Next generation of GIN

Alexander Korotkov
Oleg Bartunov
Two GIN applications

- Full-text search
  - tsvector @@ tsquery
  - Indexing tsvector data type

- Hstore
  - (key,value) storage
  - Indexing keys, values
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!
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=282) (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=282)
              (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 rank DESC
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
```
Can we improve native FTS?

156676 Wikipedia articles:

```sql
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;
```

18.511 ms vs 433.787 ms

We'll be FINE!
# 6.7 mln classifieds

<table>
<thead>
<tr>
<th></th>
<th>9.3</th>
<th>9.3+patch</th>
<th>9.3+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>9.3</th>
<th>9.3+ patch</th>
<th>9.3+ 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 !!!
Hstore

- Data
  - 1,252,973 bookmarks from Delicious in json format
- Search, contains operator @>
  - select count(*) from hs where h @> '{term=>NYC}';
  - 0.98 s (seq) vs 0.1 s (GIN) → We want faster operation!
- Observation
  - GIN indexes separately keys and values
  - Key 'tags' is very frequent -1138532, value '{term=>NYC}' is rare — 285
  - Current GIN: time (freq & rare) ~ time(freq)
Observation

- GIN indexes separately keys and values
- Key 'tags' is very frequent - 1138532, value '{{term=>NYC}}' is rare — 285
- Current GIN: time (freq & rare) ~ time(freq)

What if GIN supports

- time (freq & rare) ~ time(rare)

```
=# select count(*) from hs where h::hstore @> 'tags=>{{term=>NYC}}'::hstore;
    count
------
     285
(1 row)
```

Time: 17.372 ms

0.98 s (seq) vs 0.1 s (GIN) vs 0.017 s (GIN++)
These two examples motivate GIN improvements!
Summary of changes

- Compressed storage
- Fast scan («frequent_entry & rare_entry» case)
- Store additional information
- Return ordered results by index (ORDER BY optimization)
- Planner optimization
typedef struct ItemPointerData
{
    BlockIdData ip_blkidx;
    OffsetNumber ip_posid;
}

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

What we have:

• Offset is typically low
• Block number is ascending

What to do:

• Use var-byte encoding
• Store increments for block numbers
Varbyte compression
Compressed storage

```
161;3  210;11  223;5  240;65  241;6
```

```
161;3  49;11  13;5  17;65  1;6
```

```
0xA1; 0x01; 0x03
0x11; 0x41
```

```
0x31; 0x0B
0x1; 0x6
```

```
0xD; 0x05
```

Diff

Varbyte encoding
Tests

Dataset: mailing lists archives
976488 messages of 1300 characters average length

<table>
<thead>
<tr>
<th>Parameter</th>
<th>master</th>
<th>patched</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index build time</td>
<td>110 s</td>
<td>105 s</td>
</tr>
<tr>
<td>Initial index size</td>
<td>844 MB</td>
<td>400 MB</td>
</tr>
<tr>
<td>24K queries execution</td>
<td>1521 s</td>
<td>1447 s</td>
</tr>
<tr>
<td>Whole index update time</td>
<td>318 s</td>
<td>317 s</td>
</tr>
<tr>
<td>Index size after updates</td>
<td>1521 MB</td>
<td>683 MB</td>
</tr>
<tr>
<td>24K queries execution after updates</td>
<td>1557</td>
<td>1585</td>
</tr>
</tbody>
</table>
Fast scan: idea

entry1 && entry2

Visiting parts of 3 pages instead of 7
Fast scan interface

New consistent method using tri-state logic:
• true
• false
• unknown
Fast scan interface

Can actually we skip these?
If consistent([false, unknown]) = false then we really can.
Store additional information
typedef uint16 WordEntryPos;

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

2 bytes
OffsetNumber compression

O7 O6 O5 O4 O3 O2 O1 O0  O15 O14 O13 O12 O11 O10 O9 O8

0  N  O5 O4 O3 O2 O1 O0

1  O6 O5 O4 O3 O2 O1 O0  0  N  O12 O11 O10 O9 O8 O7

1  O6 O5 O4 O3 O2 O1 O0  1  O13 O12 O11 O10 O9 O8 O7

0  N  0  0  0  0  0  O15 O14

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

P0-P13 – position bits
W0,W1 – weight bits
ORDER BY using index

**Before**

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

**After**

```sql
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.

8002 used blocks vs 34 used block
extractValue

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

Next generation of GIN
PGConf.EU-2013, Dublin
Example: frequent entry (30%)

<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>

Before:

388 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>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>

After:

13 ms
Example: rare entry (0.08%)
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
Benefit of additional information

• 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.
Planner optimization

- ORDER BY expression is always evaluated
- When we get right ordering from index we don't need to evaluate ORDER BY expression
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)
Current state

- Patches took one round of review by Heikki Linnakangas
- Compression and planner optimization are now on commitfest
- Other patches are under reworking
Thanks for attention!