Fuzzy Matching In PostgreSQL

A Story From The Trenches

Charles Clavadetscher

Swiss PostgreSQL Users Group

Swiss PGDay 2016, Rapperswil, 24.06.2016

Outline

Charles Clavadetscher

- Senior DB Engineer at KOF ETH Zurich
  - KOF is the Center of Economic Research of the
  - ETHZ the Swiss Institute of Technology in Zurich, Switzerland
  - Independent economic research on business cycle tendencies for almost all sectors
  - Maintenance of all databases at KOF: PostgreSQL, Oracle, MySQL and MSSQL Server. Focus on migrating to PostgreSQL
  - Support in business process re-engineering
- Co-founder and treasurer of the SwissPUG, the Swiss PostgreSQL Users Group
- Member of the OK of the Swiss PGDay

Principles of searches 1/2

Searching in a collection of texts written in a natural language is a huge challenge.

- **Semantics and lexicology**
  - Meaning (FTS [1])
  - Synonyms (FTS)
  - Lexemes: Naked word base meanings
- **Morphology**
  - Phonetic (soundex, metaphone, double metaphone)
  - Morphems: Naked word base forms (FTS)
- **Language independent**
  - Words (=, LIKE, ILIKE, regexp)
  - Letters (levenshtein)
  - N-gram subsets (trigrams)

1. FTS: Full Text Search

Principles of searches 2/2

Methods can be classified based on many criteria. Here is short summary that includes the most important differentiators for our use case:

- **Language dependency**: Switzerland is a country with 4 official languages.
- **Exact/Fuzzy**: Use exact when possible and fuzzy otherwise.
- **Indexability**: Tradeoff between matching probability and execution speed.

Language dependent	Exact		Fuzzy	
	not indexable	indexable	not indexable	indexable
NO	LIKE ILIKE Regexp	= (equality) LIKE (Trigrams) ILIKE (Trigrams) Regexp (Trigrams)	Levenshtein	Trigrams
YES			Soundex Metaphone Double Metaphone	Full Text Search

Outline

Classics

- **=**: Exact match. This can easily be indexed using mostly btree.

```
db=# SELECT * FROM people WHERE people_id = 1765;
```

- **LIKE**: Partially exact match. The part that is given must match exactly, all the rest is irrelevant. In combination with trigrams (and some other operator classes if the initial part is fixed) you can use indexes.
- **SIMILAR TO**: Equivalent to LIKE.
- **ILIKE**: The same as LIKE but case insensitive.

```
db=# SELECT * FROM people WHERE firstname ILIKE 'Johann%';
db=# SELECT * FROM people WHERE firstname ILIKE '%johann%';
```

- **Regexp**: With regular expressions you can search for a specific pattern. PostgreSQL has operators for regexp matching case sensitive and insensitive. As with LIKE/ILIKE you can build an index on the base field using trigrams.

```
db=# SELECT * FROM people WHERE lastname ~ '^[CB][a-z]*(ts|tsch)er$';
db=# SELECT * FROM companies WHERE name ~ '.*[0-9]+.*';
```

Outline

Definitions and Extensions

Definitions

- **Fuzzy Search**: A process that locates entries that are likely to be relevant to a one or more search parameters even if the argument does not exactly correspond to the specific information available.
- **Distance/Difference**: A metric indicating how far are two strings apart.
- **Similarity**: A metric indicating how similar are two strings.
- **Normalization**: A technique to compute the relative strength of a metric (usually a number between 0 and 1)

Extensions

- **fuzzystrmatch**: Soundex, Metaphone, Double Metaphone, Levenshtein.
- **pg_trgm**: Trigrams.

Phonetic Methods

Soundex

```
db=# select soundex('halloween'), soundex('halouin'),
db-#        soundex('les mots'), soundex('lemó');
 soundex | soundex | soundex | soundex
---------+---------+---------+---------
 H450    | H450    | L253    | L50
```

Metaphone

```
db=# select metaphone('halloween',5), metaphone('halouin',5),
db-#        metaphone('les mots',5), metaphone('lemó',5);
 metaphone | metaphone | metaphone | metaphone
-----------+-----------+-----------+-----------
 HLWN      | HLN       | LSMTS     | LM
```

Double metaphone

```
db=# select dmetaphone('halloween'), dmetaphone('halouin'),
db-#        dmetaphone('les mots'), dmetaphone('lemó');
 dmetaphone | dmetaphone | dmetaphone | dmetaphone
------------+------------+------------+------------
 HLN        | HLN        | LSMT       | LM
```

Levenshtein Distance

The minumum number of operations (delete, insert, substitute) required to transform a string into another. Case sensitive.

```
db=# select levenshtein('Zürich','zuerich');
 levenshtein
-------------
           3
```

Normalized distance:

$$d_{norm}(a, b) = 1 - \frac{d(a, b)}{max(length(a), length(b))}$$

```
db=# SELECT 1 - (levenshtein('Zürich','zuerich')::REAL /
db-#             greatest(length('Zürich'),length('zuerich'))::REAL) AS levenshtein_norm;

 levenshtein_norm
-------------------
 0.571428567171097
```

Full Text search - 1/8

Full Text Search is a huge chapter in PostgreSQL and would deserve a dedicated session. In this context we can only give an overview of the main characteristics of the technique.

Lexeme: A basic lexical unit of a language consisting of one word or several words, the elements of which do not separately convey the meaning of the whole.

tsvector: The PostgreSQL equivalent of a collection of lexeme.

to_tsvector: Function to convert a text into a collection of lexemes.

to_tsquery: A collection of lexemes combined with boolean operators (`&`, `|`, `!`).

```
db=> SELECT to_tsvector('english','There are many nice stories about creating lexemes');
                    to_tsvector
-----------------------------------------------
 'creat':7 'lexem':8 'mani':3 'nice':4 'stori':5

db=> SELECT to_tsquery('english','People & talk & stories & lexemes');
               to_tsquery
-----------------------------------
 'peopl' & 'talk' & 'stori' & 'lexem'
```

Full Text search - 2/8

The match operator @@

```
db=> SELECT to_tsquery('english','People & talk & stories & lexemes') @@
db-> to_tsvector('english','There are many nice stories about creating lexemes');
 ?column?
----------
 f

db=> SELECT to_tsquery('english','(People & talk) | stories | lexemes') @@
db-> to_tsvector('english','There are many nice stories about creating lexemes');
 ?column?
----------
 t
```

Full Text search - 3/8

From the PostgreSQL documentation:

- Text search parsers break documents into tokens and classify each token (for example, as words or numbers).
- Text search dictionaries convert tokens to normalized form and reject stop words.
- Text search templates provide the functions underlying dictionaries. (A dictionary simply specifies a template and a set of parameters for the template.)
- Text search configurations select a parser and a set of dictionaries to use to normalize the tokens produced by the parser.

Full Text search - 4/8

Indexing

```
db=# SELECT count(*) FROM appln_abstr;
  count
----------
 33870394

db=# CREATE INDEX appln_abstract_idx ON appln_abstr
db-# USING gin(to_tsvector('english', appln_abstract));
CREATE INDEX

db=# \d appln_abstr
              Table "public.appln_abstr"
     Column     |  Type   |         Modifiers
----------------+---------+---------------------------
 appln_id       | integer | not null default 0
 appln_abstract | text    | not null default ''::text
Indexes:
    "appln_abstract_idx" gin (to_tsvector('english'::regconfig, appln_abstract))

db=# SELECT pg_size_pretty(pg_relation_size('appln_abstr'::regclass));
 pg_size_pretty
----------------
 54 GB
```

Full Text search - 5/8

Indexing

```
db=# EXPLAIN SELECT * FROM appln_abstr WHERE to_tsvector('english',appln_abstract) @@
db-# to_tsquery('english','cryptographic & process & hardware');
                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------------
 Bitmap Heap Scan on appln_abstr  (cost=76.72..445.33 rows=92 width=848)
                                  (actual time=528.567..528.815 rows=184 loops=1)
   Recheck Cond: (to_tsvector('english'::regconfig, appln_abstract) @@
                  to_tsquery('cryptographic & process & hardware'::text))
   ->  Bitmap Index Scan on appln_abstract_idx  (cost=0.00..76.69 rows=92 width=0)
                                                (actual time=528.546..528.546 rows=184 loops=1)
         Index Cond: (to_tsvector('english'::regconfig, appln_abstract) @@
                      to_tsquery('cryptographic & process & hardware'::text))
 Total runtime: 528.847 ms
```

The query needs 0.5 seconds to find the 184 matching rows out of 33.8 millions rows.

Full Text search - 6/8

Ranking

```
db=# select appln_id, ts_rank(to_tsvector('english',appln_abstract),
db-#                          to_tsquery('english','cryptographic & process & hardware'))
db-# from appln_abstr
db-# where to_tsvector('english',appln_abstract) @@
db-#       to_tsquery('english','cryptographic & process & hardware')
db-# order by ts_rank(to_tsvector('english',appln_abstract),
db-# to_tsquery('english','cryptographic & process & hardware')) desc
db-# limit 5 ;
 appln_id  | ts_rank
-----------+----------
  39552264 | 0.825291
  16385889 | 0.776024
  39591453 | 0.771596
 273993893 | 0.771596
  49903135 | 0.729577
```

Full Text search - 7/8

Highlighting

```
db=# SELECT appln_abstract, ts_headline('english',appln_abstract,
db-#                        to_tsquery('english','cryptographic & process & hardware'))
db-# FROM appln_abstr WHERE appln_id = 39552264 ;

-[ RECORD 1 ]--+-------------------------------------------------------------------
appln_abstract | <P>PROBLEM TO BE SOLVED: To provide effective cryptographic
               | processing solutions without incurring additional hardware costs.
               | <P>SOLUTION: A graphics processing unit is programmed to carry
               | out cryptographic processing so that fast, effective cryptographic
               | processing solutions can be provided without incurring additional
               | hardware costs. The graphics processing unit can efficiently carry
               | out cryptographic processing because it has an architecture that is
               | [...]
ts_headline    | <b>cryptographic</b> <b>processing</b> solutions without incurring
               | additional <b>hardware</b> costs. <P>SOLUTION: A graphics
               | <b>processing</b> unit is programmed
```

Full Text search - 8/8

Coming up (9.6+): RUM Indexing for FTS

- Still in development !
- Presented at PGCon Ottawa on April 20th, 2016
- Oleg Bartunov & Teodor Sigaev
- RUM adds to the PGBar GIN & VODKA
- Take advantage of position information to speed up ranking
- Add `<->` distance operator for ranking
- Add `<n>` phrase operator for ts_query to honour order of words

Lively discussion currently ongoing and development being continued. Information can be found at:

- http://www.sai.msu.su/ megera/postgres/talks/pgcon-2016-fts.pdf
- pgsql-hackers@postgresql.org mailing list

Trigrams

From the PostgreSQL documentation

A trigram is a group of three consecutive characters taken from a string. In order to create the set of trigrams the algorithm ignores non alphanumeric characters, strips reduntant spaces, prefixes the string with 2 spaces and appends one at the end. Trigrams that would contain a space between two characters are skipped. We can measure the similarity of two strings by counting the number of trigrams they share. This simple idea turns out to be very effective for measuring the similarity of words in **many** natural languages.

```
db=# select show_trgm('La Habana');
                        show_trgm
-----------------------------------------------------
 {"  h","  l"," ha"," la",aba,ana,ban,hab,"la ","na "}

db=# select show_trgm('     La     Habana     ');
                        show_trgm
-----------------------------------------------------
 {"  h","  l"," ha"," la",aba,ana,ban,hab,"la ","na "}

db=# select show_trgm(' La ; Habana ...');
                        show_trgm
-----------------------------------------------------
 {"  h","  l"," ha"," la",aba,ana,ban,hab,"la ","na "}
```

Trigrams Operators and Functions

Operators

- **%**: The level of similarity is above the value set with **set_limit(real)**.
- **<->**: The distance between two strings. This is the same as 1 - similarity(text,text).

Functions

- **similarity(text, text)**: Displays the level of similarity between the two text parameters.
- **show_trgm(text)**: Displays the trigram generated from text.
- **show_limit()**: Displays the current value of the minimal similarity required for the **%** operator. The default value is 0.3.
- **set_limit(real)**: Sets the value of the minimal similarity required for the **%** operator.

Trigrams Examples

Examples

```
db=# SELECT show_limit();
show_limit | 0.3

db=# SELECT similarity('Zürich','zuerich');
similarity | 0.363636

db=# SELECT 'Zürich' % 'zuerich' as matches;
matches    | t

db=# SELECT set_limit(0.4);
set_limit  | 0.4

db=# SELECT 'Zürich' % 'zuerich' as matches;
matches    | f

db=# SELECT 'Zürich' <-> 'zuerich' as distance;
distance   | 0.636364
```

Trigrams Indexing 1/2

The **pg_trgm** extension has operator classes for indexing with GIN and GiST. Use GiST if you have very dynamic data and GIN otherwise.

```
db=# CREATE INDEX trgm_gin_idx ON companies USING gin (company_name gin_trgm_ops);
CREATE INDEX

db=# SELECT * FROM companies WHERE company_name % 'Swisscom' GROUP BY company_name
db-# ORDER BY company_name <-> 'Swisscom';

     company_name
----------------------
 Swisscom
 Swisscom AG
 Swissca
 Swisscom AG Bern
 Swisscom Fixnet SA
 Swisscontact
 Swisscom IT Services
 Swisscom Solutions AG
 Swisscab SA
 viscom swiss print &
```

Trigrams Indexing 2/2

- Trigram indexes work for LIKE, ILIKE, Regexp, %.
- The operator <->uses an index if built with GiST.
- Test with EXPLAIN (ANALYZE) if your indexes are used.

```
db=# EXPLAIN SELECT * FROM companies WHERE company_name % 'Swisscom';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on companies  (cost=8.93..231.87 rows=84 width=19)
   Recheck Cond: ((company_name)::text % 'Swisscom'::text)
   ->  Bitmap Index Scan on trgm_gist_idx  (cost=0.00..8.91 rows=84 width=0)
         Index Cond: ((company_name)::text % 'Swisscom'::text)

db=# EXPLAIN SELECT * FROM companies WHERE company_name ILIKE '%Swisscom%';
                                   QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on companies  (cost=76.65..297.30 rows=83 width=19)
   Recheck Cond: ((company_name)::text ~~* '%Swisscom%'::text)
   ->  Bitmap Index Scan on trgm_gin_idx  (cost=0.00..76.63 rows=83 width=0)
         Index Cond: ((company_name)::text ~~* '%Swisscom%'::text)
```

Outline

Description

Periodical update of the company samples for the monthly business cycles surveys.

- Sources:
  - Meta data surveys with existing participants.
  - Search for new potential participants through the National Register of Companies (BUR).
- Focus for this presentation is the second case.
  - **Task:** Identify companies not yet available in the database.
  - **Issue**: No common identifier between KOF and BUR.
  - **Data**: Company names, addresses and economic activity sectors.
  - **Issue**: KOF data is often the result of manual inserts/updates and may contain typos and divergent content as compared to the official BUR data.
- Define a strategy for comparing textual entries from both data sources.
  - **Issue**: Data may refer to one and the same company and still not match exactly the textual representation.

Layered Strategy

We base our strategy on the following principles:

```
if There is one or more exact match then
    return  match
else
    while No match found or Number of searched fields = 0 do
        Decrease number of search fields and increase required similarity
        if There is one or more match then
            return  match
        end if
    end while
end if
```

In the fuzzy part we check first all fields with a similarity requirement of 0.3 and then remove one field from the search while increasing the match level requirement.

Case Specific Improvements

Indexes: Every record **always** has the name of the company. Thanks to this fact we can build a btree index starting with it and stretching over all other fields. All fields have a gist trigram index of their own.

```
db=# CREATE INDEX all_companies_all_fields_idx
db-# ON all_companies(name, street, house_number, zip, city);
db=# CREATE INDEX name_trgm_idx ON all_companies
db-# USING gist (public.unaccent_string(name) gist_trgm_ops);
```

Multiple languages with accented words. Solved using `unaccent_string()`.

```
db=# SELECT similarity('Bôle','Bole'),
db-#        similarity(unaccent_string('Bôle'),unaccent_string('Bole'));
 similarity | similarity
------------+------------
       0.25 |          1
```

Non differentiating words. Too complicated to ignore these words only pairwise.

```
db=# SELECT similarity('Bäckerei-Konditorei Hug','Bäckerei Konditorei Frey');
-[ RECORD 1 ]--------
similarity | 0.666667

db=# SELECT similarity('Metzgerei Hug','Bäckerei Konditorei Frey');
-[ RECORD 1 ]---------
similarity | 0.0882353
```

Implementation: The data table

List of available companies and its indexes.

```
            Table "public.all_companies"
     Column      |          Type          | Modifiers
-----------------+------------------------+-----------
 company_id      | bigint                 |
 contact_id      | bigint                 |
 name            | character varying(80)  |
 name_extension  | character varying(80)  |
 firstname       | character varying(50)  |
 lastname        | character varying(50)  |
 street          | character varying(80)  |
 house_number    | character varying(20)  |
 zip             | character varying(10)  |
 city            | character varying(50)  |
 surveys         | character varying[]    |
Indexes:
    "all_companies_all_fields_idx" btree (name, street, house_number, zip, city)
    "city_trgm_idx" gist (unaccent_string(city::text) gist_trgm_ops)
    "house_number_trgm_idx" gist (house_number gist_trgm_ops)
    "name_trgm_idx" gist (unaccent_string(name::text) gist_trgm_ops)
    "street_trgm_idx" gist (unaccent_string(street::text) gist_trgm_ops)
    "zip_trgm_idx" gist (zip gist_trgm_ops)
```

Implementation: The Function Signature

```
CREATE OR REPLACE FUNCTION operations.get_match(p_name TEXT,
                                                p_street TEXT,
                                                p_house_number TEXT,
                                                p_zip TEXT,
                                                p_city TEXT,
                                           +-->  p_similarity REAL,
          3 match requirement levels  <--+-->  p_similarity_strong REAL,
                                           +-->  p_similarity_strongest REAL)
RETURNS TABLE (pg_similarity FLOAT,
               pg_match_level INTEGER,
               pg_company_id BIGINT,
               pg_contact_id BIGINT,
               pg_name VARCHAR(80),
               pg_name_extension VARCHAR(80),
               pg_firstname VARCHAR(50),
               pg_lastname VARCHAR(50),
               pg_street VARCHAR(80),
               pg_house_number VARCHAR(20),
               pg_zip VARCHAR(10),
               pg_city VARCHAR(50),
               pg_surveys VARCHAR[])
```

Implementation: The Exact Search

```
DECLARE
  v_stage INTEGER := 0;
[...]
  RETURN QUERY
  SELECT 1.0::FLOAT,
         v_stage,
         company_id,
         contact_id,
         name,
         name_extension,
         firstname,
         lastname,
         street,
         house_number,
         zip,
         city,
         surveys
  FROM public.all_companies
  WHERE name = p_name
  AND CASE WHEN p_street IS NOT NULL THEN street = p_street ELSE true END
  AND CASE WHEN p_house_number IS NOT NULL THEN house_number = p_house_number ELSE true END
  AND CASE WHEN p_zip IS NOT NULL THEN zip = p_zip ELSE true END
  AND CASE WHEN p_city IS NOT NULL THEN city = p_city ELSE true END;
[...]
```

Implementation: The fuzzy Search - From higher...

```
[...]
  v_stage := v_stage + 1; -- At this point = 1

  IF NOT FOUND THEN
    -- RAISE NOTICE 'No exact match, trying fuzzy on all fields';
    RETURN QUERY
    SELECT 1.0 - avg_distance,  -- This is computed with another function
           v_stage, company_id, contact_id, name, name_extension,
           firstname, lastname, street, house_number, zip, city, surveys
    FROM public.all_companies
    WHERE public.unaccent_string(name) % public.unaccent_string(p_name)
    AND CASE WHEN p_street IS NOT NULL THEN
        public.unaccent_string(street) % public.unaccent_string(p_street) ELSE true END
    AND CASE WHEN p_house_number IS NOT NULL THEN
        house_number % p_house_number ELSE true END
    AND CASE WHEN p_zip IS NOT NULL THEN zip % p_zip ELSE true END
    AND CASE WHEN p_city IS NOT NULL
        THEN public.unaccent_string(city) % public.unaccent_string(p_city) ELSE true END;
  END IF;
[...]
```

Implementation: The fuzzy Search - ... to lower or no matching level

```
[...]
  v_stage := v_stage + 1; -- At this stage = 5

  IF NOT FOUND THEN
    -- RAISE NOTICE 'No exact match, trying fuzzy on name only';
    perform set_limit(p_similarity_strongest);
    RETURN QUERY
    SELECT 1.0 - avg_distance,  -- This is computed with another function
           v_stage, company_id, contact_id, name, name_extension,
           firstname, lastname, street, house_number, zip, city, surveys
    FROM kofdata.all_companies
    WHERE public.unaccent_string(name) % public.unaccent_string(p_name)
    ORDER BY public.unaccent_string(name) <-> public.unaccent_string(p_name)
    LIMIT 1;
  END IF;

  IF NOT FOUND THEN
    RETURN QUERY
    SELECT NULL::FLOAT, NULL::INTEGER, NULL::BIGINT, NULL::BIGINT, NULL::VARCHAR(80),
           NULL::VARCHAR(80),NULL::VARCHAR(50), NULL::VARCHAR(50),NULL::VARCHAR(80),
           NULL::VARCHAR(20), NULL::VARCHAR(10), NULL::VARCHAR(50), NULL::VARCHAR[];
  END IF;
[...]
```

Implementation: Usage

```
db=# SELECT pg_similarity, pg_match_level, pg_name, pg_firstname, pg_lastname,
db-#        pg_street, pg_house_number, pg_zip, pg_city
db-# FROM get_match('Consulting','Motorenstrasse','34','8006','Zürich',0.3,0.5,0.7)
db-# ORDER BY pg_similarity DESC;
-[ RECORD 1 ]---+---------------------------------
pg_similarity   | 0.601503765583038
pg_match_level  | 2
pg_name         | DB Consulting Test
pg_firstname    | Charles
pg_lastname     | Clavadetscher
pg_street       | Motorenstrasse
pg_house_number | 18
pg_zip          | 8005
pg_city         | Zürich
-[ RECORD 2 ]---+---------------------------------
pg_similarity   | 0.468571436405182
pg_match_level  | 2
pg_name         | TCP The Consulting Partnership AG
pg_firstname    | Robert
pg_lastname     | Hemmi
pg_street       | Gartenstrasse
pg_house_number | 36
pg_zip          | 8002
pg_city         | Zürich
[...]
Time: 67.450 ms
```

Outline

Resources

- These slides: http://www.schmiedewerkstatt.ch/documents/04-publications/fuzzy_matching_slides_pdfa.pdf
- Online documentation: http://www.postgresql.org/docs/9.4/interactive/index.html
  - Full Text Search: http://www.postgresql.org/docs/9.4/interactive/textsearch.html
  - fuzzystrmatch: http://www.postgresql.org/docs/9.4/static/fuzzystrmatch.html
  - trigrams: http://www.postgresql.org/docs/9.4/static/pgtrgm.html
- A similar presentation by Joe Conway: http://www.joeconway.com/presentations/text_search-pgconfeu2015.pdf

Contact

- Work: clavadetscher@kof.ethz.ch
  http://www.kof.ethz.ch
- SwissPUG: clavadetscher@swisspug.org
  http:www.swisspug.org
- Private: charles@schmiedewerkstatt.ch

Thank you

Thank you very much for your attention !

Feedback

Q&A