

GIN in 9.4 and further

Heikki Linnakangas, Alexander Korotkov, Oleg Bartunov

May 23, 2014



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;
```

**HEAP IS SLOW
400 ms !**

```
Limit  (cost=8087.40..8087.41 rows=3 width=32) (actual time=433.750..433.752 rows=3)
```

```
-> 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
```



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 loop
->  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:

```
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

	9.3	9.3+patch	9.3+patch functional index	Sphinx
Table size	6.0 GB	6.0 GB	2.87 GB	-
Index size	1.29 GB	1.27 GB	1.27 GB	1.12 GB
Index build time	216 sec	303 sec	718sec	180 sec*
Queries in 8 hours	3,0 mln.	42.7 mln.	42.7 mln.	32.0 mln.

WOW !!!



20 mln descriptions

	9.3	9.3+ patch	9.3+ patch functional index	Sphinx
Table size	18.2 GB	18.2 GB	11.9 GB	-
Index size	2.28 GB	2.30 GB	2.30 GB	3.09 GB
Index build time	258 sec	684 sec	1712 sec	481 sec*
Queries in 8 hours	2.67 mln.	38.7 mln.	38.7 mln.	26.7 mln.

WOW !!!



Hstore

- Data
 - 1,252,973 bookmarks from Delicious in json format
- Search, contains operator @>
 - `select count(*) from hs where h @> 'tags=>{{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)



Hstore

- 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)

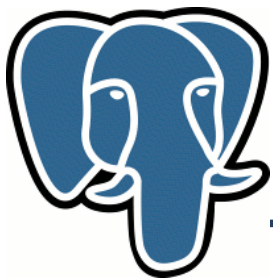
```
=# select count(*) from hs where h::hstore @> 'tags=>{{term=>NYC}}':::hstore;  
count
```

```
-----  
285          0.98 s (seq) vs 0.1 s (GIN)  vs 0.017 s (GIN++)  
(1 row)
```

Time: 17.372 ms



These two examples motivate
GIN improvements !



GIN History

- Introduced at PostgreSQL Anniversary Meeting in Toronto, Jul 7-8, 2006 by Oleg Bartunov and Teodor Sigaev

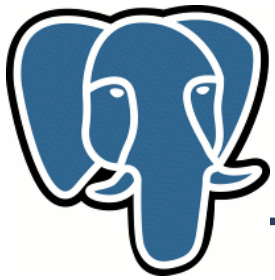


Generalized Inverted Index

- An inverted index is an index structure storing a set of (key, posting list) pairs, where 'posting list' is a set of documents in which the key occurs.
- Generalized means that the index does not know which operation it accelerates. It works with custom strategies, defined for specific data types. GIN is similar to GiST and differs from B-Tree indices, which have predefined, comparison-based operations.

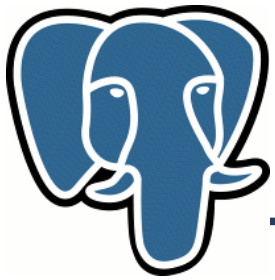
Oleg Bartunov, Teodor Sigaev

PostgreSQL Summit, Toronto, July 8-9, 2006



GIN History

- Introduced at PostgreSQL Anniversary Meeting in Toronto, Jul 7-8, 2006 by Oleg Bartunov and Teodor Sigaev
- Supported by JFG Networks (France)
- «Gin stands for Generalized Inverted iNdex and should be considered as a genie, not a drink.»
- Alexander Korotkov, Heikki Linnakangas have joined GIN++ development in 2013



GIN History

- From GIN Readme, 2006

TODO

Nearest future:

- * Opcodes for all types (no programming, just many catalog changes).

Distant future:

- * Replace B-tree of entries to something like GiST (**VODKA ! 2014**)
- * Add multicolumn support
- * Optimize insert operations (background index insertion)

Two major improvements

1. Compressed posting lists

Makes GIN indexes smaller. Smaller is better.

2. When combining two or more search keys, skip over items that match one key but not the other.

Makes “rare & frequent” type queries much faster.

Part 1: Compressed posting lists

```
create extension btree_gin;  
  
create table numbers (n int4);  
  
insert into numbers  
select g % 10 from generate_series(1, 10000000) g;  
  
create index numbers_btree on numbers (n);  
create index numbers_gin on numbers using gin (n);
```

Example table

```
select n, count(*) from numbers group by n order by n;
```

n	count
0	1000000
1	1000000
2	1000000
3	1000000
4	1000000
5	1000000
6	1000000
7	1000000
8	1000000
9	1000000

(10 rows)

GIN index size, 9.4 vs 9.3

9.3:

```
postgres=# \di+
```

Schema	Name	...	Size	...
public	numbers_btree		214 MB	
public	numbers_gin		58 MB	

9.4:

public	numbers_btree		214 MB	
public	numbers_gin		11 MB	

5x smaller!

How did we achieve this?

GIN is just a B-tree, with efficient storage of duplicates.

- ▶ Pointers to the heap tuples are stored as ItemPointers
- ▶ Each item pointer consists of a block number, and offset within the block
 - ▶ 6 bytes in total, 4 bytes for block number and 2 for offset
 - ▶ presented in text as (,)

Example of one index tuple:

key	item pointer
FOOBAR:	(12, 5)

GIN structure (version 9.3)

B-tree page:

FOOBAR: (12, 5)

FOOBAR: (12, 6)

FOOBAR: (12, 7)

FOOBAR: (13, 2)

GIN (entry tree) page:

FOOBAR: (12, 5), (12, 6), (12, 7), (13, 2)

GIN stores the key only once, making it efficient for storing duplicates

GIN entry tuple in version 9.3

FOOBAR: (12, 5), (12, 6), (12, 7), (13, 2)

Tuple size:

size (bytes)	description
6	fixed index tuple overhead
7	key, 6 characters + 1 byte header
24	item pointers, 4 * 6 bytes

37 bytes in total

GIN entry tuple in version 9.4

- ▶ The list of item pointers is sorted (as in 9.3)
- ▶ The first item pointer in the list is stored normally.
- ▶ Subsequent items are stored as a delta from previous item

9.3: (12, 5) (12, 6) (13, 2) (13, 4)

9.4: (12, 5) +1 +1 +2044 +2

Varbyte encoding

- ▶ most deltas are small integers.
- ▶ varbyte encoding uses variable number of bytes for each integer.
- ▶ single byte for a value < 128

0XXXXXXXX

1XXXXXXXX 0XXXXYYY

1XXXXXXXX 1XXXXYYY 0YYYYYYYY

1XXXXXXXX 1XXXXYYY 1YYYYYYYY 0YYYYYYYY

1XXXXXXXX 1XXXXYYY 1YYYYYYYY 1YYYYYYYY 0YYYYYYYY

1XXXXXXXX 1XXXXYYY 1YYYYYYYY 1YYYYYYYY 1YYYYYYYY
YYYYYYYY

- ▶ Each item needs 1-6 bytes (was always 6 bytes before)

Compressed posting lists

- ▶ When a GIN index tuple grows too large to fit on page, a separate tree called posting tree is created
- ▶ the posting tree is essentially a huge list of item pointers
- ▶ Posting list compression applies to both in-line posting lists and posting tree pages.

Compression algorithm

- ▶ Plenty of encoding schemes for sorted lists of integers in literature
- ▶ Experimented with Simple9
 - ▶ was somewhat more compact than the chosen varbyte encoding
- ▶ Could use a (compressed) bitmap
 - ▶ would turn GIN into a bitmap index

With the chosen algorithm, removing an item never increases index size.

- ▶ VACUUM doesn't cause you to run out of disk space.

Summary

Best case example:

Table	346 MB
B-tree index	214 MB
GIN (9.3)	58 MB
GIN (9.4)	11 MB

- ▶ The new code can read old-format pages, so `pg_upgrade` works
 - ▶ but you won't get the benefit until you `REINDEX`.
- ▶ More expensive to do random updates
 - ▶ but GIN isn't very fast with random updates anyway. . .

Part 2: “rare & frequent” queries got faster

Three fundamental GIN operations:

1. Extract keys from a value to insert or query
2. Store them on disk
3. **Combine matches of several keys efficiently, and determine which items match the overall query**

Consistent function

```
select plainto_tsquery(  
    'an advanced PostgreSQL open source database');  
           plainto_tsquery
```

```
    'postgresql' & 'advanc' & 'open' & 'sourc' & 'databas'  
(1 row)
```

```
select * from foo where col @@ plainto_tsquery(  
    'an advanced PostgreSQL open source database'  
)
```

3. Combine matches efficiently (0/4)

The query returns the following matches from the index:

advanc	databas	open	postgresql	sourc
(0,8)	(0,3)	(0,2)	(0,8)	(0,1)
(0,14)	(0,8)	(0,8)	(0,41)	(0,2)
(0,17)	(0,43)	(0,30)		(0,8)
(0,22)	(0,47)	(0,33)		(0,12)
(0,26)	(1,32)	(0,36)		(0,13)
(0,33)		(0,44)		(0,18)
(0,34)		(0,46)		(0,19)
(0,35)		(0,56)		(0,20)
(0,45)		(1,4)		(0,26)
(0,47)		(1,22)		(0,34)
(0,48)		(1,24)		(0,35)

3. Combine matches efficiently (1/4)

(0,1) contains only word "sourc" -> no match

advanc	databas	open	postgresql	sourc
(0,8)	(0,3)	(0,2)	(0,8)	(0,1)
(0,14)	(0,8)	(0,8)	(0,41)	(0,2)
(0,17)	(0,43)	(0,30)		(0,8)
(0,22)	(0,47)	(0,33)		(0,12)
(0,26)	(1,32)	(0,36)		(0,13)
(0,33)		(0,44)		(0,18)
(0,34)		(0,46)		(0,19)
(0,35)		(0,56)		(0,20)
(0,45)		(1,4)		(0,26)
(0,47)		(1,22)		(0,34)
(0,48)		(1,34)		(0,35)

3. Combine matches efficiently (2/4)

(0,2) contains words "open" and "sourc" -> no match

advanc	databas	open	postgresql	sourc
(0,9)	(0,3)	(0,2)	(0,8)	(0,1)
(0,14)	(0,8)	(0,8)	(0,41)	(0,2)
(0,17)	(0,43)	(0,30)		(0,8)
(0,22)	(0,47)	(0,33)		(0,12)
(0,26)	(1,32)	(0,36)		(0,13)
(0,33)		(0,44)		(0,18)
(0,34)		(0,46)		(0,19)
(0,35)		(0,56)		(0,20)
(0,45)		(1,4)		(0,26)
(0,47)		(1,22)		(0,34)
(0,48)		(1,34)		(0,35)

3. Combine matches efficiently (3/4)

(0,3) contains word "databas" -> no match

advanc	databas	open	postgresql	sourc
(0,8)	(0,3)	(0,2)	(0,8)	(0,1)
(0,14)	(0,8)	(0,8)	(0,41)	(0,2)
(0,17)	(0,43)	(0,30)		(0,8)
(0,22)	(0,47)	(0,33)		(0,12)
(0,26)	(1,32)	(0,36)		(0,13)
(0,33)		(0,44)		(0,18)
(0,34)		(0,46)		(0,19)
(0,35)		(0,56)		(0,20)
(0,45)		(1,4)		(0,26)
(0,47)		(1,22)		(0,34)
(0,48)		(1,24)		(0,35)

3. Combine matches efficiently (4/4)

(0,8) contains all the words $\rightarrow$ *match*

advanc	databas	open	postgresql	sourc
(0,8)	(0,3)	(0,2)	(0,8)	(0,1)
(0,14)	(0,8)	(0,8)	(0,41)	(0,2)
(0,17)	(0,43)	(0,30)		(0,8)
(0,22)	(0,47)	(0,33)		(0,12)
(0,26)	(1,32)	(0,36)		(0,13)
(0,33)		(0,44)		(0,18)
(0,34)		(0,46)		(0,19)
(0,35)		(0,56)		(0,20)
(0,45)		(1,4)		(0,26)
(0,47)		(1,22)		(0,34)
(0,48)		(1,24)		(0,35)

9.4 is smarter

Instead of scanning through the posting lists of all the keywords, only scan through the list with fewest items, and skip the other lists to the next possible match.

- ▶ Big improvement for “frequent-term AND rare-term” style queries

9.4 example (1/3)

The item lists with the fewest items (estimated) are scanned first.

postgresql	databas	open	advanc	sourc
(0,8)	(0,3)	(0,2)	(0,8)	(0,1)
(0,41)	(0,8)	(0,8)	(0,14)	(0,2)
	(0,43)	(0,30)	(0,17)	(0,8)
	(0,47)	(0,33)	(0,22)	(0,12)
	(1,32)	(0,36)	(0,26)	(0,13)
		(0,44)	(0,33)	(0,18)
		(0,46)	(0,34)	(0,19)
		(0,56)	(0,35)	(0,20)
		(1,4)	(0,45)	(0,26)
		(1,22)	(0,47)	(0,34)
		(1,34)	(0,48)	(0,35)

9.4 example (2/3)

(0,8) contains all the words -> *match*

postgresql	databas	open	advanc	sourc
(0,8)	(0,3)	(0,2)	(0,8)	(0,1)
(0,41)	(0,8)	(0,8)	(0,14)	(0,2)
	(0,43)	(0,30)	(0,17)	(0,8)
	(0,47)	(0,33)	(0,22)	(0,12)
	(1,32)	(0,36)	(0,26)	(0,13)
		(0,44)	(0,33)	(0,18)
		(0,46)	(0,34)	(0,19)
		(0,56)	(0,35)	(0,20)
		(1,4)	(0,45)	(0,26)
		(1,22)	(0,47)	(0,34)
		(1,34)	(0,48)	(0,35)

9.4 example (3/3)

(0,41) has no match for the “databas” word -> **ignore**

postgresql	databas	open	advanc	sourc
(0,8)	(0,3)	(0,2)	(0,8)	(0,1)
(0,41)	(0,8)	(0,8)	(0,14)	(0,2)
	*(0,43)	(0,30)	(0,17)	(0,8)
	(0,47)	(0,33)	(0,22)	(0,12)
	(1,32)	(0,36)	(0,26)	(0,13)
		(0,44)	(0,33)	(0,18)
		(0,46)	(0,34)	(0,19)
		(0,56)	(0,35)	(0,20)
		(1,4)	(0,45)	(0,26)
		(1,22)	(0,47)	(0,34)
		(1,34)	(0,48)	(0,35)

Tri-consistent

How does the system know that all the items have to match?

```
select plainto_tsquery(  
    'an advanced PostgreSQL open source database');  
                plainto_tsquery  
-----  
'postgresql' & 'advanc' & 'open' & 'sourc' & 'databas'  
(1 row)
```

- ▶ New support function, **tri-consistent** that an operator class can define.
- ▶ Takes tri-state input for each item: TRUE, FALSE, MAYBE

Multi-key searches

```
SELECT * FROM foo
WHERE col @@ plainto_tsquery('PostgreSQL')
AND col @@ plainto('open source database');
```

- ▶ When multiple search conditions are ANDed in the query, they must all match

Part 3: Misc improvements

In addition to the two big improvements, the code went through a lot of refactoring

- ▶ made crash recovery more robust

The future

- ▶ Alexander submitted a third patch that was not accepted
 - ▶ store additional information per item
 - ▶ speeds up certain queries, but makes indexes much larger
- ▶ Planner doesn't know that the “rare & frequent” optimization
- ▶ Various micro-optimizations:
 - ▶ Represent ItemPointers as 64-bit integers for fast comparisons
 - ▶ build a truth-table of the “consistent” function, to avoid function call overhead.
- ▶ VODKA?

Final GIN tip

GIN indexes are efficient at storing duplicates

- Use a GIN index using *btree_gin* extension for status-fields etc.

```
postgres=# \di+
```

List of relations

Schema	Name	...	Size	...
public	numbers_btree		214 MB	
public	numbers_gin		11 MB	

(2 rows)

Questions?