Parallel Query in PostgreSQL:

How not to (mis)use it?

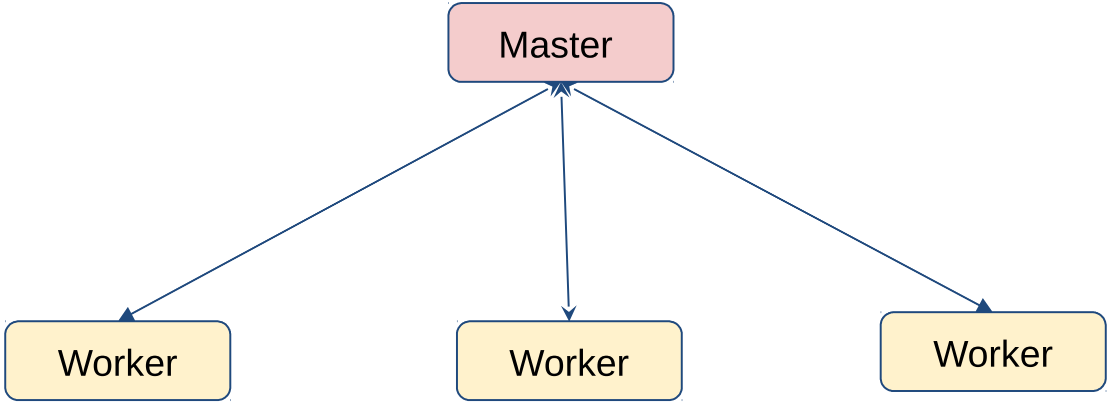
Dilip Kumar | PGCONF.INDIA 2019

Overview

- Intra-query parallelism till PG v11
  - Parallel-query flow in PG
  - Supported parallel operators
- How to get most from parallel query
  - Tuning parameters
  - Dos and don’ts of parallel operators
- Comparison with contemporary database engines for “parallel infrastructure”
- Take away

Parallel Query flow: Scans and Aggregates

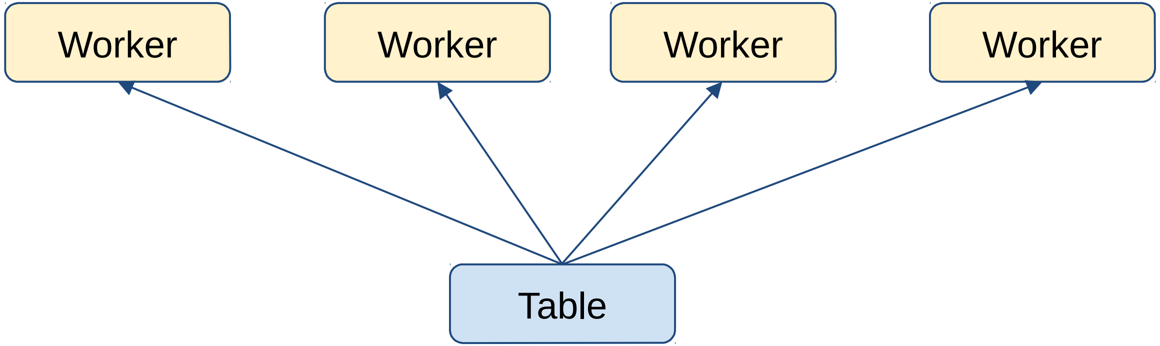
1. Master spawns the required number of workers and also works as one of the workers.
2. Each worker scans part of the relation and together they scan the complete table
3. The nodes below are the parallel ones and above it are the serial ones



Parallel Query flow: Scans and Aggregates

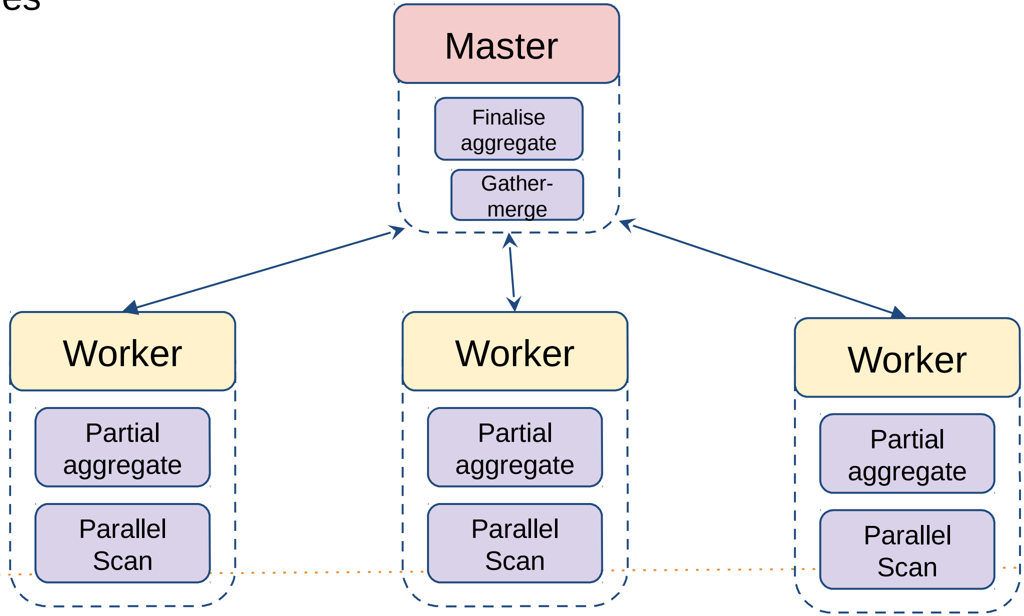
1. A number of workers are spawned once the decision to use parallel operator is made

2. The leader backend that spawns the workers runs the gather node which coordinates the task



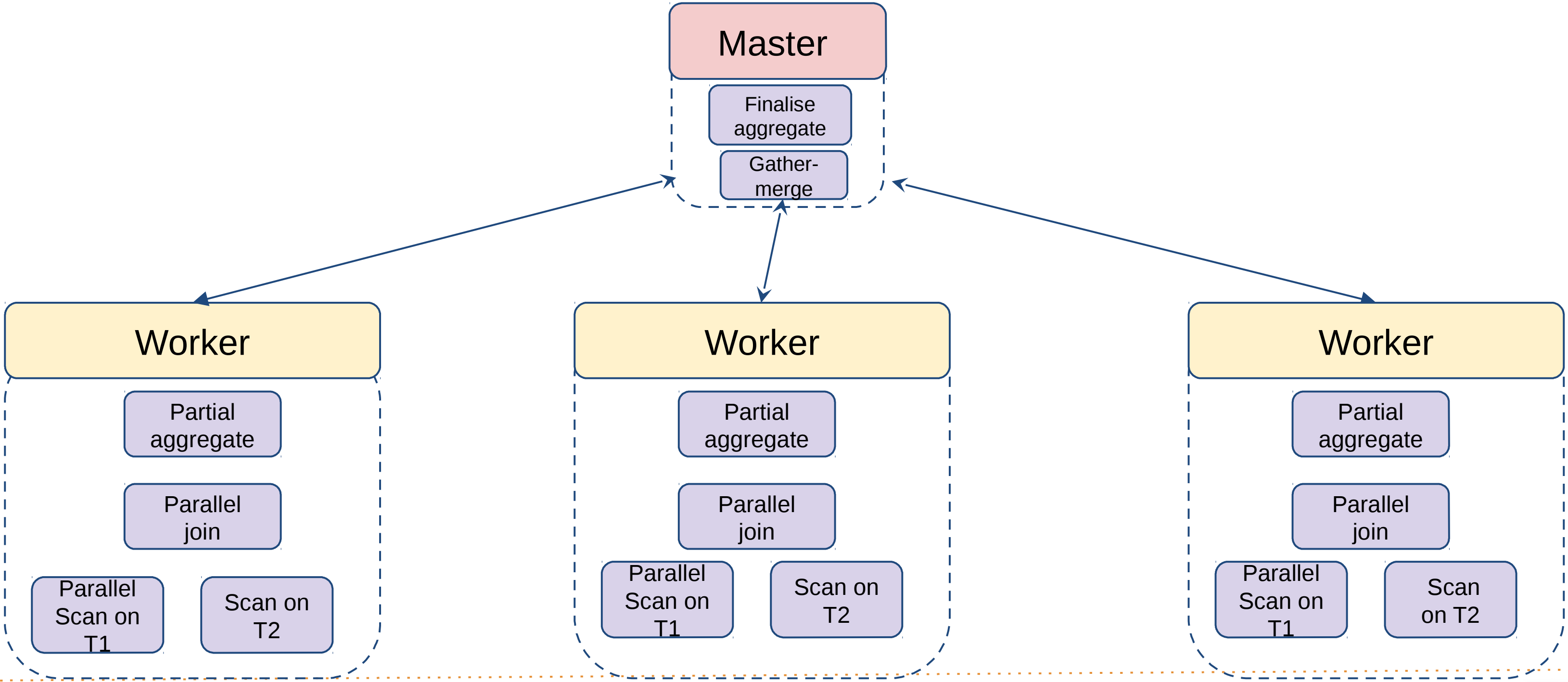
Parallel Query flow: Scans and Aggregates

1. Each of the worker performs the scan, apply the filter, etc. on the tuples of pages received by that worker
2. When completed it transfers the resultant tuples to the master
3. In case of aggregates, workers can only perform the aggregate on the tuples they received, hence master performs the final aggregate on the resultant tuples



Parallel Query flow: Scans and Aggregates

1. In case of joins, atleast one of the table is scanned by a set of parallel workers
2. Each worker then scans the inner table for the join
3. Resultant tuples are then passed to master



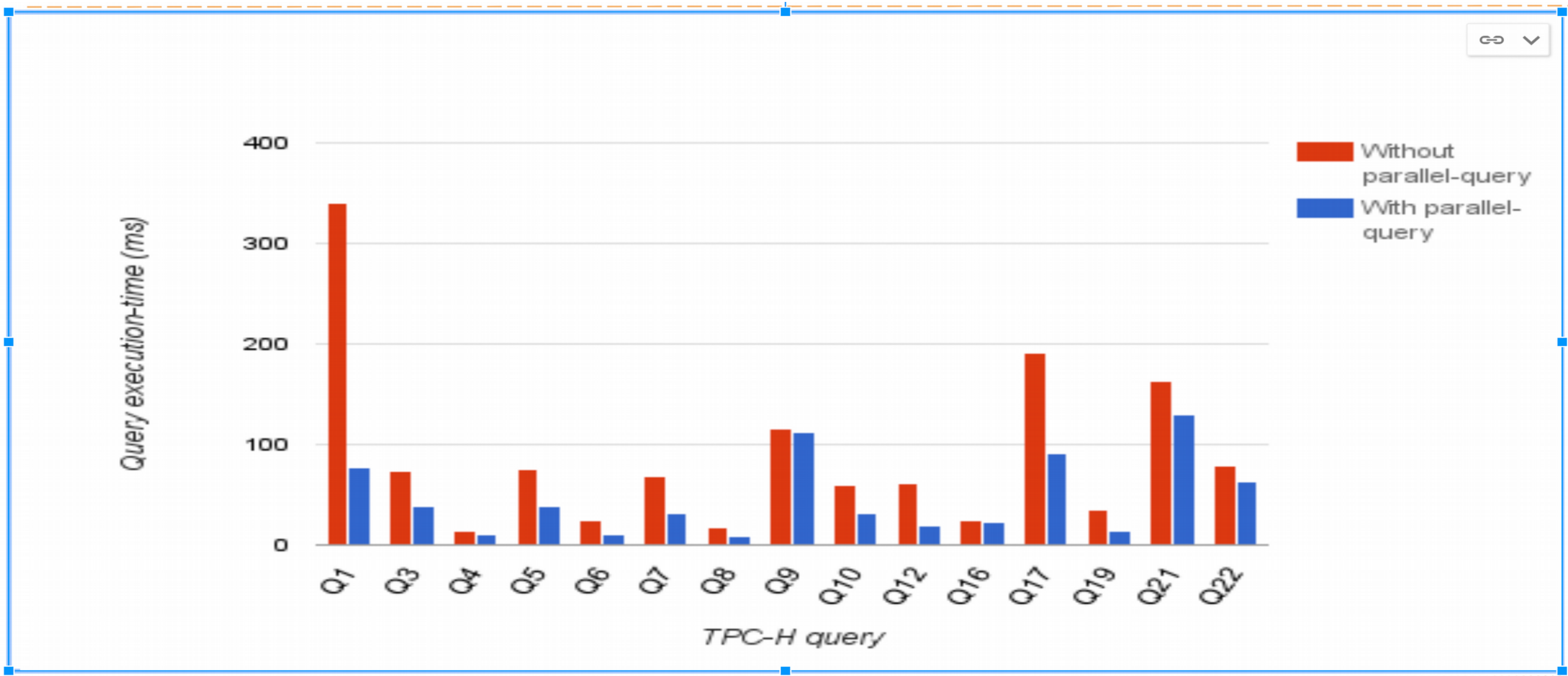
Parallel operators in PostgreSQL

- Parallel access methods
  - Parallel seq scan - **PG v9.6**
  - Parallel index, index-only scans, bitmap-heap scans - **PG v10**
- Parallel joins
  - NestedLoop and Hash joins - **PG v9.6**
  - Merge-join - **PG v10**, improved parallel hash join - **PG v11**
- Other parallel operators
  - Parallel aggregate - **PG v9.6**
  - Gather-merge, sub/init plans pushed to workers - **PG v10**, parallel append **- PG v11**
  - Parallel create index **- PG v11**

Performance evaluation of PG v10 on TPC-H

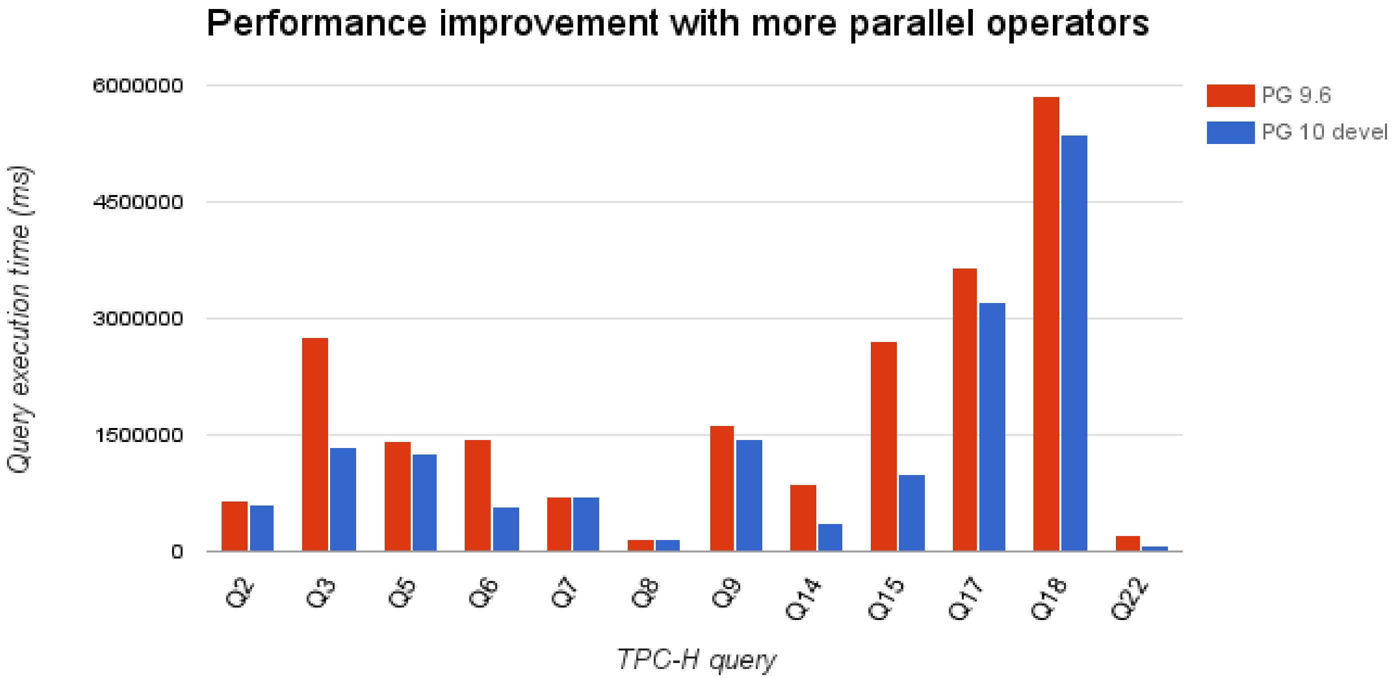
- Experimental setup
  - RAM = 512 GB
  - Number of cores = 32
- Parameter settings
  - Work_mem = 64 MB
  - Shared_buffers = 8 GB
  - Effective_cache_size = 10 GB
  - Random_page_cost = seq_page_cost = 0.1
  - Max_parallel_workers_per_gather = 4
- Database setup
  - Scale factor = 300
  - Additional indexes l_shipmode, l_shipdate, o_orderdate, o_comment

Performance evaluation of PG v9.6 on TPC-H



Results on scale factor 300

Performance evaluation of PG v10 on TPC-H



Results on scale factor 300

Tuning parallelism

- Parallel query specific parameters
  - `max_parallel_workers_per_gather`
  - `parallel_tuple_cost`
  - `parallel_setup_cost`
  - `min_parallel_table_scan_size`
  - `min_parallel_index_scan_size`
  - `parallel_leader_participation`
  - `parallel_workers`

- Other parameters
  - `work_mem`
  - `effective_cache_size`
  - `random_page_cost`

Tuning parallelism

- Max_parallel_workers_per_gather
  - Number of workers per node for a parallel operator
  - Recommended value 1 to 4
  - The ideal value of this parameter is determined by number of cores in the system and the work required at a node
    - E.g. If *the number of cores is 8* but the work required at node is enough for *2 workers* only then increasing this parameter will not help
    - Similarly, if the *number of cores is 2* and we increased this *parameter to 10*, then it’s likely to cause *degradation in performance*

Tuning parallelism

- parallel_tuple_cost
  - planner's estimate of the cost of transferring one tuple from a parallel worker process to another process
- parallel_setup_cost
  - planner's estimate for launching parallel workers and initializing dynamic shared memory
- We can lower the values of these parameters to diagnose the performance of parallel operators

Tuning parallelism

- min_parallel_table_scan_size
  - Minimum size of relations to be considered for parallel sequence scan
  - The default value of this parameter is 8MB
  - If the database mostly has large tables then it is better to increase this parameter
  - For diagnostic purposes we can decrease it to lower values to analyse the query plans, etc.

- min_parallel_index_scan_size
  - the minimum size of index to be considered for parallel scan
  - The default value is 512kB

Tuning parallelism

- parallel_leader_participation
  - Manage the involvement of the leader process
  - Queries which require to maintain the order of tuples from workers, might need the leader to work on that more than scanning a part of leader
  - When too many workers are there it might be good to keep leader free for the management of workers

- parallel_workers
  - Alter table <table_name> set (parallel_workers =<n>)
  - Control the degree of parallelism for each table, if required

Tuning parallelism

- work_mem
  - Amount of memory given to per worker per node
- random_page_cost
  - Estimated cost of accessing a random page in disk

When not to expect performance improvements from...

Parallel Sequential Scan

- Too small table
  - Lesser than the min_parallel_table_scan_size

- Too less tuples filtered out
  - Additional costs
    - dividing the work among workers
    - collection of tuples from workers
  - If the number of workers is not high enough, the additional cost of parallelism could be more than the non-parallel scan

Parallel Index Scan

- □ Size of index is too small
- □ Number of leaf pages in the index range is small
- □ All the tuples qualify the index filter

- □ Index is non-BTree
  - □ `Currently not supported`

Parallel Bitmap Heap Scan

- Size of bitmap is small
- Bitmap index scan is costlier than bitmap heap scan
  - The bitmap index scan is not supported in parallel, only bitmap-heap scan can be divided among workers
- Most of the tuples satisfy the qual
  - Though the size of bitmap is big enough, the benefit of parallelism cannot be achieved if most of the tuples are sent to the gather

Parallel Merge Join

- □ Inner relation is not small enough
  - □ Ideally, the size of inner relation should be LEQ work_mem/total number of workers, otherwise the amount of memory used might be unexpected
  - □ Every worker will keep a copy of the inner relation
  - □ If the copy does not fit in memory it will be send to the disk which will increase the I/O cost

Parallel Aggregates

- Number of groups is too high
  - The final aggregate is to be performed at the gather node
  - So, it is almost same as gather is performing the aggregate alone
- Early aggregation is not possible
  - E.g. average of an attribute
- Aggregate functions is parallel unsafe or restricted

Gather-merge

- Workers contain mutually exclusive tuple ranges
  - Gather-merge can only accept rows from one worker at a time to maintain the order of tuples
  - The remaining workers need to halt till the shared queue is empty to complete their further processing
  - The processing is similar to as if workers are giving the tuples in serial fashion

More workers may not mean better performance

Amdahl’s law

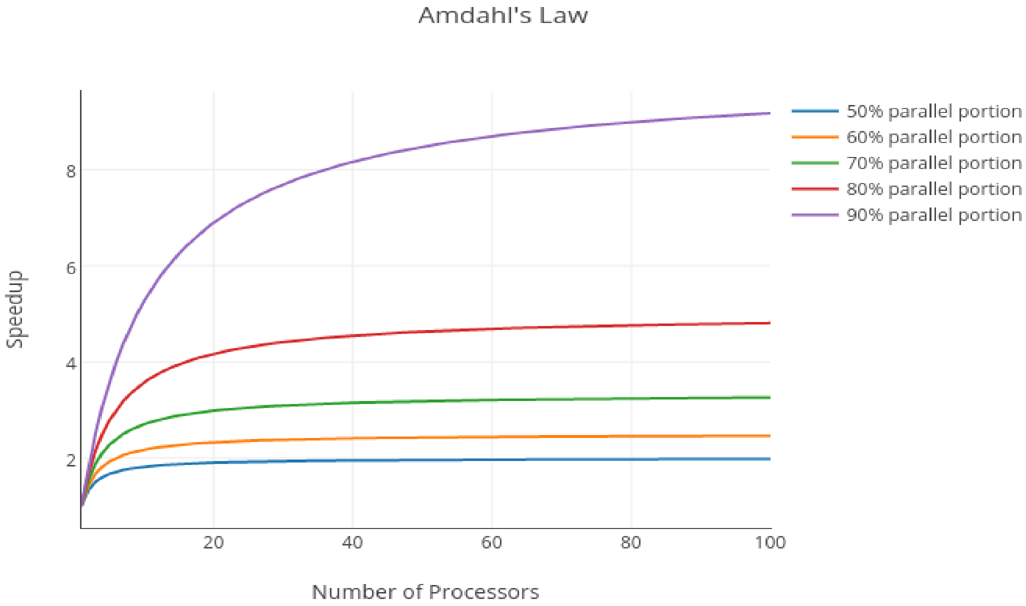
$$\text{Speedup}(N) = \frac{1}{(1-P)+\frac{P}{N}}$$

Serial part of job =

1 (100%) - Parallel part

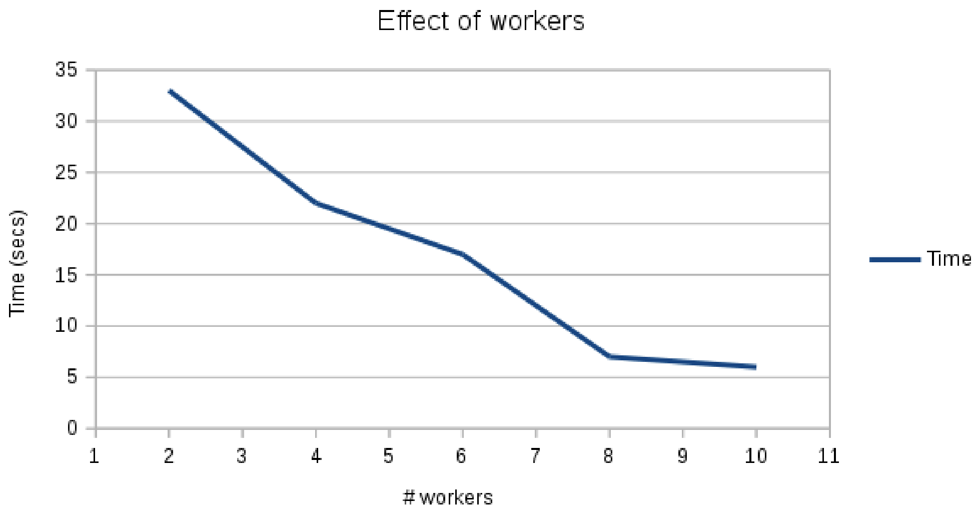
Parallel part is divided

up by N workers



Too many cooks spoil the broth!

- More workers may not translate to better performance
  - If the amount of work is limited to be distributed among n workers, the n+k workers are not going to help
  - Don’t believe us, see the results for yourself



TPC-H

- Scale factor = 50
- Additional indexes on l_shipmode, l_shipdate, o_orderdate, and o_comment
- Q6

Server settings

- random_page_cost = seq_page_cost = 0.1
- effective_cache_size = 10GB
- shared_buffers = 8GB
- work_mem = 1GB

Parallel-query architecture

PostgreSQL

Vs

Other engines

Parallel query architectures in PG

- Dynamic background worker (shm_mq)
  - Postmaster can launch the background worker processes at run time
- Dynamic shared memory
  - Allocate a chunk of memory that can be shared among co-operating processes
  - Shared memory table of contents
    - A simple scheme for carving DSM into numbered chunks
- Shared messaging capabilities
  - Shared memory message queue
  - For error/notice forwarding
    - Tuple queue reader and DestReceiver

Parallel query architectures in PG

- Parallel context
  - Core toolkit for parallel operations
  - Launch a number of workers, establish “useful” state, run C code you specify and ensure timely shutdown.
- State synchronization
  - To ensure same GUC values, libraries, and transactions with same snapshot across workers
- Group locking
  - To solve the issue of undetected deadlock
  - Leader and its workers are treated as one entity for locking purposes.

Parallel query architectures in PG

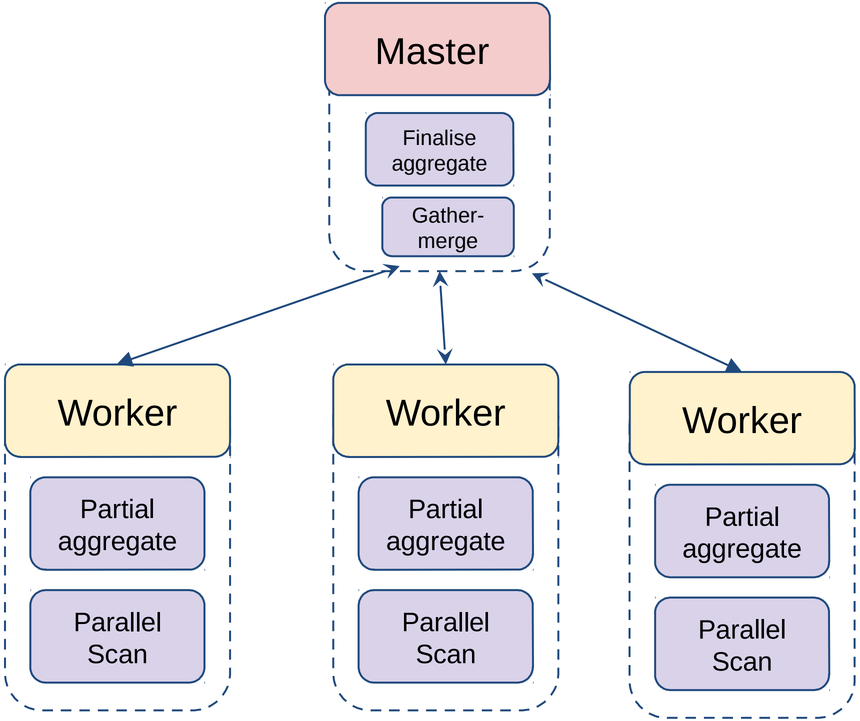
- Parallel executor support
  - Pass the plan state tree for execution to workers
  - Execute a plan by a set of workers
- Parallel aware executor nodes
  - Different behaviours when run in parallel and otherwise
- Gather (Merge)
  - Collect results across all workers and merge them into a single result stream
  - Collect all the instrumentation information across all workers and show the aggregated information

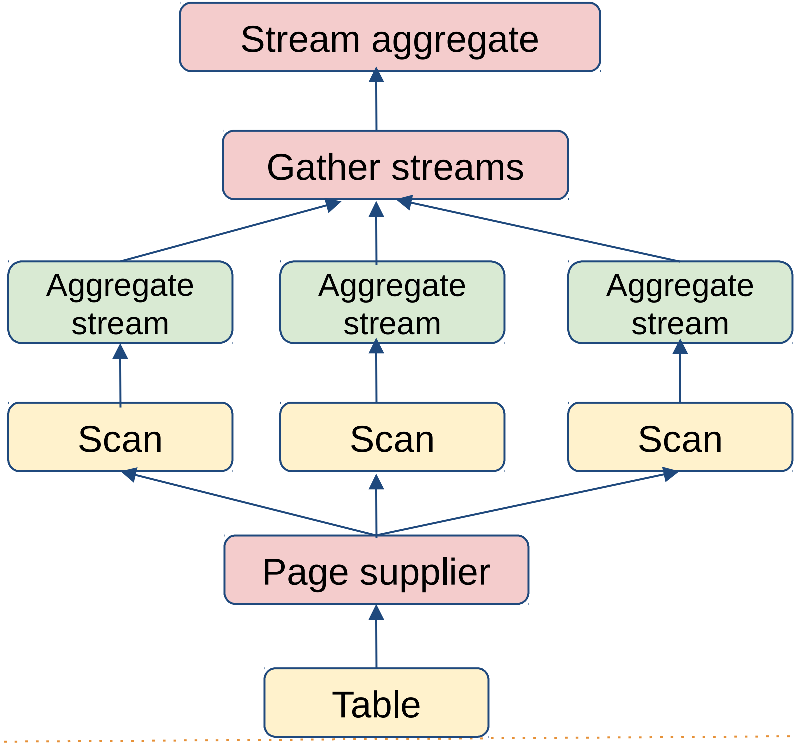
Parallel infrastructure: PG vs other engines

1. Exchange operators
   - ❑ Gather, worker
2. Each operator needs to have a parallel version as a new operator
   - ❑ Parallel scans, joins, aggregate
3. Tuples can only flow between workers and gather
   - ❑ A new node called gather-merge is used to maintain order of rows
   - ❑ Final aggregation can be done by gather only
4. Well-suited for multi-process architecture of PostgreSQL

1. Exchange operators
   - ❑ Distribute, gather, and repartition
2. Exchange operators can be placed over any operator
   - ❑ Any scan can be parallelised by placing distribute and gather operators over it
   - ❑ Aggregate can be parallelised by repartition and gather operators
3. Tuples can be routed among the streams
   - ❑ To maintain order of rows
   - ❑ For efficient aggregation
4. Advocated for any multi-threaded architecture based models

Parallel scan: PG vs other engines





Case-study: Parallel Hash join

1. All the workers work together to create the complete hash table
   - ❑ The total work of creating the hash-table is divided among the workers
2. Once the hash is prepared, each worker can probe it to perform the join of a tuple it received
   - ❑ This probing is lock or contention free
   - ❑ Total tuples of outer relation are divided among workers hence dividing the total work
3. A smart move to overcome tuple routing mechanism

1. Each stream performs a small hash join which is later combined for the final result
   - ❑ Each stream gets a well defined range of join
   - ❑ Any stream can route a tuple if it receives some that belong to the range of other stream
   - ❑ Total work of hash-join is divided among the streams

Parallel query architectures: at a glance

- In PostgreSQL, parallel-query architecture allows less communication among worker nodes, but more work per-node. This is more suited to process based architecture where inter-process communication cost is higher

- The other architecture described has more communication among workers, but less work per node which could be more-suited to thread-based architecture where there is almost no inter-process communication cost

- This conclusion is just based on our understanding of the systems

Scope of enhancements in parallel query

- Parallel bitmap index scan
- Parallel sort
  - Can improve performance for lengthy sorts
  - Costly order by is common in OLAP environments
- Parallel materialize node
  - Like shared hash, we can have one shared copy of inner table in parallel for efficient (merge/nested-loop) joins
- Improvements in parallel aggregate
  - Perform final aggregate in shared memory
  - This could remove the bottleneck of finalize aggregate
- Improvements in query optimizer
  - Improve the costing for parallel operators

Conclusion

- Remarkable performance improvements with parallel operators on TPC-H
  - Till v9.6 out of 22 queries of TPC-H, performance improved for 15 queries, in which 3 queries are at least 4 times faster and 11 queries are 2 times faster
  - Further in v10, around 10 of 22 TPC-H queries show significant improvement in performance, in which around 4 queries show more than 2x improvement

- Tuning parameters for parallelism
  - Parallel-query specific parameters
  - Other server-side parameters

- Parallel operators fail to improve performance when…
- Too many workers may not always improve query performance

- Comparison with other database engines for parallel query architecture

People who contributed

- Robert Haas
- Amit Kapila
- Thomas Munro
- Dilip Kumar
- Peter Geoghegan
- Andres Freund
- David Rowley
- Rushabh Lathia
- Rahila Syed
- Amit Khandekar
- Kuntal Ghosh
- Rafia Sabih

Special thanks to Tom Lane for helping in the fixes of many bugs

```
Output: Thank You
Gather
   Workers Planned: 2
   Workers Launched: 2
   ->  Parallel Index Scan on Common_phrases
         Index Cond: ( value = ‘Thank You’ )
          Filter: Language = ‘English’
```

Slide credits:
[1] https://www.pgcon.org/2016/schedule/events/913.en.html
[2] https://www.postgresql.eu/events/schedule/pgconfeu2016/session/1360-parallel-query-in-postgresql/
[3] http://pgconf.in/schedule/query-parallelism-in-postgresql-expectations-and-opportunities/