Seek and ye shall find: full-text search in PostgreSQL

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME
	September 18, 2012		DS(

Contents

1 It happened to me

- Summer project created large amounts of text from scanned newspapers
- Wanted an independent & useful search engine
- Lucene-based search engines like Solr and ElasticSearch are not integrated with the database
- I'm a library geek *and* a database geek and, since 2006, have been hacking on the Evergreen library system, which uses PostgreSQL full-text search
- Housekeeping: slides are at http://stuff.coffeecode.net/2012/pgopen_fulltext

2 Simplistic (straw-man) approach

I'll just use LIKE / ILIKE / regex matching...

```
CREATE SCHEMA lunews;
CREATE TABLE lunews.issues (
  id TEXT, text_uri TEXT, volume TEXT, issue TEXT, content TEXT
);
COPY lunews.issues (id, text_uri, volume, issue, content)
  FROM 'raw_lambda_text.tsv';

SELECT id FROM lunews.issues WHERE content LIKE '%picket%';
SELECT id FROM lunews.issues WHERE content ILIKE '%picket%';
SELECT id FROM lunews.issues WHERE content ~ E'p[iao]cket';
```

3 ... with simple problems

- Fast for test data (50 rows with ~10,000 words per row) but does not scale to real world data
  - Table scans are *not* our friends
  - Index expressions are wonderful but can't anticipate the wide variety of human questions
- We could build something based on `pg_trgm` extension - but that's work!
- Full-text search is built into PostgreSQL, so let's use it


(Thank you, Oleg Bartunov and Teodor Sigaev!)

4 2-minute demo - 50 rows

1. As a proof of concept, create and populate our table containing the full text of 50 newspaper issues.
2. Create a `tsvector` index on the text column of interest (*content*):

```
CREATE INDEX ON lunews.issues USING GIN (to_tsvector(content));
```

3. Run some search queries:

```
SELECT id FROM lunews.issues
  WHERE to_tsvector(content)
    @@ to_tsquery('strike & picket');

SELECT id, ts_rank_cd(to_tsvector(content),
    to_tsquery('picket | pocket | packet'))
  FROM lunews.issues
  WHERE to_tsvector(content)
    @@ to_tsquery('picket | pocket | packet')
  ORDER BY 2 DESC
;
```

5 5-minute demo - 1 million rows

1. Create and populate a table containing the titles and subjects of 1 ***MILLION*** books from OpenLibrary.
2. Create weighted and unweighted `tsvector` keyword columns.

```
CREATE TABLE openlib (id INT, title TEXT, subjects TEXT,
  unweighted_kw TSVECTOR, weighted_kw TSVECTOR);
COPY openlib (id, title, subjects) FROM STDIN;     -- ❶ ❷
UPDATE openlib SET unweighted_kw =
  to_tsvector('english', title) ||
  to_tsvector('english', COALESCE(subjects, ''));  -- ❸
UPDATE openlib SET weighted_kw =
  setweight(to_tsvector('english', title), 'A') ||
  setweight(to_tsvector('english',
    COALESCE(subjects, '')), 'D');                 -- ❹
```

❶ 2012-09-15 12:33:05-04 - started loading

❷ 2012-09-15 12:33:10-04 - finished loading

❸ 2012-09-15 12:33:51-04 - populated unweighted_kw

❹ 2012-09-15 12:34:40-04 - populated weighted_kw

6 5-minute demo - 1 million rows (cont.)

1. Create indexes on the weighted and unweighted keyword columns.
2. Create separate expression indexes on the title and subjects columns.

```
CREATE INDEX openlib_title ON openlib
  USING GIN (to_tsvector('english', title));       -- ❶
CREATE INDEX openlib_subjects ON openlib
  USING GIN (to_tsvector('english', subjects));    -- ❷
CREATE INDEX openlib_unweighted ON openlib
  USING GIN (unweighted_kw);                       -- ❸
CREATE INDEX openlib_weighted ON openlib
  USING GIN (weighted_kw);                         -- ❹
```

❶ 2012-09-15 12:35:13-04 - created openlib_title index

❷ 2012-09-15 12:35:39-04 - created openlib_subjects index

❸ 2012-09-15 12:36:04-04 - created openlib_unweighted index

❹ 2012-09-15 12:36:30-04 - created openlib_weighted index

7 5-minute demo - 1 million rows - searching

With an unweighted search:

```
SELECT id, SUBSTRING(title FROM 0 FOR 20),
    ts_rank_cd(unweighted_kw,
      to_tsquery('government & history'))
  FROM openlib
  WHERE unweighted_kw @@ to_tsquery('government & history')
  ORDER BY 3 DESC LIMIT 5;

    id    |      substring      |   rank
----------+---------------------+----------
  6836768 | Pietre e potere     |     0.35
```

```
 17995790 | Reform and insurrec | 0.236111
  8669133 | American Political  | 0.229545
   866182 | Lives in the public |    0.225
 16640636 | He:> katastatike<::> no | 0.214286(5 rows) Total runtime:84.942 ms
```

8 5-minute demo - 1 million rows - searching

With a weighted search:

```
SELECT id, SUBSTRING(title FROM 0 FOR 20),
    ts_rank_cd(weighted_kw,
      to_tsquery('government & history'))
  FROM openlib
  WHERE weighted_kw @@ to_tsquery('government & history')
  ORDER BY 3 DESC LIMIT 5;

    id    |      substring      |   rank
----------+---------------------+----------
 11331254 | Global Issues: Hist |  2.09091
  5284994 | Michigan: a history | 0.615909
  5857361 | California history  | 0.590909
 23324185 | Outlines of the his | 0.525974
 14832714 | History of federal  | 0.508333
(5 rows) Total runtime: 86.443 ms
```

9 5-minute demo - 1 million rows - searching

With a weighted search and relevancy algorithm tweak:

```
SELECT id, SUBSTRING(title FROM 0 FOR 20),
    ts_rank_cd(weighted_kw,
      to_tsquery('government & history'), 4)
  FROM openlib
  WHERE weighted_kw @@ to_tsquery('government & history')
  ORDER BY 3 DESC LIMIT 5;

    id    |      substring      | ts_rank_cd
----------+---------------------+------------
 20696993 | A study of general  |        0.5
 11830422 | 21st Century Comple |        0.5
 16606467 | History of the gove |   0.333333
  9930696 | Judiciary and Respo |   0.333333
 13681611 | history of local go |   0.333333
(5 rows) Total runtime: 92.518 ms
```

10 Parsing process

1. Start with a *document* - the text to be searched; can be a single word or an entire book
2. Parse the document into *tokens* - with types like words, numbers, paths, email addresses, white space
3. Normalize the tokens to create *lexemes* - including case-folding, stemming, and using dictionaries and thesauruses to unify different forms of a token

10.1 Text search data types

- *tsvector* - text search vector (array-ish thing) containing lexemes with positional information to inform relevance ranking
- *tsquery* - a preprocessed text search query

11 Show me the tokens!

```
SELECT alias, description FROM ts_token_type('default');
-----------------+------------------------------------------
 asciiword       | Word, all ASCII
 word            | Word, all letters
 numword         | Word, letters and digits
 email           | Email address
 url             | URL
 host            | Host
 sfloat          | Scientific notation
 version         | Version number
 hword_numpart   | Hyphenated word part, letters and digits
 hword_part      | Hyphenated word part, all letters
 hword_asciipart | Hyphenated word part, all ASCII
 blank           | Space symbols
 tag             | XML tag
 protocol        | Protocol head
 numhword        | Hyphenated word, letters and digits
 asciihword      | Hyphenated word, all ASCII
 hword           | Hyphenated word, all letters
 url_path        | URL path
 file            | File or path name
 float           | Decimal notation
 int             | Signed integer
 uint            | Unsigned integer
 entity          | XML entity
(23 rows)
```

12 Let's see some lexemes

- Normalized tokens, remember?

```
SELECT to_tsvector('english',
   'Kurt Vonnegut, Breakfast of Champions, 1973'
);
 '1973':6 'breakfast':3 'champion':5 'kurt':1 'vonnegut':2

SELECT to_tsvector('english', 'Les nuits en Paris');
 'en':3 'les':1 'nuit':2 'pari':4

SELECT to_tsvector('french', 'Les nuits en Paris'); -- ❶
 'le':1 'nuit':2 'paris':4
```

❶ Dig that multilingual support, eh?

13 Operators and functions

- `to_tsvector()` - turn a document into a vector
- `to_tsquery()` - normalize the terms in a query to return a TSQUERY
- `plainto_tsquery()` - normalize the terms in a query to return a TSQUERY, automatically ANDing the terms
- `@@` - match operator for a tsvector and a tsquery
- `ts_debug()` - display how a string gets normalized by a given text search configuration

14 Let's search!

- Boolean operators
  - `&` - AND
  - `|` - OR
  - `!` - NOT (tricksy; negation operator requires `&` or `|`)
- Wildcards for matching prefixes: `prefix:*`

```
SELECT id FROM lunews.issues
  WHERE to_tsvector(content) @@ plainto_tsquery('student strike');
SELECT id FROM lunews.issues
  WHERE to_tsvector(content) @@ to_tsquery('strike &! student');
SELECT id FROM lunews.issues
  WHERE to_tsvector(content) @@ to_tsquery('prof:* | fac:*');
SELECT id FROM lunews.issues
  WHERE to_tsvector(content) @@ to_tsquery('faculty & strike')
    AND content ILIKE '%faculty strike%';
```

15 What is this "normalization"?

- **Parsers**: tokenize and classify the text of a document (word, number, path…) - in practice, there is only one
- **Dictionaries**: convert tokens into normalized form, strip out stop words, unify synonyms
- **Templates**: provide the dictionaries' underlying functions
- **Configurations**: which dictionaries to use for each token supplied by the parser

```
SELECT to_tsvector('english', 'stochastically');      -- ❶
 'stochast':1

SELECT to_tsvector('english', 'To be or not to be'); -- ❷

SELECT to_tsvector('english', 'laptopping');          -- ❸
 'laptop':1
```

❶ The English Porter stemming algorithm is thorough :)

❷ Stopwords can eat up a lot of space if you decide to index them.

❸ The stemmer makes no guarantees that the original words were real!

16 What are we matching?

`ts_headline()` returns our `tsquery` in context:

```
SELECT ts_headline(title,
    to_tsquery('english', 'chicago & theater'),
    'MaxWords = 10, MinWords = 5')
  FROM openlib
  WHERE to_tsvector('english', title)
    @@ to_tsquery('english', 'chicago & theater')
  LIMIT 5;
                   ts_headline
=-------------------------------------------------
 Rice's <b>Theater</b>, <b>Chicago</b>, 1851-1857
(1 row) Total runtime: 0.616 ms
```

17 Relevancy

- *Image of match*

We don't just want matches; we want the best matches. PostgreSQL offers two built-in ranking functions:

- `ts_rank` ([*weights* `float4[]`,] *vector* `tsvector`, *query* `tsquery` [, *normalization* `integer`]) - frequency of matching lexemes
- `ts_rank_cd` ([*weights* `float4[]`,] *vector* `tsvector`, *query* `tsquery` [, *normalization* `integer`]) - cover density algorithm that incorporates term proximity

Relevancy scoring can be adjusted by passing a bit mask of options, to adjust rank for long documents and the number of unique words.

> COVER SETS
>
> (1) the shorter the cover, the more likely the corresponding text is relevant; and (2) the more covers contained in a document, the more likely the document is relevant.
>
> — Clarke et al *Relevance Ranking for One to Three Term Queries (1999)*

18 Beyond searching a single column

- An entire newspaper in a column doesn't allow for much precision
- A story in a column is better, but still not optimal
- Separate columns for title, author, story? Now we're talking
  - Concatenate the `tsvector` documents for maximum flexibility

19 Beyond searching a single column (cont.)

```
CREATE TABLE lunews.stories (
  id SERIAL, title TEXT, author TEXT, story TEXT
);

WITH ft AS (
  SELECT id, title || author || story AS keywords,
    to_tsvector(title) || to_tsvector(author) ||
    to_tsvector(story) AS ts_kw
  FROM lunews.stories
),
q AS (
  SELECT id, to_tsquery('exotic & horizons') AS tsq
  FROM lunews.stories
)
SELECT ft.id, ts_rank_cd(ft.ts_kw, q.tsq) AS score,
    ts_headline(ft.keywords, q.tsq) AS kic
  FROM ft INNER JOIN q ON q.id = ft.id
  WHERE q.tsq @@ ft.ts_kw
;
```

20 Weighting

- Terms can be weighted using four scores: *A*, *B*, *C*, and *D*
- Weight sets of terms at index time using the `setweight()` function
- Adjust weight values at query time by passing an array of four `float` values as the first argument to `ts_rank()`/`ts_rank_cd()`
  - Example: You can set the title to *A* and subject to *D* for a default boost to title occurrences, but adjust the weights for a subject-focused search

```
WITH ft AS (
  SELECT id, title || author || story AS keywords,
    setweight(to_tsvector(COALESCE(title, '')), 'A')
    || setweight(to_tsvector(COALESCE(author, '')), 'B')
    || setweight(to_tsvector(COALESCE(story, '')),'D')
   AS ts_kw
  FROM lunews.stories
)
SELECT ts_rank_cd('{0.1, 0.2, 0.4, 1.0}', ft.ts_kw, q.tsq) -- ❶
```

❶ Small gotcha: floats in the score array represent *D*, *C*, *B*, *A*

21 Custom search configuration

Perhaps you actually want to index stopwords (to find *To be or not to be* or *It*)

> denials, Isn't there an easier way to allow stop word, like just set a flag in some config file to True or False?
>
> — #postgresql on Freenode

```
CREATE TEXT SEARCH DICTIONARY tsd_english_full(
  template = snowball, language = english
);
CREATE TEXT SEARCH CONFIGURATION tsc_english_full(
```

```
  COPY = pg_catalog.english
);
ALTER TEXT SEARCH CONFIGURATION tsc_english_full
  ALTER MAPPING FOR asciiword, asciihword, hword_asciipart,
    hword, hword_part, word
  WITH tsd_english_full;

SELECT to_tsvector('tsc_english_full', 'to be or not to be');
 'be':2,6 'not':4 'or':3 'to':1,5
```

22 Accelerate

- GIN and GiST indexes support full-text search queries
  - Usual trade-off of better performance / slower build for GIN
- One index per column of interest, plus one general index for weighted aggregate searches, per text search configuration
- Create `tsvector` columns with an associated trigger to maintain their contents
  - `tsvector_update_trigger()` function automatically updates the `tsvector` column with the contents of one of more columns, for a single text search configuration
  - `tsvector_update_trigger_column()` function uses whatever text search configuration appears in a specified column on a per-row basis, so that you can better support multilingual data in a single table

23 Accelerate (example)

Use the built-in `tsvector_update_trigger()` function to maintain a `tsvector` column, with a corresponding GIN index:

```
CREATE TABLE openlib_subset(title TEXT, tsv TSVECTOR);
CREATE TRIGGER tsv_update
  BEFORE INSERT OR UPDATE ON openlib_subset FOR EACH ROW
  EXECUTE PROCEDURE
    tsvector_update_trigger(tsv, 'pg_catalog.english', title);

CREATE INDEX idx_title ON openlib_subset USING GIN (tsv);

INSERT INTO openlib_subset (title)
  SELECT title FROM openlib ORDER BY id DESC LIMIT 10000;

SELECT title, tsv FROM openlib_subset
  WHERE to_tsquery('Chicago') @@ tsv
    AND LENGTH(title) < 30 LIMIT 5;
            title             |                 tsv
------------------------------+--------------------------------------
 The Chicago address          | 'address':3 'chicago':2
 You were never in Chicago    | 'chicago':5 'never':3
 African Americans in Chicago | 'african':1 'american':2 'chicago':4
 Chicago                      | 'chicago':1
```

24 Preparsing

- Changing the parser to emit more / different tokens requires source code changes

- Hack: change the text before you index & query it

Desired goal: index `"term1/term2"` as a single term, *and* as separate terms

```
SELECT alias, token, lexemes FROM ts_debug('english','term1/term2');
 alias |    token    |    lexemes
-------+-------------+---------------
 file  | term1/term2 | {term1/term2}

SELECT alias, token, lexemes FROM ts_debug('english',
  regexp_replace('term1/term2',
    E'(\\S+)/(\\S+)', E'\\1/\\2 \\1 \\2')
);
  alias   |    token    |    lexemes
----------+-------------+---------------
 file     | term1/term2 | {term1/term2}
 blank    |             |
 numword  | term1       | {term1}
 blank    |             |
 numword  | term2       | {term2}
```

25 Synonyms via a dictionary

Problem: default *english* configuration does not recognize American variations as the equivalent of Canadian words

- Examples: *color* vs. *colour*, *favorite* vs. *favourite*

Solution: The *synonym* template lets you define a list of words and their synonyms, including prefix matches if applicable.

canuck.syn synonym file

```
colour color*
colours color*
```

```
CREATE TEXT SEARCH DICTIONARY canuck (
  TEMPLATE = synonym, SYNONYMS = canuck            -- ❶
);
ALTER TEXT SEARCH CONFIGURATION tsc_english_full
  ALTER MAPPING FOR asciiword WITH canuck, tsd_english_full;
SELECT ts_lexize('canuck', 'colours');
 {color}
```

❶ Do not specify the suffix; PostgreSQL adds `.syn` for you.

26 Spell-checking suggestions

- You can use the **pg_trgm** extension to calculate similarity with existing words in your corpus.

```
CREATE EXTENSION pg_trgm;
CREATE TABLE lunews.issues_words AS (
  SELECT word FROM ts_stat(                  -- ❶
    'SELECT to_tsvector(''simple'', content) -- ❷
    FROM lunews.issues'
  )
);
```

```
CREATE INDEX lunews_issues_content_idx
  ON lunews.issues_words
  USING GiST (word gist_trgm_ops);          -- ❸
```

❶ `ts_stat()` returns statistics about each distinct lexeme in the `tsvector`, such as frequency. Here we just want the distinct lexemes.

❷ We use the `simple` configuration because we do not want stemming to kick in when we process the lexemes.

❸ Use the `gin_trgm_ops` or `gist_trgrm_ops` operator class if you create a GIN or GiST index, respectively.

27 Spell-checking suggestions: query

The <→ operator calculates the distance between terms on either side of the query.

```
SELECT word, word <-> 'picket' AS dist
  FROM lunews.issues_words
  ORDER BY dist
  LIMIT 5;
    word     |   dist
-------------+----------
 picket      |        0
 pickets     | 0.333333
 picketer    |      0.4
 picketed    |      0.4
 picket-line | 0.416667
(5 rows)
```

28 Resources

- PostgreSQL Global Development Group. Full Text Search: Introduction. *PostgreSQL 9.2.0 Documentation*. 2012.
- Clarke et al. Relevance Ranking for One to Three Term Queries. *Information Processing and Management*, 36(2):291-311, 2000.
- Porter, M.F. Snowball: A language for stemming algorithms. 2001