

# Where's Waldo? - Text Search and Pattern Matching in PostgreSQL

Joe Conway  
joe.conway@crunchydata.com  
mail@joeconway.com

Crunchy Data

November 18, 2015



# Where's Waldo?

- Many potential methods
- Usually best to use simplest method that fits use case
- Might need to combine more than one method



# Agenda

- Summary of methods
- Overview by method
- Example use cases



# Caveats

- Full text search could easily fill a tutorial  
⇒ this talk provides overview
- Even other methods cannot be covered exhaustively  
⇒ this talk provides overview
- citext not covered, should be considered



# Text Search Methods

- Standard Pattern Matching
  - LIKE operator
  - SIMILAR TO operator
  - POSIX-style regular expressions
- PostgreSQL extensions
  - fuzzystrmatch
    - Soundex
    - Levenshtein
    - Metaphone
    - Double Metaphone
  - pg\_trgm
- Full Text Search



# Note on Extensions

- Extensions used are created as shown below

```
CREATE EXTENSION pg_trgm;  
CREATE EXTENSION fuzzystrmatch;
```



# Sample Data

```
CREATE TABLE messages (  
    [...]   
    _from text NOT NULL,  
    _to text NOT NULL,  
    subject text NOT NULL,  
    bodytxt text NOT NULL,  
    fti tsvector NOT NULL,  
    [...]   
);  
  
select count(1) from messages;  
   count  
-----  
 1086568  
(1 row)
```



# LIKE Syntax

- Expression returns TRUE if string matches pattern
- Typically string comes from relation in FROM clause
- Used as predicate in the WHERE clause to filter returned rows
- LIKE is case sensitive
- ILIKE is case insensitive

```
string LIKE pattern [ESCAPE escape-character]  
string ~~ pattern [ESCAPE escape-character]  
string ILIKE pattern [ESCAPE escape-character]  
string ~~* pattern [ESCAPE escape-character]  
lower(string) LIKE pattern [ESCAPE escape-character]
```





## Negating LIKE

- To negate match, use the NOT keyword
- Appropriate operator also works

```
string NOT LIKE pattern [ESCAPE escape-character]  
string !~~ pattern [ESCAPE escape-character]  
string NOT ILIKE pattern [ESCAPE escape-character]  
string !~~* pattern [ESCAPE escape-character]  
NOT (string LIKE pattern [ESCAPE escape-character])  
NOT (string ILIKE pattern [ESCAPE escape-character])
```



# Wildcards

- Pattern can contain wildcard characters
  - Underscore (" \_ ") matches any single character
  - Percent sign (" % ") matches zero or more characters
- With no wildcards, expression acts like equals
- To match literal wildcard chars, they must be escaped
- Default escape char is backslash (" \ ")
  - May be changed using ESCAPE clause
  - Match the literal escape char by doubling up



## Alternate Index Op Classes

- `varchar_pattern_ops`, `text_pattern_ops` and `bpchar_pattern_ops`
- Useful for anchored pattern matching, e.g. "`<pattern>%`"
- Used by LIKE, SIMILAR TO, or POSIX regex when not using "C" locale
- Also create "normal" index for queries with `<`, `<=`, `>`, or `>=`
- Does NOT work for ILIKE or `~~*`
  - Expression index over `lower(column)`
  - `pg_trgm` index operator class



# ESCAPE Example

```
SELECT 'A\b\C_%_def' LIKE 'A\b\C#_#%#_d%' ESCAPE '#';  
?column?  
-----  
t  
(1 row)
```

## SIMILAR TO Syntax

- Equivalent to LIKE
- Interprets pattern using SQL definition of regex

```
string SIMILAR TO pattern [ESCAPE escape-character]
```

```
string NOT SIMILAR TO pattern [ESCAPE escape-character]
```



# Wildcards

- Same as LIKE
- Also supports meta-characters borrowed from POSIX REs
  - pipe ("|"): either of two alternatives
  - asterisk ("\*"): repetition  $\geq 0$  times
  - plus ("+"): repetition  $\geq 1$  time
  - question mark ("?"): repetition 0 or 1 time
  - "{m}": repetition exactly m times
  - "{m,}": repetition  $\geq m$  times
  - "{m,n}": repetition  $\geq m$  and  $\leq n$  times
  - parentheses ("()"): group items into a single logical item



## SIMILAR TO Examples

```
SELECT 'AbCdEf' SIMILAR TO 'AbC%' AS true,  
       'AbCdEf' SIMILAR TO 'Ab(C|c)%' AS true,  
       'AbcccdEf' SIMILAR TO 'Abc{4}%' AS false,  
       'AbcccdEf' SIMILAR TO 'Abc{3}%' AS true,  
       'Abccdef' SIMILAR TO 'Abc?d?%' AS true;
```

```
true | true | false | true | true
```

```
-----+-----+-----+-----+-----
```

```
t    | t    | f    | t    | t
```

```
(1 row)
```



# Regular Expression Syntax

- Similar to LIKE and ILIKE
- Allowed to match anywhere within string  
⇒ unless RE is explicitly anchored
- Interprets pattern using POSIX definition of regex

```
string ~ pattern -- matches RE, case sensitive  
string ~* pattern -- matches RE, case insensitive  
string !~ pattern -- not matches RE, case sensitive  
string !~* pattern -- not matches RE, case insensitive
```





# Regular Expression Syntax

- POSIX-style REs complex enough to deserve own talk
- See: [www.postgresql.org/docs/9.5/static/functions-matching.html#FUNCTIONS-POSIX-REGEXP](http://www.postgresql.org/docs/9.5/static/functions-matching.html#FUNCTIONS-POSIX-REGEXP)

```
SELECT 'AbCdefzzzzdef' ~* 'Ab((C|c).*)?z+def.*' AS true,
       'AbcabcAbc' ~ '^Ab.*bc$' AS true,
       'AbcabcAbc' ~ '^Ab' AS true,
       'AbcAbcAbc' ~* 'abc' AS true,
       'AbcAbcAbc' ~* '^abc$' AS false;
```

```
 true | true | true | true | false
-----+-----+-----+-----+-----
  t   | t   | t   | t   | f
(1 row)
```



## Regular Expression Example

- Really slow without an index

```
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE bodytxt ~* 'multixact';
QUERY PLAN
```

---

```
Seq Scan on messages
  (cost=0.00..197436.10 rows=108 width=8)
  (actual time=6.435..26851.944 rows=2580 loops=1)
  Filter: (bodytxt ~* 'multixact'::text)
  Rows Removed by Filter: 1083988
Planning time: 1.682 ms
Execution time: 26852.410 ms
```



## Regular Expression Example

- Use trigram GIN index

```
CREATE INDEX trgm_gin_bodytxt_idx
ON messages USING gin (bodytxt using gin_trgm_ops);
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE bodytxt ~* 'multixact';
QUERY PLAN
```

---

```
Bitmap Heap Scan on messages
[...]
```

```
-> Bitmap Index Scan on trgm_gin_bodytxt_idx
    (cost=0.00..124.81 rows=108 width=0)
    (actual time=66.095..66.095 rows=2581 loops=1)
    Index Cond: (bodytxt ~* 'multixact'::text)
Planning time: 3.680 ms
Execution time: 192.912 ms
```



## Regular Expression Example

- Or use trigram GiST index ...oops

```
CREATE INDEX trgm_gist_bodytxt_idx  
ON messages USING gist (bodytxt using gist_trgm_ops);  
ERROR: index row requires 8672 bytes, maximum size is 8191
```



# Regular Expression Compared to FTS

- For the sake of comparison - with full text search

```
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE fti @@ 'multixact:D';
QUERY PLAN
```

---

Bitmap Heap Scan on messages

[...]

```
-> Bitmap Index Scan on messages_fti_idx
    (cost=0.00..64.75 rows=5433 width=0)
    (actual time=1.085..1.085 rows=1475 loops=1)
    Index Cond: (fti @@ '''multixact'''D'::tsquery)
```

Planning time: 0.504 ms

Execution time: 22.054 ms



# Soundex

- soundex: converts string to four character code
- difference: converts two strings, reports # matching positions
- Generally finds similarity of English names
- Part of fuzzystrmatch extension

```
SELECT soundex('Joseph'), soundex('Josef'),  
       difference('Joseph', 'Josef');
```

soundex	soundex	difference
J210	J210	4



# Levenshtein

- Calculates Levenshtein distance between two strings
- Comparisons case sensitive
- Strings non-null, maximum 255 bytes
- Part of fuzzystrmatch extension

```
SELECT levenshtein('Joseph','Josef') AS two,  
       levenshtein('John','Joan') AS one,  
       levenshtein('foo','foo') AS zero;
```

```
two | one | zero  
-----+-----+-----  
  2 |   1 |    0
```



# Metaphone

- Constructs code for an input string
- Comparisons case in-sensitive
- Strings non-null, maximum 255 bytes
- `max_output_length` arg sets max length of code
- Part of `fuzzystrmatch` extension

```
SELECT metaphone('extensive',6) AS "EKSTNS",
       metaphone('exhaustive',6) AS "EKSHST",
       metaphone('ExTensive',3) AS "EKS",
       metaphone('eXhaustivE',3) AS "EKS";
```

```
EKSTNS | EKSHST | EKS | EKS
```

```
-----+-----+-----+-----
```

```
EKSTNS | EKSHST | EKS | EKS
```





## Double Metaphone

- Computes primary and alternate codes for string
- Non-English names especially, can be different
- Comparisons case in-sensitive
- No length limit on the input strings
- Part of fuzzystmatch extension

```
SELECT dmetaphone('extensive') AS "AKST",  
       dmetaphone('exhaustive') AS "AKSS",  
       dmetaphone('Magnus') AS "MNS",  
       dmetaphone_alt('Magnus') AS "MKNS";
```

```
AKST | AKSS | MNS | MKNS
```

```
-----+-----+-----+-----
```

```
AKST | AKSS | MNS | MKNS
```



# Trigram Matching

- Functions and operators for determining similarity
- Trigram is group of three consecutive characters from string
- Similarity of two strings - count number of trigrams shared
- Index operator classes supporting fast similar strings search
- Support indexed searches for LIKE and ILIKE queries
- Comparisons case in-sensitive
- Part of pg\_trgm extension



# Trigram Matching Example

```
\timing  
SELECT set_limit(0.6); -- defaults to 0.3  
SELECT DISTINCT _from, -- uses trgm_gist_idx  
similarity(_from, 'Josef Conway <mail@joeconway.com>') AS sml  
FROM messages WHERE _from % 'Josef Conway <mail@joeconway.com>'  
ORDER BY sml DESC, _from;
```

_from	sml
Joseph Conway <mail@joeconway.com>	0.724138
Joe Conway <mail@joeconway.com>	0.703704
jconway <mail@joeconway.com>	0.678571
"Joe Conway" <joe.conway@mail.com>	0.62963

(4 rows)

Time: 502.002 ms



# Overview

- Searches documents with potentially complex criteria
- Superior to other methods in many cases because:
  - Offers linguistic support for derived words
  - Ignores stop words
  - Ranks results by relevance
  - Very flexibly uses indexes
- Topic very complex - see:
  - <http://www.postgresql.org/docs/9.5/static/textsearch.html>
  - <http://www.postgresql.org/docs/9.5/static/datatype-textsearch.html>
  - <http://www.postgresql.org/docs/9.5/static/functions-textsearch.html>
  - <http://www.postgresql.org/docs/9.5/static/textsearch-indexes.html>



# Preprocessing

- Convert text to tsvector
- Store tsvector
- Index tsvector

```
CREATE FUNCTION messages_fti_trigger_func()
RETURNS trigger LANGUAGE plpgsql AS $$
BEGIN NEW.fti =
    setweight(to_tsvector(coalesce(NEW.subject, '')), 'A') ||
    setweight(to_tsvector(coalesce(NEW.bodytxt, '')), 'D');
RETURN NEW; END $$;
```

```
CREATE TRIGGER messages_fti_trigger BEFORE INSERT OR UPDATE
OF subject, bodytxt ON messages FOR EACH ROW
EXECUTE PROCEDURE messages_fti_trigger_func();
```

```
CREATE INDEX messages_fti_idx ON messages USING gin (fti);
```



# Weighting

- Weights used in relevance ranking
- Array specifies how heavily to weigh each category
- {D-weight, C-weight, B-weight, A-weight}
- defaults: {0.1, 0.2, 0.4, 1.0}



# Creating tsvector

- Parse into tokens
  - Classes of tokens can be processed differently
  - Postgres has standard parser and predefined set of classes
  - Custom parsers can be created
- Convert tokens into lexemes
  - Dictionaries used for this step
    - ⇒ standard dictionaries provided
    - ⇒ custom ones can be created
  - Normalized: different forms of same word made alike
    - ⇒ fold upper-case letters to lower-case
    - ⇒ removal of suffixes
    - ⇒ elimination of stop words



# Writing tsquery

- The pattern to be matched
- Lexemes combined with boolean operators
  - & (AND)
  - | (OR)
  - ! (NOT)
- ! (NOT) binds most tightly  
& (AND) binds more tightly than | (OR)
- Parentheses used to enforce grouping
- Label with \* to specify prefix matching
- Supports weight labels





# Writing tsquery

- to\_tsquery() to create
  - Normalizes tokens lexemes
  - Discards stop words
- Can also cast to tsquery
  - Tokens taken at face value
  - No weight labels

```
SELECT to_tsquery('hello:A & the:A | writing:D') AS to_tsq,
       'hello & the | writing'::tsquery AS cast_tsq
```

```
UNION ALL
```

```
SELECT to_tsquery('postgres:*'),
       'postgres:'::tsquery;
```

to_tsq		cast_tsq
'hello':A   'write':D   'hello' & 'the'   'writing'		'hello & the   writing'
'postgr':*		'postgres':*



# Writing tsquery

- Alternative function plainto\_tsquery()
  - Text parsed and normalized
  - & (AND) operator inserted between surviving words
  - Should use simple strings only
  - No boolean operators,  
No weight labels,  
No prefix-match labels

```
SELECT plainto_tsquery('hello:B & the:B | writing:D') AS tsq1,  
       plainto_tsquery('postgres:*') AS tsq2;  
       tsq1                | tsq2
```

-----+-----  
'hello' & 'b' & 'b' & 'write' & 'd' | 'postgr'



# Match Operator

- Text search match operator @@
- Returns true if tsvector (preprocessed document) matches tsquery (search pattern)
- Either maybe be written first

```
SELECT split_part(_from, '<', 1) AS name, date
FROM messages
WHERE fti @@ 'multixact:A & race:D & bug:D';
```

name	date
Alvaro Herrera	2013-11-25 07:36:19-08
Andres Freund	2013-11-25 08:26:55-08
Andres Freund	2013-11-29 11:58:06-08



# Relevance Ranking

- `ts_rank()`: based on frequency of matching lexemes
- `ts_rank_cd()`: lexeme proximity taken into consideration

```
WITH ts(q) AS
(
  SELECT 'multixact:A & (crash:D | (data:D & loss:D))'::tsquery
)
SELECT ts_rank(m.fti, ts.q) as tsrank
FROM messages m, ts
WHERE m.fti @@ ts.q
ORDER BY tsrank DESC LIMIT 4;
  tsrank
-----
0.999997
0.999997
0.999997
0.999997
```



# Highlighting

- `ts_headline()`: returns excerpt with query terms highlighted
- Apply in an outer query, after inner query LIMIT  
 ⇒ avoids `ts_headline()` overhead on eliminated rows

```
SELECT subject, tsrank, ts_headline(format('%s: %s', subject, bodytxt), q)
FROM (WITH ts(q) AS
      (SELECT 'multixact:A & (crash:D | (data:D & loss:D))'::tsquery)
      SELECT ts_rank(m.fti, ts.q) as tsrank, ts.q, m.subject, m.bodytxt
      FROM messages m, ts WHERE m.fti @@ ts.q ORDER BY tsrank DESC LIMIT 4
      ) AS inner_query LIMIT 1;
```

```
-[ RECORD 1 ]-----
subject      | Is anyone aware of data loss causing MultiXact bugs in 9.3.2?
tsrank       | 0.999997
ts_headline  | <b>data</b> <b>loss</b> causing <b>MultiXact</b>
              bugs in 9.3.2?: I've had multiple complaints
              of apparent <b>data</b>
```



# Pattern Matching: Example Use Cases

- Equal
- Anchored
- Anchored case-insensitive
- Reverse Anchored case-insensitive
- Unanchored case-insensitive
- Fuzzy
- Complex Search with Relevancy Ranking



# Equal

- Find all the rows where column matches '`<pattern>`'
- Equal operator with suitable index is best
- Without an index

```
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE _from = 'Joseph Conway <mail@joeconway.com>';
QUERY PLAN
```

---

```
Seq Scan on messages
  (cost=0.00..197436.10 rows=61 width=8)
  (actual time=49.192..527.343 rows=14 loops=1)
  Filter: (_from = 'Joseph Conway <mail@joeconway.com> '::text)
  Rows Removed by Filter: 1086554
Planning time: 0.256 ms
Execution time: 527.386 ms
```



# Equal

- With an index

```
CREATE INDEX from_idx ON messages(_from);  
EXPLAIN ANALYZE SELECT date FROM messages  
    WHERE _from = 'Joseph Conway <mail@joeconway.com>';  
QUERY PLAN
```

---

Bitmap Heap Scan on messages

[...]

-> Bitmap Index Scan on from\_idx

(cost=0.00..4.88 rows=61 width=0)

(actual time=0.051..0.051 rows=14 loops=1)

Index Cond: (\_from = 'Joseph Conway <mail@joeconway.com>'::text)

Planning time: 0.267 ms

Execution time: 0.161 ms





# Anchored

- Find all the rows where column matches '`<pattern>%`'
- LIKE operator with suitable index is best
- This index does not do the job

```
CREATE INDEX from_idx ON messages(_from);
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE _from LIKE 'Joseph Conway%';
QUERY PLAN
```

---

```
Seq Scan on messages
  (cost=0.00..197436.10 rows=62 width=8)
  (actual time=52.991..536.316 rows=14 loops=1)
  Filter: (_from ~~ 'Joseph Conway% '::text)
  Rows Removed by Filter: 1086554
Planning time: 0.264 ms
Execution time: 536.362 ms
```



# Anchored

- Note `text_pattern_ops` - this works

```
CREATE INDEX pattern_idx ON messages(_from using text_pattern_ops);
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE _from LIKE 'Joseph Conway%';
QUERY PLAN
```

---

```
Index Scan using pattern_idx on messages
  (cost=0.43..8.45 rows=62 width=8)
  (actual time=0.043..0.082 rows=14 loops=1)
  Index Cond: ((_from ~>=~ 'Joseph Conway'::text)
              AND (_from ~<~ 'Joseph Conwaz'::text))
  Filter: (_from ~~ 'Joseph Conway%'::text)
Planning time: 0.490 ms
Execution time: 0.133 ms
```



## Anchored Case-Insensitive

- Find all the rows where column matches '`<pattern>%`'  
 ⇒ but in Case-Insensitive way
- LIKE operator with suitable expression index is good

```
CREATE INDEX lower_pattern_idx
ON messages(lower(_from) using text_pattern_ops);
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE lower(_from) LIKE 'joseph conway%';
    QUERY PLAN
```

-----  
 Bitmap Heap Scan on messages

[...]

```
-> Bitmap Index Scan on lower_pattern_idx
    (cost=0.00..214.76 rows=5433 width=0)
    (actual time=0.074..0.074 rows=14 loops=1)
    Index Cond: ((lower(_from) ~>~ 'joseph conway'::text)
        AND (lower(_from) ~<~ 'joseph conwaz'::text))
```

Planning time: 0.505 ms

Execution time: 0.258 ms



## Anchored Case-Insensitive

- Can also use trigram GIN index with ILIKE

```
CREATE INDEX trgm_gin_idx
ON messages USING gin (_from using gin_trgm_ops);
EXPLAIN ANALYZE SELECT date FROM messages
      WHERE _from ILIKE 'joseph conway%';
      QUERY PLAN
```

---

```
Bitmap Heap Scan on messages
  [...]
  -> Bitmap Index Scan on trgm_gin_idx
      (cost=0.00..176.46 rows=62 width=0)
      (actual time=92.980..92.980 rows=155 loops=1)
      Index Cond: (_from ~>* 'joseph conway%':::text)
Planning time: 0.857 ms
Execution time: 93.473 ms
```



## Anchored Case-Insensitive

- Or a trigram GiST index with ILIKE

```
CREATE INDEX trgm_gist_idx
ON messages USING gist (_from using gist_trgm_ops);
EXPLAIN ANALYZE SELECT date FROM messages
      WHERE _from ILIKE 'joseph conway%';
      QUERY PLAN
```

---

```
Bitmap Heap Scan on messages
[...]
```

```
-> Bitmap Index Scan on trgm_gist_idx
    (cost=0.00..8.88 rows=62 width=0)
    (actual time=53.080..53.080 rows=155 loops=1)
    Index Cond: (_from ~>* 'joseph conway% '::text)
```

```
Planning time: 1.068 ms
Execution time: 53.604 ms
```



## Reverse Anchored Case-Insensitive

- Find all the rows where column matches '%<pattern>' ⇒ but in Case-Insensitive way
- LIKE operator with suitable expression index is good

```
CREATE INDEX rev_lower_pattern_idx
ON messages(lower(reverse(_from)) using text_pattern_ops);
EXPLAIN ANALYZE SELECT date FROM messages WHERE lower(reverse(_from))
    LIKE reverse('%joeconway.com>');
QUERY PLAN
```

```
-----
Bitmap Heap Scan on messages [...]
  -> Bitmap Index Scan on rev_lower_pattern_idx
      (cost=0.00..214.76 rows=5433 width=0)
      (actual time=1.357..1.357 rows=2749 loops=1)
      Index Cond: ((lower(reverse(_from)) ~>~ '>moc.yawnocej'::text)
        AND (lower(reverse(_from)) ~<~ '>moc.yawnocek'::text))
Planning time: 0.278 ms
Execution time: 17.491 ms
```



## Reverse Anchored Case-Insensitive

- Can also use trigram GIN index with ILIKE

```
CREATE INDEX trgm_gin_idx
ON messages USING gin (_from using gin_trgm_ops);
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE _from ILIKE '%joeconway.com>';
QUERY PLAN
```

---

```
Bitmap Heap Scan on messages
[...]
```

```
-> Bitmap Index Scan on trgm_gin_idx
    (cost=0.00..177.58 rows=2344 width=0)
    (actual time=80.537..80.537 rows=2749 loops=1)
    Index Cond: (_from ~~* '%joeconway.com>'::text)
Planning time: 0.915 ms
Execution time: 88.723 ms
```



## Reverse Anchored Case-Insensitive

- Or a trigram GiST index with ILIKE

```
CREATE INDEX trgm_gist_idx
ON messages USING gist (_from using gist_trgm_ops);
EXPLAIN ANALYZE SELECT date FROM messages
      WHERE _from ILIKE '%joeconway.com>';
QUERY PLAN
```

---

Bitmap Heap Scan on messages

[...]

```
-> Bitmap Index Scan on trgm_gist_idx
    (cost=0.00..193.99 rows=2344 width=0)
    (actual time=58.386..58.386 rows=2749 loops=1)
    Index Cond: (_from ~>* '%joeconway.com> '::text)
```

Planning time: 0.921 ms

Execution time: 66.771 ms





## Unanchored Case-Insensitive

- Find all the rows where column matches '%<pattern>%'  
⇒ but in Case-Insensitive way
- This cannot use expression or pattern\_ops index 😞

```
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE _from ILIKE '%Conway%';
QUERY PLAN
```

---

```
Seq Scan on messages
  (cost=0.00..197436.10 rows=5096 width=8)
  (actual time=2.242..2002.998 rows=7402 loops=1)
  Filter: (_from ~>* '%Conway%'::text)
  Rows Removed by Filter: 1079166
  Planning time: 0.860 ms
  Execution time: 2003.667 ms
```



## Unanchored Case-Insensitive

- Use trigram GIN index with ILIKE

```
CREATE INDEX trgm_gin_idx
ON messages USING gin (_from using gin_trgm_ops);
EXPLAIN ANALYZE SELECT date FROM messages
    WHERE _from ILIKE '%Conway%';
QUERY PLAN
```

---

```
Bitmap Heap Scan on messages
[...]
```

```
-> Bitmap Index Scan on trgm_gin_idx
    (cost=0.00..94.22 rows=5096 width=0)
    (actual time=9.060..9.060 rows=7402 loops=1)
    Index Cond: (_from ~>* '%Conway% '::text)

Planning time: 0.915 ms
Execution time: 30.567 ms
```



## Unanchored Case-Insensitive

- Or a trigram GiST index with ILIKE

```
CREATE INDEX trgm_gist_idx
ON messages USING gist (_from using gist_trgm_ops);
EXPLAIN ANALYZE SELECT date FROM messages
      WHERE _from ILIKE '%Conway%';
QUERY PLAN
```

---

Bitmap Heap Scan on messages

[...]

```
-> Bitmap Index Scan on trgm_gist_idx
    (cost=0.00..422.63 rows=5096 width=0)
    (actual time=128.881..128.881 rows=7402 loops=1)
    Index Cond: (_from ~>* '%Conway% '::text)
```

Planning time: 0.871 ms

Execution time: 149.755 ms



# Fuzzy

- Find all the rows where column matches '`<pattern>`'  
 ⇒ but in an inexact way
- Use `dmetaphone` function with an expression index
- Might also use `Soundex`, `Levenshtein`, `Metaphone`, or `pg_trgm`

```
CREATE INDEX dmet_expr_idx ON messages(dmetaphone(_from));
EXPLAIN ANALYZE SELECT _from FROM messages
WHERE dmetaphone(_from) = dmetaphone('josef konwei');
      QUERY PLAN
```

-----  
 Bitmap Heap Scan on messages

[...]

```
-> Bitmap Index Scan on dmet_expr_idx
    (cost=0.00..101.17 rows=5433 width=0)
    (actual time=0.085..0.085 rows=108 loops=1)
    Index Cond: (dmetaphone(_from) = 'JSFK'::text)
```

Planning time: 0.272 ms

Execution time: 0.445 ms



# Complex Requirements

- Full Text Search
  - Complex multi-word searching
  - Relevancy Ranking

```
EXPLAIN ANALYZE SELECT date FROM messages
WHERE fti @@ 'bug:A & deadlock:D & startup:D';
QUERY PLAN
```

---

Bitmap Heap Scan on messages  
[...]

```
-> Bitmap Index Scan on messages_fti_idx
    (cost=0.00..52.02 rows=2 width=0)
    (actual time=9.261..9.261 rows=93 loops=1)
    Index Cond: (fti @@ '''bug''':A & '''deadlock''':D &
                '''startup''':D)::tsquery)
```

```
Planning time: 0.469 ms
Execution time: 12.614 ms
```



# Questions?

Thank You!  
mail@joeconway.com

