The State of (Full) Text Search in PostgreSQL 12

Jimmy Angelakos
Senior PostgreSQL Architect
Twitter: @vyru
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

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

```sql
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)
An example table

- *pgsql-hackers* mailing list archive subset

```plaintext
fts=# \d mail_messages

Table "public.mail_messages"

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
<th>Collation</th>
<th>Nullable</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>integer</td>
<td></td>
<td>not null</td>
<td>nextval('mail_messages_id_seq'::regclass)</td>
</tr>
<tr>
<td>parent_id</td>
<td>integer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>sent</td>
<td>timestamp without time zone</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>subject</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>author</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>body_plain</td>
<td>text</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

fts=# \dt+ mail_messages

List of relations

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>mail_messages</td>
<td>table</td>
<td>postgres</td>
<td>478 MB</td>
<td></td>
</tr>
</tbody>
</table>
Ranking results

ts_rank (and Cover Density variant ts_rank_cd)

```sql
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;
```

<table>
<thead>
<tr>
<th>subject</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>Re: Window functions patch v04 for the September commit fest</td>
<td>0.08969686</td>
</tr>
<tr>
<td>Re: Window functions patch v04 for the September commit fest</td>
<td>0.08940695</td>
</tr>
<tr>
<td>Re: [HACKERS] PoC: Grouped base relation</td>
<td>0.08936066</td>
</tr>
<tr>
<td>Re: [HACKERS] PoC: Grouped base relation</td>
<td>0.08931142</td>
</tr>
<tr>
<td>Re: [PERFORM] not using index for select min(...)</td>
<td>0.08925897</td>
</tr>
</tbody>
</table>
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;

<table>
<thead>
<tr>
<th>word</th>
<th>ndoc</th>
<th>nentry</th>
</tr>
</thead>
<tbody>
<tr>
<td>use</td>
<td>173833</td>
<td>380951</td>
</tr>
<tr>
<td>wrote</td>
<td>231174</td>
<td>350905</td>
</tr>
<tr>
<td>would</td>
<td>157169</td>
<td>316416</td>
</tr>
<tr>
<td>think</td>
<td>149858</td>
<td>256661</td>
</tr>
<tr>
<td>patch</td>
<td>100991</td>
<td>226099</td>
</tr>
</tbody>
</table>
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

```sql
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=26983.786..26983.786 rows=1 loops=1)
  ->  Gather  (cost=122708.34..122708.55 rows=2 width=8) (actual time=26981.649..26989.399 rows=3 loops=1)
        Workers Planned: 2
        Workers Launched: 2
  ->  Partial Aggregate  (cost=121708.34..121708.35 rows=1 width=8) (actual time=26967.335..26967.335 rows=1 loops=3)
            ->  Parallel Seq Scan on mail_messages  (cost=0.00..121706.49 rows=742 width=0)
                  Filter: (to_tsvector('english'::regconfig, body_plain) @@ to_tsquery('aggregate'::text))
                  Rows Removed by Filter: 116770

Planning Time: 0.258 ms
JIT:
  Functions: 14
  Options: Inlining false, Optimization false, Expressions true, Deforming true
  Timing: Generation 3.243 ms, Inlining 0.000 ms, Optimization 1.534 ms, Emission 13.796 ms
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

```sql
fts=# EXPLAIN ANALYZE SELECT count(*) FROM mail_messages
fts-# WHERE to_tsvector('english',body_plain) @@ to_tsquery('aggregate');
```

```
Aggregate (cost=7210.61..7210.62 rows=1 width=8) (actual time=5630.167..5630.167 rows=1 loops=1)
  ->  Bitmap Heap Scan on mail_messages (cost=330.46..7206.16 rows=1781 width=0) (actual time=32.884..5629.594 rows=5814 loops=1)
    Recheck Cond: (to_tsvector('english'::regconfig, body_plain) @@ to_tsquery('aggregate'::text))
    Rows Removed by Index Recheck: 4267
    Heap Blocks: exact=7883
  ->  Bitmap Index Scan on mail_messages_to_tsvector_idx (cost=0.00..330.02 rows=1781 width=0) (actual time=31.267..31.267 rows=10081 loops=1)
    Index Cond: (to_tsvector('english'::regconfig, body_plain) @@ to_tsquery('aggregate'::text))

Planning Time: 0.620 ms
Execution Time: 5630.249 ms

• 26.99 seconds → 5.63 seconds! → ~4.8x faster
FTS, GIN indexed

```sql
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 rows=1 loops=1)
  -> Bitmap Heap Scan on mail_messages (cost=33.96..6869.18 rows=1769 width=0)
    Recheck Cond: (to_tsvector('english'::regconfig, body_plain) @@ to_tsquery('aggregate'))
    Heap Blocks: exact=4630
  -> Bitmap Index Scan on mail_messages_to_tsvector_idx (cost=0.00..33.52 rows=1769 width=0)
    Index Cond: (to_tsvector('english'::regconfig, body_plain) @@ to_tsquery('aggregate'))
Planning Time: 0.433 ms
Execution Time: 5.684 ms
```

- 26.99 seconds → 5.684 milliseconds! → ~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:
    - c4f2cede5da57f0ace6e669b51186cbaexcruciating9635d8a26aefb2b4ee8b9845e89718577b3266f68dfffa5ae12ebfebf1a508b21
Free text but not natural?

```sql
fts=# SELECT message FROM logentries LIMIT 5 OFFSET 495;
message
------------------------------------------------------------------
da40c1006cd75105c1eb8ea70705828d195b264565f047c6d449e51cf99d01e901cf532f03018e793a394fdac9bb5d2a
aa88a5c43ec8b2a8578d44f924053e842584c0e6b8295b72230f7d19aa3ba2f2b9e1a4bffcf0f82e4d29344645b714ca
fe9731c39108a74714cad9fc8570b115b9904fa4ad86544fb778ef5edfe362e02a94c66851c3c8d7fe47b26e5
b68430decf30085cc2e7810585c5d681source2b638d61c5972f25aa3fa5c35aa2be282f04843cfca007689cc6ecdbe3
5b7ba17108e416d04788dc9ac15121fad7625fa7c216666bf54c1b0ca21ab618829262dfd67a5cd40aefd66235cf9c7f
(5 rows)
```

```sql
fts=# \dt+ logentries
List of relations
<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>logentries</td>
<td>table</td>
<td>postgres</td>
<td>1421 MB</td>
<td></td>
</tr>
</tbody>
</table>
(1 row)
```

```sql
fts=# SELECT * FROM logentries WHERE message LIKE '%source%';
```
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..9547.847 rows=5 loops=3)
       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 182.608 ms
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?

```sql
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=16 loops=1)
  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=0.129..0.129 rows=18 loops=1)
    Index Cond: (message ~~ '%source%':text)
Planning Time: 0.222 ms
Execution Time: 0.258 ms
(8 rows)
```

- **0.258 milliseconds! → ~37000x faster**
- Also work with regex
This comes at a cost

fts=# \di+ logentries_message_idx

<table>
<thead>
<tr>
<th>Schema</th>
<th>Name</th>
<th>Type</th>
<th>Owner</th>
<th>Table</th>
<th>Size</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public</td>
<td>logentries_message_idx</td>
<td>index</td>
<td>postgres</td>
<td>logentries</td>
<td>1601 MB</td>
<td></td>
</tr>
</tbody>
</table>

(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()`
  ```sql
  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](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