

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



Understanding the Public Schema and Search Path in PostgreSQL: A Practical Guide

[Open in app](#)

Medium



Search

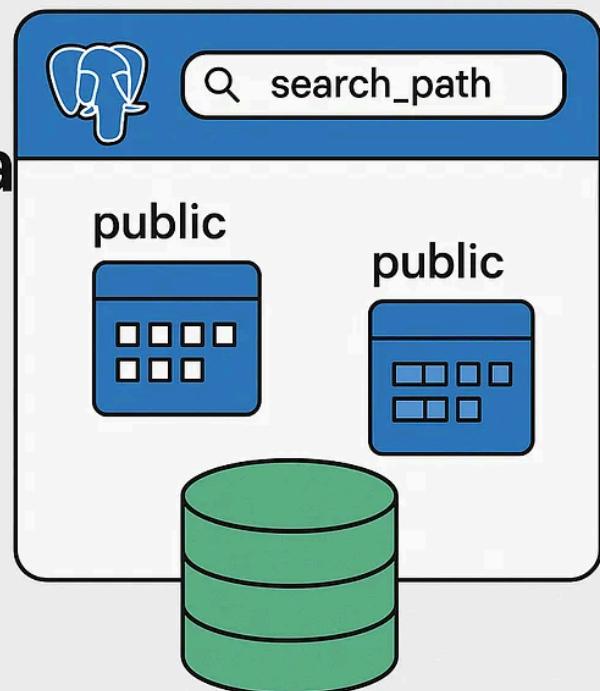


[Listen](#)

[Share](#)

[More](#)

Understanding the Public Schema and Search Path in PostgreSQL: A Practical Guide



When you work with PostgreSQL, one of the most overlooked — but incredibly powerful — features is the **public schema**. While it's convenient for quick development, improper management can lead to security, organization, and maintainability issues as your database grows.

In this article, we'll break down the public schema, its pros and cons, how PostgreSQL resolves object names using the **search path**, and most importantly — how to manage all of it with real-life SQL examples.

What Is the Public Schema in PostgreSQL?

When working with PostgreSQL, one of the first things developers and database administrators encounter is the **concept of schemas**. A *schema* is essentially a **namespace** that allows you to group and organize related database objects — like tables, views, indexes, and functions — within a database.

But there's one schema that stands out from the beginning: the **public schema**.

What Is a Schema?

Think of a schema as a folder within your database. Just as folders help you organize files on your computer, schemas help you organize database objects. This is particularly useful when you have a large application with multiple components, teams, or modules accessing the same database.

The Public Schema Explained

Every new PostgreSQL database comes with a **default schema named `public`**. Unless you explicitly tell PostgreSQL to place your objects into a specific schema, it assumes you want to use the public one.

This makes the public schema a **default working area** — ideal for quick prototyping, testing, or smaller-scale projects.

For example, take the following table creation statement:

```
CREATE TABLE my_table (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50)
);
```

Since we didn't mention any schema in this command, PostgreSQL will automatically place `my_table` into the **public schema**. In reality, PostgreSQL

interprets this command as:

```
CREATE TABLE public.my_table (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50)
);
```

Why Does This Matter?

Understanding the public schema is essential because:

-  **Access Control:** All users have privileges on the public schema by default. If you're working in a multi-user environment, you may want to **revoke or restrict permissions** to prevent accidental changes.
-  **Object Organization:** In larger databases, using multiple schemas helps separate development and production objects, or modularize features.
-  **Security and Clarity:** Relying solely on the public schema can lead to naming conflicts or security concerns if not managed carefully.

Best Practice Tip

In production environments, it's often a best practice to:

- **Create your own schema**, and
- **Avoid relying on the public schema** unless there's a specific reason to do so.

You can also revoke permissions on the public schema to limit who can create objects there:

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

This command ensures that only specified users or roles can create new objects in the public schema.

Summary

The public schema in PostgreSQL is the default namespace for your database objects. While it's convenient for quick development, understanding its role is key to building secure, scalable, and well-organized databases.

By taking control of your schemas, you gain flexibility, security, and clarity — whether you're working alone or as part of a large team.

Advantages and Disadvantages of the Public Schema in PostgreSQL

PostgreSQL's default public schema can feel like a developer's playground — easy to use, always available, and perfectly suited for rapid prototyping. But just like using a single folder to store all your files, relying too heavily on the public schema can introduce long-term challenges.

Let's explore both sides of this default schema to understand **when it helps and when it hurts** your database design.

Advantages of the Public Schema

1 Simplicity

When using the public schema, you don't need to worry about **schema-qualified names**. This means you can reference your database objects — like tables, views, and functions — by their name alone:

```
SELECT * FROM users;
```

Instead of needing:

```
SELECT * FROM my_schema.users;
```

This makes queries cleaner and easier to write, especially for beginners or for quick scripts.

2 Convenience

For early-stage development, the public schema is a blessing. Developers can immediately start building tables and writing queries **without needing to design a schema architecture**.

This streamlines:

- Prototyping
- Rapid feature testing
- Small or internal tools

It removes friction, allowing teams to move fast without getting bogged down in structure.

3 Compatibility

Many third-party tools, ORMs (Object-Relational Mappers), and database frameworks assume the existence of the public schema. They default to it when generating models or executing migration scripts.

So, using the public schema can **reduce configuration overhead** and prevent errors related to missing schema references.

✖ Disadvantages of the Public Schema

1 Lack of Organization

Using the public schema exclusively can turn your database into a mess as your project grows. With dozens (or hundreds) of tables in a single namespace, you lose the ability to:

- Group related objects by domain or module

- Separate production and staging data structures
- Collaborate across teams without collision

In larger systems, this disorganization leads to confusion, longer onboarding, and a higher chance of accidental modifications.

2 Security Concerns

One of the most critical drawbacks is **access control**.

By default, the public schema grants **create privileges to all database users**. This means:

- Any user can create new tables, views, or functions in your database.
- Sensitive objects could be exposed or altered unintentionally.
- You may lose track of who created what — and why.

To protect your data, you must manually **revoke public access**:

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

Without this, you risk violating least-privilege principles — an important tenet of database security.

💡 Final Thoughts

The *public schema* is like an open workspace — great for creativity and experimentation, but chaotic if used for everything. It shines during **early development** but quickly becomes a liability in **enterprise-grade applications**.

Pro Tip: Use the public schema to get started but design your own custom schemas as your project scales. It's a small shift that pays off with better organization, clearer access controls, and long-term maintainability.

🔑 Understanding the PostgreSQL Search Path

In PostgreSQL, when you write a query like `SELECT * FROM customers;` how does the database know **where** to find the `customers` table—especially if you haven't specified a schema?

That's where the **search path** comes into play.

The **search path** is a PostgreSQL configuration that defines the **order of schemas** PostgreSQL will check when looking for database objects like tables, functions, or views — if you don't explicitly specify a schema.

🔍 Why the Search Path Matters

Let's say you have the following schemas in your database:

- sales
- hr
- public

Each schema might contain a table named `customers`. So when you run:

```
SELECT * FROM customers;
```

PostgreSQL needs a way to determine **which schema's customers table** you're referring to. It does this by checking the schemas listed in the **search path, in order**, and stopping at the first match.

⚙️ Default Search Path in PostgreSQL

By default, PostgreSQL sets the search path to:

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

Let's break that down:

- “\$user”: PostgreSQL first checks for a schema named the same as the current user.
- `public`: If no match is found in the user schema, it checks the default `public` schema.

You can view the current search path for your session with:

```
SHOW search_path;
```

This tells you exactly where PostgreSQL will look (and in what order) when you reference an unqualified object.



Managing the Search Path

PostgreSQL offers a lot of flexibility in **how and where** you manage the search path. You can define it at various levels:

Level Scope Session level Affects only the current session Database level Applies to all sessions for that DB User level Affects commands run by a specific user

Let's explore each option:

1 Set Search Path for the Current Session

This is a temporary change that lasts only for the duration of your current connection:

```
SET search_path TO schema1, schema2, public;
```

Now, PostgreSQL will first check `schema1`, then `schema2`, and finally `public` for any unqualified object references.

2 Reset Search Path to Default

To return to the system-defined search path:

```
RESET search_path;
```

This removes any custom path you've set and reverts to the default: `"$user"`, `public`.

3 Set Search Path for a Specific Database

To make a search path the **default for all connections** to a database:

```
ALTER DATABASE mydb SET search_path TO schema1, schema2, public;
```

This is a good approach for multi-schema applications where object lookups need to follow a specific order by default.

4 Set Search Path for a Specific User

To define a search path that applies to a **particular user** regardless of the database they connect to:

```
ALTER USER myuser SET search_path TO schema1, schema2, public;
```

This is useful when different users are responsible for different schemas and need to access their respective objects without schema qualification.

✓ Summary

The **PostgreSQL search path** determines how the database resolves object names when no schema is specified. While the default search path works well for simple setups, you can and should customize it to suit your application's structure and security needs.

By mastering the search path, you gain more **control, clarity, and efficiency** in schema management — especially in multi-tenant systems or environments with strict data boundaries.

💡 Practical SQL Examples Using the Public Schema in PostgreSQL

Once you understand the basics of PostgreSQL schemas — especially the **public schema** — you'll want to apply that knowledge in real-world scenarios.

In this section, we'll explore **hands-on SQL commands** that help you list, move, create, and manage tables specifically in the `public` schema. These are essential tasks for database developers and administrators working with PostgreSQL.

1 List All Tables in the Public Schema

To get a complete list of tables that exist in the public schema, use the

`information_schema.tables` view:

```
SELECT table_name
FROM information_schema.tables
WHERE table_schema = 'public'
AND table_type = 'BASE TABLE';
```

📌 *What this does:*

This query filters the system's metadata to return only **base tables** (i.e., not views or temporary tables) that are part of the `public` schema. It's especially useful for quick audits and exploratory analysis.

2 Move a Table into the Public Schema

Suppose you created a table inside a custom schema (`my_schema`) but now want it under the `public` schema. You can **move the table between schemas** like this:

```
ALTER TABLE my_schema.my_table SET SCHEMA public;
```

📌 *Why this is useful:*

Restructuring your database or standardizing table locations often requires moving objects across schemas. This command keeps the table and all its data intact while reassigning its location.

3 Grant Privileges on a Table in the Public Schema

To allow another user (e.g., `my_user`) to interact with a table located in the public schema, use the `GRANT` command:

```
GRANT SELECT, INSERT, UPDATE, DELETE ON public.my_table TO my_user;
```

📌 What this grants:

- `SELECT` : Read access
- `INSERT` : Ability to add new rows
- `UPDATE` : Ability to modify existing rows
- `DELETE` : Ability to remove rows

This is essential in multi-user environments where you want to control what each user can do.

4 Revoke Privileges from a User

Just as you can grant access, you can also **take it away**:

```
REVOKE SELECT, INSERT, UPDATE, DELETE ON public.my_table FROM my_user;
```

📌 Use case:

When a user changes roles or no longer needs access, this command ensures your data is protected. It's a core part of PostgreSQL's fine-grained access control model.

5 Create a Table in the Public Schema

One of the conveniences of PostgreSQL is that you don't need to explicitly reference the public schema — **it's the default**.

```
CREATE TABLE my_table (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50)
);
```

Behind the scenes:

Even though you didn't write `public.my_table`, PostgreSQL automatically places this table inside the `public` schema unless you change your search path or explicitly specify a different schema.

Summary

The **public schema** is your default workspace in PostgreSQL. Whether you're listing tables, moving them, or managing user access, these SQL commands provide a solid foundation for hands-on PostgreSQL work.

They also help you:

- Maintain a well-organized schema structure
- Enforce data security
- Avoid confusion in large-scale environments

 **Pro Tip:** As your project grows, consider moving beyond the public schema to adopt a more modular, secure schema architecture. But for quick testing, demos, or smaller apps, these commands will serve you well.

Security Warning: Revoke Public Permissions in PostgreSQL

When it comes to securing your PostgreSQL database, one of the **most overlooked risks** is the default permission on the `public` schema.

⚠️ What's the Issue?

By default, PostgreSQL allows **every user** to create objects in the `public` schema:

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

- The keyword `PUBLIC` refers to **all database users**.
- Without revoking this, **any user** can create tables, functions, or other objects — potentially cluttering or compromising the database.

✓ Why You Should Revoke It

1. Prevent Unauthorized Object Creation

Restricts unapproved users from creating or modifying objects in shared namespaces.

2. Reduce Security Risks

Stops users from introducing rogue functions or tables that could be exploited.

3. Maintain Clean Schema Structure

Keeps the public schema reserved for only authorized and necessary objects.

🛠 Best Practice

- Revoke access as a **standard step in production deployments**.
- Grant `CREATE` only to trusted roles or specific users if needed.
- Review permissions regularly as part of your database security policy.

Securing the public schema is a **simple yet powerful way** to strengthen your PostgreSQL environment. Don't skip it.

🎯 Conclusion

The public schema is PostgreSQL's simple and friendly default — but that convenience can easily become a source of technical debt if left unmanaged.

By:

- Carefully organizing schemas,
- Managing the search path thoughtfully,
- Securing permissions,

👉 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

MySQL

Open Source

AWS

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

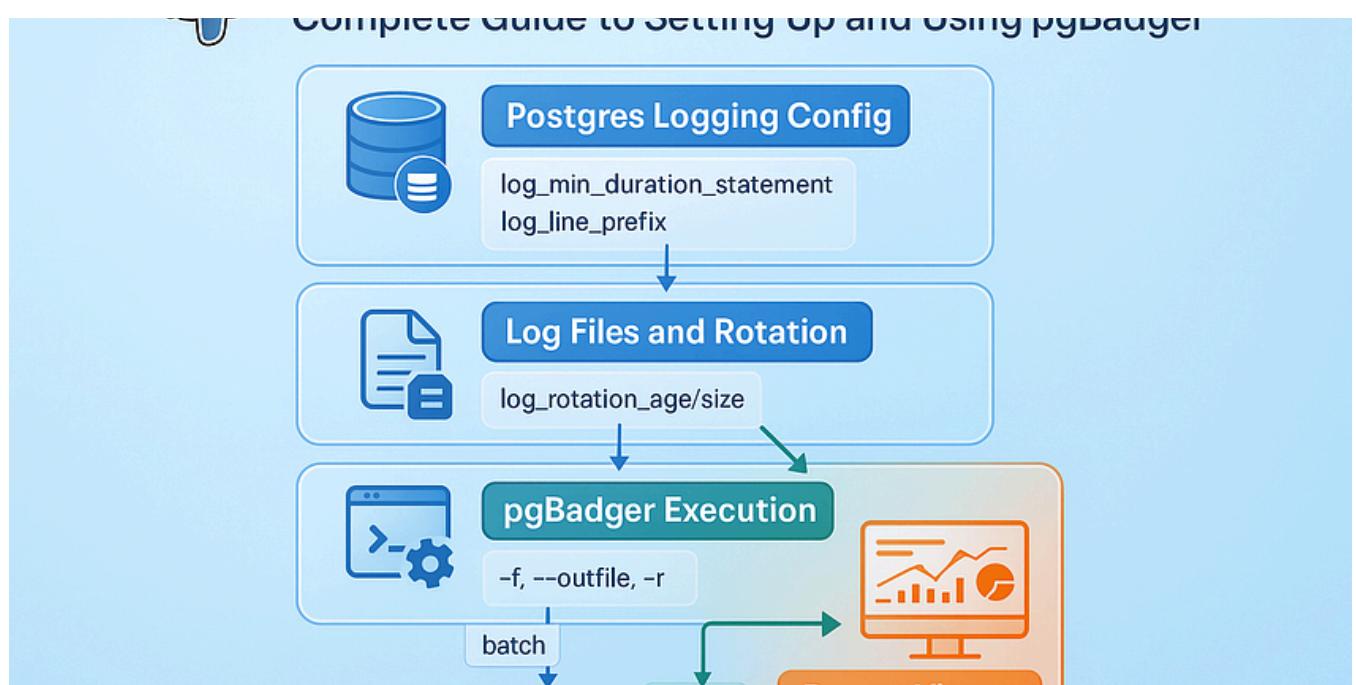
No responses yet



Gvadakte

What are your thoughts?

More from Jeyaram Ayyalusamy



Jeyaram Ayyalusamy 

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

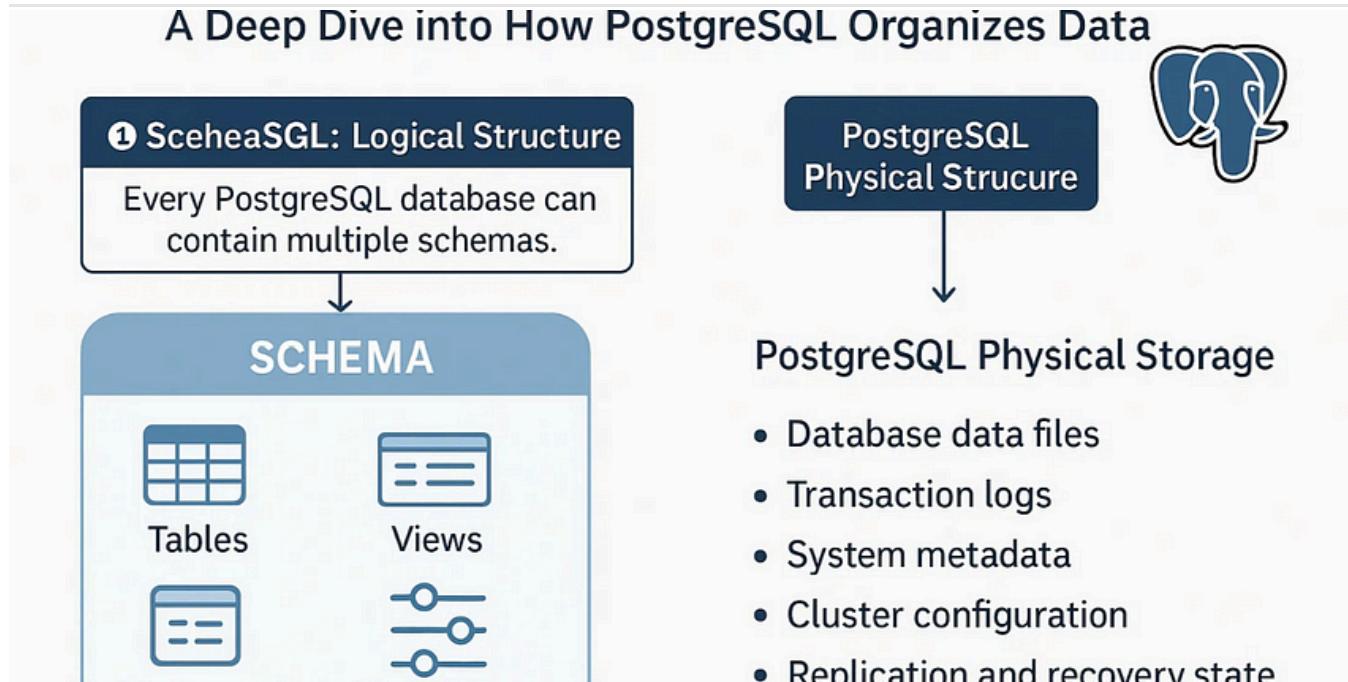
Managing a production-grade PostgreSQL instance isn't just about keeping data safe—it's about monitoring performance, identifying...

Jun 23

52



...



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



...



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 3



...



J 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

 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

...

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 and tolerance
Manage large tables efficiently	Split within one server	Time-series logs	Better performance for large tables

 Ravi Jaisawal

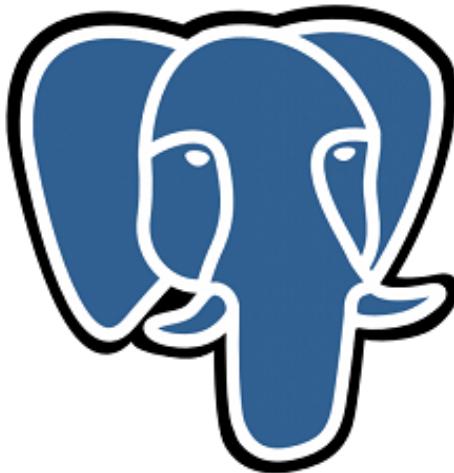
Sharding vs Replication vs Partitioning in Databases

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

Jun 11  4



...



Postare.

 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

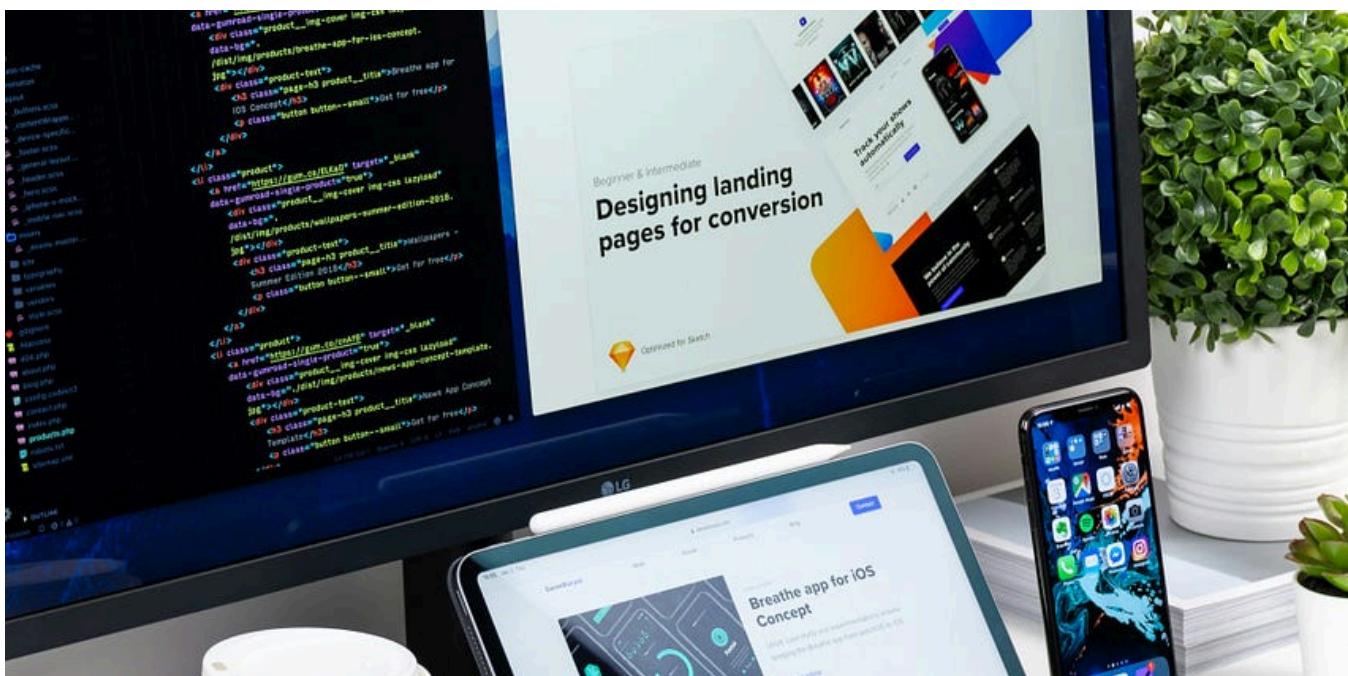


Deniz GÜRSOY

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



Sandesh I 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



 Dimitris Moutsatsos

UUID vs. BIGINT: Choosing the Right Primary Key for Your PostgreSQL Cluster

Should I use UUID or BIGINT as the primary key?

◆ Mar 1 ⌘ 18 🗣 1



See more recommendations