

Get unlimited access to the best of Medium for less than \$1/week. [Become a member](#)



PostgreSQL 17 Database Administration: Mastering max_connections and Connection Management

14 min read · Jun 13, 2025



Jeyaram Ayyalusamy

Following



Listen



Share



More



PostgreSQL is one of the most advanced open-source relational database systems, but many DBAs overlook one simple parameter that can make or break

performance: `max_connections`. Managing connections properly is essential for scaling PostgreSQL safely, especially in PostgreSQL 17 where workloads are heavier and systems are often more demanding.

In this post, we'll deep-dive into how PostgreSQL handles connections, potential pitfalls, and practical steps you can take to optimize your environment.

1 What is `max_connections` in PostgreSQL?

When managing PostgreSQL at scale or under heavy traffic, understanding connection limits is crucial for performance and stability. One of the most fundamental parameters that controls this is `max_connections`.

Definition

At its core, `max_connections` defines the **maximum number of concurrent client connections** that PostgreSQL will accept at any given time. These connections include:

- Application users
- Database administrators
- Replication processes
- Background jobs
- Monitoring tools

This setting is essential for controlling **how many active sessions** can interact with the PostgreSQL server simultaneously.

Key Details

- **Default Value:** 100

This is often sufficient for small to medium-sized applications, but too low for high-concurrency or multi-tenant systems.

- **Includes All Connection Types:**

The count includes **regular users**, **superusers**, **background workers**, and **replication roles** — so it's not just about the number of app users.

- **Startup Parameter Only:**

`max_connections` can **only be changed in the `postgresql.conf` file** or via the command line at server startup. You **must restart** the PostgreSQL instance for the change to take effect.



Important Note: More Connections ≠ Better Performance

It might be tempting to simply raise `max_connections` when facing connection limits —but be cautious. Doing so impacts several key system resources:

- **Memory Usage:**

PostgreSQL reserves a portion of memory for each connection. Increasing the limit too much can exhaust server memory, leading to swapping or crashes.

- **CPU Load:**

More connections often mean more parallel queries, which can saturate CPU cores — especially on OLTP systems.

- **I/O Pressure:**

Simultaneous disk reads/writes from many sessions can strain storage systems, slowing down the entire database.

In many cases, it's better to introduce a **connection pooler** (like PgBouncer) instead of drastically increasing `max_connections`.



Best Practices

- **Monitor:** Use `pg_stat_activity` to check how many connections are active and idle at peak times.
- **Tune Carefully:** Set `max_connections` based on real application demand and available system resources.
- **Use Pooling:** Implement connection pooling for web applications and microservices to reduce idle connections.
- **Scale Horizontally:** If needed, consider scaling out your architecture rather than pushing a single server too far.

By understanding how `max_connections` works and the implications of increasing it, you can ensure your PostgreSQL environment remains stable, performant, and resource-efficient.

2 Understanding `superuser_reserved_connections` in PostgreSQL

In PostgreSQL, connection management is not just about how many users can access the database — it's also about **ensuring availability for critical operations**. That's where `superuser_reserved_connections` comes in.

This lesser-known but vital configuration parameter plays a key role in maintaining database accessibility, especially during peak load or connection overload scenarios.

🔒 What Is `superuser_reserved_connections`?

`superuser_reserved_connections` defines the number of **connection slots reserved exclusively for superusers**—typically database administrators (DBAs).

These reserved slots act as a **safety net**, ensuring that superusers can always access the database, even if all other client connections are maxed out.

Key Characteristics

- Default Value: 3

PostgreSQL sets aside three connections for superusers by default. You can increase or decrease this value in your `postgresql.conf`.

- Counts Toward `max_connections`:

These reserved connections are **not in addition to** the `max_connections` setting—they’re **included within it**.

- For example:

- If `max_connections = 100`

- And `superuser_reserved_connections = 3`

► Then only **97 connections** are available for regular (non-superuser) clients.

- Applies Only to Superusers:

These slots can **only be used by users with superuser privileges**. Regular users will receive a “connection limit exceeded” error if all non-reserved slots are in use.

Why It Matters

In production environments, especially those with high traffic, there’s always a risk of **hitting the connection ceiling**. When that happens:

- Application clients are blocked.
- Monitoring tools can’t connect.
- And DBAs **may be locked out**, unable to even investigate the problem.

Thanks to `superuser_reserved_connections`, **DBAs retain access** even when the system is under pressure. This enables them to:

- Kill idle or runaway sessions

- Investigate blocking or contention
- Perform emergency maintenance or restarts

[Open in app ↗](#)

Medium



Search



connections are exhausted.

💡 Best Practices

- **Leave the Default or Increase for High-Risk Systems:**
Keep at least 3–5 reserved connections in environments prone to high concurrency or connection spikes.
- **Combine with Connection Pooling:**
Use a tool like PgBouncer to help manage non-superuser connections efficiently and reduce the risk of maxing out connections.
- **Monitor Usage Regularly:**
Query `pg_stat_activity` to understand which sessions are active, and monitor superuser activity separately.

📌 Final Thoughts

While `max_connections` gets most of the attention, `superuser_reserved_connections` is a critical safety valve for PostgreSQL database administrators. By ensuring you never lock yourself out of your own database, this small setting can be the difference between a smooth recovery and a total outage.

3 Why You Should Be Cautious About High `max_connections` in PostgreSQL

When it comes to PostgreSQL performance tuning, many new administrators are tempted to increase the `max_connections` setting to support more users. After all, more connections must mean more scalability, right?

Not exactly.

While it might seem harmless to allow hundreds — or even thousands — of concurrent database connections, doing so without the right infrastructure and strategy can lead to **severe performance issues**. Let's explore why.



CPU Overhead: The Context Switch Problem

PostgreSQL follows a **process-per-connection model**. This means every new connection spawns a separate OS-level process. When the number of active sessions grows, so does the overhead on the CPU.

Every process switch requires the operating system to **save the current process state and load another**. This activity is known as a **context switch**.

Why It Matters:

- Context switching increases CPU workload.
- On systems with limited CPU cores, too many concurrent processes can overwhelm the processor.
- The database spends more time **managing processes** than actually executing queries.

This leads to **slow response times**, even when individual queries aren't demanding.



I/O Bottlenecks: Too Many Requests, Too Little Bandwidth

Each client connection is capable of issuing **read and write operations** — often simultaneously. When hundreds of sessions hit the storage layer concurrently:

- Disk I/O queues grow

- Latency increases
- Cache hit ratios drop
- Checkpoint operations become more expensive

Even high-performance SSDs or SANs can become bottlenecks under heavy concurrent I/O. And for traditional spinning disks, the performance drop can be dramatic.

Session Delays: Resource Contention and Memory Pressure

More active sessions mean more processes competing for:

- CPU time
- Memory (shared_buffers, work_mem, etc.)
- Locks and semaphores
- Cache access

This leads to **performance degradation**, especially when:

- Queries start waiting for locks held by other sessions
- System memory starts swapping
- Parallel query execution stalls due to unavailable worker processes

Essentially, your system ends up spending more time juggling requests than processing them efficiently.

Better Alternatives Than Raising `max_connections`

- Use a connection pooler (e.g., PgBouncer or Pgpool-II) to manage idle connections more efficiently.

- **Scale horizontally:** Break workloads across read replicas or use application-level sharding.
- **Analyze application behavior:** Fix client-side connection leaks or unoptimized usage of long-lived connections.

Final Thoughts

Raising `max_connections` seems like a quick fix, but it often masks deeper scalability issues. Without proper planning, high connection counts can **cripple performance** instead of improving it.

Always consider **the impact on CPU, I/O, and memory**, and look toward smarter connection management strategies to maintain a fast, resilient PostgreSQL system.

How PostgreSQL Allocates Resources for Connections

PostgreSQL is known for its robustness and extensibility — but when it comes to connection handling, **every active session consumes system resources**. Understanding how these resources are allocated is critical for database administrators and DevOps engineers who want to fine-tune performance.

Let's break down how PostgreSQL handles resource allocation for each connection and what trade-offs you need to consider.

What Resources Does a Connection Use?

Every active database connection in PostgreSQL consumes three key types of memory:

1. OS-Level Memory (RAM)

Each connection creates a **dedicated PostgreSQL backend process** at the operating system level. This process consumes memory for the session itself, including stack space and other kernel resources. More connections = more system memory usage.

2. PostgreSQL Shared Buffers

Shared buffers (`shared_buffers`) act as a shared memory cache where PostgreSQL stores data blocks read from disk. Every session interacts with this shared memory pool to read/write data. With more connections, **contention increases**, and shared buffers may become a bottleneck unless tuned correctly.

3. Query-Specific Memory (`work_mem`)

When a query performs operations like sorting, hashing, or joins, it uses per-query memory called `work_mem`. This is **allocated for every operation, per connection**—and potentially multiple times within a single query. If many users are running complex queries simultaneously, total memory consumption can spike quickly.



Rule of Thumb

The more connections you allow, the less memory is available per connection.

This is a crucial design consideration. Simply increasing `max_connections` without adjusting memory parameters can lead to resource exhaustion, poor query performance, or even out-of-memory crashes.

Example Scenario: Tuning Memory Parameters

Let's look at how some key parameters interact when managing connections:

Parameter Impact `max_connections` Higher values mean more concurrent sessions, increasing total memory usage `work_mem` Needs to be carefully tuned if many users run memory-intensive queries `shared_buffers` May need to be increased to accommodate the extra load from more sessions

For example, if you set:

- `max_connections = 500`
- `work_mem = 8MB`

That's potentially **4GB of query memory** in use *per query operation*, not counting

overhead or other processes.

If you're not careful, this can easily overwhelm your server's RAM and crash the database.

❖ Best Practices

- Don't set `max_connections` too high unless absolutely needed. Use connection pooling (e.g., PgBouncer) instead.
- Balance `work_mem` according to expected concurrency and workload complexity.
- Monitor shared buffer hit ratios using tools like `pg_stat_database` or external monitoring platforms.
- Always benchmark memory settings in a staging environment before rolling out to production.

📌 Final Thoughts

PostgreSQL gives you fine-grained control over memory and connection behavior — but with great power comes great responsibility. Each connection consumes memory in multiple layers, and a poorly tuned system can become unstable under load.

Instead of throwing more connections at a performance problem, take the time to understand how memory is allocated and shared — and tune accordingly.

5 Connection Pooling in PostgreSQL: The Secret Weapon for Performance and Stability

One of the most common performance bottlenecks in PostgreSQL isn't the database engine itself — but how applications connect to it. Many developers and DBAs

instinctively try to solve connection issues by increasing the `max_connections` setting. But in most cases, there's a far better solution:

Connection Pooling.

Let's dive into what it is, why it works, and how to implement it effectively in your PostgreSQL environment.

What Is Connection Pooling?

A **connection pooler** is a middleware layer that manages a pool of reusable database connections. Instead of opening a new PostgreSQL connection for every client or request, the pooler:

- Maintains a fixed number of open connections to the database
- Reuses them across many application requests
- Reduces the overhead of frequent connection creation and teardown

This helps keep PostgreSQL resource usage low and efficient — even when thousands of users are hitting your application.

Two Types of Connection Poolers

1. Application-Level Poolers

Most modern web frameworks and ORMs (Object-Relational Mappers) offer built-in connection pooling support. These are implemented **within the application layer**, and examples include:

- SQLAlchemy (Python)
- Hibernate (Java)
- ActiveRecord (Rails)
- Node.js PG-pool (JavaScript)

These are easy to configure and can often be enabled with just a few lines of code.

2. 💬 PostgreSQL-Native Poolers

These tools sit between your app and the PostgreSQL server, acting as centralized connection managers. Two popular options:

⚡ PgBouncer

- Lightweight and extremely fast
- Ideal for handling large numbers of short-lived connections
- Recommended for most PostgreSQL deployments

💼 Pgpool-II

- More feature-rich
- Supports connection pooling, load balancing, failover, query caching, and replication
- Better suited for complex, high-availability environments

🎯 The Real Goal of Pooling

Keep actual PostgreSQL connections low, while allowing your application to scale to many users.

This gives you the best of both worlds:

- Efficient resource usage on the database server
- Scalable performance on the application side

With pooling, you can:

- Serve thousands of concurrent users
- Avoid hitting `max_connections`

- Reduce memory and CPU pressure on PostgreSQL
- Improve app response times during peak loads

Best Practices

- Always use PgBouncer or similar for production-grade applications
- Set pooling mode to `transaction` for best performance in stateless applications
- Monitor connection usage and fine-tune pool sizes based on traffic patterns
- Combine with application-level pooling when needed for layered efficiency

Final Thoughts

Raising `max_connections` might seem like the easy fix—but it's rarely the right one. Connection pooling is the **smarter, more scalable solution**. Whether you use lightweight tools like PgBouncer or feature-rich solutions like Pgpool-II, you'll dramatically improve your PostgreSQL performance, especially under high concurrency.

6 Pre-checks Before Increasing `max_connections` in PostgreSQL

It's tempting to resolve “too many connections” errors in PostgreSQL by simply increasing the `max_connections` setting. But doing so without proper preparation can quickly backfire—leading to system instability, crashes, or performance degradation.

Before you touch that number, there are a few **critical pre-checks** you should always perform. Let's walk through them step by step.

1. Ensure Your Application Isn't Leaking Connections

Before increasing `max_connections`, ask yourself: *Does my application actually need more connections—or is it just not closing them properly?*

A **connection leak** happens when the application opens database connections but doesn't close or reuse them. Over time, this saturates the available connections, even if the workload is light.

What to Do:

- Use your application's logs or database monitoring tools to identify long-lived or idle connections.
- Inspect the PostgreSQL `pg_stat_activity` view:

```
SELECT pid, username, application_name, state, backend_start FROM pg_stat_activ
```

- Fix connection lifecycle management in the application or ORM.

Leaking connections is like leaving water taps running — you don't solve it by opening more pipes.

2. Implement Connection Pooling

As discussed in earlier sections, a **connection pooler** (such as PgBouncer) allows you to efficiently reuse a limited number of database connections across many client requests.

Rule of thumb: If you're not using connection pooling, you probably don't need to raise `max_connections` —you need to fix your architecture.

Benefits:

- Reduced resource usage
- Improved query throughput

- Better control over client behavior

Only after implementing pooling should you consider raising `max_connections`, and even then—likely by less than you expected.

3. Review OS Kernel Parameters

Raising `max_connections` directly affects how much shared memory PostgreSQL uses. That means the **operating system must be able to support it**.

Key Linux Parameters to Check:

- `shmmmax` : Maximum size of a single shared memory segment
- `shmmax` : Total amount of shared memory available
- **Semaphores**: PostgreSQL uses semaphores for process coordination

Use the following to inspect:

```
cat /proc/sys/kernel/shmmmax  
cat /proc/sys/kernel/shmmax  
ipcs -l
```

If these values are too low, PostgreSQL may fail to start after changing `max_connections`. Always ensure the kernel is properly tuned before applying memory-heavy configuration changes.

4. Use Tools Like `pgtune` to Guide Proper Sizing

Rather than guessing, you can use tools like `pgtune` to help calculate appropriate settings based on your system's hardware and workload.

Why Use `pgtune` ?

- It considers RAM, CPU, and connection counts

- Adjusts related parameters like `shared_buffers`, `work_mem`, `max_worker_processes`, etc.
- Helps prevent misconfiguration that could overload your system

These tuning tools give you a baseline configuration, which you can refine through monitoring and load testing.

Final Thoughts

Increasing `max_connections` isn't just a one-line change in `postgresql.conf` —it's a **system-level decision** with far-reaching implications. Before raising the number, you must:

- Fix the real root causes (like connection leaks)
- Implement efficient connection pooling
- Ensure your OS can handle the extra load
- Rely on intelligent tuning, not guesswork

Only then can you safely scale your PostgreSQL deployment to meet growing demand — without compromising performance or stability.

How to Safely Increase `max_connections` in PostgreSQL

Raising the `max_connections` setting in PostgreSQL can help support more concurrent users—but it's not just a matter of changing a number. PostgreSQL relies on shared memory, and your operating system must be properly configured to support the extra load.

In this guide, we'll walk through the **safe and correct way to increase `max_connections`**, including kernel and PostgreSQL configuration steps.

✓ Step 1: Verify Current Kernel Shared Memory

PostgreSQL uses shared memory (`shared_buffers`) to cache data pages. When you increase `max_connections`, you may also need to increase `shared_buffers`, which in turn depends on your operating system's **shared memory limit** (`shmmmax`).

To check your current `shmmmax` value:

```
cat /proc/sys/kernel/shmmmax
```

This outputs the maximum size (in bytes) of a single shared memory segment the Linux kernel will allow. If this value is too low, PostgreSQL may fail to start after increasing memory settings.

✓ Step 2: Update Kernel Settings (if required)

If your current `shmmmax` is insufficient (e.g., below 1–2 GB), you need to increase it.

Temporary Change (until reboot):

```
sudo sysctl -w kernel.shmmmax=10066329600
```

This sets the maximum shared memory segment size to 10 GB.

Persistent Change (survives reboots):

```
echo "kernel.shmmmax=10066329600" | sudo tee -a /etc/sysctl.conf  
sudo sysctl -p
```

You can also adjust other memory parameters here, such as `shmall`, depending on your system usage.

✓ Step 3: Modify PostgreSQL Parameters

Now that your OS supports the required memory, it's time to update PostgreSQL's configuration file:

Open `postgresql.conf` (typically located in `/etc/postgresql/<version>/main/` or `$PGDATA`):

```
sudo nano /etc/postgresql/14/main/postgresql.conf
```

Update the following parameters:

```
max_connections = 200
shared_buffers = 256MB    # Adjust based on total system RAM
```

 **Tip:** For high `max_connections` values, consider increasing `work_mem`, `max_locks_per_transaction`, and other memory-related settings too.

✓ Step 4: Restart PostgreSQL

For the changes to take effect, restart the PostgreSQL service:

```
sudo systemctl restart postgresql
```

You can confirm the new settings by connecting to the database and running:

```
SHOW max_connections;
```

```
SHOW shared_buffers;
```

💡 Final Thoughts

Increasing `max_connections` in PostgreSQL is not just a database tweak—it's a **system-wide memory adjustment** that must be done carefully. Following the steps above ensures:

- Your operating system can handle the memory PostgreSQL needs
- The database is tuned for stability
- You avoid crashes, startup failures, or degraded performance

Always test these changes in a staging environment before applying them in production, and combine this with connection pooling for best results.

8 Monitoring Connections in PostgreSQL: Stay Ahead of Limits

One of the simplest yet most powerful ways to keep your PostgreSQL database running smoothly is by **actively monitoring connection usage**. Running out of available connections can lead to application errors, locked-out DBAs, and stalled services — especially if you're unaware that you're nearing your limits.

In this post, we'll go over essential SQL queries that help you **track, analyze, and prevent connection-related bottlenecks**.

📊 Why Monitor Connections?

PostgreSQL enforces a strict limit on the number of simultaneous client connections via the `max_connections` setting. Exceeding this limit causes new connection attempts to fail with errors like:

```
FATAL: remaining connection slots are reserved for non-replication superuser connections
```

To prevent this, you need visibility into how many connections are in use — and how close you are to the ceiling.

Essential Queries for Connection Monitoring

Here are the three key metrics every PostgreSQL admin should monitor regularly:

1. Total Active Connections

To get the current number of connections (both idle and active):

```
SELECT count(*) AS active_connections  
FROM pg_stat_activity;
```

This query gives you a snapshot of how many sessions are currently connected to your database. If this number frequently approaches your `max_connections`, it's time to investigate further.

2. Superuser Reserved Slots

PostgreSQL reserves a few connection slots specifically for superusers, which are critical for emergency access.

Check how many are reserved:

```
SELECT setting::int AS superuser_reserved  
FROM pg_settings
```

```
WHERE name = 'superuser_reserved_connections';
```

Note: These slots are included within the `max_connections` limit. So, if `max_connections = 100` and `superuser_reserved_connections = 3`, only **97** slots are available to regular users.

3. Maximum Allowed Connections

To confirm your configured connection ceiling:

```
SELECT setting::int AS max_connections
FROM pg_settings
WHERE name = 'max_connections';
```

Knowing this value allows you to gauge your connection usage percentage:

```
-- Optional: Quick usage percentage calculation
SELECT
    count(*) * 100.0 / (SELECT setting::int FROM pg_settings WHERE name = 'max_connections') AS connection_usage_percent
FROM pg_stat_activity;
```

Best Practices

- Automate alerts using monitoring tools like **Prometheus + Grafana**, **pgMonitor**, or **Zabbix**.
- Set thresholds (e.g., alert if usage > 80%).
- Use connection pooling to reduce idle connections.

- Investigate idle or long-running sessions that may unnecessarily hold connections.

🛡️ Final Thoughts

Proactive connection monitoring helps you avoid one of the most frustrating database issues: **hitting connection limits at the worst possible time**. By regularly running a few simple queries — or integrating them into your monitoring stack — you gain visibility and control over one of PostgreSQL's most important performance levers.

👉 If you found this guide helpful, follow me([medium](#)) for more practical PostgreSQL tutorials, database architecture guides, and hands-on DBA content.

🔗 Let's Connect!

If you enjoyed this post or would like to connect professionally, feel free to reach out to me on LinkedIn:

👉 [Jeyaram Ayyalusamy](#)

I regularly share content on **PostgreSQL, database administration, cloud technologies, and data engineering**. Always happy to connect, collaborate, and discuss ideas!

Postgresql

Oracle

Open Source

MySQL

Data

J

Following ▾

Written by Jeyaram Ayyalusamy 

50 followers · 2 following

Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

Responses (2)



Gvadakte

What are your thoughts?



Jeyaram Ayyalusamy Author

Jun 19

...

I really appreciate you following my blogs and sharing your suggestions. Thanks for the update. I'm still working on it. Initially, I planned to update all the blog posts before starting the practical part. I've already written over 150 blogs, so I want to update all of them before beginning the configuration work.



[Reply](#)



R Ragavendran

Jun 18

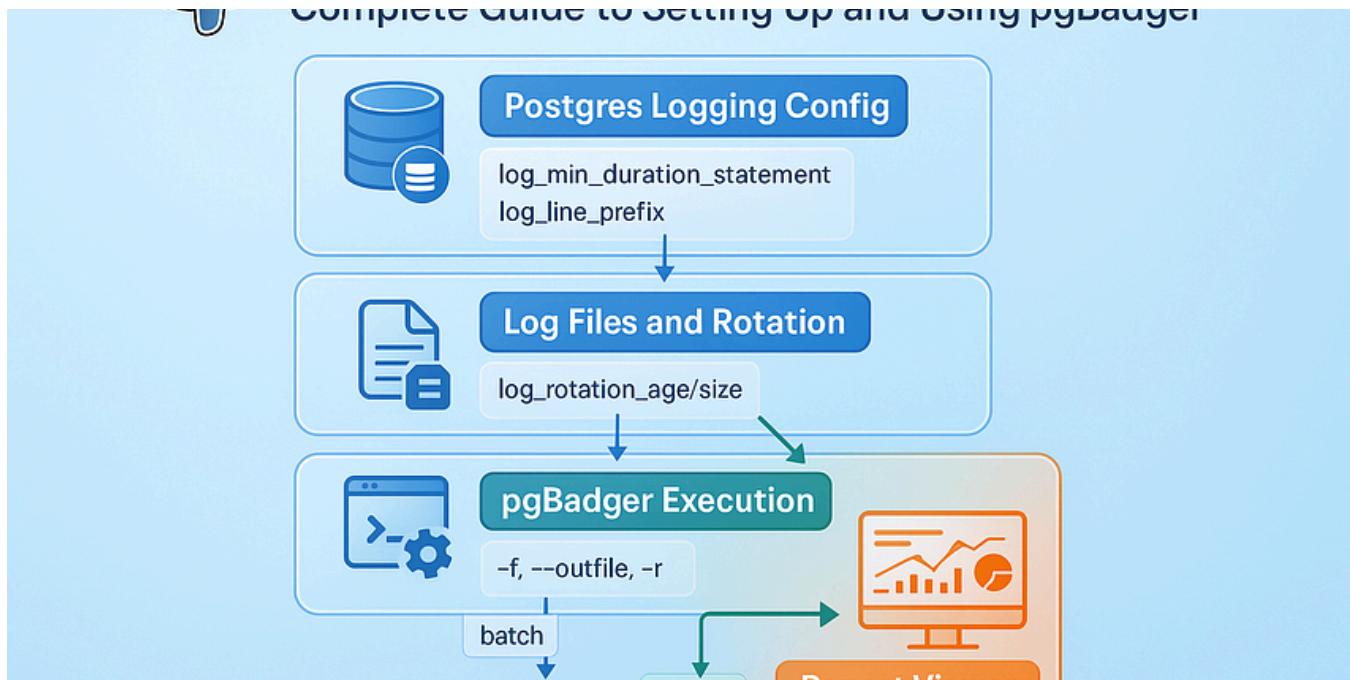
...

This blog is wonderful. You were telling about the connection pooling importance but did not give any steps to configure connection pooling .



[Reply](#)

More from Jeyaram Ayyalusamy



J Jeyaram Ayyalusamy

PostgreSQL 17 Log Analysis Made Easy: Complete Guide to Setting Up and Using pgBadger

Managing a production-grade PostgreSQL instance isn't just about keeping data safe—it's about monitoring performance, identifying...

Jun 23 52



A Deep Dive into How PostgreSQL Organizes Data

① SchemaSQL: Logical Structure

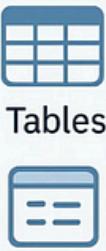
Every PostgreSQL database can contain multiple schemas.

PostgreSQL Physical Structure



PostgreSQL Physical Storage

- Database data files
- Transaction logs
- System metadata
- Cluster configuration
- Replication and recovery state



J Jeyaram Ayyalusamy

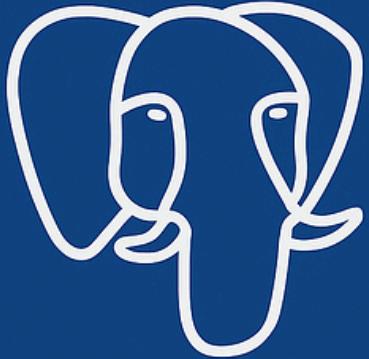
The Internal Structure of PostgreSQL: A Deep Dive into How PostgreSQL Organizes Data

PostgreSQL is one of the most powerful and popular open-source relational database systems used in production today. But while most people...

Jun 1 2



PostgreSQL 17 ADMINISTRATION



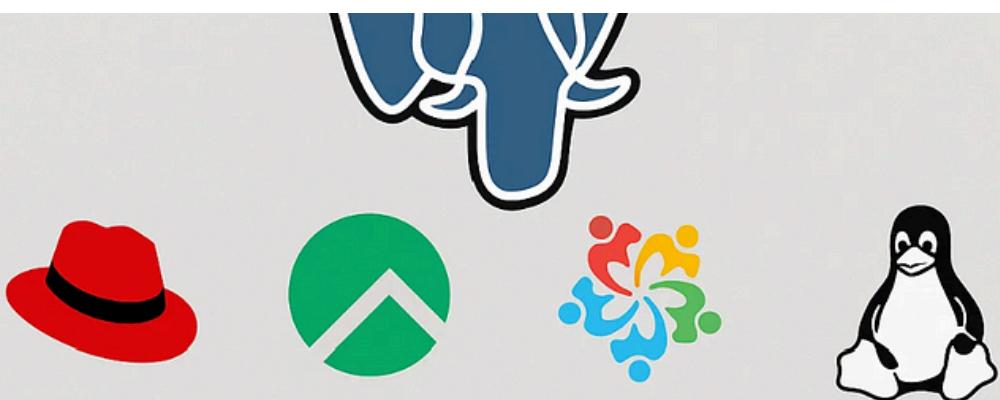
Mastering Schemas, Databases, and Roles

J Jeyaram Ayyalusamy 

PostgreSQL 17 Administration: Mastering Schemas, Databases, and Roles

PostgreSQL has evolved into one of the most feature-rich relational databases available today. With version 17, its administrative...

Jun 9 2



HOW TO INSTALL PostgreSQL 17 ON RED HAT, ROCKY, ALMALINUX,

 Jeyaram Ayyalusamy 

How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

PostgreSQL 17 is the latest release of one of the world's most advanced open-source relational databases. If you're using a Red Hat-based...

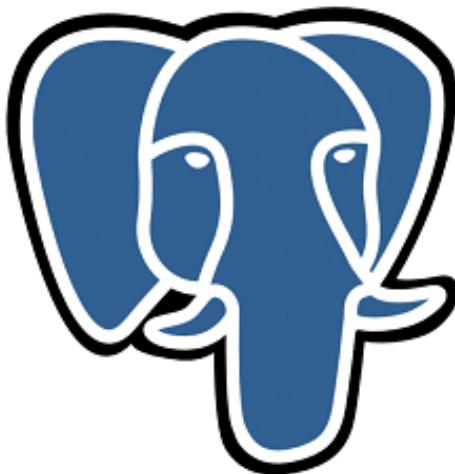
Jun 3



...

[See all from Jeyaram Ayyalusamy](#)

Recommended from Medium



PostareSQL

 Sohail Saifi

Postgres Hidden Features That Make MongoDB Completely Obsolete (From an Ex-NoSQL Evangelist)

For six years, I was that developer.

May 26

200

11



Deniz GÜRSOY

Select For Update in Postgres

The SELECT FOR UPDATE statement enables you to acquire exclusive locks on rows within PostgreSQL databases. An exclusive lock ensures that...

Jun 25

1



Ajaymaurya

PostgreSQL Show Database: Best Ways to List Databases

When you're working with PostgreSQL, especially from the command line, one of the first things you might ask is: "How do I list all the..."

Jun 25 13 1



View Table

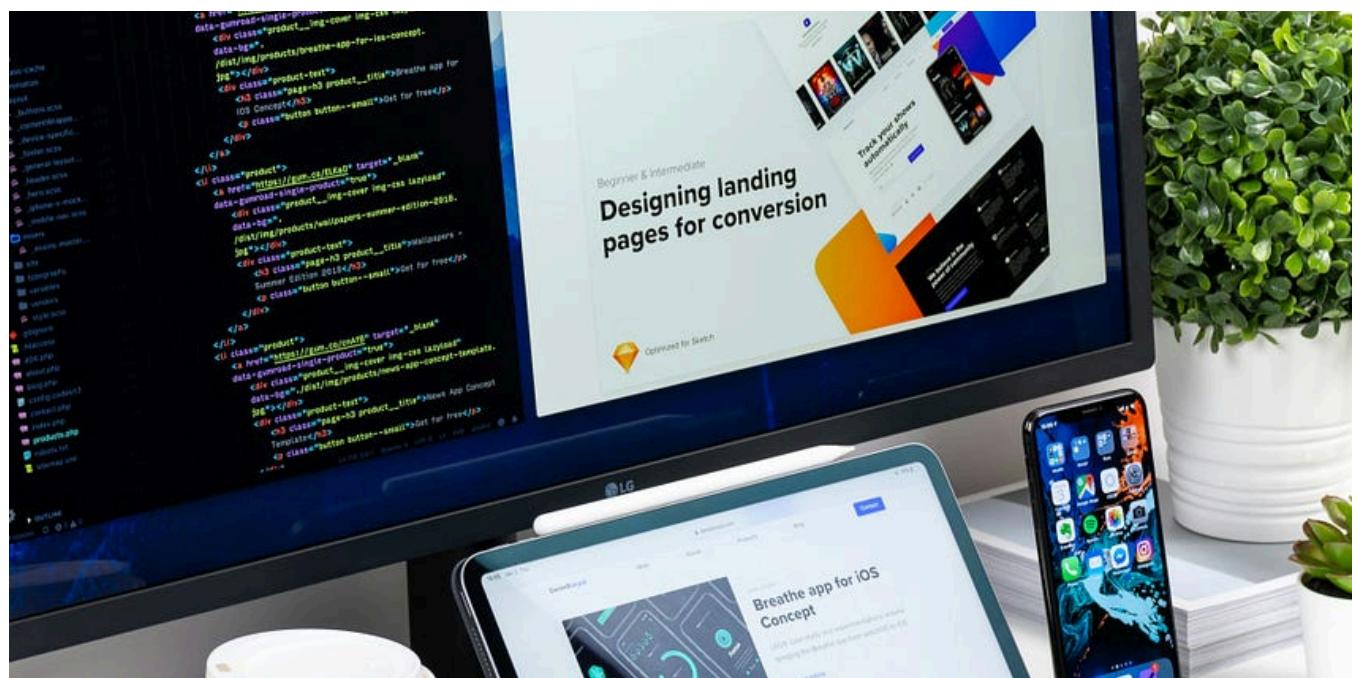
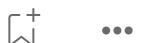
Purpose	Data Distribution	Use Case	Key Benefit
Scale horizontally	Split across servers	Huge user base (e.g. Twitter)	Handle large data volumes
Increase availability & backup	Duplicate data	Read-heavy apps (e.g. blogs)	High availability/tolerance
Manage large tables efficiently	Split within one server	Time-series logs	Better performance for tables

r Ravi Jaisawal

Sharding vs Replication vs Partitioning in Databases

Differences between Sharding, Replication, and Partitioning in databases — with practical examples and real-world use cases

Jun 11 4





Sandesh | DevOps | CI/CD | K8

“Why PostgreSQL is Beating MongoDB in 2025 (And Why I Switched Back)”

(Spoiler: My NoSQL experiment cost us 3 months of debugging headaches)



Jun 11



10



1



...



ThreadSafe Diaries

PostgreSQL 18 Just Rewrote the Rulebook, Groundbreaking Features You Can't Ignore

From parallel ingestion to native JSON table mapping, this release doesn't just evolve Postgres, it future-proofs it.



Jun 27



34



...

See more recommendations