
# Parallel Query In PostgreSQL

- Amit Kapila | 2016.12.01

# Contents

- Parallel Query capabilities in 9.6
- Tuning parameters
- Operations where parallel query is prohibited
- TPC-H results
- Parallel Query capabilities in pipeline

# Parallel Query

- PostgreSQL provides parallel query to speed up query execution for machines that have multiple CPUs.
- Parallelism is realised using background workers.
- Multiple processes working together on a SQL Statement can dramatically increase the performance of data-intensive operations.

# Parallel Query capabilities in 9.6

- Parallel Sequential Scans

```
Seq Scan on foo
```

becomes

```
Gather
   Workers Planned: 2
   Workers Launched: 2
   -> Parallel Seq Scan on foo
```

- Two workers and master backend work together to scan 'foo'. Such scans are very helpful if filter is highly-selective or contains costly expression like parallel-safe functions.

# Parallel Query capabilities in 9.6

- Parallel Nestloop Joins

```
Gather
  Workers Planned: 2
  Workers Launched: 2
   -> Nested Loop
      -> Parallel Seq Scan on foo
      -> Index Scan using bar_pkey
         Index Cond: (b = foo.b)
```

- Two workers and master backend work together to perform join between 'foo' and 'bar'. Such execution plans are very helpful, if join eliminates many rows. A filter on parallel seq scan will make it even better.

# Parallel Query capabilities in 9.6

- Parallel Hash Joins

```
Gather
  Workers Planned: 2
  -> Hash Join
      Hash Cond: (foo.b = bar.b)
     -> Parallel Seq Scan on foo
     -> Hash
          -> Seq Scan on bar
```

- When parallel query does a hash join, each backend constructs its own copy of the hash table. It uses N copies of the memory and N times the CPU.

- Such a plan can be beneficial, if there is sufficient work_mem to accommodate hash table in memory.

# Parallel Query capabilities in 9.6

- Parallel Aggregates

```
Finalize Aggregate
  -> Gather
     -> Partial Aggregate
        -> Nested Loop
           -> Parallel Seq Scan on foo
           -> Index Scan using bar_pkey
              Index Cond: (b = foo.b)
```

Here, we have invented a new PartialAggregate node that outputs transition states instead of the final aggregate results, and a FinalizeAggregate node that combines multiple sets of transition states into a final result.  We can't just push the aggregation step below the Gather.

# Contents

- Parallel Query capabilities in 9.6
- Tuning parameters
- Operations where parallel query is prohibited
- TPC-H results
- Parallel Query capabilities in pipeline

# Tuning parameters

- max_parallel_workers_per_gather must be set to a value greater than 0. A value between 1 and 4 is recommended.

- If user expects that query can benefit from parallelism, then try by reducing the value of below parameters:
  - parallel_setup_cost: planner's estimate for launching parallel workers and initializing dynamic shared memory.
  - parallel_tuple_cost: planner's estimate of the cost of transferring one tuple from a parallel worker process to another process.
  - min_parallel_relation_size: the minimum size of relations to be considered for parallel scan.

# Tuning parameters

- CREATE TABLE … WITH (parallel_workers = 2);  storage parameter parallel_workers sets the number of workers that should be used to assist a parallel scan of the table.

- Parallel workers are taken from the pool established by max_worker_processes.

- If the actual number of workers used for execution are less than the number of workers planned for query, then you can try by increasing the value of max_worker_processes.

# Contents

- Parallel Query capabilities in 9.6
- Tuning parameters
- When parallel query is prohibited
- TPC-H results
- Parallel Query capabilities in pipeline

# When parallel query is prohibited

- The sql statement that writes data or locks any database rows.
- In any situation in which the system thinks that partial or incremental execution might occur, no parallel plan is generated. For example, a cursor created using DECLARE CURSOR will never use a parallel plan.
- The query uses any function or aggregate marked PARALLEL UNSAFE.
- Parallel aggregation is not supported for ordered set aggregates or when the query involves GROUPING SETS.

# When parallel query is prohibited

- The query is running inside of another query that is already parallel.
- The transaction isolation level is serializable.
- The system must not be running in single-user mode. Since the entire database system is running in single process in this situation, no background workers will be available.

# Contents

- Parallel Query capabilities in 9.6
- Tuning parameters
- Operations where parallel query is prohibited
- TPC-H results
- Parallel Query capabilities in pipeline

# Test setup

- https://github.com/tvondra/pg_tpch
- TPC-H benchmark from http://tpc.org/tpch/default.asp
- IBM POWER8 box
- PostgreSQL9.6
- Non-default settings shared_buffers=8GB; work_mem=64MB
- max_parallel_workers_per_gather = 0 vs 4
- 20GB of input data, 43GB database size

# Summary of results

- With parallel query enabled, 15 plans used parallelism.  The other 7 query plans did not change.
- All the 15 queries that used parallelism got faster.

# Query results

- Q1: 341 seconds → 77 seconds (4.4x)
- Q3: 73 seconds → 38 seconds (1.9x)
- Q4: 14 seconds → 10 seconds (1.4x)
- Q5: 75 seconds → 38 seconds (1.9x)
- Q6: 24 seconds → 11 seconds (2.2x)
- Q7: 69 seconds → 31 seconds (2.2x)
- Q8: 17 seconds → 8 seconds (2.1x)
- Q9: 115 seconds → 112 seconds (1.02x)

# Query results

- Q10: 59 seconds → 31 seconds (1.9x)
- Q12: 61 seconds → 19 seconds (3.2x)
- Q16: 24 seconds → 23 seconds (1.04x)
- Q17: 191 seconds → 91 seconds (2.1x)
- Q19: 35 seconds → 14 seconds (2.5x)
- Q21: 163 seconds → 129 seconds (1.3x)
- Q22: 79 seconds - > 63 seconds (1.3x)

# Take-Away

- Linear scaling with 4 workers would result in a 4.4x speedup; only 1 of the 22 queries achieved that. Only 1 query had a speedup of 3x.

- Each of the 15 queries that used parallelism consumed up to 5x the resources to produce a speedup that was sometimes much less than 5x.

- 9 of those 15 queries ran close to twice or more than twice as fast, which is awesome.

# Contents

- Parallel Query capabilities in 9.6
- Tuning parameters
- Operations where parallel query is prohibited
- TPC-H results
- Parallel Query capabilities in pipeline

# Gather Merge

```
Sort
  -> Gather
     -> Parallel Seq Scan on foo
```

becomes

```
Gather Merge
 -> Sort
    -> Parallel Seq Scan on foo
```

- The Gather Merge node would assume that the results from each worker are ordered with respect to each other and then do a final merge over those.
- Can help cases where we need to sort tuples after scan (both for seq and index scans).

# TPC-H Q9 – Plan Without Gather Merge

```
Limit  (97.9 s, 1 row)
  ->  GroupAggregate  (97.9 s, 1 row)
        ->  Sort  (97.8 s, 11440 rows)
              ->  Hash Join (74.7 s, 3246126 rows)
                    ->  Nested Loop  (37.5 s, 3246126 rows)
                          ->  Hash Join  (2.6 s, 432928 rows)
                                      ->  Hash Join  (2.2 s, 432928 rows)
                                            ->  Gather  (1.6 s, 432928 rows)
                                                  Workers Launched: 4
                                                   ->  Nested Loop (1.01 s, 86586 rows, 5 loops)
                                                         ->  Parallel Seq Scan on public.part  (0.4 s, 21646 rows, 5 loops)
                                                         ->  Index Scan using idx_partsupp_partkey  (0.023 ms, 4 rows, 108232 loops)
                                            ->  Hash (0.14 s, 100000 rows)
                                                  ->  Seq Scan on public.supplier  (69.6 ms, 100000 rows)
                                ->  Hash  (0.058 ms, 25 rows)
                                      ->  Seq Scan on public.nation   (0.026 ms, 25 rows)
                          ->  Index Scan using idx_lineitem_part_supp (0.072 ms, 7 rows, 432928 loops)
                    ->  Hash  (20.4 s, 15000000 rows)
                          ->  Seq Scan on public.orders  (10.2 s, 15000000 rows)
Execution time: 98206.793 ms
```

# TPC-H Q9 – Plan With Gather Merge

```
Limit (52.5 s, 1 row)
  ->  Finalize GroupAggregate (52.5 s, 1 row)
        ->  Gather Merge (52546.571, 6 rows)
              Workers Launched: 4
              ->  Partial GroupAggregate (50.9 s, 79 rows, 5 loops)
                    ->  Sort (49.6 s, 234178 rows, 5 loops)
                          ->  Hash Join (45 s, 649225 rows, 5 loops)
                                ->  Nested Loop (13.3 s, 649225 rows, 5 loops)
                                      ->  Hash Join (1.9 s, 86586 rows, 5 loops)
                                            ->  Nested Loop (1.3 s, 86586 rows, 5 loops)
                                                  ->  Parallel Seq Scan on public.part  (0.6 s, 21646 rows, 5 loops)
                                                  ->  Index Scan using idx_partsupp_partkey (0.031 ms, 4 rows, 108232 loops)
                                            ->  Hash (248 ms, 100000 rows, 5 loops)
                                                  ->  Hash Join (167 ms, 100000 rows, 5 loops)
                                                        ->  Seq Scan on public.supplier (48.532 ms, 100000 rows, 5 loops)
                                                        ->  Hash (0.044 ms, 25 rows, 5 loops)
                                                              ->  Seq Scan on public.nation (0.025 ms, 25 rows, 5 loops)
                                      ->  Index Scan using idx_lineitem_part_supp (0.118 ms, 7 rows, 432928 loops)
                                ->  Hash  (22.5 s, 15000000, 5 loops)
                                      ->  Seq Scan on public.orders (11.5 s, 15000000 rows, 5 loops)
Execution time: 52613.132 ms
```

# Gather Merge

- The results on previous slides are based on the results postedon hackers with initial patch of Gather Merge.

# Parallel Bitmap Scans

```
Bitmap Heap Scan on foo
  -> Bitmap Index Scan on idx_foo
```

becomes

```
Gather
   -> Parallel Bitmap Heap Scan on foo
         -> Bitmap Index Scan on idx_foo
```

- One backend builds the TIDBitmap and then all the workers collaborate to scan the table.

- Benefits are visible upto 4 workers, after that Parallel Seq Scan plan gives more benefit.

# TPC-H Q6 – Serial Plan

Limit  (actual time=40921.437..40921.438 rows=1 loops=1)

-> Aggregate  (actual time=40921.435..40921.435 rows=1 loops=1)

-> Bitmap Heap Scan on lineitem  (actual time=7032.075..38997.369 rows=1140434 loops=1)

Recheck Cond: (..)

-> Bitmap Index Scan on idx_lineitem_shipdate (actual time=6951.408..6951.408 rows=1140434 loops=1)

Index Cond: (..)

Execution time: 40922.569 ms

# TPC-H Q6 – Parallel Plan

Limit (actual time=21895.008..21895.009 rows=1 loops=1)

-> Finalize Aggregate (actual time=21895.006..21895.006 rows=1 loops=1)

-> Gather (actual time=21894.341..21894.970 rows=3 loops=1)

Workers Planned: 2

Workers Launched: 2

-> Partial Aggregate (actual time=21890.990..21890.990 rows=1 loops=3)

-> Parallel Bitmap Heap Scan on lineitem (actual time=8517.126..21215.469 rows=380145 loops=3)

Recheck Cond: (..)

-> Bitmap Index Scan on idx_lineitem_shipdate (actual time=8307.291..8307.291 rows=1140434 loops=1)

Index Cond: (..)

- Execution time: 21915.931 ms
- Note – This is based on the results shared on hackers along with initial patch.

# Parallel Index Scans

```
Index Scan using idx_foo on foo
 Index Cond: (c < 10)
```

becomes

```
Gather
   Workers Planned: 2
   Workers Launched: 2
   -> Parallel Index Scan using idx_foo on foo
        Index Cond: (c < 10)
```

- As of now, the driving table for parallel query is accessed by parallel sequential scan which limits its usage to a certain degree. Parallelising index scans would further increase the usage of parallel query in many more cases like Merge Joins having index scans on both sides.

# Improved Parallel Hash Joins

- Single shared hash table instead of one per backend.
- Can even use a parallel scan to populate the hash table.

```
Hash Join
-> Stuff
-> Hash
   -> Seq Scan
```

becomes

```
Hash Join
-> Stuff
-> Parallel [Shared] Hash
   -> Parallel Seq Scan
```

# Parallel Append

- In 9.6

```
Gather
   Workers Planned: 2
   ->  Append
         ->  Parallel Seq Scan on t1
         ->  Parallel Seq Scan on t2
```

- Improved Plan

```
Gather
   Workers Planned: 2
   ->  Append
         ->  Parallel Seq Scan on t1
         ->  Seq Scan on t2
```

# Parallel Append

- Currently, the parallelism for Append node is sub-optimal as each worker runs each of the partial plan serially.
- In the previous example, first all the workers complete the scan on t1 and then on t2.
- Allow workers to run the partial or non-partail nodes parallely.
- This will allow us parallelize the I/O when tables are on separate disks.

# Parallel Maintenance / DDL Commands

- Vacuum
  - Parallel Heap Scan
  - Worker Per Index
- Create Index
  - Parallel-aware tuplesort

- Thanks to Robert Haas who has presented the paper on Parallel Query in PostgreSQL in PGConf US 2016. Some of the slides in this paper are from his paper. You can download his slides from https://sites.google.com/site/robertmhaas/presentations

Thanks!