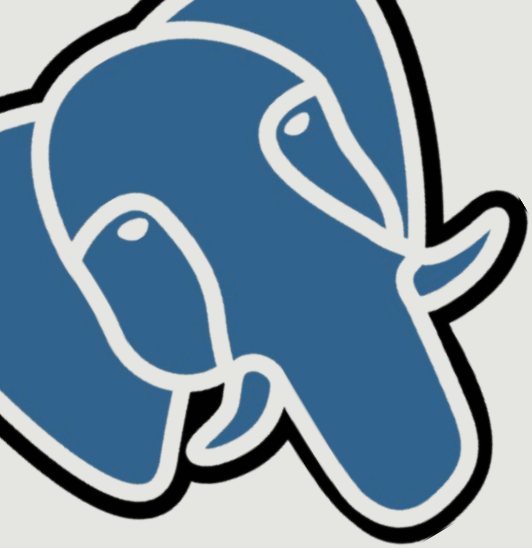


PostgreSQL

The complete course

WELCOME ONBOARD

From this moment on, you're not just a comrade you're my brother-in-arms. Shoulder to shoulder, we fight. No matter how dark the battlefield gets, remember we don’t back down, we don’t break.
We move forward. Together.
Your strength is my strength.
Your fight is my fight.
Let’s write history not as survivors, but as warriors who never gave up.

PHASE 1

1) What is Database ?

2)What is RDBMS ?

3) postgreSQL vs mySQL vs SQlite Why choose postgreSQL?

4)Installation of PgAdmin or Dbeaver.

5)Basic SQL syntax.

6)Understanding Database, Schema , Table, row, column.

WHAT IS DATABASE

A **Database** is simply a collection of organized data that can be easily accessed, managed, and updated.

Think of it like a digital version of a *well-organized notebook* , where each page has information written in a structured way so that you can search, update, and retrieve it easily.

WHAT IS DATABASE

In more technical terms:

- It **stores data** electronically.
- The data is organized in **tables** (like Excel sheets with rows and columns).
- You can **query** the data using a language (usually SQL) to read, insert, update, or delete the data.
- The database system ensures **consistency, reliability, and security** of the data.

WHAT IS DATABASE

Let’s say you’re building an app to store information about students:

Student_id	Name	Age	Grade
1	Akarsh	20	A
2	Anjali	21	B
3	Raj	22	A

This table is stored inside a database — and you can ask the database:

- Show me all students with grade A.
- Add a new student.
- Update Anjali’s age.
- Delete Raj’s record.

WHAT IS RDBMS

RDBMS stands for **Relational Database Management System**.

Relational	Data is organized into **tables** (which are related to each other).
Database	A collection of structured data.
Management System	Software that allows you to create, read, update, and delete data (using SQL).

WHAT IS RDBMS

An RDBMS is **software that helps you store data in tables, relate them to each other, and perform operations on them using SQL.**

WHAT IS RDBMS

KEY FEATURES OF RDBMS

1) **Data is stored in tables (relations)**

- Each table has rows and columns (like an Excel sheet).

2) **Relationships between tables**

- Example:
  - You have a `Students` table
  - You have a `Courses` table
  - You can link them with an `Enrollments` table

WHAT IS RDBMS

EXAMPLES OF RDBMS

- **PostgreSQL** (the one you are going to learn!)
- **MySQL**
- **SQLite**
- **Oracle Database**
- **SQL Server**

WHY POSTGRESQL

When learning databases, people often ask: why PostgreSQL, when MySQL and SQLite are also there? Let's understand the differences clearly so you know why you are learning PostgreSQL

WHY POSTGRESQL

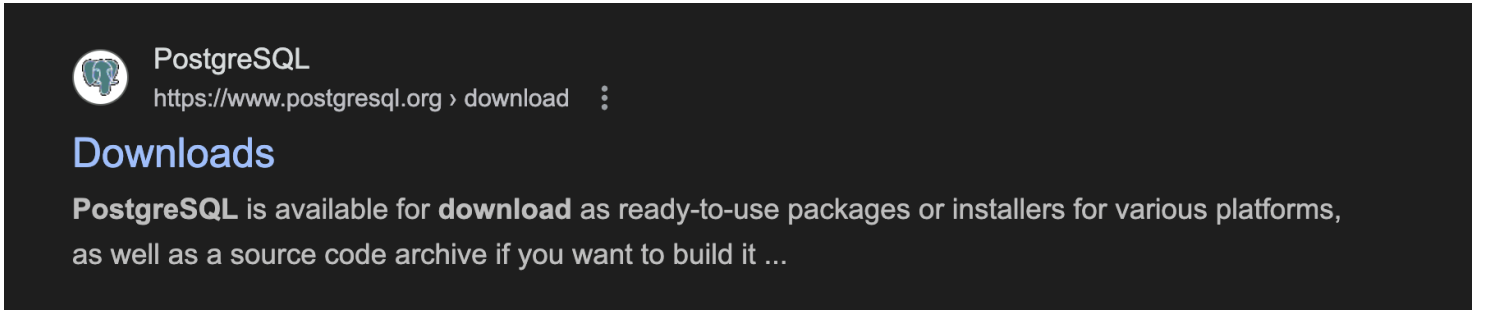
- Open Source
- Most feature-rich & powerful
- **Supports complex queries**, joins, CTEs, window functions
- **ACID compliant**
- **Highly extensible** (you can add your own data types, operators, etc.)
- **Support for NoSQL** features also (JSONB, hstore)
- Used for **enterprise-grade** applications

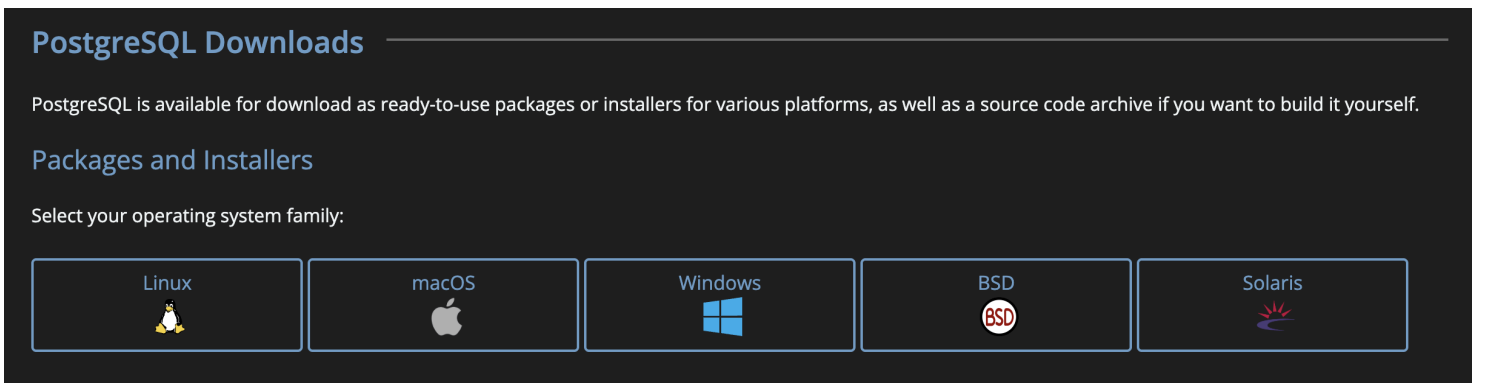
WHY POSTGRESQL

Type	Full RDBMS	RDBMS	Lightweight DB
Open Source	Yes	Yes	Yes
Complex Queries	Excellent	Good	Basic
Performance	Excellent	Fast on simple	Great for small
NoSQL Features	Yes (JSONB)	Limited	No
Concurrency	Strong	Medium	Weak
Use Case	Enterprise, big data	Web apps	Mobile apps, testing

INSTALLATION

Now installation is also easy you just have to go to browser and search postgreSQL download and click the official link of postgreSQL. And click on whatever operating system you have.



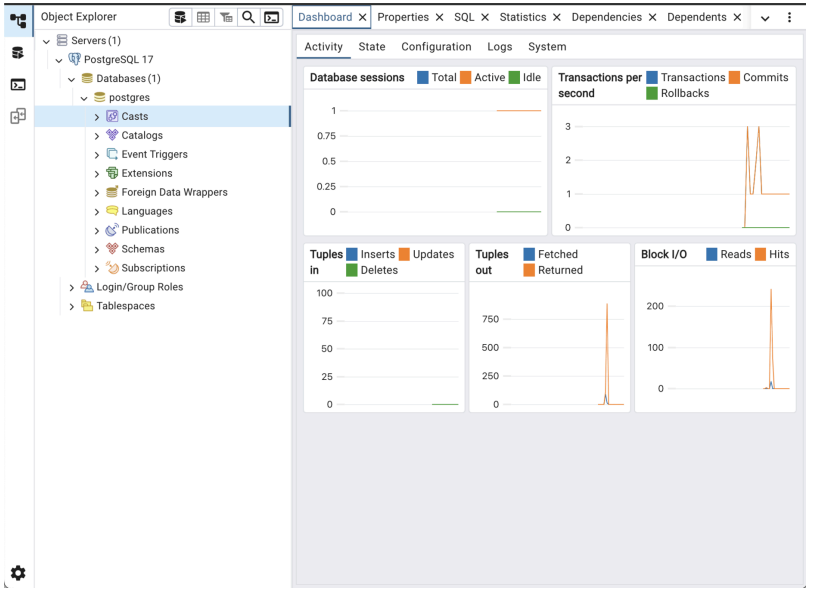


INSTALLATION

During installation you had to set up a password make your password. and after that you will have the PG admin installed on your system.

This is your pgAdmin will look like after clicking on the server and entering password.

→



INSTALLATION

now you must be thinking what is this pgAdmin.

pgAdmin is a **graphical user interface (GUI) tool** for managing PostgreSQL databases.

In simple words:

- PostgreSQL is the **engine** (it stores and manages the data)
- pgAdmin is like a **dashboard** or **control panel** — it gives you a friendly interface to work with PostgreSQL, instead of writing everything in the command line.

Wait we can use command line as well...

INSTALLATION

After installation you can also search for psql and you will get the sql shell for postgreSQL.

but to use it you may need to add the path of postgreSQL

The `psql` command (and other PostgreSQL tools) live inside a `bin` folder — so to use them in **CLI (command-line interface)**, you have to make sure your **PATH variable** includes that folder.

well look the video for clearly understanding both mac and windows are covered.

INSTALLATION

For using the shell you have to write psql -U postgres

“ *`psql: command not found`*

...it means your terminal doesn't know where to find the `psql` binary. You’ll need to **add PostgreSQL’s `bin` directory to your system’s PATH**.

echo 'export PATH="/Library/PostgreSQL/17/bin:$PATH"' >> ~/.zprofile
source ~/.zprofile

and after this use psql postgres

INSTALLATION

basic commands select a database

\l -> see the list of all the databases

\c dbname -> this will connect you to a database

CREATE DATABASE testdb;

\c test_db

WHAT IS IMPORTANT

DATABASE

- A **Database** is like a **big container** that stores all your data.
- Inside 1 PostgreSQL server, you can create **multiple databases**.
- Each database is **isolated** — data in one database is not mixed with another.

WHAT IS IMPORTANT

SCHEMA

- Inside a database, you can organize objects using **schemas**.
- Think of a **schema as a folder** — it helps organize your tables, views, functions, etc.
- By default, every PostgreSQL database has a schema called **public**.

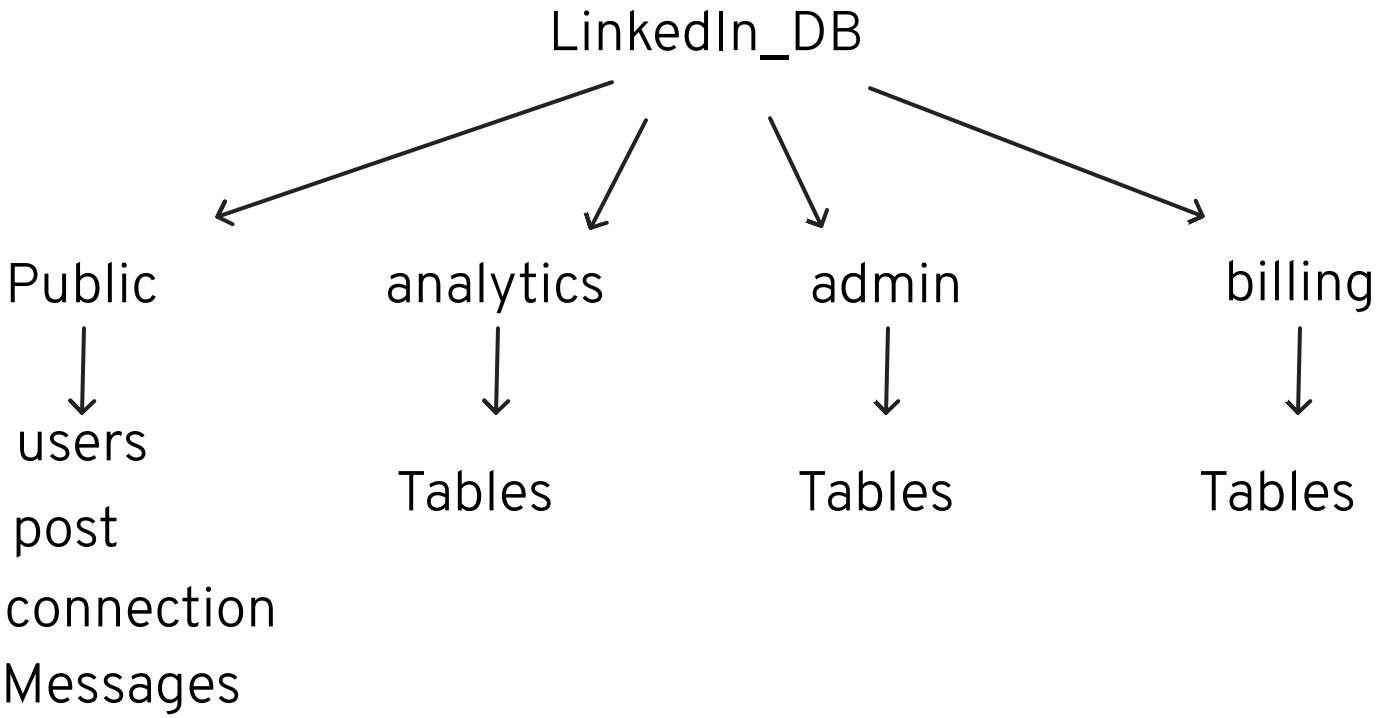
WHAT IS IMPORTANT

TABLE

- A **table** is the main place where data is stored.
- Think of a table like an **Excel sheet** — rows and columns.
- Each table has:
  - **columns** → define the type of data (name, age, salary)
  - **rows** → each record (one person, one product, one order)

WHAT IS IMPORTANT

How linkedin stores data inside the postgreSQL. example



PHASE 2

Basic Commands

CRUD operations

BASIC USE AND CRUD

ok now first of all lets see how you can create the database for creating you just need to write CREATE and the name of your database. just run this query on the query tool of pgadmin.

```
CREATE DATABASE new_db;
```

You can also create a new database using command line see the video for understanding.

BASIC USE AND CRUD

Ok so you have created a database but what actually matters is table where you can store the data so now lets actually create a table.

To create the table you need to write a simple query like this.

```
CREATE TABLE students (
    student_id INT,
    name char(50),
    age INT,
    grade CHAR(1)
);
```

Here you are creating the table and making 4 columns - student_id, name, age, grade and after that you are specifying the data types we will learn about them later.

BASIC USE AND CRUD

Ok so you have created a database but what actually matters is table where you can store the data so now lets actually create a table.

To create the table you need to write a simple query like this.

```
CREATE TABLE students (
    student_id INT,
    name char(50),
    age INT,
    grade CHAR(1)
);
```

Here you are creating the table and making 4 columns - student_id, name, age, grade and after that you are specifying the data types we will learn about them later.

BASIC USE AND CRUD

ok so database is created and table is ready now we also have to fill the table cause there are only columns so for that a very simple insert query will do.

```
INSERT INTO students (name, age, grade)
VALUES ('Akarsh', 21, 'A'),
       ('Anjali', 22, 'B');
```

Kind of works like parameters and arguments. and again everything can be used in CLI aswell.

BASIC USE AND CRUD

Ok now the database and table is created now its time to read the table. For that a universal line 🥹

```
SELECT * FROM students;
```

This will display all the columns of your data, and for specific column you can provide the name of that column.

```
SELECT name FROM students;
```

BASIC USE AND CRUD

you can also do the conditional selection using the WHERE keyword in query.

```
SELECT * FROM students WHERE grade = 'A';
```

ok thats it we will see more functions later but this is how you have to read all the data.

BASIC USE AND CRUD

Now lets see how to update the data in table. you can update any value of your table just by using UPDATE, SET and WHERE keyword.

```
UPDATE students
SET age = 23
WHERE name = 'Akarsh';
```

BASIC USE AND CRUD

So now lets do one thing and update the Student_id and gvie them 1 and 2.

```
UPDATE students
SET student_id = 1
WHERE name = 'Akarsh';
UPDATE students
SET student_id = 2
WHERE name = 'Anjali';
```

BASIC USE AND CRUD

After this now we can delete aswell using any column and the row will be removed.

```
DELETE FROM students
WHERE name = 'Anjali';
```

THE PROBLEM

Yes you were doing everything the right way but there are some problems lets see them first and understand.

THE PROBLEM

1) You already saw we can have null values with us.

2) The values can be repeated, which needs to be unique.

3)Wrong Data Type Values.

and the list goes on ...

Thats why we have to learn and fix somethings.

PHASE 3

Appropriate Data Type

constraints

DATA TYPES

If we talk about SQL we have multiple datatypes these include.

1) Numeric Data Types

2) Character/String Data Types

3) Boolean Type

4)Date and Time Types

Numeric Data Types

`SMALLINT`	2 bytes integer (-32,768 to 32,767)	`age SMALLINT`
`INTEGER / INT`	4 bytes integer (-2B to 2B)	`quantity INT`
`BIGINT`	8 bytes integer	`views BIGINT`
`DECIMAL(p, s) / NUMERIC(p, s)`	Exact precision	`price NUMERIC(8,2)`
`REAL`	4-byte floating point	`rating REAL`
`DOUBLE PRECISION`	8-byte floating point	`accuracy DOUBLE PRECISION`
`SERIAL`	Auto-increment integer	`id SERIAL PRIMARY KEY`

Character/String Data Types

Data Type	Description	Example
`CHAR(n)`	Fixed-length string (pads with space)	`code CHAR(5)`
`VARCHAR(n)`	Variable-length string (limit n chars)	`email VARCHAR(100)`
`TEXT`	Unlimited-length string	`bio TEXT`

Boolean Type

`BOOLEAN` TRUE, FALSE, NULL `is_active BOOLEAN`

Date and Time Types

`DATE`	Only date(2025-07-05)	`dob DATE`
`TIME`	Only time(14:30:00)	`login_time TIME`
`TIMESTAMP`	Date + time	`created_at TIMESTAMP`
`TIMESTAMPTZ`	With timezone	`event_at TIMESTAMPTZ`
`INTERVAL`	Time difference	`duration INTERVAL`

Constraints

`PRIMARY KEY`	Uniquely identifies each row	`id SERIAL PRIMARY KEY`
`NOT NULL`	Column must have a value	`name TEXT NOT NULL`
`UNIQUE`	No duplicate values allowed	`email TEXT UNIQUE`
`DEFAULT`	Provides default value if none	`created_at TIMESTAMP DEFAULT now()`
`CHECK`	Validates values	`age INT CHECK (age > 18)`
`FOREIGN KEY`	Links one table to another	`user_id INT REFERENCES users(id)`

WAIT RIGHT HERE

wait my fellow soldier...

You now know what are datatypes and constraints but what about implementing it and practicing it.

Pause right here and do the exercise cause you have to clear the basics before moving forward. ❤️‍🔥

I trust you brother you will achieve success.

SMALL PROJECT

Create a Database named Flipkart_db and then create a table with .

Product ID - Serial

name - String

sku_code 8 digit - String

Price max (99999999) - number

Stock_Quantity must be greater than 0- Number

Is available default true -Boolean

Category not null - String

Added_on - Date

Last_update - Date

SMALL PROJECT

Now while creating think about what should be the appropriate datatype and constraints.

SOLUTION

I think you can create the database yourself after that just create the Table with Data types and constraints.

```sql
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    sku_code CHAR(8) UNIQUE NOT NULL,
    price NUMERIC(10,2) CHECK (price > 0),
    stock_quantity INT DEFAULT 0 CHECK (stock_quantity >= 0),
    is_available BOOLEAN DEFAULT TRUE,
    category TEXT NOT NULL,
    added_on DATE DEFAULT CURRENT_DATE,
    last_updated TIMESTAMP DEFAULT now()
);
```

SOLUTION

after creating the table just insert the data and don't provide the value to the primary key as it will auto increment.

```sql
INSERT INTO products (name, sku_code, price, stock_quantity, category)
VALUES
('Wireless Mouse', 'MOU12345', 799.99, 120, 'Electronics'),
('Yoga Mat', 'YOG98765', 499.00, 50, 'Fitness'),
('LED Desk Lamp', 'LED11223', 999.50, 200, 'Home Decor');
```

SOLUTION

ok thats it now you know how to make a clean database using appropriate datatypes and constraints. But first lets discuss some of the problems you may face with datatype and constraints.

SOLUTION

1) If you have the SERIAL aka auto increment don't manually provide it see the demonstration.

2. `sku_code` not 8 characters long.

3.Using commas or ₹ in `price` (like '₹799').

4. Providing wrong data types...

PHASE 3

Clauses

Operators and Aggregation functions

CLAUSES

Clauses are mainly used for querying the data and are kind of building blocks even you all are using them from the start.

CLAUSES

`SELECT`	Choose which columns to display
`FROM`	Specify the table
`WHERE`	Filter rows based on a condition
`GROUP BY`	Group rows for aggregation
`HAVING`	Filter aggregated groups (used after `GROUP BY`)
`ORDER BY`	Sort the result in ascending or descending order
`LIMIT`	Limit the number of rows returned
`AS`	Rename columns or tables temporarily (aliasing)
`DISTINCT`	Return only unique/distinct values

CLAUSES

But understanding them like this is tough so what we do now soldier...

I have a plan lets apply everything on the table we have created I will provide you some set of questions and lets solve them one by one using clauses.

CLAUSES TEST

Q1. Show the name and price of all products.

Q2. Show all products where the category is 'Electronics'.

Q3. Group products by category. Show each category once.

Q4. Show categories that have more than 1 product. *(Use after GROUP BY)*

Q5. Show all products sorted by price in ascending order.

Q6. Show only the first 3 products from the table.

Q7. Show product name as "Item_Name" and price as "Item_Price".

Q8. Show all the unique categories from the products table.

CLAUSES SOLUTIONS

Q1. Show the name and price of all products.

```sql
SELECT name, price FROM products;
```

CLAUSES SOLUTIONS

Q2. Show all products where the category is 'Electronics'.

```sql
SELECT * FROM products WHERE category = 'Electronics';
```

CLAUSES SOLUTIONS

Q3. Group products by category. Show each category once.

```sql
SELECT category FROM products GROUP BY category;
```

CLAUSES SOLUTIONS

Q4. Show categories that have more than 1 product. *(Use after GROUP BY)*

```
SELECT category, COUNT(*) FROM products
GROUP BY category
HAVING COUNT(*) > 1;
```

CLAUSES SOLUTIONS

Q5. Show all products sorted by price in ascending order.

```sql
SELECT * FROM products ORDER BY price ASC;
```

CLAUSES SOLUTIONS

Q6. Show only the first 3 products from the table.

```sql
SELECT * FROM products LIMIT 3;
```

CLAUSES SOLUTIONS

Q7. Show product name as "Item_Name" and price as "Item_Price".

```sql
SELECT name AS Item_Name, price AS Item_Price FROM products;
```

CLAUSES SOLUTIONS

Q8. Show all the unique categories from the products table.

```sql
SELECT DISTINCT category FROM products;
```

CLAUSES WITH OPERATORS

So now you definitely got to know about all the major SQL clauses — like `SELECT`, `FROM`, `WHERE`, `ORDER BY`, `LIMIT`, *and more. These are primarily used to select and view data from the table."*

But sometimes, just selecting is not enough.

We want to make our selections more powerful — like:

- Filter products greater than a certain price
- Search names that start with a certain letter
- Count how many products are in each category
- Or even find out the most expensive item per group

That's where Operators and Aggregation Functions come into play.

CLAUSES WITH OPERATORS

Now if you have the knowledge of a language you might know about operators.

Some of the basic operators are :

- Comparison (=, !=, <, >, <=, >=)
- Range (`BETWEEN`)
- Set (`IN`)
- Pattern (`LIKE`)
- Logical (`AND`, `OR`, `NOT`)

AGGREGATION FUNCTIONS

Aggregation functions are used to **summarize data**.
Instead of looking at every single row, we use these functions to get **overall insights**, like:

- **How many rows are there?**
- **What's the total price?**
- **What's the average stock?**
- **What's the max/min value in a column?**

AGGREGATION FUNCTIONS

`COUNT()`	Count number of rows	Total number of products
`SUM()`	Add numeric values	Total stock in a category
`AVG()`	Calculate average	Average price of accessories
`MIN()`	Find smallest value	Cheapest product
`MAX()`	Find highest value	Most expensive product

TEST 2

Alright you beast 😈
You've mastered **Clauses**, **Operators**, and **Aggregation Functions** now it's time to **face the fire** 🔥 .

TEST 2

Q1. Display the **name and price** of the **cheapest product** in the entire table.

Q2.Find the **average price of products** that belong to the **'Home & Kitchen'** or **'Fitness'** category.

Q3. Show product names and stock quantity where the product is available, stock is more than 50, and price is **not equal** to ₹299.

Q4. Find the **most expensive product** in each category (name and price).

Q5. Show all unique categories **in uppercase**, sorted in descending order.

SOLUTIONS

Q1. Display the **name and price** of the **cheapest product** in the entire table.

```sql
SELECT name, price FROM products
WHERE price = (SELECT MIN(price) FROM products);
```

SOLUTIONS

Q2.Find the **average price of products** that belong to the **'Home & Kitchen'** or **'Fitness'** category.

```sql
SELECT category, AVG(price) AS avg_price
FROM products
WHERE category IN ('Home & Kitchen', 'Fitness')
GROUP BY category;
```

SOLUTIONS

Q3. Show product names and stock quantity where the product is available, stock is more than 50, and price is **not equal** to ₹299.

```sql
SELECT name, stock_quantity FROM products
WHERE is_available = TRUE
AND stock_quantity > 50
AND price != 299.00;
```

SOLUTIONS

Q4. Find the **most expensive product** in each category (name and price).

```sql
SELECT category, MAX(price) AS max_price
FROM products
GROUP BY category;
```

SOLUTIONS

Q5. Show all unique categories **in uppercase**, sorted in descending order.

```sql
SELECT DISTINCT UPPER(category) AS category_upper
FROM products
ORDER BY category_upper DESC;
```

WAIT HERE SOLDIER

You've come this far and that's no small thing.
You've crossed storms, pushed through doubts, and proved to yourself that you're capable of more than you ever thought. 💪

But remember **you're only halfway there.**
The final stretch still lies ahead. And it won't be easy.

Yes, you're tired. Yes, the path ahead looks tough.
But so what?

🔥 **Real warriors don't stop when they're tired.**

They stop when they've won.

This is your battlefield. Your future is on the line. And you've already come too far to quit now.

So take a deep breath.
Stand tall.
Sharpen your focus, tighten your grip — and charge forward.

STRING FUNCTIONS

String functions in PostgreSQL are used to **manipulate text data** — like names, categories, SKUs, etc.

They help you:

- Clean text
- Extract parts of a string
- Convert cases
- Replace or remove characters
- And much more

Lets see functions one by one...

STRING FUNCTIONS

First set of functions are Upper Lower and length function.

```
SELECT upper(name) from products;
```

```
SELECT lower(sku_code) from products;
```

```
SELECT length(name) from products;
```

the general use case table at the end

STRING FUNCTIONS

Now lets see substring - it is used to extract some portion from your string .

substring(text , location , length)

see the demonstration.

```
SELECT name, SUBSTR(sku_code,1,2) from products;
```

STRING FUNCTIONS

Next set of function are
LEFT() and RIGHT() both have the use case similar to substring
Get n leftmost and rightmost elements.

```
SELECT left(sku_code,2) from products;
```

Same working with right just gets the element from right.

STRING FUNCTIONS

Some other functions are Concat()

Concat joins 2 strings together.

```
SELECT concat(name,' ',category) from products;
```

```
SELECT concat_ws(':',name,category) from products;
```

you can also use concat_ws (with separator) of you don't want to add separator cause you can concat multiple columns at once and it will work fine.

STRING FUNCTIONS

now lets see trim() and replace()

trim() - This function will remove all the spaces from the string.

replace() - This function will replace any thing you want.

```
SELECT replace(sku_code,left(sku_code,2), 'GG') from products;
```

And trim is very easy just see the demonstration.

STRING FUNCTIONS

`LOWER(text)`	Converts to lowercase	`'LAPTOP'` → `'laptop'`
`UPPER(text)`	Converts to uppercase	`'mouse'` → `'MOUSE'`
`LENGTH(text)`	Returns length of string	`'Laptop'` → `6`
`SUBSTRING(text, start, length)`	Extracts part of string	`'Notebook'` → `'Note'`
`LEFT(text, n)`	Gets left-most n characters	`'Notebook'` → `'Note'`
`RIGHT(text, n)`	Gets right-most n characters	`'Notebook'` → `'book'`
`CONCAT(str1, str2)`	Joins two or more strings	`'Sheryians' + 'AI'` → `'SheryiansAI'`
`TRIM(text)`	Removes spaces from start and end	`' Hello '` → `'Hello'`
`REPLACE(text, from, to)`	Replaces a part of string	`'USB-C'` → `'USB'` → `'USB'`

PHASE 5

Alter

CASE

ALTER

Before we dive into what's next, let's take a quick look at what we've already mastered:

Created Tables using `CREATE TABLE`
Understood **Data Types** and **Constraints** (like `NOT NULL`, `UNIQUE`, `CHECK`, etc.)
Performed **CRUD operations** (`INSERT`, `SELECT`, `UPDATE`, `DELETE`)
Explored **SQL Clauses** (`SELECT`, `FROM`, `WHERE`, `ORDER BY`, etc.)
Used **Operators** to filter data (=, >, <, `IN`, `LIKE`, `BETWEEN`, etc.)
Practiced **Aggregation Functions** (`COUNT`, `SUM`, `AVG`, etc.)
Played with **String Functions** to clean and extract text data

ALTER

So far, we've been working with **existing data** — inserting it, reading it, filtering it, analyzing it.

But now imagine this:

You already created a table with all constraints...
You thought the column name "name" is fine, but now you want to change it to "product_name"...
Or you want to add a new column for `brand`*...*
Or you want to remove a constraint, change a data type, rename a table...

Boom That's exactly what `ALTER TABLE` is made for.

ALTER

USE CASES OF `ALTER`:

1. **Add new columns**
2. **Remove columns**
3. **Rename columns**
4. **Change data types**
5. **Set or remove default values**
6. **Add or remove constraints**
7. **Rename the table**

ALTER

For seeing the implementation of all of these things first lets create a simple table and then just see all of the commands needed to alter the table.

```
CREATE TABLE students (
  student_id SERIAL PRIMARY KEY,
  name VARCHAR(100),
  age INT
);
```

ALTER

1. **Add a New Column.**

```sql
ALTER TABLE students
ADD COLUMN email VARCHAR(100);
```

All the rows will have the null values now be aware of that. for that you can have a default value see the demonstration.

ALTER

2. **Remove a Column.**

```
ALTER TABLE students
DROP COLUMN email;
```

ALTER

3. **Rename a Column.**

```sql
ALTER TABLE students
RENAME COLUMN name TO full_name;
```

ALTER

4. Change Data Type of a Column

```
ALTER TABLE students
ALTER COLUMN age TYPE SMALLINT;
```

ALTER

5. **Set a Default Value.**

```
ALTER TABLE students
ALTER COLUMN age SET DEFAULT 18;
```

ALTER

6. **Remove a Default Value**

```
ALTER TABLE students
ALTER COLUMN age DROP DEFAULT;
```

ALTER

7. ADD A CONSTRAINT

```sql
ALTER TABLE students
ADD CONSTRAINT age_check CHECK (age >= 0);
```

ALTER

8. DROP A CONSTRAINT

```
ALTER TABLE students
DROP CONSTRAINT age_check;
```

Note: If you didn’t name the constraint manually while creating it, you'll need to find its auto-generated name via `pg_constraint` or pgAdmin.

ALTER

9. RENAME THE TABLE

```sql
ALTER TABLE students
RENAME TO school_students;
```

CASE

`CASE` is a **conditional expression** in SQL that works like an `if-else` or `switch` statement. It lets you return **different values based on different conditions** — all within a single query.

CASE

WHY DO WE USE CASE?

- To create **custom columns** on-the-fly
- To categorize data based on certain logic
- To **replace values** conditionally
- To handle **nulls or missing values** gracefully
- To simplify complex logic inside SELECT queries

CASE

Syntax of `CASE` in SQL

```
SELECT
  column1,
  CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ...
    ELSE default_result
  END AS new_column_name
FROM table_name;
```

CASE

Best way to learn the case is using a simple example where you will add a custom column in which you will have price_tag.

If the price is above 1000 you will say it is expensive.

If the price is between 500 and 1000 you will say it is moderate.

and If the price is below 500 it is cheap.

CASE

solution is simple.

```sql
SELECT
  name,
  price,
  CASE
    WHEN price > 1000 THEN 'Expensive'
    WHEN price BETWEEN 500 AND 1000 THEN 'Moderate'
    ELSE 'Cheap'
  END AS price_tag
FROM products;

Select * from products;
```

But note this will just create a virtual data but you can also alter the real data.

CASE

Step1 - you have to create a new column.

```
ALTER TABLE products
ADD COLUMN price_tag TEXT;
```

Step 2: Update that column using CASE

```
UPDATE products
SET price_tag =
  CASE
    WHEN price > 1000 THEN 'Expensive'
    WHEN price BETWEEN 500 AND 1000 THEN 'Moderate'
    ELSE 'Cheap'
  END;
```

CASE

Ok now lets do one important question inside is available column you have boolean true and false show case a new column to with in_stock and out of stock.

```sql
SELECT
  name,
  CASE
    WHEN is_available THEN 'In Stock'
    ELSE 'Out of Stock'
  END AS availability_status
FROM products;
```

CASE

HIGHLIGHT STOCK STATUS

> *Show product name, stock quantity, and label:*
>
> - *"`High Stock`" if quantity > 100*
> - *"`Medium Stock`" if between 30 and 100*
> - *"`Low Stock`" otherwise*

```sql
SELECT
  name,
  stock_quantity,
  CASE
    WHEN stock_quantity > 100 THEN 'High Stock'
    WHEN stock_quantity BETWEEN 30 AND 100 THEN 'Medium Stock'
    ELSE 'Low Stock'
  END AS stock_level
FROM products;
```

CASE

TopicCovered?	
What is CASE & why it's used	✔
CASE in `SELECT` to create custom columns	✔
Using `CASE` with numeric, boolean, and category columns	✔
Creating new columns with `ALTER + UPDATE + CASE`	✔
Practical examples: price tag, stock status, availability	✔

PHASE 6

relationships

Joins

RELATIONSHIPS

> *In a **relational database**, data is stored across **multiple tables**, and these tables are connected through **relationships**.*

Instead of repeating the same data again and again in one huge table, we split it into smaller, meaningful tables and connect them using **keys** (Primary and Foreign Keys).

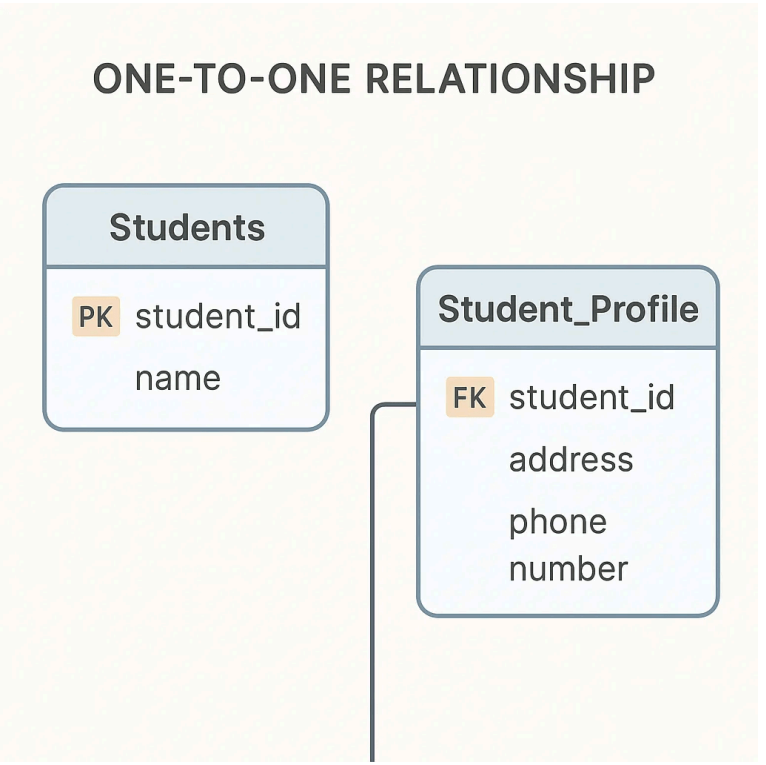
RELATIONSHIPS

Types of Relationships.

1. **One-to-One (1:1)**
2. **One-to-Many (1:N)** 🔥 *Most common*
3. **Many-to-Many (M:N)**

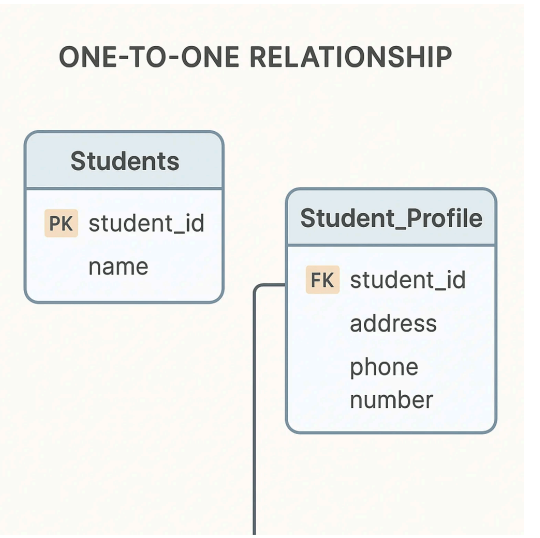
RELATIONSHIPS

One to One relationship



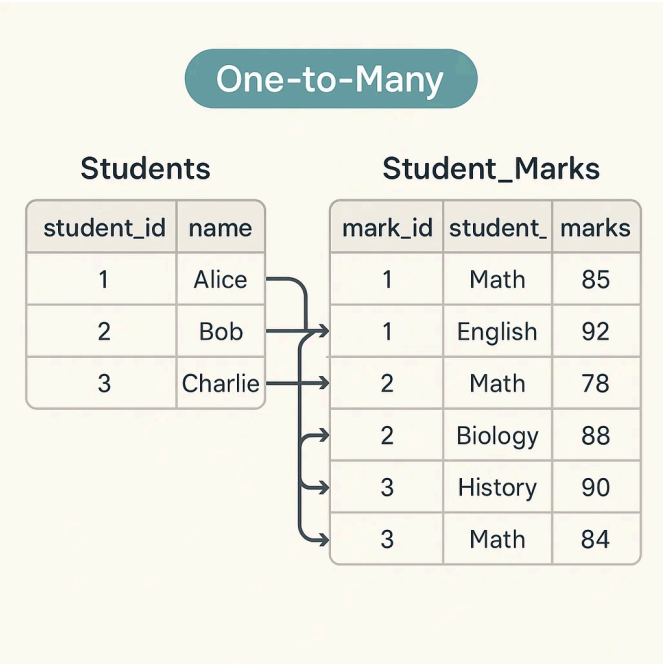
RELATIONSHIPS

- You have a **`students`** table → each student has a **unique** `student_id` (Primary Key).
- You have a **`student_profiles`** table → each profile also has a **unique** `student_id` (Foreign Key + Unique).



RELATIONSHIPS

One to many relationship.



RELATIONSHIPS

You have a `students` table → each student has a unique `student_id` (Primary Key).

You also have a `student_marks` table → each row stores marks for one subject, and contains a `student_id` as a **Foreign Key**.

The same `student_id` can appear **multiple times** in `student_marks`, representing that one student has marks in **multiple subjects**.

So, **one student** in the `students` table is connected to **many rows** in the `student_marks` table.

RELATIONSHIPS

many to many relationship

In many-to-many, both tables (e.g. `students` and `courses`) have multiple entries — and multiple relationships between them.
So how can we talk about **primary key → foreign key** relationships here?
Doesn't primary key mean *one unique row*, and many-to-many means *multiple connections*?

RELATIONSHIPS

many to many relationship

In many-to-many, both tables (e.g. `students` and `courses`) have multiple entries — and multiple relationships between them.
So how can we talk about **primary key → foreign key** relationships here?
Doesn't primary key mean *one unique row*, and many-to-many means *multiple connections*?

RELATIONSHIPS

In a **many-to-many relationship**, you **don't directly connect the two tables** using a foreign key. Instead, you create a **third table** (called a **junction table**) that **breaks the many-to-many into two one-to-many relationships**.

ONE TO ONE EXAMPLE

Now lets see the example of One to One relationship and how to connect the tables.

First lets create 2 simple tables.

ONE TO ONE EXAMPLE

Now lets see the example of One to One relationship and how to connect the tables.

First lets create 2 simple tables in a new Data base.

```
CREATE TABLE students (
  student_id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);
```

```
CREATE TABLE student_profiles (
  student_id INT PRIMARY KEY, t
  address TEXT,
  age INT,
  phone VARCHAR(15)
);
```

ONE TO ONE EXAMPLE

Now lets insert data in both the tables.

```
INSERT INTO students (name)
VALUES
('Akarsh Vyas'),
('Simran Mehta'),
('Rohan Gupta');
```

```
INSERT INTO student_profiles (student_id, address, age, phone)
VALUES
(1, 'Delhi, India', 22, '9999999999'),
(2, 'Mumbai, India', 21, '8888888888'),
(3, 'Bangalore, India', 23, '7777777777');
```

ONE TO ONE EXAMPLE

Now both the tables have been created and we can clearly see there is a similar column and that is student_id but currently there is no relationship setup between them for setting them up you have to create a foreign key in the 2nd table.

🔑 **Primary Key** uniquely identifies each row in a table.
🔗 **Foreign Key** links one table to another by referring to the Primary Key of that table.

ONE TO ONE EXAMPLE

For linking them first alter the constraints and set the student_id as foreign key and reference them to the student_id of the first table.

```
ALTER TABLE student_profiles
ADD CONSTRAINT fk_student_id
FOREIGN KEY (student_id)
REFERENCES students(student_id);
```

ONE TO ONE EXAMPLE

now what next the relationship has been set. now to see the data together you have to use the joins and I will just give you one example and after that in one to many I will tell you all about joins.

```
SELECT
  s.student_id,
  s.name,
  sp.address,
  sp.age,
  sp.phone
FROM students s
JOIN student_profiles sp
ON s.student_id = sp.student_id;
```

ONE TO MANY EXAMPLE

STEP 1: CREATE TABLES

- `students` table → student info
- `marks` table → subject-wise marks (linked to student)

STEP 2: INSERT SAMPLE DATA

STEP 3: ADD FOREIGN KEY CONSTRAINT

STEP 4: TEACH JOINS (WITH EXAMPLES LIKE INNER JOIN, LEFT JOIN, ETC.)

ONE TO MANY EXAMPLE

Creating the tables are easy so I will just create the table to explain one to many relationship.

```sql
CREATE TABLE students (
  student_id SERIAL PRIMARY KEY,
  name VARCHAR(100) NOT NULL
);
```

```sql
CREATE TABLE marks (
  mark_id SERIAL PRIMARY KEY,
  student_id INT,
  subject VARCHAR(50),
  marks INT,
  FOREIGN KEY (student_id) REFERENCES students(student_id)
);
```

And I have already referenced the Primary and Foreign key that is student_id.

ONE TO MANY EXAMPLE

Now lets insert the values and notice I have to manually provide the student_id in the second table.

```sql
INSERT INTO students (name)
VALUES ('Akarsh Vyas'), ('Simran Mehta'), ('Rohan Gupta');


INSERT INTO marks (student_id, subject, marks)
VALUES
(1, 'English', 85),
(1, 'Math', 89),
(1, 'Science', 92),

(2, 'English', 80),
(2, 'Math', 75),
(2, 'Science', 78),

(3, 'English', 72),
(3, 'Math', 70),
(3, 'Science', 74);
```

ONE TO MANY EXAMPLE

Ok so now we have connected the tables but now what to do with these tables lets explore.

ONE TO MANY EXAMPLE

These are some set of questions can you answer them.

1)Show each student's name along with their subject and marks.

2)Show marks for **only "Simran Mehta"** in all subjects.

3) Show **only those subjects** where marks are **above 80**.

4) Sort all students' subject marks in **descending order of marks**.

5)Show each student's **average marks**.

JOINS

JOINS are used to **combine rows from two or more tables** based on a **related column**, usually a **primary key** in one table and a **foreign key** in another.

> *Think of JOINs as a bridge between two tables that lets you query them together.*

JOINS

Example scenario:

- `students` table has: `student_id`, `name`
- `marks` table has: `student_id`, `subject`, `marks`

Using JOIN, we can get:
"`Simran scored 89 in Math`" by combining data from both.

JOINS

WHY DO WE USE JOINS?

- To **merge related data** spread across multiple tables.
- To write **meaningful real-world queries** like:
  - "Which student scored highest in Science?"
  - "List all students even if they haven’t appeared for any exam."

JOINS

Join Type	Description
`INNER JOIN`	Returns **only matching rows** in both tables
`LEFT JOIN`	Returns **all rows from the left table**, even if there's no match in the right table
`RIGHT JOIN`	Returns **all rows from the right table**, even if there's no match in the left table
`FULL JOIN`	Returns **all rows from both tables**, fills `NULL` for missing matches
`CROSS JOIN`	Returns **cartesian product** (every combination)

JOINS

Inner join syntax

```
SELECT columns
FROM table1
JOIN table2
ON table1.common_column = table2.common_column;
```

JOINS

ok so you must have seen the demonstration of inner joins.

```
select m.subject,s.name,m.marks
from students s join marks m
on s.student_id = m.student_id;
```

JOINS

now see the video demonstration for clearly understanding joins.

EXERSISE

I will provide you 2 csv files one will be products csv and another will be orders csv create the tables in SQl and after creating the tables with specific constraints and columns.

Import those csv files values in your SQL table.

IF YOU'RE USING PGADMIN

1. Right-click the table > **Import/Export**
2. Select file path → choose **CSV**
3. Map columns (ensure types match)
4. Click **OK** to import

EXERSISE

and importing with commands you have to use this

\COPY products FROM '/your/path/products.csv' DELIMITER ',' CSV HEADER;
\COPY orders FROM '/your/path/orders.csv' DELIMITER ',' CSV HEADER;

Ok now after setting up the tables now here are some questions pause and solve them if you can otherwise no issues see the demonstration.

EXERSISE

Q1. Show each order along with the product name and price.

Q2. Show all products even if they were never ordered.

Q3.Show orders for only ‘Electronics’ category.

Q4.List all orders sorted by product price (high to low).

Q5.Show number of orders placed for each product.

Q6.Show total revenue earned per product.

Q7.Show products where total order revenue > ₹2000.

Q8.Show unique customers who ordered ‘Fitness’ products.

MANY TO MANY RELATIONSHIP

We have already completed 90% of the real world use cases but now it is time to go towards that 10% and for that we have to understand many to many as well.

MANY TO MANY RELATIONSHIP

In a **many-to-many** relationship:

See the example to understand what is many to many relationship.

- One row in **Table A** can be linked to **many** rows in **Table B**.
- And one row in **Table B** can be linked to **many** rows in **Table A**.

MANY TO MANY RELATIONSHIP

ok after creating the table now solve these 2 questions

1) Show the list of students with the courses they are enrolled in.

2)Find all the courses taken by the student named 'Simran'.

MESSAGE

Now that you've understood how **relationships** (One-to-One, One-to-Many, Many-to-Many) work in SQL and how to use **JOINs** to connect multiple tables...

> 🔁 *It's time to get your hands dirty.*

Go on platforms like **LeetCode**, **HackerRank**, or **StrataScratch**, and start solving **real-world SQL problems**.

These platforms will:

- Force you to think in **multi-table queries**
- Improve your understanding of **JOINs, GROUP BY, filtering, aggregation**
- Show you how **complex relationships** work in practical scenarios

Remember:
"Concepts are the foundation, but practice builds mastery."

PHASE 7

Views

Procedures

VIEWS

WHAT IS A VIEW IN SQL?

> *A **view** is a **virtual table** based on a **SQL query**.*
> *It **does not store actual data**, but shows results when accessed — just like a saved query.*

VIEWS

Simplify Complex Queries	Save a long query and access it like a table
Reuse Logic	No need to rewrite JOINs or filters again and again
Security	Expose only selected columns instead of giving full access to the table
Abstraction	Hide table complexity for front-end/dashboard users
Maintainability	If logic changes, update the view once — changes reflect everywhere

VIEWS

Simplify Complex Queries	Save a long query and access it like a table
Reuse Logic	No need to rewrite JOINs or filters again and again
Security	Expose only selected columns instead of giving full access to the table
Abstraction	Hide table complexity for front-end/dashboard users
Maintainability	If logic changes, update the view once — changes reflect everywhere

VIEWS

Syntax is also simple

```
CREATE VIEW view_name AS
SELECT column1, column2
FROM table_name
WHERE condition;
```

VIEWS

If you remember your product DB lets create 2 views there.

1)Get only available fitness items regularly.

2)View for Low Stock Items.

VIEWS

you can drop the view and also update the view using CREATE OR REPLACE VIEW view_name AS

PROCEDURES

A **procedure** is a block of SQL code that performs a **series of operations** — like inserting, updating, deleting, or selecting data — and is **stored in the database**.

> *Think of it like a **function in programming** — once defined, you can call it again and again without rewriting the logic.*

PROCEDURES

BenefitExplanation	
🔁 Reusability	Write once, use many times
🔐 Security	Logic is stored in DB, no need to give direct access to all tables
⚡ Faster execution	Compiled and stored on the DB server
📦 Encapsulation	Hide complex logic in one callable block
👥 Multi-step operations	Perform multiple queries like insert + update + log creation in one go

PROCEDURES

```
CREATE PROCEDURE procedure_name(param1 datatype,
param2 datatype)
LANGUAGE plpgsql
AS $$
BEGIN
   -- Your SQL logic here
END;
$$;


CALL procedure_name(value1, value2);
```

PROCEDURES

Create a procedure for adding new product in database.