PostgreSQL 17 Administration: Mastering Schemas, Databases, and Roles

14 min read · Jun 8, 2025

Jeyaram Ayyalusamy Following

PostgreSQL has evolved into one of the most feature-rich relational databases available today. With version 17, its administrative capabilities continue to expand — providing DBAs and developers with powerful tools for managing schemas, roles, databases, and security.

In this comprehensive guide, we'll cover PostgreSQL 17 administration essentials, focusing on schema management, role creation, database operations, remote access configuration, data generation, and migration tasks — all with fully functional SQL examples.

🏷️ What is a Schema in PostgreSQL?

When working with PostgreSQL, you'll often hear about **schemas** — but what exactly are they? Think of schemas as **folders inside your database** that help you organize and manage your objects efficiently. Whether you're building a small app or managing a large enterprise system, understanding schemas is essential for clean, maintainable database architecture.

🔎 What Is a Schema in PostgreSQL?

At its core:

> **A schema is a logical container that holds database objects.**

In PostgreSQL, schemas can contain:

- 📄 **Tables** — your actual data.
- 👀 **Views** — saved queries or virtual tables.
- 🔢 **Sequences** — counters for auto-incrementing columns.
- 🗂️ **Indexes** — for faster data access.
- ⚙️ **Functions & Procedures** — reusable logic.
- 🔐 **Triggers & Constraints** — for business rules and validations.

Each database can contain multiple schemas, and every object inside the database belongs to exactly one schema.

🔑 Why Use Schemas?

Schemas aren't just for show — they bring serious power and flexibility:

1️⃣ Organization: Group Related Objects Together

Instead of dumping everything into the default `public` schema, you can create multiple schemas to group objects by:

- Application module
- Business domain
- Project phase
- Environment (dev, test, prod)

For example:

- `sales.customers`
- `hr.employees`
- `finance.invoices`

2️⃣ Namespace Separation: Avoid Name Conflicts

A **namespace** is like a dedicated space (or container) where your database objects live. PostgreSQL uses schemas to create these namespaces.

Without schemas, every object name must be unique across the entire database. But with schemas, each object only needs to be unique within its own schema.

> ✅ **Same object name, different schemas — no problem.**

✅ You can have:

```
sales.customers
hr.customers
```

Same table name (`customers`), but no conflict because they're in different schemas.

3️⃣ Security Control: Manage User Access at Schema Level

In PostgreSQL, **schemas aren't just for organizing your objects — they're also a powerful tool for managing database security and permissions.**
By controlling access at the schema level, you can create fine-grained security policies while keeping your database structure clean and scalable.

🎯 Why Use Schema-Level Security?

Schemas allow you to:

- ✅ Grant or restrict access to entire groups of tables, views, and functions.
- ✅ Avoid redundant user roles or duplicated objects.
- ✅ Simplify permission management for large, multi-user systems.

🔐 Typical Use Cases

Scenario How Schemas Help **Limit user access** Allow some users to access only specific schemas **Separate environments** Isolate development, testing, and production data **Protect sensitive data** Restrict access to HR, payroll, or financial schemas

🚀 How Schema-Level Privileges Work

PostgreSQL supports several permissions you can grant on schemas:

Privilege Description `USAGE` Allows access *into* the schema (but not to its objects by default). `CREATE` Allows creating new objects within the schema.

> Note: Having `USAGE` doesn't automatically give access to tables — you still need to grant privileges on individual objects.

💡 Example: Grant Read-Only Access to a Schema

Let's say you have a schema called `hr` that contains sensitive employee data. You want a user `hr_readonly` to be able to see the objects, but not modify anything.

1️⃣ Grant access to the schema itself:

```
GRANT USAGE ON SCHEMA hr TO hr_readonly;
```

> ✅ This allows the user to "see" into the schema.

2️⃣ Grant SELECT access to the tables inside the schema:

```
GRANT SELECT ON ALL TABLES IN SCHEMA hr TO hr_readonly;
```

3️⃣ Ensure future tables are also accessible:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA hr
GRANT SELECT ON TABLES TO hr_readonly;
```

> ✅ This way, even newly created tables will automatically inherit the correct permissions.

📌 Benefits of Schema-Level Access Control

- 🔒 Simplifies permission management

- 🎯 **Easier role-based access control**
- 🚀 **Reduces human error and accidental exposure**
- 🛡️ **Helps meet compliance requirements (GDPR, HIPAA, etc.)**

🔑 Summary

- **Schemas = Logical containers** not just for organization, but also for **security boundaries.**
- You can **grant/restrict access** at both the schema and object levels.
- Schema-level security helps you build **scalable, maintainable, and secure** PostgreSQL environments.

✅ Creating a Schema in PostgreSQL — A Simple Guide

In PostgreSQL, **schemas** act like folders inside your database. They help organize your tables, views, indexes, sequences, and functions into logical groups. This makes your database cleaner, easier to maintain, and more secure.

🚀 Why Create a Schema?

- To separate different parts of your application (e.g., `sales`, `hr`, `finance`).
- To avoid naming conflicts — same table names can exist in different schemas.
- To manage permissions at a more granular level.

✨ How to Create a New Schema

Creating a schema in PostgreSQL is very straightforward. All you need is a simple `CREATE SCHEMA` command.

```
CREATE SCHEMA my_schema;
```

✅ That's it — your new schema is now ready to store tables, views, indexes, sequences, functions, and more.

📂 Creating Objects Inside Your Schema

Once your schema exists, you can start creating tables or other database objects within it by fully qualifying their names:

```
CREATE TABLE my_schema.customers (
    id SERIAL PRIMARY KEY,
    name TEXT,
    email TEXT
);
```

- `my_schema.customers` means:
- Create a table called `customers`
- Inside the schema called `my_schema`

✅ This fully-qualified naming convention allows you to have multiple `customers` tables across different schemas without conflict.

🔎 Accessing Objects in a Schema

When you query or modify data, you simply reference the schema-qualified name:

```
SELECT * FROM my_schema.customers;
```

You can also set your **search_path** so you don't have to type the schema every time:

```
SET search_path TO my_schema;
```

Now you can query like this:

```
SELECT * FROM customers;
```

PostgreSQL will automatically look inside `my_schema` for the `customers` table.

🔐 Pro Tip: Manage Permissions

Schemas also allow you to manage access easily:

```
GRANT USAGE ON SCHEMA my_schema TO readonly_user;
GRANT SELECT ON ALL TABLES IN SCHEMA my_schema TO readonly_user;
```

✅ This way, you control who can see or modify data inside your schema.

🔑 Summary

- **Schemas = Logical containers** inside your PostgreSQL database.
- Easy to create with `CREATE SCHEMA`.

- Help organize, isolate, and secure your data.
- Allow object name reuse across different business domains.

🏁 Quick PostgreSQL Schema Facts — Everything You Need to Know

Schemas in PostgreSQL are a powerful way to organize, isolate, and secure your database objects. Here's a quick, practical guide that summarizes the key facts about schemas — perfect for daily PostgreSQL administration.

📊 Quick Schema Facts Table

Feature	Benefit
Default Schema	`public` — every database starts with this schema
Fully Qualified Name	`schema_name.object_name` — allows multiple objects with same name in different schemas
Search Path	Defines the order PostgreSQL looks for objects when schema name isn't specified
Isolation	Objects in different schemas don't collide — avoid naming conflicts
Permissions	Schema-level permissions allow fine-grained security control

🔎 Fully Qualified Object Names

When working with schemas, you can reference objects using **fully qualified names:**

```
SELECT * FROM hr.employees;
```

- `hr` is the schema.

- `employees` is the table inside that schema.
- ✅ This avoids ambiguity, even if other schemas have a table named `employees`.

🔄 Search Path — Schema Lookup Order

The **search_path** setting tells PostgreSQL where to look first if you omit the schema name:

```
SHOW search_path;
```

You can customize it:

```
SET search_path TO hr, public;
```

- PostgreSQL will first check `hr`, then `public`.
- ✅ Great for switching schema contexts without fully qualifying object names every time.

🔐 Isolation & Permissions

Schemas allow:

- 🔒 Isolating development, test, and production datasets.
- 🔐 Granting or revoking permissions at the schema level:

```
GRANT USAGE ON SCHEMA finance TO analyst;
GRANT SELECT ON ALL TABLES IN SCHEMA finance TO analyst;
```

✅ This helps apply **role-based access control** easily.

🧪 Bonus: List All Schemas in Your Database

You can see all schemas in your current database using:

```
SELECT schema_name FROM information_schema.schemata;
```

This returns a list of all schemas available for your connected session.

🚀 Conclusion

PostgreSQL schemas are not just a feature — they are **best practice** for scalable, secure, and organized databases:

- ✅ Logical separation of business domains
- ✅ Simplified permission management
- ✅ No more table name conflicts
- ✅ Easier multi-tenant or modular database design

If you're building anything serious in PostgreSQL — **schemas are your friend**.

📋 Essential PostgreSQL Administration Commands — A Quick Guide for DBAs

PostgreSQL provides a powerful set of administrative commands to help database administrators (DBAs) manage users, databases, privileges, and schemas efficiently. Whether you're setting up a new environment or maintaining production systems, these commands will become part of your daily toolkit.

🔎 Listing Schemas

Using psql Meta-command:

```
psql -c "\dn;"
```

- Lists all schemas present in the current database.

Using SQL Query:

```
SELECT schema_name FROM information_schema.schemata;
```

- Standard SQL way to retrieve all schemas.

🔎 Listing Database Roles

To see all roles and their properties:

```
psql -c "\du+;"
```

- This will show:
- Role names
- Attributes (e.g., superuser, create role, create db)
- Role memberships

⚠️ Dropping a Database

Be extremely careful — this will **permanently delete** the database and all its data:

```
psql -c "DROP DATABASE demodb;"
```

- Always double-check before executing.
- Cannot be undone!

⚠️ Dropping a Role

Before dropping a role, make sure it doesn't own any objects:

```
psql -c "DROP ROLE demouser;"
```

- If the role owns objects, you'll receive an error.
- Reassign ownership or drop dependent objects first.

✅ Creating Users (Roles)

In PostgreSQL, users are roles with login privileges.

```
psql -c "CREATE USER demouser WITH PASSWORD 'secret_passwd';"
```

- You can also add options like `CREATEDB`, `SUPERUSER`, or `CREATEROLE` depending on the user's responsibility.

✅ Creating Databases

Create a new database easily with:

```
psql -c "CREATE DATABASE demodb;"
```

- The default owner will be the user who runs this command.
- Use `OWNER` clause if you want to assign a different owner.

✅ Granting Database Privileges

Grant full privileges on a database to a user:

```
psql -d demodb -c "GRANT ALL PRIVILEGES ON DATABASE demodb TO demouser;"
```

- This allows `demouser` to connect to the database and create objects.

✅ Granting Table Privileges

To give `demouser` full access to all tables in the public schema:

```
psql -d demodb -c "GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO demou
```

Grant on Sequences (needed for AUTO_INCREMENT-like behavior):

```
psql -d demodb -c "GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO de
```

Grant on Functions:

```
psql -d demodb -c "GRANT ALL PRIVILEGES ON ALL FUNCTIONS IN SCHEMA public TO de
```

- This ensures the user can fully interact with tables, sequences, and functions inside the schema.

🚀 Why These Commands Matter

- ✅ They simplify user onboarding.
- ✅ They help enforce security and role-based access control.
- ✅ They give you fine-grained control over database resources.
- ✅ They are critical for automation and scripting in DevOps workflows.

🔐 Pro Tip

Always combine these commands with regular backups and audit logging, especially when managing production PostgreSQL environments.

🚀 PostgreSQL Quick Tips: Automation, Schemas & Test Data Generation

PostgreSQL offers many powerful features that help DBAs and developers automate tasks, manage schemas, and even generate large amounts of test data effortlessly. In

this article, we'll break down some of the most useful techniques for your day-to-day PostgreSQL administration and development.

✅ Exporting Password for Automation

When you run `psql` commands inside automation scripts or shell pipelines, entering the password manually can interrupt execution. PostgreSQL allows you to set the `PGPASSWORD` environment variable to handle passwords securely within scripts.

```
export PGPASSWORD=secret_passwd
```

- 🔒 Once set, any `psql` command you run in that session will automatically use this password.
- ✅ Great for CI/CD pipelines, cron jobs, or automated deployments.
- ⚠️ Use carefully — don't expose your credentials in shared or unsecured environments.

✅ Creating Schemas as Users

You can create schemas while logged in as any authorized user. This helps with organization and role-based data isolation.

```
psql -d demodb -U demouser -c "CREATE SCHEMA demodb;"
```

- This creates a new schema named `demodb` inside the `demodb` database.
- Useful for logically separating application modules or multi-tenant data.

🔍 Checking Connection Information

Need a quick check on your current connection status? Use `\conninfo` directly in `psql`:

```
psql -d demodb -c "\conninfo"
```

- Displays:
- Current database name
- User
- Host
- Port
- SSL status
- ✅ Excellent for troubleshooting and verifying that you're connected to the intended instance.

🗃️ Generating Large Test Datasets

Testing performance often requires big tables. PostgreSQL makes this super easy using its built-in `generate_series()` function.

✅ Creating a Table with 1 Million Rows

Here's how to instantly create a table filled with 1 million rows of synthetic data:

```
psql -h localhost -d demodb -U demouser -c "CREATE TABLE demodb.one_million_row
```

- 🔢 `generate_series(1,1000000)` produces numbers 1 to 1,000,000.
- ✅ Perfect for testing query performance, indexing, and bulk operations.

✅ Counting Rows (Validation)

After generation, verify how many rows were inserted:

```
psql -h localhost -d demodb -U demouser -c "SELECT COUNT(*) FROM demodb.one_mil
```

- Confirms data volume.
- Useful for sanity checks in performance tests.

✅ Creating Additional Schemas

Schemas help you organize your database into logical containers. Create additional schemas easily:

```
psql -h localhost -d demodb -U demouser -c "CREATE SCHEMA apps;"
```

- 🔧 This creates a new schema named `apps` inside `demodb`.
- ✅ You can create multiple schemas for different microservices, features, or customers.

🚀 Why This Matters

- ✅ Simplifies automation and scripting for DBAs and developers.
- ✅ Enables rapid synthetic data generation for stress testing and benchmarking.
- ✅ Supports better organization through schema management.
- ✅ Empowers you to build robust automation pipelines for PostgreSQL administration.

🌐 How to Configure Remote Access in PostgreSQL (Step-by-Step Guide)

PostgreSQL is designed with security as its top priority — by default, it only allows local connections to the database. This default configuration ensures that no external users can connect unless explicitly permitted. However, in many real-world scenarios, you may want to allow remote clients, application servers, or database tools to connect from external machines.

In this article, you'll learn exactly how to configure PostgreSQL for secure remote access.

🔐 Why Remote Access is Disabled by Default

- ✅ Prevents unauthorized external access.
- ✅ Ensures database is only accessible via localhost after installation.
- ✅ Forces administrators to consciously configure access rules for safety.

🛠️ 3 Easy Steps to Enable Remote Access

✅ 1️⃣ Update `postgresql.conf` — Allow External Connections

The `postgresql.conf` file controls PostgreSQL's server-level settings, including which IP addresses the server listens to.

👉 Find the file location:

You can find the config file with:

```
SHOW config_file;
```

Or locate it directly (commonly on Linux):

```
sudo vi /etc/postgresql/16/main/postgresql.conf
```

👉 Modify listen_addresses:

Inside the file, locate the `listen_addresses` setting and change it to:

```
listen_addresses = '*'
```

- `'*'` means the server will accept connections on **all network interfaces**.
- Alternatively, for more restrictive access, you can specify individual IP addresses (e.g. `listen_addresses = '192.168.1.10'`).

✅ **Important:** Save the file after editing.

✅ 2️⃣ Update `pg_hba.conf` — Define Who Can Connect

The `pg_hba.conf` file (PostgreSQL Host-Based Authentication) controls **which users** can connect, **from where**, and **using what authentication**.

👉 Open the file:

```
sudo vi /etc/postgresql/16/main/pg_hba.conf
```

👉 Add a new rule:

Example to allow remote clients from `192.168.1.0/24` subnet to connect using MD5 password authentication:

```
host    all    all    192.168.1.0/24    md5
```

Explanation of columns:

Column	Meaning
host	Allow TCP/IP connections
all (database)	Allow access to all databases
all (user)	Allow access to all users
192.168.1.0/24	Accept connections from this IP range
md5	Use password authentication

✅ **Tip:** You can specify a single IP address too:

```
host    all    all    192.168.1.50/32    md5
```

✅ 3️⃣ Reload PostgreSQL Configuration

After making changes, apply them by reloading the PostgreSQL service:

```
sudo systemctl reload postgresql
```

Alternatively, you can fully restart PostgreSQL (especially after major config changes):

```
sudo systemctl restart postgresql
```

🧪 Quick Test

Try connecting from your remote client:

```
psql -h <postgresql-server-ip> -U postgres -d your_database
```

- If your firewall allows traffic on port `5432` (default PostgreSQL port), you should successfully connect.
- Ensure your server's firewall or security group rules (AWS, GCP, Azure) permit PostgreSQL traffic.

🚩 ⚠️ Security Considerations

- Never expose PostgreSQL to public IP addresses without proper security measures.
- Use strong passwords or better — **SSL/TLS encryption**.
- Limit IP ranges to trusted networks only.

- Always audit `pg_hba.conf` for overly permissive rules.

🚀 Summary

Step	Action
1	Set `listen_addresses = '*'` in `postgresql.conf`
2	Add allowed client IPs and authentication rules in `pg_hba.conf`
3	Reload PostgreSQL to apply changes

> ✅ **Now your PostgreSQL instance is ready to securely accept remote connections!**

🔄 PostgreSQL Migration & Multi-Database Management — The Complete Guide

In large-scale applications, data migrations and multi-database setups are common. PostgreSQL is built with powerful capabilities to handle multiple databases and schemas, allowing you to organize, isolate, and manage data with ease — whether you're migrating between systems, isolating workloads, or managing multiple tenants.

Let's break it down step-by-step.

✅ Why Use Multiple Databases and Schemas?

- **Isolation:** Keep environments separate (e.g., development, staging, production).
- **Organization:** Group logically related data.

- **Migration:** Simplify large-scale data transfers without affecting existing workloads.
- **Multi-Tenancy:** Handle multiple customers or clients in the same PostgreSQL cluster.

🏗️ Creating New Databases for Migration or Isolation

PostgreSQL allows you to create as many databases as your cluster resources can support. This is extremely useful during migrations where you want to move data into a fresh database environment.

🔧 Command to Create a New Database

You can create a new database directly using `psql`:

```
psql -c "CREATE DATABASE migrationdb WITH OWNER migrationuser;"
```

Explanation:

Command	Purpose
`CREATE DATABASE migrationdb`	Creates a new database named `migrationdb`
`WITH OWNER migrationuser`	Assigns ownership of this database to `migrationuser`

✅ This ensures that `migrationuser` has full control over the database.

📂 Creating Schemas Inside the New Database

After creating the database, you can further organize your data using schemas. A schema is like a namespace inside the database where you can group related tables, views, and functions.

🔧 Command to Create a Schema

Connect to the new database and create the schema:

```
psql -h localhost -U migrationuser -d migrationdb -c "CREATE SCHEMA schemamigra
```

Explanation:

Command	Purpose
`-h localhost`	Connect to local PostgreSQL server
`-U migrationuser`	Use the specified user
`-d migrationdb`	Connect to the `migrationdb` database
`CREATE SCHEMA schemamigrationdb`	Create a new schema named `schemamigrationdb`

✅ Now you have a clean environment for staging data, running migration scripts, or organizing imported datasets.

⚠️ Dropping Schemas Safely

Sometimes you may need to remove a schema during cleanup or after completing a migration.

🚨 Drop a Schema and All Its Objects

```
DROP SCHEMA schema_name CASCADE;
```

Explanation:

- `CASCADE` ensures that **all objects** (tables, views, functions, sequences, etc.) inside the schema are also dropped.

- This operation is irreversible — once dropped, the data and objects cannot be recovered unless you have backups.

✅ Always double-check before running `DROP SCHEMA ... CASCADE;`.

🚀 Pro Tip: Best Practices During Migration

- Always perform full backups before creating or dropping schemas.
- Use schemas for logically grouping imported data.
- Assign clear ownership to migration users for better access control.
- Clean up unused schemas once the migration is fully validated.

📝 Summary Table

Task	Command
✅ Create Database	`CREATE DATABASE migrationdb WITH OWNER migrationuser;`
✅ Create Schema	`CREATE SCHEMA schemamigrationdb;`
⚠️ Drop Schema	`DROP SCHEMA schema_name CASCADE;`

> PostgreSQL's flexible database and schema management features make migrations smoother and more controlled — whether you're moving legacy systems or setting up multi-tenant architectures.

🏁 Conclusion

PostgreSQL 17 continues to offer an incredibly powerful, flexible, and enterprise-ready feature set for database administrators and developers. Mastering these essential administrative tasks will help you:

- ✅ Maintain clean and organized database structures
- ✅ Control user permissions efficiently
- ✅ Prepare for migrations and scalability
- ✅ Automate data generation for performance testing

Understanding schemas, roles, privileges, and remote access is fundamental to professional PostgreSQL database management — and will serve as the foundation for even more advanced topics such as replication, partitioning, and performance tuning.

👉 *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!

Open Source | Database | Postgresql | Oracle | MySQL

Following

Written by Jeyaram Ayyalusamy

76 followers · 2 following

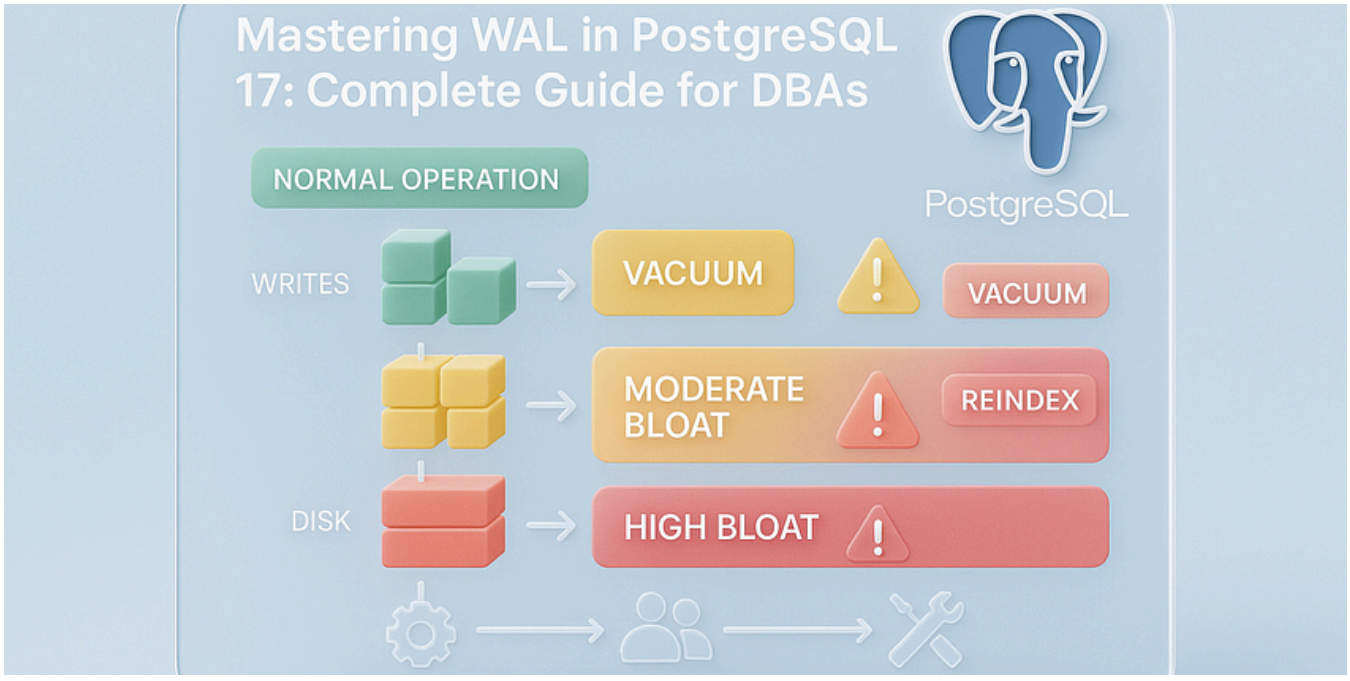
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

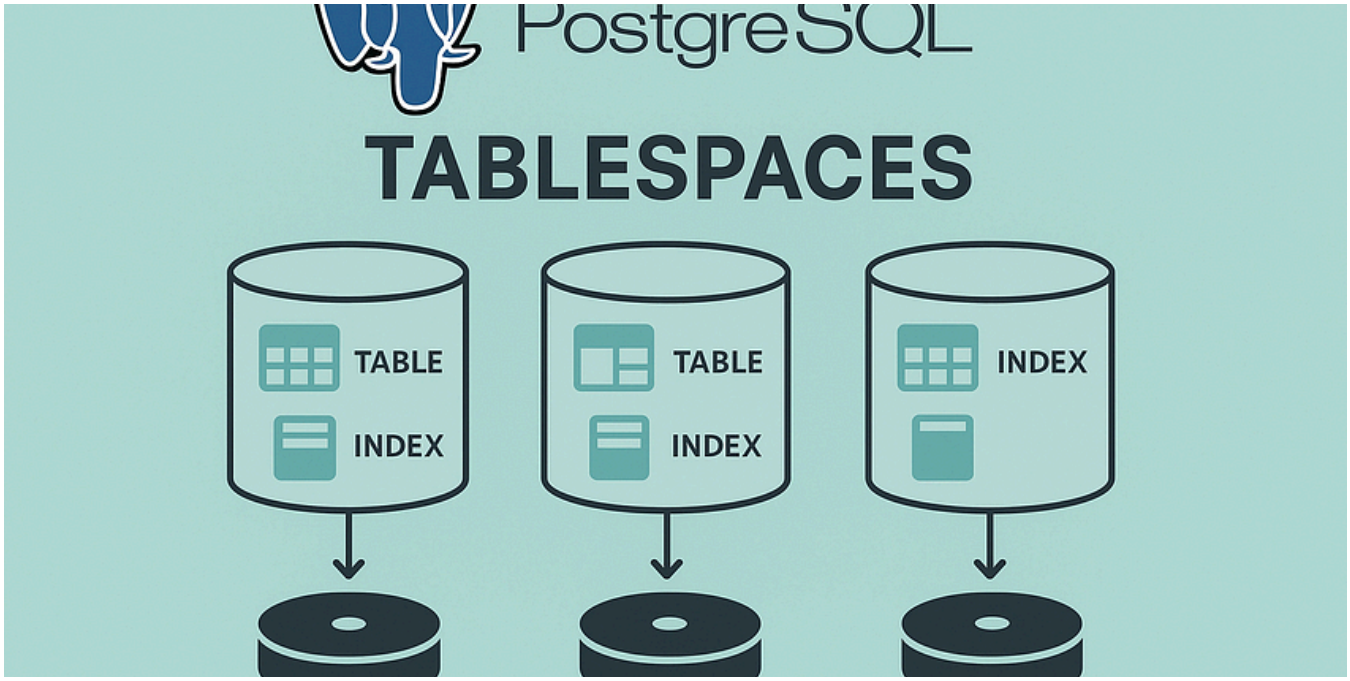


Jeyaram Ayyalusamy

Understanding and Managing Bloating in PostgreSQL: A Complete Guide for DBAs

PostgreSQL is a powerful, reliable, and feature-rich open-source relational database system. It's praised for its extensibility, ACID...

Jun 25 52

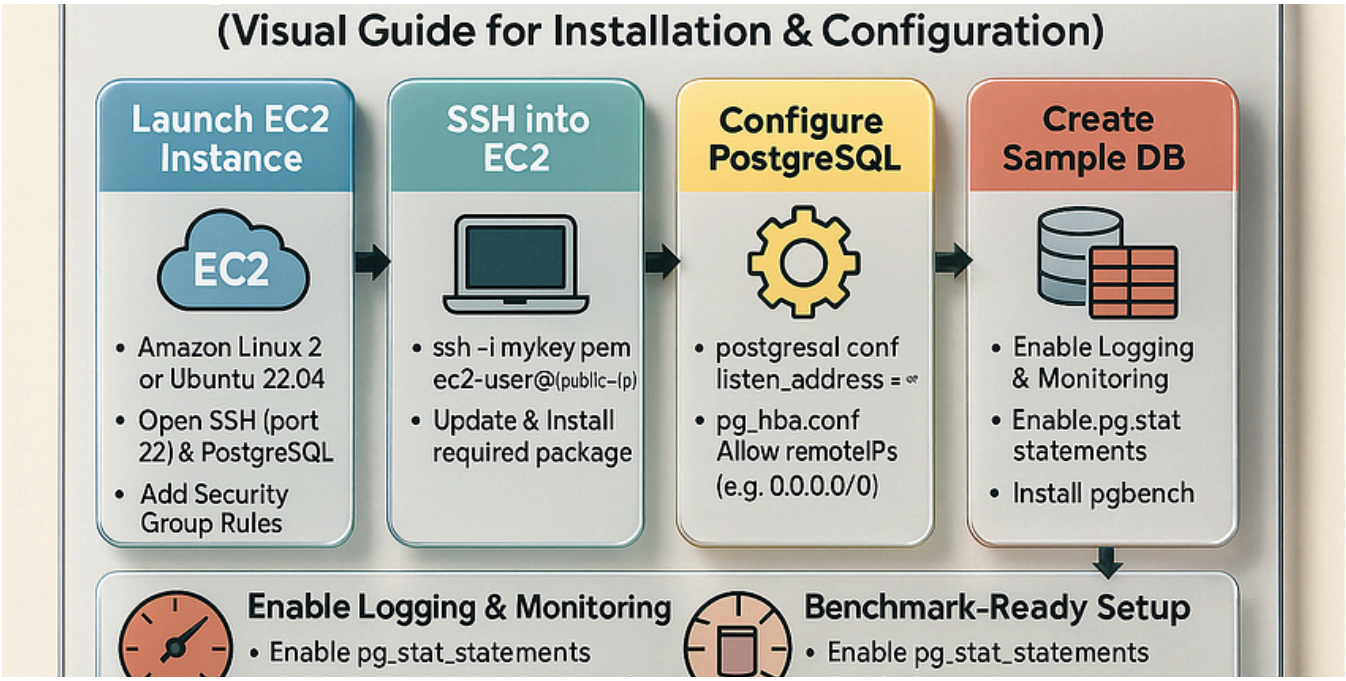


Jeyaram Ayyalusamy

PostgreSQL Tablespaces Explained: Complete Guide for PostgreSQL 17 DBAs

PostgreSQL offers a powerful feature called tablespaces, allowing database administrators to take control of how and where data is...

Jun 12 8



Jeyaram Ayyalusamy

PostgreSQL 17 on AWS EC2—Full Installation & Configuration Walkthrough

Setting up PostgreSQL 17 on AWS EC2 might seem complex at first—but once you break down each component, it becomes an efficient and...

1d ago 50

Jeyaram Ayyalusamy

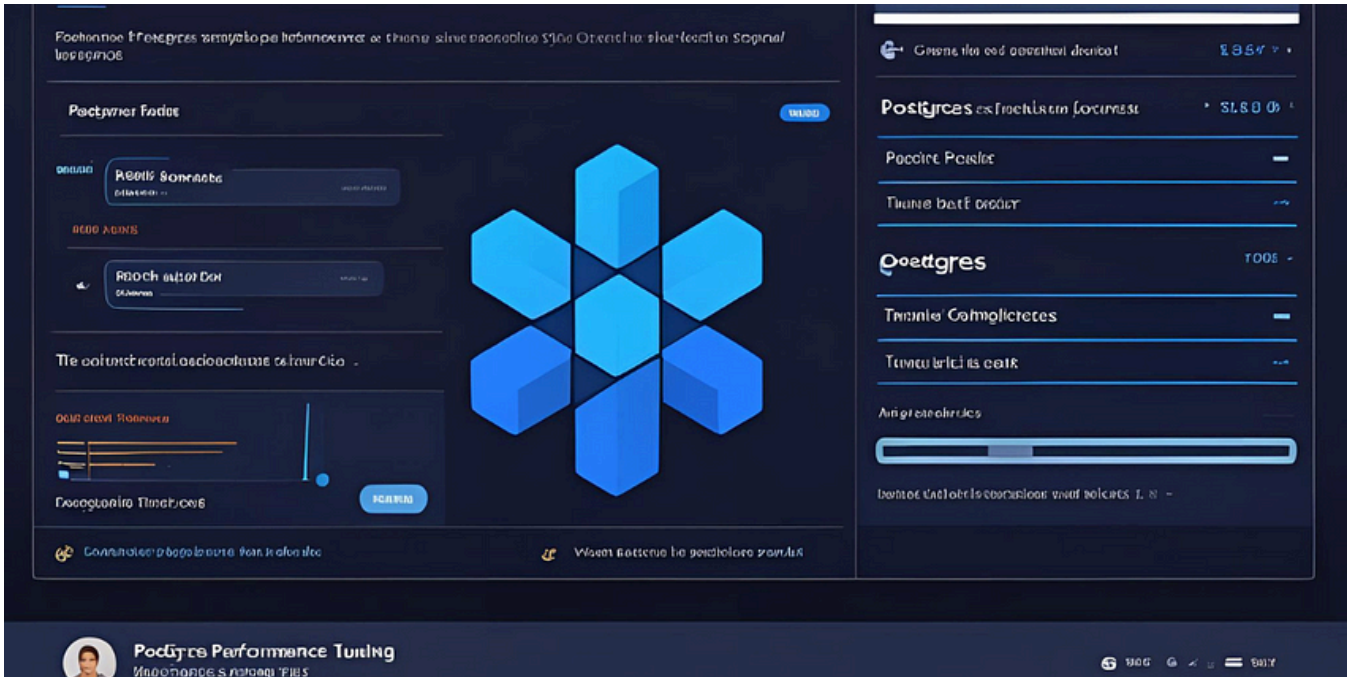
How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

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



Rizqi Mulki

Postgres Performance Tuning Like a Pro

The advanced techniques that separate database experts from everyone else

6d ago 55

Azlan Jamal

Stop Using SERIAL in PostgreSQL: Here's What You Should Do Instead

In PostgreSQL, there are several ways to create a PRIMARY KEY for an id column, and they usually involve different ways of generating the...

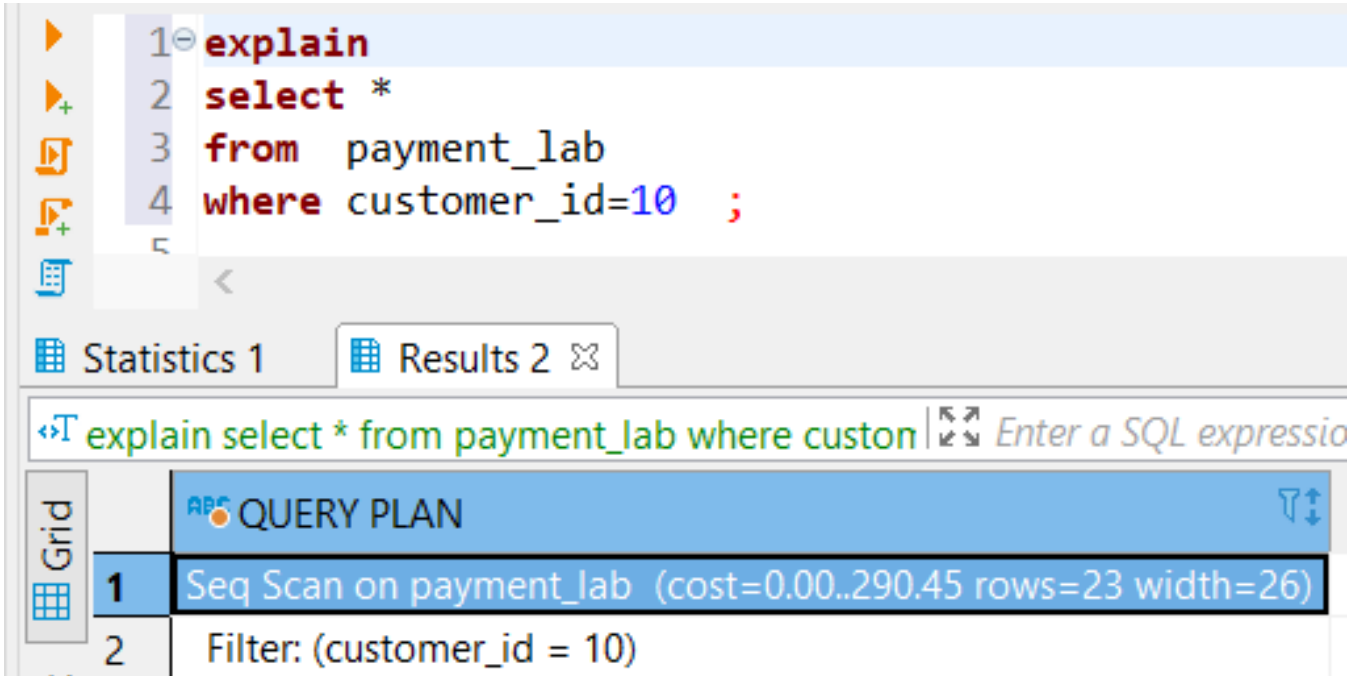
Jul 12 33

Oz

Understanding PostgreSQL Page (Block) Structure

PostgreSQL stores data on disk in fixed-size units called pages or blocks. In this post, we'll explore how PostgreSQL organizes data...

May 14 58 1



Muhammet Kurtoglu

Postgresql Query Performance Analysis

SQL tuning is critically important for ensuring database performance, reliability, and scalability. In most cases, performance issues in a...

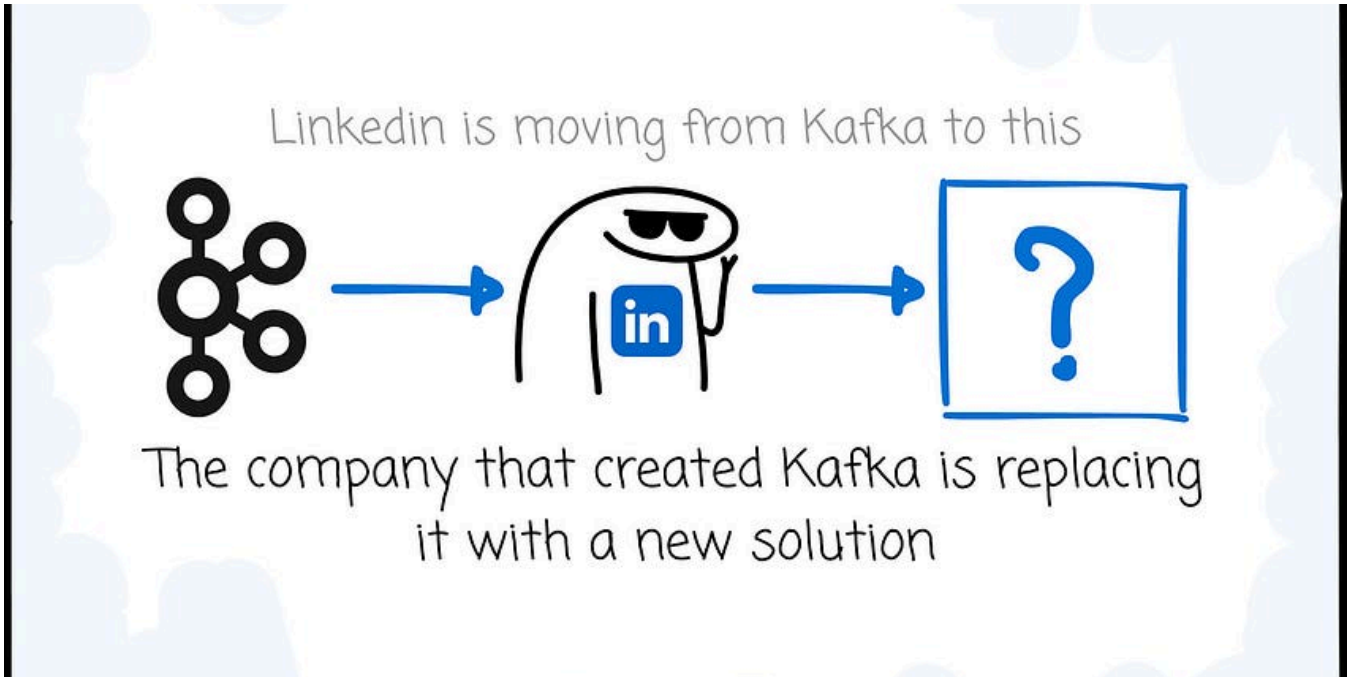
6d ago 10

techWithNeeru

This SQL Query Took 20 Seconds. Now It Takes 20ms. Here's What I Changed

Last week, I was troubleshooting a performance issue that was driving our team crazy. Our analytics dashboard was timing out, users were...

Jul 10 66



In Data Engineer Things by Vu Trinh

The company that created Kafka is replacing it with a new solution

How did LinkedIn build Northguard, the new scalable log storage

Jul 17 330 6

See more recommendations