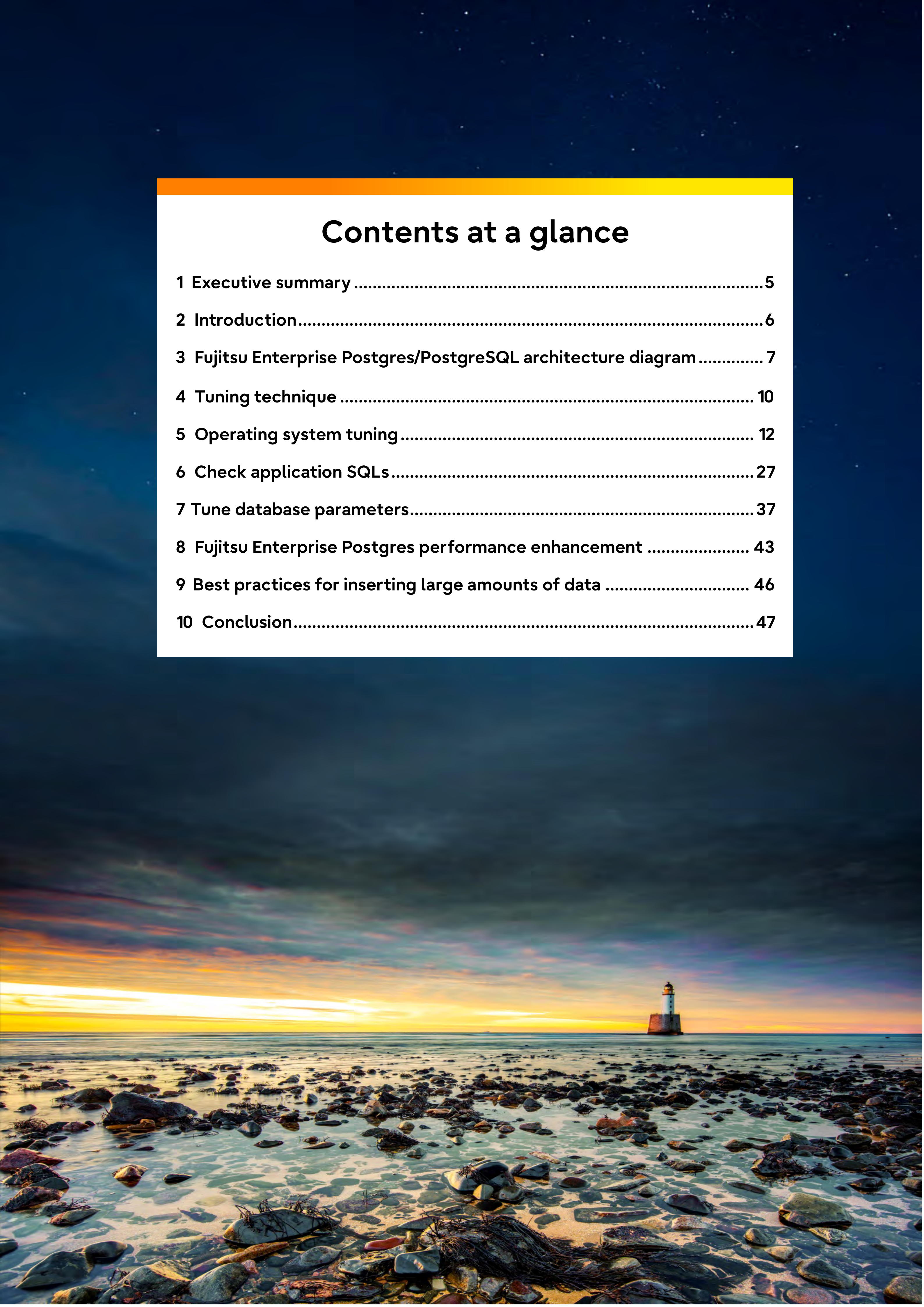
# Fujitsu Enterprise Postgres/PostgreSQL performance optimization

Best practices



## Contents at a glance

# Table of contents

## 1. Executive summary

Fujitsu Enterprise Postgres/PostgreSQL is one of the most advanced, feature-rich, and widely adopted open-source relational database systems in the world. It powers everything from small web applications to large-scale enterprise data warehouses and mission-critical financial systems. While Fujitsu Enterprise Postgres/PostgreSQL comes with smart defaults that allow it to run on a broad range of hardware, these defaults are intentionally conservative. They prioritize compatibility and safety over squeezing every drop of performance out of modern servers.

This is why performance tuning is essential. It transforms a generic Fujitsu Enterprise Postgres/PostgreSQL installation into a finely tuned engine that leverages the full power of your hardware, operating system, and workload characteristics. Proper tuning can mean the difference between an application that struggles under load and one that scales effortlessly.

This guide provides a comprehensive approach to Fujitsu Enterprise Postgres/PostgreSQL  performance tuning. It goes beyond simply listing configuration parameters. Instead, it takes a holistic approach starting with the operating system, delving into hardware choices and kernel tuning, guiding you through optimizing SQL queries and indexing strategies, and finally showing how to fine-tune Fujitsu Enterprise Postgres/PostgreSQL's own internal parameters.

Readers will learn:

- How to prepare the operating system for a high-performance database workload by adjusting memory, I/O, and scheduler settings.
- The impact of file systems, RAID levels, and disk choices on Fujitsu Enterprise Postgres/PostgreSQL throughput and latency.
- How to analyze application SQLs, spot anti-patterns, and rewrite queries for efficiency.
- How to adjust Fujitsu Enterprise Postgres/PostgreSQL settings like shared_buffers, work_mem, and checkpoint_completion_target to better fit your specific workload.

The aim of this guide is to equip database administrators, system engineers, developers, and DevOps teams with clear, actionable insights. Each recommendation is explained with its rationale, so you can understand not just *what* to tune, but also *why*. The end goal is a Fujitsu Enterprise Postgres/PostgreSQL deployment that is faster, more stable, more predictable, and capable of scaling as your demands grow.

# 2. Introduction

At its core, Fujitsu Enterprise Postgres/PostgreSQL is designed to be a general-purpose, standards-compliant database. Its out-of-the-box settings allow it to run reliably on a wide range of systems, from laptops to enterprise servers. However, the same generic approach that ensures broad compatibility also means it leaves much of your hardware potential unused.

For example:

- By default, Fujitsu Enterprise Postgres/PostgreSQL might allocate only a modest amount of memory for caching data, leading to frequent disk reads even if you have large amounts of RAM.
- Its planner cost settings are tuned for spinning disks, which could cause suboptimal plans on SSD-heavy systems.
- Checkpointing and WAL settings are conservative to ensure safety but might not be balanced for your write-heavy workload, leading to sudden I/O spikes.

All of this means that if you run Fujitsu Enterprise Postgres/PostgreSQL in production, tuning is not optional. It is a critical responsibility to ensure your database keeps up with user demand and doesn't become the bottleneck that slows down your entire application.

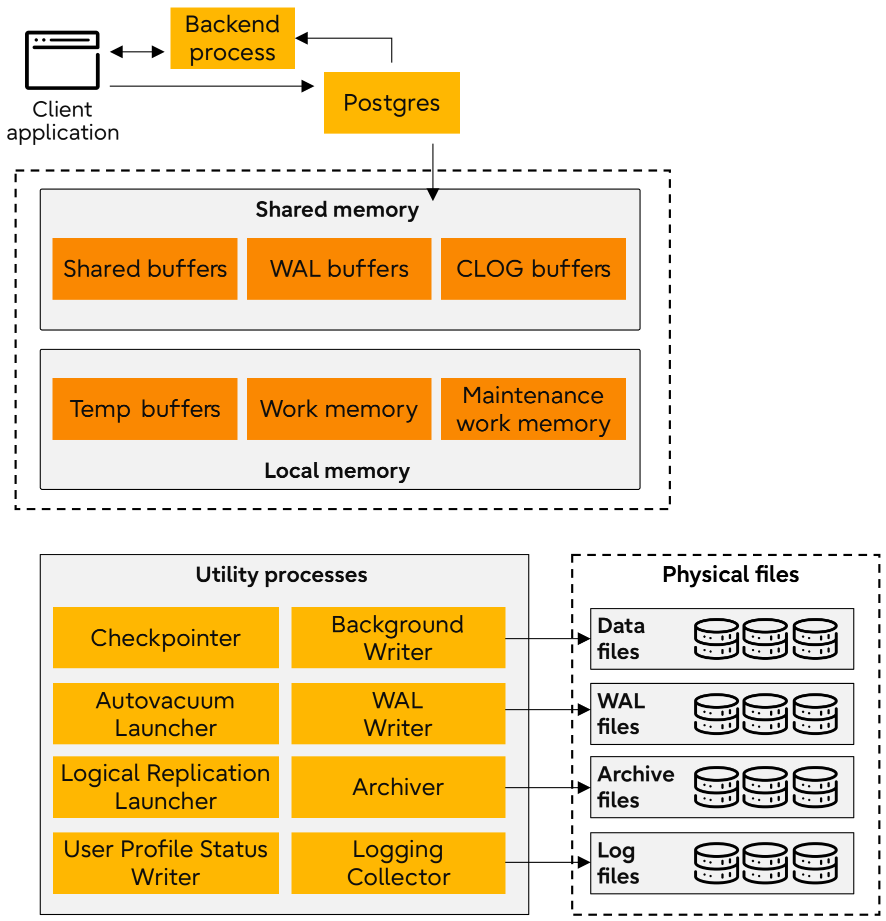
## 2.1. What this guide covers

This guide starts by examining the underlying system and hardware layer, because that's where the database's performance ultimately depends. We'll look at Linux kernel parameters that affect memory management and I/O, the right filesystems and mount options, and the best practices for SSDs and RAID configurations.

Next, we'll explore SQL optimization, because no matter how well your system is tuned, inefficient queries can cripple performance. We'll show you how to identify sequential scans that could be avoided, detect unused or duplicate indexes that waste resources, and rewrite typical anti-patterns like `SELECT *`.

Finally, we'll dive into Fujitsu Enterprise Postgres/PostgreSQL's own parameters, explaining how settings like `shared_buffers`, `work_mem`, and `checkpoint_timeout` directly impact memory, CPU, and disk usage. Each parameter is discussed in a way that connects it to real-world workload symptoms, so you can tune based on evidence not guesswork.

## 3. Fujitsu Enterprise Postgres/PostgreSQL architecture diagram



Note: **User Profile Status Writer** is specific to Fujitsu Enterprise Postgres background process.

Before we explore ways to optimize ujitsu Enterprise Postgres (built on PostgreSQL), it's important to revisit its underlying memory architecture. Understanding how memory is structured and utilized forms the foundation for any meaningful performance tuning.

### 3.1. Memory architecture of Fujitsu Enterprise Postgres

The memory architecture of a Fujitsu Enterprise Postgres (including PostgreSQL) instance is carefully organized to ensure efficient database operations. It can broadly be classified into two main categories: the local memory area, which is allocated individually by each backend process for its exclusive use, and the shared memory area, which is used collectively by all processes of a Fujitsu Enterprise Postgres/PostgreSQL server instance.

### 3.1.1. Shared memory area

When a Fujitsu Enterprise Postgres server (including PostgreSQL) starts up, it allocates a shared memory area. This area is crucial because it serves as a common workspace for all backend processes and is subdivided into several fixed-sized sub-areas, each serving a distinct purpose.

One of the key components of the shared memory area is the shared buffer pool. This is essentially the memory cache in which all data modifications and reads occur. Any SQL operation like `INSERT`, `UPDATE`, `DELETE`, or `SELECT` must go through this buffer because direct access to data files on disk by user processes is not allowed. When data is modified, it resides in the shared buffer as dirty data until it is eventually flushed to the actual data files on disk by a background process known as the background writer (BG Writer). The size of the shared buffer pool is controlled by the `shared_buffers` parameter in the postgresql.conf configuration file. Tuning this parameter effectively is critical for balancing memory usage and disk I/O.

Another important shared memory component is the WAL (Write-Ahead Log) buffer, which temporarily holds metadata about data changes. These WAL buffers ensure durability by recording the information needed to reconstruct data modifications during crash recovery. The WAL data is then flushed to persistent storage files called WAL files by the WAL writer background process. The `wal_buffers` parameter governs the amount of memory reserved for these buffers.

Additionally, there is the CLOG (commit log) buffer, which is dedicated to tracking the transaction statuses whether they are committed, in-progress, or aborted. This buffer helps in quickly determining the outcome of transactions without frequently accessing the commit log files on disk. Unlike shared buffers or WAL buffers, the CLOG buffer does not have a user-configurable parameter; it is automatically managed by Fujitsu Enterprise Postgres/PostgreSQL's internal mechanisms and is shared by all server processes.

### 3.1.2. Local memory area

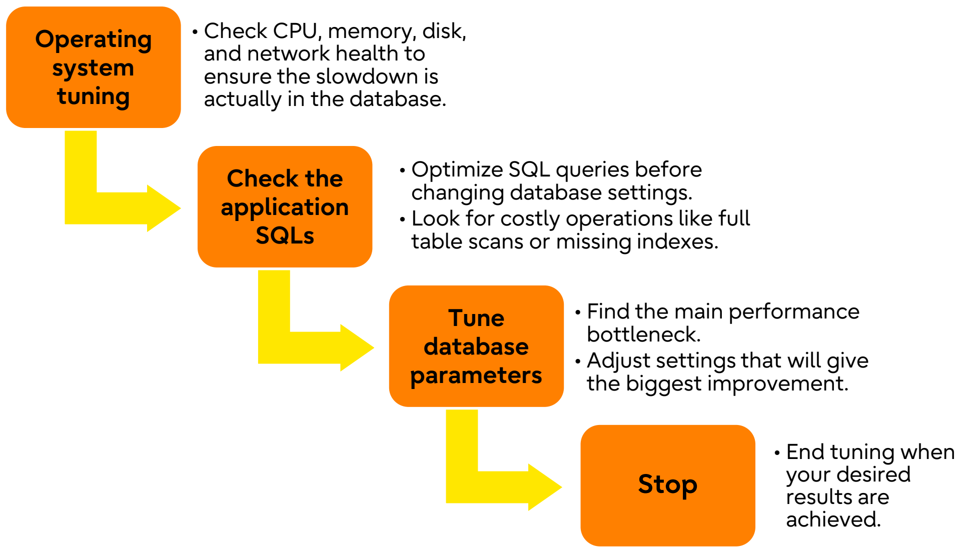
Each backend process in Fujitsu Enterprise Postgres also maintains its own local memory area, which is private to that process. This memory is allocated at the start of a session and is essential for query processing. It is subdivided into several important areas, with either fixed or dynamically determined sizes depending on the workload and configuration.

The most prominent among these is the work memory, governed by the `work_mem` parameter. This memory is allocated whenever Fujitsu Enterprise Postgres/PostgreSQL needs to perform sort operations (such as those arising from `ORDER BY`, `DISTINCT`, or `merge joins`) or to build hash tables (used in hash joins, hash-based aggregations, or IN clause evaluations). In complex SQL statements, multiple sort and hash operations can occur simultaneously, and each gets its own chunk of work memory. This is why setting `work_mem` too high can be risky, as it might consume excessive RAM if many such operations run in parallel, potentially starving the system of memory needed for other tasks.

Another vital segment is the maintenance work memory, which comes into play during maintenance activities like creating or rebuilding indexes (`REINDEX`), adding foreign key constraints, and performing `VACUUM` operations. This area is regulated by the `maintenance_work_mem` parameter and generally benefits from being larger than `work_mem` since maintenance operations can be intensive and benefit from more memory.

Lastly, there are temporary buffers, used specifically when dealing with temporary tables often during large sorts or hash operations that cannot be entirely contained within the regular work memory. These buffers are session-specific, meaning each database session manages its own pool of temporary buffers independently. This helps isolate the memory consumption of temporary operations to the scope of individual sessions.

# 4. Tuning technique


Operating system tuning
• Check CPU, memory, disk, and network health to ensure the slowdown is actually in the database.
Check the application SQLs
• Optimize SQL queries before changing database settings.
• Look for costly operations like full table scans or missing indexes.
Tune database parameters
• Find the main performance bottleneck.
• Adjust settings that will give the biggest improvement.
Stop
• End tuning when your desired results are achieved.

## 4.1. Introduction to Fujitsu Enterprise Postgres/PostgreSQL tuning approach

Performance tuning for Fujitsu Enterprise Postgres/PostgreSQL (or any enterprise-grade database like FUJITSU Enterprise Postgres) is far more than just tweaking configuration files. It is a careful, systematic process that addresses performance holistically from the underlying infrastructure all the way to the database engine itself. The diagram you've provided illustrates a proven tuning methodology in a stepwise flow. It starts by verifying the foundation, moves upward through the application layer, and finally focuses on the Fujitsu Enterprise Postgres/PostgreSQL server settings. Importantly, it also emphasizes knowing when to stop tuning to prevent unnecessary risk or complexity.

### 4.1.1. Start with the operating system

Any database system, including Fujitsu Enterprise Postgres/PostgreSQL, ultimately depends on the health and efficiency of the operating system it runs on. This is why the very first step in the tuning approach is to check the OS health to ensure that the problem truly lies within the database layer. If the underlying hardware is stressed whether due to CPU overload, memory exhaustion leading to swapping, disk I/O bottlenecks, or network instability then no amount of Fujitsu Enterprise Postgres/PostgreSQL tuning will fix the root problem. Administrators should use OS-level tools like `vmstat`, `iostat`, `sar`, `top`, `mpstat`, etc. to assess CPU, memory, and disk performance, making sure the system is not under excessive load. Only after ruling out these foundational issues should attention turn to Fujitsu Enterprise Postgres/PostgreSQL itself.

### 4.1.2. Check the application SQLs

Once the operating system is confirmed to be healthy, the next step is to examine the application layer. Often, what appears to be a database performance problem is caused by inefficient SQL queries, missing indexes, or flawed application logic. The tuning methodology stresses that before making any changes to the database server configuration, you should optimize the SQL issued by the application. This includes rewriting queries to avoid unnecessary sequential scans, ensuring appropriate indexes exist, batching operations instead of issuing thousands of individual statements, and using `EXPLAIN` plans to diagnose costly joins or sorts. Optimizing here can yield dramatic improvements because bad SQL tends to overwhelm database resources regardless of server tuning.

### 4.1.3. Tune database server configuration

With the operating system stable and the application generating efficient SQL, attention can then move to tuning Fujitsu Enterprise Postgres/PostgreSQL 's internal configuration. This step involves identifying the biggest bottleneck perhaps through tools like `pg_stat_activity`, `pg_stat_statements`, or external monitoring dashboards and then tuning the area with the greatest potential impact. That might mean increasing `shared_buffers` to better cache frequently accessed data, adjusting `work_mem` to reduce disk-based sorts, modifying WAL settings for smoother checkpoints, or fine-tuning autovacuum parameters to control table bloat and few more parameters. Crucially, each change should be made incrementally and validated with careful measurement to avoid introducing new issues.

This structured tuning methodology is designed to be logical, sequential, and evidence driven. By beginning with the operating system, then addressing the application layer, and finally tuning the Fujitsu Enterprise Postgres (including PostgreSQL) configuration itself, administrators can tackle performance issues at the right layer, in the right order. In short, this approach is not just about speeding up Fujitsu Enterprise Postgres/PostgreSQL; it's about doing so in a sustainable, risk-aware way that ensures your database infrastructure remains robust and efficient.

# 5. Operating system tuning

## 5.1. Kernel parameters

Kernel parameters are low-level settings that control how the Linux operating system manages critical system resources such as memory, shared memory segments, semaphores, file handles, and virtual memory behaviour. These parameters play a fundamental role in the overall performance and stability of applications running on the server including Fujitsu Enterprise Postgres/PostgreSQL. A Fujitsu Enterprise Postgres (including PostgreSQL) database relies heavily on efficient memory allocation, I/O handling, and process management, all of which are influenced by the Linux kernel's configuration. If these parameters are not tuned properly or are left at conservative defaults, the database server might experience degraded performance, increased latency, or even failures under heavy load.

For example, insufficient shared memory settings could prevent Fujitsu Enterprise Postgres/PostgreSQL from allocating the necessary `shared_buffers`, or poor virtual memory settings could lead to excessive swapping. Therefore, it's essential to carefully configure these kernel parameters in line with the specific hardware resources of the database server and the nature of its workload whether it's OLTP with many small transactions or large analytical queries to ensure Fujitsu Enterprise Postgres/PostgreSQL can fully utilize the underlying system capabilities.

### 5.1.1. Shared memory and semaphores: How Fujitsu Enterprise Postgres/PostgreSQL uses them

Fujitsu Enterprise Postgres/PostgreSQL requires the operating system to support inter-process communication (IPC), which mainly involves shared memory segments and semaphores. On Unix-like systems, this is typically provided via System V IPC, POSIX IPC, or sometimes both. Windows uses a different approach that isn't covered here.

By default, Fujitsu Enterprise Postgres/PostgreSQL allocates a very small amount of System V shared memory just to enough for control data and relies more on anonymous memory via mmap for its larger shared buffer needs. However, depending on the shared memory type setting, Fujitsu Enterprise Postgres/PostgreSQL can be configured to use a single large System V shared memory block instead. In addition, when the server starts, it creates many semaphores, which may be used either by System V or POSIX implementations depending on the platform. For example, Linux and FreeBSD generally use POSIX semaphores, while other systems may rely on System V.

### 5.1.2. System limits and what happens when you hit them

System V IPC resources are controlled by kernel-wide limits. If Fujitsu Enterprise Postgres/PostgreSQL tries to allocate more shared memory or semaphores than the system allows, it won't start. It will typically print an error message indicating which limit was exceeded and how to adjust it. Common errors include "`could not create shared memory segment`" or confusing messages like "`No space left on device`" even though disk space is fine.

The most important shared memory parameters are:

- **SHMMAX:** The maximum size of a single shared memory segment. This must be large enough to accommodate what Fujitsu Enterprise Postgres/PostgreSQL needs.
- **SHMMIN:** The minimum size of a shared memory segment, typically just 1 byte.
- **SHMALL:** The total amount of shared memory pages that can be used across the system. This often needs to be set to roughly same as `SHMMAX` or `ceil (SHMMAX/PAGE_SIZE)`, plus extra for other processes.
- **SHMSEG:** The maximum number of shared memory segments a single process can attach to. Fujitsu Enterprise Postgres/PostgreSQL generally only needs 1.
- **SHMMNI:** The maximum number of shared memory segments system-wide, which must allow for Fujitsu Enterprise Postgres/PostgreSQL and other applications. This often needs to be set to roughly the same as `SHMSEG` plus extra for other processes.

For semaphores, key parameters include:

- **SEMMNI:** Controls the maximum number of semaphore sets. Fujitsu Enterprise Postgres/PostgreSQL groups semaphores into sets of 19 plus a special "magic number" semaphore, so this needs to be calculated based on connection settings. This often needs to be set at least `ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 7) / 19)`, plus extra for other processes.
- **SEMMNS:** The total number of semaphores allowed on the system. It needs to be large enough to support all connections, autovacuum workers, WAL senders, and background workers. This often needs to be set at least `ceil((max_connections + autovacuum_max_workers + max_wal_senders + max_worker_processes + 7) / 19) * 20`, plus extra for other processes.
- **SEMMSL:** The maximum number of semaphores per set, which Fujitsu Enterprise Postgres/PostgreSQL needs to be at least 20.

### 5.1.3. How do these parameters matter?

For each allowed database connection (`max_connections`), autovacuum worker (`autovacuum_max_workers`), WAL sender (`max_wal_senders`), and background worker (`max_worker_processes`), Fujitsu Enterprise Postgres/PostgreSQL allocates a semaphore. For the System V model, these are grouped in sets of 19, plus one extra semaphore per set. That means even modest changes to your connection counts can require significantly more semaphores.

For example, if you raise `max_connections` from 100 to 500, your required `SEMMNS` and `SEMMNI` values must increase accordingly. If these values are too low, Fujitsu Enterprise Postgres/PostgreSQL may fail to start or accept new connections, often showing a misleading message like "`No space left on device`".

The concept of "grouped in sets of 19" is explicitly defined in PostgreSQL's source code (found in `src/backend/port/sysv_sema.c`) as:

```
#define SEMAS_PER_SET 19
```

This was chosen to stay safely below typical SEMMSL defaults (commonly around 25) on Unix/Linux systems, thereby avoiding immediate kernel limit violations while minimizing the number of separate system calls and management overhead.

### 5.1.4. POSIX vs System V

When Fujitsu Enterprise Postgres/PostgreSQL uses POSIX semaphores which is the default and preferred implementation on modern Linux systems the situation is considerably simpler. POSIX semaphores generally do not depend on hard-coded kernel-wide constraints like `SEMMNI` or `SEMMNS`, largely eliminating the need for extensive tuning of `/proc/sys/kernel/sem`. Each semaphore is created as an independent object, so there is no concept of grouping into sets of 19, nor are there strict system-wide caps that can block the server from starting. However, it's important to understand that the total number of semaphores needed by Fujitsu Enterprise Postgres/PostgreSQL does not change it still requires one semaphore per allowed connection or worker process. The key difference is that with POSIX semaphores, you are far less likely to run into operating system-level barriers that prevent the database from scaling to the workloads you've configured.

### 5.1.5. Why tune kernel parameters?

As we understand, Fujitsu Enterprise Postgres/PostgreSQL fundamentally depends on the operating system kernel to provide shared memory and semaphores for efficient multi-process operation. Kernel parameters such as `SHMMAX`, `SHMALL`, `SEMMNI`, and `SEMMNS` directly control these critical resources under the System V model. Without properly configuring these parameters to align with your `max_connections`, expected workload, and memory requirements, your database may not start or could crash under higher concurrency loads. By understanding and carefully setting these values, you ensure that Fujitsu Enterprise Postgres/PostgreSQL can run reliably, scale effectively, and take full advantage of the hardware resources you've provisioned for your environment.

### 5.1.6. getconf PAGE_SIZE

This command reports the size of a single memory page on the system. On virtually all x86_64 RHEL systems, this returns **4096 bytes (4KB)**. This page size is fundamental to how memory is allocated and managed by the Linux kernel. Knowing the page size is important when interpreting memory metrics, page faults, and database buffer configurations. In Fujitsu Enterprise Postgres/PostgreSQL, buffer management and I/O align with the operating system's page size to optimize performance.

### 5.1.7. vm.dirty_bytes

This parameter sets an absolute limit (in bytes) on how much data can be dirty (modified but not yet written to disk) across the system before the process doing writes is forced to flush data to disk. On many systems this is 0 by default, meaning the system relies instead on `vm.dirty_ratio`. Using `vm.dirty_bytes` is recommended for database servers because it gives predictable thresholds regardless of total RAM. For Fujitsu Enterprise Postgres/PostgreSQL workloads, it is typically set to **600MB (629145600)** to balance caching efficiency with writeback pressure, ensuring dirty data doesn't grow too large and cause sudden I/O spikes.

### 5.1.8. vm.dirty_background_bytes

This sets the threshold (in bytes) at which the kernel's background writeback daemon starts flushing dirty pages to disk in the background. This is the "soft" limit, designed to keep the system responsive. Like `vm.dirty_bytes`, this is often 0 by default, causing `vm.dirty_background_ratio` to apply instead. For Fujitsu Enterprise Postgres/PostgreSQL servers, it is best to explicitly set `vm.dirty_background_bytes` to a lower value than `vm.dirty_bytes`, typically **300MB (314572800)**, so that background flushing starts early, smoothing out disk writes and avoiding large bursts.

### 5.1.9. vm.dirty_ratio

This parameter specifies the maximum percentage of total system memory that can be dirty before the process performing the write is forced to flush data to disk. By default, this is usually **20%**, which can be excessively large on systems with high RAM (*e.g.*, on a 128GB server, it would allow up to 25GB dirty pages). Because of this, on database servers it's better to disable this by setting it to **0** and rely on `vm.dirty_bytes` for tighter, more predictable control.

### 5.1.10. vm.dirty_background_ratio

This parameter defines the percentage of system memory that can be dirty before the kernel's background writeback daemon begins flushing data. It defaults to **10%**, which again is quite large on high-memory systems and could delay the start of background flushing. It's advisable to set this to **0** when using `vm.dirty_background_bytes`, favouring fixed byte-based limits which scale better with modern large-RAM servers.

### 5.1.11. vm.swappiness

This controls the kernel's inclination to move processes out of RAM and onto swap space. The default is typically **60**, which is balanced for general-purpose workloads but too aggressive for database servers. For Fujitsu Enterprise Postgres/PostgreSQL, frequent swapping can severely degrade performance because database buffer caches must stay in memory. Setting this between **1** and **5** minimizes swap usage, instructing the kernel to swap only under significant memory pressure, thereby protecting database performance.

### 5.1.12. vm.zone_reclaim_mode

On NUMA (Non-Uniform Memory Access) systems, this setting controls whether the kernel will prefer to reclaim memory from its local NUMA node before using free memory on other nodes. While this can improve locality for certain workloads, for databases it risks underutilizing available memory across NUMA nodes and can introduce unwanted throttling. Therefore, on Fujitsu Enterprise Postgres/PostgreSQL servers, this should be set to **0**, disabling zone reclaim so the database can freely use memory across all NUMA nodes, ensuring maximum cache effectiveness.

### 5.1.13. vm.overcommit_memory

This governs the kernel's strategy for handling memory allocation requests that might exceed total physical RAM. A value of **0** (the default) lets the kernel make heuristic decisions, which can sometimes approve more allocations than the system can safely support, risking unexpected out-of-memory (OOM) kills under load. A value of **1** allows allocating memory beyond what is physically available without checks, which is unsafe for critical database workloads. The recommended setting for Fujitsu Enterprise Postgres/PostgreSQL is **2**, which tells Linux to enforce a strict allocation policy: memory cannot be overcommitted beyond the threshold determined by vm.overcommit_ratio. This ensures that Fujitsu Enterprise Postgres/PostgreSQL and other processes are only granted memory when there is confidence that sufficient physical memory exists, greatly reducing the chance of abrupt process termination due to OOM.

### 5.1.14. vm.overcommit_ratio

This will work in conjunction with `vm.overcommit_memory` when it is set to **2**, defining the portion of physical RAM that is available for allocation beyond what processes directly reserve. Expressed as a percentage, this ratio guides the kernel in calculating the total memory commit limit. For example, on a server with **64 GB of RAM** and `vm.overcommit_ratio` set to **50**, the system permits memory allocations up to **96 GB** which is computed by adding **50% of the RAM (32 GB)** to any available swap space. This configuration gives Fujitsu Enterprise Postgres/PostgreSQL ample flexibility to aggressively utilize memory, which is especially beneficial for resource-intensive operations such as large sorts, hash joins, and maintenance tasks, while still ensuring that total allocations remain within a predictable boundary. By preventing unrestricted overcommitment, it safeguards the server from abrupt process termination by the OOM killer, striking an optimal balance between maximizing memory utilization and preserving overall system stability.

## 5.2. Device configuration

### 5.2.1. Scheduler

On modern Linux systems, the I/O scheduler plays a crucial role in determining how efficiently the kernel manages read and write requests to disk. Since kernel version 2.6, Linux administrators have had the flexibility to choose different schedulers to optimize their systems based on the type of storage and workload.

For RHEL, which use the multi-queue block layer (`blk-mq`) by default, the recommended schedulers have evolved. For traditional spinning disks (HDDs), it is advisable to use the `mq-deadline` scheduler, which provides predictable latency by prioritizing request deadlines. For systems equipped with **SSDs** or storage devices that have their own sophisticated internal I/O scheduling (such as enterprise-grade controllers), it is generally recommended to use the `none` scheduler. The `none` scheduler effectively bypasses the kernel's reordering logic, which is unnecessary for such devices and can even degrade performance.

This shift away from older defaults like `cfq` (Completely Fair Queuing) is because `cfq` attempts to fairly distribute I/O, which can counterintuitively slow down high-performance SSDs. Always remember that the optimal choice can depend on the workload, so benchmarking different schedulers on your actual system is encouraged, provided this is done during a maintenance window with explicit approval.

To check which scheduler is currently active for a given device (*e.g.*, `nvme0n1` or `sda`), run:

```
cat /sys/block/<device>/queue/scheduler
```

The typical output might look like:

```
[mq-deadline] none
```

where the scheduler in brackets is currently active.

To temporarily change the scheduler, for example to none on an NVMe device, execute:

```
echo none > /sys/block/<device>/queue/scheduler
```

Confirm the change by checking the scheduler again. After benchmarking, you should revert to the original scheduler to ensure consistency.

If benchmarks demonstrate a significant improvement with a different scheduler, you can make the setting persistent by adding a kernel boot parameter in the GRUB configuration, such as:

```
GRUB_CMDLINE_LINUX="elevator=none"
```

followed by running `grub2-mkconfig` to regenerate the configuration. This ensures the chosen scheduler persists across reboots.

### 5.2.2. Read ahead

The read-ahead setting influences how many kilobytes the kernel pre-fetches into the page cache during sequential reads. A larger read-ahead value can improve throughput on workloads that perform large sequential reads, such as database scans. By default, many Linux distributions, including RHEL 8 and 9, use a relatively small value, commonly **128 KB**.

To view the current read-ahead value for a device, use:

```
cat /sys/block/<device>/queue/read_ahead_kb
```

For example:

```
128
```

You can experiment with increasing this value to **4096 KB** (or another appropriate figure based on your workload) by running:

```
echo 4096 > /sys/block/<device>/queue/read_ahead_kb
```

This change is immediate but not persistent across reboots. To make it permanent, include it in `/etc/rc.d/rc.local` or create a custom udev rule.

Always benchmark the impact of this change, and ensure it aligns with the access patterns of your Fujitsu Enterprise Postgres/PostgreSQL or similar database workloads. After testing, return the setting to its original value if not adopting it permanently.

## 5.3. Mount point options in Linux for databases

Mount options at the filesystem level are often overlooked, yet they play a critical role in tuning a Linux environment for high-performance database workloads such as Fujitsu Enterprise Postgres/PostgreSQL. Selecting the right mount options can reduce unnecessary I/O, prolong the life of SSDs, and optimize data reliability in the presence of advanced hardware. Below is a detailed look at some of the most impactful options.

### 5.3.1. noatime

By default, many Linux filesystems track the *last access time* (atime) of each file. This means that every time a file is read even if just queried by the database an additional write is performed to update its access timestamp. In database workloads, this can lead to a massive volume of redundant writes since databases perform frequent reads.

Using the `noatime` mount option tells the kernel not to update the last access time on files when they are read, effectively eliminating these unnecessary write operations. This reduces I/O pressure on the disk subsystem, lowers write amplification on SSDs, and indirectly improves throughput and latency, especially for read-heavy database operations.

For example, mounting a filesystem like this:

```
/dev/sda1 /var/lib/pgsql/data ext4 defaults,noatime 0 2
```

ensures that Fujitsu Enterprise Postgres/PostgreSQL's frequent index scans and sequential reads don't generate hidden writes, preserving performance and extending the lifespan of SSDs.

### 5.3.2. discard

SSDs manage data differently from spinning disks. Over time, as blocks are written and erased, the SSD's internal controller needs to know which blocks are truly unused so it can perform efficient garbage collection and wear leveling. Without this, the SSD may slow down as it runs out of clean blocks to write.

The `discard` mount option enables inline TRIM commands, which tell the SSD when files are deleted, or blocks are released. This helps maintain optimal write performance by ensuring the SSD is always aware of which blocks are safe to erase or reallocate.

However, using `discard` at mount time can introduce slight latency during normal filesystem operations because TRIM happens immediately when files are deleted. For high-performance or latency-sensitive database workloads, it's often better to omit discard from `/etc/fstab` and instead run scheduled `fstrim` jobs (e.g., weekly via cron or systemd timers). This approach batches TRIM operations at off-peak hours, reducing impact on transactional performance while still maintaining SSD health.

### 5.3.3. nobarrier

Modern filesystems like `ext4` and `xfs` implement barriers to ensure that data is physically written to disk in a safe order, protecting against corruption if the system crashes. These barriers enforce that data blocks are written and acknowledged by the hardware before metadata is updated.

The `nobarrier` mount option disables these explicit write barriers. This can improve performance by skipping forced flushes to disk after transactions, reducing latency on commit-heavy workloads like OLTP databases. However, this only makes sense if the underlying storage hardware already guarantees data integrity through its own mechanisms most commonly through battery-backed write caches (BBWC) or non-volatile cache (NVC).

If your storage array or RAID controller has battery-backed cache, then even in the event of a power failure, unwritten data in the cache can be safely written once power returns. In such scenarios, `nobarrier` can safely be used to achieve higher throughput. Conversely, enabling `nobarrier` on standard disks or SSDs without this protection risks data loss or corruption during unexpected power outages.

## 5.4. Redundant Array of Independent Disks (RAID)

For systems storing database data, using some form of RAID is a critical best practice. RAID combines multiple physical disks into a single logical unit to achieve one or more goals: increased performance, higher availability, and data redundancy. This approach protects against disk failures that could otherwise cause catastrophic data loss, while also improving throughput for database workloads that issue many parallel reads and writes.

It is essential, however, to choose the RAID level carefully. Not all RAID types are equally suitable for database systems. The wrong choice can undermine performance or resilience, or both.

### 5.4.1. Less Desirable RAID Levels (Bad) - RAID 5, RAID 6 & RAID 0

**RAID 5** (striping with distributed parity) and **RAID 6** (striping with dual distributed parity) are popular in general-purpose file storage because they maximize usable storage capacity while still tolerating disk failures (one disk for RAID 5, two disks for RAID 6). However, these RAID levels have fundamental downsides for database workloads, especially those with frequent random writes.

The reason lies in how parity is managed. Each time data is written, parity information must be recalculated and written to other disks in the array. This introduces additional I/O overhead known as the read-modify-write penalty. For writes that are small and scattered common in transactional databases this becomes a bottleneck, resulting in high write latency and reduced throughput.

For workloads that primarily perform large sequential reads (such as data warehouses), RAID 5 and RAID 6 can still deliver acceptable performance. Some modern RAID controllers attempt to mitigate the write penalty by using SSD-based non-volatile caches, which absorb writes and later flush them to the slower disks while maintaining parity. Even so, these setups are generally more complex and still may not match the consistency or raw throughput needed for a heavily loaded OLTP database.

**RAID 0** (pure striping) spreads data evenly across multiple disks without any parity or mirroring. This means it offers excellent read and write performance, making full use of all disks in parallel. However, RAID 0 comes with a severe trade-off: there is no redundancy whatsoever. If even a single disk in the array fails, the entire array becomes unrecoverable, resulting in total data loss. This is unacceptable for production database environments where data integrity is paramount. RAID 0 is typically only appropriate for scratch or temporary datasets where data can be easily recreated.

### 5.4.2. Better RAID Options (Good) - RAID 1

**RAID 1** is based on *mirroring*, where identical copies of data are written simultaneously to two or more disks. This means every piece of data exists in two places at once. As a result, RAID 1 provides excellent redundancy: if a single disk fails, no data is lost because the other disk still holds a complete, up-to-date copy.

In addition to redundancy, RAID 1 offers performance benefits. Since any read request can be fulfilled by either of the mirrored disks, the system can distribute read operations across the drives, improving throughput for read-heavy workloads. For write operations, however, each write must occur on both disks, so RAID 1 write performance is roughly equivalent to a single drive.

There are some important considerations. Because data is duplicated, RAID 1 effectively cuts storage capacity in half (two 1TB drives yield only 1TB usable). Additionally, while RAID 1 protects against the failure of a single drive, during the period after a drive fails known as degraded mode the array is vulnerable. If another drive fails before the first one is replaced and fully rebuilt, all data could be lost. For this reason, best practice is to keep hot spare drives on hand, allowing the RAID controller to immediately start rebuilding the array with a new disk, minimizing the window of risk.

RAID 1 is widely used for transactional database systems that prioritize data safety and where capacity demands are modest compared to redundancy requirements.

### 5.4.3. Optimal RAID for Databases (Ideal) - RAID 10 (RAID 1+0)

**RAID 10**, sometimes written as **RAID 1+0**, combines the benefits of both striping (RAID 0) and mirroring (RAID 1) by layering them: data is first mirrored, then striped across multiple mirrored pairs. This design allows RAID 10 to deliver both high performance and robust redundancy, making it especially well-suited for demanding database environments.

From a performance standpoint, RAID 10 inherits the parallel read and write benefits of RAID 0 striping, distributing data across multiple disks so that simultaneous I/O operations can be handled efficiently. Meanwhile, the mirrored structure means that each piece of data is still stored on at least two drives, so any single drive failure does not result in data loss.

The redundancy of RAID 10 is stronger than RAID 5 or RAID 6 in many real-world scenarios, because it does not rely on parity calculations or suffer the write penalties those RAID levels introduce. RAID 10 can tolerate multiple disk failures provided no two failures occur within the same mirrored pair. For example, in a four-disk RAID 10 array, you could lose one disk from each mirrored pair and still maintain full data integrity.

However, RAID 10 does require sacrificing storage capacity for redundancy, just like RAID 1. Half of the total raw disk space is used to store mirrored data. For instance, an eight-disk RAID 10 array with 1TB disks would provide 4TB of usable storage.

Because RAID 10 offers both excellent read and write performance and strong resilience to drive failures, it is often considered the ideal RAID level for high-performance OLTP databases like Fujitsu Enterprise Postgres/PostgreSQL, where maintaining low latency under heavy write loads is critical.

## 5.5. File systems

**File systems**

When designing storage for a Fujitsu Enterprise Postgres/PostgreSQL database, the choice of file system plays a significant role in balancing performance, durability, and operational simplicity. This is particularly true for the transaction log (WAL), which has very specific I/O patterns. The WAL primarily involves sequential writes that ensure data durability and crash recovery. Unlike the main database data files, WAL does not rely on the file system's journaling to protect against corruption. Fujitsu Enterprise Postgres/PostgreSQL's internal mechanisms already safeguard consistency by replaying or ignoring incomplete WAL segments during recovery. As a result, a journaling file system is not required for the transaction log, and minimizing journaling overhead can improve performance.

Linux provides several file system options suitable for Fujitsu Enterprise Postgres/PostgreSQL. Each of these comes with different behaviours and trade-offs that impact WAL performance and reliability.

### 5.5.1. EXT2

EXT2 is the original non-journaling file system from the extended filesystem family. It was designed before journaling became common, which means it avoids the extra writes associated with maintaining a journal. For workloads dominated by sequential writes like the WAL, this is beneficial because it provides the lowest possible write overhead, allowing the system to fully utilize the underlying disk's bandwidth.

To ensure data durability on EXT2, it's common to mount it with the `sync` option, which forces every write to be immediately flushed to disk. This guarantees that data reaches persistent storage without relying on delayed buffering. While this adds latency to everyone write operation, it ensures the WAL's primary goal is that committed transactions are safely recorded on disk. The combination of no journaling and explicit synchronous writes makes EXT2 with sync a straightforward, predictable choice for dedicated WAL storage.

However, EXT2 does lack many of the advanced features of newer file systems, such as extents for managing large files or built-in recovery tools. It is best suited for simple, isolated WAL partitions where maximum sequential write performance is desired.

### 5.5.2. EXT3

EXT3 was developed to extend EXT2 by adding a journal, improving system recovery after crashes or unexpected shutdowns. By default, EXT3 journals both metadata and (optionally) data, reducing the risk of corruption but at the cost of additional writes.

For Fujitsu Enterprise Postgres/PostgreSQL's WAL directories, these extra writes are unnecessary and can degrade performance. Fortunately, EXT3 supports the `data=writeback` mount option, which configures the file system to journal only metadata, not the actual file data. This significantly lowers write amplification while still protecting directory structures and inode consistency after a crash.

This approach provides a middle ground: performance closer to EXT2, but with faster file system checks and metadata recovery thanks to the journal. Using EXT3 in writeback mode is a practical way to minimize I/O overhead for WAL files while retaining a layer of protection that simplifies post-crash recovery of the file system structure itself.

### 5.5.3. EXT4

EXT4 is the modern evolution of the extended filesystem family and the default on most current Linux distributions, including RHEL 8 and 9. It was designed to overcome the scalability and performance limitations of EXT3. EXT4 introduces support for extents, delayed allocation, multiblock allocation, and improved performance on both small and large files.

By default, EXT4 uses the `data=ordered` journaling mode, which ensures that file data is flushed to disk before its metadata is committed to the journal. This mode balances data safety and performance well, making EXT4 a solid choice for general Fujitsu Enterprise Postgres/PostgreSQL data directories.

For the WAL, however, administrators often mount EXT4 with the `data=writeback` option, which journals only metadata. This reduces the extra writes associated with journaling file contents, aligning EXT4's behaviour more closely with EXT3 in writeback mode. At the same time, EXT4 still offers advanced allocation strategies that improve sequential write patterns, reduce fragmentation, and speed up large file writes all directly beneficial to WAL workloads.

Because EXT4 also features faster fsck operations and broad tool support, it's often chosen as a balanced default, combining the performance and minimal journaling needed for WAL with robust long-term stability for database environments.

#### 5.5.4. XFS

XFS is a high-performance, 64-bit journaling file system designed for handling large files and massive parallel I/O workloads. Unlike EXT4 or EXT3, XFS journals metadata by design and does not provide options to disable this. However, since it journals only metadata not data blocks so the impact on sequential write-heavy operations like WAL is minimal.

XFS shines in environments with large files, heavy concurrency, and multiple parallel I/O threads. It features aggressive pre-allocation, delayed logging, and efficient extent-based allocation, which together minimize fragmentation and optimize sequential write throughput. This makes XFS an excellent choice not just for WAL directories but also for main Fujitsu Enterprise Postgres/PostgreSQL data directories, especially in high-throughput OLTP or large OLAP setups.

Additionally, XFS scales well on multi-core systems and across hardware RAID volumes, maintaining consistent performance as system demand grows. Its proven stability and powerful management tools make it a frequent recommendation in enterprise database deployments.

#### 5.5.5. Remote file systems

One critical caution: remote or network file systems such as NFS should never be used to store the WAL or the main database data. Network filesystems introduce latency, inconsistent write acknowledgments, and additional failure scenarios like network partitions or timeouts. These characteristics can severely undermine Fujitsu Enterprise Postgres/PostgreSQL's guarantees around transaction durability and crash recovery. Keeping WAL on locally attached, directly controlled storage is essential for maintaining the reliability and performance of the database.

So, selecting the right file system for each component of your Fujitsu Enterprise Postgres/PostgreSQL deployment ensures that I/O patterns are efficiently handled, reducing system overhead and enhancing overall database performance and resilience.

### 5.6. Disk separation

#### 5.6.1. pg_wal

The `pg_wal` directory in Fujitsu Enterprise Postgres/PostgreSQL (or `pg_xlog` in older versions) contains the WAL files and handles exclusively sequential writes, contrasting sharply with the random I/O patterns typical of the main database heap and indexes. If WAL activity is intense, mixing these sequential writes on the same physical disks as random-access data files leads to I/O contention, where the two different patterns compete for the same disk heads or storage queues. This contention can degrade performance, increasing transaction commit latency and slowing general query response times.

To mitigate this, it is best practice to place the WAL directory on a dedicated disk or disk array. Doing this allows the sequential writes to proceed unhindered, while the main database files continue to use their own I/O channels for mixed read-write operations. When selecting the filesystem for this dedicated WAL storage, the same considerations for minimizing journaling and maximizing sequential throughput apply (as discussed in the File Systems section).

### 5.6.2. pg_stat_tmp

The `pg_stat_tmp` directory stores transient statistical data that Fujitsu Enterprise Postgres/PostgreSQL uses for tracking database activity. These statistics are not critical for long-term persistence and do not need to survive a system reboot. They also occupy very little disk space. Because of these characteristics, `pg_stat_tmp` is an ideal candidate for placement on a RAM disk (`tmpfs`). Hosting it in memory ensures extremely fast I/O for the statistics collector processes without burdening persistent storage with unnecessary tiny writes. This can slightly improve performance and reduce wear on SSDs by absorbing otherwise trivial update operations in volatile memory.

### 5.6.3. WAL archive

While WAL segments themselves can be stored on optimized filesystems that minimize journaling, it is critical to treat WAL archives differently. Archived WAL files are essential for point-in-time recovery in conjunction with base backups. For this reason, WAL archives should be kept on entirely separate storage from both the live database files and the active WAL directory. This protects against scenarios where a single storage failure might simultaneously destroy both the main data and the WAL archive, thereby compromising the ability to perform a recovery. Isolating the WAL archive ensures a higher level of disaster resilience.

### 5.6.4. Temporary files

Fujitsu Enterprise Postgres/PostgreSQL creates temporary files to handle operations that exceed the capacity of memory, such as large sorts or hash aggregations that spill beyond `work_mem`. If these temporary files are written to the same disks as the primary database storage, it introduces additional I/O contention, which can throttle both the query doing the heavy sort and all other concurrent database activity. This is especially impactful in analytics workloads that trigger many large temporary operations.

Fujitsu Enterprise Postgres/PostgreSQL allows administrators to specify dedicated locations for these files using `temp_tablespaces`. By creating and assigning one or more separate tablespaces specifically for temporary files, you can direct this spillover activity to other disks, preserving the performance of the main database volumes. This also helps prevent situations where a flood of large temporary files fills up the primary database storage, risking a forced shutdown due to running out of space.

For environments prone to generating significant temporary data, monitoring is equally important. The `log_temp_files` setting enables logging of temporary file creation events, which can then be analyzed with tools like pgBadger to identify trends and optimize workload patterns.

# 6. Check application SQLs

## 6.1. Why SQL tuning is Important

SQL tuning is a critical discipline in database performance management, directly influencing how efficiently data is retrieved and manipulated. At its core, SQL tuning ensures that database queries execute using the least possible system resources such as CPU, memory, and I/O bandwidth while returning results as quickly as possible. This is vital for maintaining responsiveness in applications, supporting business processes that rely on timely data access, and controlling the operational costs associated with compute and storage infrastructure. Poorly written or unoptimized SQL can strain a database server, leading to high latency, locking contention, excessive disk reads, and even system instability under heavy workloads. Furthermore, as databases grow in volume and concurrent user demand increases, the impact of inefficient SQL compounds, potentially degrading the performance of unrelated queries and entire applications. Effective SQL tuning addresses these challenges by refining queries and ensuring that they leverage indexes, proper joins, and optimized execution plans. This not only enhances the immediate speed of data operations but also contributes to the long-term scalability of systems, allowing them to handle larger datasets and more users without costly hardware upgrades.

## 6.2. Identifying and minimizing sequential scans

One of the most common performance pitfalls in SQL workloads is the unnecessary use of sequential scans (also called table scans). A sequential scan occurs when the database engine reads each row of a table to find records matching a query's WHERE clause. While this approach is entirely appropriate for small tables or queries that must process most of the table's data, it becomes highly inefficient on large datasets when only a small subset of rows is needed. Excessive sequential scanning leads to more disk I/O, increased buffer cache churn, higher CPU usage, and slower query response times. This not only affects the performance of the specific query but also places additional strain on shared resources, potentially slowing down unrelated operations across the database.

Identifying sequential scans typically involves examining the query execution plan. In Fujitsu Enterprise Postgres/PostgreSQL, this is done using the `EXPLAIN` or `EXPLAIN ANALYZE` command, which shows how the query planner intends to (or did) execute the query. If the plan reveals a `Seq Scan` operation on a large table, it is a strong signal that the planner could not find a more efficient index path, possibly due to missing indexes, outdated statistics, or non-selective query conditions.

For example, consider the following Fujitsu Enterprise Postgres/PostgreSQL session:

```
EXPLAIN SELECT * FROM orders WHERE customer_id = 42;
```

If the output is:

```
Seq Scan on orders  (cost=0.00..12500.00 rows=500 width=128)
Filter: (customer_id = 42)
```

this indicates that Fujitsu Enterprise Postgres/PostgreSQL will perform a sequential scan on the orders table, scanning every row to evaluate whether `customer_id = 42`. This can be highly inefficient if the orders table contains millions of rows.

To minimize unnecessary sequential scans, you can add an index that supports this query:

```
CREATE INDEX idx_orders_customer_id ON orders(customer_id);
```

Re-running the `EXPLAIN` should now produce:

```
Index Scan using idx_orders_customer_id on orders  (cost=0.15..8.20 rows=10 width=128)
Index Cond: (customer_id = 42)
```

This demonstrates that Fujitsu Enterprise Postgres/PostgreSQL can now use an Index Scan, dramatically reducing the number of rows it must examine. In real-world scenarios, this improvement translates to much faster query response times and reduced I/O.

Additionally, regularly running ANALYZE to refresh table statistics helps the planner make informed decisions:

```
ANALYZE orders;
```

This updates the statistics about data distribution in the orders table, improving the chances the planner correctly estimates the cost of using an index versus a sequential scan.

By examining execution plans and taking corrective actions, such as creating appropriate indexes and maintaining up-to-date statistics, you can effectively reduce unnecessary sequential scans. This proactive tuning not only optimizes individual query performance but also contributes to overall database stability and scalability.

## 6.3. Avoiding unused indexes

Indexes are vital tools for speeding up data retrieval, but they come with a trade-off: every index consumes disk space and adds overhead to data modification operations such as `INSERT`, `UPDATE`, and `DELETE`. Each time a row is changed, the database must also maintain all related index entries to keep them in sync with the table data. Consequently, having too many indexes especially those that are rarely or never used, which can degrade write performance, increase storage costs, and prolong maintenance operations like `VACUUM` and `REINDEX`.

Avoiding unused indexes is therefore an essential aspect of database optimization. The goal is to ensure that every index serves a meaningful purpose, supporting real queries or enforcing constraints such as uniqueness. Identifying unused indexes generally involves monitoring query execution over time. Fujitsu Enterprise Postgres/PostgreSQL, for example, provides the `pg_stat_user_indexes` system catalog view, which records how often each index is used for index scans. You can run a query such as:

```
SELECT relname AS table, indexrelname AS index,
       pg_size_pretty(pg_relation_size(indexrelid)) AS size
FROM pg_stat_user_indexes
WHERE idx_scan = 0
ORDER BY pg_relation_size(indexrelid) DESC;
```

This SQL will find indexes with an `idx_scan` count of zero. This means that since the last statistics reset, these indexes have not been used to support a query. While a low or zero scan count does not automatically mean the index is unnecessary it might be critical for occasional but performance-sensitive reports or for enforcing constraints it does signal that you should review its purpose. After carefully evaluating application requirements, removing truly redundant or obsolete indexes can streamline write operations, reduce I/O during maintenance tasks, and simplify the overall index landscape.

## 6.4. Detecting duplicate indexes

Duplicate indexes are another common source of inefficiency. A duplicate index arises when two or more indexes exist on the same table with identical or nearly identical definitions. This might happen due to evolving application requirements, oversight during iterative development, or a lack of documentation on existing schema structures. Duplicate indexes provide no benefit to query performance because the query planner will generally only pick one index, but they still incur the full costs associated with index maintenance and storage.

Detecting duplicate indexes is an important database housekeeping task. In Fujitsu Enterprise Postgres/PostgreSQL, this can be done by querying the system catalogs `pg_index` to look for indexes that cover the same columns in the same order and with the same expressions or operator classes.

You can run a query such as:

```
SELECT indrelid::regclass AS table,
       indkey AS column_numbers,
       array_agg(indexrelid::regclass) AS indexes,
       pg_catalog.pg_get_expr(indpred, indrelid, true) AS expression
FROM pg_index
GROUP BY indrelid, indkey, pg_catalog.pg_get_expr(indpred, indrelid, true)
HAVING count(*) > 1;
```

## 6.5. Avoid SELECT *

Using `SELECT *` to retrieve all columns from a table may appear convenient, especially during development, but it can lead to inefficient queries and subtle problems in production. By instructing the database to return every column, `SELECT *` forces it to fetch and transmit more data than often necessary. This means extra I/O to read disk pages that include unused columns, higher memory usage to store larger result sets, and more bandwidth consumed sending unneeded data to the client application. Additionally, `SELECT *` tightly couples queries to the exact schema layout, which can cause unexpected behaviours if new columns are later added to the table.

For example, suppose an application dashboard only needs to show a user's name and email:

```
SELECT * FROM users WHERE id = 42;
```

If the users table has dozens of columns including large JSONB profile settings or high-resolution images these are fetched even though the application ignores them. This increases response times and memory consumption unnecessarily. A better approach is to request exactly what is needed:

```
SELECT name, email FROM users WHERE id = 42;
```

This makes it clear to the database planner that only those columns are needed, allows the engine to potentially use narrower indexes, and protects against surprises when the table schema evolves. By avoiding `SELECT *`, you not only reduce overhead but also create queries that are clearer, more maintainable, and robust over time.

## 6.6. Optimize IN vs EXISTS vs JOIN

SQL provides several ways to relate data across tables, and each has distinct performance implications. The constructs `IN`, `EXISTS`, and `JOIN` often achieve similar logical outcomes but are optimized differently by the database. Take the case of finding customers who have placed orders. Using `IN` might look like:

```
SELECT customer_id, name
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders);
```

This is simple and readable but may become inefficient if the subquery returns thousands of `customer_ids`, requiring the database to hold this list in memory and perform repeated membership checks.

Alternatively, `EXISTS` is typically faster for purely checking existence, especially with correlated subqueries:

```
SELECT customer_id, name
FROM customers c
WHERE EXISTS (
  SELECT 1
  FROM orders o
  WHERE o.customer_id = c.customer_id
);
```

Here, the database stops searching for each customer as soon as it finds a matching order, minimizing work.

If you also need data from both tables, a JOIN is generally preferable:

```
SELECT DISTINCT c.customer_id, c.name, o.order_date
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id;
```

This allows the query planner to choose optimal join algorithms (like hash or merge joins) and to combine data in a single step, often outperforming correlated subqueries. The key is to understand your intent: use `EXISTS` when merely testing for the presence of related rows, `IN` for concise checks against small lists, and `JOIN` when you want to retrieve additional columns from the related table.

## 6.7. Leverage index-only scans

Index-only scans are an advanced performance feature where the database satisfies a query directly from index data without reading the underlying table rows. This is possible when all the columns needed by the query are already stored in the index and when the database's visibility map indicates that these rows don't need additional checks for recent changes. For instance, imagine you have an index:

```
CREATE INDEX idx_orders_customer_date ON orders(customer_id, order_date);
```

If you run:

```
SELECT customer_id, order_date FROM orders WHERE customer_id = 42;
```

the database can use an index-only scan to retrieve both `customer_id` and `order_date` straight from the index, avoiding extra I/O to fetch data blocks from the main orders table. This significantly reduces read overhead, especially on large tables, since index pages are usually much smaller and better clustered for the queried columns.

By contrast, if you queried:

```
SELECT customer_id, order_date, total_amount FROM orders WHERE customer_id = 42;
```

the planner must do an index scan, locating matching `customer_ids` via the index but then fetching each corresponding row from the table to get `total_amount`. This is slower, particularly if it involves many random reads.

Strategically adding frequently queried columns to multi-column indexes known as creating "`covering indexes`" which can unlock index-only scans for critical workloads. However, because larger indexes consume more disk space and slow down writes, this should be carefully planned for high-impact queries that benefit most from bypassing table lookups.

## 6.8. Consider partitioning

Partitioning is a powerful database design technique that involves splitting a large table into smaller, more manageable pieces called partitions, each of which can be treated almost like an independent table. The primary goal of partitioning is to enhance performance and manageability, especially as tables grow to tens or hundreds of millions of rows. By logically dividing data often by ranges of dates, IDs, or other key attributes queries that target a specific slice of data can be executed much more efficiently, as the database only needs to scan relevant partitions instead of the entire table. This is known as partition pruning.

For example, in a transactional system that records daily orders, partitioning the orders table by month ensures that a query seeking orders from June 2025 scans only that month's partition, bypassing data from other months entirely. This drastically reduces I/O and improves query latency. Partitioning also simplifies maintenance tasks such as archiving or purging old data. Dropping an old partition is far quicker and less resource-intensive than running a large `DELETE` statement on a monolithic table.

**When should you consider partitioning?**

- Partitioning is especially beneficial under several common circumstances:
  Large tables: When a single table grows into hundreds of millions or billions of rows, full-table scans and index maintenance become increasingly expensive. Partitioning divides this into smaller physical units, making operations more efficient.
- Time-based data: For workloads such as logs, event histories, or transactional systems that naturally accumulate data over time, partitioning by date (daily, monthly, yearly) allows easy aging out of old data and faster time-based queries.
- Frequent bulk deletes or archives: If your application regularly purges old data, partitioning enables simply dropping an old partition instead of issuing costly row-level deletes.
- Query patterns target subsets: If most queries only need a specific slice of the data (such as one region, customer group, or time period), partitioning ensures only those partitions are accessed.
- Operational management: Partitioning makes it easier to rebuild or reorganize data incrementally, reducing downtime and improving flexibility in managing very large datasets.

However, it's important to design partitioning schemes thoughtfully. Over-partitioning creating too many small partitions also introduce overhead in planning and metadata management, while a poor choice of partition key can lead to uneven data distribution, causing certain partitions to become hot spots. Thus, partitioning should follow a careful analysis of data growth patterns and query access behaviours.

## 6.9. Limiting application connections

One of the most overlooked yet critical aspects of Fujitsu Enterprise Postgres/PostgreSQL performance tuning is managing the number of concurrent client connections. Many developers mistakenly believe that simply opening more connections will increase throughput. However, Fujitsu Enterprise Postgres/PostgreSQL handles each connection by spawning a separate operating system process. Each process consumes memory for session state, shared buffer access, and various internal structures. As the connection count rises, total memory use grows linearly, often leading to considerable RAM demands. Meanwhile, the CPU must continually switch among these processes, amplifying context switching overhead and lock contention on shared resources.

Fujitsu Enterprise Postgres/PostgreSQL is designed to be highly efficient with a moderate number of active connections typically a few dozen to a few hundred, depending on workload and hardware. Beyond this, adding more connections generally results in diminishing returns and can quickly tip into outright performance degradation. For example, if hundreds of idle or short-lived connections are competing for shared buffers or transaction visibility checks, the system may spend more time coordinating these connections than executing queries. A common symptom of excessive connections error:

```
FATAL: sorry, too many clients already
```

which appears when the configured max_connections limit is exceeded. Even before hitting this hard cap, excessive connections increase memory and CPU usage due to increased context switching and lock acquisitions. The result is slower queries and unpredictable response times, hurting both throughput and user experience.

To prevent this, it is best practice to limit direct application connections and instead route them through a connection pooler like `pgBouncer` or `Pgpool-II`. These tools maintain a small, controlled set of active connections to Fujitsu Enterprise Postgres/PostgreSQL, lending them to applications on demand. This drastically cuts memory and CPU overhead, ensuring the database operates within optimal bounds. Additionally, Fujitsu Enterprise Postgres/PostgreSQL's `max_connections` parameter and per-role connection limits provide guardrails, preventing runaway scenarios where one application overwhelms the system.

By carefully managing concurrency through pooling and sensible limits, organizations can keep Fujitsu Enterprise Postgres/PostgreSQL fast, stable, and scalable even under heavy load.

## 6.10. pg_stat_statements

`pg_stat_statements` is one of the most useful PostgreSQL extensions for monitoring and improving database performance. It tracks all SQL statements that run on your database and keeps detailed statistics, such as how many times each query was executed, the total and average time taken, the number of rows returned, and how much data was read from memory or disk. Unlike basic logging, `pg_stat_statements` groups together similar queries ignoring different constant values so you can see the true impact of query patterns, not just individual calls.

This makes it a powerful tool for finding slow or frequently executed queries that might need optimization. For example, you could discover that a simple `SELECT` is being called millions of times every day, or that a monthly report query is causing heavy CPU and I/O load whenever it runs. With this information, you can decide where to add indexes, rewrite inefficient SQL, or introduce caching in your application.

Because `pg_stat_statements` keeps track over time, it also helps you spot performance regressions after deploying new code or shows evidence of improvement after tuning. In short, it turns tuning from guesswork into a data-driven process.

## 6.11. Use views or materialized views for complex queries

Views and materialized views help simplify complex SQL and make both development and performance management easier. A view is like a saved SQL query that you can call by name, which means you can centralize complicated joins, filters, or calculations in one place. This avoids repeating the same long SQL in many parts of your application, making code easier to maintain.

A materialized view goes further by storing the result of a query on disk. Instead of calculating everything fresh each time, the database reads precomputed results, which can be much faster. Materialized views are great for reports and dashboards that analyze large data sets and don't need real-time updates they can be refreshed on a schedule off-peak hour.

Both views and materialized views help the database optimizer understand your data better and break big queries into smaller, more manageable pieces. This can reduce CPU, memory, and I/O load, leading to more stable performance, especially in systems doing heavy analytics.

## 6.12. Equijoins to improve SQL efficiency

Equijoins are joins that match rows using simple equal conditions, like `table1.id = table2.id`. They are a key part of relational databases and are highly optimized by PostgreSQL. With equijoins, the database can use efficient algorithms like `nested loops`, `hash joins`, or `merge joins` and take advantage of indexes to quickly find matching rows. This makes queries run faster and scale better.

By contrast, joins that use inequality conditions (like <, >, or !=) or functions on join columns usually prevent the database from using indexes, forcing it to check every possible row combination. This is much slower, especially on large tables.

To keep queries efficient, design your tables and queries so they use equijoins with matching data types. This allows the planner to pick the fastest join method and minimizes memory and disk use. It's one of the most important habits for writing high-performance SQL.

## 6.13. Avoid implicit type conversion

Implicit type conversion happens when PostgreSQL automatically changes one data type to another so it can compare or join them. For example, comparing a `TEXT` column to a `NUMBER`, or a `VARCHAR` to an `INTEGER`, makes PostgreSQL convert values on the fly. This usually prevents it from using indexes for quick lookups, forcing it instead to do a full table scan, which is much slower.

These silent conversions also use extra CPU because each row needs to be converted before it can be compared. Sometimes they even cause unexpected errors if the conversion fails, like when trying to turn a `non-numeric` string into a `NUMBER`.

To avoid these problems, always make sure you're comparing the same data types. Store data in the right types from the start and explicitly cast values if needed. This helps the planner use indexes properly, speeds up queries, and avoids surprises in your application logic.

## 6.14. Use parallel query scans to improve query performance

PostgreSQL supports parallel query execution, which means it can split certain operations like large table scans, aggregations, and joins across multiple CPU cores at the same time. This is a huge performance boost for large datasets or complex analytical queries that would otherwise take a long time.

When a parallel plan is chosen, PostgreSQL creates multiple worker processes. Each one handles part of the job, like scanning a different section of a table. This cuts down the total query time by spreading the work across the server's cores. Parallelism also works for things like joins and aggregates, combining partial results at the end.

To make use of this feature, you need to adjust settings like `max_parallel_workers_per_gather` and make sure your hardware and workload can benefit. Also, some queries can't be parallelized for example, ones with `LIMIT` or volatile functions.

Using parallel queries can turn slow reports that take minutes into jobs that finish in seconds, fully using modern multi-core CPUs and improving the speed of data-heavy applications.

# 7. Tune database parameters

Tuning database parameters is one of the most critical tasks in achieving reliable, predictable, and high-performing PostgreSQL systems. The default configuration values that ship with PostgreSQL are intentionally conservative, designed to work on a wide range of hardware including small virtual machines or legacy systems. While this ensures safety out of the box, it means these defaults are almost always suboptimal for production workloads, especially on modern servers with large amounts of memory, multi-core CPUs, and fast storage.

Effective tuning involves adjusting the configuration parameters (found in `postgresql.conf`) so that PostgreSQL makes the best use of available hardware resources while matching the nature of your workload.

## 7.1. max_connections

The `max_connections` parameter sets the maximum number of concurrent connections that can be established to the PostgreSQL database. Each client connection consumes server resources because PostgreSQL creates a dedicated backend process for every user session. This backend handles all communication, query execution, and transaction management for that client. Once the user disconnects or logs off, the associated backend process is terminated, freeing up those resources.

However, maintaining many active backend processes simultaneously can significantly increase memory consumption and CPU overhead, due to context switching and process management. To address this, many production environments use connection pooling tools like `PgBouncer` or `Pgpool-II`. These tools reduce the overhead on the main PostgreSQL postmaster process by reusing existing backend processes, serving multiple client requests through a smaller, controlled pool of actual database connections. This keeps resource usage stable and allows PostgreSQL to handle high application loads efficiently.

## 7.2. shared_buffers

The `shared_buffers` parameter defines the amount of memory PostgreSQL sets aside for its own buffer cache, where it stores frequently accessed data pages. Each buffer is typically **8KB** in size, and the total shared buffers are determined by multiplying this size by the number of buffers set. PostgreSQL requires at least 16 buffers and often recommends having at least twice the `max_connections` value to ensure there are enough buffers to handle concurrent operations.

As a rule of thumb, a setting of about **25%** of system RAM provides a good starting point. However, PostgreSQL also relies heavily on the operating system’s file cache, so sometimes it’s beneficial to keep `shared_buffers` relatively modest and let the OS handle most of the caching. Tuning this parameter is critical because too small a setting can cause frequent disk reads, while too large a setting may reduce memory available for the OS cache, possibly hurting overall I/O performance.

## 7.3. work_mem

The `work_mem` parameter specifies the amount of memory allocated for internal operations like sorting and building hash tables before the system falls back to using temporary disk files. This value is set in kilobytes, with a minimum allowed value of **64KB**. Increasing work_mem can significantly improve the speed of operations like `ORDER BY`, `DISTINCT`, and `JOIN` operations that rely on sorting or hashing. It's important to remember that `work_mem` is allocated per sort or hash operation, and multiple such operations can occur simultaneously in a single query. Therefore, very high values could consume excessive memory if many operations run in parallel.

A key advantage of `work_mem` is that it can also be adjusted on a per-session basis, allowing resource-intensive queries to temporarily use more memory without affecting global settings.

## 7.4. maintenance_work_mem

The `maintenance_work_mem` parameter sets the maximum amount of memory PostgreSQL will use for maintenance operations such as `VACUUM`, `CREATE INDEX`, and `ALTER TABLE ADD FOREIGN KEY`. Like `work_mem`, it is specified in kilobytes, with a minimum of **1024KB**. Allocating more memory to this setting allows these operations to handle larger chunks of data in memory, reducing the need to write temporary data to disk and speeding up tasks like indexing or vacuuming. This is especially helpful during database restores or heavy maintenance periods, where minimizing disk I/O can save significant time.

A common recommendation is to set `maintenance_work_mem` to about **1/20th** of total RAM (Total RAM × 0.05). However, this must be balanced with the `autovacuum_max_workers` setting because each autovacuum worker can use up to this amount of memory. Increasing the number of autovacuum workers may require reducing `maintenance_work_mem` to avoid overcommitting system memory. Properly tuning these together ensures efficient maintenance without risking memory exhaustion.

## 7.5. effective_cache_size

The `effective_cache_size` parameter gives PostgreSQL's query planner an estimate of how much memory is available for caching data considering both the operating system's file system cache and PostgreSQL's own `shared_buffers`. Unlike `shared_buffers`, this setting doesn't allocate or reserve any memory; it simply acts as a guideline to help the planner decide whether index scans or sequential scans are more likely to be efficient. A higher `effective_cache_size` value makes the planner more inclined to use index scans, under the assumption that more data is already cached in memory.

A common practice is to set `effective_cache_size` to about **70%** to **75%** of total system RAM. This reflects the typical amount of memory that can be used for caching data files, balancing room for other system processes. For more cautious configurations, setting it closer to **50%** is also reasonable. To fine-tune this parameter, it's helpful to monitor operating system memory usage and see how much RAM is consistently available for caching. By providing an accurate estimate, you enable the planner to make smarter choices that can significantly improve query performance.

## 7.6. wal_buffers

The `wal_buffers` parameter determines how much shared memory PostgreSQL sets aside to temporarily store write-ahead log (WAL) data before it is written to disk. Each buffer is typically **8KB** in size. This memory needs to be sufficient to hold the typical volume of WAL data produced by a batch of transactions. By default, setting `wal_buffers` to -1 enables automatic tuning, which calculates the value as approximately **1/32nd** of `shared_buffers`, capped at **16 MB**. This usually strikes a good balance for most systems.

Because WAL data is flushed to disk on every transaction commit, there is generally little benefit in setting this parameter extremely high. Its main purpose is to avoid very frequent small writes by buffering enough WAL data in memory to allow more efficient I/O. While the default auto-tuned behaviour works well in most cases, some systems have shown improved performance when explicitly setting wal_buffers to around **32 MB**. However, increasing it beyond that typically yields no further advantage. Proper tuning ensures smoother transaction logging and can help reduce I/O overhead under write-intensive workloads.

## 7.7. max_wal_size

The `max_wal_size` parameter defines the maximum size that the write-ahead log (WAL) is allowed to grow between automatic checkpoints. This acts as a soft limit meaning PostgreSQL tries to trigger a checkpoint when this size is reached, but under certain conditions, such as during heavy workloads, failed archive_command operations, or high `wal_keep_size` settings, the WAL size can temporarily exceed this threshold. If no unit is specified, `max_wal_size` is interpreted in megabytes, with a default value of **1 GB**.

Larger values for `max_wal_size` reduce the frequency of checkpoints, which helps minimize I/O spikes caused by flushing dirty pages to disk. However, this also means more WAL must be processed during crash recovery, potentially extending recovery time after a failure. The parameter can only be configured in the postgresql.conf file or set on the server's command line at startup.

A practical way to tune this parameter is to calculate it based on the WAL generation rate and the checkpoint interval. For example, using SQL functions to check WAL positions:

```
postgres=# SELECT pg_current_xlog_insert_location();
 3D/B4020A58
(after waiting for next checkpoint)
postgres=# SELECT pg_current_xlog_insert_location();
 3E/2203E0F8
postgres=# SELECT pg_xlog_location_diff('3E/2203E0F8', '3D/B4020A58');
 1845614240
```

This shows approximately **1.8 GB** of WAL generated between checkpoints. A common practice is to set `max_wal_size` to about three times this amount around **6 GB** in this case which allows room for multiple checkpoints and spread I/O more evenly. This "3×" factor accounts for PostgreSQL's behaviour of overlapping WAL usage across two to three checkpoint cycles.

Proper tuning of `max_wal_size` ensures a balance between reducing checkpoint frequency (improving steady-state performance) and maintaining acceptable crash recovery times.

### 7.8. checkpoint_timeout

`checkpoint_timeout` controls the maximum time, in seconds, between automatic checkpoints. The range is **30** to **3600** seconds, with a default of **300** seconds (5 minutes). Longer intervals mean fewer checkpoints, reducing I/O overhead from writing dirty pages to disk.

However, this also means that crash recovery might take longer, since more WAL must be processed to bring the database back to a consistent state. Balancing this setting is crucial to achieving predictable write performance while keeping recovery times within acceptable limits.

### 7.9. max_parallel_workers_per_gather

`max_parallel_workers_per_gather` sets the maximum number of parallel worker processes that can be used for a single parallel operation, like a sequential scan or aggregation. This parameter enables PostgreSQL to use multiple CPU cores for a single query, potentially speeding up large table scans, joins, and sorts.

Increasing this value can significantly reduce query execution time on multi-core systems but also increases CPU usage. It's essential to find a balance to avoid starving other queries or background tasks.

### 7.10. random_page_cost

The `random_page_cost` parameter tells PostgreSQL's planner how much more expensive it is to perform a random read from disk compared to a sequential read. By default, it is set to **4.0**, which assumes that reading a random page is four times costlier than reading sequential pages. This default reflects older assumptions based on traditional spinning disks, where sequential reads were dramatically faster than random access.

However, for most modern systems especially those using SSDs or systems where the database largely fits in memory this default is typically too high. A high `random_page_cost` discourages the planner from choosing index scans in favour of sequential scans, even when an index could be faster. Lowering this value to around **2.5** or even **1.5** makes the planner more willing to use index scans, improving query performance on workloads where indexed lookups are advantageous. If your entire database fits comfortably in memory, you might even set `random_page_cost` equal to `seq_page_cost`, effectively telling PostgreSQL there is no meaningful penalty for random access.

Tuning `random_page_cost` based on your hardware characteristics using lower values for SSDs or memory-resident databases helps to generate more accurate execution plans and can significantly boost performance for index-heavy queries.

### 7.11. huge_pages

The `huge_pages` parameter controls whether PostgreSQL uses large memory pages at the operating system level. Supported only on Linux, huge pages reduce the number of entries required in the page table, which can decrease CPU overhead for memory management.

Valid settings are `tried` (the default, PostgreSQL will use huge pages if possible), `on` (require them), and `off` (disable them). This can improve performance for databases with very large memory allocations by cutting down on TLB (translation lookaside buffer) misses and improving memory mapping efficiency.

### 7.12. bgwriter_delay

The `bgwriter_delay` parameter sets the interval at which the background writer wakes up to check for dirty buffers that need flushing to disk. By default, this is set to **200** milliseconds, which is adequate for light to moderate workloads. However, on systems experiencing high volumes of write activity such as OLTP environments or large-scale batch processing this delay can result in too many dirty pages building up in memory. To reduce pressure on backend processes and avoid performance spikes, it's advisable to decrease this value. On write-heavy systems, reducing `bgwriter_delay` to around **100** milliseconds can significantly improve buffer turnover. For extremely high-frequency systems where writes are continuous and intense, lowering it even further to 10 milliseconds may be beneficial. It's important not to set it below 10 milliseconds to avoid excessive CPU usage caused by the background writer waking up too frequently.

### 7.13. bgwriter_lru_maxpages

The `bgwriter_lru_maxpages` parameter defines the maximum number of dirty buffers the background writer can write to disk in a single pass. With a default of only **100** buffers, PostgreSQL takes a very cautious approach, which may not keep up with demand in more active environments. If your system performs many inserts or updates, especially across many clients, this default can lead to backend processes being forced to write dirty buffers themselves causing latency. A commonly recommended value in such scenarios is **1000**, which enables the background writer to flush a much larger portion of the buffer pool each round. This helps maintain a healthy pool of clean buffers and improves overall system responsiveness under load.

### 7.14. bgwriter_lru_multiplier

The `bgwriter_lru_multiplier` controls how many buffers are targeted for cleaning in each round of the background writer, relative to the number of recently used buffers. The default multiplier of **2** works well in general-purpose systems. However, in systems where write traffic is high and buffers are reused quickly, this setting may not be aggressive enough. Increasing it to **3** or even **4** allows the background writer to be more proactive in cleaning buffers. This reduces the likelihood that backend processes encounter dirty pages that must be flushed before reuse, thereby improving write throughput and reducing contention under heavy loads.

### 7.15. bgwriter_flush_after

The `bgwriter_flush_after` parameter defines how much data the background writer should accumulate before flushing it to disk. This setting helps reduce the frequency of disk flushes and allows writes to be batched, which is beneficial for performance, particularly on SSDs or write-sensitive storage. By default, this is typically set to **512kB**. On high-performance storage systems, increasing this to **1MB** or more (**not more than 2MB**) can provide better throughput by reducing I/O fragmentation. However, this value should be tuned carefully, as larger values may introduce latency in write persistence if too much data is held in memory before being flushed.

## 7.16. backend_flush_after

Similar to the background writer's flush behaviour, `backend_flush_after` controls how much dirty data a backend process should accumulate before issuing a flush. The default is also around **512kB**. On systems with aggressive write workloads, particularly where many backend processes are running concurrently, increasing this value to **1MB** or more (**not more than 2MB**) can reduce the total number of flush operations and improve I/O efficiency. However, if durability requirements are strict (e.g., in financial systems), the value should not be set too high, to avoid delaying critical data persistence.

## 7.17. checkpoint_flush_after

The `checkpoint_flush_after` parameter governs how much data should be written during a checkpoint before it is flushed to disk. Its default value is typically **256kB**. During a checkpoint, many dirty buffers are written out in bulk, which can cause I/O bursts if not controlled. Raising this value to **1MB** or more (**not more than 2MB**) allows more data to be grouped together before being flushed, smoothing out disk activity and making checkpoints less disruptive. This is especially useful on systems with modern SSDs or RAID arrays where sequential write performance is strong. However, on slower disks, a lower value (default) may still be preferable to avoid long flush times.

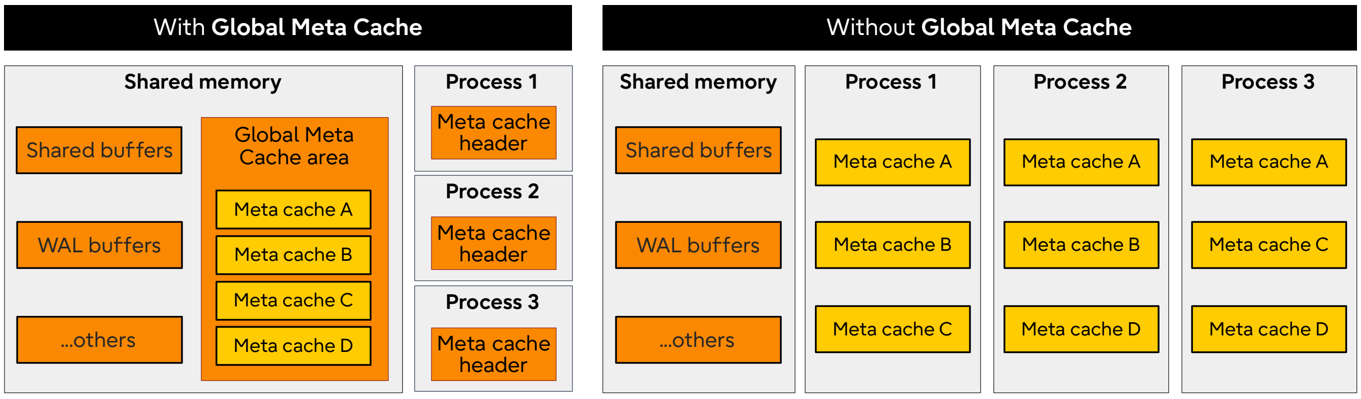
# 8. Fujitsu Enterprise Postgres performance enhancement

Fujitsu Enterprise Postgres is an enterprise-grade distribution of PostgreSQL that includes a range of performance enhancements designed to meet demanding workloads, reduce operational overhead, and ensure predictable, scalable performance in large-scale environments. While it remains fully compatible with the open-source PostgreSQL ecosystem, Fujitsu has added several unique capabilities and optimizations that target some of the most common performance bottlenecks in high-concurrency and mission-critical systems.

## 8.1. Global Meta Cache (GMC)

The Global Meta Cache (GMC) is a distinctive performance-enhancing feature of Fujitsu Enterprise Postgres, designed to significantly reduce metadata lookup overhead across database operations. In a typical PostgreSQL environment, each backend process maintains its own local cache of metadata such as table definitions, index structures, column types, and permissions. While this works well for many scenarios, it can lead to inefficiencies in environments with many concurrent connections. Each process independently loads and manages this metadata, resulting in redundant memory usage and repeated catalog lookups.

The GMC in Fujitsu Enterprise Postgres addresses this by centralizing and sharing metadata information across all database sessions. Instead of every backend process building its own catalog cache, the GMC maintains a globally accessible cache in shared memory. This approach allows all connections to quickly reference a single, consistent set of metadata, eliminating redundant lookups and reducing the CPU and memory cost associated with maintaining many individual caches.



From a performance optimization perspective, GMC delivers two primary benefits:

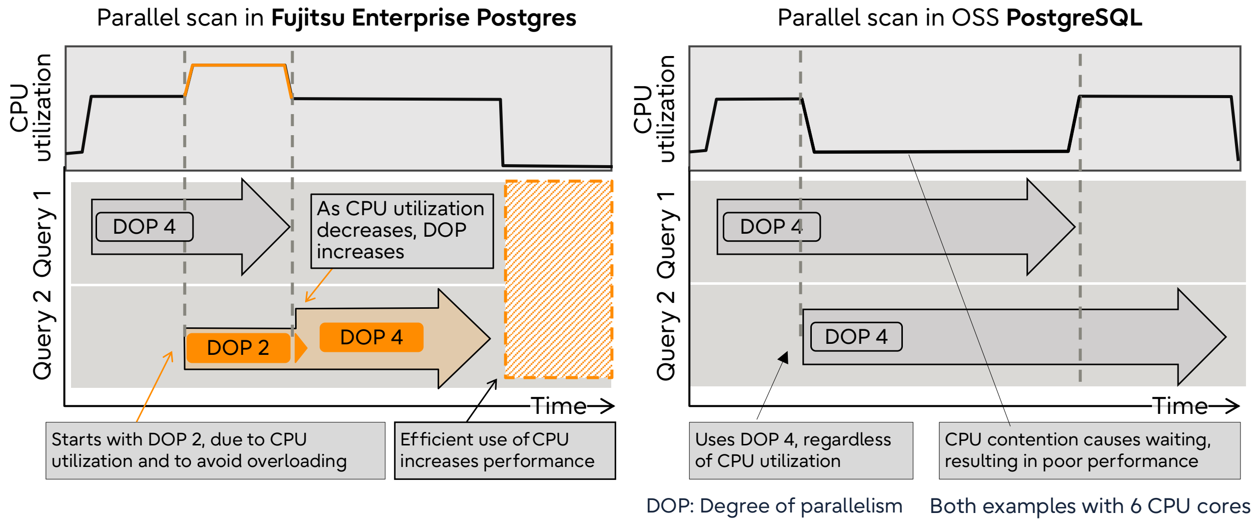
1. **Reduced metadata overhead:** By consolidating metadata management, GMC cuts down on the repeated system catalog scans and cache population that occur in high-concurrency environments. This lowers CPU usage and accelerates operations that involve metadata validation, such as preparing queries or planning execution paths.
2. **Faster query processing:** With a globally shared metadata cache, GMC allows backend processes to start executing queries more quickly, as they can bypass the step of reloading or validating metadata already known to be current and consistent. This is particularly advantageous in workloads with short-lived connections or microservice architectures, where new sessions are frequently initiated.

Additionally, GMC contributes to more predictable performance by reducing fluctuations in latency caused by catalog lookups and local cache refreshes. For systems with thousands of concurrent connections, or where applications frequently open and close sessions, this optimization ensures smoother throughput and makes better use of shared memory resources.

In essence, GMC transforms how metadata is managed in Fujitsu Enterprise Postgres, turning what is traditionally a per-process overhead into a streamlined, shared operation that boosts scalability, lowers resource consumption, and speeds up query processing directly translating into improved overall database performance.

## 8.2. Parallel scan in Fujitsu Enterprise Postgres

Parallel scan is a key feature in Fujitsu Enterprise Postgres that delivers high-speed query processing and stable database operation by making more intelligent use of system resources. While open-source PostgreSQL also supports parallel execution breaking up table scans, joins, and aggregates across multiple CPU cores it does so with a fixed degree of parallelism based solely on configuration parameters. This approach does not take actual system load into account, which can lead to inefficient resource use. For example, if the database is already under heavy CPU load, initiating more parallel workers can cause CPU contention, increasing context switching and slowing down overall performance.


Parallel scan in **Fujitsu Enterprise Postgres**
CPU utilization
Query 1
Query 2
DOP 4
As CPU utilization decreases, DOP increases
DOP 2
DOP 4
Time
Starts with DOP 2, due to CPU utilization and to avoid overloading
Efficient use of CPU increases performance
Parallel scan in OSS **PostgreSQL**
CPU utilization
Query 1
Query 2
DOP 4
DOP 4
Time
Uses DOP 4, regardless of CPU utilization
CPU contention causes waiting, resulting in poor performance
DOP: Degree of parallelism
Both examples with 6 CPU cores

Fujitsu Enterprise Postgres enhances this by adding dynamic control over parallel execution. In addition to supporting standard parallel query plans, Fujitsu Enterprise Postgres monitors current CPU utilization and adjusts the degree of parallelism on the fly. When CPU usage is high, Fujitsu Enterprise Postgres automatically reduces the number of parallel workers to prevent oversaturation of processor resources, ensuring that other critical workloads are not starved of CPU time. Conversely, when CPU load is low, Fujitsu Enterprise Postgres increases the degree of parallelism, maximizing the use of available cores to complete queries faster.

This dynamic tuning provides several practical benefits. It helps maintain consistent and stable performance, avoiding the unpredictable slowdowns that can occur when too many processes compete for limited CPU capacity. It also ensures efficient batch processing and scheduled aggregations, as the system intelligently ramps up parallelism during quieter periods, completing large analytical or maintenance tasks more quickly.

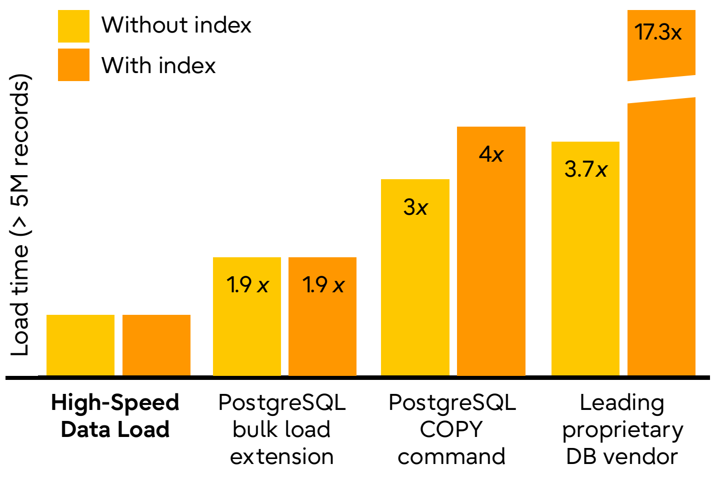
Overall, by integrating CPU-aware parallel control, Fujitsu Enterprise Postgres goes beyond traditional parallel execution models. It achieves faster processing while preventing resource contention, making it well-suited for mixed workloads where both transactional and reporting operations must coexist without degrading each other's performance.

## 8.3. High-Speed Data Load in Fujitsu Enterprise Postgres

High-Speed Data Load is a specialized feature in Fujitsu Enterprise Postgres designed to accelerate bulk data loading by intelligently leveraging multi-core systems. Unlike traditional bulk loading processes, which often rely on a single thread or require extensive manual tuning to parallelize, this feature automatically distributes the workload across multiple CPU cores to maximize throughput without requiring pre-configuration.

Fujitsu's High-Speed Data Load enhances the standard PostgreSQL COPY command by executing it in parallel. It dynamically launches as many parallel workers as the available CPU cores and system load will allow. Each worker simultaneously performs data conversion, table population, and index creation, all directly from the input file. This means the data loading pipeline is fully parallelized, dramatically reducing total load times compared to single-threaded approaches.

For mission-critical environments where large datasets need to be ingested quickly such as during ETL processes, database migrations, or initial population of new systems this feature ensures that bulk data operations make optimal use of available hardware. By automatically scaling the number of parallel workers according to system resources, it avoids the need for DBAs to manually fine-tune load parameters in advance.



This approach not only speeds up data loading but also maintains balanced system performance by adapting to the current CPU capacity. As a result, Fujitsu Enterprise Postgres's High-Speed Data Load is particularly well-suited for modern multi-core servers, enabling enterprises to ingest large volumes of data rapidly while preserving headroom for other critical database operations.

## 9. Best practices for inserting large amounts of data

Efficiently inserting large volumes of data into PostgreSQL or Fujitsu Enterprise Postgres requires careful attention to transaction handling, write-ahead logging (WAL), indexing, and background processes. Following best practices during bulk data operations can dramatically reduce load times, minimize I/O overhead, and ensure a smooth ingestion process without compromising system stability.

- When using multiple INSERT statements, always wrap them within an explicit transaction using BEGIN and COMMIT. This reduces transaction overhead, as each individual INSERT no longer needs to commit separately. However, for large-scale bulk loading, it is far more efficient to use the COPY command, which loads all rows in a single operation. COPY bypasses many of the overheads associated with row-by-row inserts and is optimized for speed.
- If the COPY command cannot be used, due to application constraints it can still help to use prepared statements. By preparing the INSERT statement once and executing it multiple times with different values, you reduce the cost of repeated parsing and planning, which improves performance for high-volume inserts.
- For newly created tables, the most optimal strategy is to first create the table, then bulk load data using COPY, and only afterward create any indexes. This avoids the overhead of maintaining indexes during the data load. Fujitsu Enterprise Postgres goes a step further with its pgx_loader utility, which is specifically designed for high-performance data ingestion and has been benchmarked to be 3 to 4 times faster than the standard COPY command.
- To further accelerate data loading, consider temporarily dropping foreign key constraints and re-adding them after the load is complete. This eliminates the costly validation of foreign keys on each row insert. Similarly, increasing configuration parameters such as maintenance_work_mem and max_wal_size during the load phase can improve performance by allowing more in-memory processing and reducing the frequency of checkpoints.
- For bulk loads that don't require immediate durability or replication consistency, consider disabling WAL archiving and streaming replication during the operation. This avoids the extra I/O overhead of transmitting and storing WAL data. Additionally, disabling triggers and autovacuum processes if safe to do so can further minimize interference during the load.

By combining these practices, database administrators can ensure that large data loads are completed efficiently, with minimal impact on overall system performance.

## 10. Conclusion

Tuning Fujitsu Enterprise Postgres or PostgreSQL is not about changing a few configuration settings and hoping for better performance. It's a structured, end-to-end process that starts with understanding your system's hardware and operating environment, moves through optimizing application SQL, and finally fine-tunes the database engine itself.

This guide has shown that real performance comes from addressing the right layer at the right time. Before adjusting any database parameters, it's critical to ensure the operating system is healthy and your application queries are efficient. Only then will changes to memory allocation, WAL behaviour, autovacuum settings, or parallelism truly deliver lasting improvements.

By following the best practices and step-by-step approach outlined in this guide, you'll not only boost performance you'll also build a PostgreSQL or Fujitsu Enterprise Postgres environment that is stable, scalable, and ready to meet future demands. With the right tools, careful observation, and a clear methodology, performance tuning becomes a manageable and impactful part of database administration.

**Fujitsu Enterprise Postgres** is the enhanced version of PostgreSQL, for enterprises seeking a more robust, secure, and fully supported edition for business-critical applications

**Contact**
Fujitsu Limited
Email: enterprisepostgresql@fujitsu.com
Website: fast.fujitsu.com

2025-08-08 WW EN