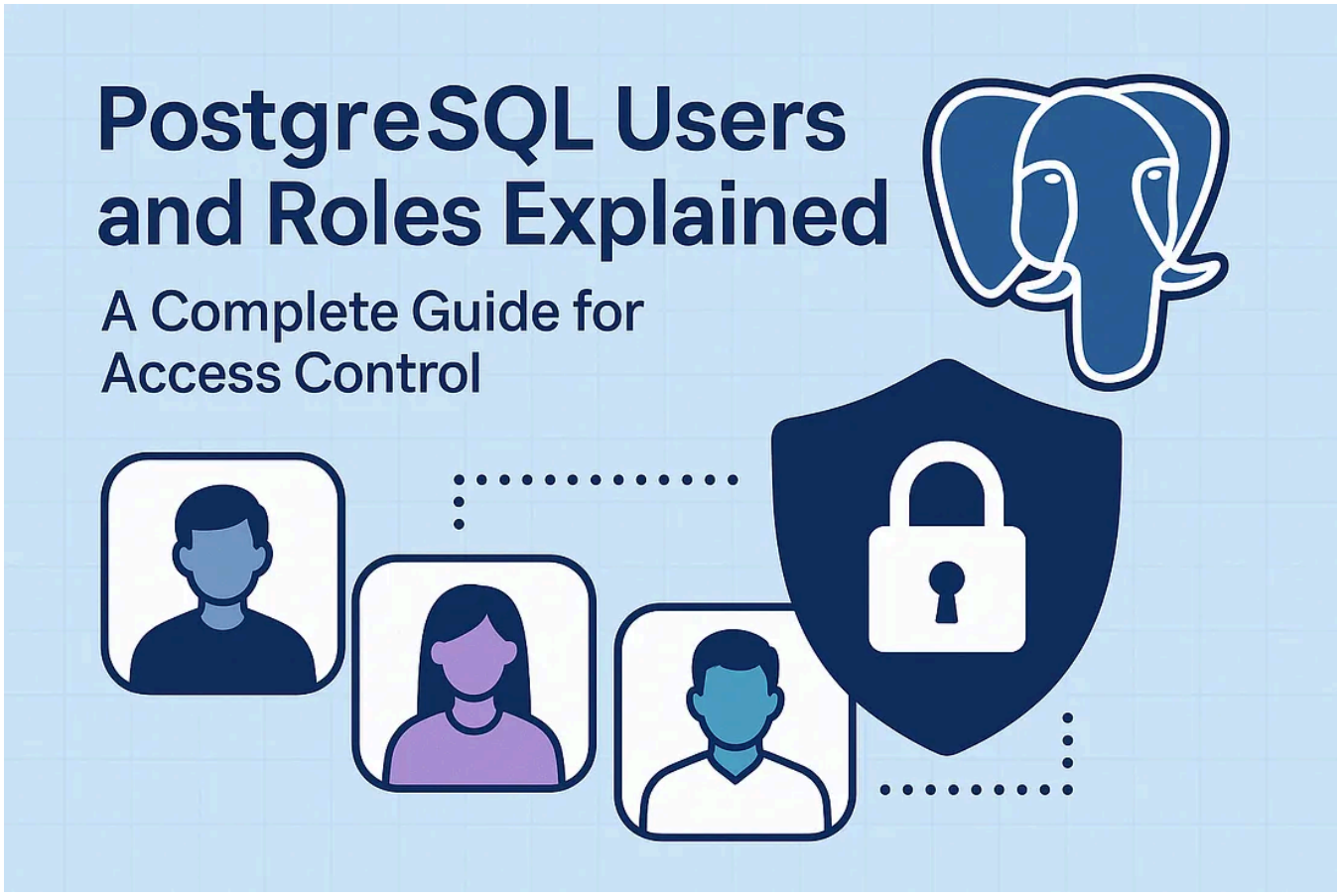
PostgreSQL Users and Roles Explained: A Complete Guide for Access Control

13 min read · Jun 10, 2025

J Jeyaram Ayyalusamy Following



Managing access and permissions is at the heart of any database system — and PostgreSQL offers a highly flexible, powerful model for this: **Users and Roles.**

Whether you're managing a small application or an enterprise-level system, understanding how users and roles work in PostgreSQL is essential for ensuring security, scalability, and efficient privilege management. This guide walks you through everything you need to know — with plenty of real-world SQL examples.

🚀 PostgreSQL Users vs. Roles: What's the Difference?

When working with PostgreSQL, understanding the difference between **users** and **roles** is crucial for managing database security, access control, and permissions effectively. Let's dive deep into how PostgreSQL handles these concepts — and why roles are at the heart of PostgreSQL's flexible permission system.

🔑 The Core Concept

In PostgreSQL, **users** and **roles** are not completely separate entities. Technically, PostgreSQL handles everything as a **role**. The difference lies in whether a role has login privileges or not.

- **User:**
  A role **with login privileges**. This is an account that can connect to the database using a username and password.
- **Role:**
  A broader concept that can represent users, groups, or collections of privileges. Roles can be granted to other roles, allowing flexible management of permissions.

🔧 How PostgreSQL Uses Roles Internally

PostgreSQL simplifies its permission model by merging the concepts of users and groups into roles. When you create a user, you're actually creating a role with the ability to log in.

For example:

```
-- Creating a user (role with login privilege)
CREATE ROLE john LOGIN PASSWORD 'secure_password';
```

```
-- Creating a role without login privilege (useful for grouping
privileges)
CREATE ROLE readonly;
```

Here, `john` is a role with login privileges — in common terms, a user. The `readonly` role is a group of privileges that can be assigned to multiple users.

🎯 Assigning Roles to Users

One of PostgreSQL's most powerful features is its role-based privilege inheritance. You can assign roles to other roles, effectively building a hierarchy of permissions.

Example:

```
-- Grant readonly privileges to john
GRANT readonly TO john;
```

Now, `john` inherits all the privileges of the `readonly` role, making permission management scalable and efficient.

🔒 Why Roles are a Best Practice

✅ Centralized Permission Management:

Instead of assigning individual privileges to each user, you define roles for various job functions or permission sets.

✅ **Simpler Administration:**
Adding or removing a user's access becomes as simple as granting or revoking roles.

✅ **Scalability:**
As your database grows, role-based permissions make it easier to onboard new users or adjust permissions without disrupting other accounts.

👉 **Pro Tip:**
Always plan your role hierarchy carefully when starting a new PostgreSQL project. Well-designed roles save you from permission headaches later!

👤 Users in PostgreSQL: A Deep Dive

In PostgreSQL, **users** are the primary way to control access to your database. Whether you're managing human access or application-level connections, PostgreSQL users represent authenticated identities that can interact with the system.

Although PostgreSQL internally treats **users as roles with login privileges**, it's common practice to refer to them as "users" when discussing connection-level access.

✅ What is a PostgreSQL User?

A PostgreSQL **user** is essentially a role that has permission to **log in** to the database server. This means it can authenticate using a username and password and then perform operations as permitted by its privileges.

Think of a user as:

- A database login identity (for developers, apps, admins)
- A role with the **LOGIN** attribute

✅ Creating a User

You can create a new user with a simple SQL command using the `CREATE USER` statement:

```
CREATE USER myuser WITH PASSWORD 'mypassword';
```

This command:

- Creates a new role named `myuser`
- Enables login with the specified password
- Adds no other privileges by default (e.g., it cannot create databases or roles unless explicitly granted)

🔧 Optional Attributes When Creating Users

You can further customize the user with additional parameters:

```
CREATE USER myuser
WITH PASSWORD 'mypassword'
VALID UNTIL '2025-12-31'
CONNECTION LIMIT 5;
```

- `VALID UNTIL` : Specifies an expiration date for the password.
- `CONNECTION LIMIT` : Restricts the number of concurrent connections for the user.

These settings are helpful for managing security and resource allocation.

✅ Listing Users

To see the list of all users (and other roles), you can use the following command in `psql`:

```
\du+
```

This displays:

- Role names (users and groups)
- Attributes like login capability, superuser status, role creation rights
- Any role memberships (which users are part of which groups)

Example output:

```
Role name | Attributes                  | Member of
-----------+-----------------------------+------------
 myuser    | Login                       | {}
 readonly  | Cannot login                | {}
 admin     | Superuser, Create role...   | {}
```

This is your go-to view for auditing and managing user permissions.

✅ Dropping a User

If a user is no longer needed, you can remove them from the system:

```
DROP USER myuser;
```

However, note:

- You cannot drop a user if they own any database objects (tables, schemas, etc.).

- You may need to reassign ownership or drop dependent objects first.

```
REASSIGN OWNED BY myuser TO another_user;
DROP OWNED BY myuser;
DROP USER myuser;
```

This ensures a clean and safe deletion without leaving orphaned objects behind.

🧠 Quick Recap

Task Command Create a user `CREATE USER myuser WITH PASSWORD 'mypassword';` List users `\du+` Drop a user `DROP USER myuser;`

By managing **users** correctly in PostgreSQL, you lay a strong foundation for secure and organized database access control. As your environment grows, consider assigning users to **roles** to streamline privilege management — but it all starts with understanding the fundamentals of users themselves.

🎭 Roles in PostgreSQL: Grouping Privileges the Smart Way

When it comes to managing access in PostgreSQL, **roles** are the backbone of a scalable and secure permission system. Instead of assigning individual permissions to every user, PostgreSQL encourages the use of roles to **group privileges** and assign them in bulk. This not only simplifies administration but also makes permission management consistent and maintainable.

🔧 What is a Role?

A **role** in PostgreSQL is a flexible concept that can act as:

- A **user** (a role with login capability)
- A **group** (a role without login, used to collect permissions)
- Or both

The real power of roles lies in their ability to **inherit privileges** from other roles, which creates a clean and scalable access control hierarchy.

✅ Creating a Role

To create a role, use the `CREATE ROLE` command:

```
CREATE ROLE myrole;
```

By default, this role **cannot log in**. It's a placeholder for privileges, designed to be granted to users or other roles.

You can think of this like creating a security group in your system — it doesn't do anything by itself but becomes powerful when connected to users.

✅ Granting Privileges to Roles

Once you create a role, you can grant specific database object privileges to it. For example:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON customers TO myrole;
```

This command gives the `myrole` role permission to perform all common operations on the `customers` table. You can repeat this for other tables or specific columns if

needed.

Roles allow you to centralize permission logic — update a role, and all users assigned to it instantly get updated access.

✅ Assigning Roles to Users

Once a role is configured with the desired privileges, you can assign it to users.

```
CREATE USER customer1 WITH PASSWORD 'password1';
GRANT myrole TO customer1;
```

Here's what happens:

- `customer1` becomes a PostgreSQL user.
- By granting `myrole` to `customer1`, the user now inherits all the privileges assigned to `myrole`.

This inheritance is immediate and dynamic — any future changes to `myrole` will automatically apply to `customer1`.

🔄 How Users and Roles Work Together

This relationship — **roles manage permissions**, and **users inherit roles** — allows for elegant and efficient access control.

Benefits:

- 🔁 **Reusable permission sets** across multiple users
- ⚙️ **Centralized updates** — change role once, affect all users
- 🔒 **Cleaner access design** for applications with many users

🔎 Example Scenario: E-Commerce Access Levels

Let's say you're designing an e-commerce system. You want all your customer accounts to have read and write access to the `customers` table — but only to certain columns.

Here's how you can structure it:

```
-- Create a reusable role for all customers
CREATE ROLE customer;
```

```
-- Grant typical privileges to that role
GRANT SELECT, INSERT, UPDATE ON customers TO customer;

-- Create individual users and assign them the role
CREATE USER customer1 WITH PASSWORD 'password1';
GRANT customer TO customer1;

CREATE USER customer2 WITH PASSWORD 'password2';
GRANT customer TO customer2;
```

Now, both `customer1` and `customer2` inherit permissions from the `customer` role. This setup ensures consistency — any changes to `customer`'s privileges will automatically apply to both users.

🧠 Quick Recap

Task Command Create a role `CREATE ROLE myrole;` Grant privileges to a role `GRANT SELECT, INSERT ON table TO myrole;` Assign role to user `GRANT myrole TO username;`

✅ Best Practice

🔑 **Design roles around job functions or access patterns**, not individuals.
Example: `customer`, `admin`, `readonly`, `support_team`

🎯 Then, assign these roles to users. This approach keeps your permission structure flexible, auditable, and future-proof.

📂 The Public Schema and Role in PostgreSQL: Understanding Defaults and Securing Access

When you install PostgreSQL and start working with a fresh database, you'll quickly notice something called the **public schema**. This schema — along with a special **public role** — is created automatically and plays a significant role in how new users interact with the database.

But here's the catch: **if not configured properly, the public schema can be a potential security loophole.** Let's explore what it is, how it works, and how to secure it.

🔍 What is the Public Schema?

In PostgreSQL, a **schema** is a logical container for database objects like tables, views, functions, and sequences. Think of it like a folder that helps organize your database contents.

By default, every PostgreSQL database contains a schema named `public`. This is:

- Automatically created when a new database is initialized
- Accessible by **all users** via the special **`PUBLIC` role**

This means that unless you change it, **any user in the database can create objects inside the `public` schema.** This design simplifies things during development but poses risks in production environments.

👥 What is the PUBLIC Role?

`PUBLIC` is a **special keyword** in PostgreSQL that refers to **all roles (users)** — both existing and future ones. When you grant privileges to `PUBLIC`, you're giving that access to *every user* in the system.

By default, PostgreSQL grants the `CREATE` privilege on the `public` schema to `PUBLIC`, which allows all users to:

- Create tables
- Create functions
- Create any other object in that schema

While convenient, this default behavior can lead to:

- Unintended object creation
- Security risks in shared databases
- Conflicts between users or applications

✅ Viewing the Current Search Path

The **search path** in PostgreSQL determines the order in which schemas are searched when you run a query without explicitly specifying the schema.

To view the current search path:

```
SHOW search_path;
```

Typical output:

```
"$user", public
```

Here's what it means:

- First, PostgreSQL checks for a schema named after the current user (`"$user"`).
- Then, it checks the `public` schema.

If you don't specify the schema when creating a table or querying one, PostgreSQL follows this search order.

✅ How to Secure the Public Schema

To prevent every user from creating objects in the `public` schema, **revoke the default create privilege** from the `PUBLIC` role:

```
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
```

🔐 What This Does:

- It stops users (other than superusers or those explicitly granted privileges) from creating objects in the `public` schema.
- It helps avoid accidental table creation in shared environments.
- It enforces better access control and schema usage practices.

💡 Best Practices

1. **Use dedicated schemas** for each application or user group instead of relying on `public` .
2. **Always revoke `CREATE` from `PUBLIC`** on the public schema in production environments.
3. **Explicitly grant privileges** to specific roles as needed.

Example:

```
-- Create a new schema
CREATE SCHEMA sales;
```

```
-- Grant create/use access to a specific role
GRANT USAGE, CREATE ON SCHEMA sales TO sales_user;
```

This ensures clean separation of concerns and better security.

🧠 Summary Table

Task Command View search path `SHOW search_path;` Revoke create on public schema `REVOKE CREATE ON SCHEMA public FROM PUBLIC;` Secure schema access Use custom schemas + targeted grants

🚨 Final Thoughts

The `public` schema and `PUBLIC` role are powerful defaults that make development easier — but they're **not safe defaults** for production. Taking time to restrict unnecessary access and use schema-based permission models is a critical step toward building secure PostgreSQL environments.

🧱 Creating Read-Only and Read/Write Roles in PostgreSQL

Effective permission management is essential in any production database environment. One of the most powerful practices in PostgreSQL is to create **custom roles** that reflect specific access needs — such as **read-only** for analysts and **read/write** for application users.

By creating dedicated roles for different use cases, you ensure security, consistency, and scalability across your PostgreSQL ecosystem.

🔍 Why Use Read-Only or Read/Write Roles?

- **Read-Only Role:** Ideal for users who only need to view data — such as data analysts, business intelligence tools, or auditors. These roles **cannot modify** any data.
- **Read/Write Role:** Perfect for application components or users who need to read from and write to the database — like CRUD operations in APIs or back-office apps.

Instead of assigning granular permissions to individual users, you define **standardized roles**, then assign users to those roles. This keeps your access control clean and manageable.

✅ Creating a Read-Only Role

Here's how you can create a read-only role in PostgreSQL and grant it the necessary privileges:

```
-- Step 1: Create the read-only role
CREATE ROLE readonly;
```

```
-- Step 2: Allow the role to connect to the database
GRANT CONNECT ON DATABASE demodb TO readonly;

-- Step 3: Grant access to the schema (but not object creation)
GRANT USAGE ON SCHEMA public TO readonly;

-- Step 4: Allow the role to select from all existing tables
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;

-- Step 5: Ensure future tables also inherit SELECT privilege
ALTER DEFAULT PRIVILEGES IN SCHEMA public GRANT SELECT ON TABLES TO
readonly;
```

🔒 What this configuration does:

- `CONNECT` : Enables the role to log into `demodb` .

- `USAGE` : Allows access to the `public` schema, without object creation rights.
- `SELECT` : Grants read-only access to all current tables.
- `ALTER DEFAULT PRIVILEGES` : Ensures all **future** tables in the schema will also be readable by `readonly` .

This is extremely useful for **reporting dashboards**, **data exports**, and **readonly replicas**.

✅ Creating a Read/Write Role

For users or services that need full data access — both read and write — you can create a read/write role as follows:

```
-- Step 1: Create the read/write role
CREATE ROLE readwrite;
```

```
-- Step 2: Allow the role to connect to the database
GRANT CONNECT ON DATABASE demodb TO readwrite;

-- Step 3: Grant access to and creation within the schema
GRANT USAGE, CREATE ON SCHEMA public TO readwrite;

-- Step 4: Grant full CRUD access to all existing tables
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA public
TO readwrite;

-- Step 5: Set default privileges for future tables
ALTER DEFAULT PRIVILEGES IN SCHEMA public
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO readwrite;
```

🛠️ What this setup allows:

- Full access to existing and future tables within the `public` schema.
- The ability to create objects (tables, functions, etc.) in the schema.
- Ideal for internal apps, automation tools, or backend microservices that require full interaction with the data.

🧠 Summary Table

Role Type Permissions Use Case `readonly` `CONNECT`, `USAGE`, `SELECT` Reporting, dashboards, auditing `readwrite` `CONNECT`, `USAGE`, `CREATE`, `SELECT`, `INSERT`, `UPDATE`, `DELETE` Full-access apps, API users, backend services

✅ Best Practice

- Use **dedicated roles** for each type of access (e.g., `readonly`, `readwrite`, `admin`).
- Assign these roles to users instead of giving direct object privileges.
- Always include `ALTER DEFAULT PRIVILEGES` to ensure future-proof access control.

🔄 Role Assignment Example

```
-- Assign a user to the read-only role
GRANT readonly TO analyst_user;
```

```
-- Assign a user to the read/write role
GRANT readwrite TO backend_user;
```

This model keeps your database secure, organized, and easy to manage — especially as the number of users and applications grows.

🔐 Managing User Permissions in PostgreSQL: A Role-Based Approach

In PostgreSQL, the most effective way to manage user access is through a **role-based permission system**. Instead of assigning privileges directly to individual users, you

define roles with specific privileges and **assign or revoke roles** as needed. This strategy scales beautifully — whether you're managing a few users or thousands.

Let's explore how to manage user permissions in PostgreSQL efficiently and securely.

✅ Assigning Roles to Users

Once you've created a role (like `readonly` or `readwrite`), you can assign it to users. This grants the user all permissions associated with that role.

```
CREATE USER demouser1 WITH PASSWORD 'secret_passwd';
GRANT readonly TO demouser1;
```

In this example:

- `demouser1` is a user with login privileges.
- By granting the `readonly` role, the user inherits all the permissions defined under `readonly`, such as `SELECT` access on tables.

This model decouples **permission logic** from individual user accounts and keeps things organized.

✅ Revoking Roles from Users

Need to revoke access? You can just remove the role assignment:

```
REVOKE readwrite FROM demouser1;
```

This command **instantly removes** all permissions associated with `readwrite` from `demouser1`, without having to revoke each privilege manually.

✅ Checking Granted Roles

To audit role assignments and verify who has what access, use this query:

```
SELECT
    r.rolname,
    ARRAY(
        SELECT b.rolname
        FROM pg_catalog.pg_auth_members m
        JOIN pg_catalog.pg_roles b ON (m.roleid = b.oid)
        WHERE m.member = r.oid
    ) as memberof
FROM pg_catalog.pg_roles r
ORDER BY 1;
```

📃 Output Example:

rolname member of demouser1 {readonly} backendapp {readwrite}

This makes it easy to **track access levels** across your database.

🗄 Full Permission Management Example

Here's a consolidated setup, perfect for enterprise environments where you want fine-grained, secure, and scalable user access:

```
-- 🔒 Step 1: Revoke default public access
REVOKE CREATE ON SCHEMA public FROM PUBLIC;
REVOKE ALL ON DATABASE demodb FROM PUBLIC;
```

```
-- 📘 Step 2: Create and configure a read-only role
CREATE ROLE readonly;
```

```
GRANT CONNECT ON DATABASE demodb TO readonly;
GRANT USAGE ON SCHEMA myschema TO readonly;
GRANT SELECT ON ALL TABLES IN SCHEMA myschema TO readonly;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema
GRANT SELECT ON TABLES TO readonly;

-- 🛠️ Step 3: Create and configure a read/write role
CREATE ROLE readwrite;
GRANT CONNECT ON DATABASE demodb TO readwrite;
GRANT USAGE, CREATE ON SCHEMA myschema TO readwrite;
GRANT SELECT, INSERT, UPDATE, DELETE ON ALL TABLES IN SCHEMA
myschema TO readwrite;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema
GRANT SELECT, INSERT, UPDATE, DELETE ON TABLES TO readwrite;

-- 🔄 Step 4: Handle sequence permissions (for auto-increment IDs,
etc.)
GRANT USAGE ON ALL SEQUENCES IN SCHEMA myschema TO readwrite;
ALTER DEFAULT PRIVILEGES IN SCHEMA myschema
GRANT USAGE ON SEQUENCES TO readwrite;
```

🧠 Why This Works:

- Users assigned `readonly` can only view data.
- Users with `readwrite` can perform full CRUD operations and even create new tables or sequences.
- Revoking public access ensures tight security from the beginning.

🔎 Why Use Roles?

PostgreSQL's role-based system provides several advantages:

✅ Centralized Permission Control

Define once, apply everywhere.

✅ Scalability

Manage hundreds or thousands of users effortlessly by grouping permissions.

✅ Security & Auditing

Know exactly who has access to what. Simplifies internal audits and compliance checks.

✅ Faster Onboarding/Offboarding

Assign or revoke roles in one step — no manual permission juggling.

🧩 Final Thoughts

PostgreSQL's role-based permission system is one of the most powerful features for database administrators and developers alike. By structuring access through well-defined roles, you maintain security, simplify operations, and scale effortlessly with your growing user base.

🚀 Conclusion

PostgreSQL gives you full flexibility to build a permission system that fits your business needs, whether you're running SaaS platforms, financial systems, or internal applications.

Mastering users and roles is **not optional** — it's your first step toward building a secure, scalable PostgreSQL environment.

👉 *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 AWS Open Source MySQL Oracle

Following

Written by Jeyaram Ayyalusamy

76 followers · 2 following

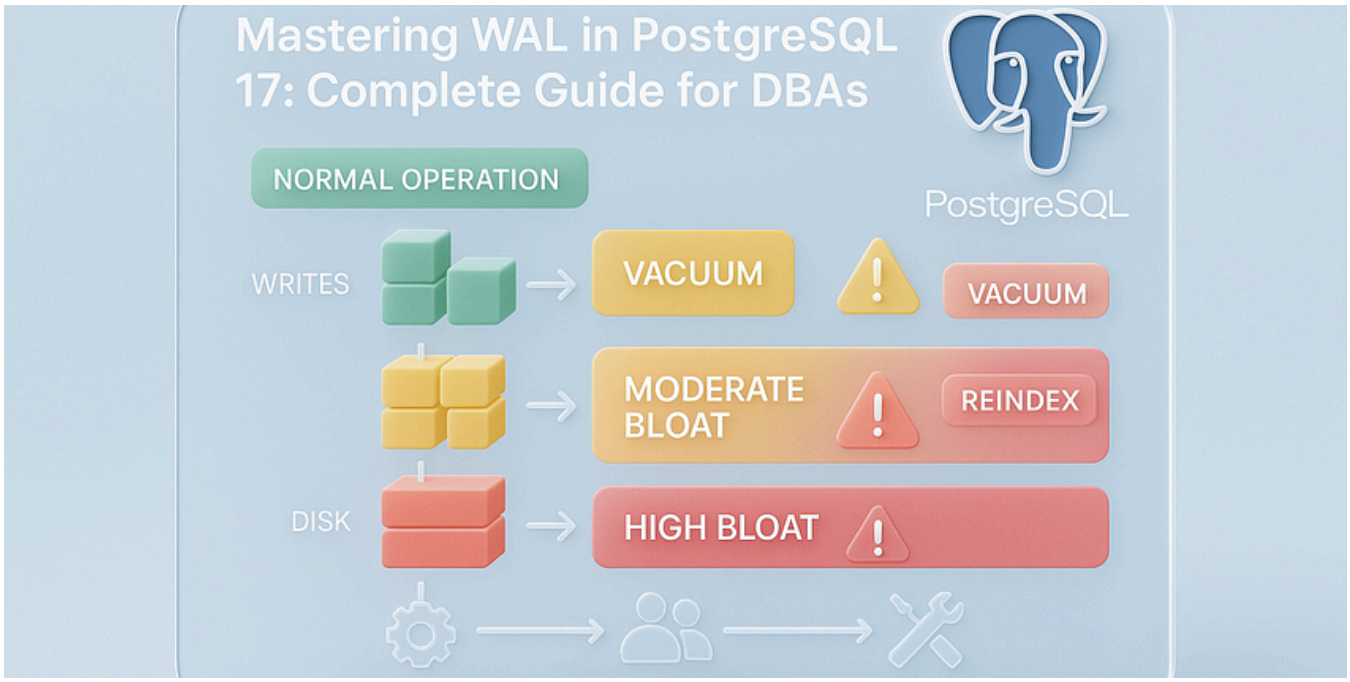
Oracle DBA | AWS Certified | 18+ yrs in DB Admin, RAC, GoldenGate, RDS, MySQL, PostgreSQL | Author of MySQL & RDS books | Cloud & Performance Expert

No responses yet

Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy

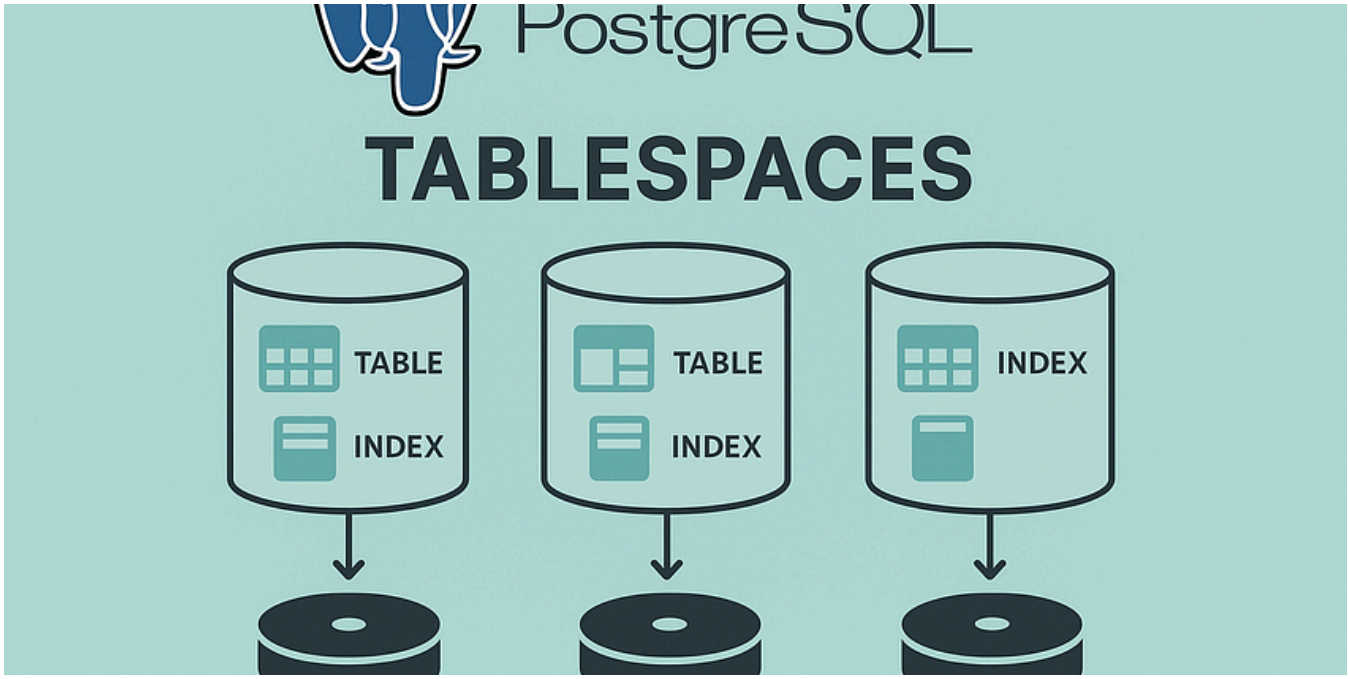


Jeyaram Ayyalusamy

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

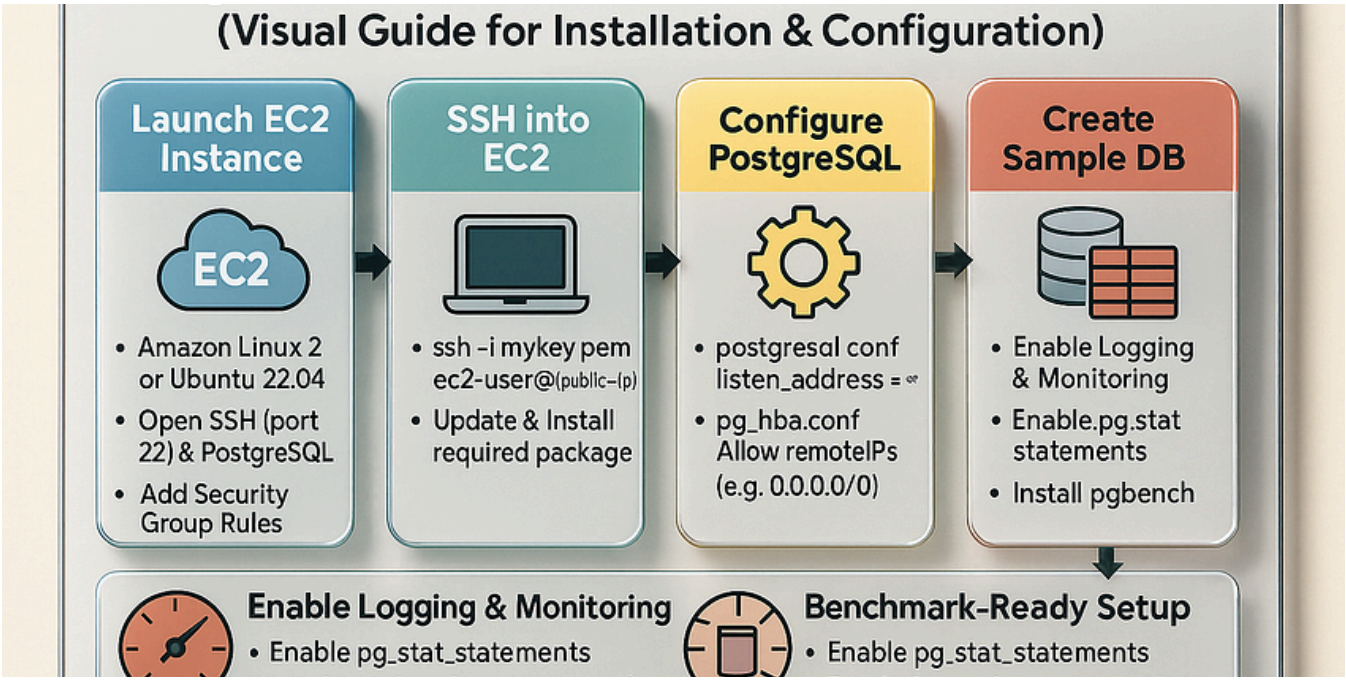


Jeyaram Ayyalusamy

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

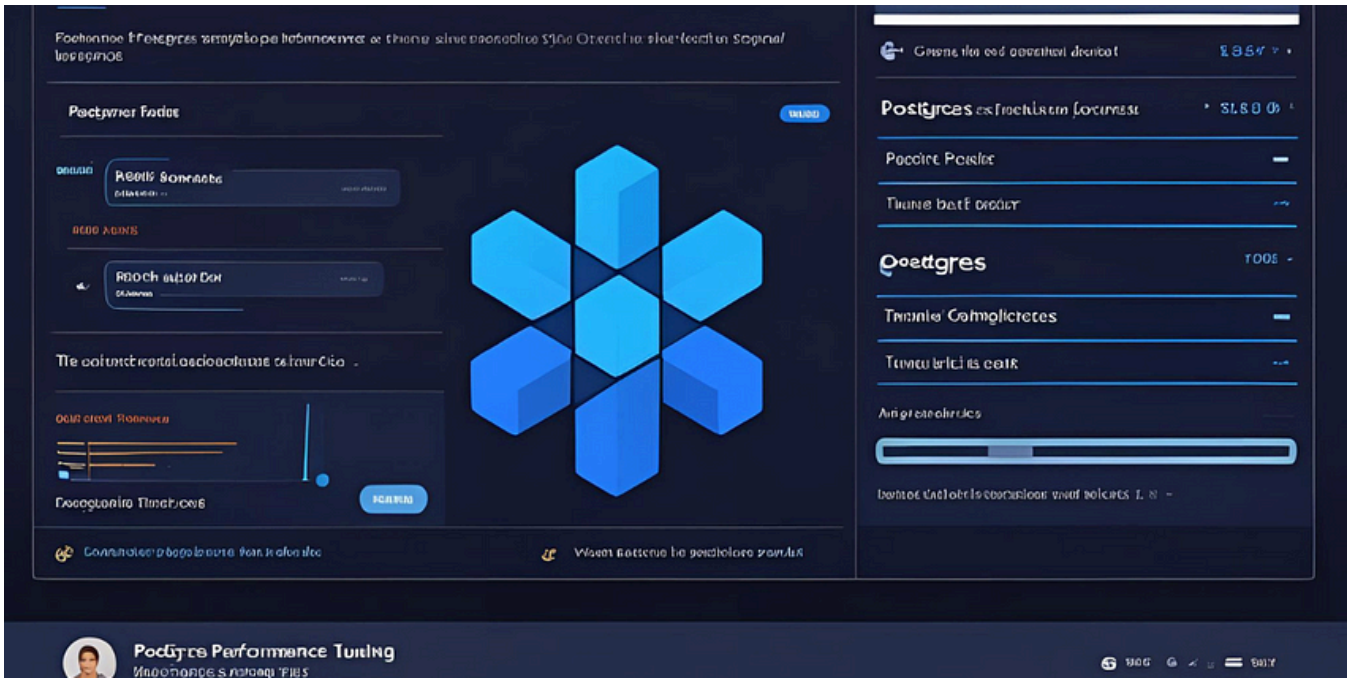
How to Install PostgreSQL 17 on Red Hat, Rocky, AlmaLinux, and Oracle Linux (Step-by-Step Guide)

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

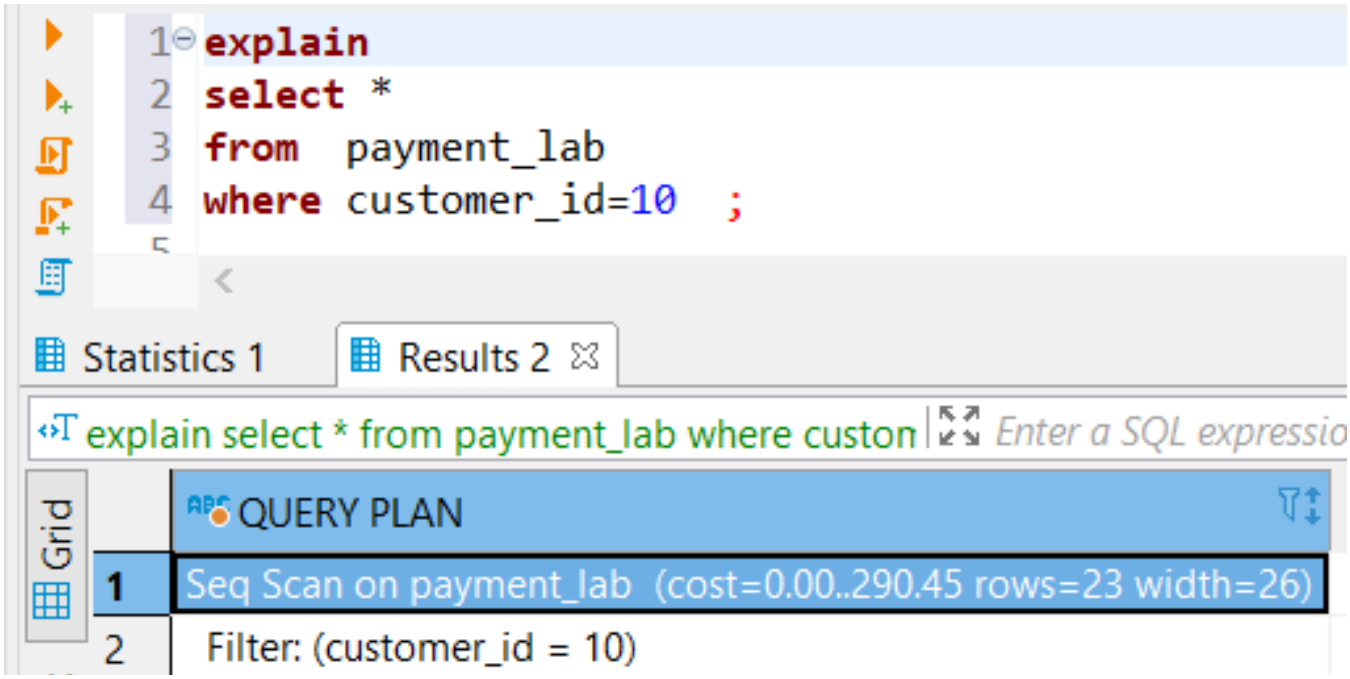


Rizqi Mulki

Postgres Performance Tuning Like a Pro

The advanced techniques that separate database experts from everyone else

6d ago 55



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

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

Harishsingh

PostgreSQL 18 in Microservices: You Don’t Need a Separate DB for Everything

Introduction: The Myth of Database-Per-Service

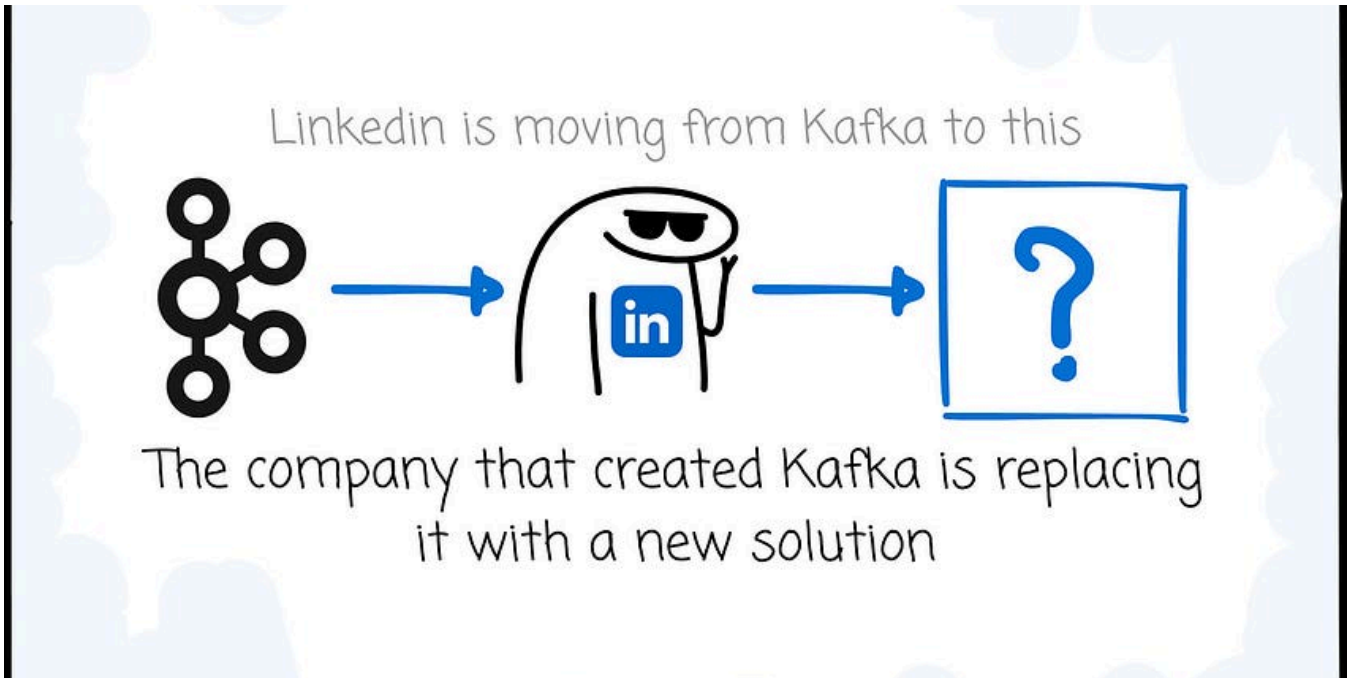
Jul 13 11 1

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