Defaults to Performance: Tuning PostgreSQL Like a Pro

Introduction

PostgreSQL performance tuning involves adjusting configuration parameters to optimize database performance for your specific workload. This guide covers the most critical parameters with practical examples and real-world scenarios.

Key Principles

1. Measure before tuning - Always baseline your current performance
2. One change at a time - Isolate the impact of each parameter change
3. Monitor continuously- Track performance metrics after changes
4. Test thoroughly - Validate changes in a test environment first

Configuration Files

1. Postgresql.conf - Main configuration file
2. Pg_hba.conf - Authentication configuration
3. Postgresql.auto.conf - Auto-generated configuration (overrides postgresql.conf)

Viewing Current Settings

```
-- View all current settings
SELECT name, setting, unit, context, short_desc
FROM pg_settings
ORDER BY name;

-- View parameters that require restart
SELECT name, setting, pending_restart
FROM pg_settings
WHERE pending_restart = true;
```

What is PostgreSQL Parameter Tuning?

Parameter tuning involves adjusting PostgreSQL's configuration parameters (found in postgresql.conf and other config files) to match your specific workload, hardware, and performance requirements. These parameters control everything from memory allocation to query execution behavior.

Key Areas of Parameter Tuning

Memory Management:

- shared_buffers - Controls how much memory PostgreSQL uses for caching data
- work_mem - Memory allocated for sorting and hash operations
- maintenance_work_mem - Memory for maintenance operations like VACUUM

Connection and Process Management:

- **max_connections**: Sets the maximum number of concurrent client connections.
- **superuser_reserved_connections**: Reserves connections for superusers.
- **max_worker_processes**: Limits background worker processes.
- **max_parallel_workers**: Controls parallel query workers.
- **max_parallel_workers_per_gather**: Limits parallel workers per query.
- **max_locks_per_transaction**: Sets maximum locks per transaction.
- **max_prepared_transactions**: Limits prepared transactions.

Best Practices:

- Set **max_connections** based on workload and hardware; avoid setting it too high, as each connection uses memory.
- Use connection pooling (e.g., PgBouncer) for high-concurrency applications.
- Reserve superuser connections for maintenance and emergencies.
- Tune parallel worker settings for complex queries and large datasets.
- Monitor active connections and processes using **pg_stat_activity**.
- Regularly review and adjust these parameters as workload changes.

Query Planning and Execution:

- **effective_cache_size**: Estimates memory available for caching data; helps planner choose index scans.
- **random_page_cost**: Sets the cost of non-sequential disk reads; lower values favor index scans.
- **seq_page_cost**: Sets the cost of sequential disk reads.
- **work_mem**: Memory for sorting and hashing operations per query.
- **maintenance_work_mem**: Memory for maintenance tasks (e.g., vacuum, index creation).
- **enable_***: Flags to enable/disable specific plan types (e.g., enable_seqscan, enable_indexscan).

Best Practices:

- Keep table statistics up-to-date with regular ANALYZE or autovacuum.
- Tune effective_cache_size and page costs based on your hardware and workload.
- Increase work_mem for complex queries, but avoid setting it too high globally.
- Use indexes appropriately and monitor query plans with EXPLAIN.
- Disable specific plan types only for troubleshooting, not as a permanent solution.

Efficient query planning and execution depend on accurate statistics, appropriate memory settings, and well-designed indexes. Regularly analyze tables, review query plans, and tune parameters to optimize performance.

Write-Ahead Logging (WAL):

- **wal_buffers**: Memory allocated for WAL data before writing to disk.
- **max_wal_size** / **min_wal_size**: Limits the total size of WAL files.
- **checkpoint_timeout**: Time between automatic checkpoints.
- **checkpoint_completion_target**: Fraction of checkpoint interval spent writing.
- **synchronous_commit**: Controls when transactions are considered committed (can be set to on, off, or local).

Best Practices:

- Increase **wal_buffers** for high write workloads.
- Adjust checkpoint settings to balance performance and recovery time.
- Enable WAL archiving for point-in-time recovery.
- Use **synchronous_commit** for critical data; set to **off** for faster but less durable commits if acceptable.

WAL provides crash recovery and supports replication. Tuning WAL parameters improves write performance, backup reliability, and disaster recovery capabilities. Monitor WAL usage and adjust settings to match your workload and data protection requirements.

Vacuum and Autovacuum Parameters

What does VACUUM do ?

Vacuum is a process by which the database cleans up the dead tuples and makes the space occupied by these dead tuples to again reusable space for more writes.These dead tuples
get created because of the update and DELETE operations on a table, the database keeps the old row data for other running queries and transactions due to its MVCC model. Once all
the other queries and transactions do not need these old rows they effectively become dead rows or dead tuples which can be removed by using VACUUM.

How does autovacuum work?

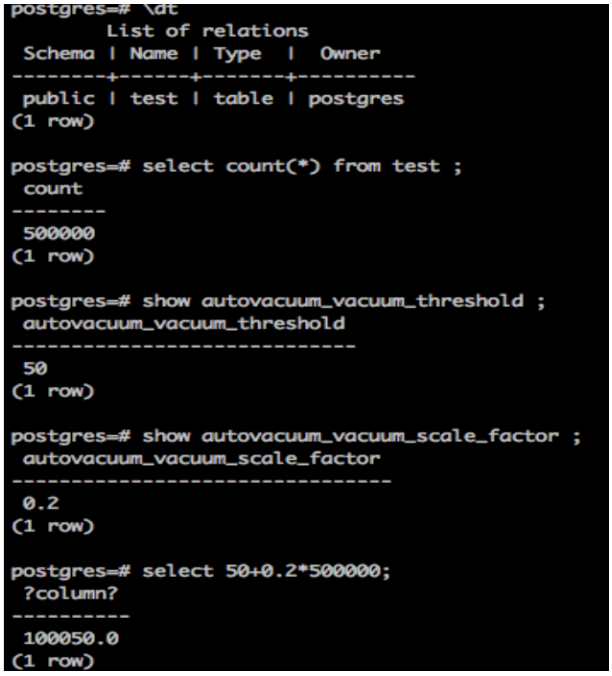
There are two different kinds of autovacuum systems the autovacuum launcher and the auotvacuum worker. The auotvacuum launcher is a continuous
running process, which is started by the postmaster.The launcher schedules autovacuum workers to start when it is needed.The auotvacuum worker process is the actual process
which does the vacuuming, they connect to a database which is determined by the launcher and once connected they read the catalog tables to select a table as a candidate for
vacuuming.There is a autovacuum shared memory area, where the launcher stores information about the tables in a database which needs a vacuum.When the autovacuum launcher
wants a new worker to start it sets a flag in the shared memory and sends a signal to the postmaster.Then the postmaster starts a worker, this new worker process connects to the
shared memory and reads the information in the autovacuum shared memory area stored by the launcher process.

When does the autovacuum run?

When you run UPDATE's and DELETE's, a lot of dead rows will accumulate, The auto vacuum runs or starts once the number of dead tuples has exceeded the threshold.The following
formula decides when to start the auto vacuuming process

vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples

To test the how the formula works I have created a table with 500000 rows, will updating the table to see how the formula works

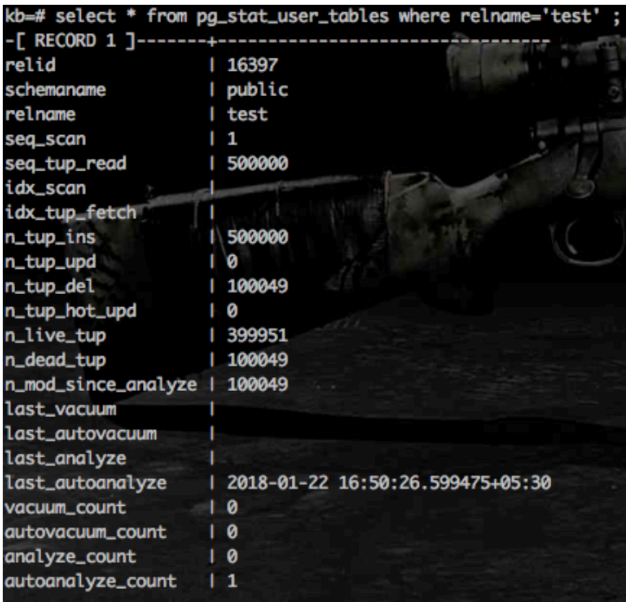


If u see the above screenshot, we have inserted 500000 rows in a table and using the above atovacuum formula we can see at every 100050 dead tuples the autovacuum process will
start vacuuming the table.

So to test if the autovacuum really works based on the above formula we checked by deleting some data from the table using the below steps:

```
delete from test where id < 100050 ;
```

Now if we see the below screenshot we can observe that after the delete statement there are 100049 dead tuples and autovacuum has not kicked in because it
autovacuum threshold's



```
kb=# select * from pg_stat_user_tables where relname='test' ;
-[ RECORD 1 ]-------+------------------------------
relid               | 16397
schemaname          | public
relname             | test
seq_scan            | 1
seq_tup_read        | 500000
idx_scan            |
idx_tup_fetch       |
n_tup_ins           | 500000
n_tup_upd           | 0
n_tup_del           | 100049
n_tup_hot_upd       | 0
n_live_tup          | 399951
n_dead_tup          | 100049
n_mod_since_analyze | 100049
last_vacuum         |
last_autovacuum     |
last_analyze        |
last_autoanalyze    | 2018-01-22 16:50:26.599475+05:30
vacuum_count        | 0
autovacuum_count    | 0
analyze_count       | 0
autoanalyze_count   | 1
```

What are the indicators to give you a clue that the autovacuum/analyze setting is not adequate enough ..?

If there are tables which are not being heavily used or there only reads and inserts the default parameters should be good which are

autovacuum_vacuum_threshold=50
autovacuum_sacle_factor=0.2

Changing default settings needs a lot of research. Assume you see a slow performance of the database due to few queries. However, this performance issue is not consistent. Same queries
run without any issues sometimes. In this scenario, you would need to find out the tables involved in the queries and observe DMLs on the tables. If you see that DMLs are very high
which create dead tuples at the same time, however, these do not fall under autovacuum threshold due to a large size of tables. You can consider this situation to alter autovacuum
settings to modify so that autovacuum does occur on these tables aggressively. If you're doing it, you need continuous monitoring to check if performance is improved/degraded.
Enable proper logging to generate pgBadger reports to see slow queries and autovacuum occurrences.

We can change the autovacuum threshold parameters by using the below command

```
# alter table <table_name> set (AUTOVACUUM_VACUUM_THRESHOLD=int) ;
# alter table <table_name> set (AUTOVACUUM_VACUUM_SCALE_FACTOR=int) ;
```

Autovacuum parameters in PostgreSQL control how and when the autovacuum process runs to clean up dead tuples and maintain database health.

Key parameters include:

- **autovacuum_vacuum_threshold**: Minimum number of tuple updates/deletes before autovacuum triggers.
- **autovacuum_analyze_threshold**: Minimum number of tuple inserts/updates/deletes before autovacuum analyze triggers.
- **autovacuum_vacuum_scale_factor**: Fraction of table size to add to threshold for vacuum.
- **autovacuum_analyze_scale_factor**: Fraction of table size to add to threshold for analyze.
- **autovacuum_naptime**: Time between autovacuum runs (default: 1 min).
- **autovacuum_max_workers**: Max number of autovacuum processes.
- **autovacuum_vacuum_cost_limit** & autovacuum_vacuum_cost_delay: Control resource usage.

When to tune:

- High write/update/delete workloads: Lower thresholds/scale factors for more frequent vacuuming.
- Large tables: Increase workers or decrease naptime for faster cleanup.
- Performance issues (bloat, slow queries): Tune cost limits/delays for more aggressive vacuuming.
- Resource contention: Increase delays/limits to reduce autovacuum impact.

Tune at database level:

  - When most tables have similar workload patterns.
  - When you want a baseline for all tables.
  - For small databases or uniform schemas.

Tune at table level:

  - For tables with high write/delete activity (e.g., logs, queues).
  - For very large tables prone to bloat.
  - When specific tables have performance issues or require more frequent vacuuming.

Best practice:

Start with database-level tuning. Monitor autovacuum logs and table bloat. Apply table-level overrides only where needed for problem tables.

Logging Parameters

Logging parameters in PostgreSQL control what information is written to the server logs, such as errors, queries, connections, and system events. Examples include logging_collector, log_directory, log_statement, and log_min_messages.

Why tune them?

- To capture enough detail for troubleshooting and auditing.
- To avoid excessive logging that can fill disk space or slow performance.
- To focus logs on relevant events (e.g., slow queries, errors).
- To comply with security or compliance requirements.

logging_collector: Enables log collection (on/off).
log_directory: Directory for log files.
log_filename: Log file name pattern.
log_rotation_age: Time before log file rotation.
log_rotation_size: Size before log file rotation.
log_statement: Logs SQL statements (none, ddl, mod, all).
log_min_messages: Minimum message level to log (debug5–fatal).
log_min_error_statement: Minimum error level for statement logging.
log_line_prefix: Customizes log line format.
log_connections: Logs new connections (on/off).
log_disconnections: Logs session ends (on/off).
log_duration: Logs statement duration (on/off).
log_temp_files: Logs temp files over size threshold.
log_checkpoints: Logs checkpoint activity.
log_lock_waits: Logs lock waits exceeding threshold.

Best practice:

Adjust logging parameters based on your monitoring needs and system resources. Start with moderate logging and increase detail only when needed for debugging or analysis. Below are the parameters that can be set if you want to generate log information into log.

log_statement
log_line_prefix
log_checkpoints
log_connections
log_disconnections
log_lock_waits
log_temp_files
log_autovacuum_min_duration
log_min_statement_duration

When Should Parameter Tuning Be Done?

Initial Setup:

- After fresh PostgreSQL installation
- When migrating to new hardware
- Before going into production

Performance Issues:

- Slow query performance
- High CPU or memory usage
- I/O bottlenecks
- Connection pool exhaustion

Workload Changes:

- Significant increase in data volume
- Change in application usage patterns
- New features requiring different query types

Regular Maintenance:

- Quarterly performance reviews
- After major application updates

- When monitoring shows degraded performance

Hands-on Use Cases

We have covered some usecases in the following git

https://github.com/RajeshMadiwale/pgday-2025_paratune

Follow the instructions to create the PostgreSQL-17 docker and be ready for the hands on in session.

Monitoring and Validation

Key Metrics to Monitor

```
-- Buffer hit ratio (target >95%)
SELECT round(100.0 * sum(blks_hit) / (sum(blks_hit) +
sum(blks_read)), 2)
FROM pg_stat_database;

-- Query performance
SELECT query, calls, mean_time
FROM pg_stat_statements
ORDER BY total_time DESC LIMIT 5;

-- Connection usage
SELECT state, count(*) FROM pg_stat_activity GROUP BY state;
```

Validation Process

Baseline - Capture metrics before changes
Test - Change one parameter at a time
Monitor - Watch for 24-48 hours
Validate - Compare performance metrics

Quick Health Checks

Buffer hit ratio stays >95%
No increase in query execution times
Connection counts remain stable
No memory pressure warnings in logs

Tools

pg_stat_statements for query tracking
pgBench/sysbench for load testing
System logs for errors/warnings

Change parameters gradually and always test on replicas first.

Best Practices

1. Begin with Defaults:

   Use PostgreSQL's default settings initially. Only tune parameters when you encounter performance issues, specific workload requirements, or after monitoring system behavior.

2. Monitor Before Tuning:

   Employ monitoring tools (pg_stat_activity, pg_stat_statements, logs) to identify bottlenecks, slow queries, table bloat, or resource contention. This helps you make informed tuning decisions.

3. Autovacuum Tuning:

   - Lower autovacuum thresholds and scale factors for high-write tables to prevent bloat.
   - Use table-level overrides for tables with unique workloads.
   - Regularly check autovacuum logs and table statistics.

4. Memory Management:

   - Adjust shared_buffers for overall cache size.
   - Set work_mem for query operations, but avoid setting it too high globally.
   - Use maintenance_work_mem for maintenance tasks like vacuum and index creation.

5. Logging Configuration:

    - Enable sufficient logging for troubleshooting (log_min_messages, log_statement, log_duration).
    - Avoid excessive logging to prevent disk space and performance issues.
    - Use log rotation settings (log_rotation_age, log_rotation_size) to manage log file sizes.

6. Checkpoint and WAL Tuning:
    - Increase checkpoint_timeout and max_wal_size for write-heavy workloads to reduce checkpoint frequency.
    - Monitor checkpoint logs for performance impact.
    - Adjust WAL settings (wal_buffers, synchronous_commit) for durability and speed.

7. Connection Management:
    - Set max_connections based on workload and hardware capacity.
    - Use connection pooling (e.g., PgBouncer) for high-concurrency environments.

8. Table-Level Overrides:
    - Apply specific settings to tables with unique requirements using ALTER TABLE ... SET.

9. Testing and Documentation:
    - Make incremental changes and test their impact in a staging environment.
    - Document all parameter changes for future reference and troubleshooting.

10. Continuous Monitoring:
    - Regularly review system performance, logs, and statistics.
    - Adjust parameters as workload and data volume evolve

Conclusion

PostgreSQL parameter tuning is an iterative process that requires understanding your workload, measuring performance, and making incremental improvements. Start with the most impactful parameters (memory settings), monitor continuously, and always test changes thoroughly before applying to production.

Remember: The best configuration is workload-specific. Use this guide as a starting point, but always validate changes with your actual data and usage patterns.