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
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
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
/* 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

sort duration on 50M rows (NUMERIC)

- sorted (ASC)
- sorted (DESC)
- random

PostgreSQL 9.4
PostgreSQL 9.5
Sorting speedups on PostgreSQL 9.5

speedup on 50M rows (TEXT and NUMERIC)
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 dimension ON (dim_id = id);
PostgreSQL 9.5 Hash Join Improvements

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

<table>
<thead>
<tr>
<th>Hash size (number of tuples in &quot;dimension&quot; table)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>1000000</td>
</tr>
<tr>
<td>2000000</td>
</tr>
<tr>
<td>3000000</td>
</tr>
<tr>
<td>4000000</td>
</tr>
<tr>
<td>5000000</td>
</tr>
<tr>
<td>6000000</td>
</tr>
<tr>
<td>7000000</td>
</tr>
<tr>
<td>8000000</td>
</tr>
<tr>
<td>9000000</td>
</tr>
<tr>
<td>1000000</td>
</tr>
<tr>
<td>1100000</td>
</tr>
<tr>
<td>1200000</td>
</tr>
<tr>
<td>1300000</td>
</tr>
<tr>
<td>1400000</td>
</tr>
<tr>
<td>1500000</td>
</tr>
<tr>
<td>1600000</td>
</tr>
<tr>
<td>1700000</td>
</tr>
<tr>
<td>1800000</td>
</tr>
<tr>
<td>1900000</td>
</tr>
<tr>
<td>2000000</td>
</tr>
</tbody>
</table>

- Blue line: 9.4 (NTUP_PER_BUCKET=10)
- Red line: 9.5 (NTUP_PER_BUCKET=1)
Indexes

• Speed-up Bitmap Index Scan
  - in some cases up to 50% was spent in t bm_add_tuples
  - cache the last accessed page in t bm_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

cache last page in tbm_add_tuples()

<table>
<thead>
<tr>
<th>duration [milliseconds]</th>
</tr>
</thead>
<tbody>
<tr>
<td>3000</td>
</tr>
<tr>
<td>2500</td>
</tr>
<tr>
<td>2000</td>
</tr>
<tr>
<td>1500</td>
</tr>
<tr>
<td>1000</td>
</tr>
<tr>
<td>500</td>
</tr>
<tr>
<td>0</td>
</tr>
</tbody>
</table>

query 1

query 2

PostgreSQL 9.4

PostgreSQL 9.5
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

Bitmap Index Scan on 100M rows (table sorted)

- **BRIN vs. BTREE**
- **Bitmap Index Scan on 100M rows (table sorted)**
- **duration [milliseconds]**
- **fraction of table matching the condition**
- **BTREE**
- **BRIN (128)**
- **BRIN (4)**
BRIN vs. BTREE

Index size on 100M rows (sorted)

- btree: 2142 MB
- BRIN (1): 11 MB
- BRIN (4): 2.8 MB
- BRIN (128): 0.13 MB
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

using 128-bit integers for state (instead of NUMERIC)
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
pgbench -S -M prepared -j $N -c $N
PostgreSQL 9.6
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)
Parallel Query Architecture

- Gather: nodeGather.c
- Parallel Executor Support: execParallel.c

- Parallel-Aware Executor Nodes:
  - nodeSeqScan.c
  - nodeForeignScan.c
  - nodeCustom.c

- Tuple Queue Reader and DestReceiver: tqueue.c
- Parallel Context: parallel.c
- State Synchronization:
  - dfmgr.c, guc.c, combocid.c
  - snapmgr.c, xact.c

- Shared Memory Message Queue: shm_mq.c
- Error/Notice Forwarding: pqmq.c
- Group Locking: lock.c
- Dynamic Background Workers: bgworker.c

- Shared Memory Table of Contents: shm_toc.c
- Dynamic Shared Memory:
  - dsm.c, dsm_impl.c

https://www.youtube.com/watch?v=ysHZ1PDnH-s
SELECT * FROM table
ORDER BY random() LIMIT 100;
I DON'T ALWAYS SAMPLE MY TABLES
BUT WHEN I DO, I READ ALL THE DATA
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);
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

sharing aggregate state

<table>
<thead>
<tr>
<th></th>
<th>PostgreSQL 9.5</th>
<th>PostgreSQL 9.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGINT</td>
<td>5438</td>
<td>4056</td>
</tr>
<tr>
<td>NUMERIC</td>
<td>12858</td>
<td>8103</td>
</tr>
</tbody>
</table>
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)

max latency [ms] (3 x 7.2k SATA in RAID0)

- 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
WIP

- parallelization
  - additional nodes (Gather Merge, Index Scans)
  - CREATE INDEX (parallel sort)
  - vacuum

- optimizations
  - WARM (less-strict HOT)
  - partial sort
  - Unique Joins