Performance improvements in PostgreSQL 9.5 and 9.6

5432meet.us 2016, June 29, Milan

Tomas Vondra
tomas.vondra@2ndquadrant.com
many improvements
  - many of them related to performance
  - many quite large

release notes are good overview, but ...
  - many changes not mentioned explicitly
  - often difficult to get an idea of the impact

many talks about new features in general
  - this talk is about changes affecting performance
What we'll look at?

- PostgreSQL 9.5 & 9.6
- only “main” improvements
  - complete “features” (multiple commits)
  - try to showcase the impact
  - no particular order
- dozens of additional optimizations
  - see release notes for the full list
PostgreSQL 9.5
Sorting

- allow sorting by in-lined, non-SQL-callable functions
  - reduces per-call overhead
- use abbreviated keys for faster sorting (strxfrm)
  - VARCHAR, TEXT, NUMERIC
  - does not apply to CHAR values!
- places using “Sort Support” benefits from this
  - CREATE INDEX, REINDEX, CLUSTER
  - ORDER BY (when not evaluated using an index)
Sorting

-- randomly sorted table
CREATE TABLE test_text_random AS
SELECT md5(i::text) AS val
  FROM generate_series(1, 50.000.000) s(i);

-- correctly sorted table
CREATE TABLE test_text_asc AS
SELECT * from test_text_random ORDER BY 1;

-- test query
SELECT COUNT(1) FROM ( SELECT * FROM test_text_random ORDER BY 1
 ) foo;
Sorting improvements in PostgreSQL 9.5

sort duration on 50M rows (TEXT)

- asc
- desc
- almost asc
- almost desc
- random

duration [seconds]

- PostgreSQL 9.4
- PostgreSQL 9.5

2ndQuadrant
Professional PostgreSQL
Sorting improvements in PostgreSQL 9.5

sort duration on 50M rows (NUMERIC)

- **asc**
- **desc**
- **almost asc**
- **almost desc**
- **random**

PostgreSQL 9.4 vs PostgreSQL 9.5

**duration [seconds]**

- **asc**
- **desc**
- **almost asc**
- **almost desc**
- **random**

**dataset type**

**Legend:**
- Blue: PostgreSQL 9.4
- Red: PostgreSQL 9.5

---

2ndQuadrant
Professional PostgreSQL
Hash Joins

- reduce palloc overhead
  - dense packing of tuples (trivial local allocator, same life-span)
  - significant reduction of overhead (both space and time)
- reduce NTUP_PER_BUCKET to 1 (from 10)
  - goal is less that 1 tuple per bucket (on average)
  - significant speedup of lookups
- dynamically resize the hash table
  - handle under-estimates gracefully
  - otherwise easily 100s of tuples per bucket (linked list)
Hash Joins
Hash Joins
Hash Joins

-- dimension table (small one, will be hashed)
CREATE TABLE test_dim AS
SELECT (i-1) AS id, md5(i::text) AS val
  FROM generate_series(1, 100.000) s(i);

-- fact table (large one)
CREATE TABLE test_fact AS
SELECT mod(i, 100.000) AS dim_id, md5(i::text) AS val
  FROM generate_series(1, 50.000.000) s(i);

-- example query (join of the two tables)
SELECT count(*) FROM test_fact
  JOIN test_dim ON (dim_id = id);
PostgreSQL 9.5 Hash Join Improvements

join duration - 50M rows (outer), different NTUP_PER_BUCKET

NTUP_PER_BUCKET=10

NTUP_PER_BUCKET=1

hash size (number of tuples in dimension)
BRIN Indexes

WHERE c = 1999
WHERE c BETWEEN 100 AND 200

min=95, max=985
min=11, max=212
min=1, max=45
min=139, max=450
min=33, max=75
min=1223, max=2392
min=3456, max=7800
BRIN Indexes

-- table with 100M rows
CREATE TABLE test_bitmap AS
    SELECT mod(i, 100.000) AS val
    FROM generate_series(1, 100.000.000) s(i);
CREATE INDEX test_btree_idx ON test_bitmap(val);
CREATE INDEX test_brin_idx ON test_bitmap USING brin(val);

-- benchmark (enforce bitmap index scan)
SET enable_seqscan = off;
SET enable_indexscan = off;

SELECT COUNT(*) FROM test_bitmap WHERE val <= $1;
BRIN vs. BTREE

Bitmap Index Scan on 100M rows (sorted)

duration [milliseconds]

fraction of table matching the condition

- BTREE
- BRIN (128)
- BRIN (4)

2ndQuadrant
Professional PostgreSQL
BRIN vs. BTREE

index size on 100M rows

size (MB)

<table>
<thead>
<tr>
<th></th>
<th>btree</th>
<th>BRIN (1)</th>
<th>BRIN (4)</th>
<th>BRIN (128)</th>
</tr>
</thead>
<tbody>
<tr>
<td>index size</td>
<td></td>
<td>11</td>
<td>2.8</td>
<td>0.13</td>
</tr>
<tr>
<td>size (MB)</td>
<td>2142</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Other Index Improvements

- **CREATE INDEX**
  - avoid copying index tuples when building an index (palloc overhead)

- **Index-only scans with GiST**
  - support to range type, inet GiST opclass and btree_gist

- **Bitmap Index Scan**
  - in some cases up to 50% spent in `tbm_add_tuples`
  - cache the last accessed page in `tbm_add_tuples`
Other Improvements

- locking and shared_buffers scalability
  - reduce overhead, make it more concurrent
  - large (multi-socket) systems
  - reduce lock strength for some DDL commands
- CRC optimizations (--data-checksums)
  - use SSE when available, various optimizations
  - significantly improved throughput (GB/s)
- planner optimizations
  - make the planning / execution smarter
- PL/pgSQL improvements
read-only scalability improvements in 9.5

pgbench -S -M prepared -j $N -c $N

transactions per second

number of clients

PostgreSQL 9.4  PostgreSQL 9.5
PostgreSQL 9.6
Parallel Query

- until now, each query limited to 1 core
- 9.6 parallelizes some operations
  - sequential scan, aggregation, joins (NL + hash)
  - limited to read-only queries
  - setup overhead, efficient on large tables
- in the future
  - utility commands (CREATE INDEX, VACUUM, …)
  - additional operations (Sort, …)
  - improving supported ones (sharing hashtable in hashjoins)
Parallel Query

-- table with 1 billion rows (~80GB on disk)

CREATE TABLE f AS
    SELECT MOD(i,100000) AS id, MD5(i::text) AS h, random() AS amount
    FROM generate_series(1,1000000000) s(i);

EXPLAIN SELECT SUM(amount) FROM f JOIN d USING (id);

QUERY PLAN

Aggregate (cost=35598980.00..35598980.01 rows=1 width=8)
  -> Hash Join (cost=3185.00..33098980.00 rows=1000000000 width=8)
      Hash Cond: (f.id = d.id)
      -> Seq Scan on f (cost=0.00..19345795.00 rows=1000000000 ...)
      -> Hash (cost=1935.00..1935.00 rows=100000 width=4)
          -> Seq Scan on d (cost=0.00..1935.00 rows=100000 ...)

(6 rows)
SET max_parallel_workers_per_gather = 32;

EXPLAIN SELECT SUM(amount) FROM f JOIN d USING (id);

QUERY PLAN

Finalize Aggregate  (cost=14488869.82..14488869.83 rows=1 width=8)
  ->  Gather  (cost=14488868.89..14488869.80 rows=9 width=8)
    Workers Planned: 9
      ->  Partial Aggregate  (cost=14487868.89..14487868.90 rows=1 width=8)
        ->  Hash Join  (cost=3185.00..11987868.89 rows=1000000000 width=8)
          Hash Cond: (f.id = d.id)
            ->  Parallel Seq Scan on f  (cost=0.00..10456906.11 ...)
            ->  Hash  (cost=1935.00..1935.00 rows=100000 width=4)
              ->  Seq Scan on d  (cost=0.00..1935.00 rows=100000 ...)

(9 rows)
<table>
<thead>
<tr>
<th>PID</th>
<th>VIRT</th>
<th>RES</th>
<th>SHR</th>
<th>S</th>
<th>%CPU</th>
<th>%MEM</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>19018</td>
<td>32.8g</td>
<td>441m</td>
<td>427m</td>
<td>R</td>
<td>100</td>
<td>0.2</td>
<td>postgres: sekondquad test [local] SELECT</td>
</tr>
<tr>
<td>20134</td>
<td>32.8g</td>
<td>80m</td>
<td>74m</td>
<td>R</td>
<td>100</td>
<td>0.0</td>
<td>postgres: bgworker: parallel worker for PID 19018</td>
</tr>
<tr>
<td>20135</td>
<td>32.8g</td>
<td>80m</td>
<td>74m</td>
<td>R</td>
<td>100</td>
<td>0.0</td>
<td>postgres: bgworker: parallel worker for PID 19018</td>
</tr>
<tr>
<td>20136</td>
<td>32.8g</td>
<td>80m</td>
<td>74m</td>
<td>R</td>
<td>100</td>
<td>0.0</td>
<td>postgres: bgworker: parallel worker for PID 19018</td>
</tr>
<tr>
<td>20140</td>
<td>32.8g</td>
<td>80m</td>
<td>74m</td>
<td>R</td>
<td>100</td>
<td>0.0</td>
<td>postgres: bgworker: parallel worker for PID 19018</td>
</tr>
<tr>
<td>20141</td>
<td>32.8g</td>
<td>80m</td>
<td>74m</td>
<td>R</td>
<td>99</td>
<td>0.0</td>
<td>postgres: bgworker: parallel worker for PID 19018</td>
</tr>
<tr>
<td>20142</td>
<td>32.8g</td>
<td>80m</td>
<td>74m</td>
<td>R</td>
<td>99</td>
<td>0.0</td>
<td>postgres: bgworker: parallel worker for PID 19018</td>
</tr>
<tr>
<td>20137</td>
<td>32.8g</td>
<td>80m</td>
<td>74m</td>
<td>R</td>
<td>99</td>
<td>0.0</td>
<td>postgres: bgworker: parallel worker for PID 19018</td>
</tr>
<tr>
<td>20138</td>
<td>32.8g</td>
<td>80m</td>
<td>74m</td>
<td>R</td>
<td>99</td>
<td>0.0</td>
<td>postgres: bgworker: parallel worker for PID 19018</td>
</tr>
<tr>
<td>20139</td>
<td>32.8g</td>
<td>80m</td>
<td>74m</td>
<td>R</td>
<td>99</td>
<td>0.0</td>
<td>postgres: bgworker: parallel worker for PID 19018</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>[watchdog/2]</td>
</tr>
<tr>
<td>281</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>S</td>
<td>0</td>
<td>0.0</td>
<td>[khugepaged]</td>
</tr>
</tbody>
</table>
speedup with parallel query

example query without and with parallelism

<table>
<thead>
<tr>
<th></th>
<th>duration [seconds]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 workers</td>
<td>360</td>
</tr>
<tr>
<td>9 workers</td>
<td>40</td>
</tr>
</tbody>
</table>
Parallel Query Has Arrived!

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

-- table with 50M rows
CREATE TABLE test_aggregates AS
SELECT i AS a
    FROM generate_series(1, 50.000.000) s(i);

-- compute both SUM and AVG on a column
SELECT SUM(a), AVG(a) FROM test_aggregates;
Aggregate functions

sharing aggregate state

```
<table>
<thead>
<tr>
<th></th>
<th>BIGINT</th>
<th>NUMERIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>du</td>
<td>5438</td>
<td>12858</td>
</tr>
<tr>
<td>tr</td>
<td>4056</td>
<td>8103</td>
</tr>
</tbody>
</table>
```
Checkpoints

• we need to write dirty buffers to disk regularly
  – data written to page cache (no O_DIRECT)
  – kernel responsible for actual write out
• until now, we simply walked shared buffers
  – random order of buffers, causing random I/O
  – 9.6 sorts the buffers first, to get sequential order
• until now, we only only did fsync at the end
  – a lot of dirty data in page cache, latency spikes
  – 9.6 allows continuous flushing (disabled by default)
Improving Postgres' Buffer Manager

Andres Freund
PostgreSQL Developer & Committer
Citus Data – citusdata.com - @citusdata

http://anarazel.de/talks/fosdem-2016-01-31/io.pdf
pgbench -M prepared -c 32 -j 32

shared_buffers = 16GB, max_wal_size = 100GB
pgbench -M prepared -c 32 -j 32

shared_buffers = 16GB, max_wal_size = 100GB, target = 0.9; OS tuning (no dirty)
Sort (again)

- abbreviated keys extended to
  - additional data types: uuid, bytea, char(n)
  - ordered set aggregates
- use quicksort (instead of replacement selection) for “external sort” case
- ... and many other optimizations
Freezing

- XIDs are 64-bit, but we only store the low 32 bits
  - need to do “freeze” every ~2 billion transactions
  - that means reading all the data (even unmodified parts)
  - problem on large databases (time consuming)
  - users often postpone until it’s too late (outage)

- PostgreSQL 9.6 introduces “freeze map”
  - similar to “visibility map” (and stored in the same file)
  - “all rows on page are frozen” - we can skip this 8kB page
Future

- extending parallel query (additional operations)
- declarative partitioning (smart joins, ...)
- columnar features
  - vectorized execution, compression, ...
  - do more with the same amount of resources
- improving planner
  - correlation statistics, optimizations (unijoins)
Questions?