

PostgreSQL Internals and Performance Optimization



Speaker Bio

- Name: Shiv Iyer
- Occupation: Founder and Principal of MinervaDB
- Technology focus:
 - Open Source Database Systems
 - Transaction processing systems
 - Database Systems optimizers and internals
 - Performance optimization and tuning
 - Capacity Planning and Sizing
 - MySQL – InnoDB and RocksDB (MyRocks)
 - MariaDB
 - PostgreSQL



The scope of the talk

- PostgreSQL Architecture and Internals
- PostgreSQL Query Performance Troubleshooting
- Optimal Indexing
 - Missing Indexes
 - Unused Indexes
- Configuring PostgreSQL for Performance
- PostgreSQL Partitioning



Safe Harbor Statement

The following is intended to outline our general direction in PostgreSQL infrastructure operations. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for MinervaDB's products, consulting, support and remote DBA services remains at the sole discretion of MinervaDB Inc.



PostgreSQL Architecture components

PostgreSQL is a multi-process Relational Database Management System:

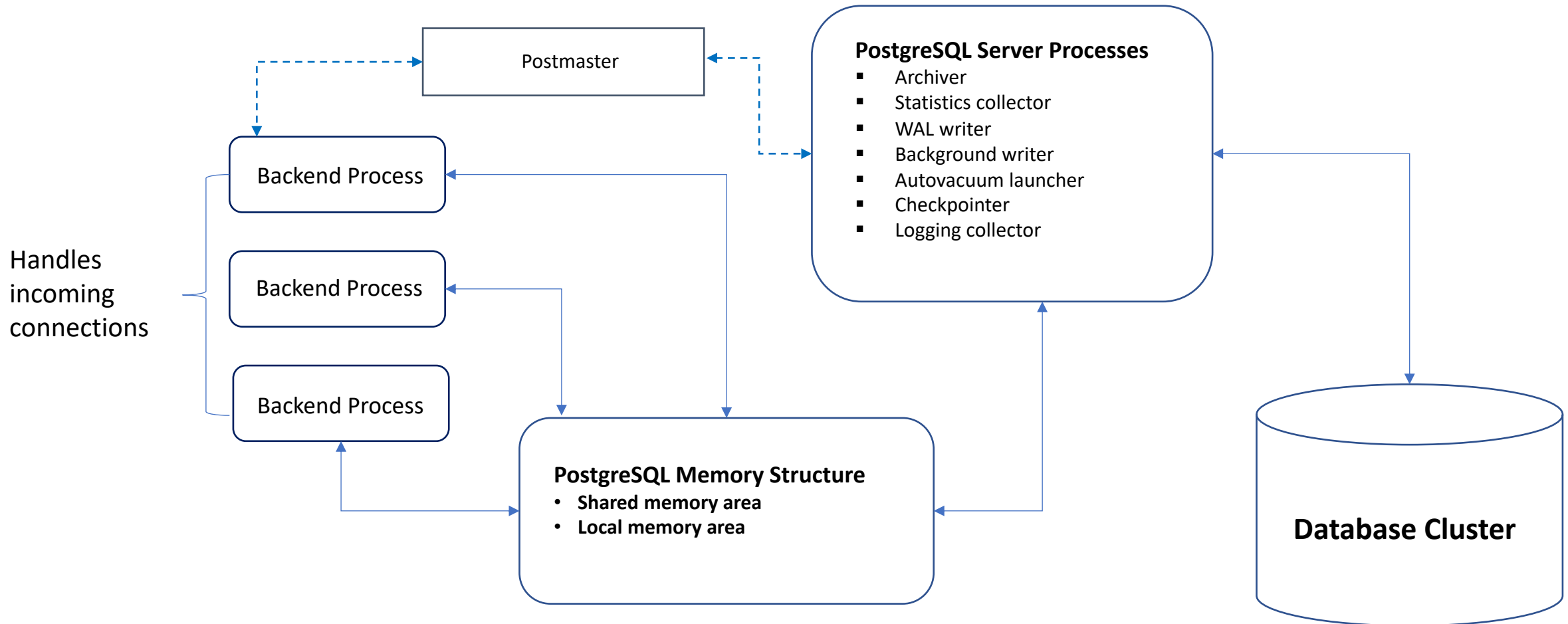
- Server processes
 - Archiver
 - Statistics collector
 - WAL writer
 - Background writer
 - Autovacuum launcher
 - Checkpointer
 - Logging collector
- Client processes
 - Postgres backend processes
- Memory structure
 - Shared memory area
 - Local memory area
- Database Cluster



PostgreSQL Architecture Explained



PostgreSQL Architecture and Internals

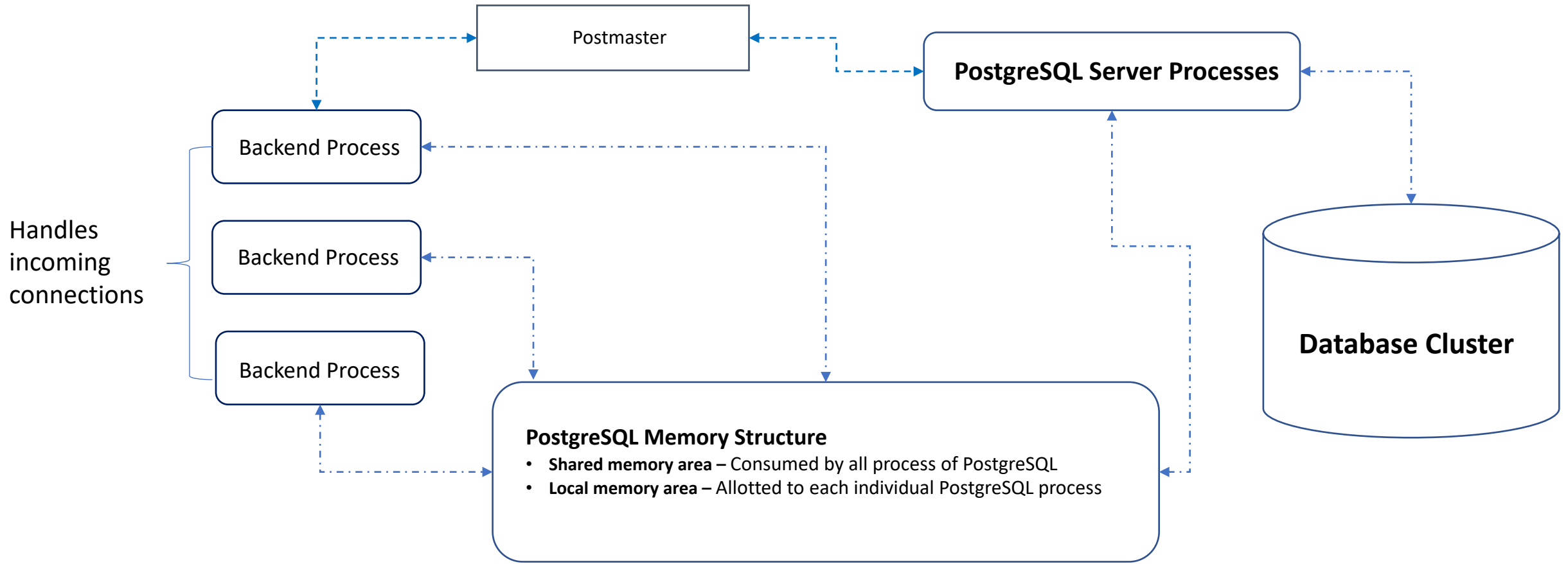


PostgreSQL Processes

Background process	Description
Archiver	Process accountable for archive logging
Statistics collector	Statistics data (pg_stat_activity, pg_stat_user_indexes etc.) required for troubleshooting PostgreSQL more intuitively is collected by the process Statistics Collector
WAL writer	Process to flush WAL data from WAL buffer to a persistent storage / disk
Background writer	Process to flush dirty pages in shared buffer pool to a persistent storage / disk (In PostgreSQL 9.1 or earlier, Background writer was also accountable for checkpoint process)
Autovacuum launcher	Process involved in periodical auto vacuum activity
Checkpointer	Checkpoint process (included from PostgreSQL 9.2)
Logging collector	Process to record error incidents to log files



PostgreSQL Memory Structure explained



Shared Memory Area

Shared Memory Area	Description
Shared buffer pool	PostgreSQL caches pages from tables and indexes to shared buffer pool from persistent storage / disk for optimal performance.
WAL buffer	To ensure zero data loss, PostgreSQL retains transaction logs in WAL file. The WAL data is stored in WAL buffer before writing to a persistent storage.
Commit log	To ensure data concurrency and reliability, the commit log maintains states of entire transactions – <i>in-progress, committed and aborted</i> .



Local Memory Area

Local Memory Area	Description
Temporary buffer	The local buffer area used to access temporary table data.
Work_mem	The local buffer area used by sort memory operations and hash tables before using temporary data files.
Maintenance_work_mem	The local buffer area used for maintenance operations like adding foreign keys to existing tables, creating indexes, vacuum etc.



Building PostgreSQL Infrastructure Operations for Performance



Why performance tuning is a journey ?

- Measure PostgreSQL query performance to understand Data Access Path:
 - Data logic changes with new business rules, This internally sometimes conflicts with PostgreSQL statistics and execution plans.
 - PostgreSQL tools used to troubleshoot performance
- Indexes causes pain with the data volume growth, We may have to create or remove indexes on a regular interval for retaining an optimal PostgreSQL performance.
- PostgreSQL infrastructure footprint (database volume and transactions) grows continuously with your business velocity and acceleration, So every fast growing company should plan building database infrastructure capacity for the future to avoid unpleasant performance bottlenecks at a scale.
- Larger tables are operationally expensive and exhaustive, Table partitioning efficiently compartmentalize data across multiple partitions to distribute I/O and also address database archiving.



Troubleshooting PostgreSQL query performance using pg_stat_statements



pg_stat_statements

- Officially bundled with PostgreSQL for troubleshooting the performance.
- pg_stat_statements is also known as PostgreSQL contrib extension, Which can found in **contrib** directory of PostgreSQL distribution.
- pg_stat_statements does not record individual queries, It parameterizes them and save as aggregated reports.
- By using pg_stat_statements, You can measure PostgreSQL performance by “Response Time”.
- We strongly recommend you to increase OS shared memory limits while using pg_stat_statements for optimal performance.



How to use pg_stat_statements?



Install and configure pg_stat_statements

On Debian / Ubuntu:

Replace 9.X with your installed Postgres version:

```
sudo apt-get install postgresql-contrib-9.X
```

On CentOS / RedHat:

```
sudo yum install postgresql-contrib
```

Configure PostgreSQL adding following entries to postgres.conf:

```
shared_preload_libraries = 'pg_stat_statements'
```

You can increase the max size of the query strings Postgres records if you have large complicated queries executed in PostgreSQL infra.

```
track_activity_query_size = 2048
```

Record queries generated by stored procedures also

```
pg_stat_statements.track = all
```



Example

```
SELECT campaign_master.campaign_category,  
campaign_master.campaign_id,  
campaign_master.publisher_id  
FROM campaign_master,  
publisher_master  
WHERE campaign_master.publisher_id = publisher_master.publisher_id  
AND  
publisher_master.publisher_category = 'Social Gaming';
```



pg_stat_statements parameterize the queries

```
SELECT campaign_master.campaign_category,  
campaign_master.campaign_id,  
campaign_master.publisher_id  
FROM campaign_master,  
publisher_master  
WHERE campaign_master.publisher_id = publisher_master.publisher_id  
AND  
publisher_master.publisher_category = '?';
```



Monitoring PostgreSQL query performance with pg_stat_statements

```
SELECT  
(total_time / 100 / 60) as Total,  
(total_time/calls) as Avg,  
query  
FROM pg_stat_statements  
ORDER BY 1  
DESC LIMIT 100;
```

In the SQL above we have captured queries on “**Response Time**” , The total latency of individual queries is reported in minutes and average latency on milliseconds.



Output from pg_stat_statements

Total | Avg | query

-----+-----+-----

936.72 | 91.86 | SELECT campaign_category FROM campaign_master...

513.18 | 16.38 | SELECT campaign_id FROM campaign_master...

(2 rows)



PostgreSQL Auto Explain

- Automatically logs the execution plans of slow queries in PostgreSQL without you manually running EXPLAIN.
- A module in the contrib package.
- Can be enabled in postgresql.conf by setting parameter *auto_explain.log_min_duration* and the library for all sessions by setting *shared_preload_libraries = auto_explain*
- We strongly recommend you not to enable *auto_explain.log_analyze* in production unless you are really confident about handling extra performance bottleneck due to this setting.



2020-07-09 17:26:54.645 IST [15129] postgres@postgres LOG: duration: 0.292 ms plan:

```
SELECT campaign_master.campaign_category,  
campaign_master.campaign_id,  
campaign_master.publisher_id  
FROM campaign_master,  
publisher_master  
WHERE campaign_master.publisher_id =publisher_master.publisher_id  
AND  
publisher_master.publisher_category = 'Social Gaming';
```

Aggregate (cost=26.34..26.35 rows=168 width=8) (actual time=4.278..0.278 rows=1 loops=1)

-> Hash Join (cost=19.70..25.93 rows=11364 width=0) (actual time=5.215..0.267 rows=128 loops=1)

Hash Cond: (publisher_master.publisher_id = campaign_master.publisher_id)

-> Seq Scan on pg_index (cost=0.00..5.80 rows=3164 width=4) (actual time=0.014..0.042 rows=1128 loops=1)

Filter: indisunique

Rows Removed by Filter: 18

-> Hash (cost=15.42..15.42 rows=342 width=4) (actual time=6.166..0.166 rows=4367 loops=1)

Buckets: 1024 Batches: 1 Memory Usage: 7MB

-> Seq Scan on pg_class (cost=0.00..15.42 rows=3142 width=4) (actual time=3.007..0.106 rows=4367 loops=1)



Monitoring index usage

- Optimal indexing is very important for PostgreSQL performance
- You can efficiently troubleshoot PostgreSQL performance with following data:
 - Missing indexes
 - Unused indexes
- Enable PostgreSQL statistics collector to monitor missing and unused indexes by setting following parameters:
 - track_activities
 - track_counts
 - track_functions
 - track_io_timing



Missing indexes

```
SELECT
relname,
seq_scan - idx_scan AS too_much_seq,
CASE
WHEN
seq_scan - coalesce(idx_scan, 0) > 0
THEN
'Missing Index?'
ELSE
'OK'
END,
pg_relation_size(relname::regclass) AS rel_size, seq_scan, idx_scan
FROM
pg_stat_all_tables
WHERE schemaname = 'public'
AND pg_relation_size(relname::regclass) > 80000
ORDER BY
too_much_seq DESC;
```



Unused indexes

```
SELECT
indexrelid::regclass as index,
relid::regclass as table,
'DROP INDEX ' || indexrelid::regclass || ';' as drop_statement
FROM pg_stat_user_indexes
JOIN pg_index USING (indexrelid)
WHERE idx_scan = 0
AND
indisunique is false;
```



PostgreSQL partitioning

- Efficient distribution of data and transaction I/O.
- Most often / frequently accessed data and indexes can be partitioned for performance by efficiently fitting in the memory.
- Partitioning enables high performance sequential scan of records in the table compared to using an index and random access reads scattered across the whole table.
- PostgreSQL partition types:
 - **RANGE**: Data in Tables partitioned over a range of values.
 - **HASH**: Tables Partitioned by specifying a modulus and a remainder for each partition.
 - **LIST**: Tables partitioned by explicitly listing which key values appear in each partition.



Configuring PostgreSQL for Performance

- `shared_buffers`
 - PostgreSQL support double buffering, i.e. PostgreSQL uses its native internal buffer and kernel buffered IO. Increasing `shared_buffers` in a READ intensive infrastructure will benefit in performance, fitting entire database in RAM is good.
- `wal_buffers`
 - PostgreSQL writes WAL records to `wal_buffers`. You can set higher values for `wal_buffers` if expected more concurrent session with heavy WRITES / UPDATES.
- `effective_cache_size`
 - The `effective_cache_size` is how much memory is available for disk caching by the operating system and within the database itself , This parameter is used by PostgreSQL optimizer / query planner to decide whether the execution plans can fit in the memory, We strongly recommend to set this value up to 80% of the RAM and setting this value too low may influence query planner not to use few indexes at all causing serious performance bottleneck.



Configuring PostgreSQL for Performance

- `work_mem`
 - `work_mem` configuration parameter makes complex PostgreSQL SORT / SEARCH operations optimal. But, This is a “per session” (allocated exclusively for each concurrent SORT operations) configuration parameter and so we strongly recommend to keep this value low to avoid excessive swapping and out-of-memory errors. The default value is 4MB.
- `maintenance_work_mem`
 - `maintenance_work_mem` defines how much memory is allocated for routine Database Infrastructure Operations maintenance activities like adding foreign keys to existing tables, index operations (CREATE / ALTER / DROP) and VACUUM. Since PostgreSQL maintenance operations do not run concurrently you can set this value definitely higher than `work_mem`. The default value is 64MB



Contacts

- Email: shiv@minervadb.com
- Twitter: <https://twitter.com/thewebscaledba>
- Facebook page: <https://www.facebook.com/DataOpsGeek/>
- GitHub: <https://github.com/shiviyer>
- Contact MinervaDB: contact@minervadb.com
- Contact MinervaDB Support: support@minervadb.com

