CREATE FUNCTION parse( parser, document TEXT ) RETURNS SETOF tokenout

Parses the given document and returns a series of records, one for each token produced by parsing. Each record includes a tokid giving its type and a token which gives its content.

postgres=# select * from parse('default','123 - a number');

<table>
<thead>
<tr>
<th>tokid</th>
<th>token</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>123</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>a</td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>number</td>
</tr>
</tbody>
</table>
```

```sql
CREATE FUNCTION token_type( parser ) RETURNS SETOF tokentype

Returns a table which defines and describes each kind of token the parser may produce as output. For each token type the table gives the tokid which the parser will label each token of that type, the alias which names the token type, and a short description for the user to read.

postgres=# select * from token_type('default');

<table>
<thead>
<tr>
<th>tokid</th>
<th>alias</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>lword</td>
<td>Latin word</td>
</tr>
<tr>
<td>2</td>
<td>nlword</td>
<td>Non-latin word</td>
</tr>
<tr>
<td>3</td>
<td>word</td>
<td>Word</td>
</tr>
<tr>
<td>4</td>
<td>email</td>
<td>Email</td>
</tr>
<tr>
<td>5</td>
<td>url</td>
<td>URL</td>
</tr>
<tr>
<td>6</td>
<td>host</td>
<td>Host</td>
</tr>
<tr>
<td>7</td>
<td>sfloat</td>
<td>Scientific notation</td>
</tr>
<tr>
<td>8</td>
<td>version</td>
<td>VERSION</td>
</tr>
</tbody>
</table>
```
Chapter 2. FTS Operators and Functions

2.5. Ranking

Ranking attempts to measure how relevant documents are to particular queries by inspecting the number of times each search word appears in the document, and whether different search terms occur near each other. Note that this information is only available in unstripped vectors -- ranking functions will only return a useful result for a tsvector which still has position information!

Notice, that ranking functions supplied are just an examples and doesn’t belong to the FTS core, you can write your very own ranking function and/or combine additional factors to fit your specific interest.

The two ranking functions currently available are:

CREATE FUNCTION rank( [ weights float4[] ], vector TSVECTOR, query TSQUERY, [ normalization int4 ] ) RETURNS float4

This is the ranking function from the old version of OpenFTS, and offers the ability to weight word instances more heavily depending on how you have classified them. The weights specify how heavily to weight each category of word:

\{D-weight, C-weight, B-weight, A-weight\}

If no weights are provided, then these defaults are used:

\{0.1, 0.2, 0.4, 1.0\}

Often weights are used to mark words from special areas of the document, like the title or an initial abstract, and make them more or less important than words in the document body.

CREATE FUNCTION rank_cd( [ weights float4[] ], vector TSVECTOR, query TSQUERY, [ normalization int4 ] ) RETURNS float4

This function computes the cover density ranking for the given document vector and query, as described in Clarke, Cormack, and Tudhope's "Relevance Ranking for One to Three Term Queries" in the 1999 Information Processing and Management.

Both of these ranking functions take an integer normalization option that specifies whether a document’s length should impact its rank. This is often desirable, since a hundred-word document with five instances of a search word is probably more relevant than a thousand-word document with five instances. The option can have the values, which could be combined using | (for example, 2 | 4) to take into account several factors:
Chapter 2. FTS Operators and Functions

- 0 (the default) ignores document length.
- 1 divides the rank by the 1 + logarithm of the document length
- 2 divides the rank by the length itself.
- 4 divides the rank by the mean harmonic distance between extents
- 8 divides the rank by the number of unique words in document
- 16 divides the rank by 1 + logarithm of the number of unique words in document

2.6. Headline

CREATE FUNCTION headline([ id int4, ] ts_name text, [ document text, ] query TSQUERY, [ options text ])
RETURNS text

Every form of the the headline() function accepts a document along with a query, and returns one or more ellipse-separated excerpts from the document in which terms from the query are highlighted. The configuration with which to parse the document can be specified by either its id or ts_name; if none is specified that the current configuration is used instead.

An options string if provided should be a comma-separated list of one or more 'option=value' pairs. The available options are:

- StartSel, StopSel -- the strings with which query words appearing in the document should be delimited to distinguish them from other excerpted words.
- MaxWords, MinWords -- limits on the shortest and longest headlines you will accept.
- ShortWord -- this prevents your headline from beginning or ending with a word which has this many characters or less. The default value of 3 should eliminate most English conjunctions and articles.
- HighlightAll -- boolean flag, if TRUE, than the whole document will be highlighted.

Any unspecified options receive these defaults:

StartSel=<b>, StopSel=<b>, MaxWords=35, MinWords=15, ShortWord=3, HighlightAll=FALSE

Notice, that cascade dropping of headline function cause dropping of parser, used in fulltext configuration tsname.

```sql
select headline('a b c', 'c':tsquery);
-- headline
---------------
a b <b>c</b>
```
2.7. Full-text indexes

There are two kinds of indexes which can be used to speedup FTS operator (Section 2.1). Notice, indexes are not mandatory for FTS!

```
CREATE INDEX name ON table USING gist(column);
```

Creates GiST (The Generalized Search Tree) based index.

```
CREATE INDEX name ON table USING gin(column);
```

Creates GIN (The Generalized Inverted Index) based index. \texttt{column} is one of the \texttt{TSVECTOR}, or \texttt{TEXT}, or \texttt{VARCHAR} types.

GiST index is lossy, which means it’s required to consult heap to check results for false hits. PostgreSQL does this automatically. Filter: in an example below.

```
=# explain select * from apod where fts @@ to_tsquery('supernovae');

QUERY PLAN
------------------------------------------------------------------------------------------------
Index Scan using fts_gidx on apod (cost=0.00..12.29 rows=2 width=1469)
  Index Cond: (fts @@ '"supernova"':tsquery)
  Filter: (fts @@ '"supernova"':tsquery)
```

Lossiness is the result of two factors - we represent a document by its fixed-length signature. We obtain signature in the following way - we hash (crc32) each word into random bit in a n-bit length strings and their superposition produces n-bit document signature. Because of hashing, there is a chance, that some words hashed to the same position and it could be resulted in false hit. Signatures, calculated for each document in collection, are stored in RD-tree (Russian Doll tree), invented by Hellerstein, which is an adaptation of the R-tree to sets. In our case transitive containment relation realized with superimposed coding (Knuth,1973) of signatures - parent is ‘OR’-ed bit-strings of all children. This is a second factor of lossiness. It’s clear, that parents tend to be full of ‘1’ (degenerates) and become quite useless because of it’s little selectivity. Searching performs as a bit comparison of a signature represented query and RD-tree entry. If all ‘1’ of both signatures are in the same position we say that this branch probably contains query, but if there is even one discrepancy we could definitely reject this branch. Lossiness causes serious performance degradation, since random accessing of heap records is slow and limits applicability of GiST index. Probability of false drops is depends on several factors and the number of unique words is one of them, so using dictionaries to reduce this number is practically mandatory.

Actually, it’s not the whole story. GiST index has optimization for storing small tsvectors (< \texttt{TOAST_INDEX_TARGET} bytes, 512 bytes). On leaf pages small tsvectors stored as is, while longer one are represented by their signatures, which introduce some losiness. Unfortunately, existing index API doesn’t allow to say index that it found an exact values (tsvector) or results need to be checked. That’s why GiST index currently is marked as lossy. We hope in future to overcome this issue.

Contrary, GIN index isn’t lossy and it’s performance depends logarithmically on the number of unique words.

There is one side-effect of "non-lossiness" of GIN index and using queries with lexemes and weights, like ‘\texttt{supernovae:a}’. Since information about these labels stored in heap only and GIN index is not lossy, there is no necessity to access heap, one should use special FTS operator @@, which forces using
Chapter 2. FTS Operators and Functions

of `heap` to get information about labels. GiST index is lossy, so it reads `heap` anyway and there is no need in special operator. In example below, `fts_idx` is a GIN index.

```sql
=# explain select * from apod where fts @@ to_tsquery('supernovae:a');

QUERY PLAN
--------
Index Scan using fts_idx on apod (cost=0.00..12.30 rows=2 width=1469)
  Index Cond: (fts @@ "supernova":"A":tsquery)
  Filter: (fts @@ "supernova":"A":tsquery)
```

Experiments lead to the following observations:

- creation time - GiN takes 3x time to build than GiST
- size of index - GiN is 2-3 times bigger than GiST
- search time - GiN is 3 times faster than GiST
- update time - GiN is about 10 times slower than GiST

Overall, GiST index is very good for online update and fast for collections with the number of unique words about 100,000, but is not as scalable as Gin index, which in turn isn’t good for updates. Both indexes support concurrency and recovery.

Partitioning of big collections and proper use of GiST and GIN indexes allow implementation of very fast search with online update. Partitioning can be done on database level using table inheritance and Constraint Exclusion, or distributing documents over servers and collecting search results using contrib/dblink extension module. The latter is possible, because ranking functions use only local information.

2.8. Dictionaries

```sql
CREATE FUNCTION lexize([ oid, ] dict_name text, lexeme text) RETURNS text[]

Returns an array of lexemes if input lexeme is known to the dictionary dictname, or void array if a lexeme is known to the dictionary, but it is a stop-word, or NULL if it is unknown word.

=# select lexize('en_stem', 'stars');
lexize
--------
{star}

=# select lexize('en_stem', 'a');
lexize
--------
{}
```

**Note:** lexize function expects lexeme, not text! Below is a didactical example:

```sql
apod=$# select lexize('tz_astro','supernovae stars') is null;
?column?
```
Chapter 2. FTS Operators and Functions

---

The Thesaurus dictionary tz_astro does know what is a supernovae stars, but lexize fails, since it does not parse input text and considers it as a single lexeme. Use plainto_tsquery, to_tsvector to test thesaurus dictionaries.

```sql
apod=# select plainto_tsquery('supernovae stars');
plainto_tsquery
-----------------
'sn'
```

There are several predefined dictionaries and templates. Templates used to create new dictionaries overriding default values of parameters. FTS Reference Part I contains description of SQL commands (CREATE FULLTEXT DICTIONARY, DROP FULLTEXT DICTIONARY, ALTER FULLTEXT DICTIONARY) for managing of dictionaries.

### 2.8.1. Simple dictionary

This dictionary returns lowercased input word or NULL if it is a stop-word. Example of how to specify location of file with stop-words.

```sql
=# CREATE FULLTEXT DICTIONARY public.my_simple
   OPTION 'english.stop'
   LIKE pg_catalog.simple;
```

Relative paths in OPTION resolved respective to $PGROOT/share. Now we could test our dictionary:

```sql
=# select lexize('public.my_simple','YeS');
lexize
---------
{yes}
=# select lexize('public.my_simple','The');
lexize
---------
{}
```

### 2.8.2. Ispell dictionary

Ispell template dictionary for FTS allows creation of morphological dictionaries, based on Ispell\(^1\), which has support for a large number of languages. This dictionary try to reduce an input word to its infinitive

---

form. Also, more modern spelling dictionaries are supported - MySpell\(^2\) (OO < 2.0.1) and Hunspell\(^3\) (OO \(\geq 2.0.2\)). A big list of dictionaries is available on OpenOffice Wiki\(^4\).

Ispell dictionary allow search without bothering about different linguistic forms of a word. For example, a search on bank would return hits to all declensions and conjugations of the search term bank - banking, banked, banks, banks’ and bank’s etc.

```sql
=# select lexize('en_ispell','banking');
lexize
--------
{bank}
=# select lexize('en_ispell','bank”s');
lexize
--------
{bank}
=# select lexize('en_ispell','banked');
lexize
--------
{bank}
```

To create ispell dictionary one should use built-in ispell_template dictionary and specify several parameters.

```sql
CREATE FULLTEXT DICTIONARY en_ispell
OPTION 'DictFile="/usr/local/share/dicts/ispell/english.dict",
       AffFile="/usr/local/share/dicts/ispell/english.aff",
       StopFile="/usr/local/share/dicts/ispell/english.stop"
LIKE ispell_template;
```

Here, DictFile, AffFile, StopFile are location of dictionary files and file with stop words. Relative paths in OPTION resolved respective to \$PGROOT/share/dicts_data.

```sql
CREATE FULLTEXT DICTIONARY en_ispell
OPTION 'DictFile="ispell/english.dict",
       AffFile="ispell/english.aff",
       StopFile="english.stop"
LIKE ispell_template;
```

Ispell dictionary usually recognizes a restricted set of words, so it should be used in conjunction with another "broader" dictionary, for example, stemming dictionary, which recognizes "everything".

Ispell dictionary has support for splitting compound words based on an ispell dictionary. This is a nice feature and FTS in PostgreSQL supports it. Notice, that affix file should specify special flag with the compoundwords controlled statement, which used in dictionary to mark words participated in compound formation.

```sql
compoundwords  controlled z
```

---

3. [http://sourceforge.net/projects/hunspell](http://sourceforge.net/projects/hunspell)

26
Chapter 2. FTS Operators and Functions

Several examples for Norwegian language:

```verbatim
=## select lexize('norwegian_ispell','overbuljongterningpakkmesterassistent');
{over,buljong,terning,pakk,mester,assistent}
=## select lexize('norwegian_ispell','sjokoladefabrikk');
{sjokoladefabrikk,sjokolade,fabrikk}
```

**Note:** MySpell doesn’t supports compound words, Hunspell has sophisticated support of compound words. At present, FTS implements only basic compound word operations of Hunspell.

### 2.8.3. Snowball stemming dictionary

Snowball template dictionary is based on the project of Martin Porter, an inventor of popular Porter’s stemming algorithm for English language, and now supported many languages (see Snowball site for more information). FTS contains a large number of stemmers for many languages. The only option, which accepts snowball stemmer is a location of a file with stop words. It can be defined using `ALTER FULLTEXT DICTIONARY` command.

```verbatim
ALTER FULLTEXT DICTIONARY en_stem
OPTION '/usr/local/share/dicts/ispell/english-utf8.stop';
```

Relative paths in `OPTION` resolved respective to `$PGROOT/share/dicts/data`.

```verbatim
ALTER FULLTEXT DICTIONARY en_stem OPTION 'english.stop';
```

Snowball dictionary recognizes everything, so the best practice of usage is to place it at the end of the dictionary stack. It it uselessness to have it before any dictionary, because a lexeme will not pass through a stemmer.

### 2.8.4. Synonym dictionary

This dictionary template is used to create dictionaries which replaces one word by synonym word. Phrases are not supported, use thesaurus dictionary (Section 2.8.5) if you need them. Synonym dictionary can be used to overcome linguistic problems, for example, to avoid reducing of word ‘Paris’ by a english stemmer dictionary to ‘pari’. In that case, it’s enough to have `Paris paris` line in synonym dictionary and put it before `en_stemm` dictionary.

```verbatim
=## select * from ts_debug('english','Paris');
Alias | Description | Token | Dicts list | Lexized token
--- | --- | --- | --- | ---
Paris | Paris | Paris | | Paris
```

2.8.5. Thesaurus dictionary

Thesaurus - is a collection of words with included information about the relationships of words and phrases, i.e., broader terms (BT), narrower terms (NT), preferred terms, non-preferred, related terms, etc.

Basically, thesaurus dictionary replaces all non-preferred terms by one preferred term and, optionally, preserves them for indexing. Thesaurus used when indexing, so any changes in thesaurus require reindexing. Current realization of thesaurus dictionary (TZ) is an extension of synonym dictionary with phrase support. Thesaurus is a plain file of the following format:

    # this is a comment
    sample word(s) : indexed word(s)

where colon (:) symbol is a delimiter.

TZ uses subdictionary (should be defined FTS configuration) to normalize thesaurus text. It's possible to define only one dictionary. Notice, that subdictionary produces an error, if it couldn’t recognize word. In that case, you should remove definition line with this word or teach subdictionary to know it. Use asterisk (*) at the beginning of indexed word to skip subdictionary. It’s still required, that sample words should be known.

Thesaurus dictionary looks for the most longest match.

Stop-words recognized by subdictionary replaced by 'stop-word placeholder', i.e., important only their position. To break possible ties thesaurus applies the last definition. To illustrate this, consider thesaurus (with simple subdictionary) rules with pattern 'swsw, where 's' designates any stop-word and 'w' - any known word:

    a one the two : swsw
    the one a two : swsw2

Words 'a' and 'the' are stop-words defined in the configuration of a subdictionary. Thesaurus considers texts 'the one the two' and 'that one then two' as equal and will use definition 'swsw2'.

As a normal dictionary, it should be assigned to the specific lexeme types. Since TZ has a capability to recognize phrases it must remember its state and interact with parser. TZ use these assignments to check if it should handle next word or stop accumulation. Compiler of TZ should take care about proper
configuration to avoid confusion. For example, if TZ is assigned to handle only lword lexeme, then TZ definition like ’one 1:11’ will not work, since lexeme type digit doesn’t assigned to the TZ.

### 2.8.5.1. Thesaurus configuration

To define new thesaurus dictionary one can use thesaurus template, for example:

```sql
CREATE FULLTEXT DICTIONARY tz_simple
OPTION 'DictFile="dicts_data/thesaurus.txt.sample", Dictionary="en_stem"'
LIKE thesaurus_template;
```

Here:

- `tz_simple` - is the thesaurus dictionary name
- `DictFile="/path/to/tz_simple.txt"` - is the location of thesaurus file
- `Dictionary="en_stem"` defines dictionary (snowball english stemmer) to use for thesaurus normalization. Notice, that en_stem dictionary has its own configuration (stop-words, for example).

Now, it’s possible to bind thesaurus dictionary `tz_simple` and selected tokens, for example:

```sql
ALTER FULLTEXT MAPPING ON russian_utf8 FOR lword,lhword,lpart_hword WITH tz_simple;
```

### 2.8.5.2. Thesaurus examples

Let’s consider simple astronomical thesaurus `tz_astro`, which contains some astronomical word-combinations:

- supernovae stars : sn
- crab nebulae : crab

Below, we create dictionary and bind some types of tokens with astronomical thesaurus and english stemmer.

```sql
=# CREATE FULLTEXT DICTIONARY tz_astro OPTION
    'DictFile="dicts_data/tz_astro.txt", Dictionary="en_stem"'
LIKE thesaurus_template;
=# ALTER FULLTEXT MAPPING ON russian_utf8 FOR lword,lhword,lpart_hword WITH tz_astro,en_stem;
```

Now, we could see how it works. Notice, that `lexize` couldn’t use for testing (see description of `lexize`) thesaurus, so we could use `plainto_tsquery` and `to_tsvector` functions, which accept text argument, not a lexeme.

```sql
=# select plainto_tsquery('supernova star');
plainto_tsquery
--------------
29
```
Chapter 2. FTS Operators and Functions

'sn'

```sql
# select to_tsvector('supernova star');
to_tsvector
----------
'sn':1
```

In principle, one can use `to_tsquery` if quote argument.

```sql
# select to_tsquery('"supernova star"');
to_tsquery
----------
'sn'
```

Notice, that `supernova star` matches `supernovae stars` in `tz_astro`, because we specified `en_stem` stemmer in thesaurus definition.

To keep an original phrase in full-text index just add it to the right part of definition:

```sql
supernovae stars : sn supernovae stars
-----------------------------
# select plainto_tsquery('supernova star');
plainto_tsquery
-----------------------------
'sn' & 'supernova' & 'star'
```

2.9. FTS Configuration

A FTS configuration specifies all of the equipment necessary to transform a document into a `tsvector`: the parser that breaks its text into tokens, and the dictionaries, which then transform each token into a lexeme. Every call to `to_tsvector()`, `to_tsquery()` uses a configuration to perform its processing. Default FTS configurations contain in 4 tables in `pg_catalog` schema, namely, `pg_ts_cfg`, `pg_ts_parser`, `pg_ts_dict`, `pg_ts CFGmap`.

To facilitate management of FTS objects a set of SQL commands, described in FTS Reference Part I, is available. This is a recommended way.

Predefined system FTS objects are available in `pg_catalog` schema. If you need a custom configuration you can create a new FTS object and modify it using SQL commands, described in FTS Reference Part I. For example, to customize parser, create full-text configuration and change the value of the `PARSER` parameter.

```sql
# CREATE FULLTEXT CONFIGURATION public.testcfg LIKE russian_utf8 WITH MAP;
# ALTER FULLTEXT CONFIGURATION public.testcfg SET PARSER htmlparser;
```

New FTS objects created in the current schema on default, usually, in `public` schema, but schema-qualified name could be used to create object in the specified schema. It owned by the current user and can be changed using `ALTER FULLTEXT ... OWNER` SQL command. Visibility of FTS objects conforms to the standard PostgreSQL rule and defined by `search_path` variable, see example in `ALTER FULLTEXT`
Chapter 2. FTS Operators and Functions

... OWNER. By default, the first visible schema is the pg_catalog, so that system FTS objects always mask users. To change that, explicitly specify pg_catalog in the search_path variable.

GUC variable tsearch_conf_name (optionally schema-qualified) defines the name of the current active configuration. It can be defined in postgresql.conf or using SQL command.

Notice, that pg_catalog schema, if not explicitly specified in the search_path, implicitly placed as the first schema to browse.

```sql
=# alter fulltext configuration public.russian_utf8 SET AS DEFAULT;
ALTER FULLTEXT CONFIGURATION

=# \dF *.russ*utf8
List of fulltext configurations
<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Locale</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>russian_utf8</td>
<td>ru_RU.UTF-8</td>
<td>Y</td>
<td>default configuration for Russian/UTF-8</td>
</tr>
<tr>
<td>public</td>
<td>russian_utf8</td>
<td>ru_RU.UTF-8</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>
(2 rows)

=# show tsearch_conf_name;
   tsearch_conf_name
   -------------------------
   pg_catalog.russian_utf8
(1 row)

=# set search_path=public, pg_catalog;
SET
=# show tsearch_conf_name;
   tsearch_conf_name
   -------------------------
   public.russian_utf8
```

There are several psql commands, which display various information about FTS objects (Section 2.11).

### 2.10. Debugging

Function ts_debug allows easy testing your full-text configuration.

```sql
ts_debug( [cfgname | oid ],document TEXT) RETURNS SETOF tsdebug
```

It displays information about every token from document as they produced by a parser and processed by dictionaries as it was defined in configuration, specified by cfgname or oid.

tsdebug type defined as

```sql
CREATE TYPE tsdebug AS (  
  "Alias" text,  
  "Description" text,  
  "Token" text,  
```

31
Chapter 2. FTS Operators and Functions

"Dicts list"  text[],
"Lexized token"  text

For demonstration of how function ts_debug works we first create public.english configuration and ispell dictionary for English language. You may skip test step and play with standard english configuration.

CREATE FULLTEXT CONFIGURATION public.english LIKE pg_catalog.english WITH MAP AS DEFAULT;
CREATE FULLTEXT DICTIONARY en_ispell OPTION 'DictFile="/usr/local/share/dicts/ispell/english-utf8.dict",
AffFile="/usr/local/share/dicts/ispell/english-utf8.aff",
StopFile="/usr/local/share/dicts/english.stop"'
LIKE ispell_template;
ALTER FULLTEXT MAPPING ON public.english FOR lword WITH en_ispell,en_stem;

=# select * from ts_debug('public.english','The Brightest supernovaes');
<table>
<thead>
<tr>
<th>Alias</th>
<th>Description</th>
<th>Token</th>
<th>Dicts list</th>
</tr>
</thead>
<tbody>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>The</td>
<td>{public.en_ispell,pg_catalog.en_stem} public.en_ispell: {}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>Brightest</td>
<td>{public.en_ispell,pg_catalog.en_stem} public.en_ispell: {bright}</td>
</tr>
<tr>
<td>blank</td>
<td>Space symbols</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Latin word</td>
<td>supernovaes</td>
<td>{public.en_ispell,pg_catalog.en_stem} pg_catalog.en_stem: {supernova}</td>
</tr>
</tbody>
</table>

(5 rows)

In this example, the word 'Brightest' was recognized by a parser as a Latin word (alias lword) and came through a dictionaries public.en_ispell,pg_catalog.en_stem. It was recognized by public.en_ispell, which reduced it to the noun bright. Word supernovaes is unknown for public.en_ispell dictionary, so it was passed to the next dictionary, and, fortunately, was recognized (in fact, public.en_stem is a stemming dictionary and recognizes everything, that is why it placed at the end the dictionary stack).

The word The was recognized by public.en_ispell dictionary as a stop-word (Section 1.3.6) and will not indexed.

You can always explicitly specify what columns you want to see

=# select "Alias", "Token", "Lexized token"
from ts_debug('public.english','The Brightest supernovaes');
<table>
<thead>
<tr>
<th>Alias</th>
<th>Token</th>
<th>Lexized token</th>
</tr>
</thead>
<tbody>
<tr>
<td>lword</td>
<td>The</td>
<td>public.en_ispell: {}</td>
</tr>
<tr>
<td>blank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>Brightest</td>
<td>public.en_ispell: {bright}</td>
</tr>
<tr>
<td>blank</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lword</td>
<td>supernovaes</td>
<td>pg_catalog.en_stem: {supernova}</td>
</tr>
</tbody>
</table>

(5 rows)
2.11. Psql support

Information about FTS objects can be obtained in psql using a set of commands

```
\dF{,d,p}{+} [PATTERN]
```

Optional + used to produce more details.

Optional parameter PATTERN is a name (can be schema-qualified) of the FTS object. If PATTERN is not specified, then information about default object (configuration, parser, dictionaries) will be displayed. Visibility of FTS objects conforms PostgreSQL rule. PATTERN can be a regular expression and should apply separately to schema name and object name. Following examples illustrate this.

```
=# \dF *fts*
List of fulltext configurations
Schema | Name | Locale | Description
--------+-------+--------+------------
public | fts_cfg | ru_RU.UTF-8 |

=# \dF *.fts*
List of fulltext configurations
Schema | Name | Locale | Description
--------+-------+--------+------------
fts | fts_cfg | ru_RU.UTF-8 |
public | fts_cfg | ru_RU.UTF-8 |
```

```
\dF[+] [PATTERN]
```

List full-text configurations (add "+" for more detail)

By default (without PATTERN), information about all visible full-text configurations will be displayed.

```
=# \dF russian_utf8
List of fulltext configurations
Schema | Name | Locale | Default | Description
--------+-------+--------+---------+-----------------------------------------
pg_catalog | russian_utf8 | ru_RU.UTF-8 | Y | default configuration for Russian/UTF-8

=# \dF+ russian_utf8
Configuration "pg_catalog.russian_utf8"
Parser name: "pg_catalog.default"
Locale: ‘ru_RU.UTF-8’ (default)
Token | Dictionaries
                    --------------------------
email | pg_catalog.simple
file | pg_catalog.simple
float | pg_catalog.simple
host | pg_catalog.simple
hword | pg_catalog.ru_stem_utf8
int | pg_catalog.simple
lword | public.tz_simple
lpart_hword | public.tz_simple
Chapter 2. FTS Operators and Functions

List full-text dictionaries (add "+" for more detail).

By default (without PATTERN), information about all visible dictionaries will be displayed.

```
postgres=# \dFd
List of fulltext dictionaries

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pg_catalog</td>
<td>danish_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>danish_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>dutch_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>dutch_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>en_stem</td>
<td>English stemmer. Snowball.</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>finnish_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>finnish_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>french_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>french_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>german_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>german_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>hungarian_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>hungarian_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>ispell_template</td>
<td>Ispell dictionary template</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>italian_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>italian_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>norwegian_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>norwegian_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>portuguese_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>portuguese_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>ru_stem_koi8</td>
<td>KOI-8 russian stemmer. Snowball.</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>ru_stem_utf8</td>
<td>UTF-8 russian stemmer. Snowball.</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>ru_stem_win1251</td>
<td>WIN1251 russian stemmer. Snowball.</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>simple</td>
<td>simple dictionary: just lower case and check for stopword</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>spanish_iso_8859_1</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>spanish_utf_8</td>
<td>Snowball stemmer</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>synonym</td>
<td>synonym dictionary: replace word by its synonym</td>
</tr>
<tr>
<td>pg_catalog</td>
<td>thesaurus_template</td>
<td>Thesaurus template. Phrase by phrase substitution</td>
</tr>
</tbody>
</table>

```

34
List full-text parsers (add "+" for more detail)

By default (without PATTERN), information about all visible full-text parsers will be displayed.

```
postgres=# \dFp
List of fulltext parsers
Schema | Name               | Description
--------+--------------------+------------------
pg_catalog | default | default word parser
(1 row)
```

```
postgres=# \dFp+
Fulltext parser "pg_catalog.default"
Method          | Function                   | Description
-----------------+---------------------------+-------------
Start parse      | pg_catalog.prsd_start     |             |
Get next token   | pg_catalog.prsd_nexttoken |             |
End parse        | pg_catalog.prsd_end       |             |
Get headline     | pg_catalog.prsd_headline  |             |
Get lexeme’s type| pg_catalog.prsd_lextype   |             |

Token’s types for parser "pg_catalog.default"
Token name | Description
-----------+-----------------------------------
blank      | Space symbols
email      | Email                             
entity     | HTML Entity                       
file       | File or path name                 
float      | Decimal notation                  
host       | Host                              
hword      | Hyphenated word                   
int        | Signed integer                    
lhword     | Latin hyphenated word             
lpart_hword| Latin part of hyphenated word     
lword      | Latin word                        
nlhword    | Non-latin hyphenated word         
nlpart_hword| Non-latin part of hyphenated word |
nlword     | Non-latin word                    
part_hword | Part of hyphenated word           
protocol   | Protocol head                     
sfloat     | Scientific notation               
tag       | HTML Tag                          
uint       | Unsigned integer                  
uri        | URI                               
url        | URL                               
version    | VERSION                           
word       | Word                              |
```

(23 rows)
I. FTS Reference
I. SQL Commands

This part contains reference information for the SQL commands related to the full-text search (FTS), supported by PostgreSQL.
CREATE FULLTEXT CONFIGURATION

Name
CREATE FULLTEXT CONFIGURATION — create full-text configuration

Synopsis

CREATE FULLTEXT CONFIGURATION cfgname
PARSER prsname [ LOCALE localename]
[AS DEFAULT];

CREATE FULLTEXT CONFIGURATION cfgname
[{ PARSER prsname | LOCALE localename } [ ...]]
LIKE template_cfg [WITH MAP]
[AS DEFAULT];

Description

CREATE FULLTEXT CONFIGURATION will create a new FTS configuration. The new configuration will be owned by the user issuing the command.

If a schema name is given (for example, CREATE FULLTEXT CONFIGURATION myschema.cfgname ... ) then the configuration is created in the specified schema. Otherwise it is created in the current schema.

Parameters

cfgname
The name (optionally schema-qualified) of the full-text configuration to be created.

PARSER
prsname is the name (optionally schema-qualified) of the parser.

LOCALE
localename is the name of the locale. It should match server’s locale (lc_cctype) to identify full-text configuration used by default.

LIKE
Existing full-text configuration templatecfg (optionally schema-qualified) will be used to create new configuration. Values of PARSER, LOCALE parameters, if defined, will substitute default values of the template configuration.
CREATE FULLTEXT CONFIGURATION

WITH MAP

If specified, then full-text mapping of template configuration is copied to the new configuration.

AS DEFAULT

Set default flag for the configuration, which used to identify if this configuration is selectable on
default (see LOCALE description above). It is possible to have maximum one configuration with the
same locale and in the same schema with this flag enabled.

Examples

Create new configuration test with default parser and ru_RU.UTF-8 locale.

=# CREATE FULLTEXT CONFIGURATION test PARSER default LOCALE 'ru_RU.UTF-8';
=# \dF+ test
Configuration "public.test"
Parser name: "pg_catalog.default"
Locale: 'ru_RU.UTF-8'
  Token | Dictionaries
--------+-----------------
Now we create configuration using english configuration (parser and full-text mapping) but with
ru_RU.UTF-8 locale.

=# CREATE FULLTEXT CONFIGURATION test LOCALE 'ru_RU.UTF-8' LIKE english WITH MAP;
CREATE FULLTEXT CONFIGURATION
=# \dF+ test
Configuration "public.test"
Parser name: "pg_catalog.default"
Locale: 'ru_RU.UTF-8'
  Token | Dictionaries
--------+-----------------
 email | pg_catalog.simple
  file | pg_catalog.simple
 float | pg_catalog.simple
 host | pg_catalog.simple
 hword | pg_catalog.simple
  int | pg_catalog.simple
 lhword | pg_catalog.en_stem
 lpart_hword | pg_catalog.en_stem
 lword | pg_catalog.en_stem
 nhword | pg_catalog.en_stem
 nlpart_hword | pg_catalog.simple
 nlnword | pg_catalog.simple
 part_hword | pg_catalog.simple
 sfloat | pg_catalog.simple
 uint | pg_catalog.simple
 uri | pg_catalog.simple
 url | pg_catalog.simple
 version | pg_catalog.simple
CREATE FULLTEXT CONFIGURATION

| word | pg_catalog.simple |

In the example below we first create test configuration (in public schema by default) with default flag enabled using system configuration pg_catalog.russian=utf8 as template. Then, we create another configuration with the same parameters as earlier and show that default flag was removed from test configuration.

```sql
=# CREATE FULLTEXT CONFIGURATION test LIKE pg_catalog.russian_utf8 AS DEFAULT;
CREATE FULLTEXT CONFIGURATION
=# \dF public.test
--- List of fulltext configurations
| Schema | Name | Locale | Default | Description |
|--------+------|--------|---------|------------|
| public | test | ru_RU.UTF-8 | Y | |

=# CREATE FULLTEXT CONFIGURATION test2 LIKE pg_catalog.russian_utf8 AS DEFAULT;
NOTICE: drop default flag for fulltext configuration "public.test"
=# \dF public.test*
--- List of fulltext configurations
| Schema | Name | Locale | Default | Description |
|--------+------|--------|---------|------------|
| public | test | ru_RU.UTF-8 | Y | |

=# ALTER FULLTEXT CONFIGURATION test2 DROP DEFAULT;
ALTER FULLTEXT CONFIGURATION
=# \dF public.test*
--- List of fulltext configurations
| Schema | Name | Locale | Default | Description |
|--------+------|--------|---------|------------|
| public | test | ru_RU.UTF-8 | Y | |

See Also

DROP FULLTEXT CONFIGURATION, ALTER FULLTEXT CONFIGURATION
DROP FULLTEXT CONFIGURATION

Name

DROP FULLTEXT CONFIGURATION — remove a full-text configuration

Synopsis

DROP FULLTEXT CONFIGURATION [IF EXISTS] cfgname [ CASCADE | RESTRICT ];

Description

DROP FULLTEXT CONFIGURATION removes full-text configuration from the database. Only its owner may destroy a configuration.

To drop a configuration and all FTS objects, which depends on it, CASCADE must be specified.

Parameters

IF EXISTS

Do not throw an error if the configuration does not exist. A notice is issued in this case.

cfgname

The name (optionally schema-qualified) of the configuration to drop.

CASCADE

Automatically drop FTS objects that depend on the configuration

RESTRICT

Refuse to drop the configuration if any objects depend on it. This is the default.

See Also

CREATE FULLTEXT CONFIGURATION
ALTER FULLTEXT CONFIGURATION

Name
ALTER FULLTEXT CONFIGURATION — change a full-text configuration

Synopsis

ALTER FULLTEXT CONFIGURATION cfgname RENAME TO newcfgname;

ALTER FULLTEXT CONFIGURATION cfgname SET { LOCALE localename | PARSER prsname } [, ...];

ALTER FULLTEXT CONFIGURATION cfgname { SET AS | DROP } DEFAULT;

Description
ALTER FULLTEXT CONFIGURATION changes an existing full-text configuration.

Parameters

cfgname

The name (optionally schema-qualified) of the configuration to rename.

RENAME TO

newcfgname is the new name of the configuration. Notice, that schema cannot be changed.

SET

Values of LOCALE, PARSER parameters, if defined, will substitute current values.

SET AS DEFAULT

Set default flag for the configuration.

DROP DEFAULT

Remove default flag for the configuration.

Examples
There are could be maximum one configuration with DEFAULT flag defined in the same schema and with the same locale.

=## \DF public.test*

List of fulltext configurations

| Schema | Name | Locale | Default | Description |
|--------+------|--------|---------|------------|
|        |      |        |         |            |
ALTER FULLTEXT CONFIGURATION

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Locale</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>test</td>
<td>ru_RU.UTF-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>test2</td>
<td>ru_RU.UTF-8</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

=# ALTER FULLTEXT CONFIGURATION test2 DROP DEFAULT;

ALTER FULLTEXT CONFIGURATION

=# \dF public.test*

List of fulltext configurations

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Locale</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>test</td>
<td>ru_RU.UTF-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>test2</td>
<td>ru_RU.UTF-8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

=# ALTER FULLTEXT CONFIGURATION test2 SET AS DEFAULT;

ALTER FULLTEXT CONFIGURATION

Time: 1.629 ms

postgres=# \dF public.test*

List of fulltext configurations

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Locale</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>test</td>
<td>ru_RU.UTF-8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>public</td>
<td>test2</td>
<td>ru_RU.UTF-8</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

See Also

CREATE FULLTEXT CONFIGURATION
CREATE FULLTEXT DICTIONARY

Name

CREATE FULLTEXT DICTIONARY — create a dictionary for full-text search

Synopsis

CREATE FULLTEXT DICTIONARY dictname
  LEXIZE lexize_function
  [INIT init_function ]
  [OPTION opt_text ]
;

CREATE FULLTEXT DICTIONARY dictname
  [ { INIT init_function
     | LEXIZE lexize_function
     | OPTION opt_text }]
  [ ... ]] LIKE template_dictname;

Description

CREATE FULLTEXT DICTIONARY will create a new dictionary used to transform input word to a lexeme.

If a schema name is given (for example, CREATE FULLTEXT DICTIONARY myschema.dictname ...) then the dictionary is created in the specified schema. Otherwise it is created in the current schema.

Parameters

dictname

The name (optionally schema-qualified) of the new dictionary.

LEXIZE

lexize_function is the name of the function, which does transformation of input word.

INIT

init_function is the name of the function, which initialize dictionary.

OPTION

opt_text is the meaning of the opt_text varies among dictionaries. Usually, it stores various options required for the dictionary, for example, location of stop words file. Relative paths are defined with regard to PGROOT/share/dicts_data directory.
CREATE FULLTEXT DICTIONARY

LIKE

template_dictname is the name (optionally schema-qualified) of existing full-text dictionary used as a template. Values of INIT, LEXIZE, OPTION parameters, if defined, will substitute default values of the template dictionary.

Examples

Create dictionary public.my_simple in public schema, which uses functions defined for system pg_catalog.simple dictionary. We specify location of stop-word file.

```sql
=# CREATE FULLTEXT DICTIONARY public.my_simple LEXIZE dsimple_lexize INIT dsimple_init OPTION '/usr/local/share/dicts/english.stop';
=# select lexize('public.my_simple','YeS');
lexize
-------
{yes}
=# select lexize('public.my_simple','The');
lexize
-------
{}
```

This could be done easier using template.

```sql
=# CREATE FULLTEXT DICTIONARY public.my_simple OPTION '/usr/local/share/dicts/english.stop';
=# select lexize('public.my_simple','YeS');
lexize
-------
{yes}
=# select lexize('public.my_simple','The');
lexize
-------
{}
```

See Also

DROP FULLTEXT DICTIONARY, ALTER FULLTEXT DICTIONARY
DROP FULLTEXT DICTIONARY

Name

DROP FULLTEXT DICTIONARY — remove a full-text dictionary

Synopsis

DROP FULLTEXT DICTIONARY [IF EXISTS]dictname [ CASCADE | RESTRICT ];

Description

DROP FULLTEXT DICTIONARY removes full-text dictionary from the database. Only its owner may destroy a configuration.

To drop a dictionary and all FTS objects, which depends on it, CASCADE must be specified.

Parameters

IF EXISTS

Do not throw an error if the dictionary does not exist. A notice is issued in this case.

dictname

The name (optionally schema-qualified) of the dictionary to drop.

CASCADE

Automatically drop FTS objects that depend on the dictionary.

RESTRICT

Refuse to drop the dictionary if any objects depend on it. This is the default.

See Also

CREATE FULLTEXT DICTIONARY
ALTER FULLTEXT DICTIONARY

Name
ALTER FULLTEXT DICTIONARY — change a full-text dictionary

Synopsis

ALTER FULLTEXT DICTIONARY dictname RENAME TO newdictname;
ALTER FULLTEXT DICTIONARY dictname SET OPTION opt_text;

Description
ALTER FULLTEXT DICTIONARY change an existing full-text dictionary.

Parameters

dictname
The name (optionally schema-qualified) of the dictionary to rename.

newdictname
The new name of the dictionary. Notice, that schema cannot be changed.

SET OPTION
Define a new value opt_text of the existing full-text dictionary.

See Also
CREATE FULLTEXT DICTIONARY
CREATE FULLTEXT MAPPING

Name

CREATE FULLTEXT MAPPING — binds tokens and dictionaries

Synopsis

CREATE FULLTEXT MAPPING ON \texttt{cfgname} FOR \texttt{tokentypename}, ... WITH \texttt{dictname1}, ...;

Description

CREATE FULLTEXT MAPPING bind token of type \texttt{lexemetype} and full-text dictionaries in given configuration \texttt{cfgname}. The order of dictionaries is important, since lexeme processed in that order.

Parameters

cfgname

The name (optionally schema-qualified) of the full-text configuration.

FOR

\texttt{tokentypename} is the type of token full-text mapping created for.

WITH

\texttt{dictname1} is the name of full-text dictionary, which binds to the \texttt{tokentypename}.

Examples

In example below, we first create \texttt{testcfg} full-text configuration and then create mapping for token of types \texttt{lword, lhword, lpart_hword}.

```sql
=# CREATE FULLTEXT CONFIGURATION testcfg LOCALE 'testlocale' LIKE russian_utf8;
CREATE FULLTEXT CONFIGURATION
=# CREATE FULLTEXT MAPPING ON testcfg FOR lword, lhword, lpart_hword WITH simple, en_stem;
CREATE FULLTEXT MAPPING
=# \dF+ testcfg
Configuration 'testcfg'
Parser name: 'default'
Locale: 'testlocale'
    Token  | Dictionaries
          ---------------
      lhword  | simple, en_stem
```
CREATE FULLTEXT MAPPING

lpart_hword | simple, en_stem
lword | simple, en_stem

See Also

ALTER FULLTEXT MAPPING
ALTER FULLTEXT MAPPING

Name
ALTER FULLTEXT MAPPING — change token binding with FTS dictionaries

Synopsis
ALTER FULLTEXT MAPPING ON cfuname FOR tokentypename[, ...] WITH dictname1[, ...];
ALTER FULLTEXT MAPPING ON cfuname [FOR tokentypename[, ...]] REPLACE olddictname TO newdictname;

Description
ALTER FULLTEXT MAPPING change binding of token of tokentypename or create one if binding doesn’t exists.

Parameters

cfuname
The name (optionally schema-qualified) of the full-text configuration.

FOR
tokentypename is the type of token full-text mapping created for.

WITH
dictname1 is the name of full-text dictionary, which binds to the tokentypename.

REPLACE
olddictname is the name of full-text dictionary to be replaced by a newdictname.

TO
newdictname is the name of full-text dictionary, which replaces olddictname.

Examples

=# ALTER FULLTEXT MAPPING ON testcfg FOR lword WITH simple;
ALTER FULLTEXT MAPPING
=# ALTER FULLTEXT MAPPING ON testcfg FOR lhword WITH simple, en_stem;
ALTER FULLTEXT MAPPING
=# dF+ testcfg
  Configuration ‘testcfg’
Parser name: ’default’
Locale: ’testlocale’
ALTER FULLTEXT MAPPING

<table>
<thead>
<tr>
<th>Token</th>
<th>Dictionaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>lhword</td>
<td>simple, en_stem</td>
</tr>
<tr>
<td>lword</td>
<td>simple</td>
</tr>
</tbody>
</table>

See Also

CREATE FULLTEXT MAPPING
DROP FULLTEXT MAPPING

Name
DROP FULLTEXT MAPPING — remove a binding between token and dictionaries

Synopsis
DROP FULLTEXT MAPPING [IF EXISTS] ON cfgname FOR tokentypename;

Description
DROP FULLTEXT MAPPING remove a full-text mapping in a given configuration for a token of a specific type.

Parameters
IF EXISTS
Do not throw an error if the specified full-text mapping does not exist. A notice is issued in this case.
cfgname
The name (optionally schema-qualified) of the configuration.
tokentypename
A token type for which full-text mapping dropped.

See Also
CREATE FULLTEXT MAPPING
CREATE FULLTEXT PARSER

Name

CREATE FULLTEXT PARSER — create a parser for full-text search

Synopsis

```
CREATE FULLTEXT PARSER prsname
    START=    start_function
    GETTOKEN gettoken_function
    END      end_function
    LEXTYPES lextypes_function
        [ HEADLINE    headline_function ]
;
```

Description

CREATE FULLTEXT PARSER will create a new parser used to break document onto lexemes. If a schema name is given (for example, CREATE FULLTEXT PARSER myschema.prsname ...) then the parser is created in the specified schema. Otherwise it is created in the current schema. More information about developing custom parser is available from this Appendix B.

Parameters

prsname
The name (optionally schema-qualified) of the new parser.

START
start_function is the name of the function, that initialize a parser.

GETTOKEN
gettoken_function, is the name of the function, that returns a token.

END
end_function, is the name of the function, that called after parsing is finished.

LEXTYPES
lextypes_function, is the name of the function, that returns an array containing the id, alias and the description of the tokens of a parser.
HEADLINE

headline_function, is the name of the function, that returns a representative piece of document.

See Also

DROP FULLTEXT PARSER, ALTER FULLTEXT PARSER
DROP FULLTEXT PARSER

Name

DROP FULLTEXT PARSER — remove a full-text parser

Synopsis

```
DROP FULLTEXT PARSER [IF EXISTS] prsname [ CASCADE | RESTRICT ];
```

Description

DROP FULLTEXT PARSER removes full-text parser from the database. Only its owner may destroy a parser.

To drop a parser and all FTS objects, which depends on it, CASCADE must be specified.

Parameters

IF EXISTS

Do not throw an error if the parser does not exist. A notice is issued in this case.

prsname

The name (optionally schema-qualified) of the parser to drop.

CASCADE

Automatically drop FTS objects that depend on the parser.

RESTRICT

Refuse to drop the parser if any objects depend on it. This is the default.

See Also

CREATE FULLTEXT PARSER
ALTER FULLTEXT PARSER

Name
ALTER FULLTEXT PARSER — change a full-text parser

Synopsis
ALTER FULLTEXT PARSER prsname RENAME TO newprsname;

Description
ALTER FULLTEXT PARSER changes an existing full-text parser.

Parameters
prsname
  The name (optionally schema-qualified) of the parser to rename.
newprsname
  The new name of the parser. Notice, that schema cannot be changed.

See Also
CREATE FULLTEXT PARSER
ALTER FULLTEXT ... OWNER

Name

ALTER FULLTEXT ... OWNER — change the owner of a full-text object

Synopsis

ALTER FULLTEXT { PARSER|DICTIONARY|CONFIGURATION } name OWNER TO newowner;

Description

ALTER FULLTEXT ... OWNER changes the owner of an existing full-text object.

Parameters

name

The name (optionally schema-qualified) of the full-text object.

newowner

The new owner of the full-text object.

Examples

In this example we want to create new dictionary in schema test using predefined dictionary from system catalog. Then we change owner of the new dictionary. To demonstrate visibility rule we use the name of the dictionary without schema but setting the proper search_path. The name of the new dictionary is the same by intent.

=# CREATE SCHEMA test;
=# CREATE FULLTEXT DICTIONARY test.synonym LIKE pg_catalog."synonym";
=# SHOW search_path;
   search_path
----------
"$user",public
=# SET search_path TO test,public;
=# ALTER FULLTEXT DICTIONARY synonym OWNER TO megera;
COMMENT ON FULLTEXT

Name

COMMENT ON FULLTEXT — define or change the comment of a full-text object

Synopsis

COMMENT ON FULLTEXT { CONFIGURATION | DICTIONARY | PARSER } objname IS text;

Description

COMMENT ON FULLTEXT stores a comment about a full-text object (configuration, dictionary, parser).

To modify a comment, issue a new COMMENT ON FULLTEXT command for the same full-text object. Only one comment string is stored for each object. To remove a comment, write NULL in place of the text string. Comments are automatically dropped when the object is dropped.

Parameters

objname

The name (optionally schema-qualified) of the full-text object.

text

The new comment, written as a string literal; or NULL to drop the comment.

Examples

=# COMMENT ON FULLTEXT DICTIONARY intdict IS 'Dictionary for integers';
=# \dFd+ intdict

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Init method</th>
<th>Lexize method</th>
<th>Init options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>intdict</td>
<td>dinit_intdict</td>
<td>dlexize_intdict</td>
<td>MAXLEN=6,REJECTLONG=false</td>
<td>Dictionary for integers</td>
</tr>
</tbody>
</table>
II. Appendixes
Appendix A. FTS Complete Tutorial

This tutorial is about how to setup typical FTS application using PostgreSQL.

We create our configuration `pg`, which will be default for locale `ru_RU.UTF-8`. To be safe, we do this in transaction.

```
BEGIN;
CREATE FULLTEXT CONFIGURATION public.pg LOCALE 'ru_RU.UTF-8' LIKE english WITH MAP AS

CREATE FULLTEXT DICTIONARY pg_dict OPTION 'pg_dict.txt' LIKE synonym;
```

We’ll use postgresql specific dictionary using `synonym` template dictionary and store it under `PG_ROOT/share/dicts_data` directory. The dictionary looks like:

```
postgres    pg
pgsql       pg
postgresql  pg
```

```
CREATE FULLTEXT DICTIONARY pg_dict OPTION 'pg_dict.txt' LIKE synonym;
```

Register ispell dictionary `en_ispell` using `ispell_template` template.

```
CREATE FULLTEXT DICTIONARY en_ispell
OPTION 'DictFile="english-utf8.dict",
       AffFile="english-utf8.aff",
       StopFile="english-utf8.stop")'
LIKE ispell_template;
```

Use the same stop-word list for snowball stemmer `en_stem`, which is available on default.

```
ALTER FULLTEXT DICTIONARY en_stem SET OPTION 'english-utf8.stop';
```

Modify mappings for Latin words for configuration `pg`

```
ALTER FULLTEXT MAPPING ON pg FOR lword,lhword,lpart_hword
WITH pg_dict,en_ispell,en_stem;
```

We won’t index/search some tokens

```
DROP FULLTEXT MAPPING ON pg FOR email, url, sfloat, uri, float;
```

Now, we could test our configuration.

```
SELECT * FROM ts_debug('public.pg', 'PostgreSQL, the highly scalable, SQL compliant, open source object-relational
```
Appendix A. FTS Complete Tutorial

database management system, is now undergoing beta testing of the next version of our software: PostgreSQL 8.2.

We have a table pgweb, which contains 11239 documents from PostgreSQL web site. Only relevant columns are shown.

```bash
=\d pgweb
Table "public.pgweb"
Column | Type | Modifiers
-----------+-------------------+-----------
tid | integer | not null
path | character varying | not null
body | character varying |
title | character varying |
dl | integer |
```

First we should take care about default FTS configuration - we want our public.pg to be default. To do so, we need to redefine search_path, since we already have predefined default full-text configuration (for ru_RU.UTF-8 locale) in pg_catalog.

```sql
=\dF
pg_catalog | russian_utf8 | ru_RU.UTF-8 | Y
public | pg | ru_RU.UTF-8 | Y

=# show tsearch_conf_name;
tsearch_conf_name
-------------------------
pg_catalog.russian_utf8

=# SET search_path=public, pg_catalog;

=# show tsearch_conf_name;
tsearch_conf_name
-------------------------
public.pg
```

The very simple full-text search without ranking is already available here. Select top 10 fresh documents (dlm is last-modified date in seconds since 1970), which contains query create table.

```sql
=# select title from pgweb where textcat(title,body) @@ plainto_tsquery('create table') order by dlm desc limit 10;
```

We can create index to speedup search.
For clarity, we omitted here \texttt{coalesce} function to prevent unwanted effect of \texttt{NULL} concatenation.

To implement FTS with ranking support we need \texttt{tsvector} column to store preprocessed document, which is a concatenation of \texttt{title} and \texttt{body}. We assign different labels to them to preserve information about origin of every word.

```sql
=# alter table pgweb add column fts_index tsvector;
=# update pgweb set fts_index =
    setweight( to_tsvector( coalesce (title,"")), 'A' ) || ' ' ||
    setweight( to_tsvector(coalesce (body,"")), 'D' );
```

Then we create GIN index to speedup search.

```sql
=# create index fts_idx on pgweb using gin(fts_index);
```

After vacuuming, we are ready to perform full-text search.

```sql
=# select rank_cd(fts_index, q)as rank, title from pgweb,
    plainto_tsquery('create table') q
    where q @@ fts_index order by rank desc limit 10;
```
Appendix B. FTS Parser Example

SQL command CREATE FULLTEXT PARSER creates a parser for full-text search. In our example we will implement a simple parser, which recognize space delimited words and has only two types (3, word, Word; 12, blank, Space symbols). Identifiers were chosen to keep compatibility with default headline(), since we won’t implement our version.

To implement parser one need to realize minimum four functions (CREATE FULLTEXT PARSER).

START = start_function
Initialize the parser. Arguments are a pointer to the parsed text and its length.
Returns a pointer to the internal structure of a parser. Note, it should be malloced or palloced in TopMemoryContext. We name it ParserState.

GETTOKEN = gettoken_function
Returns the next token. Arguments are (ParserState *),(char **), (int *).
This procedure will be called so long as the procedure return token type = 0.

END = end_function,
Void function, will be called after parsing is finished. We have to free our allocated resources in this procedure (ParserState). Argument is (ParserState *).

LEXTYPES = lextypes_function
Returns an array containing the id, alias and the description of the tokens of our parser. See LexDescr in src/include/utils/ts_public.h

Source code of our test parser, organized as a contrib module, available in the next section.

Testing:

```sql
=# SELECT * FROM parse('testparser','That”s my first own parser');
tokid | token
-------+--------
  3 | That’s
 12 |
  3 | my
 12 |
  3 | first
 12 |
  3 | own
 12 |
  3 | parser

=# SELECT to_tsvector('testcfg','That”s my first own parser');
to_tsvector
-----------------
‘my’:2 ‘own’:4 ‘first’:3 ‘parser’:5 ‘that”s’:1

=# SELECT headline('testcfg','Supernovae stars are the brightest phenomena in galaxies’
```
B.1. Parser sources

Parser sources was adapted to 8.3 release from original tutorial by Valli parser HOWTO¹.

To compile an example just do

```
make
make install
psql regression < test_parser.sql
```

This is a `test_parser.c`

```c
#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

/* types */

/* self-defined type */
typedef struct {
    char * buffer; /* text to parse */
    int len;     /* length of the text in buffer */
    int pos;     /* position of the parser */
} ParserState;

/* copy-paste from wparser.h of tsearch2 */
typedef struct {
    int lexid;
    char * alias;
    char * descr;
} LexDescr;

/* prototypes */

PG_FUNCTION_INFO_V1(testprs_start);
Datum testprs_start(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(testprs_getlexeme);
Datum testprs_getlexeme(PG_FUNCTION_ARGS);
```

¹ http://www.sai.msu.su/~megera/postgres/gist/tsearch/V2/docs/HOWTO-parser-tsearch2.html
Appendix B. FTS Parser Example

PG_FUNCTION_INFO_V1(testprs_end);
Datum testprs_end(PG_FUNCTION_ARGS);

PG_FUNCTION_INFO_V1(testprs_lextype);
Datum testprs_lextype(PG_FUNCTION_ARGS);

/\n/ * functions 
/ *
Datum testprs_start(PG_FUNCTION_ARGS)
{
ParserState *pst = (ParserState *) palloc(sizeof(ParserState));
pst->buffer = (char *) PG_GETARG_POINTER(0);
pst->len = PG_GETARG_INT32(1);
pst->pos = 0;

PG_RETURN_POINTER(pst);
}

Datum testprs_getlexeme(PG_FUNCTION_ARGS)
{
ParserState *pst = (ParserState *) PG_GETARG_POINTER(0);
char **t = (char **) PG_GETARG_POINTER(1);
int *tlen = (int *) PG_GETARG_POINTER(2);
int type;

*tlen = pst->pos;
*t = pst->buffer + pst->pos;

if ((pst->buffer)[pst->pos] == ' ') {
 /* blank type */
 type = 12;
 /* go to the next non-white-space character */
 while (((pst->buffer)[pst->pos] == ' ') && (pst->pos < pst->len)) {
 (pst->pos)++;
 }
} else {
 /* word type */
 type = 3;
 /* go to the next white-space character */
 while (((pst->buffer)[pst->pos] != ' ') && (pst->pos < pst->len)) {
 (pst->pos)++;
 }
}

*tlen = pst->pos - *tlen;

/* we are finished if (*tlen == 0) */
if (*tlen == 0) type=0;

PG_RETURN_INT32(type);
Datum testprs_end(PG_FUNCTION_ARGS) {
    ParserState *pst = (ParserState *) PG_GETARG_POINTER(0);
    pfree(pst);
    PG_RETURN_VOID();
}

Datum testprs_lextype(PG_FUNCTION_ARGS) {
    /*
     Remarks:
     - we have to return the blanks for headline reason
     - we use the same lexids like Teodor in the default
     word parser; in this way we can reuse the headline
     function of the default word parser.
     */
    LexDescr *descr = (LexDescr *) palloc(sizeof(LexDescr) * (2+1));

    /* there are only two types in this parser */
    descr[0].lexid = 3;
    descr[0].alias = pstrdup("word");
    descr[0].descr = pstrdup("Word");
    descr[1].lexid = 12;
    descr[1].alias = pstrdup("blank");
    descr[1].descr = pstrdup("Space symbols");
    descr[2].lexid = 0;

    PG_RETURN_POINTER(descr);
}

This is a Makefile

override CPPFLAGS := -I. $(CPPFLAGS)

MODULE_big = test_parser
OBJJS = test_parser.o

DATA_built = test_parser.sql
DATA =
DOCS = README.test_parser
REGRESS = test_parser

ifdef USE_PGXS
PGXS := $(shell pg_config --pgxs)
include $(PGXS)
else
subdir = contrib/test_parser
top_builddir = ../..
include $(top_builddir)/src/Makefile.global
include $(top_srcdir)/contrib/contrib-global.mk
endif
This is a test_parser.sql.in

```sql
SET search_path = public;

BEGIN;

CREATE FUNCTION testprs_start(internal, int4)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE 'C' with (isstrict);

CREATE FUNCTION testprs_getlexeme(internal, internal, internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE 'C' with (isstrict);

CREATE FUNCTION testprs_end(internal)
RETURNS void
AS 'MODULE_PATHNAME'
LANGUAGE 'C' with (isstrict);

CREATE FUNCTION testprs_lextype(internal)
RETURNS internal
AS 'MODULE_PATHNAME'
LANGUAGE 'C' with (isstrict);

CREATE FULLTEXT PARSER testparser
START 'testprs_start'
GETTOKEN 'testprs_getlexeme'
END 'testprs_end'
LEXTYPES 'testprs_lextype'
;

CREATE FULLTEXT CONFIGURATION testcfg PARSER 'testparser' LOCALE NULL;
CREATE FULLTEXT MAPPING ON testcfg FOR word WITH simple;

END;
```
Appendix C. FTS Dictionary Example

Motivation for this dictionary is to control indexing of integers (signed and unsigned), and, consequently, to minimize the number of unique words, which, in turn, greatly affects to performance of searching.

Dictionary accepts two init options:

- MAXLEN parameter specifies maximum length of the number considered as a 'good' integer. Default value is 6.
- REJECTLONG parameter specifies if 'long' integer should be indexed or treated as a stop-word. If REJECTLONG=FALSE (default), then dictionary returns prefixed part of integer number with length MAXLEN. If REJECTLONG=TRUE, than dictionary consider integer as a stop word.

Similar idea can be applied to the indexing of decimal numbers, for example, DecDict dictionary. Dictionary accepts two init options: MAXLENFRAC parameter specifies maximum length of the fraction part considered as a 'good' decimal, default value is 3. REJECTLONG parameter specifies if decimal number with 'long' fraction part should be indexed or treated as a stop word. If REJECTLONG=FALSE (default), than dictionary returns decimal number with length of fraction part MAXLEN. If REJECTLONG=TRUE, than dictionary consider number as a stop word. Notice, that REJECTLONG=FALSE allow indexing 'shortened' numbers and search results will contain documents with original 'garbage' numbers.

Examples:

```sql
=# select lexize('intdict', 11234567890);
 lexize
----------
     {112345}
```

Now, we want to ignore long integers.

```sql
=# ALTER FULLTEXT DICTIONARY intdict SET OPTION 'MAXLEN=6, REJECTLONG=TRUE';
=# select lexize('intdict', 11234567890);
 lexize
----------
   {}
```

Create contrib/dict_intdict directory with files dict_tmpl.c, Makefile, dict_intdict.sql.in, then

```bash
make && make install
psql DBNAME < dict_intdict.sql
```
This is a dict_tmpl.c file.

```
#include "postgres.h"
#include "utils/builtins.h"
#include "fmgr.h"

#ifdef PG_MODULE_MAGIC
PG_MODULE_MAGIC;
#endif

#include "utils/ts_locale.h"
#include "utils/ts_public.h"
#include "utils/ts_utils.h"

typedef struct {
    int maxlen;
    bool rejectlong;
} DictInt;

PG_FUNCTION_INFO_V1(dinit_intdict);
Datum dinit_intdict(PG_FUNCTION_ARGS);

Datum dinit_intdict(PG_FUNCTION_ARGS) {
    DictInt *d = (DictInt*)malloc( sizeof(DictInt) );
    Map *cfg, *pcfg;
    text *in;
    
    if ( !d )
        elog(ERROR, "No memory");
    memset(d,0,sizeof(DictInt));
    /* Your INIT code */
    /* defaults */
    d->maxlen = 6;
    d->rejectlong = false;
    
    if ( PG_ARGISNULL(0) || PG_GETARG_POINTER(0) == NULL ) { /* no options */
        PG_RETURN_POINTER(d);
    }
    
    in = PG_GETARG_TEXT_P(0);
    parse_keyvalpairs(in,&cfg);
    PG_FREE_IF_COPY(in, 0);
    pcfg=cfg;
    
    while (pcfg->key) {
        if ( strcmp(pcfg->key, "MAXLEN") == 0 ) {
            d->maxlen=atoi(pcfg->value);
        } else if ( strcmp(pcfg->key, "REJECTLONG") == 0 ) {
            if ( strcmp("true", pcfg->value) == 0 ) {
                d->rejectlong=true;
            } else if ( strcmp("false", pcfg->value) == 0 ) {
            }
            
    } else {
        
    } else {
        
    }
```
Appendix C. FTS Dictionary Example

```c
#define rejectlong=false;

} else {
    elog(ERROR,"Unknown value: %s => %s", pcfg->key, pcfg->value);
}

} else {
    elog(ERROR,"Unknown option: %s => %s", pcfg->key, pcfg->value);
}

pfree(pcfg->key);
pfree(pcfg->value);

pcfg++;
}
pfree(cfg);

PG_RETURN_POINTER(d);

PG_FUNCTION_INFO_V1(dlexize_intdict);
Datum dlexize_intdict(PG_FUNCTION_ARGS);
Datum dlexize_intdict(PG_FUNCTION_ARGS) {
    DictInt *d = (DictInt*)PG_GETARG_POINTER(0);
    char *in = (char*)PG_GETARG_POINTER(1);
    char *txt = pnstrdup(in, PG_GETARG_INT32(2));
    TSLexeme *res = palloc(sizeof(TSLexeme)*2);

    /* Your INIT dictionary code */
    res[1].lexeme = NULL;
    if ( PG_GETARG_INT32(2) > d->maxlen ) {
        if ( d->rejectlong ) { /* stop, return void array */
            pfree(txt);
            res[0].lexeme = NULL;
        } else { /* cut integer */
            txt[d->maxlen] = '\0';
            res[0].lexeme = txt;
        }
    } else {
        res[0].lexeme = txt;
    }

    PG_RETURN_POINTER(res);
}
```

This is a Makefile:

```bash
subdir = contrib/dict_intdict
top_builddir = ../..
include $(top_builddir)/src/Makefile.global

MODULE_big = dict_intdict
OBJS = dict_tmpl.o
DATA_built = dict_intdict.sql
```
DOCS =

include $(top_srcdir)/contrib/contrib-global.mk

This is a dict_intdict.sql.in:

SET search_path = public;
BEGIN;

CREATE OR REPLACE FUNCTION dinit_intdict(internal)
  returns internal
  as 'MODULE_PATHNAME'
  language 'C';

CREATE OR REPLACE FUNCTION dlexize_intdict(internal,internal,internal,internal)
  returns internal
  as 'MODULE_PATHNAME'
  language 'C'
  with (isstrict);

CREATE FULLTEXT DICTIONARY intdict
  LEXIZE 'dlexize_intdict' INIT 'dinit_intdict'
  OPTION 'MAXLEN=6,REJECTLONG=false'
;

COMMENT ON FULLTEXT DICTIONARY intdict IS 'Dictionary for Integers';

END;
Index

Symbols

!! TSQUERY, ?

A

ALTER FULLTEXT ... OWNER, ?
ALTER FULLTEXT CONFIGURATION, ?
ALTER FULLTEXT DICTIONARY, ?
ALTER FULLTEXT MAPPING, ?
ALTER FULLTEXT PARSER, ?

B

Btree operations for TSQUERY, ?
Btree operations for tsvector, ?

C

COMMENT ON FULLTEXT, ?
CREATE FULLTEXT CONFIGURATION, ?
CREATE FULLTEXT DICTIONARY, ?
CREATE FULLTEXT MAPPING, ?
CREATE FULLTEXT PARSER, ?

D

document, ?
DROP FULLTEXT CONFIGURATION, ?
DROP FULLTEXT DICTIONARY, ?
DROP FULLTEXT MAPPING, ?
DROP FULLTEXT PARSER, ?

F

FTS, ?
full-text index
GIN, ?

GIST, ?

H

headline, ?

I

index
full-text, ?

L

length(tsvector), ?
lexize, ?

N

numnode, ?

P

parse, ?
plainto_tsquery, ?

Q

querytree, ?

R

rank, ?
rank_cd, ?
rewrite - 1, ?
rewrite - 2, ?
rewrite - 3, ?
setweight, ?
stat, ?
strip, ?

TEXT @@ TEXT, ?
TEXT @@ TSQUERY, ?
text::tsquery casting, ?
text::tsvector, ?
token_type, ?
to_tsquery, ?
to_tsvector, ?
tsearch trigger, ?
tquery, ?
TSQUERY && TSQUERY, ?
TSQUERY <@ TSQUERY, ?
TSQUERY @> TSQUERY, ?
TSQUERY @@ TSVECTOR, ?
TSQUERY || TSQUERY, ?
tsvector, ?
tsvector concatenation, ?