2ndQuadrant®
PostgreSQL

The State of (Full) Text Search in PostgreSQL 12

FOSDEM 2020

Jimmy Angelakos
Senior PostgreSQL Architect
Twitter: @vyruss

Contents

- (Full) Text Search
- Operators
- Functions
- Dictionaries
- Examples
- Indexing
- Non-natural text
- Collation
- Other “text” types
- Maintenance

Your attention please

Allergy advice

- This presentation contains **linguistics, NLP, Markov chains, Levenshtein distances,** and various other **confounding terms.**
- These have been known to induce **drowsiness** and **inappropriate sleep onset** in lecture theatres.

What is Text?

(Baby don't hurt me)

- PostgreSQL character types
  - `CHAR(n)`
  - `VARCHAR(n)`
  - `VARCHAR`, `TEXT`
    - Trailing spaces: significant (e.g. for `LIKE` / regex)
- Storage
  - Character Set (e.g. `UTF-8`)
  - 1+126 bytes → 4+*n* bytes
  - Compression, TOAST

What is Text Search?

- Information retrieval → Text retrieval
- Search on metadata
  - Descriptive, bibliographic, tags, etc.
  - Discovery & identification
- Search on parts of the text
  - Matching
  - Substring search
  - Data extraction, cleaning, mining

Text search operators in PostgreSQL

- `LIKE, ILIKE (~~, ~~*)`
- `~, ~*` (POSIX regex)
- `regexp_match(string text, pattern text)`
- But are SQL/regular expressions enough?
  - No ranking of results
  - No concept of language
  - Cannot be indexed
    - Okay okay, can be somewhat indexed*
- `SIMILAR TO` → best forget about this one

What is Full Text Search (FTS)?

- Information retrieval → Text retrieval → Document retrieval
- Search on words (on tokens) in a database (all documents)
- No index → Serial search (e.g. `grep`)
- Indexing → Avoid scanning whole documents
- Techniques for criteria-based matching
  - Natural Language Processing (NLP)
- Precision vs Recall
  - Stop words
  - Stemming

Documents? Tokens?

- Document: a chunk of text (a field in a row)
- Parsing of documents into classes of tokens
  - PostgreSQL parser (or write your own… in C)
- Conversion of tokens into *lexemes*
  - *Normalisation* of strings
- Lexeme: an abstract lexical unit representing related words (i.e. word root)
  - `SEARCH → searched, searcher`

Stop words

- Very common and have no value for our search
- Filtering them out increases ***precision*** of search
- Removal based on dictionaries
  - Some check stoplist first
- But: phrase search?

Stemming

- Reducing words to their roots (lexemes)
- Increases number of results (***recall***)
- Algorithms
  - Normalisation using dictionaries
  - Prefix/suffix stripping
  - Automatic production rules
  - Lemmatisation rules
  - *n*-gram models
- Multilingual stemming?

FTS representation in PostgreSQL

- `tsvector`
  - A document!
  - Preprocessed
- `tsquery`
  - Our search query!
  - Normalized into lexemes
- Utility functions
  - `to_tsvector()`, `plainto_tsquery()`, `ts_debug()`, etc.

FTS operators in PostgreSQL

<table>
<tr><td>@@</td><td>tsvector matches tsquery</td></tr>
<tr><td>||</td><td>tsvector concatenation</td></tr>
<tr><td>&&, ||, !!</td><td>tsquery AND, OR, NOT</td></tr>
<tr><td><-></td><td>tsquery followed by tsquery</td></tr>
<tr><td>@></td><td>tsquery contains</td></tr>
<tr><td><@</td><td>tsquery is contained in</td></tr>
</table>

Dictionaries in PostgreSQL

- Programs!
- Accept tokens as input
- Improve search quality
  - Eliminate stop words
  - Normalise words into lexemes
- Reduce size of `tsvector`
- `CREATE TEXT SEARCH DICTIONARY` ***`name`***
  `(TEMPLATE = simple, STOPWORDS = english);`
- Can be chained: most specific → more general
  `ALTER TEXT SEARCH CONFIGURATION` ***`name`***
  `ADD MAPPING FOR word WITH english_ispell, simple;`
- ispell, myspell, hunspell, etc.

Text matching example (1)

```
fts=# SELECT to_tsvector('A nice day for a car ride')
fts-# @@ plainto_tsquery('I am riding');
 ?column?
----------
 t
(1 row)

fts=# SELECT to_tsvector('A nice day for a car ride');
            to_tsvector
-----------------------------------
 'car':6 'day':3 'nice':2 'ride':7
(1 row)

fts=# SELECT plainto_tsquery('I am riding');
 plainto_tsquery
-----------------
 'ride'
(1 row)
```

Text matching example (2)

```
fts=# SELECT to_tsvector('A nice day for a car ride')
fts-# @@ plainto_tsquery('I am riding a bike');
 ?column?
----------
 f
(1 row)

fts=# SELECT to_tsvector('A nice day for a car ride');
            to_tsvector
-----------------------------------
 'car':6 'day':3 'nice':2 'ride':7
(1 row)

fts=# SELECT plainto_tsquery('I am riding a bike');
 plainto_tsquery
-----------------
 'ride' & 'bike'
(1 row)
```

Text matching example (3)

```
fts=# SELECT 'Starman' @@ 'star';
 ?column?
----------
 f
(1 row)

fts=# SELECT 'Starman' @@ to_tsquery('star:*');
 ?column?
----------
 t
(1 row)

fts=# SELECT websearch_to_tsquery('"The Stray Cats" -"cat shelter"');
             websearch_to_tsquery
----------------------------------------------
 'stray' <-> 'cat' & !( 'cat' <-> 'shelter' )
(1 row)
```

2ndQuadrant PostgreSQL

An example table

- pgsql-hackers mailing list archive subset

```
fts=# \d mail_messages
                                        Table "public.mail_messages"
   Column    |            Type             | Collation | Nullable |           
-------------+-----------------------------+-----------+----------+-----------
 id          | integer                     |           | not null | nextval('mai
 parent_id   | integer                     |           |          |
 sent        | timestamp without time zone |           |          |
 subject     | text                        |           |          |
 author      | text                        |           |          |
 body_plain  | text                        |           |          |

fts=# \dt+ mail_messages
                          List of relations
 Schema |     Name      | Type  |  Owner   |  Size  | Description
--------+---------------+-------+----------+--------+-------------
 public | mail_messages | table | postgres | 478 MB |
```

Ranking results

ts_rank (and Cover Density variant ts_rank_cd)

```
fts=# SELECT subject, ts_rank(to_tsvector(coalesce(body_plain,'')),
fts(# to_tsquery('aggregate'), 32) AS rank
fts-# FROM mail_messages ORDER BY rank DESC LIMIT 5;
                           subject                               |    rank
-----------------------------------------------------------------+-------------
 Re: Window functions patch v04 for the September commit fest |  0.08969686
 Re: Window functions patch v04 for the September commit fest |  0.08940695
 Re: [HACKERS] PoC: Grouped base relation                         |  0.08936066
 Re: [HACKERS] PoC: Grouped base relation                         |  0.08931142
 Re: [PERFORM] not using index for select min(...)                |  0.08925897
```

FTS Stats

ts_stat for verifying your TS configuration, identifying stop words

```
fts=# SELECT * FROM ts_stat(
fts(#     'SELECT to_tsvector(body_plain)
fts'#      FROM mail_messages')
fts-# ORDER BY nentry DESC, ndoc DESC, word
fts-# LIMIT 5;
 word  |  ndoc  | nentry
-------+--------+--------
 use   | 173833 | 380951
 wrote | 231174 | 350905
 would | 157169 | 316416
 think | 149858 | 256661
 patch | 100991 | 226099
```

Text indexing

Normal default:

- B-Tree
  - with B-Tree `text_pattern_ops` for left, right anchored text
  - `CREATE INDEX` ***`name`*** `ON` ***`table`*** `(`***`column`*** `varchar_pattern_ops);`

For FTS we have:

- GIN
  - Inverted index: one entry per lexeme
  - Larger, slower to update → Better on less dynamic data
  - On `tsvector` columns
- GiST
  - Lossy index, smaller but slower (to eliminate false positives)
  - Better on fewer unique items
  - On `tsvector` or `tsquery` columns

FTS, unindexed

```
fts=# EXPLAIN ANALYZE SELECT count(*) FROM mail_messages
fts-# WHERE to_tsvector('english',body_plain) @@ to_tsquery('aggregate');
                                                                   QUERY PLAN
-----------------------------------------------------------------------------------
 Finalize Aggregate  (cost=122708.56..122708.57 rows=1 width=8) (actual time=26
   ->  Gather  (cost=122708.34..122708.55 rows=2 width=8) (actual time=26981.64
         Workers Planned: 2
         Workers Launched: 2
         ->  Partial Aggregate  (cost=121708.34..121708.35 rows=1 width=8) (act
               ->  Parallel Seq Scan on mail_messages  (cost=0.00..121706.49 ro
                     Filter: (to_tsvector('english'::regconfig, body_plain) @@
                     Rows Removed by Filter: 116770
 Planning Time: 0.258 ms
 JIT:
   Functions: 14
   Options: Inlining false, Optimization false, Expressions true, Deforming tru
   Timing: Generation 3.243 ms, Inlining 0.000 ms, Optimization 1.534 ms, Emiss
 Execution Time: 26991.805 ms
```

FTS indexing

```
CREATE INDEX ON mail_messages USING GIN
    (to_tsvector('english',
    subject ||' '|| body_plain));
```

- New in PG12: Generated columns (stored):

```
ALTER TABLE mail_messages
ADD COLUMN fts_col tsvector
GENERATED ALWAYS AS (to_tsvector('english',
    coalesce(subject, '') ||' '||
    coalesce(body_plain, ''))) STORED;
```

```
CREATE INDEX ON mail_messages USING GIN (fts_col);
```

FTS, GiST indexed

```
fts=# EXPLAIN ANALYZE SELECT count(*) FROM mail_messages
fts-# WHERE to_tsvector('english',body_plain) @@ to_tsquery('aggregate');
                                                                    QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=7210.61..7210.62 rows=1 width=8) (actual time=5630.167..5630.
   ->  Bitmap Heap Scan on mail_messages  (cost=330.46..7206.16 rows=1781 width
         Recheck Cond: (to_tsvector('english'::regconfig, body_plain) @@ to_tsc
         Rows Removed by Index Recheck: 4267
         Heap Blocks: exact=7883
         ->  Bitmap Index Scan on mail_messages_to_tsvector_idx  (cost=0.00..33
               Index Cond: (to_tsvector('english'::regconfig, body_plain) @@ to
 Planning Time: 0.620 ms
 Execution Time: 5630.249 ms
```

- 26.99 seconds → 5.63 seconds! → ~4.8x faster

FTS, GIN indexed

```
fts=# EXPLAIN ANALYZE SELECT count(*) FROM mail_messages
fts-# WHERE to_tsvector('english',body_plain) @@ to_tsquery('aggregate');
                                                                     QUERY PLAN
----------------------------------------------------------------------------------
 Aggregate  (cost=6873.60..6873.61 rows=1 width=8) (actual time=6.133..6.134 ro
   ->  Bitmap Heap Scan on mail_messages  (cost=33.96..6869.18 rows=1769 width=
         Recheck Cond: (to_tsvector('english'::regconfig, body_plain) @@ to_tsq
         Heap Blocks: exact=4630
         ->  Bitmap Index Scan on mail_messages_to_tsvector_idx  (cost=0.00..33
               Index Cond: (to_tsvector('english'::regconfig, body_plain) @@ to
 Planning Time: 0.433 ms
 Execution Time: 5.684 ms
```

- 26.99 seconds → 5.684 **milli**seconds! → **~4700x faster**

GIN, GiST indexed operations

- GIN
  - tsvector: @@
  - jsonb: ? ?& ?| @> @? @@
- GIST
  - tsvector: @@
  - tsquery: <@ @>

Super useful modules

- pg_trgm
  - Trigram indexing operations
- unaccent
  - Dictionary: removes accents / diacritics
- fuzzystrmatch
  - String similarity: Levenshtein distances (also Soundex, Metaphone, Double Metaphone)
  - `SELECT name FROM users WHERE levenshtein('Stephen', name) <= 2;`

Other index types

- VODKA =)
- RUM
  - https://github.com/postgrespro/rum
  - Lexeme positional information stored
  - Faster ranking
  - Faster phrase search
  - `<=>` Distance between `timestamps`, `floats`, `money`

Free text but not natural?

- One use case: identifying arbitrary strings
  - e.g. keywords in device logs
- Dictionaries not very helpful here
- Arbitrary example: 10M * ~100 char “IoT device” log entries
  - Some contain strings that are significant to user (but we don’t know these keywords)
  - Populate table with random hex codes but 1% of log entries contains a keyword from `/etc/dictionaries-common/words`:

    `c4f2cede5da57f0ace6e669b51186cba`**`excruciating`**`9635d8a26a`
    `efb2b4ee8b9845e89718577b3266f68dffa5ae12ebfebf1a508b21`

Free text but not natural?

```
fts=# SELECT message FROM logentries LIMIT 5 OFFSET 495;
                                                  message
--------------------------------------------------------------------------------------------------
 da40c1006cd75105c1eb8ea70705828d195b264565f047c6d449e51cf99d01e901cf532f03018e793a394fdac9bb5d2a
 aa88a5c43ec8b2a8578d44f924053e842584c0e6b8295b72230f7d19aa3ba2f2b9e1a4bffcf0f82e4d29344645b714ca
 fe9731c39108a74714cad9fc8570b115howlingb9904fa4ad86544fb778ef5edfe362e02a94c66851c3c8d7fe47b26e5
 b68430decf30085cc2e7810585c5d681source2b638d61c5972f25aa3fa5c35aa2be282f04843cfca007689cc6ecdbe3
 5b7ba17108e416d04788dc9ac15121fad7625fa7c216666bf54c1b0ca21ab618829262dfd67a5cd40aefd66235cf9c7f
(5 rows)

fts=# \dt+ logentries
                       List of relations
 Schema |    Name    | Type  |  Owner   |  Size   | Description
--------+------------+-------+----------+---------+-------------
 public | logentries | table | postgres | 1421 MB |
(1 row)

fts=# SELECT * FROM logentries WHERE message LIKE '%source%';
```

2ndQuadrant PostgreSQL

How long?

```
fts=# EXPLAIN ANALYZE SELECT * FROM logentries WHERE message LIKE '%source%';
                                                            QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Gather  (cost=1000.00..235029.95 rows=1000 width=109) (actual time=143.010..9654.769 rows=16 loops=1)
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Seq Scan on logentries  (cost=0.00..233929.95 rows=417 width=109) (actual time=1017.442..
         Filter: (message ~~ '%source%'::text)
         Rows Removed by Filter: 3333594
 Planning Time: 0.220 ms
 JIT:
   Functions: 6
   Options: Inlining false, Optimization false, Expressions true, Deforming true
   Timing: Generation 18.918 ms, Inlining 0.000 ms, Optimization 41.736 ms, Emission 121.955 ms, Total 18
 Execution Time: 9673.582 ms
(12 rows)
```

- 9.6 seconds!

Trigrams

- ***n-gram*** model: probabilistic language model (Markov Chains)
- 3 characters → trigrams
- Similarity of alphanumeric text → number of shared trigrams
- **`CREATE EXTENSION pg_trgm;`**
- `fts=#` **`SELECT show_trgm('source');`**

```
              show_trgm
-------------------------------------
 {"  s"," so","ce ",our,rce,sou,urc}
```

- ```
  fts=# CREATE INDEX ON logentries
  fts-# USING GIN (message gin_trgm_ops);
  ```
  ```

Did trigrams help?

```
fts=# EXPLAIN ANALYZE SELECT * FROM logentries WHERE message LIKE '%source%';
                                                             QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on logentries  (cost=87.75..3870.45 rows=1000 width=109) (actual time=0.152..0.206 rows
   Recheck Cond: (message ~~ '%source%'::text)
   Rows Removed by Index Recheck: 2
   Heap Blocks: exact=18
   ->  Bitmap Index Scan on logentries_message_idx  (cost=0.00..87.50 rows=1000 width=0) (actual time=0.1
         Index Cond: (message ~~ '%source%'::text)
 Planning Time: 0.222 ms
 Execution Time: 0.258 ms
(8 rows)
```

- 0.258 **milli**seconds! → **~37000x faster**
- Also work with regex

This comes at a cost

```
fts=# \di+ logentries_message_idx
                                    List of relations
 Schema |          Name          | Type  |  Owner   |   Table    |  Size   | Description
--------+------------------------+-------+----------+------------+---------+-------------
 public | logentries_message_idx | index | postgres | logentries | 1601 MB |
(1 row)
```

Other neat trigram tricks

- **`similarity(text, text)`** → `real`
- **`text <-> text`** → Distance (1-similarity)
- **`text % text`** → `true` if over `similarity_threshold`
- Supported by indexes:
  - GIN
  - GiST is efficient: k-nearest neighbour (k-NN)

Character set support

- `pg_client_encoding()`
- `convert(string bytea, src_encoding name, dest_encoding name)`
- `convert_from, convert_to`
- Automatic character set conversion
  `SET CLIENT_ENCODING TO '`***`value`***`';`

Collation in PostgreSQL

- Sort order and character classification
  - Per-column: `CREATE TABLE test1 (a text COLLATE "de_DE" …`
  - Per-operation: `SELECT a < b COLLATE "de_DE" FROM test1;`
  - Not restricted by DB `LC_COLLATE, LC_CTYPE`
- New in PG12: Nondeterministic collations (case-insensitive, ignore accents)

Other types of documents → JSON

- Also a real world use case
- JSONB supports indexing

```
(article ->> 'title' ||''||
 article ->> 'author')::tsvector
```

- `jsonb_to_tsvector()`

```
SELECT jsonb_to_tsvector('english', column,
'["numeric","key","string","boolean"]') FROM table;
```

- New in PG12: SQL/JSON (SQL:2016) → jsonpath expressions
- JsQuery: JSONB query language with GIN support
  - Equivalent to `tsquery`, JSON query as a single value
  - https://github.com/postgrespro/jsquery

Finally, maintenance

- `VACUUM ANALYZE`
  - Keep your table statistics up-to-date
  - Pending GIN entries
- `ALTER TABLE SET STATISTICS`
  - Keep your table statistics accurate
    - Number of distinct values
    - Correlated columns
- `EXPLAIN ANALYZE` from time to time
  - Your query works now – but a year from now?
- `maintenance_work_mem`

The curious case of TEXT NAME 🤪

```
CREATE TABLE user (id serial, text name)
```

Type **NAME**

- Sleepy developer 😴
- Internal type for object names, 64 bytes

Thanks! More info:

- **Dictionaries:** https://www.postgresql.org/docs/current/textsearch-dictionaries.html
- **Parsers:** https://www.postgresql.org/docs/current/textsearch-parsers.html
- **Ranking/Weights:** https://www.postgresql.org/docs/current/textsearch-controls.html
- **FTS functions:** https://www.postgresql.org/docs/current/functions-textsearch.html
- **Trigrams:** https://www.postgresql.org/docs/current/pgtrgm.html
- **Collations:** https://www.postgresql.org/docs/current/collation.html