PostgreSQL: From Scratch to Advanced

Introduction

Welcome! If you are entirely new to databases and PostgreSQL, this guide is for you. We will start at the very beginning to build a solid foundational understanding of how PostgreSQL works, how it is installed and managed on a Linux system, what roles and privileges mean, and how all these pieces fit together.

Databases:

- A **database** is an organized collection of data, stored and accessed electronically.
- Think of it like a sophisticated spreadsheet: rows and columns, but with powerful querying, transactions, and reliability guarantees.

Imagine you're managing your personal finances using Microsoft Excel:

- Each **sheet** is like a **table** in a database.
- The **columns** in the sheet represent different types of information (e.g., "Date", "Amount", "Category").
- Each **row** is one record (e.g., one financial transaction).
- **Now imagine Excel could automatically validate data, let multiple people edit it safely at the same time, keep a log of every change, and handle millions of rows — that's the power of a database!**

Relational Databases & SQL:

In a **relational database**, data is organized into **tables**. Each table has **columns** (fields) and **rows** (records), you use **SQL** (Structured Query Language) to **CREATE**, **READ**, **UPDATE**, and **DELETE** (`CRUD`) data.

- Common relational systems are: MySQL, Oracle DB, Microsoft SQL Server — and **PostgreSQL** (which we focus here).

ACID Properties:

Relational databases strive for **ACID** guarantees:

- **Atomicity:** A transaction is a single package of work. Either the database applies every step in that package, or it applies none. There's no in-between. This is Implemented via a Write–Ahead Log (WAL) and rollback which means before changing data pages, intent is logged, on failure the log drives an automatic undo of partial updates.

eg: Imagine you're ordering a pizza with sides and a drink. If the payment for any one item fails, the whole order is canceled — no half-orders sneak through.

- **Consistency:** Every transaction must respect the database's rules (like "each user must have a unique email," or "an order can't reference a non-existent product"). When you're done, the data still follows all those rules. This ensures every committed transaction transitions the database from one valid state to another, preserving **integrity constraints** (e.g., primary/foreign keys, unique constraints, check constraints, triggers). Violations are detected at commit time, causing an automatic transaction abort.

eg: The pizzeria's system enforces its menu rules. You can't order a topping that doesn't exist or pick an invalid combo. Every order must match the menu's constraints.

- **Isolation:** If two people update the database at the same time, neither sees the other's halfway-done changes. It's as if each transaction runs alone, even when they actually overlap. This controls the visibility of concurrent transactions' intermediate states so each transaction effects appear as if executed serially.

eg: Even if hundreds of others order at once, you never see their cart updates. Your checkout behaves as if you're the only customer, so you won't accidentally add someone else's extra "soft drink".

- Durability: Once a transaction is marked "committed," its changes survive power failures or crashes. The database makes sure those changes are safely stored before saying "OK, done." This is achieved by flushing **log records** and **modified data pages** to non-volatile storage before acknowledging commit.

eg: Once the pizzeria confirms your order, it's saved — rain or shine, system crash or power flicker — they'll still make your pizza. Your confirmed order lives on their order board until it's fulfilled.

> ***Why it matters:***
> *These four guarantees together mean you can trust the database to handle your money transfers, inventory updates, or user sign-ups without ever ending up in a halfway, broken, or lost state. Just like the pizza example ensures you get exactly what you paid for, ACID properties make sure a database transaction is reliable, consistent, and permanent — even under failure or heavy load.*

Why PostgreSQL?

- **Open-Source** and Free: No licensing fees, large community.
- **Standards-Compliant**: Implements most of SQL specifications.
- **Reliability**: Used by both startups and enterprises. etc.

Operating System Integration on Linux:

System Users vs. Database Roles:

When you `apt install postgresql`, the package creates a Linux user account named **`postgres`**. This OS user owns the database server processes and data files on disk. **A database role** (or “user”) is an internal concept within PostgreSQL that manages connection permissions and object ownership.

The Linux user `postgres` and the database superuser `postgres` share a name by convention, but live in different "worlds":

1. **Linux**: owns files under `/var/lib/postgresql/...`, runs the server as a systemd service.
2. **PostgreSQL**: a superuser role, can bypass all checks inside the database.

Let’s understand this thing in depth first:

On Linux, every program that runs is owned by some operating-system user account. When you log in as “john,” your shell and every application you start inherit your user identity (your UID) and run with your permissions. You can read and write files you own, but you cannot touch files owned by other users unless those files are world-readable or you elevate your rights.

Some programs are started not by you, but by the system itself at boot time. The PostgreSQL server daemon is one of these: the system’s init service (systemd) launches it automatically under the special OS user account named `postgres`. That account exists solely to own the database’s files on disk (in `/var/lib/postgresql/...`) and to isolate the database process from other parts of the system. By running the database under a dedicated OS user, the system ensures that even if the database software

were compromised, it could only access the files and resources granted to that one user.

Inside the database, PostgreSQL maintains its own separate set of users, called **roles**. A role has nothing to do with Linux file permissions; it lives entirely within the database engine and governs what SQL commands you're allowed to execute, which tables you can read or modify, and what objects you can create or drop. When you connect with a tool like `psql`, you authenticate as one of these roles.

The command:

```
sudo -u postgres psql
```

ties these two worlds together. `sudo -u postgres` tells the Linux kernel "run the following program as the OS user `postgres`." Because the `psql` client then connects over the local Unix socket using PostgreSQL's default "peer" authentication (discussed later in this blog), the database sees that the connecting OS user is `postgres` and therefore grants you the database superuser role also named `postgres`. From that point on you are operating inside PostgreSQL itself, with full superuser privileges at the SQL level—but still under the OS user `postgres` at the system level.

Data Directory and Configuration:

- **Data Directory**: default is `/var/lib/postgresql/<version>/main` (Debian/Ubuntu). Contains all database files: tables, WAL (write-ahead log), indexes.
- **Configuration Files** (in `/etc/postgresql/<version>/main`):
- `postgresql.conf`: tuning parameters (memory, logging, listen addresses, port).

- `pg_hba.conf`: controls client authentication (which users may connect, from where, and by which methods).
- `pg_ident.conf`: optional map of OS usernames to database roles.

Initializing Your First Cluster:

`initdb` Explained:

`initdb` is the command that sets up a fresh "cluster" — a collection of databases under one server instance. On Ubuntu/Debian, this step is automated by the package installer.

What `initdb` does:

1. Creates the data directory structure.
2. Writes default config files (`postgresql.conf`, `pg_hba.conf`).
3. Initializes system catalogs (special tables PostgreSQL uses to track your objects).

Understanding the Cluster:

A **PostgreSQL cluster** is not a distributed system here — it's simply all the databases managed by one server process. You can have multiple clusters (e.g. different versions) on one machine by using different data directories and ports.

Starting, Stopping, and Managing the Service:

Use **systemd** commands on modern Linux distributions:

```
sudo systemctl start postgresql    # Start the server
sudo systemctl status postgresql   # Check if it's running and view logs
sudo systemctl stop postgresql     # Stop gracefully
sudo systemctl restart postgresql  # Stop then start
sudo systemctl reload postgresql   # Reload configuration without downtime
```

Listen Addresses: by default PostgreSQL listens only on the local Unix socket. To allow TCP connections, edit `listen_addresses` in `postgresql.conf`.

Port: default TCP port is 5432. You can change it in `postgresql.conf` under `port = 5432`.

Connecting with `psql`:

Invoking `psql`:

- As the Linux `postgres` user:

```
sudo -u postgres psql
```

Switches to OS user `postgres` and runs the `psql` client, which by default connects to the `postgres` database. The prompt shows as `postgres=#`, indicating you are superuser inside the `postgres` database.

`psql` Meta-Commands:

A quick reference: (These commands work when you see the prompt similar to `postgres=#`)

- `\l` – List all databases.
- `\c <dbname>` – Connect to database `<dbname>`.
- `\du` – List all database roles.
- `\dn` – List schemas inside current database.
- `\dt` – List tables in current schema.
- `\d <table>` – Describe columns, indexes, constraints of `<table>`.
- `\?` – Help on psql commands.

All commands starting with a backslash (`\`) are specific to `psql` and are not SQL.

Databases, Schemas, and Roles, The Three Pillars:

Think of your database system like a big library, and the pieces break down like this:

Database:

- A **database** is a named collection of schemas and objects (tables, indexes, etc.).
- You cannot cross-query from one database to another inside the same session.

Analogy: Database is like one entire library branch. It contains all its own collections, rooms, catalogs, rules, etc. You can't wander into another branch (another database) without leaving and "checking in" there.

Schemas

- A **schema** is a namespace inside a database. The default schema is named `public`.
- Schemas allow logically grouping objects (e.g., `analytics`, `sales`, `hr`)

Analogy: A **schema** is like one section or floor in the library (e.g. Fiction, Non-Fiction, Reference). It's just a way to **group** related things together so they don't get mixed up. Most systems give you a default section called `public` where you start.

Roles (Users & Groups)

- A **role** can be a user (with `LOGIN` privilege) or a group (without `LOGIN`).
- Roles can own objects and have privileges granted.

- **Superuser** roles bypass all privilege checks

Analogy: A role is like a library card that grants you certain rights:

- **User-roles** (with LOGIN) are actual people who can check out books.
- **Group-roles** are like "Staff" or "Researchers" cards — collections of people you can give the same set of rights to.
- Some roles (superusers) are like head librarians: they can roam every section, override rules, and read or change any book.

Analogies for more topics:

Table, A Bookshelf:

- A **table** is a single bookshelf within a section. On that shelf you keep books that all share the same format — e.g. every book on the shelf is about "Customers," or every book is about "Orders."

Row & Column, Book and Its Fields:

- Each **row** is one book on that shelf — one individual record (one customer, one order). Each **column** is a particular attribute of that book — like "Title," "Author," "ISBN," or in database terms "CustomerName," "SignupDate," "Email."

Index, The Catalog Card:

- An **index** is like the little catalog cards you use to find a book faster. Instead of scanning every spine on the shelf, you look in the index and it tells you exactly which shelf (table) and position (row) to go to.

The database operations looks similar to these processes:

- You enter the **library building** (connect to a database).
- You move to the **section** you need (choose a schema).

- You go to the right **bookshelf** (open a table).
- You pull out **books** (rows) and read the **catalog** (index) to find things quickly.
- Your **library card** (role) determines which sections you can enter and which books you can touch.

Creating Your Own Database and Role:

Create a Database

In `psql` as superuser:

```
CREATE DATABASE mydb;
```

This creates a new, independent database named `mydb`. Under the hood, PostgreSQL clones the **template database** `template1` , **template1** is a special database containing the default structure (schemas, extensions, tables, etc.) that each new database inherits. You can customize `template1` (for example, by installing extensions or creating utility objects), and those changes will be included in any subsequently created databases. This cloning process ensures your new database starts with a consistent environment matching `template1`. (usually `template1`).

Creating a role:

```
CREATE ROLE myrole
  WITH LOGIN
       PASSWORD 'mypwd'
       NOSUPERUSER
       NOCREATEDB
       NOCREATEROLE;
```

Here:

- **CREATE ROLE:** Make a new role (a database account).
- **myrole**: The name of this role — your application will connect as this.
- **WITH LOGIN:** Allows this role to **log in/authenticate** (i.e. open a DB connection).
- **PASSWORD '...':** Sets its login password (what it must supply to authenticate) . PostgreSQL **hashes** this behind the scenes.
- **NOSUPERUSER: Disables** superuser powers — this role can't override security or do admin tasks.
- **NOCREATEDB:** Prevents it from creating new databases.
- **NOCREATEROLE:** Prevents it from creating, altering, or granting other roles.

Authentication: `pg_hba.conf:`

File Location

- Typically located at: `/etc/postgresql/<version>/main/pg_hba.conf` in ubuntu.

File Purpose and Structure

This file controls who can connect to which database, from where, and using what authentication method.

Each line in `pg_hba.conf` follows the format:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
```

Example Entries:

```
# Allow local connections using peer authentication
local   all             postgres                                peer

# Allow myrole user to connect to mydb database locally with password
local   mydb            myrole                                  md5

# Allow remote connections from a specific IP range using password authentication
host    mydb            myrole      192.168.1.0/24          md5

# Allow all users from local network to connect to all databases using SCRAM
host    all              all             10.0.0.0/8              scram-sha-256
```

Explanation of Columns

TYPE:

- `local`: Unix-domain socket (for same-machine connections).
- `host`: TCP/IP.
- `hostssl`: TCP/IP over SSL.
- `hostnossl`: TCP/IP without SSL.

DATABASE: Name of database(s), or `all`.

USER: Role(s) allowed to connect.

ADDRESS: For `host` types, the IP address range allowed (e.g. `192.168.1.0/24`). Ignored for `local`.

METHOD: Authentication method (see below).

Authentication Methods

- `trust`: No password needed (insecure).
- `peer`: OS user must match PostgreSQL user (only for `local`).
- `md5`: Password encrypted using MD5 hash.
- `scram-sha-256`: More secure password method.

Important Notes and Pitfalls

- **Order matters!** PostgreSQL checks each line from top to bottom and uses the first match.
- **Be careful with `trust`** — it can expose your database if misused.
- If your changes don't work, double-check IP addresses, role names, and the method.
- **Always reload PostgreSQL** after editing the file:

```
sudo systemctl reload postgresql
```

- Or in `psql`:

```
SELECT pg_reload_conf();
```

Granting Privileges:

Think of privileges as "keys" that let your role do certain things at different "locations" in the database world. We'll build up from the biggest scope (the whole database) down to individual objects, and then show how to make future objects inherit the same permissions.

Database-Level Privileges:

These control what you can do with *the entire database* named `mydb`.

```
GRANT CONNECT   ON DATABASE mydb TO myrole;
GRANT TEMPORARY ON DATABASE mydb TO myrole;
```

Now let's understand what every privilege is, what it means and its analogy:

- CONNECT: Role can open a session/connection to that database.

Analogy: You have the library's front-door key.

- TEMPORARY: Role can create *temporary* tables in that database session.

Analogy: You can set up folding tables for a one-day event inside the library, but they vanish when you leave.

Why both?

- Without CONNECT, your app can't even "step inside."
- Without TEMPORARY, your app can't use temp tables for caching or intermediate work.

Schema-Level Privileges:

Once inside the database, schemas are like sections (e.g. `public`). You need rights to walk around and build there.

```
-- First switch into that database (psql example)
\c mydb

GRANT USAGE  ON SCHEMA public TO myrole;
GRANT CREATE ON SCHEMA public TO myrole;
```

What each privilege is, what it means:

USAGE: Role can *see* and *refer* to objects (tables, sequences, functions) in that schema.

CREATE: Role can *create* new objects (tables, functions) in that schema.

- **Sequence of steps:**

1. CONNECT to the database.
2. USAGE lets you run queries that mention `public.customers` or `public.orders`.
3. CREATE lets you do `CREATE TABLE public.logs (…)`.

Object-Level Privileges:

These control what you can do with *each* table, sequence, or function inside the schema.

```
GRANT ALL PRIVILEGES    ON ALL TABLES    IN SCHEMA public TO mrole;
GRANT ALL PRIVILEGES    ON ALL SEQUENCES IN SCHEMA public TO myrole;
GRANT EXECUTE           ON ALL FUNCTIONS IN SCHEMA public TO myrole;
```

Tables

- **ALL PRIVILEGES** = `SELECT, INSERT, UPDATE, DELETE, TRUNCATE, REFERENCES, TRIGGER`
- **SELECT**: read rows
- **INSERT**: add new rows
- **UPDATE**: change existing rows
- **DELETE**: remove rows
- **TRUNCATE**: quickly empty the table
- **REFERENCES**: use table's keys in foreign-key constraints
- **TRIGGER**: create triggers on the table

> Analogy: You not only can borrow books (SELECT), but also donate new ones (INSERT), correct typos inside books (UPDATE), remove old volumes (DELETE), rearrange by tossing all at once (TRUNCATE), link books to one another (REFERENCES), and attach little sticky-notes that fire off events (TRIGGER).

Sequences:

Before understanding these privileges let's first deep dive into sequences:

What's a Sequence?

A **sequence** is simply a database object whose job is to hand out ever-increasing numbers — one at a time — whenever you ask. You use it most often to generate unique IDs for new rows, like order numbers or invoice IDs.

How you use a sequence in SQL

1. **Create one:**

```
CREATE SEQUENCE invoice_id_seq
  START 1        -- begin at 1
  INCREMENT 1;   -- go up by 1 each time
```

2 Get the next number:

```
SELECT nextval('invoice_id_seq');
-- returns 1, then next call returns 2, then 3, etc.
```

3. Peek at the current number:

```
SELECT currval('invoice_id_seq');
-- shows the last number you received in this session
```

4. Reset or jump:

```
SELECT setval('invoice_id_seq', 100);
-- now the next nextval() will give 101
```

Privileges on sequences

- ALL PRIVILEGES = `USAGE, SELECT, UPDATE`
- USAGE → You can call `nextval()` to *take* a number.
- SELECT → You can call `currval()` to *see* what number was last handed out.
- UPDATE → You can call `setval()` to *reset or jump* the counter.

Analogy: A sequence is like a numbered ticket dispenser. These rights let you take a ticket (USAGE), look at the last number called (SELECT), or reset the counter (UPDATE).

Functions

What's a Function?

A **function** in the database is a little block of code — like a mini-program — stored in the database itself. You call it by name (just like you call a function in your application code) and it can:Take inputs (parameters)

- Run SQL or procedural logic
- Return a result (a value or a set of rows)
- Optionally make changes to the database

You create them when you have a piece of logic you'll need many times, or when you want to keep complex calculations centralized.

Example:

```
-- Define a function that adds 10% tax
CREATE FUNCTION add_tax(price numeric)
  RETURNS numeric AS $$
BEGIN
  RETURN price * 1.10;
END;
$$ LANGUAGE plpgsql;

-- Use it:
SELECT add_tax(200);  -- returns 220
```

Privilege on functions

- **EXECUTE** → Allows you to *call* (run) the function.

> **Analogy:**
> Think of a database function as a coffee machine button:
>
> *The button **is** the function (e.g. "Espresso").*
>
> *Pressing it tells the machine to run its internal routine (grind beans, force water through).*
>
> ***EXECUTE** permission is simply the right to press that button.*

Bringing it together

When you do:

```
GRANT ALL PRIVILEGES ON ALL SEQUENCES IN SCHEMA public TO mrole;
```

you're giving your `myrole` role the rights to **take**, **peek**, and **reset** any of those number-generators (sequences).

And

```
GRANT EXECUTE ON ALL FUNCTIONS IN SCHEMA public TO myrole;
```

you're letting it **call** every function you've defined there — just like letting your application press the buttons on any "machine" you've set up inside the database.

Default Privileges for Future Objects:

```
ALTER DEFAULT PRIVILEGES IN SCHEMA public
  GRANT ALL PRIVILEGES ON TABLES    TO myrole;
ALTER DEFAULT PRIVILEGES IN SCHEMA public
  GRANT ALL PRIVILEGES ON SEQUENCES TO myrole;
```

Ensures any new tables/sequences you create later automatically grant the same rights and it remember defaults apply to objects created by the role running the `ALTER` command.

Working with Data: SQL CRUD:

1. **INSERT** data:

```
INSERT INTO public.users (name) VALUES ('Alice') RETURNING id;
-- Returns the generated ID, e.g. "id: 1"
```

2. SELECT data:

```
SELECT id, name FROM public.users;
-- Example output:
--  id | name
-- ----+-------
--   1 | Alice
```

3. **UPDATE** rows:

```
UPDATE public.users SET name = 'Bob' WHERE id = 1 RETURNING *;
-- Example output:
--  id | name
-- ----+------
--   1 | Bob
```

4. **DELETE** rows:

```
DELETE FROM public.users WHERE id = 1 RETURNING *;
-- Example output:
--  id | name
-- ----+------
--   1 | Bob
```

Transactions:

What's a transaction?

A **transaction** is like a little "all-or-nothing" box you use to group a series of steps. Either **every** step in the box happens, or **none** of them do. This keeps your data safe and consistent.

Analogy: Two Envelopes of Cash:

Imagine you have two envelopes labeled **A** and **B**, each with some money inside:

- Envelope A: $500
- Envelope B: $300

You want to move $100 from A to B. That's two steps:

1. Take $100 out of Envelope A
2. Put $100 into Envelope B

If you only do step 1 (take out from A) and then something goes wrong (you drop the cash, or the power goes out), Envelope A is short $100 and Envelope B never got it — that's bad!

So you decide to use a "transaction box":

1. **BEGIN** (you close both envelopes inside your transaction box)
2. **Step 1:** Remove $100 from A (still inside the box)
3. **Step 2:** Add $100 to B (still inside the box)
4. **COMMIT** (you open the box and let both changes stick)

If something goes wrong **before** you commit (say you changed your mind, or spilled coffee), you can do a **ROLLBACK**:

- **ROLLBACK** = "never mind, undo everything inside the box" — both envelopes go back to their original amounts.

Mapping to SQL:

```
BEGIN;
  -- Start the transaction box

UPDATE accounts
   SET balance = balance - 100
```

```
 WHERE id = 1;
   -- Step 1: take $100 out of account #1

UPDATE accounts
   SET balance = balance + 100
 WHERE id = 2;
   -- Step 2: add $100 into account #2

COMMIT;
  -- If both updates succeeded, make them permanent
```

- **BEGIN;**
  Tells the database: “Start grouping my next commands into one atomic transaction.”
- **COMMIT;**
  Tells the database: “Everything in that group worked — go ahead and save it all.”
- **ROLLBACK;**
  (If you call this instead of COMMIT) tells the database: “Something went wrong — undo every change in that group. Leave the data exactly as it was before the BEGIN.”

System Catalogs (`pg_catalog`)

PostgreSQL has internal tables like `pg_class`, `pg_attribute`, `pg_roles`. Use them if you need deep introspection.