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;

HEAP IS SLOW
400 ms !

```
 Limit  (cost=8087.40..8087.41 rows=3 width=282) (actual time=433.750..433.752 rows=
   ->  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, '''titl'''::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 @@ '''titl'''::tsquery)
               ->  Bitmap Index Scan on ti2_index  (cost=0.00..517.69 rows=47692 wi
(actual time=13.736..13.736 rows=47855 loops=1)
                     Index Cond: (text_vector @@ '''titl'''::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=28
(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
(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

ItemPointer

```
typedef struct ItemPointerData
{
   BlockIdData ip_blkid;
   OffsetNumber ip_posid;
}

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

6 bytes

Compressed storage

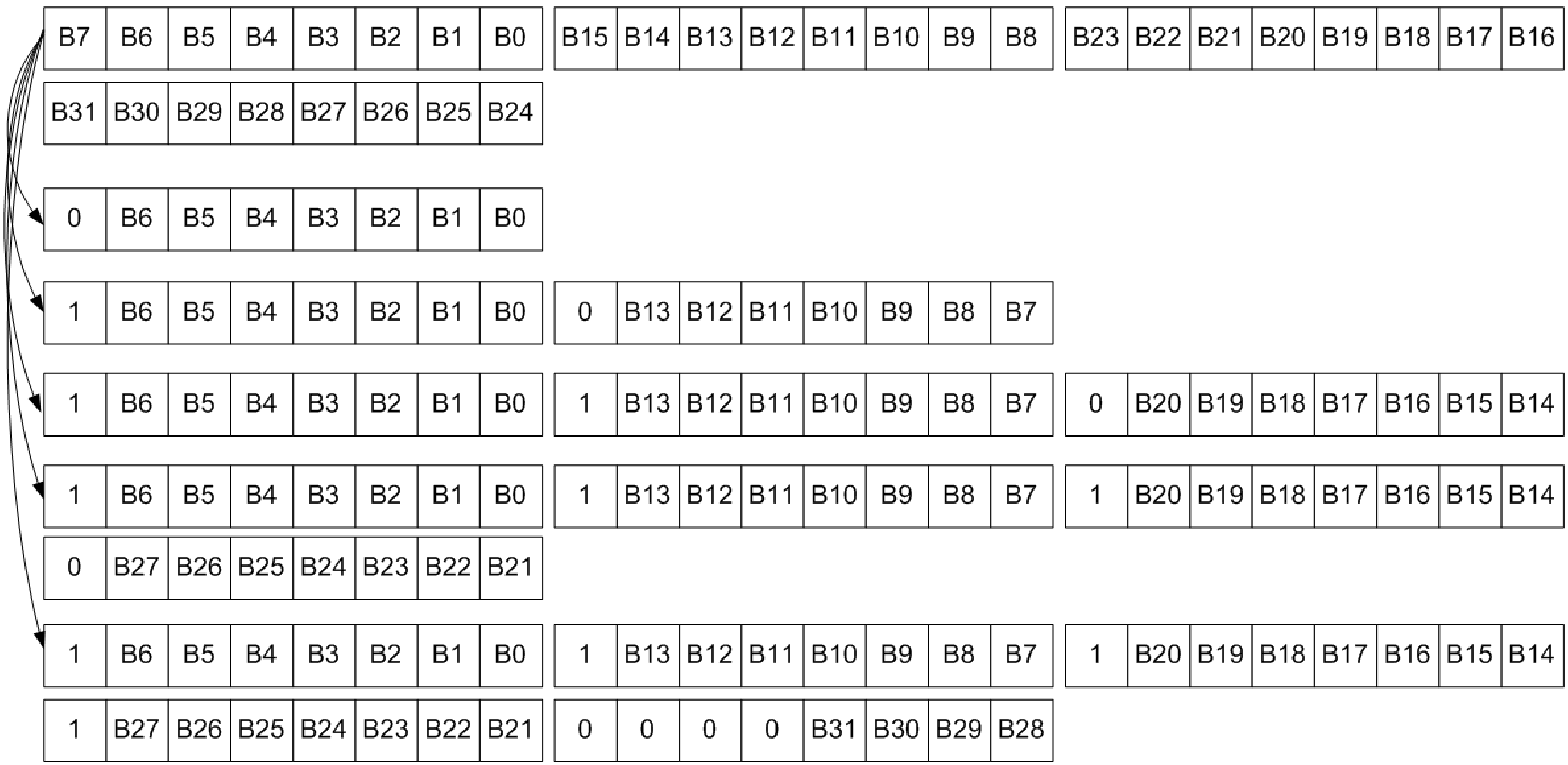
What we have:

- Offset is typically low
- Block number is ascending

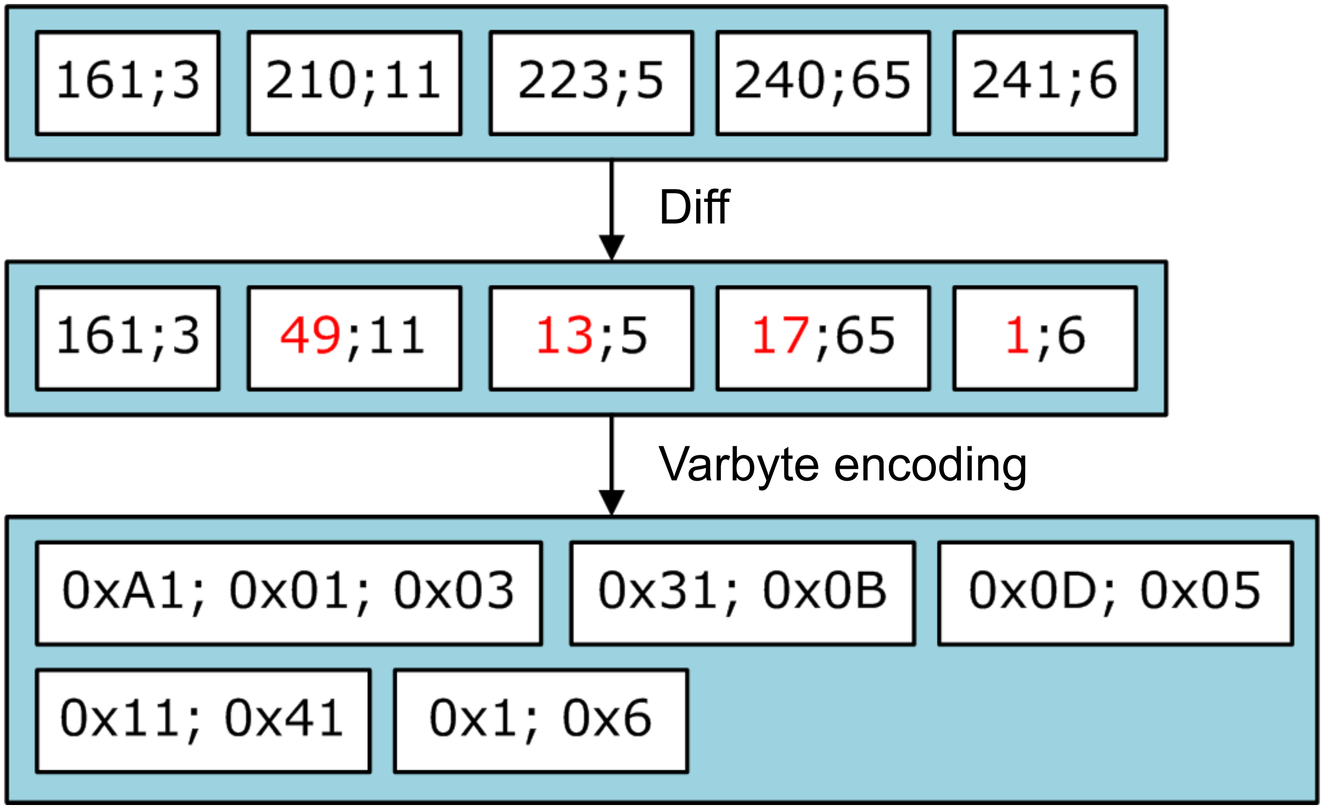
What to do:

- Use var-byte encoding
- Store increments for block numbers

Varbyte compression



Compressed storage



Tests

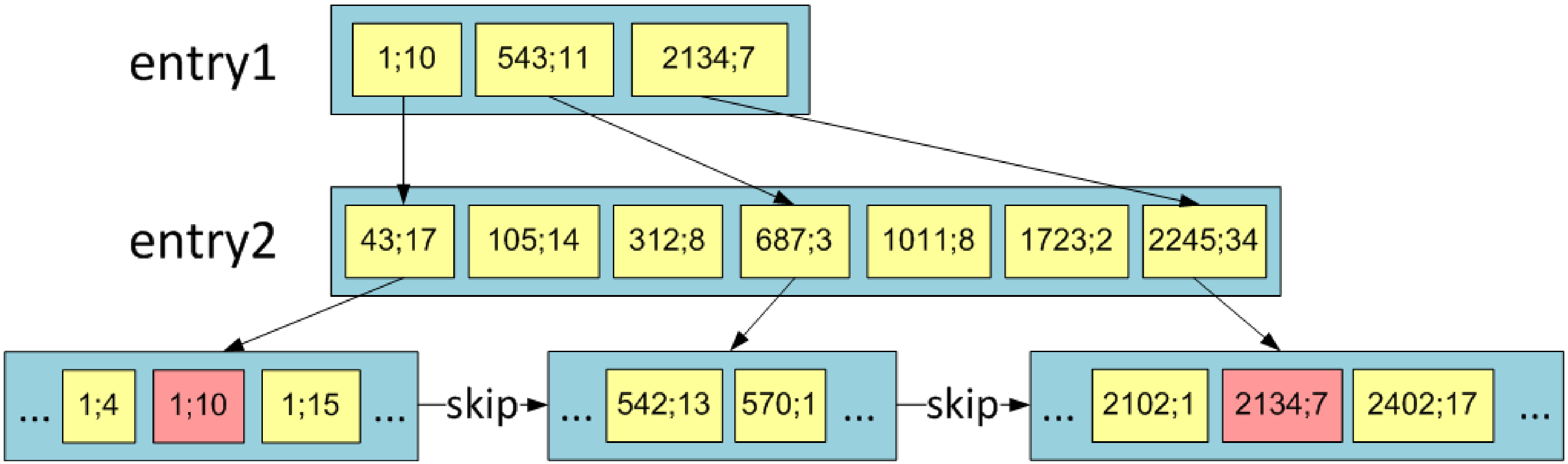
Dataset: mailing lists archives

976488 messages of 1300 characters average length

Parameter	master	patched
Index build time	110 s	105 s
Initial index size	844 MB	400 MB
24K queries execution	1521 s	1447 s
Whole index update time	318 s	317 s
Index size after updates	1521 MB	683 MB
24K queries execution after updates	1557	1585

Fast scan: idea

entry1 && entry2



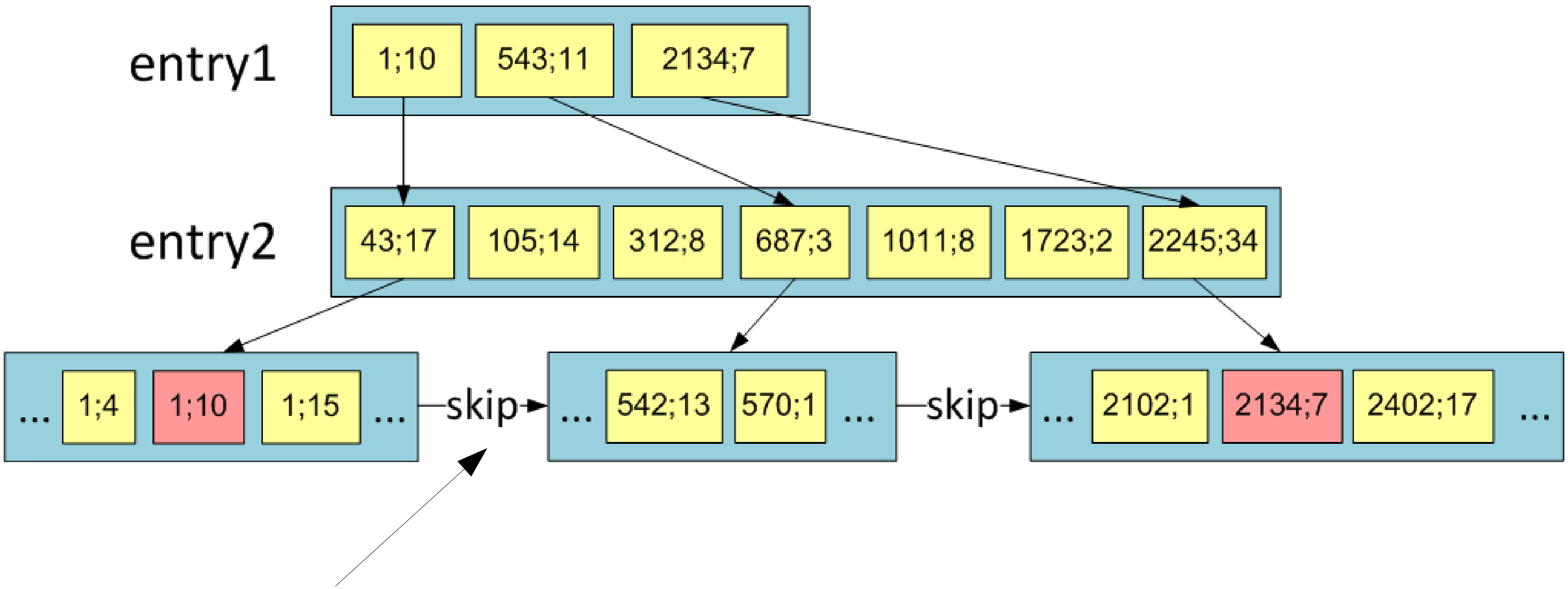
Visiting parts of 3 pages instead of 7

Fast scan interface

New consistent method using tri-state logic:

- true
- false
- unknown

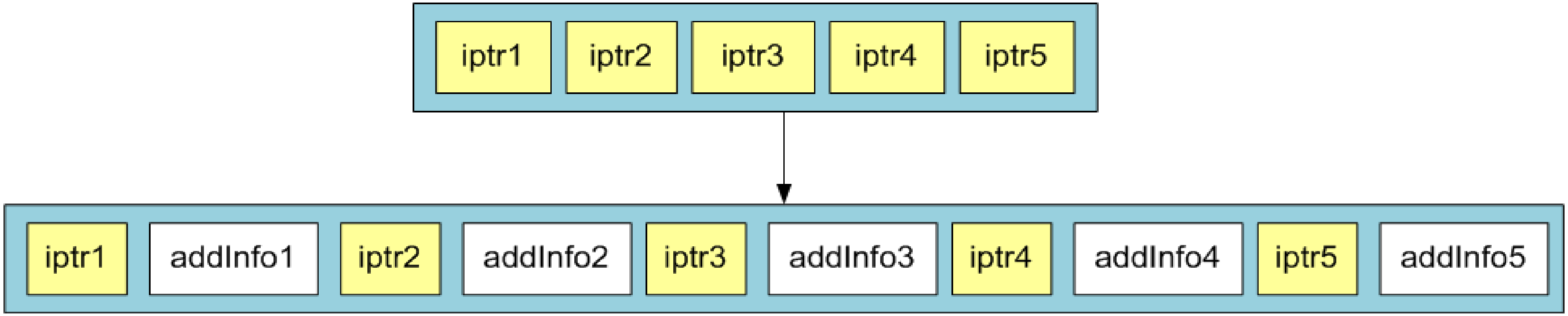
Fast scan interface



Can actually we skip these?

If consistent([false, unknown]) = false then we really can.

Store additional information



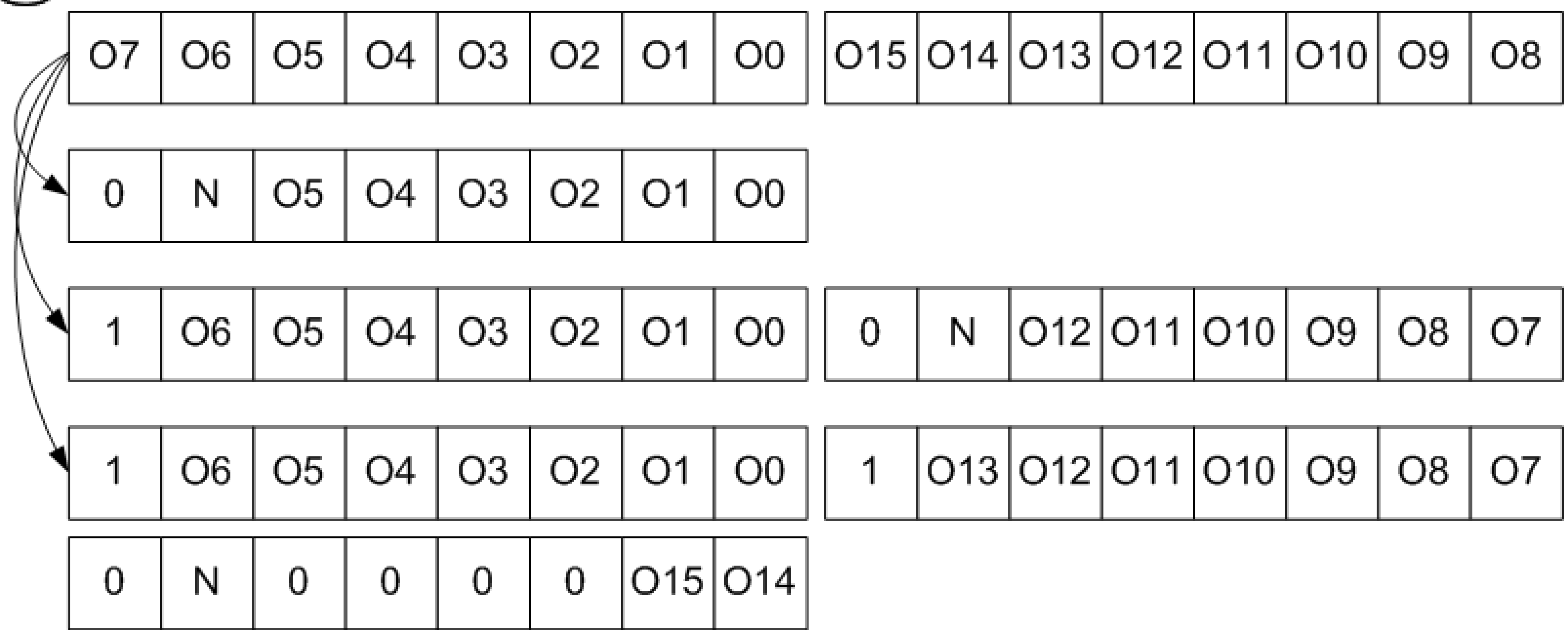
WordEntryPos

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

typedef uint16 WordEntryPos;
```

2 bytes

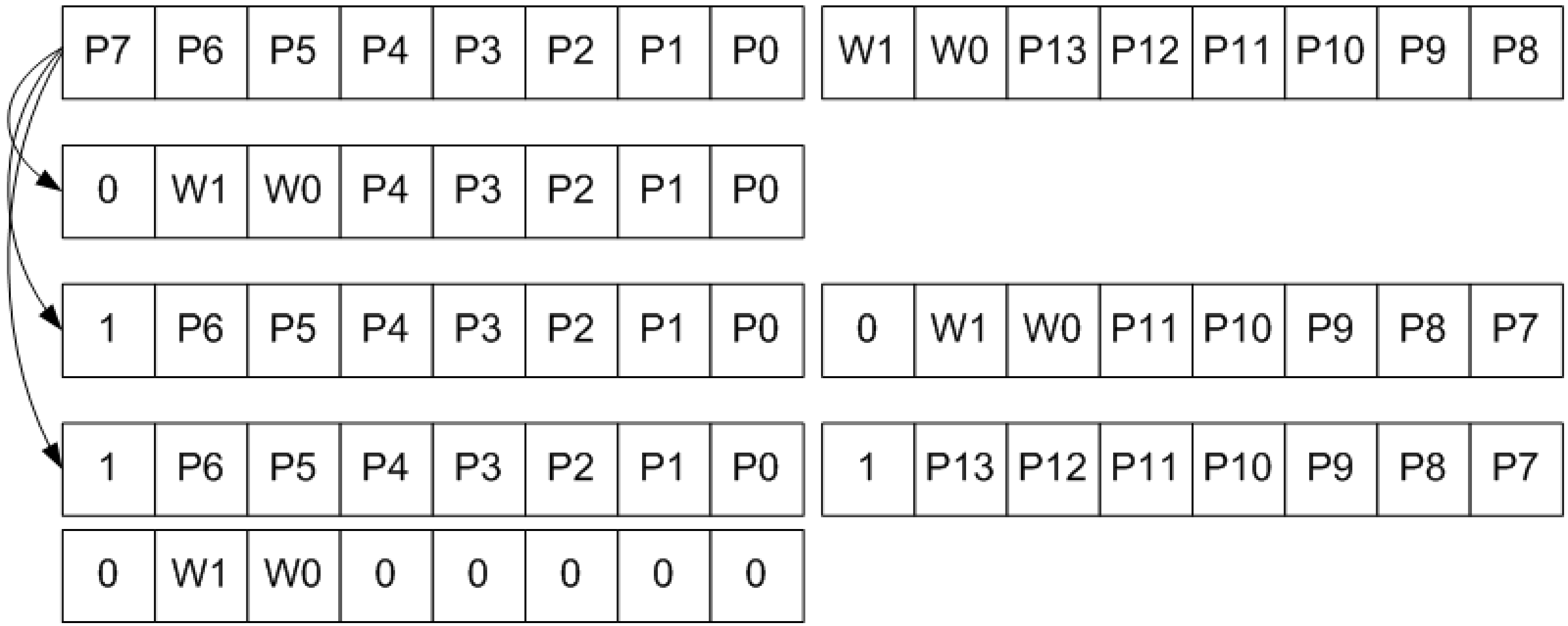
OffsetNumber compression



O0-O15 – OffsetNumber bits

N – Additional information NULL bit

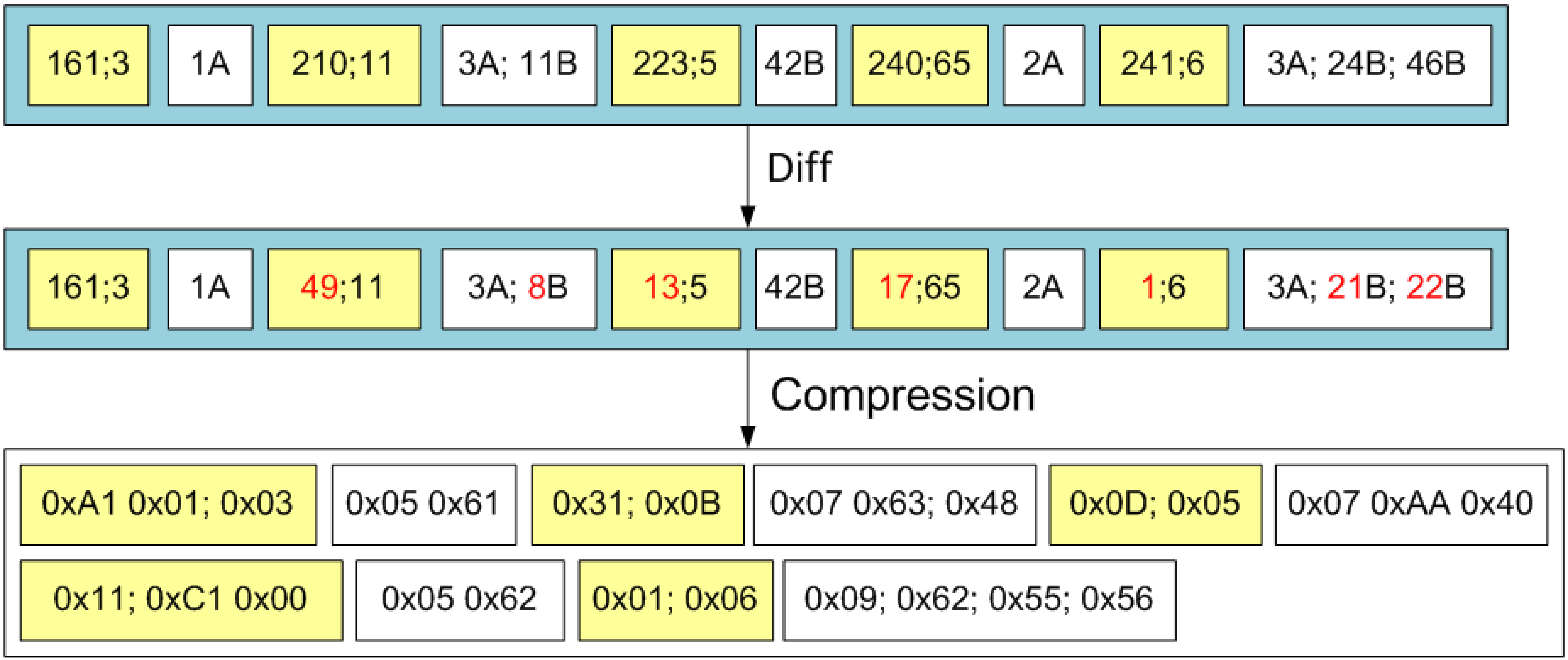
WordEntryPos compression



P0-P13 – position bits

W0,W1 – weight bits

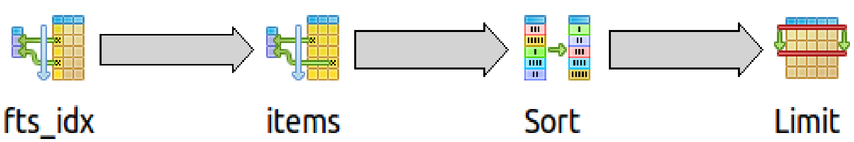
Example


161;3
1A
210;11
3A; 11B
223;5
42B
240;65
2A
241;6
3A; 24B; 46B
Diff
161;3
1A
49;11
3A; 8B
13;5
42B
17;65
2A
1;6
3A; 21B; 22B
Compression
0xA1 0x01; 0x03
0x05 0x61
0x31; 0x0B
0x07 0x63; 0x48
0x0D; 0x05
0x07 0xAA 0x40
0x11; 0xC1 0x00
0x05 0x62
0x01; 0x06
0x09; 0x62; 0x55; 0x56

ORDER BY using index

Before

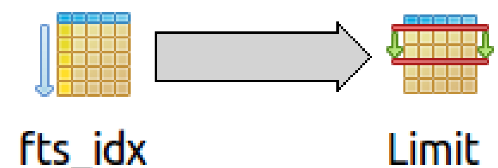
SELECT itemid, title
FROM items
WHERE fts @@ to_tsquery('english', 'query')
ORDER BY
ts_rank(fts, to_tsquery('english', 'query')) DESC
LIMIT 10;



Ranking and sorting are outside the fulltext index

After

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



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
)
```

calcRank

```
float8 calcRank
(
    bool check[],
    StrategyNumber n,
    Datum query,
    int32 nkeys,
    Pointer extra_data[],
    bool *recheck,
    Datum queryKeys[],
    bool nullFlags[],
    Datum addInfo[],
    bool addInfoIsNull[]
)
```

???joinAddInfo???

```
Datum joinAddInfo
(
   Datum addInfo[]
)
```

Example: frequent entry (30%)

Before:

node type	count	sum of times	% of query
Bitmap Heap Scan	1	367.687 ms	94.6 %
Bitmap Index Scan	1	6.570 ms	1.7 %
Limit	1	0.001 ms	0.0 %
Sort	1	14.465 ms	3.7 %

388 ms

After:

node type	count	sum of times	% of query
Index Scan	1	13.346 ms	100.0 %
Limit	1	0.001 ms	0.0 %

13 ms

Example: rare entry (0.08%)

Before:

node type	count	sum of times	% of query
Bitmap Heap Scan	1	0.959 ms	93.4 %
Bitmap Index Scan	1	0.027 ms	2.6 %
Limit	1	0.001 ms	0.1 %
Sort	1	0.040 ms	3.9 %

1.1 ms

After:

node type	count	sum of times	% of query
Index Scan	1	0.052 ms	98.1 %
Limit	1	0.001 ms	1.9 %

0.07 ms

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

Before:

node type	count	sum of times	% of query
Bitmap Heap Scan	1	1.547 ms	23.0 %
Bitmap Index Scan	1	5.151 ms	76.7 %
Limit	1	0.000 ms	0.0 %
Sort	1	0.022 ms	0.3 %

6.7 ms

After:

node type	count	sum of times	% of query
Index Scan	1	0.998 ms	100.0 %
Limit	1	0.000 ms	0.0 %

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 expresson

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)
```

After

```
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 taked one round of review by Heikki Linnakangas
- Compression and planner optimization are now on commitfest
- Other patches are under reworking

Thanks for attention!