PostgresPro's contribution to PostgreSQL's performance

Alexander Korotkov

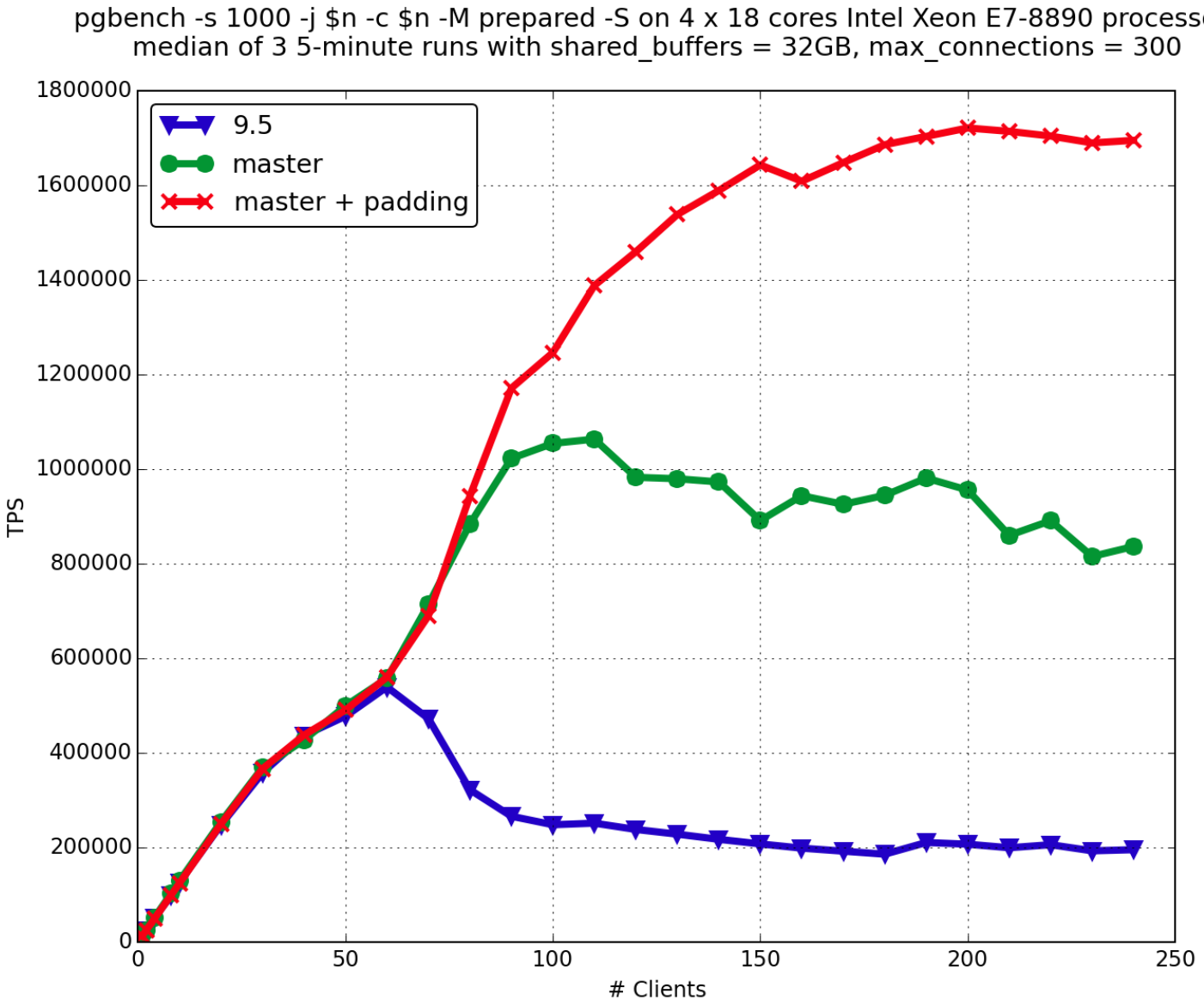
Postgres Professional

2017

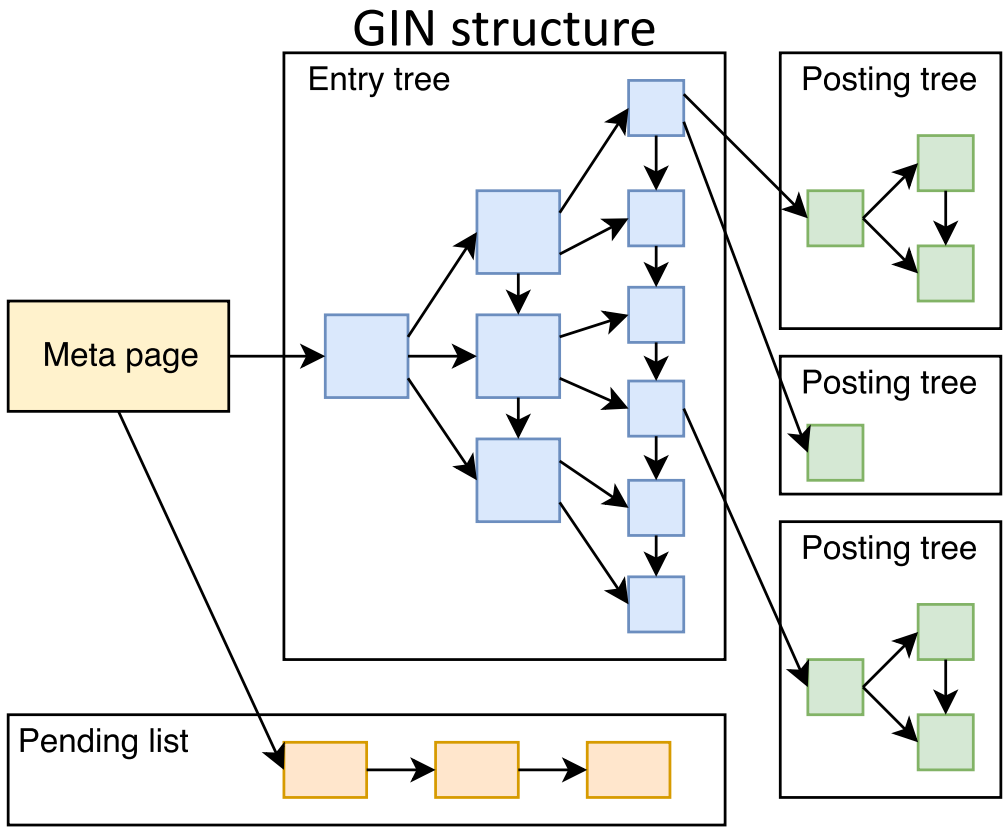
Pin/UnpinBuffer optimization (1/2)

- Spinlock is implemented in assembly. When no implementation for particular processor, then using UNIX semaphore.
- Since 9.5 PostgreSQL supports atomic instructions. Compiler builtins are used or assembly implementations. If none of them are present, then spinlocks are used.
- Since 9.5 LWLocks are implemented using atomic state variable. Before 9.5 spinlock was used.
- Since 9.6 Pin/UnpinBuffer are implemented using atomic state variable. Before 9.6 spinlock was used.

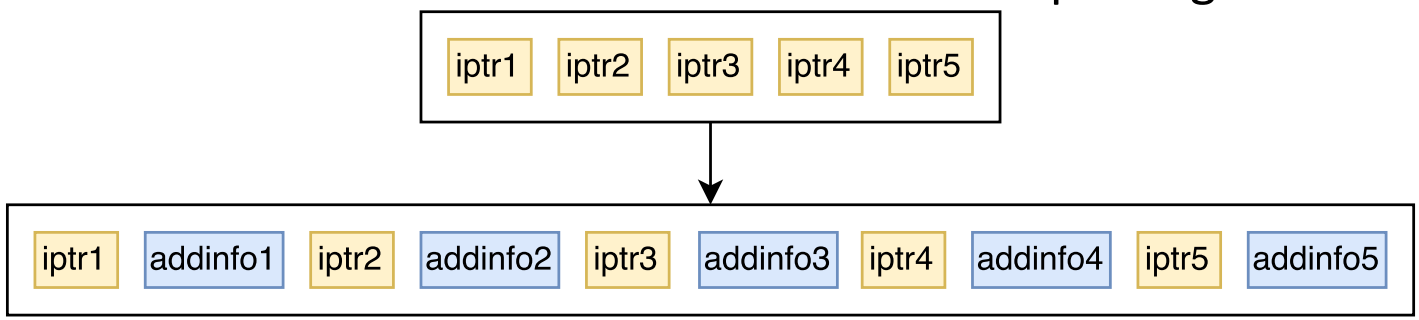
Pin/UnpinBuffer optimization (2/2)



From GIN to RUM



RUM introduces additional information to posting lists.



How did things begin in 2012?

From:	Alexander Korotkov <aekorotkov(at)gmail(dot)com>
To:	pgsql-hackers <pgsql-hackers(at)postgresql(dot)org>
Subject:	WIP: store additional info in GIN index
Date:	2012-11-18 21:54:53
Message-ID:	CAPpHfdtSt47PpRQBK6OawHePLJk8PF-wNhswaUpre7_+cc_kmA@mail.gmail.com (view raw)

Hackers,

Attached patch enables GIN to store additional information with item
pointers in posting lists and trees.
Such additional information could be positions of words, positions of
trigrams, lengths of arrays and so on.
This is the first and most huge patch of serie of GIN improvements which
was presented at PGConf.EU
http://wiki.postgresql.org/images/2/25/Full-text_search_in_PostgreSQL_in_milliseconds-extended-version.pdf

Patch modifies GIN interface as following:
1) Two arguments are added to extractValue
Datum **addInfo, bool **addInfoIsNull
2) Two arguments are added to consistent
Datum addInfo[], bool addInfoIsNull[]
3) New method config is introduced which returns datatype oid of addtional
information (analogy with SP-GiST config method).

Patch completely changes storage in posting lists and leaf pages of posting
trees. It uses varbyte encoding for BlockNumber and OffsetNumber.
BlockNumber are stored incremental in page. Additionally one bit of
OffsetNumber is reserved for additional information NULL flag. To be able
to find position in leaf data page quickly patch introduces small index in
the end of page.

With best regards,
Alexander Korotkov.

RUM use-cases

- Fulltext search with ranking in index (offsets of lexemes inside document as addinfo)
- Fulltext search with alternative ordering (custom column as addinfo)
- Jsonb indexing with positional information (offsets of elements in array as addinfo)
- Positional n-grams (vgrams) for better fuzzy string matching
- Inversed fulltext search (find queries matching given document)
- Inversed regex search (find regexes matching given string)
- Array similarity search using array lengths from the index

GIN: ranking form heap

Search for top-10 (from 222813) postings with «Tom Lane».

```
SELECT subject, ts_rank(fts,plainto_tsquery('english', 'tom lane')) AS rank
FROM pglist WHERE fts @@ plainto_tsquery('english', 'tom lane')
ORDER BY rank DESC LIMIT 10;
                                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------
 Limit (actual time=1374.277..1374.278 rows=10 loops=1)
   ->  Sort (actual time=1374.276..1374.276 rows=10 loops=1)
         Sort Key: (ts_rank(fts, '''tom'' & ''lane'''::tsquery)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  Bitmap Heap Scan on pglist (actual time=98.413..1330.994 rows=22
               Recheck Cond: (fts @@ '''tom'' & ''lane'''::tsquery)
               Heap Blocks: exact=105992
               ->  Bitmap Index Scan on pglist_gin_idx (actual time=65.712..6
rows=222813 loops=1)
                     Index Cond: (fts @@ '''tom'' & ''lane'''::tsquery)
 Planning time: 0.287 ms
 Execution time: 1374.772 ms
(11 rows)
```

RUM: ranking form index

Search for top-10 (from 222813) postings with «Tom Lane».

```
CREATE INDEX pglist_rum_fts_idx on pglist using rum(fts rum_tsvector_ops);

SELECT subject  FROM pglist WHERE fts @@ plainto_tsquery('tom lane')
ORDER BY fts <=> plainto_tsquery('tom lane') LIMIT 10;
                                                QUERY PLAN
-----------------------------------------------------------------------------------------
 Limit (actual time=215.115..215.185 rows=10 loops=1)
   ->  Index Scan using pglist_rum_fts_idx on pglist (actual time=215.113..21
                                                        rows=10 loops=1)
         Index Cond: (fts @@ plainto_tsquery('tom lane'::text))
         Order By: (fts <=> plainto_tsquery('tom lane'::text))
 Planning time: 0.264 ms
 Execution time: 215.833 ms
(6 rows)
```

RUM acceleration **215.833 ms vs 1374.772 ms**.

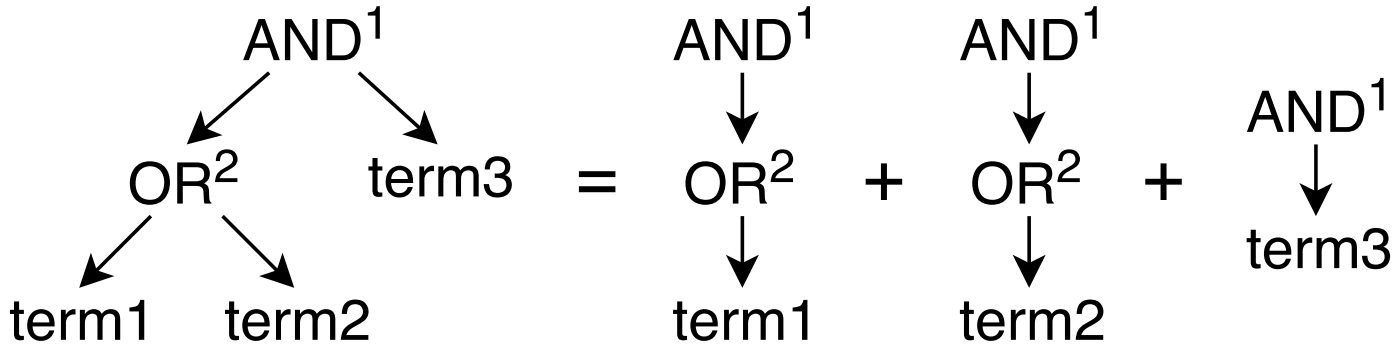
Inverse FTS (FQS)

- Find queries, which match given document
- Automatic text classification

```
SELECT * FROM queries;
                 q                  |  tag
------------------------------------+-------
 'supernova' & 'star'               | sn
 'black'                            | color
 'big' & 'bang' & 'black' & 'hole'  | bang
 'spiral' & 'galaxi'                | shape
 'black' & 'hole'                   | color
(5 rows)

SELECT * FROM queries WHERE
 to_tsvector('black holes never exists before we think about them')
@@ q;
        q          |  tag
-------------------+-------
 'black'           | color
 'black' & 'hole'  | color
(2 rows)
```

RUM index for inverse FTS



- term1: (AND 1, OR 2)
- term2: (AND 1, OR 2)
- term3: (AND 1)

RUM index for inverse FTS

- RUM index supported – store branches of query tree in addinfo.
- Find queries for the first message in postgres mailing lists.

```
create index pg_query_rum_idx on pg_query using rum(q);
select q from pg_query pgq, pglist where q @@ pglist.fts and pglist.id=1;
                                        QUERY PLAN
-----------------------------------------------------------------------------
 Nested Loop (actual time=0.719..0.721 rows=2 loops=1)
   ->  Index Scan using pglist_id_idx on pglist
(actual time=0.013..0.013 rows=1 loops=1)
         Index Cond: (id = 1)
   ->  Bitmap Heap Scan on pg_query pgq
(actual time=0.702..0.704 rows=2 loops=1)
         Recheck Cond: (q @@ pglist.fts)
         Heap Blocks: exact=2
         ->  Bitmap Index Scan on pg_query_rum_idx
(actual time=0.699..0.699 rows=2 loops=1)
               Index Cond: (q @@ pglist.fts)
 Planning time: 0.212 ms
 Execution time: 0.759 ms
(10 rows)
```

RUM status

- Extension for PostgreSQL 9.6+ (thanks to CREATE ACCESS METHOD & generic WAL)
- Available at `https://github.com/postgrespro/rum`.
- Available at PGDG yum & apt repositories.
- Quite stable, passed test of time.

Bitmap index-only count

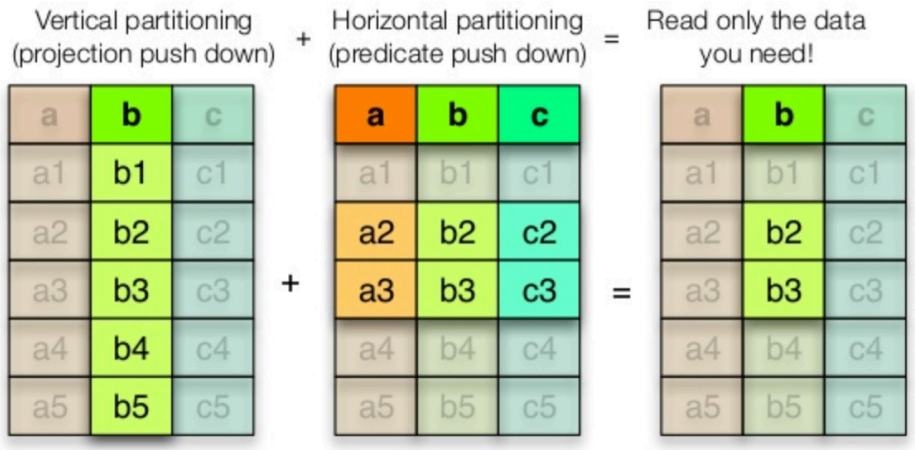
Before:

```
# explain (analyze, buffers) select count(*) from dblp_titles
  where to_tsvector('english', s) @@ to_tsquery('english', 'system');
 Aggregate  (cost=25000.21..25000.22 rows=1 width=8) (actual time=134.394..134.394 rows=1 loops=1)
   Buffers:  shared hit=20169
   ->  Bitmap Heap Scan on dblp_titles  (cost=129.92..24968.63 rows=12634 width=0) (actual time=24.468..120.878 row
         Recheck Cond: (to_tsvector('english'::regconfig, s) @@ '''system'''::tsquery)
         Heap Blocks: exact=20139
         Buffers: shared hit=20169
         ->  Bitmap Index Scan on dblp_titles_fts_idx  (cost=0.00..126.76 rows=12634 width=0) (actual time=20.859..
               Index Cond: (to_tsvector('english'::regconfig, s) @@ '''system'''::tsquery)
               Buffers: shared hit=2 read=28
 Planning time: 0.098 ms
 Execution time:  134.783 ms
```

After:

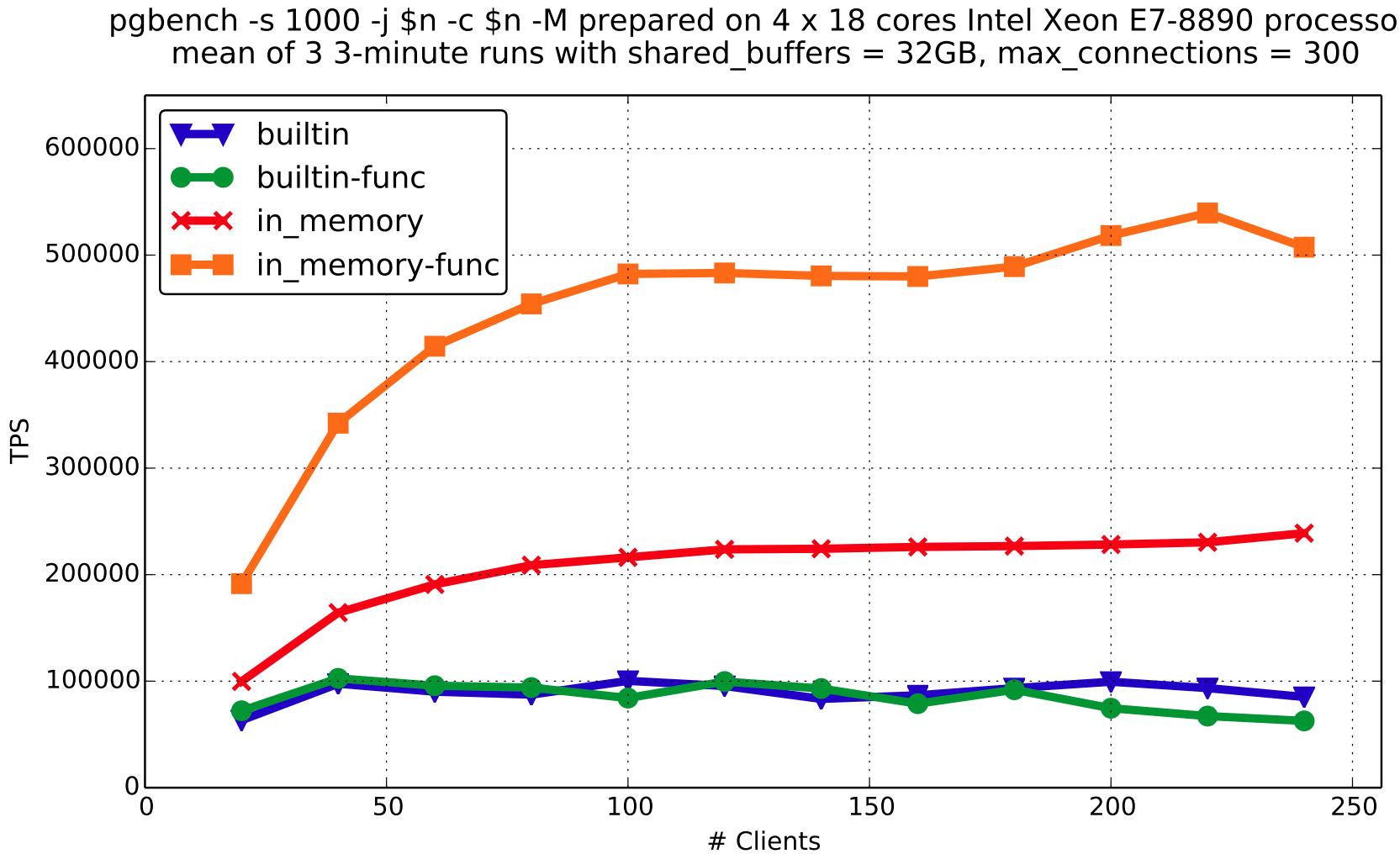
```
# explain (analyze, buffers) select count(*) from dblp_titles
  where to_tsvector('english', s) @@ to_tsquery('english', 'system');
 Aggregate  (cost=71618.52..71618.53 rows=1 width=8) (actual time=56.594..56.594 rows=1 loops=1)
   Buffers:  shared hit=31
   ->  Bitmap Heap Scan on dblp_titles  (cost=1553.15..71201.37 rows=166858 width=0) (actual time=21.003..41.196 ro
         Recheck Cond: (to_tsvector('english'::regconfig, s) @@ '''system'''::tsquery)
         Heap Blocks: exact=20139
         Buffers: shared hit=31
         ->  Bitmap Index Scan on dblp_titles_fts_idx  (cost=0.00..1511.43 rows=166858 width=0) (actual time=18.103
               Index Cond: (to_tsvector('english'::regconfig, s) @@ '''system'''::tsquery)
               Buffers: shared hit=30
 Planning time: 0.235 ms
 Execution time:  56.707 ms
```

vops – parquet layout



Query	Seq. exec., msec	Parallel exec. (msec)
Original Q1 for lineitem	38028	10997
Original Q1 for lineitem_projection	33872	9656
Vectorized Q1 for vops_lineitem	3372	951
Mixed Q1 for vops_lineitem_projection	1490	396
Original Q6 for lineitem	16796	4110
Original Q6 for lineitem_projection	4279	1171
Vectorized Q6 for vops_lineitem	875	284

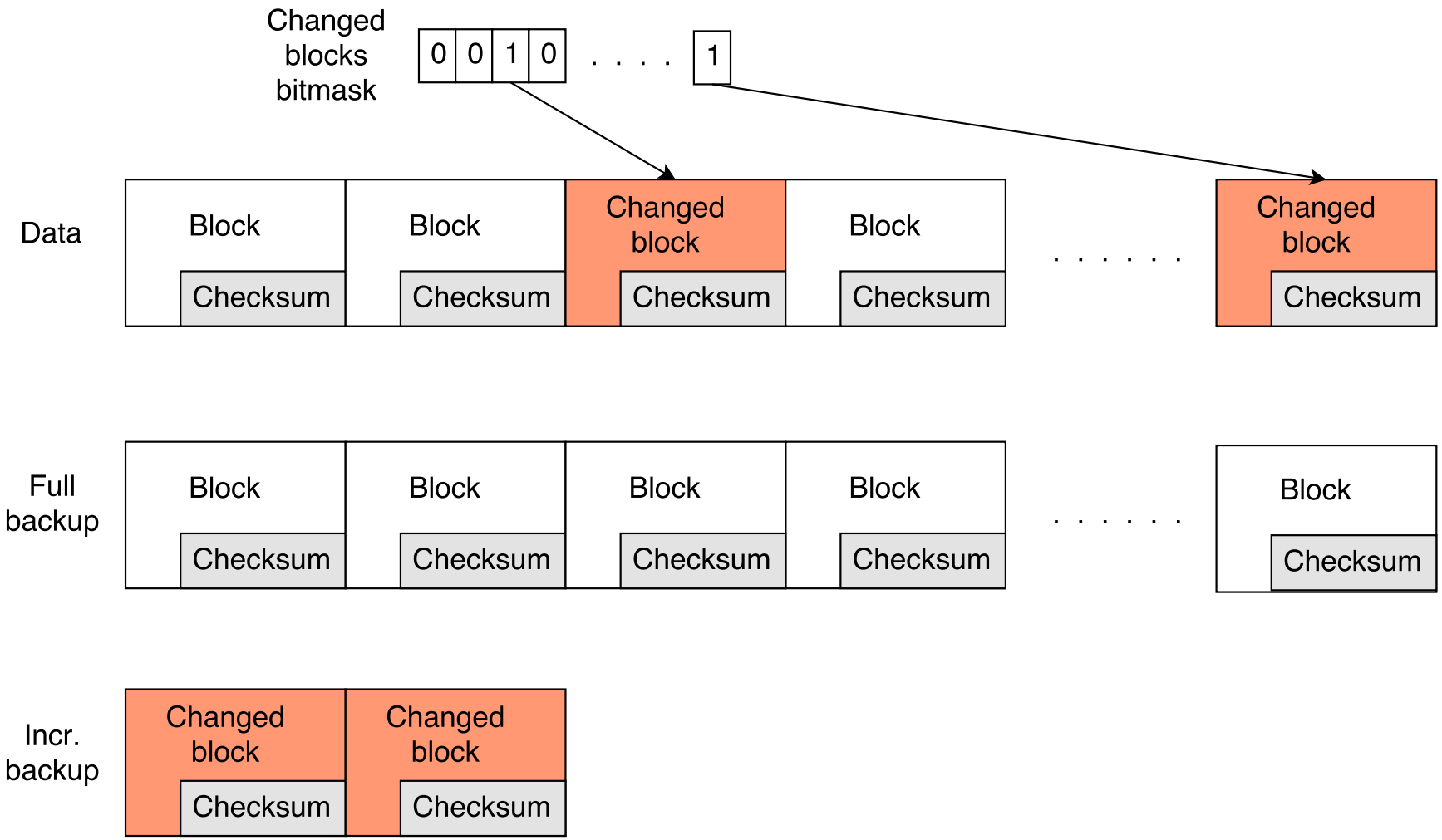
Pluggable storage engines
+ in-memory engine



SP-GiST for boxes
mapping to 4D space

Parameter	heap	GiST	SP-GiST
Build time, sec		68	47
Size, MB	871	663	505
&&, ms	924	23.7	14.7
@>, ms	909	26.9	14.1
<@, ms		12.6	0.3

Incremental backup



pg_probackup

- Available at `https://github.com/postgrespro/pg_probackup`.
- Supports incremental backup with Postgres Pro Enterprise or patched PostgreSQL.
- Various useful features.

Index-only scans for GiST

Before:

```
$ explain analyze select id, some_point, some_point <-> point(500,500)
  from test order by some_point <-> point(500,500) limit 10;
-------------------------------------------------------------------------
 Limit  (cost=0.28..1.09 rows=10 width=20) (actual time=0.128..0.205 rows=10
   ->  Index Scan using tst_idx on test  (cost=0.28..8136.28 rows=100000 widt
         Order By: (some_point <-> '(500,500)'::point)
 Planning time: 0.168 ms
 Execution time: 0.254 ms
```

After:

```
$ explain analyze select id, some_point, some_point <-> point(500,500)
  from test order by some_point <-> point(500,500) limit 10;
-------------------------------------------------------------------------
 Limit  (cost=0.28..1.17 rows=10 width=20) (actual time=0.066..0.107 rows=10
   ->  Index Only Scan using tst_idx on test  (cost=0.28..8908.28 rows=100000
         Order By: (some_point <-> '(500,500)'::point)
         Heap Fetches: 0
 Planning time: 0.114 ms
 Execution time: 0.096 ms
```

KNN-GiST with recheck

PostGIS users can have results sorted by true distance using <-> operator.

No need for hackery approximate queries anymore.

Before:

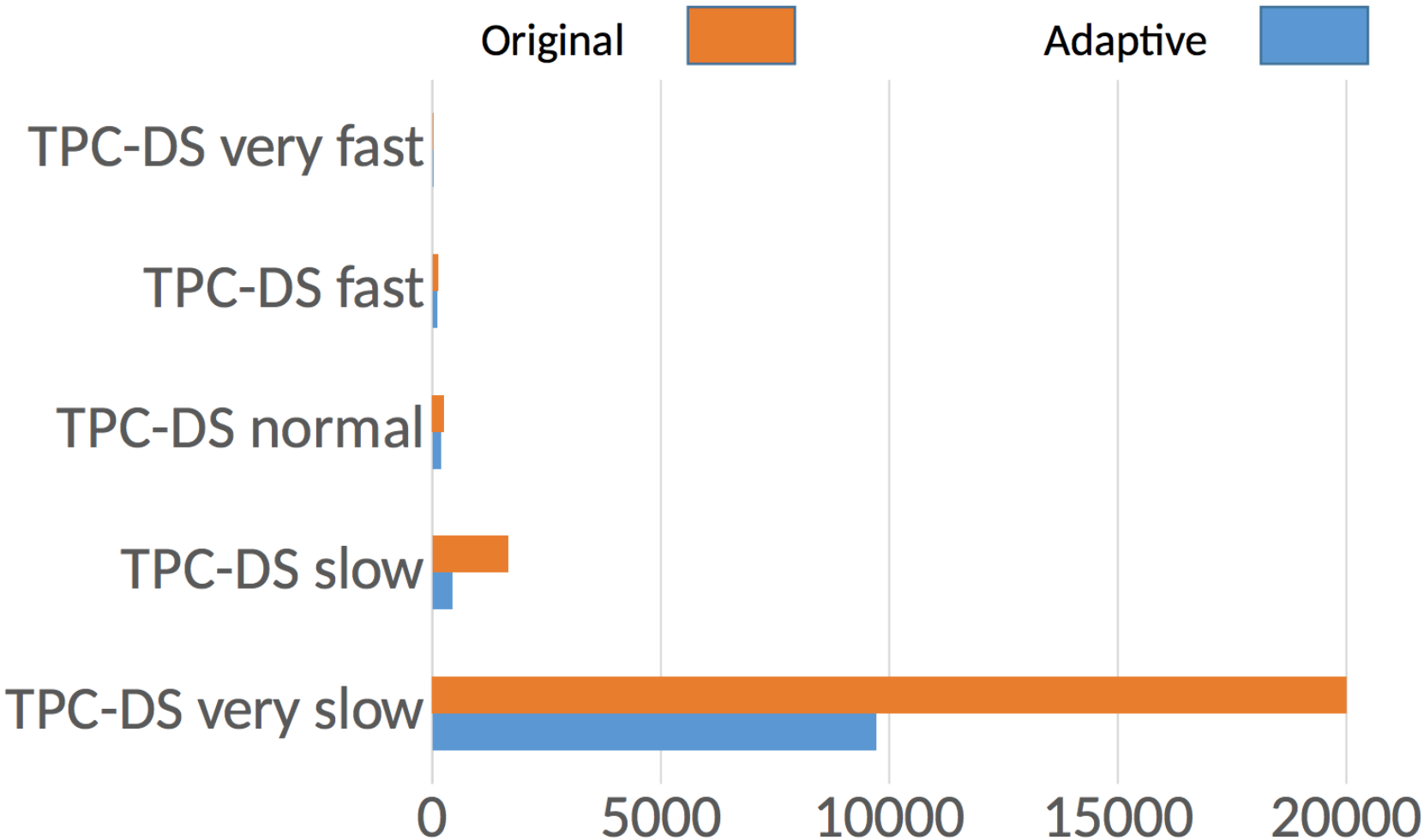
```
WITH closest_candidates AS (
  SELECT streets.gid, streets.name, streets.geom
  FROM nyc_streets streets
  ORDER BY streets.geom <->
    'SRID=26918;POINT(583571.905921312 4506714.34119218)'::geometry
  LIMIT 100)
SELECT gid, name FROM closest_candidates
ORDER BY ST_Distance(geom,
 'SRID=26918;POINT(583571.905921312 4506714.34119218)'::geometry
) LIMIT 1;
```

After:

```
SELECT streets.gid, streets.name, streets.geom
FROM nyc_streets streets
ORDER BY streets.geom <->
  'SRID=26918;POINT(583571.905921312 4506714.34119218)'::geometry
LIMIT 1;
```

Adaptive query optimization

Let database learn on its own mistakes!



Page-level table compression

pgbench -i -s 1000

Configuration	Size (Gb)	Time (sec)
no compression	15.31	92
snappy	5.18	99
lz4	4.12	91
postgres internal lz	3.89	214
lzfse	2.80	1099
zlib (best speed)	2.43	191
zlib (default level)	2.37	284
zstd	1.69	125

Available in Postgres Pro Enterprise.

Covering + unique indexes

Before:

```
CREATE UNIQUE INDEX olduniqueidx ON oldt
USING btree (c1, c2);
CREATE INDEX oldcoveringidx ON oldt
USING btree (c1, c2, c3, c4);
```

After:

```
CREATE UNIQUE INDEX newidx ON newt
USING btree (c1, c2) INCLUDING (c3, c4);
```

One index instead or two!

Available in Postgres Pro, patch is submitted.

Other improvements

- 2PC speedup
- precalculate stable functions
- contrib/bloom
- Use pg_rewind when old master timeline was switched

Postgres PROFESSIONAL

Thank you for attention!