Performance improvements in PostgreSQL 9.5 (and beyond)

pgconf.de 2015, November 27, Hamburg

Tomas Vondra
tomas.vondra@2ndquadrant.com
PostgreSQL 9.5, 9.6, ...

- 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
- PostgreSQL 9.6+
  - committed
  - still being worked on (commitfests)
- only “main” improvements
  - complete “features” (multiple commits)
  - try to showcase them, show the impact
  - no particular order
- won't mention too many low-level optimizations
slides

http://www.slideshare.net/fuzzycz/performance-in-pg95

test scripts

https://github.com/2ndQuadrant/performance-in-pg95
Sorting

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

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

CREATE TABLE test_text_asc AS
SELECT * from test_text_random
    ORDER BY 1;

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)

<table>
<thead>
<tr>
<th>dataset type</th>
<th>PostgreSQL 9.4</th>
<th>PostgreSQL 9.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>asc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>desc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>almost asc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>almost desc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sorting improvements in PostgreSQL 9.5

sort duration on 50M rows (TEXT)

duration [seconds]

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

Dataset type

- PostgreSQL 9.4
- PostgreSQL 9.5
Sorting speedups on PostgreSQL 9.5

speedup on 50M rows (TEXT and NUMERIC)
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

CREATE TABLE test_dim AS
SELECT (i-1) AS id, md5(i::text) AS val
FROM generate_series(1,100.000) s(i);

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);

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

duration [milliseconds]

hash size (number of tuples)
PostgreSQL 9.5 Hash Join Improvements

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

- NTUP_PER_BUCKET=10
- NTUP_PER_Bucket=1
- PostgreSQL 9.5
Indexes

- 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% was spent in tnm_add_tuples
  - cache the last accessed page in tnm_add_tuples
- BRIN
  - block range indexes, tracking min/max per block
  - only bitmap index scans (equality and range queries)
CREATE EXTENSION btree_gin;

CREATE TABLE t AS

SELECT (v / 10)::int4 AS i
  FROM generate_series(1, 5.000.000) AS v;

CREATE INDEX idx ON t USING gin (i);

SET enable_seqscan = off;

SELECT * FROM t WHERE i >= 0;

SELECT * FROM t WHERE i >= 100 AND i <= 100;
Bitmap build speedup

cache last page in tbm_add_tuples()
BRIN Indexes

-- data preparation
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
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 vs. fraction of table matching the condition

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

2ndQuadrant
Professional PostgreSQL
BRIN vs. BTREE

Index size on 100M rows (sorted)

<table>
<thead>
<tr>
<th>Type</th>
<th>Size (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>btree</td>
<td>2142</td>
</tr>
<tr>
<td>BRIN (1)</td>
<td>11</td>
</tr>
<tr>
<td>BRIN (4)</td>
<td>2.8</td>
</tr>
<tr>
<td>BRIN (128)</td>
<td>0.13</td>
</tr>
</tbody>
</table>
Aggregate functions

• Use 128-bit math to accelerate some aggregation functions.
  – some INT aggregate functions used NUMERIC for internal state
  – requires support for 128-bit integers (if provided by compiler).

• impacted aggregates
  – sum(int8)
  – avg(int8)
  – var_*(int2)
  – var_*(int4)
  – stdev_*(int2)
  – stdev_*(int4)
Aggregate functions

CREATE TABLE test_aggregates AS
SELECT i AS a, i AS b
FROM generate_series(1, 50.000.000) s(i);

SELECT SUM(a), AVG(b) FROM test_aggregates;
Aggregate functions / 128-bit state

using 128-bit integers for state (instead of NUMERIC)
PL/pgSQL

- Support "expanded" objects, particularly arrays, for better performance.
- Allocate ParamListInfo once per plpgsql function, not once per expression.
- Use standard casting mechanism to convert types in plpgsql, when possible.
- Use fast path in plpgsql's RETURN/RETURN NEXT in more cases.
Planner and optimizer

- remove unnecessary references to left outer join subqueries
- pushdown of query restrictions into window functions
- simplification of EXISTS() subqueries containing LIMIT
- teach predtest.c that "foo" implies "foo IS NOT NULL"
- improve predtest.c's ability to reason about operator expressions
Locking and concurrency

• checksum improvements
  – 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.

• add a basic atomic ops API abstracting away platform/architecture details.

• reduce lock levels of some trigger DDL and add FKs
Locking and concurrency

- 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

transactions per second

number of clients

- PostgreSQL 9.4
- PostgreSQL 9.5
PostgreSQL 9.6+
Parallel Seq Scan

SET max_parallel_degree = 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 Seq Scan

duration [seconds]

max_parallel_degree

speedup for selectivity and parallel degree (100M rows)

1.00% 25.00% 90.00% 100.00%

2ndQuadrant
Professional PostgreSQL
TABLESAMPLE

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

SELECT * FROM t TABLESAMPLE BERNULLI (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

sampling duration

<table>
<thead>
<tr>
<th>sample size (percent of tuples)</th>
<th>duration [milliseconds]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>25000</td>
</tr>
<tr>
<td>50</td>
<td>25000</td>
</tr>
<tr>
<td>90</td>
<td>25000</td>
</tr>
</tbody>
</table>

- seq scan
- bernoulli
- system

2ndQuadrant
Professional 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

CREATE TABLE test_aggregates AS
SELECT i AS a
    FROM generate_series(1, 50.000.000) s(i);

SELECT SUM(a), AVG(a) FROM test_aggregates;
Aggregate functions

sharing aggregate state

<table>
<thead>
<tr>
<th>Type</th>
<th>Before</th>
<th>After</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>
Disabling HOT cleanup

- HOT allows UPDATEs without bloating indexes
  - a page may have and many “previous” tuple versions
  - the dead versions are cleaned by VACUUM or by queries reading the block
  - single query may be forced to cleanup the whole table (e.g. after a batch update)
  - clear impact on performance, a bit unpredictable

- the patch attempts to somehow limit the impact
  - query only fixes limited number of pages, etc.
Checkpoints

• continuous flushing (and sorting writes)
  – more about variance than about throughput
  – eliminate latency stalls / spikes due to checkpoints
  – effect depends on I/O scheduler, storage, ...

• compensate for full_page_writes
  – spread checkpoints assume constant WAL rate
  – not really true due to initial rush to write full pages
  – scheduling gets confused by this difference
  – patch tries to compensate for this effect
Freezing large tables

- every time we “run out of XIDs” we need to freeze tuples
  - we have to scan all the tables to freeze all pages
  - even if many of the pages are already “fully frozen”
  - serious problem on large databases
  - users often postpone the freezing (and then DB shuts down)
- add “all tuples frozen” into visibility map
  - allows skipping already frozen pages
- patch seems mostly ready
  - mostly discussions about renaming (vm or vfm?)
Additional 9.6+ changes

- Locking and concurrency
  - Reduce ProcArrayLock contention by removing backends in batches.
- PL/pgSQL
  - Further reduce overhead for passing plpgsql variables to the executor.
- Planner / Optimizer
  - Unique Joins
  - Index-only scans with partial indexes
  - FK join estimates
  - Selectivity estimation for intarray
  - Table Partition + Join Pushdown
  - FDW join pushdown
Additional 9.6+ changes

• Declarative partitioning
  - easier maintenance (huge improvement)
  - allows advanced planning (insight into partitioning rules)

• 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"