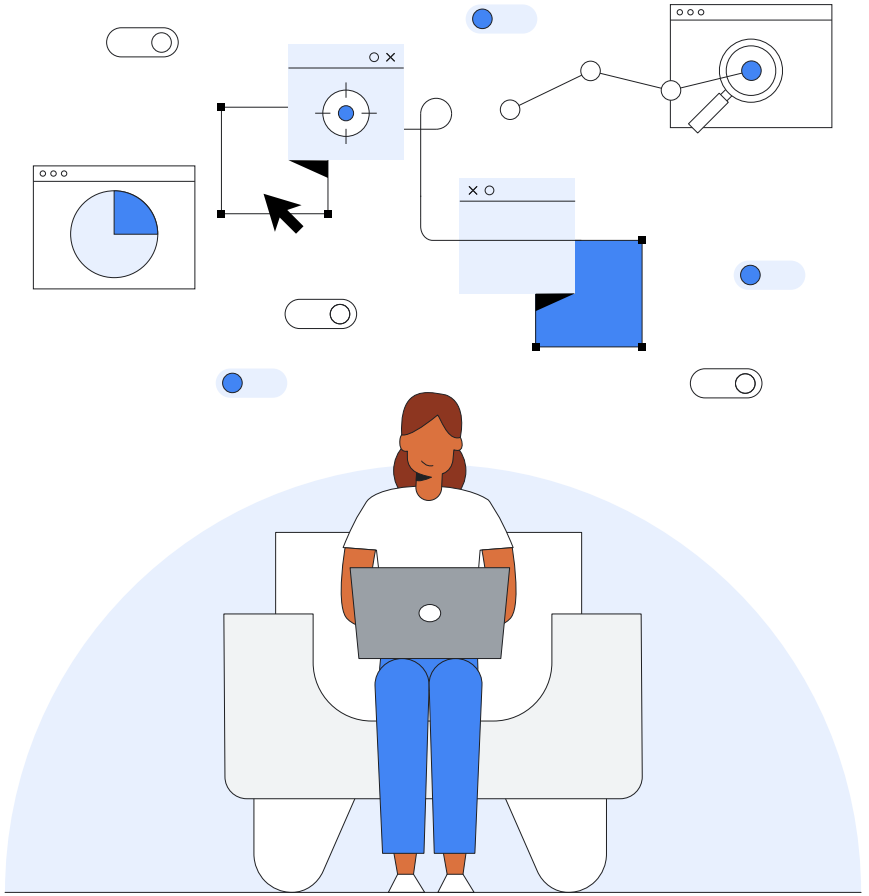
Google Cloud

PostgreSQL

Advance Performance Tuning



Somdyuti Paul (Som)
Google, EMEA DataManagement Specialist

Contents

Parameters/Flags

Configure Memory Related Parameters

Oracle	Postgres	Recommended Values in CloudSQL	Comment	Needs Restart in CloudSQL	Effect on System Resources	Recommended to modify in Postgres
sga_target	shared_buffers	30-50% of RAM	Default in CSQL-33% of RAM,50% in AlloyDB	Yes	Higher value will reduce disk I/O but anything > 70% of RAM can cause swapping	Yes-if workload is big enough; there is more disk access.
memory_target	shared_buffers+work_mem+maintenance_work_mem	shared_buffers=30-50% of RAM, work_mem=16MB-128MB, maintenance_work_mem= 64MB to 2 GB	Maintenance_work_mem can be set up to 20% of RAM for applications doing large ETL; for large bulk vacuums	Yes(shared_buffers), No(work_mem,maintenance_work_mem)	Higher values will lead to more RAM Usage	Yes- maintenance_work_mem,larger settings might improve performance for vacuuming, bulk index build and for restoring database dumps.Higher value recommended for apps doing large ETL operations.
pga_aggregate_target	work_mem/hash_mem_multiplier	Work_mem to 16MB-128MB, hash_mem_multiplier between 2-8 especially for workloads doing many/large hash operations	Default 4MB needs to be changed in Production workload	No	Higher value will increase memory usage, be aware of concurrent sessions performing huge sorts, hash joins.	Yes- work_mem should be set higher if the workload is performing a lot of sorts, hash operations.
log_buffers	wal_buffers	On very busy, high-core machines it can be useful to raise this to 64 MB - 128MB	Default is approx < 1% of shared_buffers	Yes	I/O. Default value can be too low for a busy OLTP system as this will increase commits. Higher value will reduce commit latency.	Yes-WAL Records are flushed from WAL Buffers to Disk not only during transaction commit but also when WAL buffers get filled.

Google Cloud

Tips:-Oracle DBs with Large SGA and/or low Buffer Cache hit ratio will benefit with **higher values for shared_buffers. Write intensive workloads will benefit with larger log_buffers**

Configure Checkpointing Parameters

Oracle	Postgres	Recommended Values in CloudSQL	Comment	Needs Restart in CloudSQL	Effect on System Resources	Recommended to modify in Postgres
log_checkpoint_interval	max_wal_size	By default checkpoint_timeout is 5 mins and max_wal_size is 1.5GB in CloudSQL and AlloyDB So a checkpoint will happen either every 5 minutes or when 1.5GB of WAL Records/segments are generated	In a busy prod system MAX_WAL_SIZE can be increased to 5-30GB so that checkpoints do not happen that frequently.	No	I/O. Lower value will increase checkpoint frequencies causing more I/O. Too much higher value (> 30 GB) may increase database recoverability time.	Yes. Set it to a value such that checkpoints happen every 2-5 minutes.
log_checkpoint_timeout	checkpoint_timeout	5-8 minutes (default is 5 min).	Set **checkpoint_completion_target** to 0.9 to evenly spread the IOs between 2 checkpoints.	No	I/O.	No. Default of 5 minutes is good enough
log_checkpoints_to_alert	log_checkpoints	On	Log checkpoint messages to Postgres log	No (dynamic)	More disk space usage.	No. Turn it off only when disk space is becoming an issue due to excessive untuned checkpoints

Google Cloud

Tips:-max_wal_size should be configured properly to reduce checkpoint frequency.

Configure I/O Parameters

Oracle	Postgres	Recommended Values in CloudSQL	Comment	Needs Restart in CloudSQL	Effect on System Resources	Recommended to modify in Postgres
commit_write/commit_wait	commit_delay	improve group commit throughput by allowing a larger number of transactions to commit via a single WAL flush. If Oracle uses Batch commit then setting commit_delay to 5000-1000 can be useful.	Change it only when Source Oracle uses Batch commit and/or Nowait option.Useful for Batch operations/Batch Load	No (Dynamic)	I/O. If set then reasonable values are 200 to 1000	No. Do not change it until and unless absolutely necessary.
db_file_multiblock_read_count	Seq_page_cost, random_page_cost	If MBRC is higher then set seq_page_cost to lower value (0.5-1.0) or increase random_page_cost (2.0-4.0)	If db_file_multiblock_read_count is high then increasing random_page_cost (default is 4) relative to seq_page_cost(default 1) will make optimizer prefer Full Table scans	No	Do not set random_page_cost < seq_page_cost until and unless most of the indexes can fit in the buffer cache.	Yes. random_page_cost of 1.1 recommended for production workloads using SSDs.
disk_asynch_io/filesystemio_options	effective_io_concurrency	If disk_asynch_io is set to True or filesystemio_options set to SETALL or ASYNC set effective_io_concurrency to a higher value for the database	Check Source aio-nr and aio-max-nr kernel settings and set effective_io_concurrency to 500-1000.Effects bitmap heap scans	Cannot be set at Instance level. Set at Database level	I/O. Do not set > 500. 200-500 is good enough if this needs to be set.	No. Set it only when the execution of parallel bitmap scans is slow.

Tips:- random_page_cost should not be more than 1.5x times than seq_page_cost for small random reads workloads.

Configure Database Access Path Parameters

Oracle	Postgres	Recommended Values in CloudSQL	Comment	Needs Restart in CloudSQL	Effect on System Resources	Recommended to modify in Postgres
optimizer_index_cost_adj	random_page_cost	If Oracle's optimizer_index_cost_adj is lower (towards 1) then the planner will prefer index scans. Set random_page_cost to 1.1 if SSDs are used.	Reducing random_page_cost (default is 4) relative to seq_page_cost(default 1) will make optimizer prefer index scans	No	I/O. Smaller values are recommended if workload is typically small random reads.	Yes. It should not be altered unless you're using special storage (SSDs, high end SANs, etc.)
optimizer_index_caching	effective_cache_size	a higher value makes it more likely index scans will be used, a lower value makes it more likely sequential scans will be used. If optimizer_index_caching is close to 100, set effective_cache_size to a higher value.	Set it to higher value to favor index scans and if optimizer_index_caching is set to higher value.	No	No effect. It just tells the PostgreSQL query planner how much RAM is estimated to be available for caching data, in both shared_buffers and in the filesystem cache. This setting just helps the planner make good cost estimates; it does not actually allocate the memory.	Yes.
db_file_multiblock_read_count	Seq_page_cost, random_page_cost	If MBRC is higher then set seq_page_cost to lower value (0.5-1.0) or increase random_page_cost (2.0-4.0)	If db_file_multiblock_read_count is high then increasing random_page_cost (default is 4) relative to seq_page_cost(default 1) will make optimizer prefer Full Table scans	No	Do not set random_page_cost < seq_page_cost until and unless most of the indexes can fit in the buffer cache.	Yes. random_page_cost of 1.1 recommended for production workloads using SSDs.

Google Cloud

Tips:- effective_cache_size is often an ignored parameter in Postgres. Set it to **higher value to make use of FS Cache.**

Configure Parallelism Parameters

Oracle	Postgres	Recommended Values in CloudSQL	Comment	Needs Restart in CloudSQL	Effect on System Resources	Recommended to modify in Postgres
parallel_max_servers	Max_parallel_workers. Related-max_parallel_workers_per_gather, max_parallel_maintenance_workers	Max_parallel_workers can be set to the number of vCPUs for workloads using high parallel queries	Set max_parallel_workers_per_gather to max_parallel_workers/2	No	CPU. Do not set more than the number of vCPUs	Yes. If workload is doing large sequential reads,parallel queries then higher value will improve performance.Setting max_parallel_maintenance_workers higher during large index builds and Vacuum on large tables.

Tips:-max_parallel_workers if configured properly is a champion setting for parallel execution in Postgres.

Google Cloud

Configure Performance related other Parameters

Oracle	Postgres	Recommended Values in CloudSQL	Comment	Needs Restart in CloudSQL	Effect on System Resources	Recommended to modify in Postgres
processes	Max_worker_processes and max_connections	Max_worker_processes- controls the background processes, can be set to 2*vCPUs, max_connections controls max number of concurrent connections. (> 300 recommended to use Connection pooling)	In Oracle processes control both foreground (user) and background processes. For postgres max_worker_processes control background processes	Yes	CPU and memory. max_worker_processes >= max_parallel_workers<= #vCPUs. max_connections must be set to 1.5x times the concurrent sessions you expect at peak load.	Yes. If you need more than 200 concurrent connections, make use of connection pooling
cursor_sharing	plan_cache_mode	If cursor_sharing=EXACT, plan_cache_mode=force_custom_plan. If cursor_sharing=FORCE, then plan_cache_mode=force_generic_plan	This flag is not modifiable at Instance level, can be set at database and/or session level	Dynamic- can be set at session level		No. Recommended to change only at session/database level if there are plan changes for sql for different bind variable values.
dml_locks	max_locks_per_transaction	The default value of 64 may not be sufficient. If dml_locks is high in source (default is four tables referenced for each transaction) then increase max_locks_per_transaction	If transaction touches many tables in single transaction, or touch large partitioned tables then set the value higher	Yes	Can be set to the number of partitions of the largest partitioned table.	No. Set it only when queries on large partitioned tables are slow and wait on LW Locks.

Google Cloud

Tips:-max_worker_processes control the background processes in Postgres. max_connections control the sessions established. For huge concurrent connections configure pgBouncer or HAProxy which will also help to distribute load between Read Replicas.

Checkpoints, Commits and Performance

Checkpointing and Commit in PostgreSQL-A little more..

- Checkpoints are determined by 2 parameters- CHECKPOINT_TIMEOUT and MAX_WAL_SIZE
- A checkpoint is begun every checkpoint_timeout seconds, or if max_wal_size is about to be exceeded, whichever comes first. The default settings are 5 minutes and 1 GB, respectively.
- If no WAL has been written since the previous checkpoint, new checkpoints will be skipped even if checkpoint_timeout has passed.
- Reducing checkpoint_timeout and/or max_wal_size causes checkpoints to occur more often. This allows faster after-crash recovery, since less work will need to be redone. However, one must balance this against the increased cost of flushing dirty data pages more often
- Usually in a busy prod system MAX_WAL_SIZE should be increased to 4–10GB (and sometimes to 30GB) so that checkpoints do not happen that frequently. **Ideally make checkpoints happen every 3–5 minutes.**

max_wal_size will need to be changed in production workload. 5GB is a good starting point. You can also use the redo generation/sec in source Oracle DB to more accurately configure the max_wal_size.

Checkpointing in PostgreSQL

- During checkpoint the CKPT process will flush/write dirty buffers/modified data and index pages from Buffer cache to Disk (Data files). Postgres will try to finish the checkpoint based on checkpoint_completion_target (default 0.9). **Setting this to a higher value will evenly spread the IOs between 2 checkpoints.**
- The Background Writer (BGWriter) also periodically flushes dirty pages from buffer cache to disk on a regular interval with sleeping time between 2 activities determined by **bgwriter_delay** (200ms default) and maximum number of LRU/Dirty pages to flush every round is determined by **bgwriter_lru_maxpages** (default 100 pages)
- So Background writer flushes no more than 100 dirty pages per round and sleeps for 200ms between every round (this is independent of the CKPT process which flushes dirty pages during checkpoint). You **can edit this parameter** to make bgwriter flushes more aggressive so that checkpoints have less work to do.

```
 bgwriter_lru_maxpages
-----------------------
 200
(1 row)

postgres=> show bgwriter_delay;
 bgwriter_delay
----------------
 50ms
(1 row)

postgres=> show max_wal_size;
 max_wal_size
--------------
 5GB
(1 row)
```

Optimized- BGWr doing more work

```
demo=> show bgwriter_lru_maxpages;
 bgwriter_lru_maxpages
-----------------------
 100
(1 row)

demo=> show bgwriter_delay;
 bgwriter_delay
----------------
 200ms
(1 row)
```

Default

To avoid flooding the I/O system with a burst of page writes, writing dirty buffers during a checkpoint is spread over a period of time determined by checkpoint_completion_target. Do not set it more than 0.9

Google Cloud

Commit in PostgreSQL

- WAL Writes- Backend processes write WAL records from WAL Buffers to File System buffer cache.
- WAL Flush- The WAL Records gets flushed/written to WAL Segments on Disk.
- Commit-> WAL Writes + WAL Flush (synchronous_commit)
- With async commit, the WAL Writer flushes the WAL records and NOT the Backend processes
- WAL Record Inserts (local): WAL records are first created in WAL buffers(**XLogInsertRecord**). Since multiple backend processes will be creating the WAL records at a time, it is properly protected by locks. The writing of WAL records in wal_buffers gets continuously written/flushed(**XLogFlush**) to WAL segments by different backend processes(WAL Writes). If the sychronous_commit is completely off, the flush won't be happening immediately but relies on wal_writer_delay settings
- How much data we lose if we opt for full asynchronous commit (synchronous_commit = off)
- The answer is slightly complex, and it depends on wal_writer_delay settings. By default it is 200ms. That means WALs will be flushed in every wal_writer_delay to disk. The WAL writer periodically wakes up and calls **XLogBackgroundFlush**(). This checks for completely filled WAL pages. If they are available, it writes all the buffers up to that point
- commit_delay-Sets the delay in microseconds between transaction commit and flushing WAL to disk

Increase wal_buffers if your workload is write-heavy and you see "WAL buffering" wait event.

Commit in PostgreSQL

- Flushes WAL Records from WAL Buffers (3% of shared_buffers) to WAL Files/Segments on disk (wal_segment_size=16MB) . If a transaction is too large and exceeds WAL Records > wal_buffer_size even uncommitted changes will get flushed to WAL Segments on disk. But during applying WAL Records to data files during crash/instance recovery only committed records since last checkpoint will get applied (the CLOG records help to identify committed transactions)
- PG 17- Increased the **WAL segment size from 16MB to 64MB.** This enhancement has resulted in a 10%-20% performance improvement with various workloads.
- So WAL Records are flushed from WAL Buffers to Disk **not only during transaction commit but also when WAL buffers get filled.**
- Every Checkpoint maintains a Checkpoint record in WAL Segments so that the WAL Records prior to the checkpoint record can be reused/deleted when WAL segments need to be overwritten. Also Archiving will need to archive only completely filled WAL Segments before they get overwritten/recycled. But WAL Segments can be switched without getting full either by setting archive_timeout or pg_switch_wal.

Higher value of wal_segment_size makes log switches and archiving less aggressive

Reduce Query Planning Time

PLAN_CACHE_MODE

- **Use Case- Complex queries,with lot of parameterized values ,on tables with uniform data distribution**
- PLAN_CACHE_MODE= AUTO (Custom Plan)- Custom plans are made afresh for each execution using its specific set of parameter values
- Generic plans do not rely on the parameter values and can be re-used across executions.
- Use of a generic plan saves planning time, but if the ideal plan depends strongly on the parameter values then a generic plan may be inefficient.
- Uniform data distribution and different bind variable values- Generic plan is always better
- Example (query with different parameter values passed/exec and plan_cache_mode=auto)

Buffers: shared hit=17896
Planning Time: 35.451 ms
Execution Time: 1.887 ms

- Set it for a particular user

```
alter user test_user set plan_cache_mode to force_generic_plan;
select * from pg_user where usename = 'test_user';
  usename  | usesysid | usecreatedb | usesuper | userepl | usebypassrls |  passwd  | valuntil |                                  useconfig
-----------+----------+-------------+----------+---------+--------------+----------+----------+-----------------------------------------------------------
 test_user |   429868 | t           | f        | f       | f            | ******** | NULL     | {"search_path=\"$user\", public,
oracle",pgaudit.log=all,plan_cache_mode=force_generic_plan}
```

```
SET SESSION plan_cache_mode=force_generic_plan;
PREPARE my_query(integer, varchar, varchar, varchar, timestamp, integer) AS
SELECT
COALESCE(COUNT(DISTINCT exltspline.spl_pax.splpax_seq_cdg), 0) AS levelclasifygoss
FROM exltspline.spl_tramos
INNER JOIN exltspline.spl_vuelos
ON DATE_TRUNC('day', exltspline.spl_tramos.vlos_fch) = DATE_TRUNC('day', exltspline.spl_vuelos.vlos_fch)
AND exltspline.spl_tramos.vlos_nmr = exltspline.spl_vuelos.vlos_nmr
AND exltspline.spl_tramos.vlos_cdg_carrier = exltspline.spl_vuelos.vlos_cdg_carrier
INNER JOIN exltspline.spl_tramos_pax
ON exltspline.spl_tramos_pax.spltrm_seq_cdg = exltspline.spl_tramos.spltrm_seq_cdg
INNER JOIN exltspline.spl_pax
ON exltspline.spl_pax.splpax_seq_cdg = exltspline.spl_tramos_pax.splpax_seq_cdg
INNER JOIN exltspline.spl_clasificaciones_goss
ON exltspline.spl_clasificaciones_goss.splpax_seq_cdg = exltspline.spl_pax.splpax_seq_cdg
AND exltspline.spl_clasificaciones_goss.splcsg_est_reconocido = 1
AND exltspline.spl_clasificaciones_goss.splcsg_tpo = 'G'
WHERE
exltspline.spl_vuelos.vlos_nmr = $1
AND exltspline.spl_vuelos.vlos_cdg_carrier = $2
AND exltspline.spl_tramos.grpr_cdg_origen = $3
AND exltspline.spl_tramos.grpr_cdg_destino = $4
AND exltspline.spl_vuelos.vlos_fch = $5
AND exltspline.spl_tramos.spltrm_seq_cdg = $6;
```

```
EXPLAIN (analyze, verbose, costs, settings, buffers, wal, timing, summary, format text) EXECUTE my_query(3, 'LA', 'SCL', 'GRU', DATE_TRUNC('day','2024-07-31 00:00:00.0'::timestamp), 4069367);
EXPLAIN (analyze, verbose, costs, settings, buffers, wal, timing, summary, format text) EXECUTE my_query(841,'LA','SCL', 'GRU', DATE_TRUNC('day','2024-07-31 00:00:00.0'::timestamp), 4069365);
EXPLAIN (analyze, verbose, costs, settings, buffers, wal, timing, summary, format text) EXECUTE my_query(7, 'LA', 'SCL', 'GRU', DATE_TRUNC('day','2024-07-31 00:00:00.0'::timestamp), 4069367);
EXPLAIN (analyze, verbose, costs, settings, buffers, wal, timing, summary, format text) EXECUTE my_query(883,'LA','SCL', 'GRU', DATE_TRUNC('day','2024-07-31 00:00:00.0'::timestamp), 4069367);
EXPLAIN (analyze, verbose, costs, settings, buffers, wal, timing, summary, format text) EXECUTE my_query(901,'LA','SCL', 'GRU', DATE_TRUNC('day','2024-07-31 00:00:00.0'::timestamp), 4069367);

DEALLOCATE my_query;
```

First entry of values get:

Buffers: shared hit=17896
Planning Time: 31.447 ms
Execution Time: 2.366 ms

Subsequential:

Planning Time: 0.053 ms
Execution Time: 0.708 ms

Added 31 combinations of values to the above list and the parameter reduces / eliminates the planing time after first 2 executions:

First values:
Buffers: shared hit=17896
Planning Time: 33.671 ms
Execution Time: 2.828 ms

Second:
Planning Time: 0.064 ms
Execution Time: 1.212 ms

Third:
Execution Time: 1.121 ms

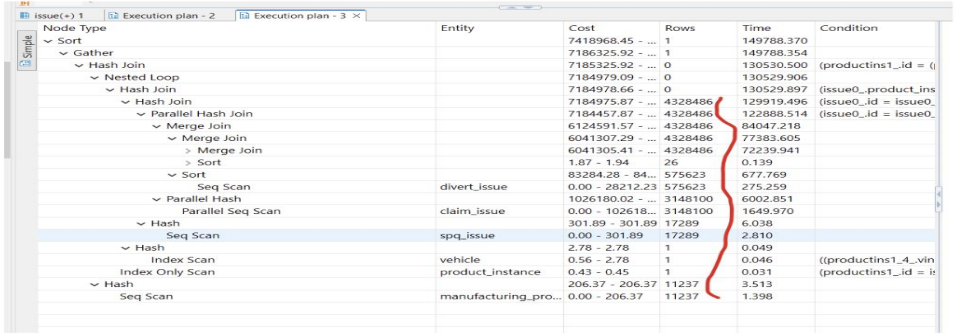
15th:
Execution Time: 0.548 ms

Better Join Methods and Join Orders

Join_collapse_limit and GEQO

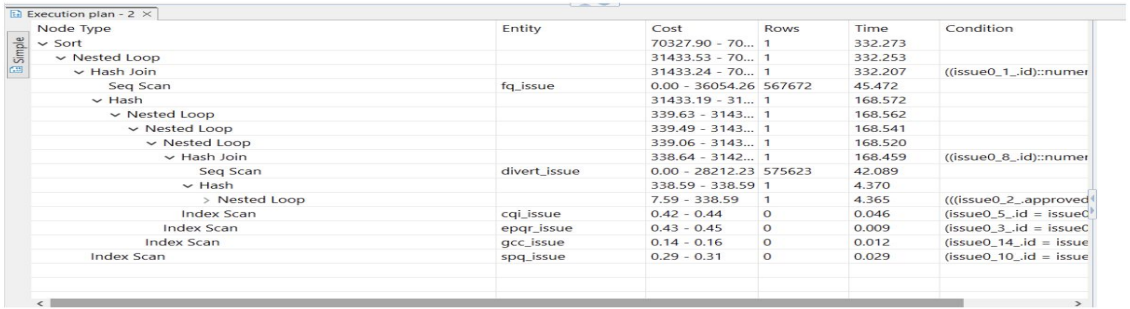
- **Use Case: Queries joining many tables suffering from bad execution plan due to suboptimal join orders.**
- Performance Issues when joining many large tables (and in most cases they are partitioned tables) in PostgreSQL , a quick thing to try out is to change join_collapse_limit from its default 8 to higher value (can set as high as 20 on faster and more CPUs machine).
- This gives the PostgreSQL Optimizer **better Join orders to choose from** and in turn **better join types.**
- The **traditional problem** with **higher values** of **join_collapse_limit** has been **higher query planning time** (to determine best join order for N tables takes an O(N!) factorial approach).
- But with **newer versions** of PostgreSQL the **planning time** to evaluate more join orders have been greatly reduced because join plans are now developed using the **genetic approach** (GEQO).
- Setting join_collapse_limit = 20 reduced query execution times (on many large partitioned tables) by more than 10x (for one of them it was reduced to < 1 second from 8 minutes!).
- The **genetic query optimizer** (GEQO) is an algorithm that does **query planning using heuristic searching**. This **reduces planning time for complex queries** (those joining many relations), at the cost of producing plans that are sometimes inferior to those found by the normal exhaustive-search algorithm
- **Geqo_threshold**- Sets the threshold of FROM items beyond which GEQO is used. The default is 12.
- For **simpler queries** it is usually best to use the **regular, exhaustive-search planner**, but for **queries with many tables the exhaustive search takes too long.**
- **Set geqo_threshold to 16 or 18 with better and faster Machine types now**.

join_collapse_limit (result in 8 min)



Node Type	Entity	Cost	Rows	Time	Condition
Sort		7418968.45 - ...	1	149788.370	
Gather		7186325.92 - ...	1	149788.354	
Hash Join		7185325.92 - ...	0	130530.500	(productins1_.id = (i
Nested Loop		7184979.09 - ...	0	130529.906	
Hash Join		7184978.66 - ...	0	130529.897	(issue0_.product_ins
Hash Join		7184975.87 - ...	4328486	129919.496	(issue0_.id = issue0_
Parallel Hash Join		7184457.87 - ...	4328486	122888.514	(issue0_.id = issue0_
Merge Join		6124591.57 - ...	4328486	84047.218	
Merge Join		6041307.29 - ...	4328486	77383.605	
Merge Join		6041305.41 - ...	4328486	72239.941	
Sort		1.87 - 1.94	26	0.139	
Sort		83284.28 - 84...	575623	677.769	
Seq Scan	divert_issue	0.00 - 28212.23	575623	275.259	
Parallel Hash		1026180.02 - ...	3148100	6002.851	
Parallel Seq Scan	claim_issue	0.00 - 102618...	3148100	1649.970	
Hash		301.89 - 301.89	17289	6.038	
Seq Scan	spq_issue	0.00 - 301.89	17289	2.810	
Hash		2.78 - 2.78	1	0.049	
Index Scan	vehicle	0.56 - 2.78	1	0.046	((productins1_4_.vin
Index Only Scan	product_instance	0.43 - 0.45	1	0.031	(productins1_.id = i
Hash		206.37 - 206.37	11237	3.513	
Seq Scan	manufacturing_pro...	0.00 - 206.37	11237	1.398	

After setting join_collapse limit =40 (result in 1 sec)



Node Type	Entity	Cost	Rows	Time	Condition
Sort		70327.90 - 70...	1	332.273	
Nested Loop		31433.53 - 70...	1	332.253	
Hash Join		31433.24 - 70...	1	332.207	((issue0_1_.id)::numer
Seq Scan	fq_issue	0.00 - 36054.26	567672	45.472	
Hash		31433.19 - 31...	1	168.572	
Nested Loop		339.63 - 3143...	1	168.562	
Nested Loop		339.49 - 3143...	1	168.541	
Nested Loop		339.06 - 3143...	1	168.520	
Hash Join		338.64 - 3142...	1	168.459	((issue0_8_.id)::numer
Seq Scan	divert_issue	0.00 - 28212.23	575623	42.089	
Hash		338.59 - 338.59	1	4.370	
Nested Loop		7.59 - 338.59	1	4.365	(((issue0_2_.approved
Index Scan	cqi_issue	0.42 - 0.44	0	0.046	(issue0_5_.id = issue0
Index Scan	epqr_issue	0.43 - 0.45	0	0.009	(issue0_3_.id = issue0
Index Scan	gcc_issue	0.14 - 0.16	0	0.012	(issue0_14_.id = issue
Index Scan	spq_issue	0.29 - 0.31	0	0.029	(issue0_10_.id = issue

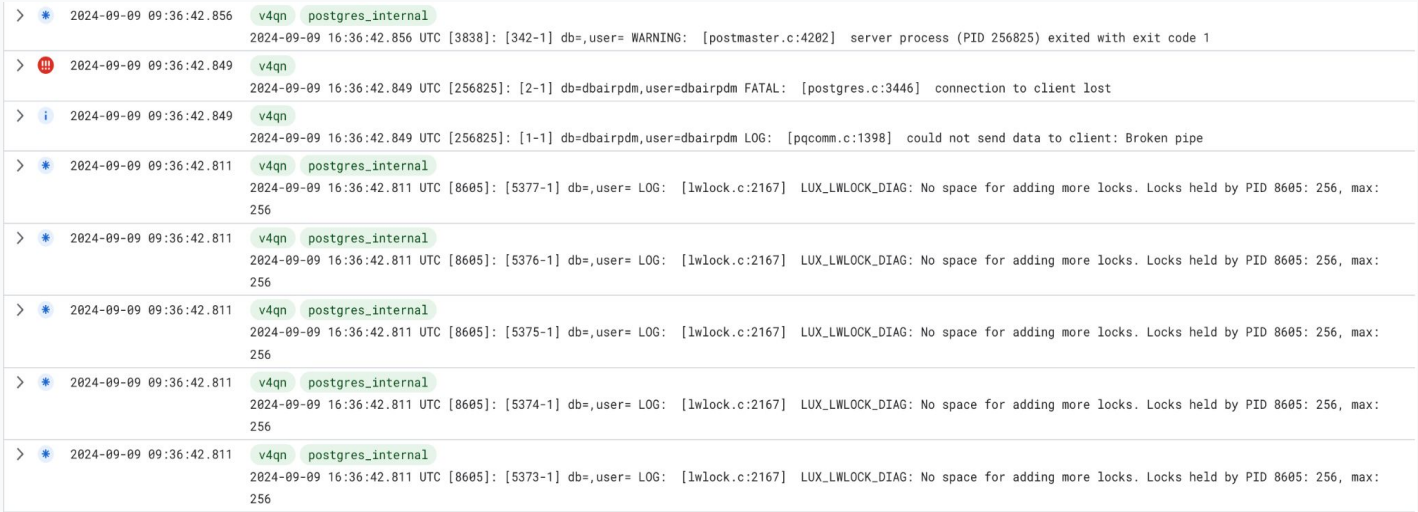
```
test=> show geqo;
 geqo
------
 on
(1 row)

test=> show geqo_threshold;
 geqo_threshold
----------------
 12
(1 row)
```

Optimized-join_collapse_limit

Google Cloud

Optimize Queries on Partitioned Tables

max_locks_per_transaction

- **Use Case:- Queries on large partitioned tables slow due to lock contention**
- If transaction touches many tables in single transaction, or touches large partitioned tables then set the value higher than the default value of 64
- Set it to the number of partitions of the largest partitioned table multiplied by the number of such partitioned tables a single transaction is going to touch
- 

Best Practices in Partitioning

- Choose the **right partition size**.
- Keep your **partition size consistent**.
- **Choose the right partitioning key**- Opt for a key that aligns with your query patterns. For instance, if most of your queries filter by date, a timestamp or date column would be an ideal partitioning key.
- Create partitions in advance.
- Take advantage of **data retention policies to maintain old partitions**. For example, if you're partitioning by time and data has a limited useful life, schedule regular tasks to drop or archive old partitions.
- **Inefficient indexing**-Avoid creating unnecessary indexes on your partitions. Only index the columns that are frequently filtered or joined on.
- **Unoptimized query pattern-** Queries spanning multiple partitions or not using the partition key in the WHERE clause might suffer in performance. Ensure that the majority of your queries are optimized for the partitioning scheme.
- **Partition-wise Join**- Combining partitions that have the same range and values eliminates unnecessary JOIN processing, thus improving performance. Each partition pair is joined by a "Nested Loop" and the results are summarised at the end. Without Partition wise join, scanning results of partitions in the table are obtained separately and the join process is performed at the end
- **Partition-wise Aggregate**- Processing time can be shortened by performing aggregation for each partition.

```
demo=> show enable_partitionwise_join;
 enable_partitionwise_join
---------------------------
 off
(1 row)

demo=> show enable_partitionwise_aggregate;
 enable_partitionwise_aggregate
--------------------------------
 off
(1 row)
```

```
som=> show enable_partitionwise_join;
 enable_partitionwise_join
---------------------------
 off
(1 row)

som=> explain(analyze,buffers) select * from prt1 t1, prt2 t2 where t1.a = t2.b and t1.b = 0 and t2.b between 0 and 10000;
                                                              QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------
 Hash Join  (cost=29948.08..164867.51 rows=1263271 width=24) (actual time=158.228..158.318 rows=0 loops=1)
   Hash Cond: (t2.b = t1.a)
   Buffers: shared hit=14171
   ->  Append  (cost=0.00..106558.08 rows=4194304 width=12) (actual time=0.007..0.009 rows=1 loops=1)
         Buffers: shared hit=1
         ->  Seq Scan on prt2_p1 t2_1  (cost=0.00..64189.92 rows=3145728 width=12) (actual time=0.007..0.007 rows=1 loops=1)
               Filter: ((b >= 0) AND (b <= 10000))
               Buffers: shared hit=1
         ->  Seq Scan on prt2_p2 t2_2  (cost=0.00..21396.64 rows=1048576 width=12) (never executed)
               Filter: ((b >= 0) AND (b <= 10000))
   ->  Hash  (cost=29948.04..29948.04 rows=3 width=12) (actual time=158.216..158.304 rows=0 loops=1)
         Buckets: 1024  Batches: 1  Memory Usage: 8kB
         Buffers: shared hit=14170
         ->  Gather  (cost=1000.00..29948.04 rows=3 width=12) (actual time=158.216..158.303 rows=0 loops=1)
               Workers Planned: 2
               Workers Launched: 2
               Buffers: shared hit=14170
               ->  Parallel Append  (cost=0.00..28947.74 rows=3 width=12) (actual time=148.737..148.738 rows=0 loops=3)
                     Buffers: shared hit=14170
                     ->  Parallel Seq Scan on prt1_p1 t1_1  (cost=0.00..11129.33 rows=1 width=12) (actual time=65.807..65.807 rows=0 loops=3)
                           Filter: (b = 0)
                           Rows Removed by Filter: 349525
                           Buffers: shared hit=5668
                     ->  Parallel Seq Scan on prt1_p2 t1_2  (cost=0.00..11129.33 rows=1 width=12) (actual time=87.599..87.599 rows=0 loops=2)
                           Filter: (b = 0)
                           Rows Removed by Filter: 524288
                           Buffers: shared hit=5668
                     ->  Parallel Seq Scan on prt1_p3 t1_3  (cost=0.00..6689.06 rows=1 width=12) (actual time=73.582..73.582 rows=0 loops=1)
                           Filter: (b = 0)
                           Rows Removed by Filter: 524288
                           Buffers: shared hit=2834
 Planning:
   Buffers: shared hit=72
 Planning Time: 0.258 ms
 Execution Time: 158.404 ms
(35 rows)
```

No Partition wise Join

```
som=> set enable_partitionwise_join=on;
SET
som=> show enable_partitionwise_join;
 enable_partitionwise_join
---------------------------
 on
(1 row)

som=>  explain(analyze,buffers) select * from prt1 t1, prt2 t2 where t1.a = t2.b and t1.b = 0 and t2.b between 0 and 10000;
                                                              QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------
 Append  (cost=18775.21..154652.64 rows=1052468 width=24) (actual time=94.372..94.377 rows=0 loops=1)
   Buffers: shared hit=11338
   ->  Hash Join  (cost=18775.21..100036.42 rows=527481 width=24) (actual time=46.756..46.759 rows=0 loops=1)
         Hash Cond: (t2_1.b = t1_1.a)
         Buffers: shared hit=5669
         ->  Seq Scan on prt2_p1 t2_1  (cost=0.00..64189.92 rows=3145728 width=12) (actual time=0.008..0.008 rows=1 loops=1)
               Filter: ((b >= 0) AND (b <= 10000))
               Buffers: shared hit=1
         ->  Hash  (cost=18775.20..18775.20 rows=1 width=12) (actual time=46.735..46.737 rows=0 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 8kB
               Buffers: shared hit=5668
               ->  Seq Scan on prt1_p1 t1_1  (cost=0.00..18775.20 rows=1 width=12) (actual time=46.734..46.735 rows=0 loops=1)
                     Filter: (b = 0)
                     Rows Removed by Filter: 1048576
                     Buffers: shared hit=5668
   ->  Hash Join  (cost=18775.21..49353.88 rows=524987 width=24) (actual time=47.613..47.615 rows=0 loops=1)
         Hash Cond: (t2_2.b = t1_2.a)
         Buffers: shared hit=5669
         ->  Seq Scan on prt2_p2 t2_2  (cost=0.00..21396.64 rows=1048576 width=12) (actual time=0.011..0.011 rows=1 loops=1)
               Filter: ((b >= 0) AND (b <= 10000))
               Buffers: shared hit=1
         ->  Hash  (cost=18775.20..18775.20 rows=1 width=12) (actual time=47.595..47.595 rows=0 loops=1)
               Buckets: 1024  Batches: 1  Memory Usage: 8kB
               Buffers: shared hit=5668
               ->  Seq Scan on prt1_p2 t1_2  (cost=0.00..18775.20 rows=1 width=12) (actual time=47.594..47.594 rows=0 loops=1)
                     Filter: (b = 0)
                     Rows Removed by Filter: 1048576
                     Buffers: shared hit=5668
 Planning Time: 0.159 ms
 Execution Time: 94.409 ms
(30 rows)
```

Partition wise Join

Google Cloud

Optimize Parallel Queries

Parameters

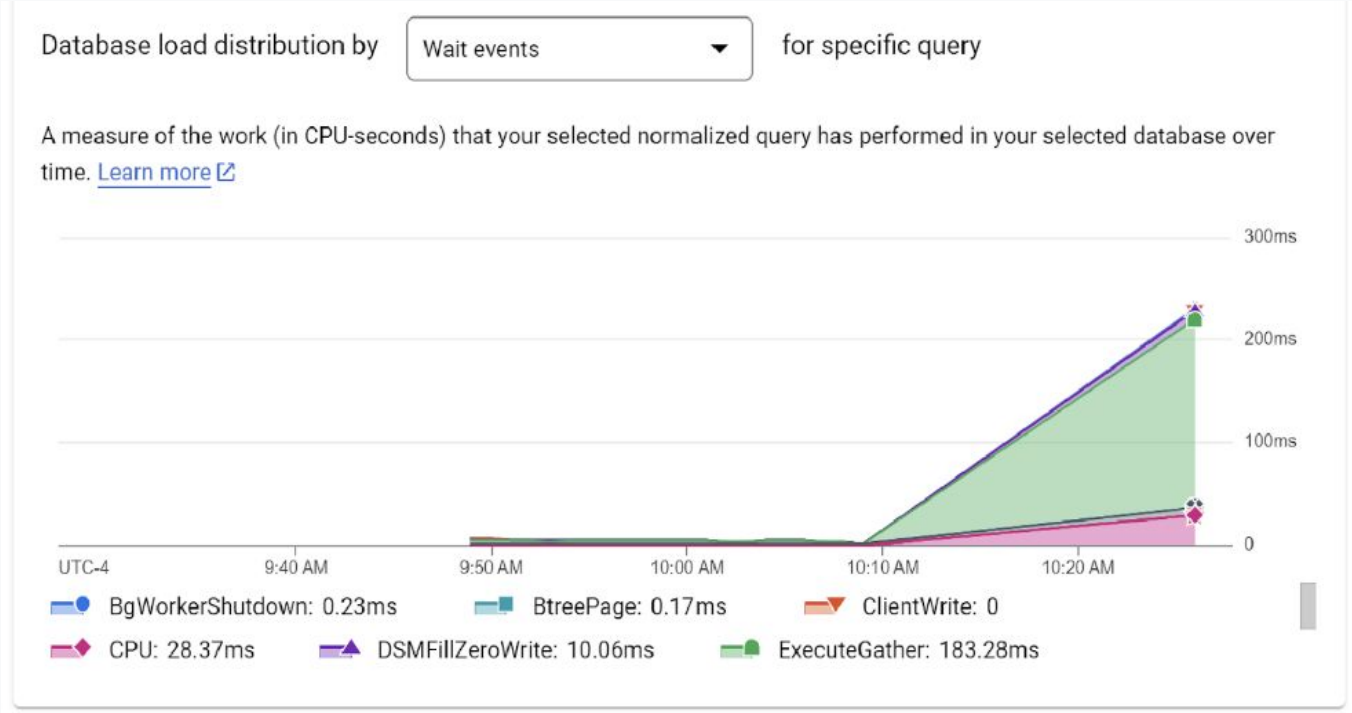
Use Case:- Bad performance on parallel queries

Serial #	Parameter	Default	Remarks
1	Max_worker_processes	8	Irrespective of Instance type
2	Max_parallel_workers	8	Irrespective of Instance type
3	Max_parallel_workers_per_gather	2	Irrespective of Instance type
4	Max_parallel_maintenance_workers	2	Irrespective of Instance type

- Too much Parallelism- Good or Bad?

```
Gather  (cost=60140.09..209462.74 rows=1 width=12) (actual time=21.388..22.079 rows=0 loops=1)
  Output: spl_tramos_pax.spltrm_seq_cdg, spl_pax.splpax_seq_cdg
  Workers Planned: 7
  Workers Launched: 7
```

Parameters



- Parallel query on too many tables and tables with hundreds of partition may not perform well if Partition-wise join, partition-wise aggregates do not happen. The final Gather operation will be detrimental to performance.

Full Index Scans? Vacuum?

Visibility Map and Vacuuming

Use Case:- Performance Issues with Index Only Scan

Index Only Scan does not always eliminates Heap fetches, why?

Index Only Scan can be slower than Index Scan at times, why?

Example:-

Index Only Scan using som_pkey on som (actual time=0.019..0.147 rows=999 loops=1)

Index Cond: (id < 1000)

Heap Fetches: 186

Planning Time: 0.189 ms

Execution Time: 0.283 ms

Why Heap Fetches?

This is because the visibility map(VM) is outdated, PostgreSQL can't rely on it to determine tuple visibility and this forces the database to check the heap directly, even during an Index Only Scan.

In the above example, only 1 row was updated which made page 0 and page 5408 'all_visible' flag to be set to 'f'. (Page 5408 contains the updated version of the tuple and page 0 contains the dead/old version)

```
select count(*) from pg_visibility_map('som') where all_visible='f';
count
-------
     2
```

Visibility Map and Vacuuming

```
SELECT * FROM pg_visibility_map('som');
blkno | all_visible | all_frozen
-------+-------------+------------
 0 | f    | f
 1 | t    | f
 2 | t    | f

SELECT * FROM pg_visibility_map('som') where blkno=5408;
 blkno | all_visible | all_frozen
-------+-------------+------------
 5408 | f     | f
oratest=> select count(*) from pg_visibility_map('som') where all_visible='t';
 count
-------
 5407
```

Why Heap Fetches: 186- Page 0 has 185 rows and Page 5408 has 1 row

```
select ctid, * from som;
(0,184)  |  184 |  0.4253149976251245
 (0,185)  |  185 |  0.7433661580552404
 (1,1)  |  186 |  0.40145769235950435
...
```

This can become a problem if your production tables have lot of changes and VACUUM or AUTOVACUUM does not run regularly. This will make the VM outdated and **INDEX ONLY SCANs have to do heap fetches for all the pages that have all_visible=f and scan the rows in those pages to find out whether it is visible to the transaction.**

So **AutoVacuum/Vacuum does another important job (apart from cleaning dead tuples, preventing transaction ID Wraparound) of updating the Visibility Map to make Index Only scan faster.**

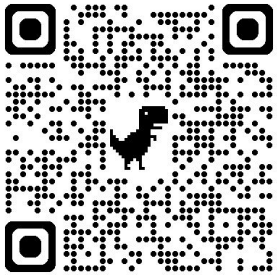
Basic Housekeeping

Let’s Do the Basics Right

1. Vacuum Regularly.
2. Autovacuum- Adjust autovacuum_vacuum_scale_factor, autovacuum_vacuum_cost_limit, autovacuum_vacuum_cost_delay for large tables that changes frequently
3. Analyze regularly.
4. Set deafult_statistics_target higher and analyze- can be set for the table or individual columns. Important for skewed columns.
5. Column Statistics, Extended Statistics/Multivariate statistics.
6. Indexes- Index what you need. Not too many.
7. Correct Index types.
8. Log_min_duration_statement, log_statement,log_line_prefix
9. Connection Pooler
10. Performance Monitoring in Place.

EMEA Database Community

🔗 Join Here



Webinars 🎬 Fireside Chats 🔥 Special Events 🎉 Tech Workshops 🛠️ Roadmap Reveals 🗺️
Monthly Newsletter 📰

Google Cloud