Full-text search in PostgreSQL in milliseconds

Oleg Bartunov (thanks 1C for support)
Alexander Korotkov
**FTS in PostgreSQL**

- Full integration with PostgreSQL
- 27 built-in configurations for 10 languages
- Support of user-defined FTS configurations
- Pluggable dictionaries (ispell, snowball, thesaurus), parsers
- Relevance ranking
- GiST and GIN indexes with concurrency and recovery support
- Rich query language with query rewriting support

**It's cool, but we want faster FTS!**
**FTS in PostgreSQL**

- OpenFTS — 2000, Pg as a storage
- GiST index — 2000, thanks Rambler
- Tsearch — 2001, contrib:no ranking
- Tsearch2 — 2003, contrib:config

- GIN — 2006, thanks, JFG Networks
- FTS — 2006, in-core, thanks, EnterpriseDB
- E-FTS — Enterprise FTS, thanks ????
ACID overhead is really big :(

- Foreign solutions: Sphinx, Solr, Lucene....
  - Crawl database and index (time lag)
  - No access to attributes
  - Additional complexity
  - BUT: Very fast!

Can we improve native FTS?
Can we improve native FTS?

156676 Wikipedia articles:

postgres=# explain analyze
SELECT docid, ts_rank(text_vector, to_tsquery('english', 'title')) AS rank
FROM ti2
WHERE text_vector @@ to_tsquery('english', 'title')
ORDER BY rank DESC
LIMIT 3;

Limit (cost=8087.40..8087.41 rows=3 width=82) (actual time=433.750..433.752 rows=3 loops=1)
  ->  Sort (cost=8087.40..8206.63 rows=47692 width=282)
(actual time=433.749..433.749 rows=3 loops=1)
    Sort Key: (ts_rank(text_vector, 'title')::tsquery)
    Sort Method: top-N heapsort  Memory: 25kB
  ->  Bitmap Heap Scan on ti2 (cost=529.61..7470.99 rows=47692 width=282)
( actual time=15.094..423.452 rows=47855 loops=1)
    Recheck Cond: (text_vector @@ 'title'::tsquery)
  ->  Bitmap Index Scan on ti2_index (cost=0.00..517.69 rows=47692 width=0)
( actual time=13.736..13.736 rows=47855 loops=1)
    Index Cond: (text_vector @@ 'title'::tsquery)
Total runtime: 433.787 ms

HEAP IS SLOW 400 ms!
Can we improve native FTS?

156676 Wikipedia articles:

postgres=# explain analyze
SELECT docid, ts_rank(text_vector, to_tsquery('english', 'title')) AS rank
FROM ti2
WHERE text_vector @@ to_tsquery('english', 'title')
ORDER BY text_vector>< plainto_tsquery('english','title')
LIMIT 3;

What if we have this plan?

Limit  (cost=20.00..21.65 rows=3 width=282) (actual time=18.376..18.427 rows=3 loops=1)
  ->  Index Scan using ti2_index on ti2  (cost=20.00..26256.30 rows=47692 width=282)
      (actual time=18.375..18.425 rows=3 loops=1)
      Index Cond: (text_vector @@ ''titl''::tsquery)
      Order By: (text_vector >< ''titl''::tsquery)

Total runtime: 18.511 ms vs 433.787 ms

We'll be FINE!
## 6.7 mln classifieds

<table>
<thead>
<tr>
<th></th>
<th>Without patch</th>
<th>With patch</th>
<th>With patch functional index</th>
<th>Sphinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table size</td>
<td>6.0 GB</td>
<td>6.0 GB</td>
<td>2.87 GB</td>
<td>-</td>
</tr>
<tr>
<td>Index size</td>
<td>1.29 GB</td>
<td>1.27 GB</td>
<td>1.27 GB</td>
<td>1.12 GB</td>
</tr>
<tr>
<td>Index build time</td>
<td>216 sec</td>
<td>303 sec</td>
<td>718 sec</td>
<td>180 sec*</td>
</tr>
<tr>
<td>Queries in 8 hours</td>
<td>3,0 mln.</td>
<td>42.7 mln.</td>
<td>42.7 mln.</td>
<td>32.0 mln.</td>
</tr>
</tbody>
</table>

WOW !!!
# 20 mln descriptions

<table>
<thead>
<tr>
<th></th>
<th>Without patch</th>
<th>With patch</th>
<th>With patch functional index</th>
<th>Sphinx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table size</td>
<td>18.2 GB</td>
<td>18.2 GB</td>
<td>11.9 GB</td>
<td>-</td>
</tr>
<tr>
<td>Index size</td>
<td>2.28 GB</td>
<td>2.30 GB</td>
<td>2.30 GB</td>
<td>3.09 GB</td>
</tr>
<tr>
<td>Index build time</td>
<td>258 sec</td>
<td>684 sec</td>
<td>1712 sec</td>
<td>481 sec*</td>
</tr>
<tr>
<td>Queries in 8 hours</td>
<td>2.67 mln.</td>
<td>38.7 mln.</td>
<td>38.7 mln.</td>
<td>26.7 mln.</td>
</tr>
</tbody>
</table>

WOW !!!
GIN improvements
Inverted Index

Report Index

A
abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 61
aerospace propulsion, 52
aerospace robotics, 68
aluminium, 17
amorphous state, 67
angular velocity measurement, 58
antenna phased arrays, 41, 46, 66
argon, 21
assembling, 22
atomic force microscopy, 13, 27, 35
atomic layer deposition, 15
attitude control, 60, 61
attitude measurement, 59, 61
automatic test equipment, 71
automatic testing, 24
compensation, 30, 68
compressive strength, 54
compressors, 29
computational fluid dynamics, 23, 29
computer games, 56
concurrent engineering, 14
contact resistance, 47, 66
convertors, 22
coplanar waveguide components, 40
Couette flow, 21
creep, 17
crystallisation, 64
current density, 13, 16
D
design for manufacture, 25
design for testability, 25
diamond, 3, 27, 43, 54, 67
dielectric losses, 31, 42
dielectric polarisation, 31
dielectric relaxation, 64
dielectric thin films, 16
differential amplifiers, 28
diffraction gratings, 68
discrete wavelet transforms, 72
displacement measurement, 11
display devices, 56
distributed feedback lasers, 38
Inverted Index

Query: compensation accelerometers

Index: accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
       compensation, 30, 68

Result: 30
Inverted Index in PostgreSQL

Report Index

A
abrasives, 27
acceleration measurement, 58
accelerometers, 5, 10, 25, 28, 30, 36, 58, 59, 61, 73, 74
actuators, 4, 37, 46, 49
adaptive Kalman filters, 60, 61
adhesion, 63, 64
adhesive bonding, 15
adsorption, 44
aerodynamics, 29
aerospace instrumentation, 6
aerospace propulsion, 52
aerospace robotics, 68
aluminium, 17
amorphous state, 67
angular velocity measurement, 13
antenna phased arrays, 41, 42
argon, 21
assembling, 22
atomic force microscopy, 13
atomic layer deposition, 15
attitude control, 60, 61
attitude measurement, 59, 61
automatic test equipment, 71
automatic testing, 24

B
backward wave oscillators, 45
compensation, 30, 68
compressive strength, 54
compressors, 29
computational fluid dynamics, 23, 29
computer games, 56
concurrent engineering, 14
couette flow, 21
creep, 17
crystallisation, 64
coupler waveguide components, 40

No positions in index!

Posting list
Posting tree

GIN Structure

Entry page, level 0: keywords
abc  bar  foo

Entry page, level 0 (leaf)
Pointer to posting tree: B-Tree over ItemPointer to heap
aaa

Entry page, level 0 (leaf)
Posting list: sorted array of ItemPointer to heap
abc

Entry page, level 0
baa  bar

Posting page, level N: ItemPointer
14:17  218:1  1021:6

Posting page, level 0 (leaf)
1:33  2:7  14:17
Right bound 14:17

Posting page, level 0 (leaf)
123:1  158:18
Right bound 218:1

Oleg Bartunov
Alexander Korotkov

Full-text search in PostgreSQL in milliseconds
PGConf.EU-2012, Prague
Summary of changes

• Compressed storage with additional information
• Optimized search («frequent_entry & rare_entry» case)
• Return ordered results by index (ORDER BY optimization)

interface changes needs for all this stuff
Every GIN application can have a benefit

- Fulltext search: store word positions, get results in relevance order.
- Trigram indexes: store trigram positions, get results in similarity order.
- Array indexes: store array length, get results in similarity order.
Store additional information
See Appendix 1 for more details

Use increments and variable byte encoding to keep index small

1034, 1036, 1038 (12 bytes) => 1034, 2, 2 (4 bytes)
Fast scan

entry1 && entry2

Visiting 3 pages instead of 7
ORDER BY using index

**Before**

```
SELECT itemid, title
FROM items
WHERE fts @@ to_tsquery('english', 'query')
ORDER BY ts_rank(fts, to_tsquery('english', 'query')) DESC
LIMIT 10;
```

**After**

```
SELECT itemid, title
FROM items
WHERE fts @@ to_tsquery('english', 'query')
ORDER BY fts > to_tsquery('english', 'query')
LIMIT 10;
```

Ranking and sorting are outside the fulltext index

Index returns data ordered by rank. Ranking and sorting are inside.

368 ms vs 13 ms
Example: frequent entry (30%)

### Before:

<table>
<thead>
<tr>
<th>node type</th>
<th>count</th>
<th>sum of times</th>
<th>% of query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmap Heap Scan</td>
<td>1</td>
<td>367.687 ms</td>
<td>94.6 %</td>
</tr>
<tr>
<td>Bitmap Index Scan</td>
<td>1</td>
<td>6.570 ms</td>
<td>1.7 %</td>
</tr>
<tr>
<td>Limit</td>
<td>1</td>
<td>0.001 ms</td>
<td>0.0 %</td>
</tr>
<tr>
<td>Sort</td>
<td>1</td>
<td>14.465 ms</td>
<td>3.7 %</td>
</tr>
</tbody>
</table>

### After:

<table>
<thead>
<tr>
<th>node type</th>
<th>count</th>
<th>sum of times</th>
<th>% of query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Scan</td>
<td>1</td>
<td>13.346 ms</td>
<td>100.0 %</td>
</tr>
<tr>
<td>Limit</td>
<td>1</td>
<td>0.001 ms</td>
<td>0.0 %</td>
</tr>
</tbody>
</table>
Example: rare entry (0.08%)  

<table>
<thead>
<tr>
<th>node type</th>
<th>count</th>
<th>sum of times</th>
<th>% of query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before: Bitmap Heap Scan</td>
<td>1</td>
<td>0.959 ms</td>
<td>93.4 %</td>
</tr>
<tr>
<td>Before: Bitmap Index Scan</td>
<td>1</td>
<td>0.027 ms</td>
<td>2.6 %</td>
</tr>
<tr>
<td>Before: Limit</td>
<td>1</td>
<td>0.001 ms</td>
<td>0.1 %</td>
</tr>
<tr>
<td>Before: Sort</td>
<td>1</td>
<td>0.040 ms</td>
<td>3.9 %</td>
</tr>
<tr>
<td>After: Index Scan</td>
<td>1</td>
<td>0.052 ms</td>
<td>98.1 %</td>
</tr>
<tr>
<td>After: Limit</td>
<td>1</td>
<td>0.001 ms</td>
<td>1.9 %</td>
</tr>
</tbody>
</table>

1.1 ms

0.07 ms
Example: frequent entry (30%) & rare entry (0.08%)

<table>
<thead>
<tr>
<th>node type</th>
<th>count</th>
<th>sum of times</th>
<th>% of query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bitmap Heap Scan</td>
<td>1</td>
<td>1.547 ms</td>
<td>23.0 %</td>
</tr>
<tr>
<td>Bitmap Index Scan</td>
<td>1</td>
<td>5.151 ms</td>
<td>76.7 %</td>
</tr>
<tr>
<td>Limit</td>
<td>1</td>
<td>0.000 ms</td>
<td>0.0 %</td>
</tr>
<tr>
<td>Sort</td>
<td>1</td>
<td>0.022 ms</td>
<td>0.3 %</td>
</tr>
</tbody>
</table>

Before: 6.7 ms

<table>
<thead>
<tr>
<th>node type</th>
<th>count</th>
<th>sum of times</th>
<th>% of query</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index Scan</td>
<td>1</td>
<td>0.998 ms</td>
<td>100.0 %</td>
</tr>
<tr>
<td>Limit</td>
<td>1</td>
<td>0.000 ms</td>
<td>0.0 %</td>
</tr>
</tbody>
</table>

After: 1.0 ms
Sponsors are welcome!

• 150 Kb patch for 9.3

• Todo:
  – Fix everything we broke :(
  – Fast scan interface
  – Accelerate index build
  – Partial match support

• Datasets and workloads are welcome
Appendix 1

Compressed storage of additional information in GIN
Add additional information (word positions)
typedef struct ItemPointer Data
{
    BlockIdData ip_blkid;
    OffsetNumber ip_posid;
}

typedef struct BlockIdData
{
    uint16 bi_hi;
    uint16 bi_lo;
} BlockIdData;

6 bytes
/**
 * Equivalent to
 * typedef struct {
 *    uint16
 *    weight: 2,
 *    pos: 14;
 * } WordEntryPos;
 */

typedef uint16 WordEntryPos;

2 bytes
BlockIdData compression

Oleg Bartunov
Alexander Korotkov
Full-text search in PostgreSQL in milliseconds
PGConf.EU-2012, Prague
OffsetNumber compression

O0-O15 – OffsetNumber bits
N – Additional information NULL bit
WordEntryPos compression

P0-P13 – position bits
W0,W1 – weight bits
Example

![Diagram showing a process of Diff and Compression with hexadecimal values.](image)

Oleg Bartunov
Alexander Korotkov
Full-text search in PostgreSQL in milliseconds
PGConf.EU-2012, Prague
GIN interface changes
extractValue

Datum *extractValue
(
    Datum itemValue,
    int32 *nkeys,
    bool **nullFlags,
    Datum **addInfo,
    bool **addInfoIsNull
)

Datum *extractValue
(
    Datum query,
    int32 *nkeys,
    StrategyNumber n,
    bool **pmatch,
    Pointer **extra_data,
    bool **nullFlags,
    int32 *searchMode,
    ???bool **required???
)
bool consistent(
    bool check[],
    StrategyNumber n,
    Datum query,
    int32 nkeys,
    Pointer extra_data[],
    bool *recheck,
    Datum queryKeys[],
    bool nullFlags[],
    Datum addInfo[],
    bool addInfoIsNull[]
)
```c
float8 calcRank(
    bool check[],
    StrategyNumber n,
    Datum query,
    int32 nkeys,
    Pointer extra_data[],
    bool *recheck,
    Datum queryKeys[],
    bool nullFlags[],
    Datum addInfo[],
    bool addInfoisNull[]
)
```
Datum joinAddInfo
(
    Datum addInfo[]
)

Full-text search in PostgreSQL in milliseconds
PGConf.EU-2012, Prague
Planner optimization

Remove unused targets when ORDER BY uses index
Before

test=# EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM test ORDER BY slow_func(x,y) LIMIT 10;

QUERY PLAN

Limit (cost=0.00..3.09 rows=10 width=16) (actual time=11.344..103.443 rows=10 loops=1)
  Output: x, y, (slow_func(x, y))
  -> Index Scan using test_idx on public.test (cost=0.00..309.25 rows=1000 width=16)
      (actual time=11.341..103.422 rows=10 loops=1)
        Output: x, y, slow_func(x, y)
Total runtime: 103.524 ms
(5 rows)
test=# EXPLAIN (ANALYZE, VERBOSE) SELECT * FROM test ORDER BY slow_func(x,y) LIMIT 10;

QUERY PLAN

 Limit (cost=0.00..3.09 rows=10 width=16) (actual time=0.062..0.093 rows=10 loops=1)
 Output: x, y
 -> Index Scan using test_idx on public.test (cost=0.00..309.25 rows=1000 width=16)
  (actual time=0.058..0.085 rows=10 loops=1)
   Output: x, y
Total runtime: 0.164 ms
(5 rows)