GIN in 9.4 and further

Heikki Linnakangas, Alexander Korotkov, Oleg Bartunov

May 23, 2014

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   |
```

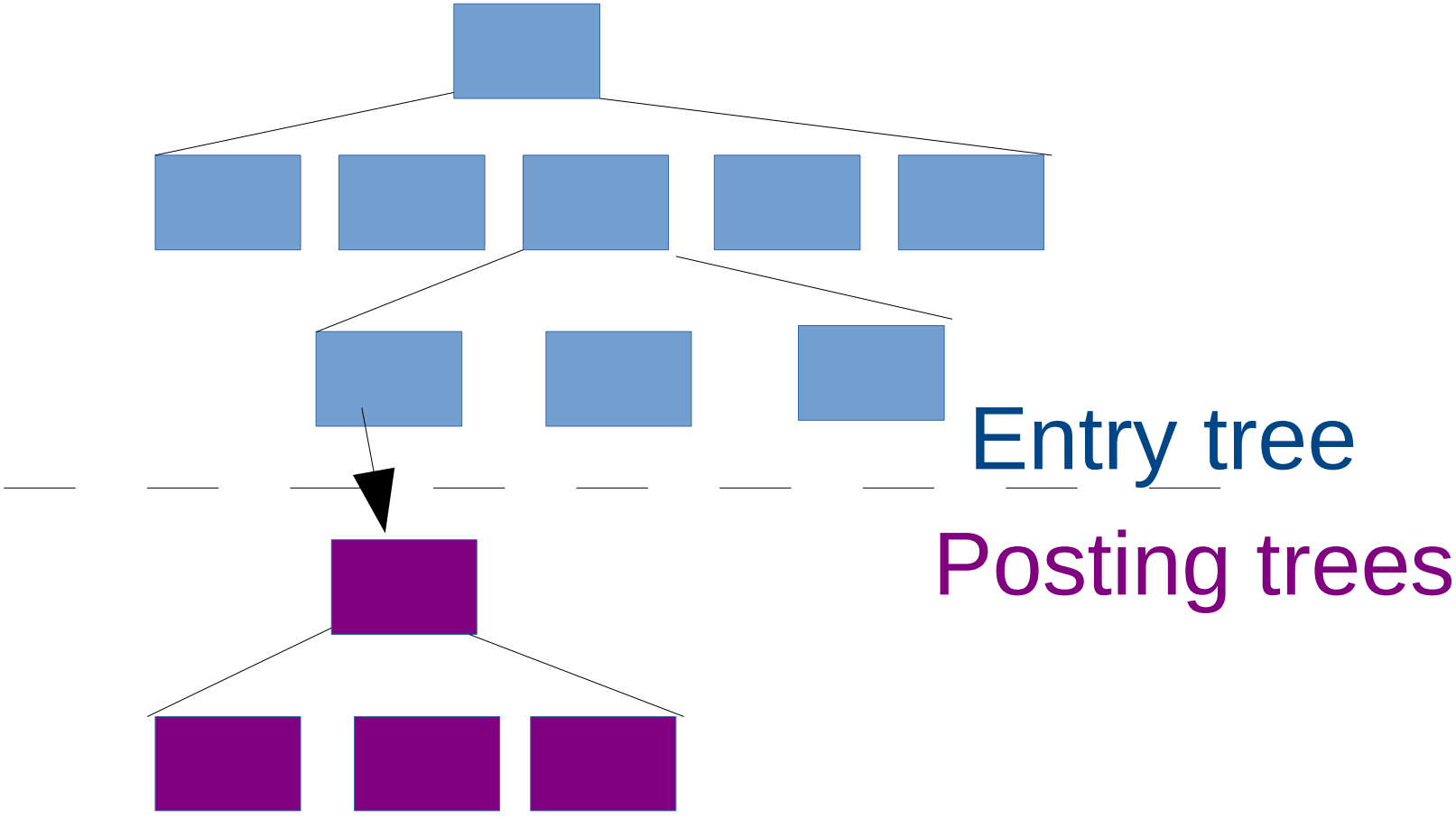
9.4:

```
 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.



GIN structure

- 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 (block, offset)

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

0XXXXXXX
1XXXXXXX 0XXXXYYY
1XXXXXXX 1XXXXYYY 0YYYYYYY
1XXXXXXX 1XXXXYYY 1YYYYYYY 0YYYYYYY
1XXXXXXX 1XXXXYYY 1YYYYYYY 1YYYYYYY 0YYYYYYY
1XXXXXXX 1XXXXYYY 1YYYYYYY 1YYYYYYY 1YYYYYYY
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**

Example query

```
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'
)
```

Combine the matches (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)

Combine the matches (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)		(1,24)		(0,35)

Combine the matches (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)		(1,24)		(0,35)

Combine the matches (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)

Combine the matches (4/4)

(0,8) contains all the words -> *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)
		(1,24)	(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)
		(1,24)	(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)
		(1,24)	(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 (like B-tree)

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.
- Substitute for bitmap indexes

```
postgres=# \di+
                    List of relations
 Schema |      Name       | ... |  Size   | ...
--------+-----------------+-----+---------+-----
 public | numbers_btree   |     | 214 MB  |
 public | numbers_gin     |     | 11 MB   |
(2 rows)
```

Questions?