PostgreSQL Configuration Parameters

When tuning PostgreSQL for performance, shared memory settings are critical — they control how the database handles disk I/O, caching, and transactional safety. Here's a quick breakdown:

□ Shared Buffers
✅ Stores frequently used data blocks in memory to reduce disk I/O.
✅ Default:
– PostgreSQL ≤ 9.2 → 32 MB
– PostgreSQL ≥ 9.3 → 128 MB
✅ On dedicated servers, a good starting point: 25% of total memory.

□ WAL Buffers (Write-Ahead Logging)
✅ Temporarily holds database changes before writing them to WAL files.
✅ Essential for recovery and durability.
✅ Default: minimum 32 KB, or computed automatically with wal_buffers = -1 (based on shared_buffers).

□ Work Memory
✅ Allocated per query for sort operations, hashes, and temporary data.
✅ Default:
– PostgreSQL ≤ 9.3 → 1 MB
– PostgreSQL ≥ 9.4 → 4 MB

□ Maintenance Work Memory
✅ Used for heavy tasks like VACUUM, ANALYZE, ALTER TABLE, CREATE INDEX.
✅ Larger settings improve maintenance performance.
✅ Default:
– PostgreSQL ≤ 9.3 → 16 MB
– PostgreSQL ≥ 9.4 → 64 MB

1. `max_parallel_maintenance_workers`

- **Description**: Specifies the maximum number of parallel workers to be used by maintenance operations (such as `VACUUM`, `CREATE INDEX`, etc.).
- **Default**: 2
- **Impact**: Higher values can speed up large maintenance tasks but use more CPU resources. Lower values reduce resource contention but may result in slower maintenance tasks.

==========

2. `lock_timeout`

- **Description**: Defines the maximum amount of time (in milliseconds) a query will wait to acquire a lock before throwing an error. This helps to avoid long waits on locks.
- If a query tries to access something that is already locked by another query, it will wait for a certain amount of time. If it can't get the lock within that time, it will stop and show an error.
- **Default**: 0 (no timeout, meaning queries will wait indefinitely)
- **Impact**: Setting this to a non-zero value can help prevent queries from hanging indefinitely due to lock contention.

==========

3. `statement_timeout`

- **Description**: Sets the maximum allowed time (in milliseconds) for a query to run before being automatically terminated. This is useful to prevent long-running queries from consuming excessive system resources.
- **Default**: 0 (no timeout)
- **Impact**: Useful for preventing inefficient queries from running forever. However, be cautious when setting it too low, as complex queries might fail unexpectedly.

==========

4. `idle_in_transaction_session_timeout`

- **Description**: Specifies the maximum time (in milliseconds) that a session with an idle transaction is allowed to remain open. Once this time is exceeded, the session will be terminated.
- **Default**: 0 (no timeout)
- **Impact**: Helps avoid sessions that hold transactions open for long periods, which can cause locking issues or bloat in the database.

==========

5. `max_parallel_workers_per_gather`

- **Description**: Controls the maximum number of parallel workers that can be used for a single `Gather` or `Gather Merge` node in a query plan.

- **Default**: 2
- **Impact**: Increasing this value allows more parallel workers to be used for query execution, which can speed up large queries, especially for parallelizable operations like large `JOIN`s and `SCAN`s.

==========

6. `Default_statistics_target`

- **Description**: Determines the default statistics collection level for table columns (used by the query planner to optimize queries).

It tells PostgreSQL how much information to gather about the data in each table to help it make smarter decisions when running queries.

- **Higher value**: PostgreSQL collects more detailed data, which can make queries run faster, but it may take longer to gather the statistics.
- **Lower value**: PostgreSQL collects less data, which might make it run queries less efficiently but will speed up the statistics gathering process.

- **Default**: 100
- **Impact**: Higher values improve the planner's ability to create efficient query plans by collecting more detailed statistics, but can also increase the overhead of `ANALYZE` operations.

==========

7. `autovacuum_vacuum_threshold`

- **Description**: Specifies the minimum number of tuple updates or deletes before `VACUUM` will be triggered by the autovacuum daemon.
- **Default**: 50
- **Impact**: A lower value causes more frequent vacuuming, which can reduce table bloat but might increase I/O usage. Higher values reduce vacuum frequency but can result in more bloat.

==========

8. `autovacuum_vacuum_scalefactor`

Description: The `autovacuum_vacuum_scalefactor` parameter in PostgreSQL helps decide when to run the `VACUUM` operation on tables based on their size.

In simple terms: It controls how large a table must be (based on the number of rows) before `VACUUM` is triggered by autovacuum.

- **Higher value**: It means the table needs to have more changes (inserts, updates, deletes) before autovacuum runs.
- **Lower value**: It makes autovacuum run more frequently, even on smaller tables with fewer changes.
- So, it adjusts how aggressively PostgreSQL cleans up old data in tables, balancing between performance and maintenance overhead.

- **Default**: 0.2
- **Impact**: The higher the value, the less frequently autovacuum will run. Reducing the scale factor makes autovacuum more aggressive.

==========

9. `autovacuum_analyze_threshold`

Description: The `autovacuum_analyze_threshold` parameter in PostgreSQL determines when the `ANALYZE` operation should be triggered automatically by the autovacuum process.

In simple terms: It controls how many changes (inserts, updates, or deletes) must occur in a table before PostgreSQL runs the `ANALYZE` operation, which updates the table statistics for query optimization.

- **Higher value**: Means the table must have more changes before autovacuum triggers an `ANALYZE`.
- **Lower value**: Means `ANALYZE` will run more frequently, even with fewer changes to the table.
- The goal is to ensure the database has up-to-date statistics for efficient query planning without running ANALYZE too often, which could impact performance.

- **Default**: 50
- **Impact**: Like `autovacuum_vacuum_threshold`, a lower value leads to more frequent analysis and better statistics, which can improve query planning.

==========

10. `autovacuum_analyze_scale_factor`

Description: The `autovacuum_analyze_scale_factor` parameter in PostgreSQL works together with the `autovacuum_analyze_threshold` to determine when the `ANALYZE` operation should be triggered by autovacuum.

In simple terms: It multiplies the number of rows in a table by this factor to help decide when to run `ANALYZE`.

- **Higher value**: It means the table needs to have more changes (in proportion to its size) before autovacuum will trigger an `ANALYZE`.
- **Lower value**: It makes autovacuum trigger an `ANALYZE` sooner, even for smaller changes.
- This parameter ensures that ANALYZE happens when there are enough changes in a table relative to its size, helping maintain accurate statistics without excessive processing.

- **Default**: 0.1
- **Impact**: Similar to `autovacuum_vacuum_scalefactor`, this setting influences how often `ANALYZE` operations are performed. Lower values result in more frequent analysis.

==========

11. `log_statement`

- **Description**: Controls which types of SQL statements are logged. It can be set to `none`, `ddl`, `mod`, or `all`.
- **Default**: `none`
- **Impact**: Setting this to `all` logs every SQL statement, which can generate significant log data. `ddl` only logs data definition statements like `CREATE`, `ALTER`, and `DROP`.

12. `log_connections`

- **Description**: If enabled, logs each new connection made to the database.
- **Default**: `off`
- **Impact**: Useful for tracking access and debugging connection issues. However, enabling it can increase log size, especially in high-traffic environments.

==========

13. `log_disconnections`

- **Description**: If enabled, logs each disconnection from the database.
- **Default**: `off`

- **Impact**: Helpful for monitoring connection lifecycle events, but it can lead to larger log files.

==========

14. `log_min_duration_statement`

- **Description**: Specifies the minimum execution time (in milliseconds) for a query to be logged. Queries running longer than this threshold are logged.
- **Default**: -1 (do not log)
- **Impact**: Useful for identifying slow queries, but setting this value too low can result in excessive logging.

==========

15. `log_statement_stats`

- **Description**: If enabled, logs detailed statistics on the execution of each SQL statement, including total time spent in execution.
- **Default**: `off`
- **Impact**: Provides detailed performance data, but can generate a large amount of log data, which might impact performance.

==========

16. `log_lock_waits`

- **Description**: If enabled, logs queries that wait for a lock for more than the specified duration (`deadlock_timeout`).
- **Default**: `off`
- **Impact**: Helps identify locking issues but may lead to large log files in a system with frequent lock contention.

==========

17. `log_filename`

- **Description**: Defines the name pattern for the log files.
- **Default**: `postgresql-%Y-%m-%d_%H%M%S.log`
- **Impact**: Useful for log rotation, allows specifying a custom naming convention for log files.

==========

18. `log_rotation_size`

- **Description**: Specifies the maximum size of a log file before it is rotated. The default is 10 MB.
- **Default**: 10MB
- **Impact**: Controls how often log files are rotated. Larger sizes mean fewer rotations but larger files.

==========

19. `checkpoint_timeout`

- **Description**: Specifies the maximum time (in seconds) between automatic WAL checkpoints. A checkpoint is when PostgreSQL writes all changes to disk.
- **Default**: 5 minutes
- **Impact**: Increasing this can improve performance by reducing the frequency of I/O operations but can delay recovery in the event of a crash. Decreasing this can increase I/O but improve crash recovery speed.

==========
==========

1. max_connections = 300

- **Description**: Defines the maximum number of concurrent connections allowed to the database.
- **Default**: 100
- **Impact**: A higher value allows more users or applications to connect to the database simultaneously. However, too many connections can lead to resource exhaustion, potentially affecting system performance. It should be set based on the expected number of concurrent connections.

==========

---

2. shared_buffers = 32000MB

- **Description**: Specifies the amount of memory used by PostgreSQL to cache data (shared memory).

shared_buffers is a PostgreSQL configuration setting that determines how much memory PostgreSQL allocates to store data in memory (RAM) to improve performance. This memory is used to cache the most frequently accessed data, so when a query is made, the database can quickly retrieve this data from memory instead of going to the disk, which is slower.

In simple terms, think of it as a storage area in your computer's memory that PostgreSQL uses to keep important data ready for fast access. The more memory allocated to shared_buffers, the less time PostgreSQL spends reading data from the hard disk, which can speed up database operations.

The default value is often set too low for many workloads, and increasing it can help improve performance, especially for larger databases. However, it shouldn't be set too high, as it could take memory away from other processes on the server.

- **Default**: 128MB
- **Impact**: A larger value can improve performance by reducing disk I/O because more data is kept in memory. However, setting it too high can cause memory contention with other processes, especially if the system doesn't have enough RAM.

==========

---

3. effective_cache_size = 96000MB

- **Description**: Represents the total amount of memory available for disk caching, including both PostgreSQL and the OS cache.

effective_cache_size is another PostgreSQL configuration setting that tells the database planner how much memory is available for caching data, not just within PostgreSQL but also the operating system's file system cache.

In simple terms, it helps PostgreSQL make better decisions when planning queries. If you set it too low, PostgreSQL might assume it has less memory available and make less efficient query plans, which could slow down performance. If you set it too high, it could overestimate available memory and also cause inefficient planning.

How it works:

- effective_cache_size doesn't allocate memory. It just estimates how much memory PostgreSQL can expect to use for caching data from both shared_buffers and the OS cache.
- For example, if your system has 16GB of RAM, and PostgreSQL has 4GB set for shared_buffers, you might set effective_cache_size to 12GB or higher, assuming that the operating system's file system cache can also hold a large portion of the data.

Setting effective_cache_size too low might result in PostgreSQL not taking advantage of the available memory and might make it choose slower query plans.

- **Default**: 4GB
- **Impact**: This value helps PostgreSQL decide whether to use indexes for queries. A larger value indicates that more data is likely to stay in memory, improving query

performance. If the value is too low, PostgreSQL may avoid using indexes, leading to slower query execution.

==========

---

4. maintenance_work_mem = 2GB

- **Description**: Defines the amount of memory allocated for maintenance operations, such as `VACUUM`, `CREATE INDEX`, and `ALTER TABLE`.
- **Default**: 64MB
- **Impact**: A larger value speeds up maintenance operations because more work can be done in memory, reducing the time needed for disk-based operations. However, setting it too high can lead to memory exhaustion if many maintenance tasks run concurrently.

---

5. checkpoint_completion_target = 0.9

- **Description**: Controls how aggressively PostgreSQL performs checkpoint operations (writes changes to disk).
- **Default**: 0.5
- **Impact**: A value closer to 1.0 spreads the checkpoint writes over a longer period, reducing I/O spikes and preventing performance degradation during checkpointing. However, setting it too high can delay the completion of checkpoints, causing longer recovery times in case of a crash.

---

6. wal_buffers = 16MB

- **Description**: Specifies the amount of memory used for Write-Ahead Log (WAL) data before it is written to disk.
- **Default**: 16KB
- **Impact**: A higher value reduces the frequency of disk writes for WAL data, improving write performance during heavy write operations. However, it uses more memory, so it should be set based on available system resources.

---

7. random_page_cost = 1.1

- **Description**: Represents the cost of fetching a random page from disk, relative to the cost of sequential access.

What is `random_page_cost`?

It is a setting in PostgreSQL that estimates the "cost" of fetching a page of data randomly from disk compared to fetching data sequentially. It helps PostgreSQL decide whether to use:

1. An **index scan** (random reads: fetch data from scattered locations).
2. A **sequential scan** (sequential reads: read data continuously).

Default Value

By default:

- `random_page_cost = 4`: Random reads are assumed to be **4 times more expensive** than sequential reads.

Example Scenario

Suppose we have a table called `sales` with **10 million rows**, and you want to find rows where `order_date = '2025-01-01'`.

With Sequential Scan:

1.

   - PostgreSQL reads all rows in the `sales` table sequentially, checking if `order_date` matches `2025-01-01`.
   - This uses sequential I/O, which is faster on traditional hard disks because the data is read in a continuous flow.

With Index Scan:

2.

   - PostgreSQL uses an index on `order_date` to directly locate the matching rows.
   - This involves random I/O, as PostgreSQL fetches data from scattered parts of the disk.

How `random_page_cost` Affects the Decision

Case 1: Traditional Hard Disk (`random_page_cost = 4`)

- PostgreSQL assumes random reads (index scan) are 4 times more expensive than sequential reads.
- If the query would touch a large number of rows (e.g., 100,000 rows), PostgreSQL might decide **a sequential scan is better** because reading continuously is faster.

Case 2: SSD (`random_page_cost = 1.5`)

- SSDs have faster random I/O, so the cost of random reads is closer to sequential reads.
- PostgreSQL may decide **an index scan is better**, even for larger numbers of rows, because random reads are no longer as expensive.

- **Lower** `random_page_cost`: Encourages PostgreSQL to use index scans (better for SSDs).

- **Higher** `random_page_cost`: Encourages PostgreSQL to use sequential scans (better for HDDs).

- **Default**: 4.0
- **Impact**: Lowering this value makes PostgreSQL prefer index scans over sequential scans. This is beneficial if you're using fast storage (like SSDs) where random access is cheaper. Setting it too low might result in suboptimal query plans for certain workloads.

==========

---

8. effective_io_concurrency = 200

- **Description**: Specifies the number of concurrent disk I/O operations PostgreSQL can perform for parallel queries.

What is effective_io_concurrency in Simple Terms?

effective_io_concurrency is a PostgreSQL setting that tells the database how many disk operations (Input/Output or I/O) it can perform **simultaneously**. It helps PostgreSQL optimize how it reads large amounts of data from the disk.

Simple Example to Understand

Imagine you are at a library trying to fetch 100 books:

1. If **10 people** help you grab the books (simultaneously), you can finish much faster.
2. But if only **1 person** is fetching the books (sequentially), it will take much longer.

Similarly:

- A **fast disk (SSD)** can handle many "helpers" fetching data at the same time.
- A **slow disk (HDD)** can only handle one or a few "helpers" at once.

effective_io_concurrency sets the number of "helpers" PostgreSQL should use.

- **Default**: 1
- **Impact**: A higher value improves performance on systems with high disk I/O throughput, such as SSDs, by allowing more parallel reads. However, setting it too high can lead to I/O contention and degrade performance.

Key Values to Use:

- **For SSDs or RAID storage:** Use higher values like `100` or `200`.
- **For HDDs:** Use a lower value like `1` or `4`.

- **High value:** Better for modern, fast disks that support many simultaneous operations.
- **Low value:** Better for older, slow disks to avoid overloading them.

==========

---

9. work_mem = 27306kB

- **Description**: Defines the amount of memory used for internal query operations like sorting and hashing (e.g., `ORDER BY`, `GROUP BY`).
- **Default**: 4MB

What is `work_mem` in PostgreSQL?

`work_mem` is a setting in PostgreSQL that determines how much **memory** each query operation (e.g., sorting, joining) can use **in RAM** before it switches to disk. It directly affects the performance of queries that process large amounts of data.

Simple Explanation:

- If a query operation (like a sort) needs more memory than `work_mem`, PostgreSQL will use temporary disk space, which is much slower than RAM.
- By increasing `work_mem`, you can allow queries to perform these operations in RAM, making them faster.

How it Works with an Example:

Scenario:

You are running this query on a table with **10 million rows**:

```
sql

Copy code

SELECT * FROM orders ORDER BY order_date;
```

This query sorts a large number of rows.

Case 1: `work_mem = 4MB`

- PostgreSQL allocates **4 MB of memory** for the sorting operation.
- Since sorting requires more than 4 MB (e.g., 50 MB is needed), PostgreSQL uses temporary files on disk for the excess data.
- **Result:** Sorting takes longer because disk access is slower than memory.

Case 2: `work_mem = 64MB`

- PostgreSQL allocates **64 MB of memory** for the sorting operation.

- The entire sorting can now happen in RAM without needing disk space.
- **Result:** Sorting is much faster because everything happens in memory.

Key Points:

1.Per Operation, Not Per Query:

1. `work_mem` applies to **each operation** in a query.
2. If a query has multiple operations (e.g., sorting and joining), each one gets its own `work_mem`.

2. Danger of Setting It Too High:

1. If too many queries run at the same time, and each uses a large `work_mem`, the server can run out of RAM, causing crashes or slowdowns.

Recommended Values:

- **Small databases or low activity:** Use the default value (4 MB).
- **High-performance systems or large queries:** Increase it to `32 MB` or more but monitor memory usage.
- For heavy analytic queries, you can temporarily set a higher value like `128 MB` for that session.

- **Impact**: A larger value allows more memory for complex queries, improving query performance by avoiding disk-based operations. However, setting it too high can cause excessive memory usage when many queries run concurrently.

==========

---

10. huge_pages = try

- **Description**: Enables the use of huge pages (large memory pages) for PostgreSQL memory allocation if the system supports them.
- **Default**: off

What is `huge_pages` in PostgreSQL?

`huge_pages` is a PostgreSQL setting that allows the database to use **large memory pages** (called "huge pages") provided by the operating system. It helps reduce memory overhead and improve performance, especially for systems with a large amount of RAM.

Simple Explanation:

- Normally, memory is allocated in **small pages** (e.g., 4 KB each).
- With `huge_pages`, memory is allocated in **larger pages** (e.g., 2 MB or more).
- This reduces the number of memory pages the system needs to manage, leading to:
  - Less overhead for managing memory.
  - Improved performance for large databases.

How to Enable `huge_pages`:

The `huge_pages` setting can have three values:

1. `on`**:** Use huge pages if the operating system supports them. PostgreSQL won't start if huge pages aren't available.
2. `off`**:** Do not use huge pages (default).
3. `try`**:** Use huge pages if possible; fallback to regular memory if huge pages aren't available.

Example to Understand:

Scenario Without Huge Pages:

- A server has **32 GB of shared_buffers** for PostgreSQL.
- The operating system uses **4 KB pages** to manage this memory.
- Total memory pages = 32 GB÷4 KB=8,388,608 pages

The OS has to manage over **8 million pages**, which adds overhead.

Scenario With Huge Pages:

- The OS is configured to use **2 MB huge pages**.
- Total memory pages = 32 GB÷2 MB=16,384 pages

Now, the OS only has to manage **16,384 pages**, significantly reducing overhead and improving performance.

Why is `huge_pages` Useful?

1. **Less Overhead:** The OS spends less time managing memory.
2. **Better Performance:** Applications (like PostgreSQL) run faster due to more efficient memory management.
3. **Improved Consistency:** Helps avoid memory fragmentation in large-memory servers.

When Should You Use `huge_pages`?

- **Recommended for:**
  - Servers with **high memory allocations** (e.g., `shared_buffers` > 8 GB).
  - Databases with heavy workloads.
- **Not Necessary for:**
  - Small or low-memory servers.

Checking Huge Pages Availability:

To verify if your system supports huge pages:

Check current huge page configuration:

1.cat /proc/meminfo | grep Huge

Output:-

HugePages_Total: 4096

HugePages_Free: 4090

Hugepagesize: 2048 kB

2. Configure huge pages in the OS (e.g., Linux): Add the following to /etc/sysctl.conf:-

```
vm.nr_hugepages = 2048

Then apply the changes:-

sysctl -p
```

Important Notes:

- If `huge_pages = on` and huge pages are not available, PostgreSQL will **fail to start**.
- Use `huge_pages = try` to enable huge pages without risking startup failure.

- **Impact**: Enabling huge pages improves memory efficiency and can enhance performance by reducing memory management overhead. However, it may not be supported on all systems and can lead to memory fragmentation if not configured correctly.

==========

11. min_wal_size = 2GB & max_wal_size = 8GB

What Are They?

1. `min_wal_size`:
The **minimum amount of disk space** reserved for WAL (Write-Ahead Log) files. PostgreSQL will keep at least this much WAL data even if it's not actively needed.
2. `max_wal_size`:
The **maximum amount of disk space** PostgreSQL can use for WAL files during high activity. If WAL usage exceeds this size, PostgreSQL triggers a checkpoint to free up space.

Key Differences:-

Parameter	Purpose	Behavior
min_wal_size	Minimum WAL storage on disk	WAL files below this size are never deleted.

Parameter	Purpose	Behavior
max_wal_size	Maximum WAL storage during activity	Triggers a checkpoint if this size is exceeded.

Key Concepts

1. **WAL File Size:** Fixed at 16 MB.

Each WAL file is 16 MB, and `min_wal_size`/`max_wal_size` determine how many such files PostgreSQL retains or uses.

2.`min_wal_size`: The total size of WAL files PostgreSQL guarantees to keep on disk, even if they're not actively used.

3.`max_wal_size`: The maximum size of WAL files PostgreSQL allows before triggering a **checkpoint** to recycle or archive files.

min_wal_size = 80MB

max_wal_size = 1GB

1. **WAL File Size:**

$$\text{WAL file size} = 16\,\text{MB}$$

2. **Minimum WAL Files (`min_wal_size`):**

$$\text{min\_wal\_size (in WAL files)} = \frac{\text{min\_wal\_size}}{\text{WAL file size}} = \frac{80}{16} = 5\,\text{files}$$

PostgreSQL keeps **5 WAL files** (80 MB) on disk at all times.

3. **Maximum WAL Files (`max_wal_size`):**

$$\text{max\_wal\_size (in WAL files)} = \frac{\text{max\_wal\_size}}{\text{WAL file size}} = \frac{1024}{16} = 64\,\text{files}$$

PostgreSQL allows up to **64 WAL files** (1 GB) before triggering a checkpoint.

Example Scenarios with Calculations

Scenario 1: Low Activity

- The database generates **2 WAL files** (32 MB) during the day.
- PostgreSQL retains **5 WAL files** (80 MB) due to `min_wal_size`.

Calculation:

- Files generated = **2 files × 16 MB = 32 MB**.
- Files retained (due to `min_wal_size`) = **5 files × 16 MB = 80 MB**.

Even though only 2 files are needed, PostgreSQL keeps **3 extra files (48 MB)** to meet the `min_wal_size` requirement.

Scenario 2: Medium Activity

- The database generates **20 WAL files** (320 MB) during peak hours.
- Since **320 MB < max_wal_size (1 GB)**, no checkpoint is triggered.

Calculation:

- Files generated = **20 files × 16 MB = 320 MB**.
- Files retained after cleanup = **5 files × 16 MB = 80 MB**.

PostgreSQL may clean up excess files after peak activity, but it keeps at least 5 files (80 MB) due to `min_wal_size`.

Scenario 3: High Activity (Exceeds `max_wal_size`)

- The database generates **80 WAL files** (1.28 GB) during a large bulk operation.
- This exceeds `max_wal_size` (1 GB), so PostgreSQL triggers a checkpoint.

Calculation:

- Files generated = **80 files × 16 MB = 1.28 GB**.
- Max allowed = **64 files × 16 MB = 1 GB**.
- Excess = **1.28 GB - 1 GB = 0.28 GB (or ~18 extra files)**.

PostgreSQL triggers a checkpoint to flush changes and recycle WAL files. After cleanup:

- PostgreSQL keeps **5 files × 16 MB = 80 MB** (due to `min_wal_size`).

==========

13. max_worker_processes , max_parallel_maintenance_workers, max_parallel_workers_per_gather,max_parallel_workers

max_worker_processes

What is a "worker process"?

In PostgreSQL, a **worker process** is like a helper that does tasks in the background. These helpers do things like running complex queries faster, cleaning up the database, or running maintenance tasks.

What does `max_worker_processes` do?

`max_worker_processes` tells PostgreSQL **how many helpers** (worker processes) it can use at once. If you have a higher value, PostgreSQL can do more things at the same time (in the background), which can make the system faster, especially when running big queries or doing tasks like database maintenance.

Example:

- If you set `max_worker_processes = 4`, PostgreSQL can only have **4 helpers** working at the same time.
- If you set `max_worker_processes = 10`, it can have **10 helpers** working at once.

If there are more tasks than there are workers, some tasks will have to **wait** for a free worker to become available.

Why does it matter?

- If your database is doing **a lot of work** at once (like running big reports or maintaining data), having more workers helps it get the work done faster.
- But, if you set it too high and your system doesn't have enough resources (like CPU or memory), it might actually slow things down because your computer will get overloaded.

Quick analogy:

Imagine a restaurant kitchen. The **workers** are the chefs, and the **tasks** are cooking orders.

- If you only have 2 chefs (workers), then only 2 orders can be cooked at once. If there are 10 orders, 8 of them will have to wait.
- If you have 10 chefs, you can cook all 10 orders at once, making the restaurant work faster.

So, `max_worker_processes` is like deciding how many chefs you want in your kitchen. More chefs (workers) mean faster work, but it's important not to overload the kitchen (your system).

---

max_parallel_maintenance_workers

`max_parallel_maintenance_workers` is a PostgreSQL configuration setting that controls the **maximum number of parallel workers** that can be used for **maintenance operations** like **vacuuming**, **index creation**, or **analyze**.

Simple Explanation:

- **Maintenance operations** are tasks that PostgreSQL performs to keep the database healthy, such as cleaning up old data (vacuuming) or updating statistics (analyze).
- With `max_parallel_maintenance_workers`, you are telling PostgreSQL how many "helper workers" (background processes) it can use to do these maintenance tasks **in parallel** (at the same time).
- More workers = faster maintenance tasks, but only if your system has the resources to handle it.

Example:

Let's say you set `max_parallel_maintenance_workers = 4`:

- When PostgreSQL needs to **vacuum** a large table (remove dead rows and free up space), it can use **up to 4 parallel workers** to do this task faster.
- If you set it to `2`, PostgreSQL can only use **2 parallel workers** at most for such tasks.

Why does it matter?

- If you have a large database with tables that need frequent maintenance (vacuum, index creation, etc.), increasing `max_parallel_maintenance_workers` can speed up these operations.
- But, **too many parallel workers** may strain your system's CPU or memory, so it's important to adjust it based on your hardware.

Example Scenario:

Imagine you need to rebuild an index on a very large table. By setting `max_parallel_maintenance_workers = 4`, PostgreSQL could use 4 parallel workers to rebuild the index, which can **finish faster** than if only 1 worker was used (default).

==========

Sure! Let me break it down into simpler terms with examples.

1. `max_worker_processes`

- **What it is:** This setting controls the **maximum number of background helper processes** PostgreSQL can use for **any type of task**.
- **Example:** If you set `max_worker_processes = 8`, PostgreSQL can run up to 8 background workers simultaneously for tasks like parallel queries, maintenance, and other operations.

When it's used:

- It defines the total pool of workers available for various tasks like running parallel queries, maintenance tasks, etc.
- Think of it like saying, "PostgreSQL can have a maximum of 8 helpers available to do different jobs at the same time."

---

2. `max_parallel_maintenance_workers`

- **What it is:** This setting controls how many **parallel worker processes** PostgreSQL can use specifically for **maintenance operations** like vacuuming, creating indexes, or analyzing tables.
- **Example:** If you set `max_parallel_maintenance_workers = 4`, when you run a maintenance task like vacuuming a large table, PostgreSQL can use **up to 4 parallel workers** to speed up the process.

When it's used:

- This is specific to maintenance operations, such as cleaning up dead rows (vacuum) or building indexes.
- Think of it like saying, "When we're cleaning the database or building indexes, we can use a maximum of 4 helpers to do the job faster."

---

3. `max_parallel_workers_per_gather`

- **What it is:** This setting controls how many parallel workers can be used for a **single query** when PostgreSQL is running a **parallel query**. It determines how many workers can work together to process parts of a large query.

- **Example:** If you set `max_parallel_workers_per_gather = 4`, when PostgreSQL executes a large query, it can use up to **4 workers** in parallel to break down and process the query faster.

When it's used:

- This setting specifically affects **parallel queries**, not maintenance tasks.
- Think of it like saying, "When we're running a big query, we can use up to 4 helpers to split the work and speed things up."

---

4. `max_parallel_workers`

- **What it is:** This setting controls the **total number of parallel workers** that PostgreSQL can use across **all queries and tasks**. It's the upper limit of workers for all parallel tasks combined.
- **Example:** If you set `max_parallel_workers = 8`, it means that across the entire PostgreSQL instance, there can be a **maximum of 8 workers** used for any parallel tasks.

When it's used:

- This is the global limit for parallel workers across all operations (queries, maintenance, etc.).
- Think of it like saying, "The maximum number of helpers PostgreSQL can use for parallel tasks across everything is 8."

Key Differences:

`max_worker_processes`:

- Sets the **total** number of worker processes available for all tasks, including parallel queries, maintenance, and background jobs.
- It is a global setting that defines the maximum helpers PostgreSQL can have.

`max_parallel_maintenance_workers`:

- Limits the number of parallel workers only for **maintenance operations** (like vacuuming or index creation).
- It is more specific to maintenance tasks.

`max_parallel_workers_per_gather`:

- Limits the number of parallel workers PostgreSQL can use **per query** during a **parallel query execution**.
- It specifically affects how many helpers are used to speed up **individual queries**.

`max_parallel_workers`:

- Sets the global limit on **how many parallel workers** PostgreSQL can use in total across **all parallel operations** (queries and maintenance tasks).
- It affects the entire database's ability to run tasks in parallel.

Example Scenario for Clarity:

Let's say you have a large database and you want to speed up the performance of both your maintenance tasks and your queries.

- If you set:
  - `max_worker_processes = 8` (total available workers),
  - `max_parallel_maintenance_workers = 4` (for maintenance tasks like vacuuming),
  - `max_parallel_workers_per_gather = 2` (workers per query for parallel queries),
  - `max_parallel_workers = 8` (total workers for all parallel tasks).

Now, when PostgreSQL runs:

- For a **large query**, it will use **up to 2 parallel workers** (based on `max_parallel_workers_per_gather`).
- For **maintenance tasks** like vacuuming or indexing, PostgreSQL can use **up to 4 parallel workers** (based on `max_parallel_maintenance_workers`).
- **In total**, PostgreSQL can have **8 parallel workers** (based on `max_parallel_workers`) running tasks at the same time.

If there are more tasks, some may have to wait until a worker is available.

Parameter	max_worker_processes	max_parallel_workers
Purpose	Limits **total background worker processes** available for any task (queries, maintenance, extensions, etc.)	Limits the **total number of parallel workers** for **parallel query execution**
Scope	Affects **all tasks** in PostgreSQL (queries, maintenance, background jobs, etc.)	Affects only **parallel queries** (SELECT, JOIN, aggregation)
Use cases	Parallel queries, maintenance (vacuum, indexing), background tasks	Parallel queries and operations that split work between multiple workers
Example	max_worker_processes = 8: 8 total worker processes for all operations	max_parallel_workers = 4: 4 parallel workers for query execution

If you have:

- `max_worker_processes = 8` (up to 8 workers available),
- `max_parallel_workers = 4` (maximum 4 workers for parallel query execution).

This means that up to **8 worker processes** can be running at the same time for all tasks (queries, maintenance, etc.), but only **4 workers can be used for parallel query processing** at any given time.

==========

Check point:-

What is a Checkpoint?

A checkpoint in PostgreSQL ensures that all changes in memory (shared buffers) are written to disk. This is important for data safety and crash recovery. However, writing a lot of data to disk at once can use a lot of system resources (CPU, disk I/O), causing performance slowdowns.

What Does `checkpoint_completion_target` Do?

The `checkpoint_completion_target` setting determines how much of the checkpoint interval (time between checkpoints) PostgreSQL should use to complete the checkpoint. By spreading the work across a longer time, the system avoids sudden resource spikes.

The value is a fraction (e.g., `0.5`, `0.7`, `0.9`) that represents how much of the checkpoint interval PostgreSQL should use.

Example Scenario

Let's say:

- The `checkpoint_timeout` is set to **10 minutes** (600 seconds).
- `checkpoint_completion_target` is set to **0.5**.

1.

How it works:

2.

- The checkpoint timeout is 10 minutes.
- With a `checkpoint_completion_target` of **0.5**, PostgreSQL will aim to complete the checkpoint in **5 minutes** (50% of 10 minutes).
- This means PostgreSQL will write changes to disk more quickly, but it might cause higher disk I/O usage during that 5 minutes.

3.

Changing the target:

4.

- If you increase `checkpoint_completion_target` to **0.9**, PostgreSQL will spread the checkpoint work over **9 minutes** (90% of 10 minutes).
- This spreads out the disk writes more evenly, reducing spikes in disk activity and lowering the impact on database performance.

Why Is This Useful?

If the checkpoint is completed too quickly (e.g., with a low `checkpoint_completion_target`), it can create performance problems:

- **High disk usage** as PostgreSQL writes a large amount of data in a short time.
- **Slower queries** because disk I/O is being used heavily by the checkpoint.

By increasing the `checkpoint_completion_target`, PostgreSQL writes the data more slowly over time, keeping performance more stable.

Key Points:

1. `checkpoint_completion_target = 0.5`: Complete the checkpoint in **50% of the checkpoint timeout** (faster, but more resource-intensive).
2. `checkpoint_completion_target = 0.9`: Complete the checkpoint in **90% of the checkpoint timeout** (slower, smoother performance).

Best Practices:

- If your system has high disk I/O during checkpoints, consider increasing `checkpoint_completion_target` to **0.7 or 0.9** to reduce the impact.
- If you need faster checkpoint completion (e.g., for crash recovery), keep it lower (e.g., **0.5**).

Work_mem explaination:-

"Per Operation, Not Per Query" - Meaning in Simple Terms

The `work_mem` setting applies to **each operation** within a query, not the entire query as a whole.

What Does This Mean?

When you run a query in PostgreSQL, the query can have multiple steps (or operations), such as:

1. Sorting
2. Joining tables
3. Grouping data
4. Creating hash tables

Each of these operations gets its **own memory allocation** based on the `work_mem` setting.

Example to Understand:

Query:

```
SELECT customers.name, SUM(orders.amount)FROM customersJOIN orders ON customers.id = orders.customer_idGROUP BY customers.nameORDER BY SUM(orders.amount) DESC;
```

What Happens Here:

This query has **three operations**:

1. **Joining:** Combine `customers` and `orders` tables.
2. **Grouping:** Group data by `customers.name` and calculate the total amount (`SUM`).
3. **Sorting:** Sort the results by the total amount in descending order.

If `work_mem = 4MB`:

- **Join Operation:** PostgreSQL gets **4 MB** to create a hash table in memory for joining the tables. If this exceeds 4 MB, it uses the disk.
- **Grouping Operation:** Another **4 MB** is allocated to group data and calculate `SUM`.
- **Sorting Operation:** A separate **4 MB** is allocated to sort the results.

Total Memory Used:
If all operations happen simultaneously, PostgreSQL will use **4 MB × 3 = 12 MB** for this query.

If `work_mem = 64MB`:

- Each operation gets **64 MB** instead of 4 MB.
- PostgreSQL is less likely to use the disk, making the query much faster.

Total Memory Used:
If all operations happen simultaneously, PostgreSQL will use **64 MB × 3 = 192 MB** for this query.

Why is This Important?

- If you increase `work_mem` too much and many queries run simultaneously, the total memory usage can exceed the server's available RAM, causing performance issues or crashes.
- Example: If 50 queries each have 3 operations and `work_mem = 64MB`:
    - Total memory used = `50 × 3 × 64 MB = 9.6 GB`.

Summary:

- **Each operation** in a query gets its own `work_mem` allocation.
- For complex queries with many operations, the total memory usage will be **work_mem × number of operations**.
- Be cautious when setting a high `work_mem`, especially on systems with many concurrent queries.

The work_mem setting in PostgreSQL defines how much memory a single operation in a query can use before switching to disk. If you have multiple SELECT queries running, each query (and each operation in those queries) will use its own work_mem allocation. Let's break it down step by step.

Key Points About `work_mem`:

Per Operation, Not Per Query:

1. `work_mem` applies to each operation in a query (e.g., sorting, hashing, joining), not the query as a whole.
2. Complex queries with multiple operations (e.g., multiple joins or sorts) can use `work_mem` multiple times.

Multiple Queries:

1. If multiple `SELECT` queries run at the same time, each query (and its operations) gets its own `work_mem`.

Total Memory Usage

1. Total memory consumed depends on: Total Memory Usage=

Total Memory Usage=work_mem×Number of Operations per Query)×(Number of Queries Running Simultaneously)

Example Scenario:

Database Setup:

- `work_mem = 4MB` (default value).
- You are running 3 `SELECT` queries simultaneously.

Query 1: Simple Query

```
SELECT * FROM orders ORDER BY order_date;
```

- **Operation:** Sorting (`ORDER BY`).
- Memory Usage:
  - Sorting uses **4 MB** of memory (1 operation × 4 MB).

Query 2: Complex Query

```
SELECT o.customer_id, SUM(o.total)
FROM orders o
JOIN customers c ON o.customer_id = c.id
```

```
GROUP BY o.customer_id

ORDER BY SUM(o.total) DESC;
```

Operations in Query 2:

1. **Hash Join:** Joins `orders` and `customers`.
2. **Group By:** Groups data by `customer_id` and calculates the `SUM`.
3. **Sort:** Orders the results by `SUM(total)`.

Memory Usage: 3operations×4MB=12MB per query

What Happens If More Queries Run?

If more queries are executed, PostgreSQL will allocate more memory for each query's operations. If the total memory usage exceeds the server's physical RAM, the system may:

1. Start swapping (using disk instead of RAM), which slows everything down.
2. Fail if the system runs out of memory entirely.

Best Practices for `work_mem`:

1.

Estimate Total Usage:
Set `work_mem` based on your system's available RAM and expected query concurrency:

$$\text{Max Work Memory} = \frac{\text{Available RAM}}{\text{Max Concurrent Queries} \times \text{Operations per Query}}$$

Example:

- Server has 16 GB RAM.
- Max 50 concurrent queries.
- Average 2 operations per query.

$$work\_mem = \frac{16\,\text{GB}}{50 \times 2} = 160\,\text{MB per operation.}$$