Do you need a Full-Text Search in PostgreSQL?

Oleg Bartunov
Postgres Professional, Moscow University
obartunov@postgrespro.ru

PGConf.eu, Oct 26, 2018, Lisbon
Oleg Bartunov, major PostgreSQL contributor since Postgres95
What is a Full Text Search?

- Full text search
  - Find documents, which match a query
  - Sort them in some order (optionally)

- Typical Search
  - Find documents with **all words** from the query
  - Return them sorted by relevance
What is a document?

- Arbitrary text attribute
- Combination of text attributes from the same or different tables (result of join)

```sql
msg (id, lid, subject, body);
lists (lid, list);

SELECT l.list || m.subject || m.body_plain as doc;

Don’t forget about COALESCE (text,’”’)
```
What is a query?

- Arbitrary text
  ‘open source’
- Text with some query language

'postgresql "open source * database" -die +most'
Why FTS in PostgreSQL?

- Feed database content to external search engines
  - They are fast!

**BUT**

- They can't index all documents - could be totally virtual
- They don't have access to attributes - no complex queries
- They have to be maintained — headache for DBA
- Sometimes they need to be certified
- They don't provide instant search (need time to download new data and reindex)
- They don't provide consistency — search results can be already deleted from database
Your system may looks like this
FTS in PostgreSQL

- **FTS requirements**
  - Full integration with database engine
    - Transactions
    - Concurrent access
    - Recovery
    - Online index
  - Configurability (parser, dictionary...)
  - Scalability
Text Search Operators

• Traditional text search operators
  ( TEXT op TEXT, op - ~, ~*, LIKE, ILIKE)

```sql
=# select title from apod where title ~* 'x-ray' limit 5;
title
----------------------------------------
The X-Ray Moon
Vela Supernova Remnant in X-ray
Tycho's Supernova Remnant in X-ray
ASCA X-Ray Observatory
Unexpected X-rays from Comet Hyakutake
(5 rows)

=# select title from apod where title ilike '%x-ray%' limit 5;
title
----------------------------------------
The Crab Nebula in X-Rays
X-Ray Jet From Centaurus A
The X-Ray Moon
Vela Supernova Remnant in X-ray
Tycho's Supernova Remnant in X-ray
(5 rows)
```
Text Search Operators

- Traditional text search operators
  \[(\text{TEXT op TEXT}, \text{op} = \sim, \sim^*, \text{LIKE}, \text{ILIKE})\]
  - No linguistic support
    - What is a word?
    - What to index?
    - Word «normalization»?
    - Stop-words (noise-words)
  - No ranking - all documents are equally similar to query
  - Slow, documents should be seq. scanned

9.3+ index support of \sim^* (pg_trgm)

```sql
select * from man_lines where man_line \sim^* '(?:postgresql|pgsql|psql|sql|postgres|gsql) (?:(?:do|is|use|make))';
```

One of (postgresql,sql,postgres,pgsql,psql) space One of (do,is,use,make)
FTS in PostgreSQL

- OpenFTS — 2000, Pg as a storage
- GiST index — 2000, thanks Rambler
- Tsearch — 2001, contrib:no ranking
- Tsearch2 — 2003, contrib:config
- GIN — 2006, thanks, JFG Networks
- FTS — 2006, in-core, thanks, EnterpriseDB
- RUM — 2016, extension, Postgres Pro

Team:
Teodor Sigaev, Oleg Bartunov, Alexander Korotkov, Arthur Zakirov
FTS data types and operators

- **tsvector** – data type for document optimized for search
  - Sorted array of lexems
  - Positional information
  - Structural information (importance)
- **tsquery** – textual data type for query with boolean operators & | ! ()

- **Full text search operator:** `tsvector @@ tsquery`

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

1) Parser breaks text on tokens (token, type) pairs
2) Tokens converted to lexems using dictionaries specific for token type

• Extendability:
  • Pluggable parser and dictionaries
  • FTS configuration defines parser and dictionaries
  • FTS configurations used for document and query processing

• \dF{,p,d}[+] [pattern] — psql FTS

• SQL interface:

  {CREATE | ALTER | DROP} TEXT SEARCH {CONFIGURATION | DICTIONARY | PARSER}
Document to tsvector:

- `to_tsvector([cfg], text|json|jsonb)`
  - `cfg` — FTS configuration, GUC `default_text_search_config`

```
select to_tsvector('It is a very long story about true and false');
    to_tsvector
---------------------------------------
   'fals':10 'long':5 'stori':6 'true':8
(1 row)
```

```
select to_tsvector('simple', 'It is a very long story about true and false');
    to_tsvector
---------------------------------------------------------------------------------------
   'a':3 'about':7 'and':9 'false':10 'is':2 'it':1 'long':5 'story':6 'true':8 'very':4
(1 row)
```
FTS in PostgreSQL

- JSON[b] to tsvector:
  - Notice, results are different for json and jsonb!
    Jsonb: keys are sorted, Json: spaces are preserved
  - Phrases are preserved

```sql
select to_tsvector(jb) from (values ('{
    "abstract": "It is a very long story about true and false",
    "title": "Peace and War",
    "publisher": "Moscow International house"
}')) foo(jb) as tsvector_json

---------------------------------------------------------------
fals':10 'hous':18 'intern':17 'long':5 'moscow':16 'peac':12 'stori':6 'true':8 'war':14 (1 row)

---------------------------------------------------------------
tsvecto_jsonb

---------------------------------------------------------------
fals':14 'hous':18 'intern':17 'long':9 'moscow':16 'peac':1 'stori':10 'true':12 'war':3 (1 row)
```
Tsvector editing functions

- Different parts of document can be marked to use for ranking at search time.
  `setweight(tsvector, «char», text[])` - add label to lexemes from text[]

```sql
select setweight( to_tsvector('english', '20-th anniversary of PostgreSQL'), 'A', '{postgresql,20}');
setweight
------------------------------------------------
'20':1A 'anniversari':3 'postgresql':5A 'th':2
(1 row)
```

- `ts_delete(tsvector, text[])` - delete lexemes from tsvector

```sql
select ts_delete( to_tsvector('english', '20-th anniversary of PostgreSQL'), '{20,postgresql}'::text[]);
ts_delete
------------------------
'anniversari':3 'th':2
(1 row)
```
Tsvector editing functions

- **unnest(tsvector)**

```sql
select * from unnest( setweight( to_tsvector('english', '20-th anniversary of PostgreSQL'),'A', '{postgresql,20}'));
```

<table>
<thead>
<tr>
<th>lexeme</th>
<th>positions</th>
<th>weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>{1}</td>
<td>{A}</td>
</tr>
<tr>
<td>anniversari</td>
<td>{3}</td>
<td>{D}</td>
</tr>
<tr>
<td>postgresql</td>
<td>{5}</td>
<td>{A}</td>
</tr>
<tr>
<td>th</td>
<td>{2}</td>
<td>{D}</td>
</tr>
</tbody>
</table>

(4 rows)

- **tsvector_to_array(tsvector) — tsvector to text[]**

```sql
array_to_tsvector(text[])
```

```sql
select tsvector_to_array( to_tsvector('english', '20-th anniversary of PostgreSQL'));
```

```text
{20,anniversari,postgresql,th}
```

(1 row)
Tsvector editing functions

- `ts_filter(tsvector, text[])` - fetch lexemes with specific label(s)

```
select ts_filter($$'20':2A 'anniversari':4C 'postgresql':1A,6A 'th':3$$::tsvector, '{C}');
  ts_filter
--------------------
 'anniversari':4C
(1 row)

select ts_filter($$'20':2A 'anniversari':4C 'postgresql':1A,6A 'th':3$$::tsvector, '{C,A}');
  ts_filter
---------------------------------------------------------------
 '20':2A 'anniversari':4C 'postgresql':1A,6A
(1 row)
```
DOCUMENT

PARSER

(token, token_type)

dicts(token_type)

i=0

NO

YES

i=i+1

i < N

NO

YES

IS STOP ?

YES

NO

NO

tsvector

to_tsvector(cfg, doc)
Parser breaks document into tokens

```sql
=# select * from ts_token_type('default');

<table>
<thead>
<tr>
<th>tokid</th>
<th>alias</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>asciiword</td>
<td>Word, all ASCII</td>
</tr>
<tr>
<td>2</td>
<td>word</td>
<td>Word, all letters</td>
</tr>
<tr>
<td>3</td>
<td>numword</td>
<td>Word, letters and digits</td>
</tr>
<tr>
<td>4</td>
<td>email</td>
<td>Email address</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 number</td>
</tr>
<tr>
<td>9</td>
<td>hword_numpart</td>
<td>Hyphenated word part, letters and digits</td>
</tr>
<tr>
<td>10</td>
<td>hword_part</td>
<td>Hyphenated word part, all letters</td>
</tr>
<tr>
<td>11</td>
<td>hword_asciipart</td>
<td>Hyphenated word part, all ASCII</td>
</tr>
<tr>
<td>12</td>
<td>blank</td>
<td>Space symbols</td>
</tr>
<tr>
<td>13</td>
<td>tag</td>
<td>XML tag</td>
</tr>
<tr>
<td>14</td>
<td>protocol</td>
<td>Protocol head</td>
</tr>
<tr>
<td>15</td>
<td>numhword</td>
<td>Hyphenated word, letters and digits</td>
</tr>
<tr>
<td>16</td>
<td>asciihword</td>
<td>Hyphenated word, all ASCII</td>
</tr>
<tr>
<td>17</td>
<td>hword</td>
<td>Hyphenated word, all letters</td>
</tr>
<tr>
<td>18</td>
<td>url_path</td>
<td>URL path</td>
</tr>
<tr>
<td>19</td>
<td>file</td>
<td>File or path name</td>
</tr>
<tr>
<td>20</td>
<td>float</td>
<td>Decimal notation</td>
</tr>
<tr>
<td>21</td>
<td>int</td>
<td>Signed integer</td>
</tr>
<tr>
<td>22</td>
<td>uint</td>
<td>Unsigned integer</td>
</tr>
<tr>
<td>23</td>
<td>entity</td>
<td>XML entity</td>
</tr>
</tbody>
</table>
```

(23 rows)
**Dictionaries**

- **Dictionary** – is a **program**, which accepts token on input and returns an array of lexems, NULL if token doesn't recognized and empty array for stop-word.

- `ts_lexize(dictionary)`

  ```sql
  SELECT ts_lexize('english_hunspell','a') as stop,
         ts_lexize('english_hunspell','elephants') AS elephants,
         ts_lexize('english_hunspell','elephantus') AS unknown;
  ```

<table>
<thead>
<tr>
<th>stop</th>
<th>elephants</th>
<th>unknown</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
<td>{elephant}</td>
<td>(null)</td>
</tr>
</tbody>
</table>

  (1 row)

- Dictionary API allows to develop any custom dictionaries
  - Truncate too long numbers
  - Convert colors
  - Convert URLs to canonical way
    
    `http://a.in/a/./index.html` → `http://a.in/a/index.html`
Dictionaries

- Dictionary — is a program!

```sql
=# select ts_lexize('intdict', 11234567890);
    ts_lexize
----------
   {112345}
=# select ts_lexize('roman', 'XIX');
    ts_lexize
----------
   {19}
=# select ts_lexize('colours', '#FFFFFF');
    ts_lexize
----------
   {white}
```
Astronomical dictionary

Dictionary with regexp support (pcre library)

# Messier objects
(M|Messier)(\s-)?((\d){1,3}) M$3
# catalogs
(NGC|Abell|MKN|IC|H[DHR]|UGC|SAO|MWC)(\s-)?((\d){1,6}[ABC]?) $1$3
(PSR|PKS)(\s-)?([JB]?)(\d\d\d\d)\s?([-+\d\d]\d? $1$4$5
# Surveys
OGLE(\s-)?((I){1,3}) ogle
2MASS twomass
# Spectral lines
H(\s-)?(alpha|beta|gamma) h$2
(Fe|Mg|Si|He|Ni)(\s-)?((\d)|([IXV]+) $1$3
# GRBs
gamma\s?ray\s?burst(s?) GRB
GRB\s?((\d)\d\d\d\d\d)([abcd]?) GRB$1$2

SELECT ts_lexize('regex', 'ngc 1234');
  ts_lexize
     ----------
    {ngc1234}
(1 row)
Dictionary templates:

1. Simple
   - convert the input token to lower case
   - exclude stop words

2. Synonym (also, contrib/xsyn)
   - replace word with a synonym

Example of .syn file:

postgres   pgsql
postgresql pgsql
postgre    pgsql
3. Thesaurus
   • replace phrase by indexed phrase

   Example of .ths file:
   
   booking tickets : order invitation cards
   booking ? tickets : order invitation Cards

4. Snowball stemmer
   • reduce words by stemming algorithms
   • recognizes everything
   • exclude stop words

   SELECT ts_lexize('portuguese_stem','responsáveis');
   ts_lexize
   ------------
   {respons}
   (1 row)
Built-in Dictionaries

- Portuguese snowball stemmer dictionary

  viva  | vivo  | viver  
  -------+-------+--------
  {viv}  | {viv}  | {viv}  

  select ts_lexize('portuguese_stem','responsáveis');
  ts_lexize
  ------------
  {respons}  
  (1 row)

- Available as a part of PostgreSQL core
5. Ispell

- normalize different linguistic forms of a word into the same lexeme. Try to reduce an input word to its infinitive form.
- exclude stop words.

<table>
<thead>
<tr>
<th>viva</th>
<th>vivo</th>
<th>viver</th>
</tr>
</thead>
<tbody>
<tr>
<td>{viva,vivo,viver}</td>
<td>{vivo,viver}</td>
<td>{viver}</td>
</tr>
</tbody>
</table>
Filter dictionary – unaccent

contrib/unaccent - unaccent text search dictionary and function to remove accents (suffix tree, ~ 25x faster translate() solution)

1. Unaccent dictionary does nothing and returns NULL. (lexeme 'Hotels' will be passed to the next dictionary if any)

```sql
=# select ts_lexize('unaccent','Hotels') is NULL;
?column?
----------
t
```

2. Unaccent dictionary removes accent and returns 'Hotel'. (lexeme 'Hotel' will be passed to the next dictionary if any)

```sql
=# select ts_lexize('unaccent','Hôtel');
   ts_lexize
----------
{Hotel}
```
CREATE TEXT SEARCH CONFIGURATION fr ( COPY = french );
ALTER TEXT SEARCH CONFIGURATION fr ALTER MAPPING FOR hword, hword_part, word
WITH unaccent, french_stem;

=# select to_tsvector('fr','Hôtel de la Mer') @@ to_tsquery('fr','Hotels');
?column?  
---------
t

=# select ts_headline('fr','Hôtel de la Mer',to_tsquery('fr','Hotels'));
  ts_headline
------------------------
  <b>Hôtel</b> de la Mer
Each token processed by a set of dictionaries

```sql
=# \dF+ russian

Text search configuration "pg_catalog.russian"
Parser: "pg_catalog.default"

<table>
<thead>
<tr>
<th>Token</th>
<th>Dictionaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>asciihword</td>
<td>english_stem</td>
</tr>
<tr>
<td>asciiword</td>
<td>english_stem</td>
</tr>
<tr>
<td>email</td>
<td>simple</td>
</tr>
<tr>
<td>file</td>
<td>simple</td>
</tr>
<tr>
<td>float</td>
<td>simple</td>
</tr>
<tr>
<td>host</td>
<td>simple</td>
</tr>
<tr>
<td>hword</td>
<td>russian_stem</td>
</tr>
<tr>
<td>hword_asciipart</td>
<td>english_stem</td>
</tr>
<tr>
<td>hword_numpart</td>
<td>simple</td>
</tr>
<tr>
<td>hword_part</td>
<td>russian_stem</td>
</tr>
<tr>
<td>int</td>
<td>simple</td>
</tr>
<tr>
<td>numhword</td>
<td>simple</td>
</tr>
<tr>
<td>numword</td>
<td>simple</td>
</tr>
<tr>
<td>sfloat</td>
<td>simple</td>
</tr>
<tr>
<td>uint</td>
<td>simple</td>
</tr>
<tr>
<td>url</td>
<td>simple</td>
</tr>
<tr>
<td>url_path</td>
<td>simple</td>
</tr>
<tr>
<td>version</td>
<td>simple</td>
</tr>
<tr>
<td>word</td>
<td>russian_stem</td>
</tr>
</tbody>
</table>
```

```sql
ts_lexize('english_stem','stars')
```

star
FTS in PostgreSQL

- Token processed by dictionaries until it recognized
- It is discarded, if it's not recognized

Rule: from «specific» dictionary to a «common» dictionary

```sql
=# \dF+ pg
Configuration "public.pg"
Parser name: "pg_catalog.default"
Locale: 'ru_RU.UTF-8' (default)

<table>
<thead>
<tr>
<th>Token</th>
<th>Dictionaries</th>
</tr>
</thead>
<tbody>
<tr>
<td>file</td>
<td>pg_catalog.simple</td>
</tr>
<tr>
<td>host</td>
<td>pg_catalog.simple</td>
</tr>
<tr>
<td>hword</td>
<td>pg_catalog.simple</td>
</tr>
<tr>
<td>int</td>
<td>pg_catalog.simple</td>
</tr>
<tr>
<td>lhword</td>
<td>public.pg_dict, public.en_ispell, pg_catalog.en_stem</td>
</tr>
<tr>
<td>lpart_hword</td>
<td>public.pg_dict, public.en_ispell, pg_catalog.en_stem</td>
</tr>
<tr>
<td>Lword</td>
<td>public.pg_dict, public.en_ispell, pg_catalog.en_stem</td>
</tr>
<tr>
<td>nlhword</td>
<td>pg_catalog.simple</td>
</tr>
<tr>
<td>nlpart_hword</td>
<td>pg_catalog.simple</td>
</tr>
</tbody>
</table>
```

lowercase
Stemmer recognizes everything
FTS in PostgreSQL

What is the benefit?
Document processed only once when inserting to table, no overhead in search

• Document parsed into tokens using pluggable parser
• Tokens converted to lexems using pluggable dictionaries
• Words positions and importance are stored and used for ranking
• Stop-words ignored
Query processing

• Query to tsquery:
  • to_tsquery([cfg], text)

    Better, always specify cfg (immutable vs stable)!

    ```sql
    select to_tsquery('supernovae & stars');
    to_tsquery
    ----------------------
    'supernova' & 'star'
    (1 row)
    ```

• plainto_tsquery([cfg],text) – words are AND-ed

    ```sql
    select plainto_tsquery('supernovae stars');
    plainto_tsquery
    ----------------------
    'supernova' & 'star'
    (1 row)
    ```
Query processing

• Queries 'A & B'::tsquery and 'B & A'::tsquery are equivalent

```
select 'a:1 b:2'::tsvector @@ 'a & b'::tsquery,
     'a:1 b:2'::tsvector @@ 'b & a'::tsquery;
?column? | ?column?
----------+----------
t        | t
```

• Phrase query: FOLLOWED BY operators <n>,<->

• Guarantee an order (and distance) of operands

• Precendence of tsquery operators - '!' <-> & |

```
select 'a:1 b:2'::tsvector @@ 'a <-> b'::tsquery,
     'a:1 b:2'::tsvector @@ 'b <-> a'::tsquery;
?column? | ?column?
----------+----------
t        | f
```
Phrase search - properties

- Precendence of tsquery operators - '!' '<->' '&' '|'  
  Use parenthesis to control nesting in tsquery

```sql
select 'a & b <-> c'::tsquery;
  tsquery
-------------------
 'a' & 'b' <-> 'c'

select 'b <-> c & a'::tsquery;
  tsquery
-------------------
 'b' <-> 'c' & 'a'

select 'b <-> (c & a)'::tsquery;
  tsquery
---------------------------
 'b' <-> 'c' & 'b' <-> 'a'
```
Phrase search - example

- `phraseto_tsquery([CFG,] TEXT)`
  
  ```
  select phraseto_tsquery('english','PostgreSQL can be extended by the user in many ways');
  phraseto_tsquery
  'postgresql' <3> 'extend' <3> 'user' <2> 'mani' <-> 'way'
  (1 row)
  ```

  Stop words are taken into account!

- It’s possible to combine tsquery’s

  ```
  select phraseto_tsquery('PostgreSQL can be extended by the user in many ways') ||
  to_tsquery('oho<->ho & ik');
  ?column?
  'postgresql' <3> 'extend' <3> 'user' <2> 'mani' <-> 'way' | 'oho' <-> 'ho' & 'ik'
  (1 row)
  ```
Query processing

- `websearch_to_tsquery([cfg], text)`
  - Recognizes “phrases”, AND, OR, *, +word, -word

```sql
select websearch_to_tsquery('english','postgresql "open source * database" -die +most');
  websearch_to_tsquery
  'postgresql' & 'open' <-> 'sourc' <2> 'databas' & !'die'
(1 row)
```

```sql
select to_tsvector('english', 'PostgreSQL: The Worlds Most Advanced Open Source Relational Database') @@
websearch_to_tsquery('english','postgresql "open source * database" -die +most');
?column?
-----------------------------
t
(1 row)
```
Supernovae & stars

FTS PostgreSQL
to_tsquery

Supernovae

stars

{supernova, sn}

star

I

supernova

sn

(supernova | sn) & star

Supernovae

&

star

{supernova, sn}

&

star

I

supernova

sn

(supernova | sn) & star

FTS PostgreSQL
to_tsquery

Supernovae & stars

Foreach leaf node

Supernovae

stars

{supernova, sn}

star

I

supernova

sn

(supernova | sn) & star

FTS PostgreSQL
to_tsquery

Supernovae & stars

Foreach leaf node

Supernovae

stars

{supernova, sn}

star

I

supernova

sn

(supernova | sn) & star
FTS: additional functions

- `ts_debug(cfg, text)` – good for debugging FTS configuration
- `ts_stat` – word frequencies
- `ts_parse(parser, text)` – produces *(token_type, token)* from a text
- `ts_rewrite` – rewrite query online, no reindexing needed
- `ts_headline` – pieces of documents with words from query

Ordering result of FTS:
- `ts_rank` – the more occurrences of words, the bigger rank
  good for OR queries, no query language
- `ts_rank_cd` – the closer words, the bigger rank
  good for AND queries, supports query language
- `rum_ts_score` (requires RUM extension) – combination of the above, the best *(NIST TREC, AD-HOC coll.)*
• FTS in PostgreSQL is a flexible search engine,

• It is a «collection of bricks» you can build your search engine using
  • Custom parser
  • Custom dictionaries
  + All power of SQL (FTS+Spatial+Temporal)
the only weapon that is effective against a werewolf, witch, or other monsters.
Indexes!

- Index is a search tree with tuple pointers in the leaves
- Index has no visibility information (MVCC !)
- Indexes used only for accelerations: Index scan should produce the same results as sequence scan with filtering
- Indexes can be: **partial** (where price > 0.0), **functional** (to\_tsvector(text)), **multicolumn** (timestamp, tsvector)
- Indexes not always useful !
  - Low selectivity
  - Maintainance overhead
FTS Indexes

- **CREATE INDEX ... USING GIST/GIN/RUM (tsvector)**
- **GiST — Generalized Search Tree**
  - Document, query as a signature, documents → signature tree, Bloom filter used for search
- **GIN — inverted tree, basically it’s a B-tree**
  - Optimized for storing a lot of duplicate keys
  - Duplicates are ordered by heap TID
- **RUM (extension)**
  - GIN with additional information (words positions, timestamp, ...)

Understanding GiST (array example)

- **Intarray** - Access Method for array of integers
  - Operators overlap, contains

S1 = \{1,2,3,5,6,9\}
S2 = \{1,2,5\}
S3 = \{0,5,6,9\}
S4 = \{1,4,5,8\}
S5 = \{0,9\}
S6 = \{3,5,6,7,8\}
S7 = \{4,7,9\}

Q = \{2,9\}
"THE RD-TREE: AN INDEX STRUCTURE FOR SETS", Joseph M. Hellerstein
RD-Tree

QUERY

{2,9}

{0,5,6,9} {1,2,3,5,6,9} {1,3,4,5,6,7,8,9}

{0,9} {0,5,6,9} {1,2,3,5,6,9} {1,2,5} {1,4,5,8} {1,3,5,6,7,8} {4,7,9}

S5 S3 S1 S2 S4 S6 S7
**RD-Tree**

```
QUERY

{2,9}  

{0,1,2,3,5,6,9}  
{1,3,4,5,6,7,8,9}  

{0,5,6,9}  
{1,2,3,5,6,9}  
{1,2,3,5,6,9}  

{0,9}  
{0,5,6,9}  
{1,2,3,5,6,9}  
{1,2,3,5,6,9}  

S5  S3  S1  S2  S4  S6  S7
```

**Diagram Description:**

- **QUERY** set: {2,9}
- **Tree Structure**:
  - Node 1: {0,1,2,3,5,6,9}  
    - Children: {0,5,6,9}  
      - Left: {0,9}  
      - Right: {0,5,6,9}  
  - Node 2: {1,3,4,5,6,7,8,9}  
    - Right: {4,7,9}  
      - Left: {1,4,5,8}  
      - Right: {1,3,5,6,7,8}  
      - Right: {4,7,9}
• **Word signature** — words hashed to the specific position of '1'
  
  \[
  \begin{align*}
  w1 & \rightarrow S1: 01000000 & \text{Document: w1 w2 w3} \\
  w2 & \rightarrow S2: 00010000 \\
  w3 & \rightarrow S3: 10000000
  \end{align*}
  \]

• **Document (query) signature** — superposition (bit-wise OR) of signatures
  
  \[
  S: 11010000
  \]

• **Bloom filter**
  
  \[
  \begin{align*}
  Q1: & \quad 00000001 \quad \text{– exact not} \\
  Q2: & \quad 01010000 \quad \text{- may be contained in the document, } \text{{false drop}}
  \end{align*}
  \]

• **Signature is a lossy representation of document**
  
  • + fixed length, compact, + fast bit operations
  
  • - lossy (false drops), - saturation with \#words grows
• Latin proverbs

<table>
<thead>
<tr>
<th>id</th>
<th>proverb</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ars longa, vita brevis</td>
</tr>
<tr>
<td>2</td>
<td>Ars vitae</td>
</tr>
<tr>
<td>3</td>
<td>Jus vitae ac necis</td>
</tr>
<tr>
<td>4</td>
<td>Jus generis humani</td>
</tr>
<tr>
<td>5</td>
<td>Vita nostra brevis</td>
</tr>
</tbody>
</table>
FTS Index (GiST): RD-Tree

<table>
<thead>
<tr>
<th>word</th>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>00000011</td>
</tr>
<tr>
<td>brevis</td>
<td>00001010</td>
</tr>
<tr>
<td>generis</td>
<td>01000100</td>
</tr>
<tr>
<td>humani</td>
<td>00110000</td>
</tr>
<tr>
<td>jus</td>
<td>00010001</td>
</tr>
<tr>
<td>longa</td>
<td>00100100</td>
</tr>
<tr>
<td>necis</td>
<td>01001000</td>
</tr>
<tr>
<td>nostra</td>
<td>10000001</td>
</tr>
<tr>
<td>vita</td>
<td>01000001</td>
</tr>
<tr>
<td>vitae</td>
<td>00011000</td>
</tr>
</tbody>
</table>

Root: 11011011

Internal nodes: 11011001, 10010011

Leaf nodes: 1101000, 11010001, 11011000, 10010010, 10010001

QUERY
## RD-Tree (GiST)

<table>
<thead>
<tr>
<th>id</th>
<th>proverb</th>
<th>signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ars longa, vita brevis</td>
<td>11101111</td>
</tr>
<tr>
<td>2</td>
<td>Ars vitae</td>
<td>11011000</td>
</tr>
<tr>
<td>3</td>
<td>Jus vitae ac necis</td>
<td>01011011</td>
</tr>
<tr>
<td>4</td>
<td>Jus generis humani</td>
<td>01110101</td>
</tr>
<tr>
<td>5</td>
<td>Vita nostra brevis</td>
<td>11001011</td>
</tr>
</tbody>
</table>
• Problems
  • Not good scalability with increasing of cardinality of words and records.
  • Index is lossy, need check for false drops
    (Recheck в EXPLAIN ANALYZE)
Report Index

A
abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61,
73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 61
aerospace propulsion, 52
aerospace robotics, 68
aluminium, 17
amorphous state, 67
angular velocity measurement, 58
antenna phased arrays, 41, 46, 66
argon, 21
assembling, 22
atomic force microscopy, 13, 27, 35
atomic layer deposition, 15
attitude control, 60, 61
attitude measurement, 59, 61
automatic test equipment, 71
automatic testing, 24

B
backward wave oscillators, 45
compensation, 30, 68
compressive strength, 54
compressors, 29
computational fluid dynamics, 23, 29
computer games, 56
concurrent engineering, 14
contact resistance, 47, 66
converters, 22
coplanar waveguide components, 40
Couette flow, 21
creep, 17
crystallisation, 64
current density, 13, 16

design for manufacture, 25
design for testability, 25
diamond, 3, 27, 43, 54, 67
dielectric losses, 31, 42
dielectric polarisation, 31
dielectric relaxation, 64
dielectric thin films, 16
differential amplifiers, 28
diffraction gratings, 68
discrete wavelet transforms, 72
displacement measurement, 11
display devices, 56
distributed feedback lasers, 38
QUERY: compensation accelerometers

INDEX:  accelerometers  compensation
        5,10,25,28,30,36,58,59,61,73,74  30,68

RESULT: 30
Inverted Index in PostgreSQL

No positions in index!

Posting list
Posting tree

Report Index

A
abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 61
aerospace propulsion, 52
aerospace robotics, 68
aluminium, 17
amorphous state, 67
angular velocity measurement, 58
antenna phased arrays, 41, 46, 66
argon, 21
assembling, 22
atomic force microscopy, 13, 27, 35
atomic layer deposition, 15
attitude control, 60, 61
attitude measurement, 59, 61
automatic test equipment, 71
automatic testing, 24

B
backward wave oscillators, 45
GIN

• Internal structure is basically just a B-tree
  • Optimized for storing a lot of duplicate keys
  • Duplicates are ordered by heap TID
• Interface supports indexing more than one key per indexed value
  • Full text search: “foo bar” → “foo”, “bar”
• Bitmap scans only
## Demo collections – latin proverbs

<table>
<thead>
<tr>
<th>id</th>
<th>proverb</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ars longa, vita brevis</td>
</tr>
<tr>
<td>2</td>
<td>Ars vitae</td>
</tr>
<tr>
<td>3</td>
<td>Jus vitae ac necis</td>
</tr>
<tr>
<td>4</td>
<td>Jus generis humani</td>
</tr>
<tr>
<td>5</td>
<td>Vita nostra brevis</td>
</tr>
</tbody>
</table>
GIN Index

Inverted Index

<table>
<thead>
<tr>
<th>word</th>
<th>posting</th>
</tr>
</thead>
<tbody>
<tr>
<td>ac</td>
<td>{3}</td>
</tr>
<tr>
<td>ars</td>
<td>{1,2}</td>
</tr>
<tr>
<td>brevis</td>
<td>{1,5}</td>
</tr>
<tr>
<td>generis</td>
<td>{4}</td>
</tr>
<tr>
<td>humani</td>
<td>{4}</td>
</tr>
<tr>
<td>jus</td>
<td>{3,4}</td>
</tr>
<tr>
<td>longa</td>
<td>{1}</td>
</tr>
<tr>
<td>necis</td>
<td>{3}</td>
</tr>
<tr>
<td>nostra</td>
<td>{5}</td>
</tr>
<tr>
<td>vita</td>
<td>{1,5}</td>
</tr>
<tr>
<td>vitae</td>
<td>{2,3}</td>
</tr>
</tbody>
</table>

Entry tree

Posting tree

- Fast search
- Slow update
RUM index (GIN ++)
• Solve problem of slow ranking

postgres=# explain analyze
SELECT docid, ts_rank(text_vector, to_tsquery('english', 'title')) AS rank
FROM ti2
WHERE text_vector @@ to_tsquery('english', 'title')
ORDER BY rank DESC
LIMIT 3;

Limit (cost=8087.40..8087.41 rows=3 width=282) (actual time=433.752 rows=3 loops=1)
  ->  Sort (cost=8087.40..8206.63 rows=47692 width=282)
      (actual time=433.749..433.749 rows=3 loops=1)
         Sort Key: (ts_rank(text_vector, '''titl'''::tsquery))
         Sort Method: top-N heapsort  Memory: 25kB
      ->  Bitmap Heap Scan on ti2  (cost=529.61..7470.99 rows=47692 width=282)
          (actual time=15.094..423.452 rows=47855 loops=1)
             Recheck Cond: (text_vector @@ '''titl'''::tsquery)
          ->  Bitmap Index Scan on ti2_index  (cost=0.00..517.69 rows=47692 width=0)
              (actual time=13.736..13.736 rows=47855 loops=1)
                 Index Cond: (text_vector @@ '''titl'''::tsquery)
Total runtime: 433.787 ms
Improve ranking performance

- Store positions in RUM to calculate rank and order results
- Introduce distance operator `tsvector <-> tsquery`

```sql
CREATE INDEX ti2_rum_fts_idx ON ti2 USING rum(text_vector rum_tsvector_ops);

SELECT docid, ts_rank(text_vector, to_tsquery('english', 'title')) AS rank
FROM ti2
WHERE text_vector @@ to_tsquery('english', 'title')
ORDER BY
    text_vector <-> plainto_tsquery('english','title')
LIMIT 3;

QUERY PLAN
----------------------------------------------------------------------------------------
Limit (actual time=13.843..13.884 rows=3 loops=1)
  ->  Index Scan using ti2_rum_fts_idx on ti2 (actual time=13.841..13.881 rows=3 loops=1)
      Index Cond: (text_vector @@ '''titl'''::tsquery)
      Order By: (text_vector <-> '''titl'''::tsquery)
Planning time: 0.134 ms
Execution time: 14.030 ms vs 433 ms!
(6 rows)
Combine FTS with ordering by timestamp

Search for «fresh» documents

select date, subject from msg
where tsvector @@ to_tsquery('server & crashed')
order by date <= '2000-01-01'::timestamp limit 5;

Limit (actual time=12.089..12.091 rows=5 loops=1)
  ->  Sort (actual time=12.088..12.089 rows=5 loops=1)
      Sort Key: ((date < '2000-01-01 00:00:00'::timestamp without time zone))
      Sort Method: top-N heapsort  Memory: 25kB
  ->  Bitmap Heap Scan on msg (actual time=5.285..10.784 rows=7467 loops=1)
      Recheck Cond: (tsvector @@ to_tsquery('server & crashed'::text))
      Heap Blocks: exact=6927
  ->  Bitmap Index Scan on msg_gin_idx (actual time=4.196..4.196 rows=7467 loops=1)
      Index Cond: (tsvector @@ to_tsquery('server & crashed'::text))

Planning Time: 0.153 ms
Execution Time: **12.121** ms
(11 rows)
Combine FTS with ordering by timestamp

- Combine FTS with ordering by timestamp
- Store timestamps in additional information
- Order posting tree/list by timestamp

```sql
create index msg_date_rum_idx on msg using rum(tsvector rum_tsvector_timestamp_ops, date) WITH
(attach=date, "to"=tsvector, order_by_attach='t');

select date, subject from msg
where tsvector @@ to_tsquery('server & crashed')
order by date <=| '2000-01-01'::timestamp limit 5;
```

Limit (actual time=0.048..0.071 rows=5 loops=1)
  ->  Index Scan using msg_date_rum_idx on msg (actual
time=0.047..0.069 rows=5 loops=1)
      Index Cond: (tsvector @@ to_tsquery('server &
crashed'::text))
      Order By: (date <=| '2000-01-01 00:00:00'::timestamp without
time zone)
Planning Time: 0.196 ms
Execution Time: **0.095 ms** vs **12.21 s**
(6 rows)
RUM improves phrase search

• 1.1 mln postings
Overhead of phrase search for seqscan is not big

```sql
select count(*) from pglist where fts @@ to_tsquery('english', 'tom <-> lane');
```

```
Finalize Aggregate (actual time=1700.280..1700.280 rows=1 loops=1)
 | Workers Planned: 2
 | Workers Launched: 2
 | Workers Planned: 2
 | Workers Launched: 2
``` 

```
-> Gather (actual time=1700.228..1700.277 rows=3 loops=1)
-> Partial Aggregate (actual time=1696.119..1696.119 rows=1 loops=3)
 | Filter: (fts @@ 'tom' <-> 'lane':tsquery)
 | Rows Removed by Filter: 263664

Planning time: 0.270 ms
Execution time: 1709.092 ms
```

(10 rows)
RUM improves phrase search

- **1.1 mln postings**
  
  Overhead of phrase search for index scan is big!

**GIN index** (1.1 s (\(-\rightarrow)) vs 0.48 s (\&)): Use recheck, phrase is slow vs fts

```sql
select count(*) from pglist where fts @@ to_tsquery('english', 'tom <-> lane');
```

**QUERY PLAN**

```
Aggregate (actual time=1074.983..1074.984 rows=1 loops=1)
  -> Bitmap Heap Scan on pglist (actual time=84.424..1055.770 rows=222777 loops=1)
    Recheck Cond: (fts @@ ''tom'’ <-> ''lane''::tsquery)
    Rows Removed by Index Recheck: 36
    Heap Blocks: exact=105992
  -> Bitmap Index Scan on pglist_gin_idx (actual time=53.628..53.628 rows=222813 loops=1)
    Index Cond: (fts @@ ''tom'’ <-> ''lane''::tsquery)
Planning time: 0.329 ms
Execution time: 1075.157 ms
(9 rows)
```
RUM improves phrase search

• 1.1 mln postings
RUM descreases the overhead of phrase search!

RUM index (0.5 s (<> vs 0.48 s (&)): Use positions in addinfo, no overhead of phrase search!

```
select count(*) from pglist where fts @@ to_tsquery('english', tom <-> lane);
```

```
QUERY PLAN

Aggregate (actual time=513.517..513.517 rows=1 loops=1)
  ->  Bitmap Heap Scan on pglist (actual time=134.109..497.814 rows=221919 loops=1)
      Recheck Cond: (fts @@ to_tsquery('tom <-> lane'::text))
      Heap Blocks: exact=105509
  ->  Bitmap Index Scan on pglist_rum_fts_idx (actual time=98.746..98.746 rows=221919 loops=1)
      Index Cond: (fts @@ to_tsquery('tom <-> lane'::text))
Planning time: 0.223 ms
Execution time: 515.004 ms
(8 rows)
```
Inverse FTS (FQS)

- Find queries, which match given document
- Automatic text classification, subscription service

```
SELECT * FROM queries;

<table>
<thead>
<tr>
<th>q</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>'supernova' &amp; 'star'</td>
<td>sn</td>
</tr>
<tr>
<td>'black'</td>
<td>color</td>
</tr>
<tr>
<td>'big' &amp; 'bang' &amp; 'black' &amp; 'hole'</td>
<td>bang</td>
</tr>
<tr>
<td>'spiral' &amp; 'galaxi'</td>
<td>shape</td>
</tr>
<tr>
<td>'black' &amp; 'hole'</td>
<td>color</td>
</tr>
</tbody>
</table>

(5 rows)
```

```
SELECT * FROM queries WHERE to_tsvector('black holes never exists before we think about them') @@ q;

<table>
<thead>
<tr>
<th>q</th>
<th>tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>'black'</td>
<td>color</td>
</tr>
<tr>
<td>'black' &amp; 'hole'</td>
<td>color</td>
</tr>
</tbody>
</table>

(2 rows)
```
Inverse FTS (FQS)

- RUM index – store branches of query tree in addinfo

Find queries for the first message in postgres mailing lists

```
\d pg_query
  Table "public.pg_query"
  Column | Type    | Modifiers
 --------+---------+-----------
    q   | tsquery |           |
  count  | integer |           |
Indexes:
    "pg_query_rum_idx" rum (q)                  33818 queries

select q from pg_query pgq, pglist where q @@ pglist.fts and pglist.id=1;
  q
   --------------------------
   'one' & 'one'
   'postgresql' & 'freebsd'
(2 rows)
```
Inverse FTS (FQS)

- RUM index support – store branches of query tree in addinfo

Find queries for the first message in postgres mailing lists

```sql
create index pg_query_rum_idx on pg_query using rum(q);
select q from pg_query pgq, pglist where q @@ pglist.fts and pglist.id=1;
```

**QUERY PLAN**

```
Nested Loop (actual time=0.719..0.721 rows=2 loops=1)
  ->  Index Scan using pglist_id_idx on pglist
      (actual time=0.013..0.013 rows=1 loops=1)
      Index Cond: (id = 1)
  ->  Bitmap Heap Scan on pg_query pgq
      (actual time=0.702..0.704 rows=2 loops=1)
      Recheck Cond: (q @@ pglist.fts)
      Heap Blocks: exact=2
      ->  Bitmap Index Scan on pg_query_rum_idx
      (actual time=0.699..0.699 rows=2 loops=1)
      Index Cond: (q @@ pglist.fts)
```

Planning time: 0.212 ms
Execution time: 0.759 ms
(10 rows)```
Inverse FTS (FQS)

- RUM index supported – store branches of query tree in addinfo

Monstrous postings

```
select id, t.subject, count(*) as cnt into pglist_q from pg_query, (select id, fts, subject from pglist) t where t.fts @@ q group by id, subject order by cnt desc limit 1000;
select * from pglist_q order by cnt desc limit 5;
```

<table>
<thead>
<tr>
<th>id</th>
<th>subject</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>248443</td>
<td>Packages patch</td>
<td>4472</td>
</tr>
<tr>
<td>282668</td>
<td>Re: release.sgml, minor pg_autovacuum changes</td>
<td>4184</td>
</tr>
<tr>
<td>282512</td>
<td>Re: release.sgml, minor pg_autovacuum changes</td>
<td>4151</td>
</tr>
<tr>
<td>282481</td>
<td>release.sgml, minor pg_autovacuum changes</td>
<td>4104</td>
</tr>
<tr>
<td>243465</td>
<td>Re: [HACKERS] Re: Release notes</td>
<td>3989</td>
</tr>
</tbody>
</table>

(5 rows)
• **1.1 mln posts**

msg (id, list, subject, author, body, tsvector, date)

GiST: create index msg_gist_idx on msg using gist(tsvector);
GIN:  create index msg_gin_idx on msg using gin(tsvector);
RUM: create index msg_rum_idx on msg using rum(tsvector);
RUM: create index msg_rum_date_idx on msg using rum(tsvector rum_tsvector_timestamp_ops, date) WITH (attach=date, "to"=tsvector);
RUM: create index msg_date_rum_idx on msg using rum(tsvector rum_tsvector_timestamp_ops, date) WITH (attach=date, "to"=tsvector, order_by_attach='t');
RUM size, create index

1.1 mln posts

Size and create index (sec)

```
select pg_size_pretty(pg_table_size('msg')) as msg,
    pg_size_pretty(sum(pg_column_size(tsvector))) as fts,
    pg_size_pretty(pg_table_size('msg_gist_idx')) as gist,
    pg_size_pretty(pg_table_size('msg_gin_idx')) as gin,
    pg_size_pretty(pg_table_size('msg_rum_idx')) as rum,
    pg_size_pretty(pg_table_size('msg_rum_date_idx')) as rum_date,
    pg_size_pretty(pg_table_size('msg_date_rum_idx')) as date_rum
from msg;
```

<table>
<thead>
<tr>
<th>msg</th>
<th>tsvector</th>
<th>gist</th>
<th>gin</th>
<th>rum</th>
<th>rum_date</th>
<th>date_rum</th>
</tr>
</thead>
<tbody>
<tr>
<td>3178 MB</td>
<td>1558 MB</td>
<td>394 MB</td>
<td>462 MB</td>
<td>1130 MB</td>
<td>1812 MB</td>
<td>2596 MB</td>
</tr>
</tbody>
</table>

318  49  112  215  229  706  (sec)
FTS indexes

- GiST
  - document, query as a signature, documents → signature tree, Bloom filter used for search
  - Fast insert, small size, good for small collections
- GIN — inverted tree, basically it’s a B-tree
  - Optimized for storing a lot of duplicate keys
  - Duplicates are ordered by heap TID
  - Not as fast as GiST for updates, good performance and scalability
- RUM (extension) — GIN++
  - Slow for updating, big size, high WAL traffic, best for mostly read workload, very fast for ranking, good for phrase search, no need tsvector column
Ispell shared dictionaries

- Working with dictionaries can be difficult and slow
- Installing dictionaries can be complicated
- Dictionaries are loaded into memory for every session (slow first query symptom) and eat memory.

```bash
time for i in {1..10}; do echo $i; psql postgres -c "select ts_lexize('english_hunspell', 'evening')" > /dev/null; done
1
2
3
4
5
6
7
8
9
10
real 0m0.656s
user 0m0.015s
sys 0m0.031s

For russian hunspell dictionary:

real 0m3.809s
user 0m0.015s
sys 0m0.029s

Each session «eats» 20MB!
Dictionaries as extensions

- Easy installation of hunspell dictionaries

```sql
CREATE EXTENSION hunspell_ru_ru; -- creates russian_hunspell dictionary
CREATE EXTENSION hunspell_en_us; -- creates english_hunspell dictionary
CREATE EXTENSION hunspell_nn_no; -- creates norwegian_hunspell dictionary
SELECT ts_lexize('english_hunspell', 'evening');
  ts_lexize
-----------------
{evening, even}
(1 row)
Time: 57.612 ms
```

```
SELECT ts_lexize('russian_hunspell', 'туши');
  ts_lexize
------------------------
{туша, тушь, тушить, туш}
(1 row)
Time: 382.221 ms
```

```
SELECT ts_lexize('norwegian_hunspell', 'fotballklubber');
  ts_lexize
--------------------------------
{fotball, klubb, fot, ball, klubb}
(1 row)
Time: 323.046 ms
```

Slow first query syndrome
CREATE EXTENSION shared_ispell;
CREATE TEXT SEARCH DICTIONARY english_shared (
    TEMPLATE = shared_ispell,
    DictFile = en_us,
    AffFile = en_us,
    StopWords = english
);
CREATE TEXT SEARCH DICTIONARY russian_shared (
    TEMPLATE = shared_ispell,
    DictFile = ru_ru,
    AffFile = ru_ru,
    StopWords = russian
);

time for i in {1..10}; do echo $i; psql postgres -c "select ts_lexize('russian_shared', 'туши')" > /dev/null; done

1
2
......
10

real  0m0.170s   VS   real  0m3.809s
user  0m0.015s   user  0m0.015s
sys   0m0.027s   sys   0m0.029s
Search Mailing list archive

- https://postgrespro.com/list
- Custom parser — fixes several problems in default parser

```sql
select * from ts_parse('default','1914-1999');
tokid | token
-------+-------
 22 | 1914
 21 | -1999
(2 rows)

select * from ts_parse('tsparser','1914-1999');
tokid |   token
-------+-----------
 15 | 1914-1999
  9 | 1914
 12 | -
  9 | 1999
(4 rows)

select * from ts_parse('default','pg_catalog');
tokid |  token
-------+---------
  1 | pg
 12 | _
  1 | catalog
(3 rows)

select * from ts_parse('tsparser','pg_catalog');
tokid |   token
-------+------------
 16 | pg_catalog
 11 | pg
 12 | _
 11 | catalog
(4 rows)
```
Search Mailing list archive

- https://postgrespro.com/list
- Faceted search - grouping search results by lists
- Strip citation from posts
- Uses pg_trgm for suggestions
- Advanced query language
  - Support «phrase» search
Search Mailing list archive

server crash - Search results in mailing lists

pgsql-general (1037)

2018-10-16 21:25:54 | postgres server process crashes when using odbc_fdw (Ravi Krishna)
server. I also created foreign table. When I run a sql 'select * from odbctest' postgres crashes
Thread >> Search in thread (12)

2018-09-26 14:46:10 | Re: Setting up continuous archiving (Stephen Frost)
server crashes or there's some kind of issue with it after the rsync finishes
Thread

2018-08-29 04:02:45 | WAL replay issue from 9.6.8 to 9.6.10 (Dave Peticolas)
server to 9.6.8 and I was able to replay WAL past the point where 9.6.10 would PANIC and crash
Thread

2018-08-24 19:07:41 | Re: unorthodox use of PG for a customer (David Gauthier)
crash them. Of course any DB running would die too and have to be restarted/recovered. So the place for the DB is really elsewhere, on an external server
Thread

pgsql-hackers (1199)

2018-10-23 21:06:49 | Re: [HACKERS] Transactions involving multiple postgres foreignservers, take 2 (Masahiko Sawada)
References

• Slides of this talk
• Text search documentation
  http://www.postgresql.org/docs/current/static/textsearch.html
• Dictionaries as extensions
  https://github.com/postgrespro/hunspell_dicts
• Improved text search parser
  https://github.com/postgrespro/pg_tsparser
• RUM access method
  https://github.com/postgrespro/rum
• Shared ispell template
  https://github.com/postgrespro/shared_ispell
• Full text search example
  https://github.com/postgrespro/apod_fts
• Dictionary for regular expressions
  https://github.com/obartunov/dict_regex
• Setrank - TF*IDF ranking
  https://github.com/obartunov/setrank
References

- Dictionary for roman numbers
  https://github.com/obartunov/dict_roman
- Faceted search in one query
  http://akorotkov.github.io/blog/2016/06/17/faceted-search/
- FTS real example: Search mailing list archives
  https://postgrespro.com/list
- FTS slides with a lot of info
  http://www.sai.msu.su/~megera/postgres/talks/fts_postgres_by_authors_2.pdf
- Pg_trgm documentation
  https://www.postgresql.org/docs/11/static/pgtrgm.html
- My postings about FTS
  https://obartunov.livejournal.com/tag/fts
Agradeço a vossa atenção!