2ndQuadrant®
PostgreSQL

Performance improvements in PostgreSQL 9.5 (and beyond)

Tomas Vondra (tomas@2ndquadrant.com)

pgconf.asia 2016, December 3, Tokyo

PostgreSQL 9.5, 9.6, ...

- many improvements
  - often quite large patches
  - many of them related to performance
- release notes are nice, but ...
  - many changes not mentioned explicitly
  - difficult to get an idea of the impact
- “what's new” talks
  - usually about new features in general
  - this talk is about changes affecting performance

2ndQuadrant® PostgreSQL

What we'll look at?

- PostgreSQL 9.5, 9.6
  - and maybe a bit of 10.0
- only “main” improvements
  - complete “features” (patch series)
  - many additional fixes / improvements (would take hours)
  - no particular order (9.5 → 9.6 → 10.0)
- will try to showcase them
  - demonstrate patch impact on simple examples

PostgreSQL 9.5

2ndQuadrant®
PostgreSQL

Sorting

- sorting is important, often beats other approaches
  - ORDER BY, GROUP BY, CREATE INDEX, ...
- many significant improvements in recent releases (9.5, 9.6, …)

(1) allow sorting by inlined, non-SQL-callable functions

- significant reduction of per-call overhead

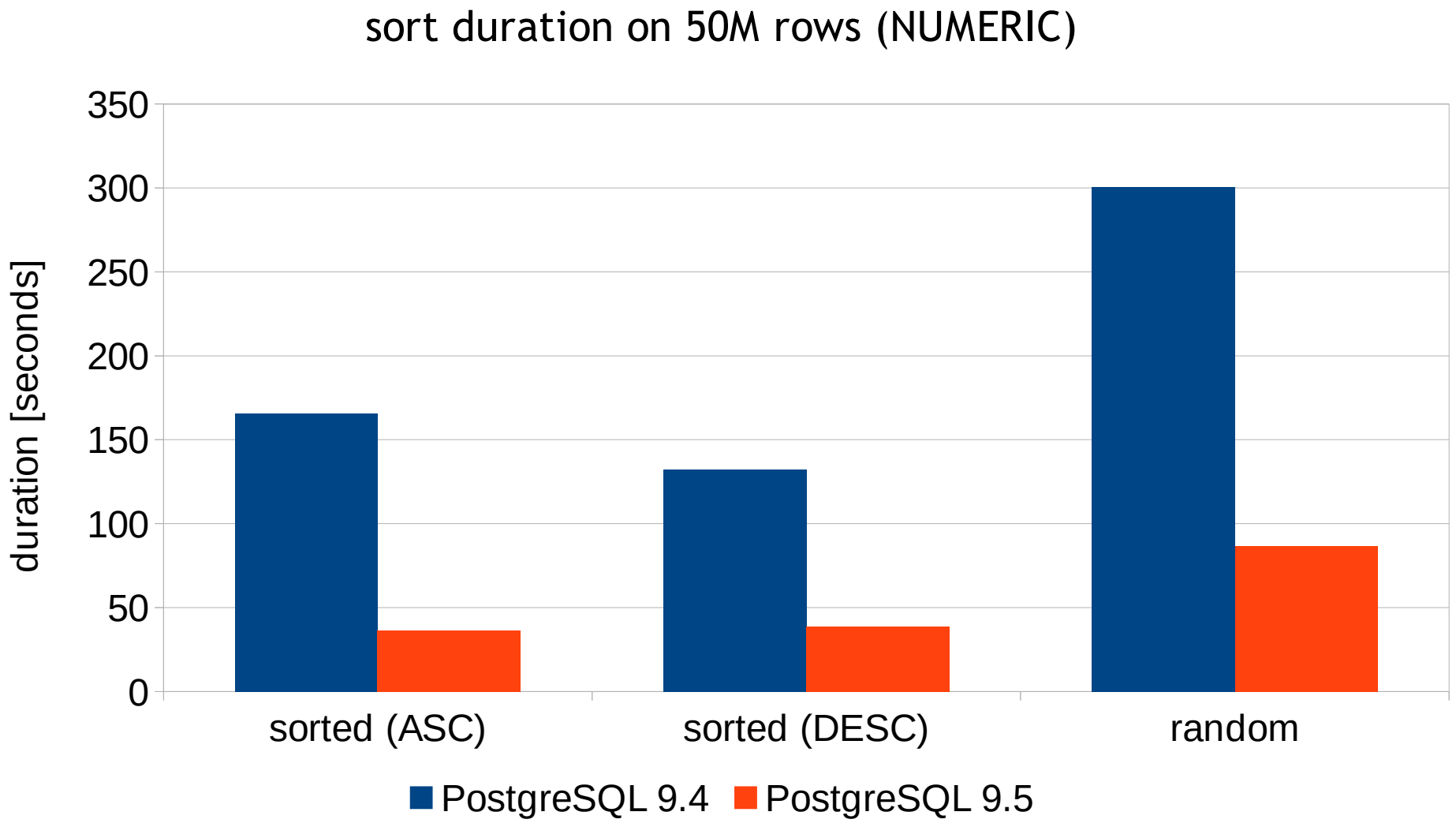
(2) use abbreviated keys for faster sorting

- VARCHAR, TEXT, NUMERIC (but problems with locales)
- does not apply to CHAR values

Sorting

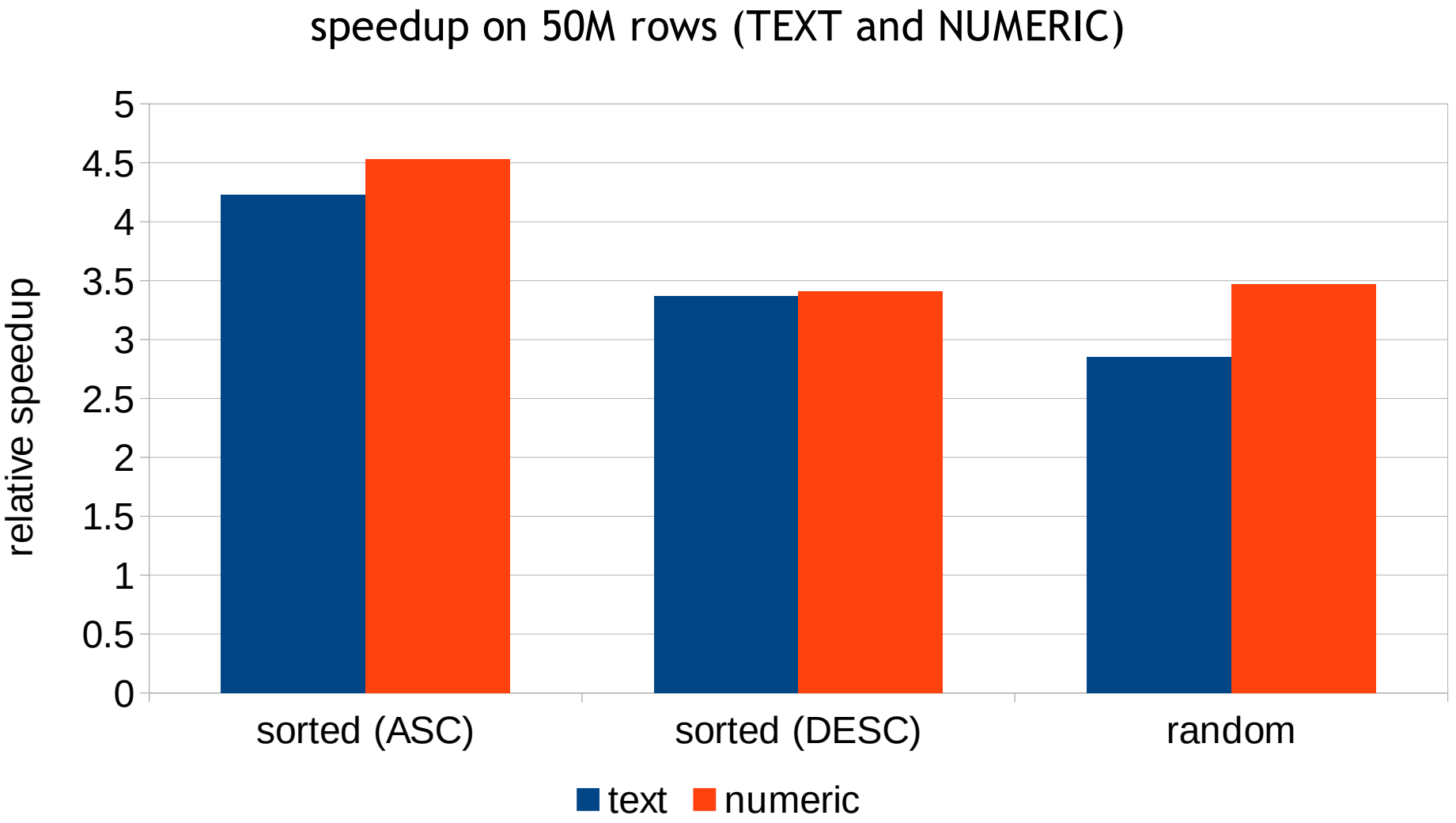
```
/* unsorted table */
CREATE TABLE numeric_random AS
       SELECT random()::numeric AS val
       FROM generate_series(1, 50.000.000);

/* already sorted table */
CREATE TABLE numeric_sorted_asc AS
       SELECT * from numeric_random ORDER BY 1;

/* test queries */
SELECT * FROM numeric_random ORDER BY 1;
SELECT * FROM numeric_sorted_asc ORDER BY 1;
```

Sorting improvements in PostgreSQL 9.5



2ndQuadrant® PostgreSQL

Sorting speedups on PostgreSQL 9.5



2ndQuadrant® PostgreSQL

Hash Joins

(1) reduce NTUP_PER_BUCKET to 1 (from 10)

- shorter chains in buckets (1 tuple on average) => faster lookups

(2) dynamically resize the hash table

- handle under-estimates gracefully
- was trivial to get 100s of tuples in a single bucket

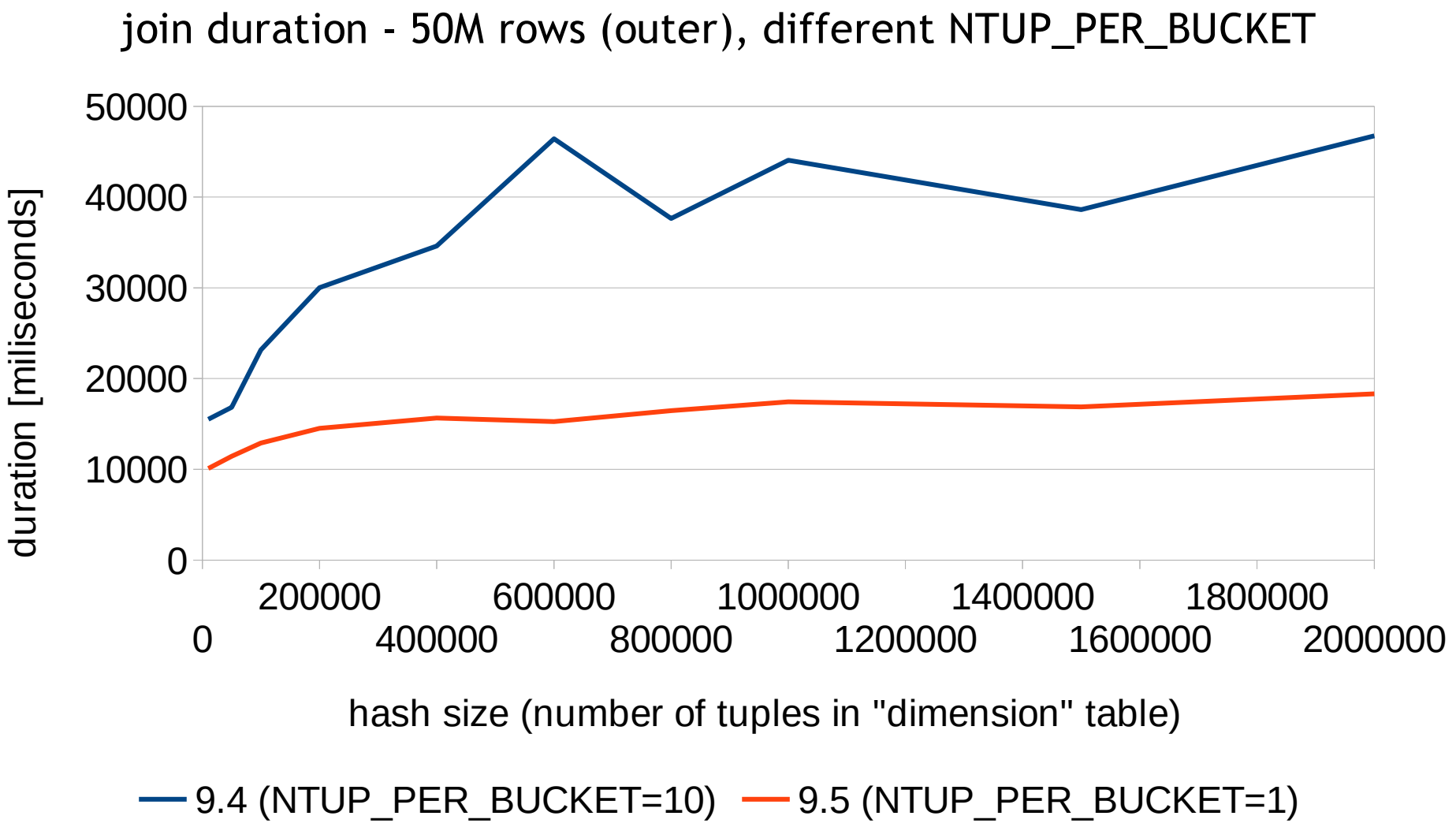
(3) reduce palloc overhead

- dense packing of tuples (trivial local memory allocator)
- significant reduction of palloc overhead (both time and memory)

Hash Joins

```
-- small “dimension” table
CREATE TABLE dimension AS
SELECT i AS id FROM generate_series(1, 100.000) s(i);

-- large “fact” table
CREATE TABLE fact AS
SELECT (1 + mod(i,100.000)) AS dim_id
  FROM generate_series(1, 50.000.000) s(i);

-- test query
SELECT count(*) FROM fact JOIN dim ON (dim_id = id);
```

PostgreSQL 9.5 Hash Join Improvements



2ndQuadrant® PostgreSQL

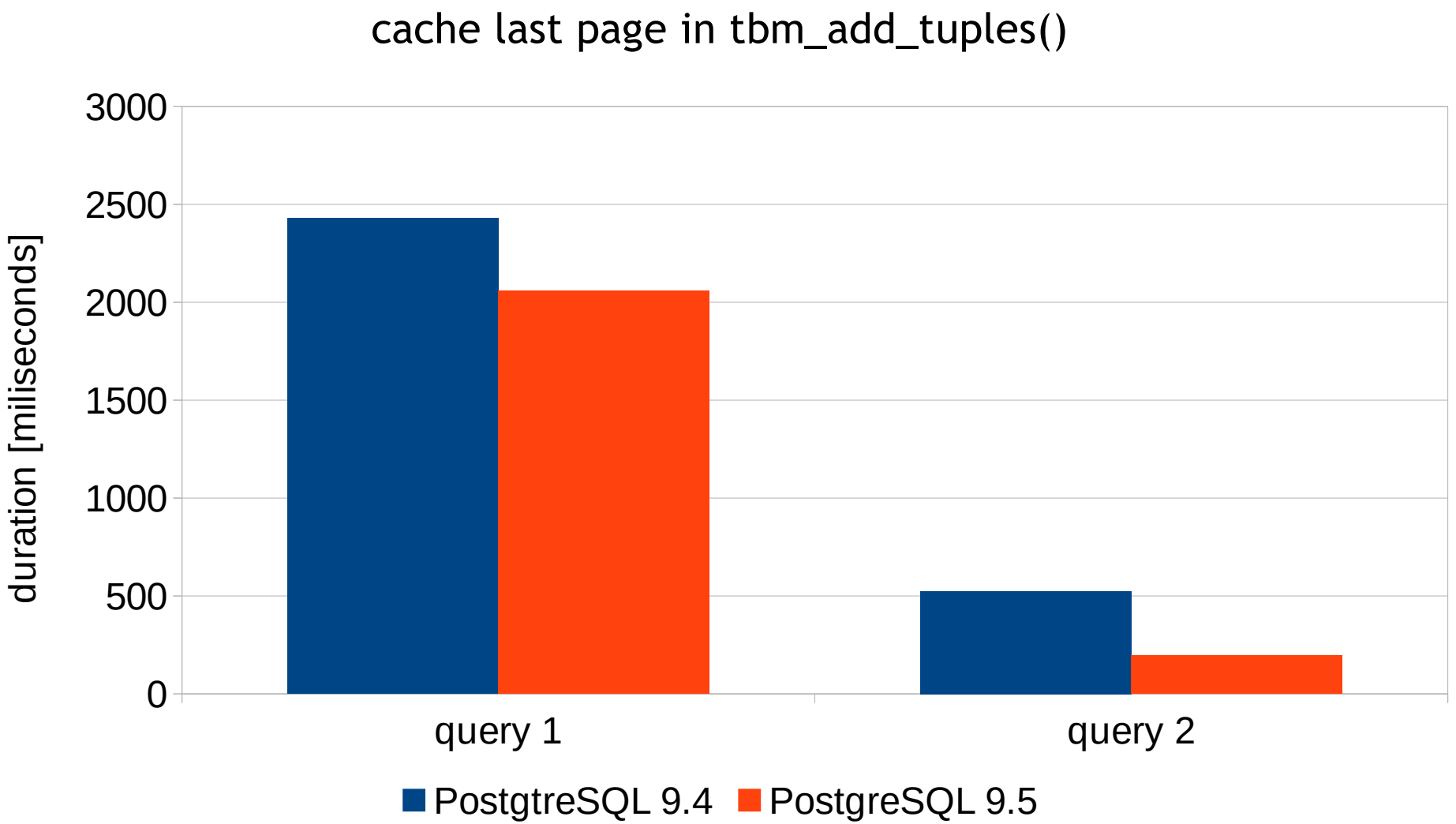
Indexes

- Speed-up Bitmap Index Scan
  - in some cases up to 50% was spent in tbm_add_tuples
  - cache the last accessed page in tbm_add_tuples
- BRIN
  - block range indexes, tracking block summary (e.g. min / max)
  - only bitmap index scans (equality and range queries)
- other improvements
  - avoid copying index tuples when building an index (CREATE INDEX)
  - index-only scans with GiST with more data types (range, inet, btree_gist)

Bitmap build speedup

```
-- table with 5M rows
CREATE EXTENSION btree_gin;
CREATE TABLE t (id int);
INSERT INTO t SELECT (v / 10)
                FROM generate_series(1, 5.000.000) AS v;
CREATE INDEX idx ON t USING gin (id);

-- test queries
SET enable_seqscan  = off;
SELECT * FROM t WHERE id >= 0;
SELECT * FROM t WHERE id >= 100 AND id <= 100;
```

Bitmap build speedup

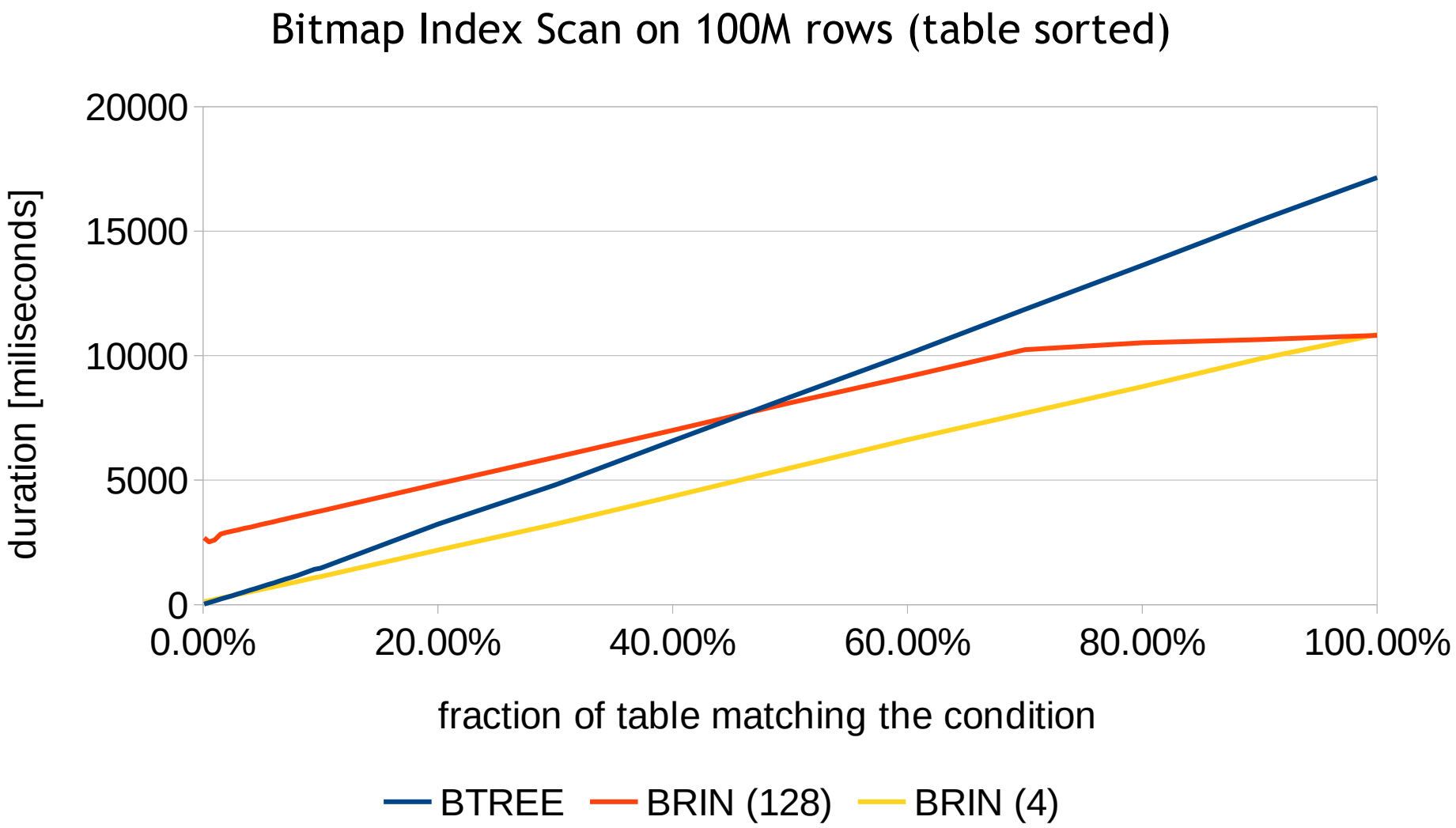


2ndQuadrant®
PostgreSQL

BRIN Indexes

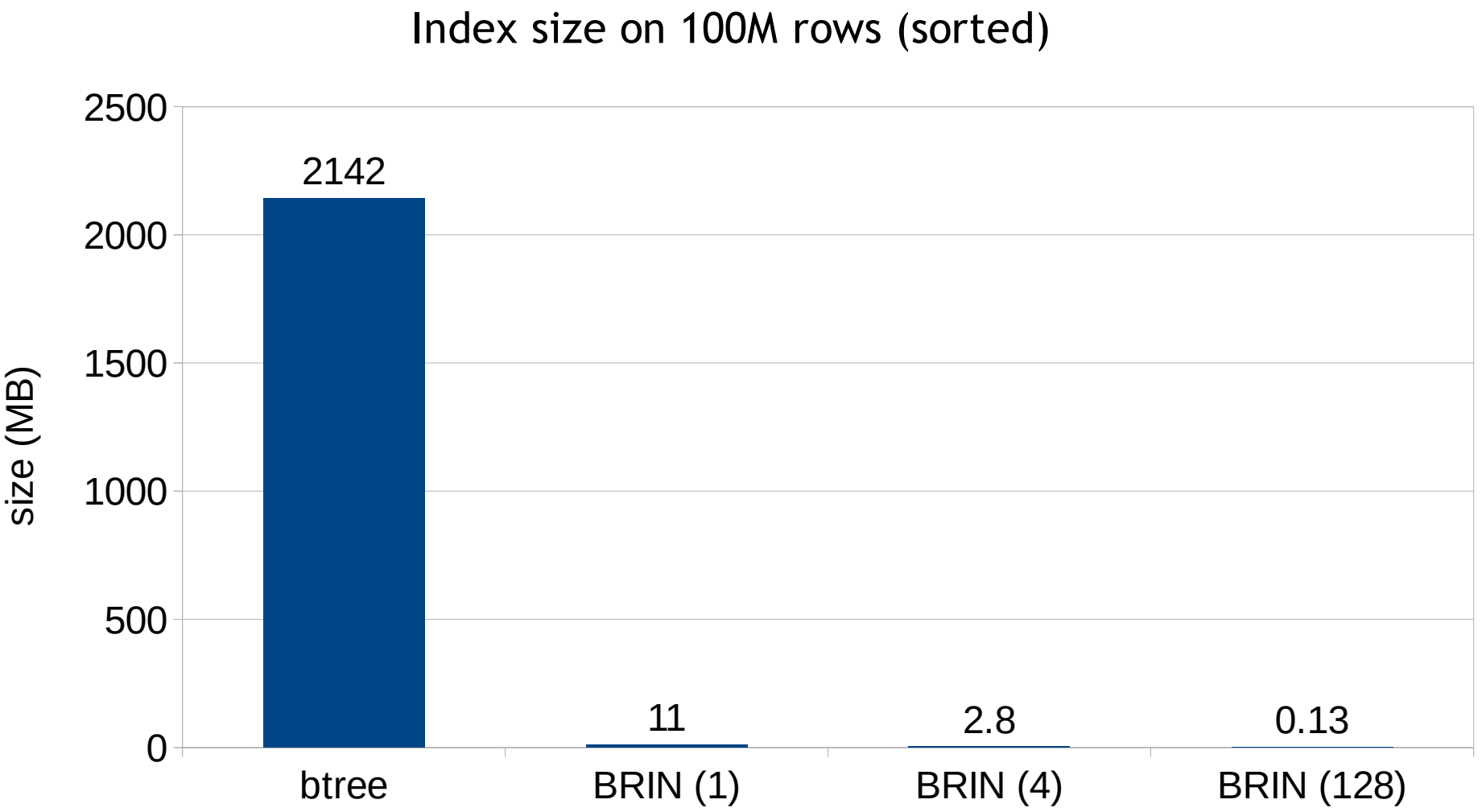
```
-- simple table with 100M rows
CREATE TABLE brin_test (val INT);

INSERT INTO brin_test SELECT mod(i, 100.000)
  FROM generate_series(1, 100.000.000) s(i);

-- btree and brin indexes
CREATE INDEX test_btree_idx ON brin_test(val);
CREATE INDEX test_brin_idx ON brin_test USING brin(val);

-- test query
SELECT COUNT(*) FROM brin_test WHERE val <= $1;
```

BRIN vs. BTREE



2ndQuadrant® PostgreSQL

BRIN vs. BTREE



2ndQuadrant® PostgreSQL

Aggregate functions

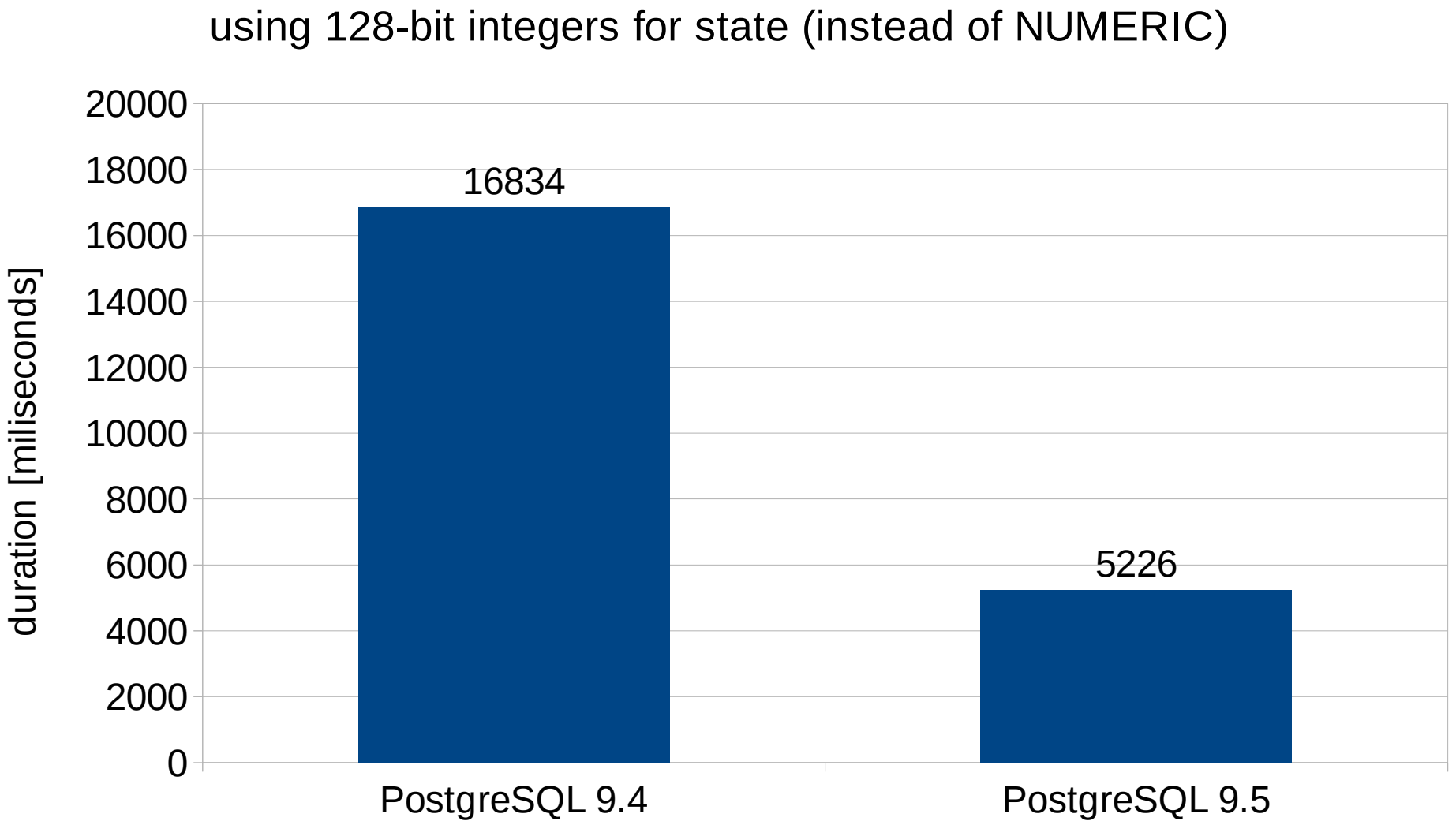
- Use 128-bit math to accelerate some aggregation functions.
  - some aggregates on INT used NUMERIC for internal state
  - modern CPUs/compilers support 128-bit integers
  - more efficient than NUMERIC, requires compiler support

- applies to aggregates on integer types
  - sum(int8), avg(int8)
  - var_*(int2), var_*(int4)
  - stdev_*(int2), stdev_*(int4)

Aggregate functions

```
-- test table with 50M rows
CREATE TABLE aggregates (a int, b int);
INSERT INTO aggregates SELECT i, i
  FROM generate_series(1, 50.000.000) s(i);

-- test query
SELECT SUM(a), AVG(b) FROM aggregates;
```

Aggregate functions / 128-bit state



2ndQuadrant® PostgreSQL

FIXEDDECIMAL

- extension
- fixed precision decimal type
- stored in 8B, faster than NUMERIC
- precision limited to FIXEDDECIMAL(17,2)

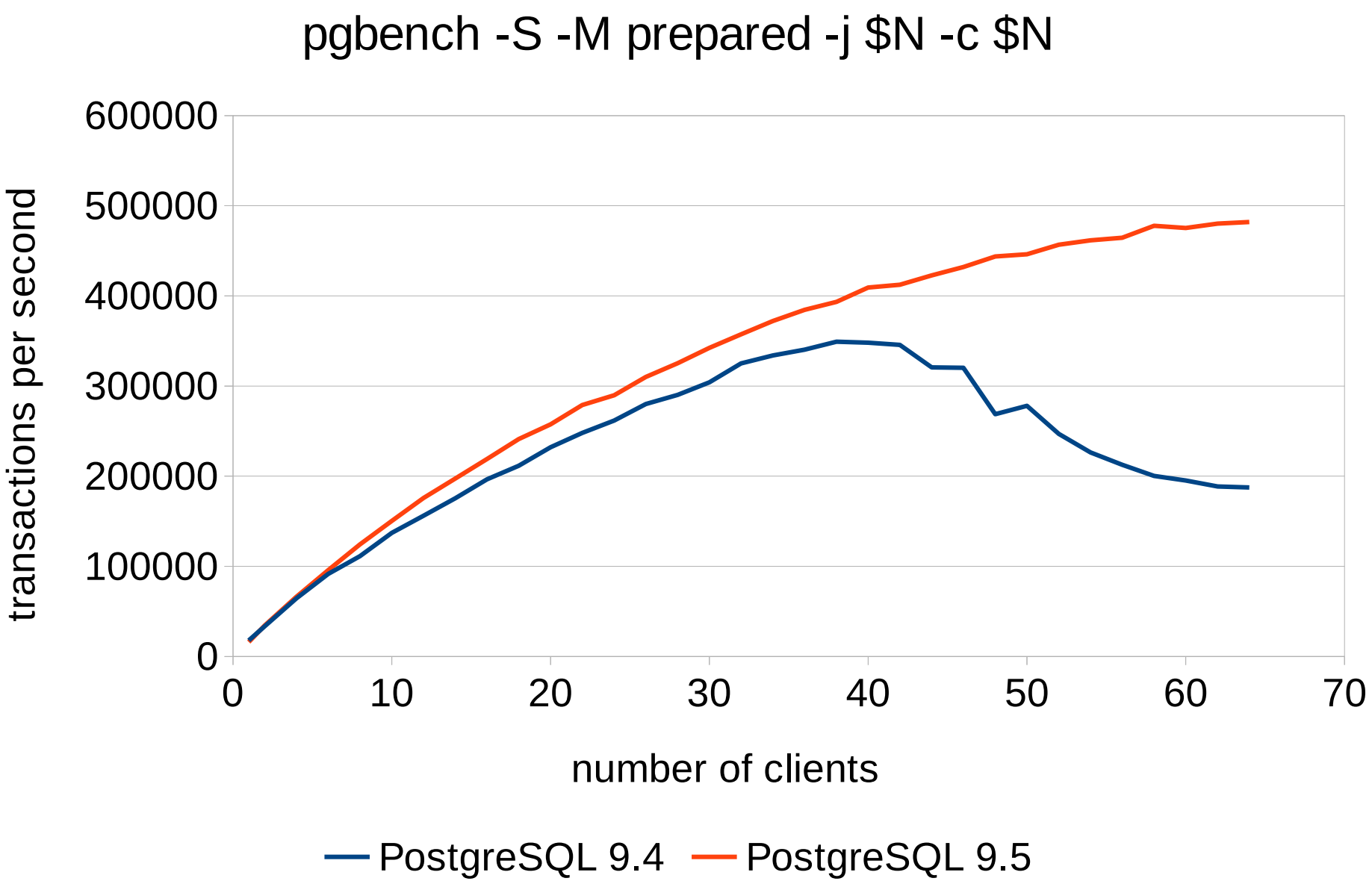
https://github.com/2ndQuadrant/fixeddecimal

Locking and concurrency

- checksum improvements (XLOG)
  - Speed up CRC calculation using slicing-by-8 algorithm.
  - Use Intel SSE 4.2 CRC instructions where available.
  - Optimize pg_comp_crc32c_sse42 routine slightly, and also use it on x86.
- reduce lock levels of some trigger and FK DDL

shared_buffers, LWLock

- Improve LWLock scalability.
- various shared buffer improvements
  - Improve concurrency of shared buffer replacement
  - Increase the number of buffer mapping partitions to 128.
  - Lockless StrategyGetBuffer clock sweep hot path.
  - Align buffer descriptors to cache line boundaries.
  - Make backend local tracking of buffer pins memory efficient
  - Reduce the number of page locks and pins during index scans
  - Optimize locking a tuple already locked by another subxact



2ndQuadrant®
PostgreSQL

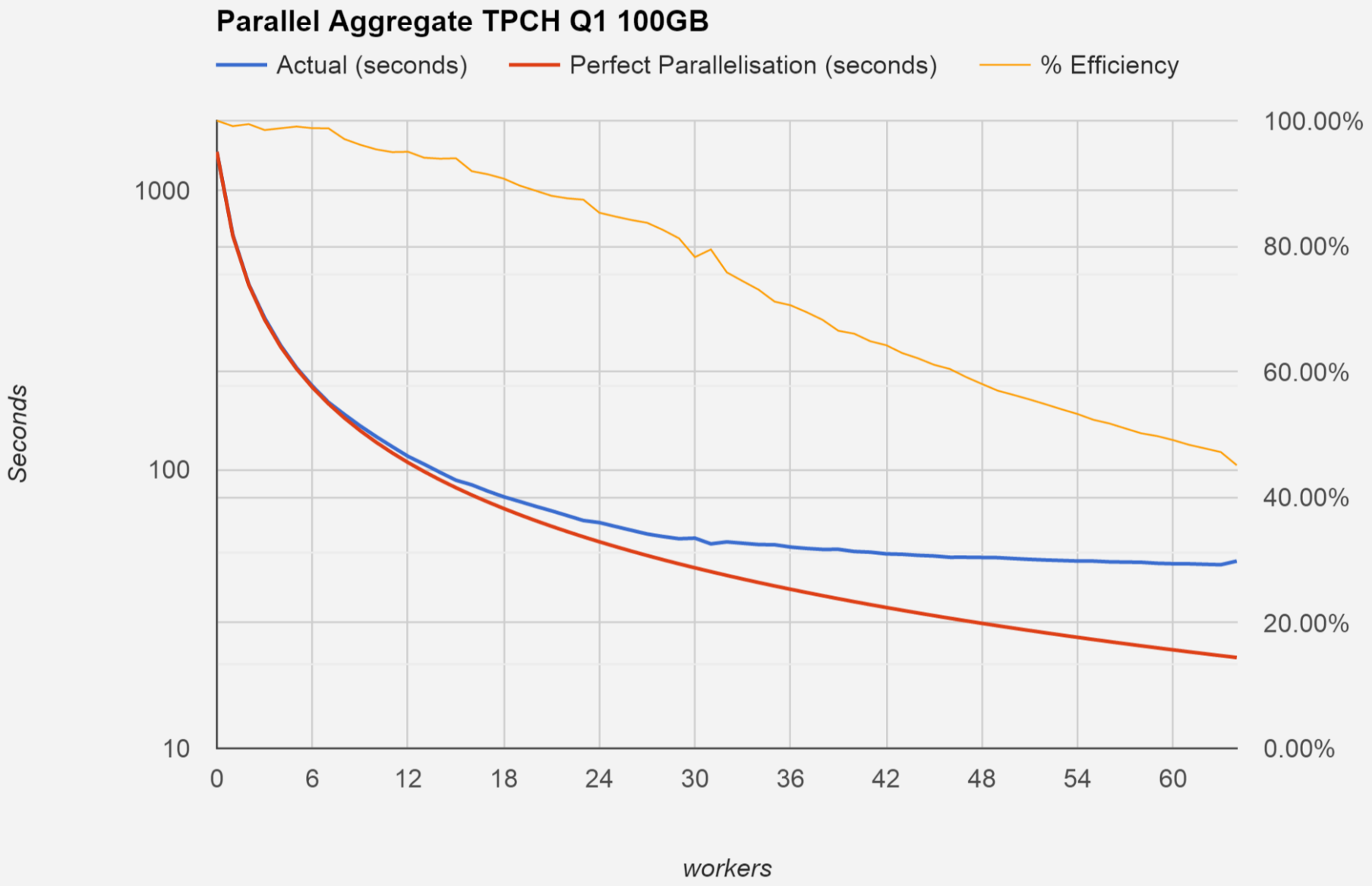
PostgreSQL 9.6

2ndQuadrant®
PostgreSQL

Parallel Queries

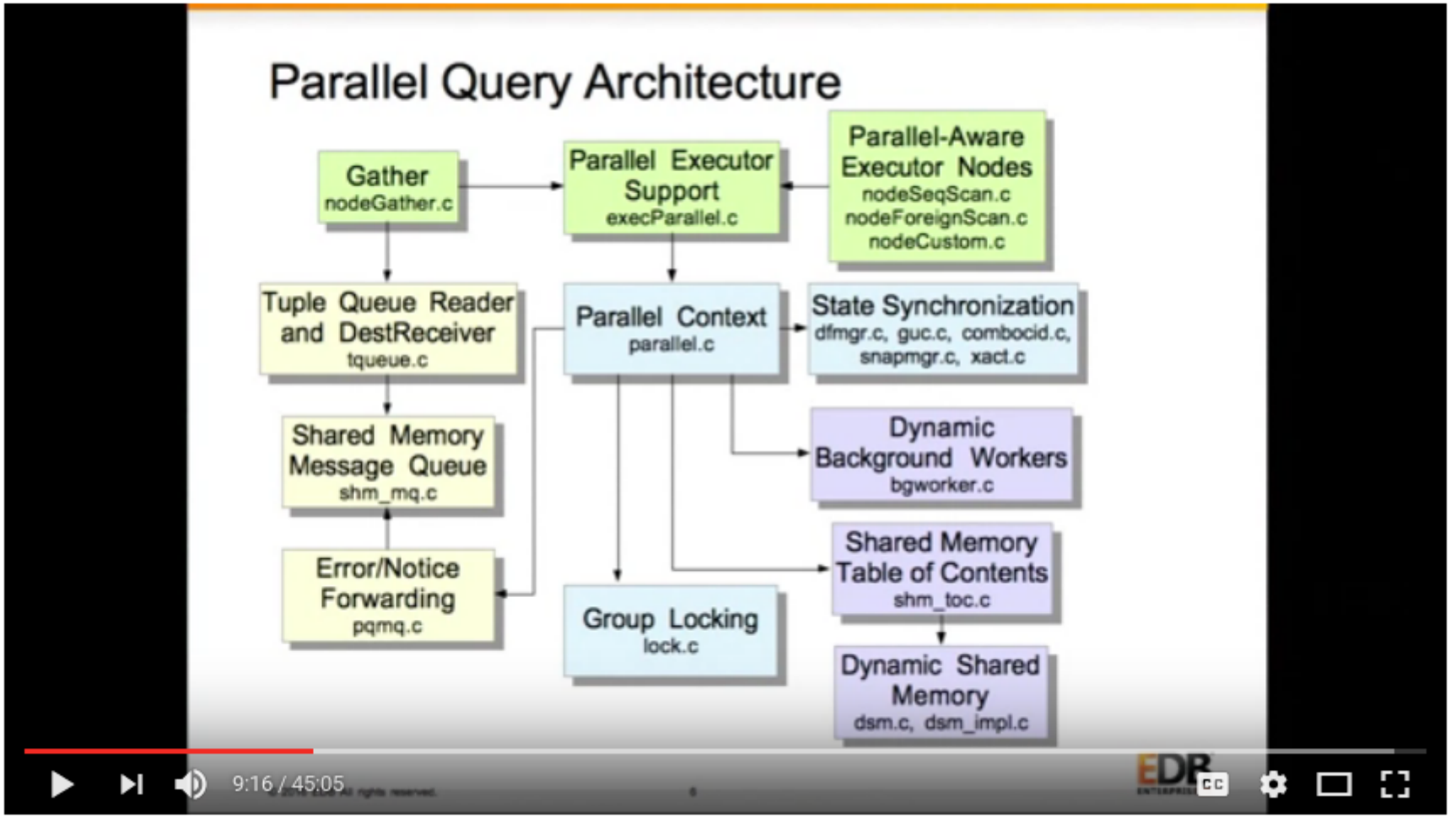
```
SET max_parallel_workers_per_gather = 4;
SELECT COUNT(*) FROM test_parallel WHERE test_func(a, 1);

                                QUERY PLAN
-----------------------------------------------------------------------
 Aggregate  (cost=15411721.93..15411721.94 rows=1 width=0)
   ->  Gather  (cost=1000.00..15328388.60 rows=33333330 width=0)
         Number of Workers: 4
         ->  Partial Seq Scan on test_parallel
                   (cost=0.00..5327388.60 rows=33333330 width=0)
               Filter: test_func(a, 1)
```

2ndQuadrant® PostgreSQL



http://blog.2ndquadrant.com/parallel-aggregate/

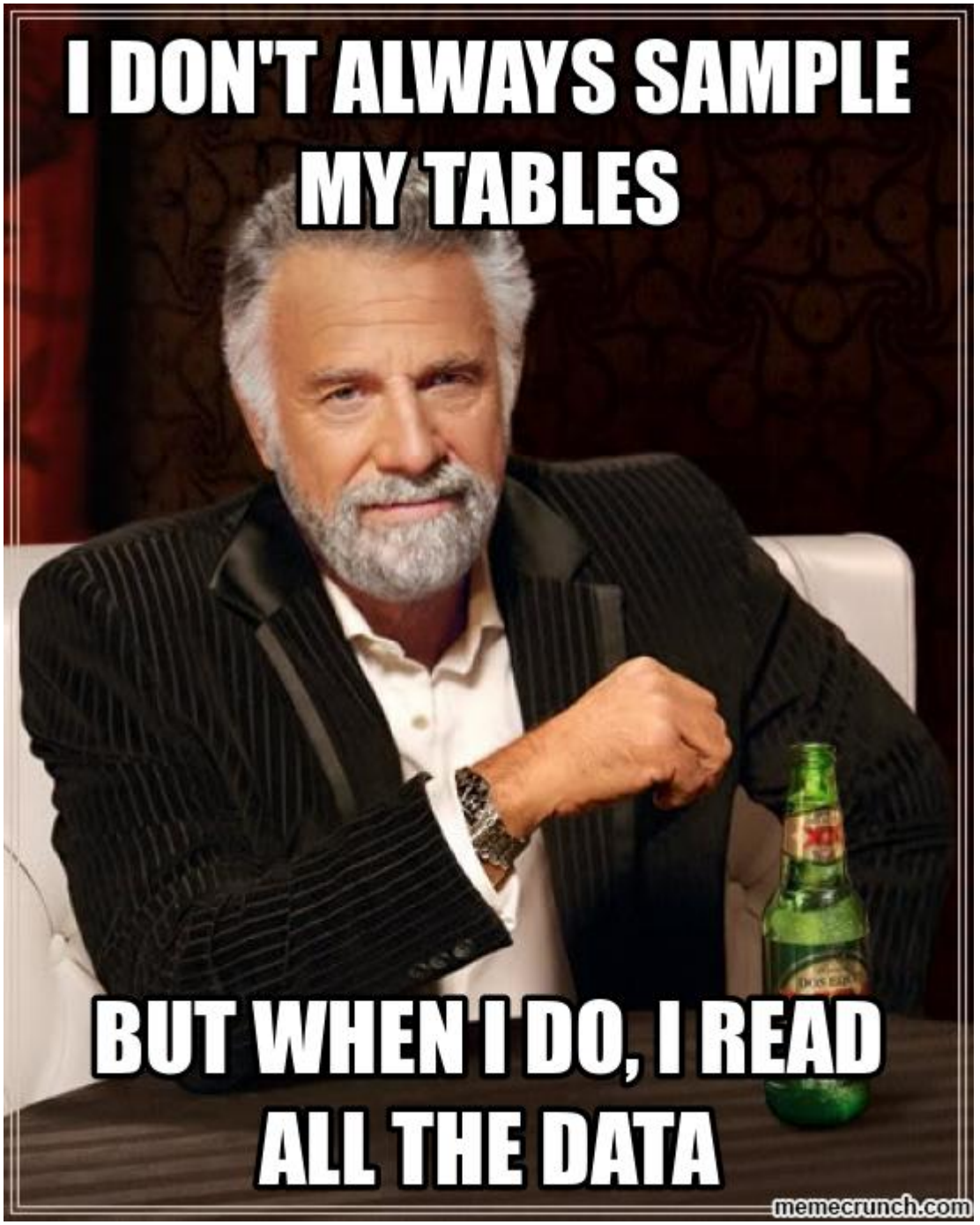
2ndQuadrant®
PostgreSQL



https://www.youtube.com/watch?v=ysHZ1PDnH-s

2ndQuadrant®
PostgreSQL

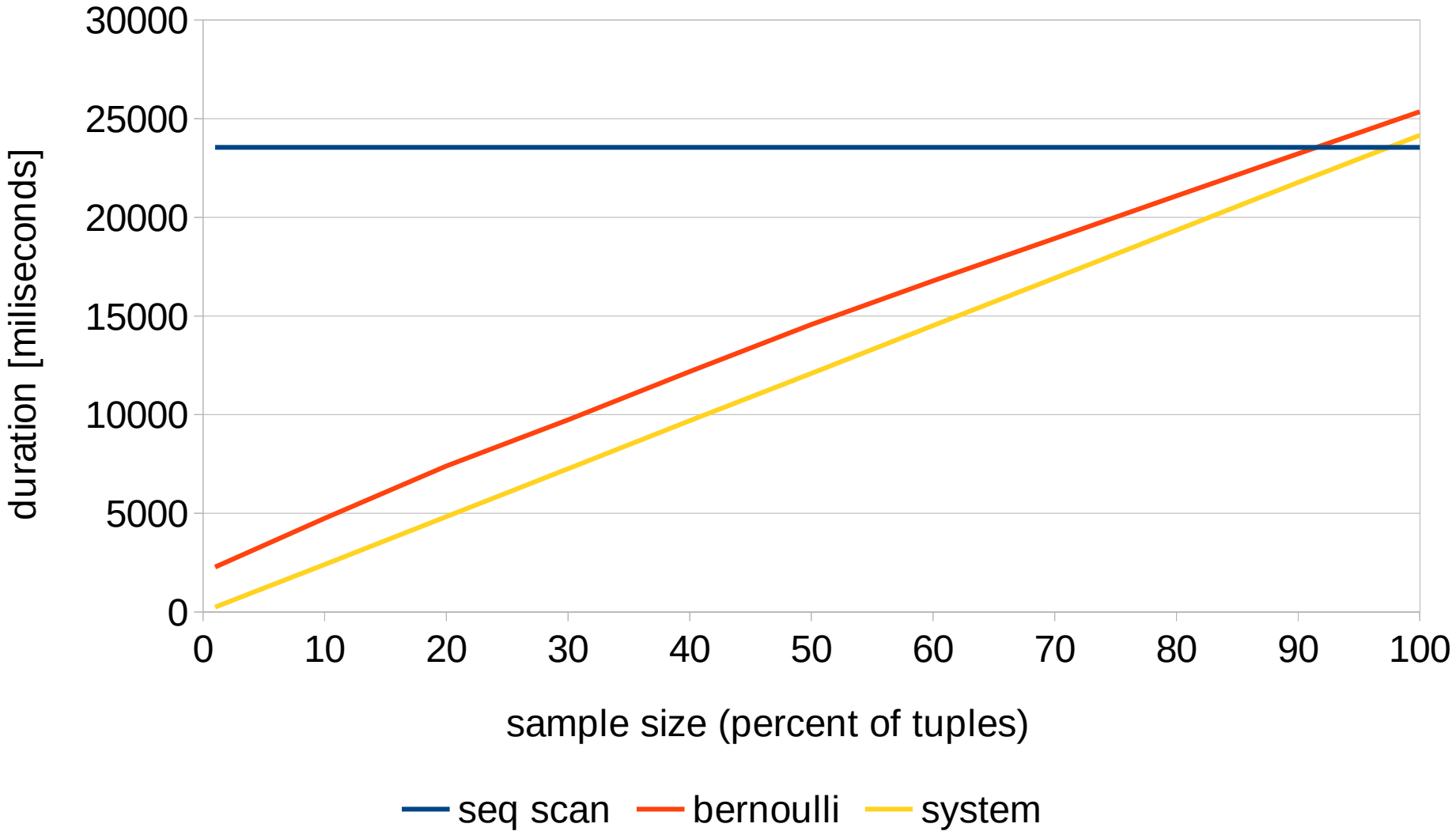
```
SELECT * FROM table
ORDER BY random() LIMIT 100;
```


I DON'T ALWAYS SAMPLE MY TABLES
BUT WHEN I DO, I READ ALL THE DATA
memecrunch.com

TABLESAMPLE

```
SELECT * FROM t TABLESAMPLE sampling_method (args)
                [REPEATABLE (seed)]

SELECT * FROM t TABLESAMPLE BERNOULLI (33.3);
SELECT * FROM t TABLESAMPLE SYSTEM (33.3);

-- tsm_system_rows
SELECT * FROM t TABLESAMPLE SYSTEM_ROWS (1000);

-- tsm_system_time
SELECT * FROM t TABLESAMPLE SYSTEM_TIME (1000);
```

TABLESAMPLE



2ndQuadrant®
PostgreSQL

Aggregate functions

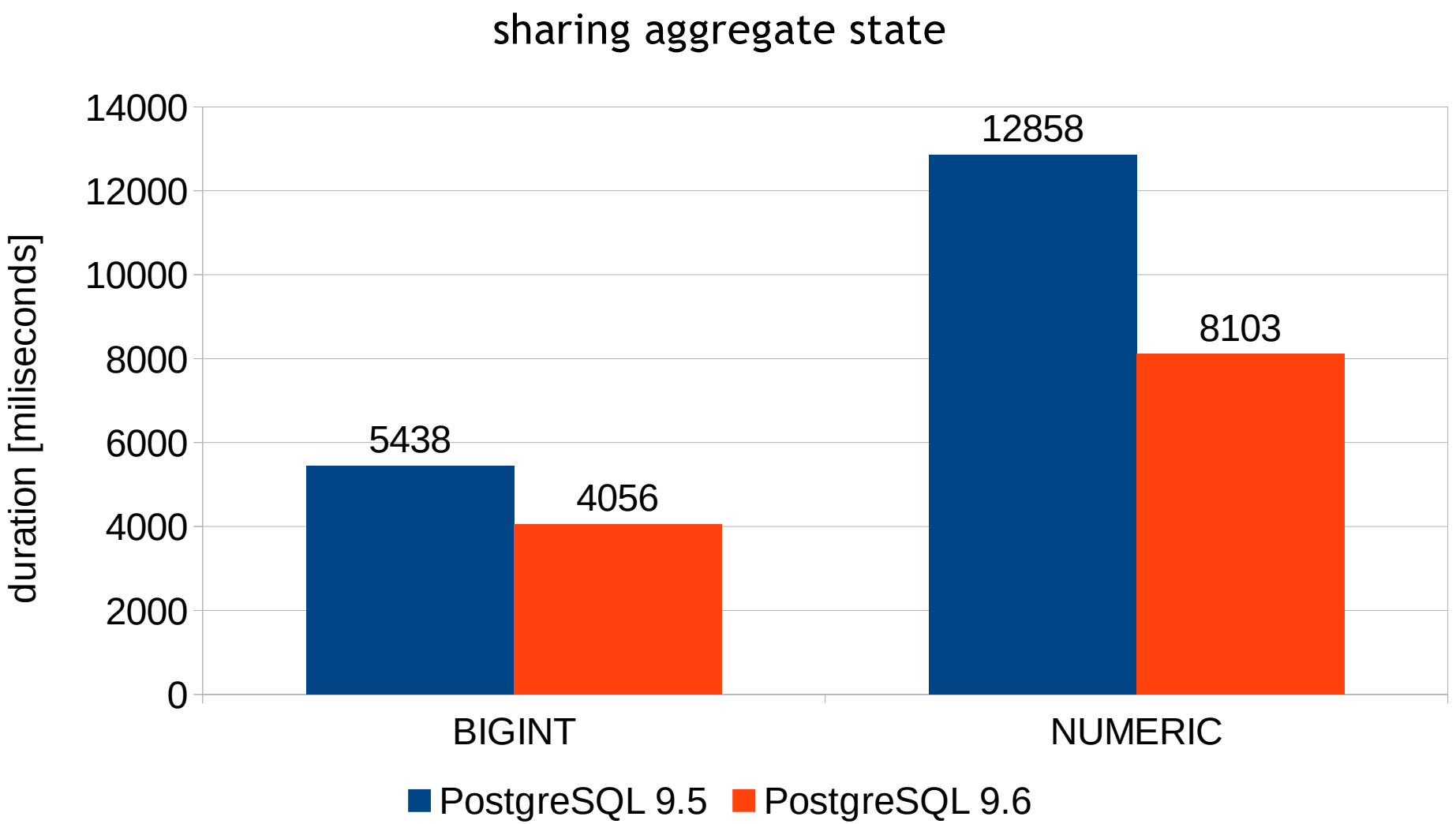
- some aggregates use the same state
  - AVG, SUM, …
  - we're keeping it separate and updating it twice
  - but only the final function is actually different
- so …

Share transition state between different aggregates when possible.

Aggregate functions

```
-- test table with 50M rows
CREATE TABLE aggregates (a INT);

INSERT INTO aggregates
SELECT * FROM generate_series(1, 50.000.000);

-- test query
SELECT SUM(a), AVG(a) FROM aggregates;
```

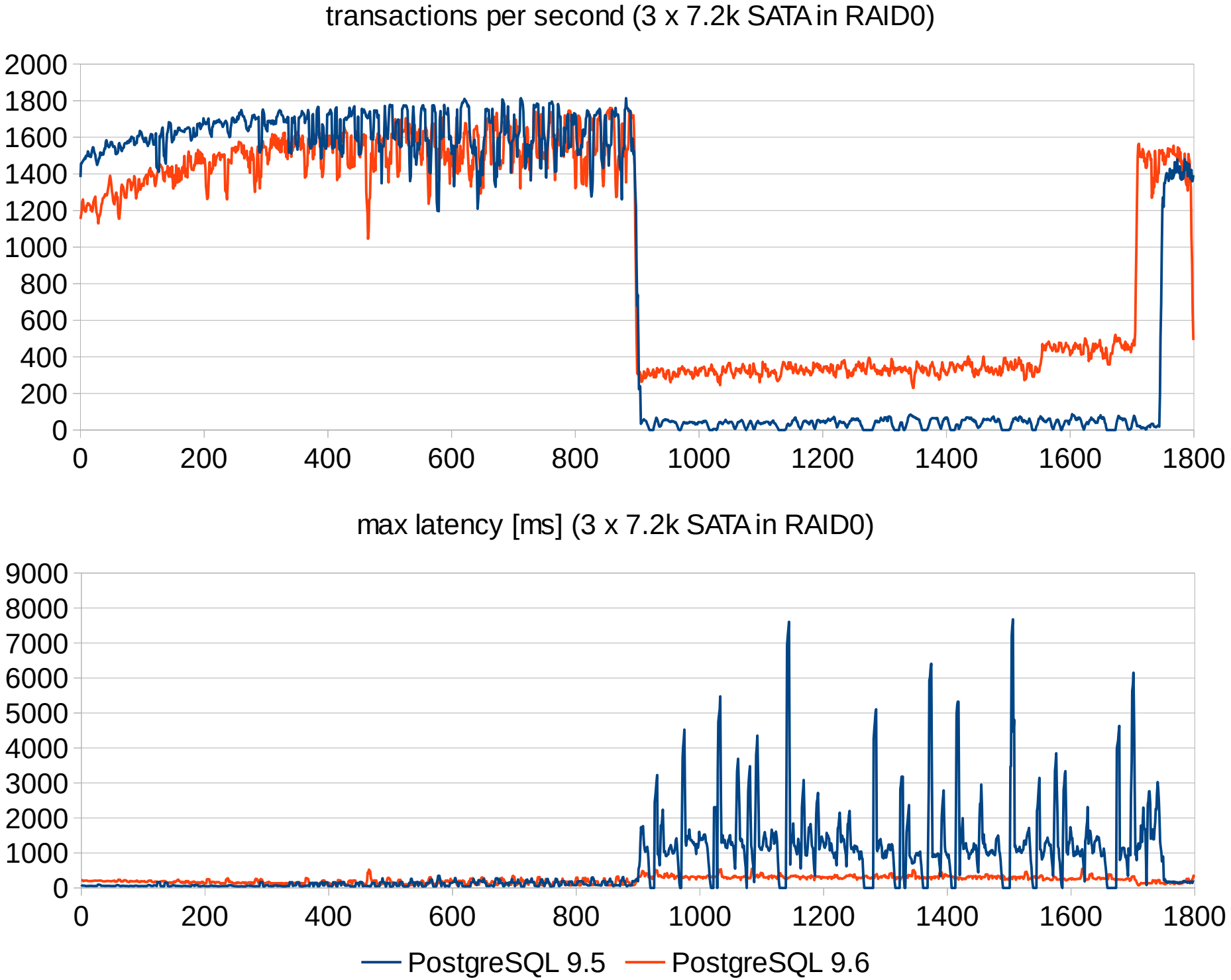
Aggregate functions

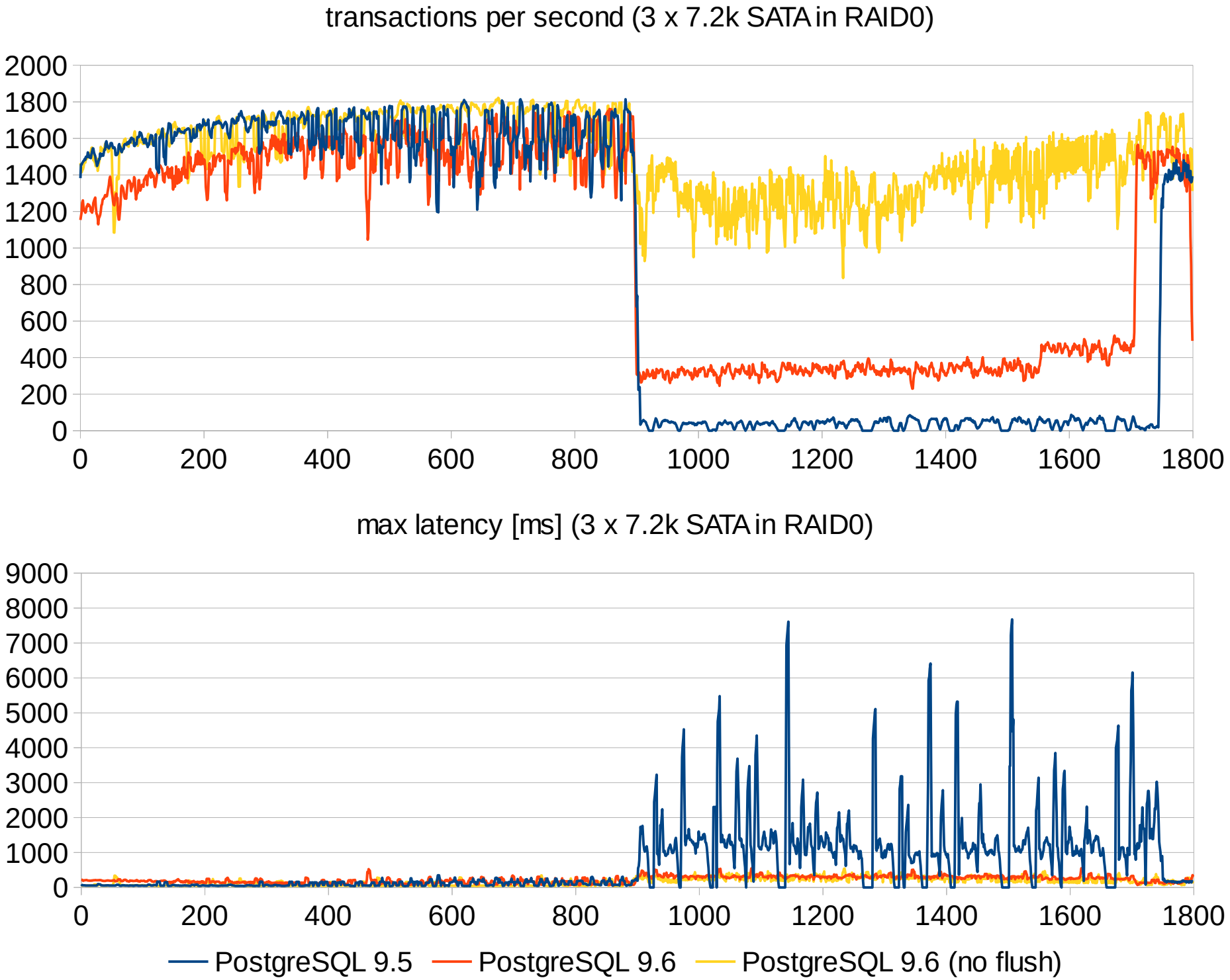


2ndQuadrant® PostgreSQL

Checkpoints

- issue #1: arbitrary order of blocks in shared_buffers
  - writing buffers => random I/O
  - PostgreSQL 9.6 sorts blocks => sequential
- issue #2: accumulation of dirty data in page cache
  - kernel writes data in bursts, affecting rest of the system
  - flush data continuously (default: every 256/512kB)
- more about variance than about throughput
- effect depends on I/O scheduler, storage, ...


transactions per second (3 x 7.2k SATA in RAID0)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
0
200
400
600
800
1000
1200
1400
1600
1800
max latency [ms] (3 x 7.2k SATA in RAID0)
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
200
400
600
800
1000
1200
1400
1600
1800
PostgreSQL 9.5
PostgreSQL 9.6


transactions per second (3 x 7.2k SATA in RAID0)
2000
1800
1600
1400
1200
1000
800
600
400
200
0
0
200
400
600
800
1000
1200
1400
1600
1800
max latency [ms] (3 x 7.2k SATA in RAID0)
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
0
200
400
600
800
1000
1200
1400
1600
1800
PostgreSQL 9.5
PostgreSQL 9.6
PostgreSQL 9.6 (no flush)

Freeze Map

- 32bit XIDs → freezing needed every ~2B transactions
  - read everything, remove old XIDs from tuples
  - postponing is bad idea (automatic shutdown)
- we've been reading everything (all pages)
  - requires a lot of resources (I/O and CPU)
  - most pages often “fully frozen” (no freezing needed)
- solution: maintain “freeze map” (part of visibility map)
  - during freezing, skip already “fully frozen” pages

Additional Improvements

- Optimizer
  - Index-Only Scans with partial indexes
  - FK join estimates
- Sorting
  - Reusing abbreviated keys during second pass of ordered [set] aggregates
  - SortSupport for text - strcoll() and strxfrm() caching
  - Memory prefetching while sequentially fetching from SortTuple array, tuplestore
  - Using quicksort and a merge step to significantly improve on tuplesort's single run "external sort"

PostgreSQL 10

2ndQuadrant®
PostgreSQL

WIP

- parallelization
  - additional nodes (Gather Merge, Index Scans)
  - CREATE INDEX (parallel sort)
  - vacuum
- optimizations
  - WARM (less-strict HOT)
  - partial sort
  - Unique Joins

Questions