

PostgreSQL Cheatsheet

Harmain Ali Butt


Table of contents

1. INSTALLATION	
2. CREATING A DATABASE	
3. CREATING A TABLE	
4. INSERTING VALUES INTO A TABLE	
IMPORTING DATA FROM EXTERNAL FILLES	
5. DELETING A TABLE	
6. MODIFYING A TABLE	
1. Add a Column	
2. Rename a Column	
3. Change Column Data Type	
4. Set or Change a Default Value	
5. Remove a Default Value	
6. Delete a Column	
7. Rename the Table	
8. Add a Constraint	
9. Drop a Constraint	
10. Truncate Table (Delete All Rows but Keep Structure)	
7. SELECT ... FROM COMMAND	
1. Basic SELECT Query	
2. Select All Columns	
3. Rename Columns with Aliases	
8. WHERE COMMAND	
4. Filter Rows Using WHERE	
7. Sorting Results – ORDER BY	
8. Limiting Results – LIMIT	
9. DISTINCT Values	
10. IN, BETWEEN, IS NULL	
9. AGGREGATE FUNCTIONS	

10. GROUP BY	20
2. Multiple Columns in GROUP BY	
3. Using GROUP BY with ORDER BY	
4. Filter Groups with HAVING	
5. Combine with Aggregate Functions	
11. WILDCARDS	
1. Wildcards with LIKE / ILIKE	
2. Wildcards with SIMILAR TO	
3. Wildcards in Regular Expressions	
3. PostgreSQL Array Operator: @>	
12. UPDATE COMMAND	
1. Update a Single Column	
2. Update Multiple Columns	
3. Update All Rows	
4. Update Using Expressions	
5. Update Using Data From Another Table	
6. Conditional Updates with CASE	
13. DELETE COMMAND	
1. Delete Specific Rows	
2. Delete All Rows	
3. Delete Using a Condition with Operators	
4. Delete Using Subquery	
5. Delete All Rows Quickly	
14. SUBQUERY	
1. Subquery in WHERE Clause	
2. Subquery in FROM Clause	
3. Subquery in SELECT Clause	
4. Correlated Subquery	
5. EXISTS with Subquery	
15. FOREIGN KEYS	
1. Create Table with Foreign Key	
2. Add Foreign Key to an Existing Table	
3. Delete or Update Behavior	
4. Drop a Foreign Key Constraint	
16. RELATIONSHIPS BETWEEN TABLES	
1. One-to-One (1:1)	
2. One-to-Many (1:N)	
3. Many-to-Many (M:N)	
17. JOINS	
1. INNER JOIN	
2. LEFT JOIN	
3. RIGHT JOIN	
4. FULL OUTER JOIN	

5. CROSS JOIN	5
6. SELF JOIN	
18. COMMON TABLE EXPRESSIONS – CTEs	
19. VIEWS & MATERIALIZED VIEWS	
Creating a View	
Using the View	
Updating a View (Replace/Modify)	
Dropping a View	
Creating a Materialized View	
Refreshing a Materialized View	
Dropping a Materialized View	
20. STORED PROCEDURES	
Call the Procedure	
Dropping a Procedure	
21. Functions	
Example 1: Simple Function to Add Two Numbers	
Example 2: Function Returning Text	
Example 3: Function Returning a Table	
Dropping a Function	
22. TRIGGERS	
Example 1: Log Updates	
Example 2: Prevent Negative Salary (Validation)	
Dropping a Trigger	
23. WINDOW FUNCTIONS	
Common Window Functions	
a) Ranking Functions	
b) Aggregate Window Functions	
c) Running Totals & Moving Averages	
d) LAG() and LEAD()	
24. INDEXES	
B-Tree Index	
Hash Index	
GIN (Generalized Inverted Index)	
4) GiST (Generalized Search Tree)	
25. USER / ROLES MANAGEMENT	
1. Create User / Role	
2. Grant Role Membership	
3. Grant Privileges	
4. Revoke Privileges	
5. Alter User	
6. Drop User / Role	
26. CONNECTING PostgreSQL TO PYTHON	

POSTGRE SQL CHEAT SHEET

 **NOTE:** This notebook is intended to:

- Provide exhaustive and comprehensive guide of the Postgresql.
- Discuss the syntax and code of the most used functions in Postgresql.

1. INSTALLATION

PostgreSQL is a free, powerful, and open-source relational database management system (RDBMS). It's widely used in *Data Science*, *Web Development*, and *large-scale applications* due to its performance, reliability, and features like **full ACID compliance** and support for **complex queries**.

pgAdmin is a graphical user interface (GUI) tool that makes it easier to interact with PostgreSQL databases. Instead of typing commands in the terminal, you can create, manage, and query databases **visually**.

When you're getting started, installing PostgreSQL automatically installs pgAdmin (unless you choose not to). These tools work together:

- **PostgreSQL** handles the *backend database system*.
- **pgAdmin** helps you *view, manage, and edit* your databases with **clicks instead of code**.

Steps to Install PostgreSQL & pgAdmin

Step 1: Download Installer

- Go to the official site: <https://www.postgresql.org/download/>
- Select your operating system (Windows, macOS, Linux)
- Choose the "Graphical Installer" by EDB (EnterpriseDB)

Step 2: Run the Installer

- Open the downloaded `.exe` or `.dmg` file
- The setup wizard will launch

Step 3: Installation Options

- **Make sure the following are selected:**
 - PostgreSQL Server
 - pgAdmin 4
 - Stack Builder (optional)
- Click “Next” to proceed

Step 4: Set Superuser Password

- Set a strong password for the default PostgreSQL superuser account (usually `postgres`)
- You will need this password to log in later

Step 5: Choose Port

- Leave the default port as **5432**
- Click “Next”

Step 6: Finish Installation

- Click “Finish” to complete the installation

2. CREATING A DATABASE

A **database** in PostgreSQL is like a container or folder where all your data is stored. You can create *multiple databases* on the same PostgreSQL server — each database can hold its own tables, schemas, functions, and data.

For example:

- You might have a `sales_data` database for business analytics.
- A `machine_learning` database for storing datasets and model outputs.

Each database is isolated, which means:

- Data in one database can't directly interact with data in another.
- You must connect to a specific database before running SQL queries.

```
In [ ]: -- SQL Syntax to Create a Database
CREATE DATABASE database_name;
-- Example:
CREATE DATABASE sales_data;
```

```
-- Optional Parameters: You can specify additional options like encoding, ow
CREATE DATABASE database_name
WITH
  OWNER = username
  ENCODING = 'UTF8'
  CONNECTION LIMIT = 20;

-- OWNER — Who owns the DB (default is current user)
-- ENCODING — Character set (UTF8 is standard)
-- CONNECTION LIMIT — Max number of active connection
```

How to Create a Database in pgAdmin GUI (Windows App)

Step-by-Step Instructions

1. Open pgAdmin

Launch from Start Menu → pgAdmin 4

2. Connect to Server

Enter your master password

3. Navigate to the Database Section

Expand: Servers > PostgreSQL

4. Right-click on "Databases"

Click *Create* → *Database...*

5. Fill in the Form:

- **Database name:** e.g., *sales_data*
- **Owner:** Leave as *postgres* or choose another user
- Leave other settings default unless you need advanced configuration

6. Click Save

Your new database will now appear in the left panel under **Databases**.

3. CREATING A TABLE

What is a Table in PostgreSQL?

A table in PostgreSQL is like a spreadsheet inside your database. It stores your data in rows (records) and columns (fields).

- Each column has a name and a data type (like text, number, date, etc.).
- Each row is a single entry — like one person, product, or transaction.

In []: -- SQL Syntax to Create a Table

```
CREATE TABLE table_name (  
    column1 datatype constraints,  
    column2 datatype constraints,  
    ...  
);
```

-- Example:

```
CREATE TABLE employees (  
    employee_id SERIAL PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    age INTEGER,  
    email VARCHAR(100) UNIQUE,  
    hire_date DATE DEFAULT CURRENT_DATE  
);
```

-- Use IF NOT EXISTS to avoid errors **if** table already exists:

```
CREATE TABLE IF NOT EXISTS employees (  
    employee_id SERIAL PRIMARY KEY,  
    name VARCHAR(100) NOT NULL,  
    age INTEGER,  
    email VARCHAR(100) UNIQUE,  
    hire_date DATE DEFAULT CURRENT_DATE  
);
```

How to Create a Table in pgAdmin (GUI Method)

Step-by-Step:

1. Open pgAdmin and connect to your server.
2. Navigate to:
Servers > PostgreSQL > Databases > [Your Database] > Schemas > public > Tables
3. Right-click on **Tables** → Click **Create > Table...**
4. In the Dialog Box:
 - **General Tab**
Name: employees (or your desired table name)
Owner: Leave as postgres (default)
 - **Columns Tab**
Click + Add for each column
Name: e.g. employee_id
Datatype: Choose from dropdown (e.g. serial, varchar, integer, date)
Set constraints like Primary Key, Not Null, Default

- **Constraints Tab (Optional)**

Set Primary Key or Foreign Key if needed (can also be done in the Columns tab)

5. Click **Save**

6. Your table now appears in the left panel.

PostgreSQL data types:

Category	Data Type	Description	When to Use
Numeric	INTEGER (INT)	Whole numbers (-2,147,483,648 to 2,147,483,647)	General-purpose whole numbers
	SMALLINT	Smaller integers (-32,768 to 32,767)	Memory-efficient integers
	BIGINT	Large integers (-9 quintillion to +9 quintillion)	Very large whole numbers (e.g., population count)
	DECIMAL(p,s) / NUMERIC(p,s)	Exact fixed-point numbers (e.g., 12.34)	Financial data, currency, precise calculations
	REAL	Approximate floating-point (single precision)	Scientific data (lower precision)
	DOUBLE PRECISION	Approximate floating-point (double precision)	Scientific or engineering use cases requiring high precision
	SERIAL	Auto-incrementing 4-byte integer	Primary keys, unique IDs
Monetary	BIGSERIAL	Auto-incrementing 8-byte integer	Large auto-generated keys
	MONEY	Currency representation	Basic financial amounts with formatting
Character	CHAR(n)	Fixed-length string	Use when all values are same length (e.g., country code)
	VARCHAR(n)	Variable-length string with limit	Short text with length restriction (e.g., names, emails)
	TEXT	Variable-length string without limit	Large or unrestricted text (e.g., comments, descriptions)
Date & Time	DATE	Calendar date (YYYY-MM-DD)	For date fields (e.g., DOB, created_at)
	TIME	Time without time zone	For storing time alone (e.g., office hours)

Category	Data Type	Description	When to Use
	TIMESTAMP	Date and time without time zone	When date-time is important (e.g., order timestamps)
	TIMESTAMPTZ	Date and time with time zone	Global apps, consistent time tracking
Boolean	BOOLEAN	True or False	Status flags (e.g., is_active, verified)
UUID	UUID	Universally Unique Identifier	Unique user/session IDs, distributed systems
Network Address	INET	IP address (IPv4 or IPv6)	Storing IP addresses
	CIDR	IP address with subnet	Network configurations
JSON	JSON	Textual JSON data	Semi-structured data that needs structure validation
	JSONB	Binary-optimized JSON	Efficient queries and indexing on JSON
Geometric	POINT, LINE, CIRCLE	Geometric shapes	Mapping, GIS, coordinates
Arrays	any[] (e.g., INT[])	Arrays of any data type	Multiple values in one column (e.g., tags, scores)
HStore	HSTORE	Key-value pairs	Schema-less key-value data
Binary	BYTEA	Binary data	Files, images, encrypted content
Enumerated	ENUM	Predefined set of values	Gender, status, category fields
Range Types	INT4RANGE, TSRANGE	Ranges of numbers or timestamps	Date ranges, score bands
Full Text Search	TSVECTOR	Optimized format for full-text search	Searchable documents, blog content

PostgreSQL Constraints:

Constraint Name	Description	When to Use	Applicable Data Types
PRIMARY KEY	Uniquely identifies each row in a table. Implicitly adds NOT NULL & UNIQUE	Use for unique row identifiers like id, user_id	INT, BIGINT, UUID, SERIAL, etc.

Constraint Name	Description	When to Use	Applicable Data Types
FOREIGN KEY	Ensures values match values in another table's primary key	Use to create relationships between tables (e.g., orders.customer_id)	Same type as referenced primary key
UNIQUE	Ensures all values in a column (or combination) are different	Use when no duplicates are allowed (e.g., email, username)	TEXT, VARCHAR, INT, etc.
NOT NULL	Prevents null (missing) values	Use when every row must have a value (e.g., name, price, date_of_birth)	Any type
CHECK	Validates that a value meets a condition	Use for constraints like age > 18, salary > 0	Depends on condition (e.g., NUMERIC)
DEFAULT	Assigns a default value if none is given	Use to auto-fill common values (e.g., status = 'pending')	Any type
EXCLUDE	Prevents rows with overlapping values in specified columns using operators	Use for advanced rules (e.g., prevent event time overlaps in booking apps)	RANGE, GEOMETRY, TIMESTAMP
NULL	Allows NULL values (implicitly default if NOT NULL is not specified)	Use when some fields are optional (e.g., middle_name)	Any type
DEFERRABLE / INITIALLY DEFERRED	Controls when constraints are checked (at transaction end, not immediately)	Use in complex transactional logic (e.g., circular foreign keys)	Used with PRIMARY, FOREIGN, etc.
ON DELETE / ON UPDATE (CASCADE, SET NULL, etc.)	Action to take when referenced rows are deleted/updated	Use with FOREIGN KEYs for cascading behaviors	FOREIGN KEY referenced data types

4. INSERTING VALUES INTO A TABLE

Once you've created a table in PostgreSQL, you need to add actual data into it — this is called inserting rows.

You can:

- Add one row at a time
- Add multiple rows in one query
- Insert values into specific columns
- Use default values if you don't want to provide all columns

```
In [ ]: -- 1. Insert a Single Row

INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);

-- Example:
INSERT INTO employees (name, age, email, hire_date)
VALUES ('Ali Khan', 30, 'ali.khan@example.com', '2023-07-01');
```

```
In [ ]: -- 2. Insert Multiple Rows at Once

INSERT INTO table_name (column1, column2, ...)
VALUES
    (value1a, value2a, ...),
    (value1b, value2b, ...),
    (value1c, value2c, ...);

-- Example:

INSERT INTO employees (name, age, email, hire_date)
VALUES
    ('Sara Malik', 28, 'sara@example.com', '2023-08-01'),
    ('Zain Raza', 35, 'zain@example.com', '2023-08-10'),
    ('Ayesha Iqbal', 32, 'ayesha@example.com', '2023-08-15');
```

```
In [ ]: -- 3. Insert Only Some Columns

INSERT INTO employees (name, age)
VALUES ('Hamza Noor', 29);

-- This will insert a new row with the specified name and age, while other c
```

How to Insert Data Using pgAdmin GUI

Step-by-Step (Using Data View)

1. Open pgAdmin 4
2. Go to:

```
Servers > Databases > [Your DB] > Schemas > public > Tables >
[Your Table]
```

3. Right-click your table → Select **View/Edit Data > All Rows**
4. A spreadsheet-style window will open
5. In the blank row at the bottom:
 - Enter your data in each cell
 - Hit Enter or click into the next row to save
6. pgAdmin will auto-run an `INSERT` query in the background

You can also click the pencil icon on the top toolbar to edit existing rows.

IMPORTING DATA FROM EXTERNAL FILLES

Importing Data into PostgreSQL

When working with PostgreSQL, you often have data stored in external files like:

- `.csv` (Comma-Separated Values)
- `.txt` (Text files)
- `.xlsx` (Excel files — indirectly)

To import this data into PostgreSQL, you usually:

1. Create a table that matches the file's structure (same column names and data types)
2. Use either SQL (`COPY` command) or pgAdmin's GUI to load the file

This is extremely useful for data scientists working with survey results, Kaggle datasets, or client data exports.

If Table Doesn't Exist

1. Right-click **Tables** → **Create** → **Table**
2. Define columns and data types manually

```
In [ ]: -- Import Using SQL

COPY table_name (column1, column2, ...)
FROM 'absolute/path/to/file.csv'
DELIMITER ','
CSV HEADER;

-- Example:
```

```
COPY sales_data (order_id, customer_name, total_amount)
FROM 'C:/Users/JHON/Documents/sales.csv'
DELIMITER ','
CSV HEADER;
```

```
-- Make sure:
-- The file path is absolute
-- The file is accessible by the PostgreSQL server
-- If you're on Windows, use / or double \\ in the path
```

```
In [ ]: -- Import Using \COPY (Client-Side Method)

\COPY sales_data (order_id, customer_name, total_amount)
FROM 'C:/Users/Harmain/Documents/sales.csv'
DELIMITER ','
CSV HEADER;
```

Import Using pgAdmin GUI

Step-by-Step (CSV File Import)

If table already exists:

1. Open pgAdmin
2. Expand:
Servers > Databases > [Your DB] > Schemas > public > Tables
3. Right-click the table you want to import into → Select *Import/Export Data...*

In the dialog:

- **Filename:** Browse to your .csv file
- **Format:** CSV
- **Header:** Check this if your CSV has column names
- **Delimiter:** Usually ,
- **Quote:** Usually " (default)
- **Encoding:** Keep as UTF8

Click **OK** to import.

5. DELETING A TABLE

When you delete a table in PostgreSQL, you are permanently removing:

- The table structure (its columns)
- All the data stored inside it

This operation is done using the `DROP TABLE` command.

Warning: It cannot be undone, so use it carefully — especially in production or live environments.

Deleting a table is useful when:

- You imported wrong data and want to start over
- You created a test table and no longer need it
- You want to clean up unused tables

```
In [ ]: -- SQL Syntax to Delete a Table

DROP TABLE table_name;

-- Example:

DROP TABLE employees;
-- This will permanently remove the employees table from the database.
```

```
In [ ]: -- Safe Deletion with IF EXISTS

DROP TABLE IF EXISTS employees;
```

```
In [ ]: -- Delete Multiple Tables at Once

DROP TABLE IF EXISTS table1, table2, table3;
```

How to Delete a Table in pgAdmin (GUI Method)

Step-by-Step:

1. Open pgAdmin
2. Navigate to:
Servers > Databases > [Your DB] > Schemas > public > Tables
3. Right-click the table you want to delete (e.g., employees)
4. Click **Delete/Drop**
5. Confirm when prompted

The table is now permanently removed.

6. MODIFYING A TABLE

Modifying a table means changing its structure after it has been created. This is useful when:

- You forgot a column
- Need to rename something
- Want to change data types
- Add or remove constraints
- Adjust default values, etc.

In []: -- The command used **for** most modifications **is**:

```
ALTER TABLE table_name ...;
```

1. Add a Column

In []: -- SQL Syntax:

```
ALTER TABLE table_name ADD COLUMN column_name data_type;
```

-- Example:

```
ALTER TABLE employees ADD COLUMN age INT;
```

Using pgAdmin GUI to Add a Column

1. Go to: **Tables > [Your Table] > Columns**
2. Right-click → **Create > Column**
3. Set name, type, and click Save

2. Rename a Column

In []: -- SQL Syntax:

```
ALTER TABLE table_name RENAME COLUMN old_name TO new_name;
```

-- Example:

```
ALTER TABLE employees RENAME COLUMN age TO employee_age;
```

Rename a Column using pgAdmin GUI

1. Right-click on the column
2. Click **Properties**
3. Change the name → Save

3. Change Column Data Type

In []: -- SQL Syntax:

```
ALTER TABLE table_name ALTER COLUMN column_name TYPE new_data_type;
```

```
-- Example:  
ALTER TABLE employees ALTER COLUMN age TYPE VARCHAR(3);
```

Change Column Data Type using pgAdmin GUI

1. Right-click on the column → **Properties**
2. Change the data type
3. Click **Save**

4. Set or Change a Default Value

```
In [ ]: -- SQL Syntax:  
ALTER TABLE table_name ALTER COLUMN column_name SET DEFAULT default_value;  
  
-- Example:  
ALTER TABLE employees ALTER COLUMN age SET DEFAULT 25;
```

Set Default Value using pgAdmin GUI

1. Open the column → **Properties**
2. Scroll to the **Default** field
3. Set the desired value
4. Click **Save**

5. Remove a Default Value

```
In [ ]: -- SQL Syntax:  
ALTER TABLE table_name ALTER COLUMN column_name DROP DEFAULT;
```

6. Delete a Column

```
In [ ]: -- SQL Syntax:  
ALTER TABLE table_name DROP COLUMN column_name;  
  
-- Example:  
ALTER TABLE employees DROP COLUMN age;
```

Delete Column using pgAdmin GUI

1. Right-click the column under **Columns**
2. Click **Delete/Drop**

7. Rename the Table

```
In [ ]: -- SQL Syntax:
ALTER TABLE old_table_name RENAME TO new_table_name;

-- Example:
ALTER TABLE employees RENAME TO staff;
```

Rename a Table using pgAdmin GUI

1. Right-click the table
2. Select **Rename**
3. Enter the new name

8. Add a Constraint

```
In [ ]: -- You can add constraints like NOT NULL, UNIQUE, CHECK, FOREIGN KEY, PRIMARY KEY, etc.

-- Add NOT NULL:
ALTER TABLE employees ALTER COLUMN name SET NOT NULL;

-- Add UNIQUE:
ALTER TABLE employees ADD CONSTRAINT unique_email UNIQUE(email);

-- Add CHECK:
ALTER TABLE employees ADD CONSTRAINT age_check CHECK (age > 18);
```

Add a Constraint using pgAdmin GUI

1. Go to: **Constraints > [Type]**
2. Right-click → **Create**
3. Set values and condition → **Save**

9. Drop a Constraint

```
In [ ]: -- SQL Syntax:
ALTER TABLE table_name DROP CONSTRAINT constraint_name;

-- Example:
ALTER TABLE employees DROP CONSTRAINT age_check;
```

10. Truncate Table (Delete All Rows but Keep Structure)

```
In [ ]: -- SQL Syntax:
TRUNCATE TABLE table_name;

-- This deletes all records instantly – much faster than DELETE.
```

Task	SQL Command	pgAdmin Path
Add Column	ADD COLUMN	Columns → Create
Drop Column	DROP COLUMN	Right-click column → Delete
Rename Column	RENAME COLUMN	Column → Properties
Change Data Type	ALTER COLUMN ... TYPE	Column → Properties
Add Constraint	ADD CONSTRAINT	Constraints → Create
Drop Constraint	DROP CONSTRAINT	Constraints → Delete
Rename Table	RENAME TO	Table → Rename
Truncate	TRUNCATE TABLE	Right-click table → Truncate/Empty

7. SELECT ... FROM COMMAND

SELECT ... FROM is used to retrieve data from one or more tables.

You tell the database:

- What columns you want to see → **SELECT**
- Where to get them from → **FROM**

This command is the most frequently used in SQL and is the base of all data exploration, filtering, reporting, and joining.

1. Basic SELECT Query

```
In [ ]: -- SQL Syntax:
SELECT column1, column2 FROM table_name;

-- Example:C
SELECT name, age FROM employees;
```

2. Select All Columns

```
In [ ]: -- SQL Syntax:
SELECT * FROM table_name;
```

```
-- Example:C
SELECT * FROM employees;
```

3. Rename Columns with Aliases

```
In [ ]: -- SQL Syntax:
SELECT column_name AS alias_name FROM table_name;

-- Example:
SELECT name AS employee_name FROM employees;
```

8. WHERE COMMAND

4. Filter Rows Using WHERE

```
In [ ]: -- SQL Syntax:
SELECT * FROM table_name WHERE condition;

-- Examples:

SELECT * FROM employees WHERE age > 30;
SELECT * FROM employees WHERE department = 'HR';
```

5. Use Comparison Operators

Operator	Meaning
=	Equals
!= or <>	Not equal
>, <	Greater/Less than
>=, <=	Greater/Less than or equal

6. Use Logical Operators

Operator	Use Case
AND	Combines conditions (both must be true)
OR	Either condition must be true
NOT	Negates a condition

7. Sorting Results – ORDER BY

```
In [ ]: SELECT * FROM employees ORDER BY age ASC;
        SELECT * FROM employees ORDER BY salary DESC;

        -- ASC is for ascending order, DESC is for descending order.
```

8. Limiting Results – LIMIT

```
In [ ]: -- LIMIT is used to limit the number of rows returned by a query.
        -- Example:

        SELECT * FROM employees LIMIT 10;
        -- This will return the first 10 rows of the result set.

        -- Use OFFSET to skip rows:
        -- OFFSET specifies how many rows to skip before starting to return rows from

        -- SQL Syntax:

        SELECT column1, column2
        FROM table_name
        ORDER BY column_name
        OFFSET n;

        -- where n is the number of rows to skip.

        -- Example:
        SELECT * FROM employees LIMIT 10 OFFSET 5;
```

9. DISTINCT Values

```
In [ ]: -- The DISTINCT keyword in SQL is used to remove duplicate rows from the result set.

        -- SQL Syntax:

        SELECT DISTINCT column1, column2, ...
        FROM table_name;

        -- Example:
        SELECT DISTINCT department FROM employees;
```

10. IN, BETWEEN, IS NULL

Special Conditions for Filtering Rows in PostgreSQL

PostgreSQL provides several useful operators for filtering rows in a **WHERE** clause:

- **IN** – Checks if a value exists in a list of values.
- **BETWEEN** – Checks if a value lies within a range (inclusive).

- **IS NULL / IS NOT NULL** – Checks if a column is empty (NULL) or not empty.

```
In [ ]: -- Syntax:

SELECT column1, column2
FROM table_name
WHERE column_name IN (value1, value2, ...);

-- Example:
SELECT *
FROM employees
WHERE department IN ('IT', 'HR');
-- Find employees in IT or HR department
```

```
In [ ]: -- NOT IN:
-- The NOT IN operator is used to exclude rows that match any value in a spe

-- Example:
SELECT *
FROM employees
WHERE department NOT IN ('IT', 'HR');
-- Employees not in IT or HR
```

```
In [ ]: -- Syntax:

SELECT column1, column2
FROM table_name
WHERE column_name BETWEEN value1 AND value2;

-- Example:
SELECT *
FROM employees
WHERE age BETWEEN 25 AND 35;
-- Employees aged between 25 and 35
```

```
In [ ]: NOT BETWEEN:
-- The NOT BETWEEN operator is used to exclude rows that fall within a speci

-- example:
SELECT *
FROM employees
WHERE age NOT BETWEEN 25 AND 35;
-- Employees outside age range 25 to 35

-- Note: BETWEEN works with numbers, dates, and text (alphabetical range).
```

```
In [ ]: -- Syntax:

SELECT column1, column2
FROM table_name
WHERE column_name IS NULL;
```

```
-- Example:
SELECT *
FROM employees
WHERE email IS NULL;
-- Find employees with no email

SELECT *
FROM employees
WHERE email IS NOT NULL;
-- Find employees who have an email
```

```
In [ ]: -- Combine These Operators
-- we can combine them using AND / OR:

SELECT *
FROM employees
WHERE department IN ('IT', 'HR')
  AND age BETWEEN 25 AND 35
  AND email IS NOT NULL;
-- Employees in IT or HR, age 25-35, and email is present
```

9. AGGREGATE FUNCTIONS

Aggregate Functions in PostgreSQL

Aggregate functions are functions that take multiple rows as input and return a single value as output.

They are mainly used to summarize or analyze data.

Example: Counting rows, calculating average salary, finding maximum sales.

Aggregate functions ignore NULL values (except **COUNT(*)** which counts all rows).

They are most often used with:

- SELECT
- GROUP BY
- HAVING

Common Aggregate Functions

Function	Description
COUNT()	Returns the number of rows
SUM()	Returns the sum of values
AVG()	Returns the average of values
MAX()	Returns the largest value

Function	Description
MIN()	Returns the smallest value
ARRAY_AGG()	Combines values into an array
STRING_AGG()	Concatenates strings with a separator
VARIANCE() / VAR_POP() / VAR_SAMP()	Measures data spread
STDDEV() / STDDEV_POP() / STDDEV_SAMP()	Standard deviation

In []: `-- 1. COUNT() - Count Rows`

```
SELECT COUNT(*) FROM employees;           -- Count all rows
SELECT COUNT(email) FROM employees;       -- Count non-NULL emails
```

In []: `-- 2. SUM() - Total of Values`

```
SELECT SUM(salary) AS total_salary FROM employees; -- Total salary of all employees
```

In []: `-- 3. AVG() - Average Value`

```
SELECT AVG(salary) AS average_salary FROM employees; -- Average salary of all employees
```

In []: `-- 4. MAX() and MIN() - Largest & Smallest`

```
SELECT MAX(salary) AS highest_salary FROM employees; -- Highest salary among all employees
SELECT MIN(salary) AS lowest_salary FROM employees; -- Lowest salary among all employees
```

In []: `-- 5. ARRAY_AGG() - Combine Values into Array`

```
SELECT ARRAY_AGG(name) AS all_names FROM employees; -- Combine all employee names into an array
```

In []: `-- 6. STRING_AGG() - Combine Strings`

```
SELECT STRING_AGG(name, ', ') AS all_names FROM employees;
-- Combine all employee names into a single string, separated by commas
```

10. GROUP BY

The GROUP BY statement in PostgreSQL is used to group rows that have the same values in specified columns. Then, we can apply aggregate functions like COUNT(), SUM(), AVG(), MAX(), MIN() on each group instead of the entire table.

Think of it like this:

1. First, PostgreSQL divides your data into groups based on the column(s) you choose.

2. Then, it summarizes each group with your aggregate functions.

Use Cases for Data Science:

- Count users in each city
- Average salary per department
- Total sales per month

```
In [ ]: -- Basic Syntax

SELECT column1, AGGREGATE_FUNCTION(column2)
FROM table_name
GROUP BY column1;

-- Example

SELECT department, COUNT(*) AS employee_count
FROM employees
GROUP BY department;

-- Explanation:
-- Groups all employees by department
-- Counts the number of employees in each department
```

2. Multiple Columns in GROUP BY

```
In [ ]: -- Multiple Columns in GROUP BY

SELECT department, job_title, AVG(salary) AS avg_salary
FROM employees
GROUP BY department, job_title;

-- Explanation:
-- Groups employees by both department and job title
-- Calculates the average salary for each group
```

3. Using GROUP BY with ORDER BY

```
In [ ]: -- Sexample

SELECT department, SUM(salary) AS total_salary
FROM employees
GROUP BY department
ORDER BY total_salary DESC;

-- Explanation:
-- Groups salaries by department
-- Orders departments by total salary in descending order
```

4. Filter Groups with HAVING


```
In [ ]: -- HAVING is like a WHERE for groups.
-- WHERE filters rows before grouping
-- HAVING filters groups after aggregation
```

```
In [ ]: -- example

SELECT department, COUNT(*) AS emp_count
FROM employees
GROUP BY department
HAVING COUNT(*) > 5;

-- Explanation:
-- Only shows departments with more than 5 employees
```

5. Combine with Aggregate Functions

```
In [ ]: -- Example:

SELECT department,
       COUNT(*) AS emp_count,
       AVG(salary) AS avg_salary,
       MAX(salary) AS top_salary
FROM employees
GROUP BY department;

-- Explanation:
-- Groups employees by department
-- Calculates the number of employees, average salary, and maximum salary for
```

11. WILDCARDS

Wildcards are special characters used in SQL queries to match unknown characters or sequences of characters in a string.

They are mainly used with:

- **LIKE / ILIKE** (for simple pattern matching)
- **SIMILAR TO** (for regex-like matching)
- **Regular Expressions** (~, ~*)

Wildcards allow you to search for partial matches instead of exact matches.

Example: Finding all emails ending with @gmail.com or all names starting with A.

1. Wildcards with LIKE / ILIKE

LIKE = Case-sensitive pattern matching
ILIKE = Case-insensitive pattern matching

Wildcards:

Wildcard	Meaning
%	Matches 0 or more characters
_	Matches exactly 1 character

```
In [ ]: -- Examples: Using %

-- Names starting with A
SELECT * FROM employees WHERE name LIKE 'A%';

-- Names ending with n
SELECT * FROM employees WHERE name LIKE '%n';

-- Names containing 'li'
SELECT * FROM employees WHERE name LIKE '%li%';

-- Examples: Using _

-- Names with exactly 5 letters
SELECT * FROM employees WHERE name LIKE '_____';

-- Names starting with J and 4 letters total
SELECT * FROM employees WHERE name LIKE 'J_____';
```

2. Wildcards with SIMILAR TO

SIMILAR TO allows SQL-style regular expressions for pattern matching. It combines features of **LIKE** with more regex-like flexibility.

Wildcards:

Wildcard	Meaning
%	Matches 0 or more characters
_	Matches exactly 1 character
[]	Matches any one of the characters inside the brackets

```
In [ ]: -- Example:

-- Names starting with Ali or Ahmed
SELECT * FROM employees WHERE name SIMILAR TO '(Ali|Ahmed)%';
```

```
-- Names starting with A or B
SELECT * FROM employees WHERE name SIMILAR TO '[AB]%' ;
```

3. Wildcards in Regular Expressions

PostgreSQL Regular Expressions Operators

PostgreSQL supports full regular expressions using operators like ~ (case-sensitive) and ~* (case-insensitive) for pattern matching.

Regex Operator	Description	Example Pattern	Matches	Does Not Match
.	Any single character	'a.c'	'abc', 'axc'	'ac', 'abbc'
*	0 or more occurrences	'ab*c'	'ac', 'abc', 'abbc', 'abbbc'	'adc'
+	1 or more occurrences	'ab+c'	'abc', 'abbc', 'abbbc'	'ac'
?	0 or 1 occurrence	'ab?c'	'ac', 'abc'	'abbc'
^	Start of string	'^abc'	'abcdef'	'xabc'
\$	End of string	'abc\$'	'xyzabc'	'abcx'
[abc]	Match a, b, or c	'gr[ae]y'	'gray', 'grey'	'groy'
[^abc]	Match anything except a, b, or c	'[^abc]'	'd', 'x'	'a', 'b', 'c'
{n}	Exactly n times	'a{3}'	'aaa'	'aa', 'aaaa'
{n,}	At least n times	'a{2,}'	'aa', 'aaa', 'aaaa'	'a'
{n,m}	Between n and m times	'a{2,4}'	'aa', 'aaa', 'aaaa'	'a', 'aaaaa'

```
In [ ]: -- Examples:

-- Names ending with 'son'
SELECT * FROM employees WHERE name ~ 'son$';

-- Names starting with A or B
SELECT * FROM employees WHERE name ~* '^[ab]';

-- Names with 2 consecutive vowels
SELECT * FROM employees WHERE name ~* '[aeiou]{2}';
```

3. PostgreSQL Array Operator: @>

The @> operator is an **array containment operator** in PostgreSQL.

It means:

"Does the left array contain all the elements of the right array?"

Example:

```
SELECT ARRAY[1, 2, 3, 4] @> ARRAY[2, 3];
```

Explanation:

This checks whether the array `[1, 2, 3, 4]` contains both `2` and `3`.

✓ Yes → Returns `true`

Use Case:

This operator is especially useful when you're storing lists (arrays) in a table column and want to check if certain elements exist inside them.

```
In [ ]: -- SQL Syntax
SELECT *
FROM your_table
WHERE your_array_column @> ARRAY[element1, element2, ...];

-- Example:

SELECT *
FROM users
WHERE interests @> ARRAY['python', 'sql'];
```

12. UPDATE COMMAND

The **UPDATE** command is used to modify existing records in a table.

- You can change one column, multiple columns, or all rows.
- Usually combined with a **WHERE** clause to avoid changing every row by mistake.
- **Warning:** If you omit the WHERE clause, PostgreSQL will update **all rows** in the table!

```
In [ ]: -- Basic Syntax

UPDATE table_name
SET column1 = value1,
    column2 = value2
WHERE condition;
```

1. Update a Single Column

```
In [ ]: -- Increase salary of employee with id 101 to 60000
UPDATE employees
SET salary = 60000
WHERE id = 101;
```

2. Update Multiple Columns

```
In [ ]: -- Update both salary and department for employee 101
UPDATE employees
SET salary = 65000,
    department = 'Finance'
WHERE id = 101;
```

3. Update All Rows

```
In [ ]: -- Set all employees to active
UPDATE employees
SET status = 'Active';
```

4. Update Using Expressions

```
In [ ]: -- Increase all salaries by 10%
UPDATE employees
SET salary = salary * 1.1;
```

5. Update Using Data From Another Table

```
In [ ]: -- Sync employees department from departments table
UPDATE employees e
SET department = d.name
FROM departments d
WHERE e.department_id = d.id;
```

6. Conditional Updates with CASE

```
In [ ]: -- Give a 10% raise to IT employees, 5% to HR
UPDATE employees
SET salary = salary *
    CASE
        WHEN department = 'IT' THEN 1.10
        WHEN department = 'HR' THEN 1.05
        ELSE 1.00
    END;
```

13. DELETE COMMAND

The **DELETE** command is used to remove rows from a table in PostgreSQL.

- It only removes the **data**, not the table structure.
- Combine with a **WHERE** clause to delete specific rows.

Warning: If you omit **WHERE**, all rows will be deleted and the table will be empty.

```
In [ ]: -- Basic Syntax
DELETE FROM table_name
WHERE condition;
```

1. Delete Specific Rows

```
In [ ]: -- Delete employee with id 101
DELETE FROM employees
WHERE id = 101;

-- Delete all employees in HR department
DELETE FROM employees
WHERE department = 'HR';
```

2. Delete All Rows

```
In [ ]: DELETE FROM employees;
```

3. Delete Using a Condition with Operators

```
In [ ]: -- Delete employees younger than 18
DELETE FROM employees
WHERE age < 18;

-- Delete employees hired before 2020
DELETE FROM employees
WHERE hire_date < '2020-01-01';
```

4. Delete Using Subquery

```
In [ ]: -- Delete employees who belong to inactive departments
DELETE FROM employees
WHERE department_id IN (
    SELECT id FROM departments WHERE active = false
);
```

5. Delete All Rows Quickly

```
In [ ]: TRUNCATE TABLE employees RESTART IDENTITY;  -- RESTART IDENTITY resets auto
```

14. SUBQUERY

A **subquery** (or inner query) is a query inside another query.

PostgreSQL executes the **inner query first**, and its result is used by the outer query.

Subqueries can return:

- **Single value** (scalar subquery)
- **Single column** (used with **IN**)
- **Entire table** (used in **FROM**)

```
In [ ]:  -- Basic Syntax

SELECT column_list
FROM table_name
WHERE column OPERATOR (SELECT column FROM another_table WHERE condition);
```

1. Subquery in WHERE Clause

```
In [ ]:  -- Employees earning more than average salary
SELECT *
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

```
In [ ]:  -- Employees in active departments
SELECT *
FROM employees
WHERE department_id IN (
    SELECT id FROM departments WHERE active = true
);
```

2. Subquery in FROM Clause

```
In [ ]:  -- Top 3 highest average salaries by department
SELECT department, avg_salary
FROM (
    SELECT department, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department
) AS dept_avg
ORDER BY avg_salary DESC
LIMIT 3;
```

```
-- Treats the subquery as a temporary table
```

3. Subquery in SELECT Clause

```
In [ ]: -- Show each employee with the company-wide average salary
SELECT name,
       salary,
       (SELECT AVG(salary) FROM employees) AS company_avg
FROM employees;

-- Subquery returns a single value per row.
```

4. Correlated Subquery

```
In [ ]: -- Employees whose salary is above their department's average
SELECT e.name, e.salary
FROM employees e
WHERE salary > (
    SELECT AVG(salary)
    FROM employees
    WHERE department = e.department
);

-- The inner query depends on the outer query row
-- Runs once for each row of the outer query
```

5. EXISTS with Subquery

```
In [ ]: -- Employees who belong to at least one project
SELECT *
FROM employees e
WHERE EXISTS (
    SELECT 1 FROM projects p
    WHERE p.employee_id = e.id
);

-- Checks if any row exists in the subquery result
```

15. FOREIGN KEYS

A **foreign key** is a column (or set of columns) in one table that refers to the primary key in another table.

Purpose: To create relationships between tables and maintain data integrity.

Foreign keys prevent invalid data from being inserted:

- You cannot insert a value in the foreign key column that doesn't exist in the referenced primary key column.
- You cannot delete a parent row if child rows reference it (unless cascading rules are used).

Think of it like linking tables:

- **Parent table** → Has a primary key
- **Child table** → Has a foreign key pointing to that primary key

Example Scenario

Table: departments

id (PK)	department_name
-----	-----
1	IT
2	HR

Table: employees

id (PK)	name	department_id (FK)
-----	-----	-----
101	Ali	1
102	Sara	2

Here, **department_id** in **employees** is a foreign key referencing **departments.id**.

Means Ali belongs to the IT department because his **departments.id** is 1 and Sara belongs to the HR department because her **departments.id** is 2

1. Create Table with Foreign Key

```
In [ ]: CREATE TABLE departments (
        id SERIAL PRIMARY KEY,
        department_name VARCHAR(50) NOT NULL
    );

CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    department_id INT,
    CONSTRAINT fk_department
        FOREIGN KEY (department_id)
        REFERENCES departments(id)
);
```

2. Add Foreign Key to an Existing Table

```
In [ ]: ALTER TABLE employees
        ADD CONSTRAINT fk_department
        FOREIGN KEY (department_id)
        REFERENCES departments(id);
```

3. Delete or Update Behavior

When a parent row is deleted or updated, PostgreSQL checks what to do with child rows.

Option	Behavior
ON DELETE CASCADE	Delete child rows automatically
ON DELETE SET NULL	Set child rows to NULL
ON DELETE RESTRICT	Prevent deletion if child rows exist (default)
ON DELETE NO ACTION	Similar to RESTRICT
ON UPDATE CASCADE	Update child rows if parent key changes

```
In [ ]: -- Example:

CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    department_id INT,
    FOREIGN KEY (department_id)
        REFERENCES departments(id)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

4. Drop a Foreign Key Constraint

```
In [ ]: ALTER TABLE employees
        DROP CONSTRAINT fk_department;
```

pgAdmin GUI Steps

Create Foreign Key When Creating Table

1. Right-click Tables → Create → Table
2. In Columns tab, define your column
3. In Constraints tab → Foreign Keys, add:
 - Referenced table & column
 - Optional ON DELETE / ON UPDATE behavior

Add Foreign Key to Existing Table

1. Right-click your table → Properties → Constraints → Foreign Keys → Add
2. Select referenced table and column
3. Set cascade rules if needed
4. Click Save

View Foreign Keys

1. Right-click the table → Properties → Constraints tab
2. Check all foreign keys and their referenced tables

16. RELATIONSHIPS BETWEEN TABLES

In relational databases like PostgreSQL, **relationships** between tables define how data in one table relates to data in another table.

Relationships help organize data efficiently and are based on keys:

- **Primary Key (PK)** - Uniquely identifies a record in a table.
- **Foreign Key (FK)** - Refers to the PK in another table.

Types of Relationships:

1. **One-to-One (1:1)** - Each row in Table A relates to exactly one row in Table B.
2. **One-to-Many (1:N)** - A row in Table A can relate to multiple rows in Table B (most common).
3. **Many-to-Many (M:N)** - Rows in Table A can relate to multiple rows in Table B and vice versa, usually implemented using a junction table.

1. One-to-One (1:1)

Definition:

Each row in Table A is related to only one row in Table B, and vice versa.

Example:

- **employees** table → basic info
- **employee_details** table → detailed info (1 row per employee)

```
In [ ]: CREATE TABLE employees (  
        id SERIAL PRIMARY KEY,  
        name VARCHAR(50)
```

```
);

CREATE TABLE employee_details (
    emp_id INT PRIMARY KEY,
    address VARCHAR(100),
    phone VARCHAR(20),
    FOREIGN KEY (emp_id) REFERENCES employees(id)
);
```

2. One-to-Many (1:N)

Definition:

A single row in Table A can have multiple related rows in Table B.
This is the most common relationship.

Example:

- One department can have many employees.

```
In [ ]: CREATE TABLE departments (
        id SERIAL PRIMARY KEY,
        department_name VARCHAR(50)
    );

CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(id)
);

-- departments.id → Primary Key
-- employees.department_id → Foreign Key
```

3. Many-to-Many (M:N)

Definition:

A row in Table A can relate to many rows in Table B, and vice versa.
Requires a third table (junction table) to manage the relationship.

Example:

- Students can enroll in many courses
- Courses can have many students

```
In [ ]: CREATE TABLE students (
        id SERIAL PRIMARY KEY,
```

```

    name VARCHAR(50)
);

CREATE TABLE courses (
    id SERIAL PRIMARY KEY,
    course_name VARCHAR(50)
);

-- Junction table for many-to-many relationship
CREATE TABLE student_courses (
    student_id INT,
    course_id INT,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(id),
    FOREIGN KEY (course_id) REFERENCES courses(id)
);

```

17. JOINS

JOINS in SQL are used to combine rows from two or more tables based on a related column (usually a foreign key).

Why use JOINS?

- To retrieve data spread across multiple tables
- To analyze relationships (e.g., employees & departments)

PostgreSQL supports these main types of joins:

- INNER JOIN
- LEFT JOIN (LEFT OUTER JOIN)
- RIGHT JOIN (RIGHT OUTER JOIN)
- FULL OUTER JOIN
- CROSS JOIN
- SELF JOIN

Example Tables:

departments

id	department_name
1	IT
2	HR
3	Finance

employees

id	name	department_id
101	Ali	1
102	Sara	2
103	John	1
104	Ahmed	NULL

1. INNER JOIN

```
In [ ]: -- Returns only rows with matching values in both tables.
SELECT e.id, e.name, d.department_name
FROM employees e
INNER JOIN departments d
ON e.department_id = d.id;
```

id	name	department_name
101	Ali	IT
102	Sara	HR
103	John	IT

2. LEFT JOIN

```
In [ ]: -- Returns all rows from the left table + matching rows from the right table
-- If no match, right columns are NULL.

SELECT e.id, e.name, d.department_name
FROM employees e
LEFT JOIN departments d
ON e.department_id = d.id;
```

id	name	department_name
101	Ali	IT
102	Sara	HR
103	John	IT
104	Ahmed	NULL

3. RIGHT JOIN

```
In [ ]: -- Returns all rows from the right table + matching rows from the left table
-- If no match, left columns are NULL.

SELECT e.id, e.name, d.department_name
FROM employees e
```

```
RIGHT JOIN departments d
ON e.department_id = d.id;
```

id	name	department_name
101	Ali	IT
102	Sara	HR
103	John	IT
NULL	NULL	Finance

4. FULL OUTER JOIN

```
In [ ]: -- Returns all rows from both tables.
-- If there's no match, missing columns are NULL.
```

```
SELECT e.id, e.name, d.department_name
FROM employees e
FULL OUTER JOIN departments d
ON e.department_id = d.id;
```

id	name	department_name
101	Ali	IT
102	Sara	HR
103	John	IT
104	Ahmed	NULL
NULL	NULL	Finance

5. CROSS JOIN

```
In [ ]: -- Returns every combination of rows from both tables
-- (No condition → Cartesian product)
```

```
SELECT e.name, d.department_name
FROM employees e
CROSS JOIN departments d;
```

name	department_name
Ali	IT
Ali	HR
Ali	Finance
Sara	IT
Sara	HR
Sara	Finance

name	department_name
John	IT
John	HR
John	Finance
Ahmed	IT
Ahmed	HR
Ahmed	Finance

6. SELF JOIN

```
In [ ]: -- Join a table with itself using aliases.
-- Useful for hierarchies like employee → manager relationships.

-- Employees with their managers
SELECT e.name AS employee, m.name AS manager
FROM employees e
LEFT JOIN employees m
ON e.department_id = m.department_id
AND e.id <> m.id;
```

18. COMMON TABLE EXPRESSIONS – CTEs

The WITH Clause in PostgreSQL

The **WITH** clause lets you create temporary named result sets (also called **Common Table Expressions** or **CTEs**) that can be used later in your main query.

Think of it as:

“I’ll prepare some data first, give it a name, and then use it in my main query without writing it again.”

When to Use WITH:

- To break a complex query into smaller, readable parts.
- To reuse the same calculation or dataset multiple times in a query.
- To perform recursive (self-referencing) queries.

Key Points:

- CTEs exist only during the execution of the query.
- They improve query readability and maintainability.
- You can define multiple CTEs by separating them with commas.

In []: -- General Syntax:

```
WITH cte_name AS (  
    SELECT columns  
    FROM table_name  
    WHERE condition  
)  
SELECT *  
FROM cte_name;
```

In []: -- Example 1 – Simple CTE:

```
WITH high_salary_employees AS (  
    SELECT employee_id, name, salary  
    FROM employees  
    WHERE salary > 50000  
)  
SELECT name, salary  
FROM high_salary_employees  
ORDER BY salary DESC;
```

-- This first makes a temporary dataset of employees **with** salary > 50,000, t

In []: -- Example 2 – Multiple CTEs:

```
WITH sales_per_product AS (  
    SELECT product_id, SUM(quantity) AS total_sold  
    FROM sales  
    GROUP BY product_id  
)  
top_products AS (  
    SELECT product_id, total_sold  
    FROM sales_per_product  
    WHERE total_sold > 100  
)  
SELECT *  
FROM top_products;
```

-- Here, the first CTE calculates total sales per product, **and** the second CT

In []: -- Example 3 – Recursive CTE:

```
WITH RECURSIVE countdown AS (  
    SELECT 5 AS num  
    UNION ALL  
    SELECT num - 1  
    FROM countdown  
    WHERE num > 1  
)  
SELECT * FROM countdown;
```

19. VIEWS & MATERIALIZED VIEWS

Views

A View is like a saved query that you can treat as a virtual table.

It does not store actual data; instead, it pulls fresh data every time you query it.

Useful when:

- You want to simplify complex queries (e.g., join multiple tables but reuse the result often).
- You want to give users restricted access to only specific columns/rows.
- You want to present data in a specific format without modifying the original tables.

Key Points:

- Lightweight (doesn't store data).
- Always up-to-date because it fetches live data from the underlying tables.
- If base tables change, the view reflects it automatically.

Materialized Views

A Materialized View is like a snapshot of a query result that gets stored physically in the database.

Unlike normal views, it stores the data.

Useful when:

- The query is very expensive (e.g., joining millions of rows).
- You want to query the result multiple times without recalculating.
- Data does not change too frequently.

Key Points:

- Faster performance for repeated queries.
- Needs manual refreshing (to update with the latest data).
- Takes up storage because it stores data.

Creating a View

```
In [ ]: -- SQL Syntax
CREATE VIEW view_name AS
SELECT column1, column2
```

```
FROM table_name  
WHERE condition;
```

```
-- Example:  
CREATE VIEW high_salary_employees AS  
SELECT name, department, salary  
FROM employees  
WHERE salary > 60000;
```

Using the View

```
In [ ]: SELECT * FROM high_salary_employees;
```

Updating a View (Replace/Modify)

```
In [ ]: CREATE OR REPLACE VIEW view_name AS  
SELECT ...
```

Dropping a View

```
In [ ]: DROP VIEW view_name;
```

Creating a Materialized View

```
In [ ]: -- SQL Syntax  
CREATE MATERIALIZED VIEW mv_name AS  
SELECT column1, column2  
FROM table_name  
WHERE condition;  
  
-- Example:  
CREATE MATERIALIZED VIEW sales_summary AS  
SELECT product_id, SUM(quantity) AS total_sold  
FROM sales  
GROUP BY product_id;
```

Refreshing a Materialized View

```
In [ ]: REFRESH MATERIALIZED VIEW mv_name;
```

Dropping a Materialized View

```
In [ ]: DROP MATERIALIZED VIEW mv_name;
```

pgAdmin GUI Steps

For Views:

1. Open pgAdmin → connect to your database.
2. Navigate to **Schemas → public → Views**.
3. Right-click **Views → Create → View**.
4. Fill in:
 - **Name:** (e.g., high_salary_employees)
 - **Definition tab:** write the SQL query (SELECT statement).
5. Click **Save**.
6. To run it: Right-click the view → **View/Edit Data → All Rows**.

For Materialized Views:

1. In pgAdmin, go to **Schemas → public → Materialized Views**.
2. Right-click → **Create → Materialized View**.
3. Enter the **Name** and **Definition** (SQL query).
4. Click **Save**.
5. To update data: Right-click the materialized view → **Refresh**.
6. To query: Run `SELECT * FROM mv_name;` in Query Tool or **View/Edit Data**.

20. STORED PROCEDURES

A **Stored Procedure** is a block of SQL code (sometimes mixed with logic like loops, conditions, and variables) that is saved in the database and can be executed whenever needed.

Think of it like a **function in programming**, but stored inside the database.

It allows you to bundle multiple SQL statements together (insert, update, delete, complex logic).

Why Use Stored Procedures?

- **Automation** – Perform repetitive tasks (e.g., monthly report generation).
- **Performance** – Runs directly inside the database (less network overhead).
- **Reusability** – Write once, call many times.
- **Security** – Can restrict direct table access and only expose procedures.

Difference from Functions:

- **Functions** return a value and are mostly used in queries.
- **Procedures** don't return a value (but can use OUT parameters) and are executed with `CALL`.

```

In [ ]: -- SQL Syntax
        Creating a Stored Procedure
        CREATE PROCEDURE procedure_name (parameters)
        LANGUAGE plpgsql
        AS $$
        BEGIN
            -- SQL statements go here
        END;
        $$;

        -- Example: Insert Employee Record
        CREATE PROCEDURE add_employee(emp_name TEXT, emp_salary NUMERIC)
        LANGUAGE plpgsql
        AS $$
        BEGIN
            INSERT INTO employees (name, salary)
            VALUES (emp_name, emp_salary);
        END;
        $$;

```

Call the Procedure

```

In [ ]: CALL add_employee('Joe', 75000);

```

```

In [ ]: -- Example 2: Procedure with Conditional Logic

        CREATE PROCEDURE update_salary(emp_id INT, new_salary NUMERIC)
        LANGUAGE plpgsql
        AS $$
        BEGIN
            IF new_salary < 30000 THEN
                RAISE NOTICE 'Salary too low, update rejected!';
            ELSE
                UPDATE employees
                SET salary = new_salary
                WHERE id = emp_id;
            END IF;
        END;
        $$;

        -- Execute:

        CALL update_salary(3, 90000);

```

Dropping a Procedure

```

In [ ]: -- SQL Syntax
        DROP PROCEDURE procedure_name(parameters);
        -- Example:
        DROP PROCEDURE add_employee(TEXT, NUMERIC);

```

21. Functions

A **Function** in PostgreSQL is a reusable block of SQL code (sometimes with logic, conditions, and loops) that is stored in the database and can be used anywhere in queries.

Key Points:

- Functions always return a value (unlike procedures).
- They can take input parameters and return a single value or a table.
- Functions can be written in multiple languages (SQL, PL/pgSQL, Python, etc.), but most commonly in PL/pgSQL.

Common Use Cases:

- Custom calculations
- Data transformations
- Wrapping complex queries
- Reusing business logic across multiple queries

Difference between Function and Procedure:

Feature	Function	Procedure
Return Value	Must return a value (scalar or table)	Doesn't return a value (but can use OUT parameters)
Call Method	Used inside queries (<code>SELECT my_function(...)</code>)	Called with <code>CALL procedure_name(...)</code>
Usage	Data retrieval, transformations	Multi-step operations (insert/update/delete + logic)

```
In [ ]: -- SQL Syntax
CREATE OR REPLACE FUNCTION function_name (parameters)
RETURNS return_type
LANGUAGE plpgsql
AS $$
BEGIN
    -- SQL code and logic
    RETURN something;
END;
$$;
```

Example 1: Simple Function to Add Two Numbers

```
In [ ]: CREATE OR REPLACE FUNCTION add_numbers(a INT, b INT)
RETURNS INT
```

```

LANGUAGE plpgsql
AS $$
BEGIN
    RETURN a + b;
END;
$$;
-- Example: Call the Function
SELECT add_numbers(5, 10);

```

Example 2: Function Returning Text

```

In [ ]: CREATE OR REPLACE FUNCTION greet_user(name TEXT)
        RETURNS TEXT
        LANGUAGE plpgsql
        AS $$
        BEGIN
            RETURN 'Hello, ' || name || '!';
        END;
        $$;
        -- Example: Call the Function
        SELECT greet_user('Joe');

```

Example 3: Function Returning a Table

```

In [ ]: CREATE OR REPLACE FUNCTION get_high_salary_employees(min_salary NUMERIC)
        RETURNS TABLE (emp_id INT, emp_name TEXT, emp_salary NUMERIC)
        LANGUAGE plpgsql
        AS $$
        BEGIN
            RETURN QUERY
            SELECT id, name, salary
            FROM employees
            WHERE salary > min_salary;
        END;
        $$;
        -- Example: Call the Function
        SELECT * FROM get_high_salary_employees(60000);

```

Dropping a Function

```

In [ ]: -- SQL Syntax
        DROP FUNCTION function_name(parameters);

        -- Example:
        DROP FUNCTION add_numbers(INT, INT);

```

22. TRIGGERS

A **Trigger** in PostgreSQL is like a "watchdog" that automatically runs a function when a specific event happens on a table or view.

Triggers help automate tasks such as:

- Keeping logs or audit trails (who changed what, when).
- Enforcing business rules.
- Automatically updating or validating data.

They are always linked to a **table/view + event + function**.

A trigger does not contain logic itself – it only calls a trigger function.

Events that can fire a Trigger:

- **INSERT** → when a new row is added.
- **UPDATE** → when an existing row is modified.
- **DELETE** → when a row is removed.
- **TRUNCATE** → when all rows in a table are deleted.

When does the trigger run?

- **BEFORE** → before the event happens (e.g., validate/modify data).
- **AFTER** → after the event happens (e.g., logging).
- **INSTEAD OF** → replaces the action (used in views).

```
In [ ]: -- SQL Syntax
General Syntax
CREATE TRIGGER trigger_name
{ BEFORE | AFTER | INSTEAD OF } { event [ OR event ... ] }
ON table_name
[ FOR EACH ROW | FOR EACH STATEMENT ]
EXECUTE FUNCTION function_name();
```

Steps to Create a Trigger

1. **Create a trigger function** (must return `TRIGGER`).
2. **Create a trigger** that calls this function when a specific event occurs.

Example 1: Log Updates

```
In [ ]: -- Step 1: Create a log table
CREATE TABLE employee_logs (
    log_id SERIAL PRIMARY KEY,
    emp_id INT,
    old_salary NUMERIC,
    new_salary NUMERIC,
    changed_on TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```



```

-- Step 2: Create trigger function
CREATE OR REPLACE FUNCTION log_salary_changes()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    INSERT INTO employee_logs(emp_id, old_salary, new_salary)
    VALUES(OLD.id, OLD.salary, NEW.salary);
    RETURN NEW;
END;
$$;

-- Step 3: Create trigger
CREATE TRIGGER salary_update_trigger
AFTER UPDATE OF salary ON employees
FOR EACH ROW
EXECUTE FUNCTION log_salary_changes();

-- Whenever salary is updated in employees, a log entry is automatically ins

```

Example 2: Prevent Negative Salary (Validation)

```

In [ ]: CREATE OR REPLACE FUNCTION prevent_negative_salary()
RETURNS TRIGGER
LANGUAGE plpgsql
AS $$
BEGIN
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION 'Salary cannot be negative!';
    END IF;
    RETURN NEW;
END;
$$;

CREATE TRIGGER validate_salary_trigger
BEFORE INSERT OR UPDATE ON employees
FOR EACH ROW
EXECUTE FUNCTION prevent_negative_salary();
-- Before inserting/updating, if salary < 0 → the transaction fails.

```

Dropping a Trigger

```

In [ ]: -- SQL Syntax
DROP TRIGGER trigger_name ON table_name;

-- Example:
DROP TRIGGER salary_update_trigger ON employees;

```

23. WINDOW FUNCTIONS

Window Function

A **Window Function** performs a calculation across a set of rows related to the current row, but **without collapsing rows** like `GROUP BY` does.

They are often used for:

- Ranking
- Running totals
- Moving averages
- Percentiles
- Lag/Lead values
- Cumulative calculations

The "set of rows" is called a **window** (defined using `OVER (...)`).

Key Difference from Aggregate Functions:

- **Aggregate:** Groups rows → returns one row per group.
- **Window Function:** Keeps all rows → adds extra calculated column.

```
In [ ]: -- General Syntax

function_name (expression)
OVER (
    PARTITION BY column_name    -- optional, divides data into groups
    ORDER BY column_name        -- optional, defines row order
    ROWS BETWEEN ...           -- optional, frame definition
);

-- PARTITION BY: Like GROUP BY but does not reduce rows.
-- ORDER BY: Defines the sequence of rows.
-- ROWS BETWEEN: Defines the "window frame" (e.g., last 3 rows, unbounded).
```

Common Window Functions

a) Ranking Functions

```
In [ ]: -- ROW_NUMBER(): Assigns a unique number to each row.
-- RANK(): Gives ranking, skips numbers if there are ties.
-- DENSE_RANK(): Like rank, but no gaps in numbering.

-- Example:
SELECT id, name, salary,
       ROW_NUMBER() OVER (ORDER BY salary DESC) AS row_num,
       RANK()        OVER (ORDER BY salary DESC) AS rank,
```

```
DENSE_RANK() OVER (ORDER BY salary DESC) AS dense_rank
FROM employees;
```

Table Example

id	name	salary	row_num	rank	dense_rank
1	Ali	9000	1	1	1
2	Sara	8000	2	2	2
3	John	8000	3	2	2
4	Mary	7000	4	4	3

b) Aggregate Window Functions

```
In [ ]: -- SUM(), AVG(), MIN(), MAX(), COUNT()
-- Unlike normal aggregates, they don't collapse rows.

-- Example:
SELECT id, department, salary,
       SUM(salary) OVER (PARTITION BY department) AS dept_total,
       AVG(salary) OVER (PARTITION BY department) AS dept_avg
FROM employees;
```

Table Example

id	dept	salary	dept_total	dept_avg
1	HR	4000	9000	4500
2	HR	5000	9000	4500
3	IT	6000	14000	7000
4	IT	8000	14000	7000

c) Running Totals & Moving Averages

```
In [ ]: SELECT id, name, salary,
       SUM(salary) OVER (ORDER BY id) AS running_total,
       AVG(salary) OVER (ORDER BY id ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS dept_avg
FROM employees;
```

d) LAG() and LEAD()

```
In [ ]: -- LAG(column, n): Look back at previous rows.
-- LEAD(column, n): Look ahead at next rows.

-- Example:
SELECT id, name, salary,
```

```
LAG(salary, 1) OVER (ORDER BY id) AS prev_salary,
LEAD(salary, 1) OVER (ORDER BY id) AS next_salary
FROM employees;
```

Table Example

id	name	salary	prev_salary	next_salary
1	Ali	4000	NULL	5000
2	Sara	5000	4000	6000
3	John	6000	5000	8000
4	Mary	8000	6000	NULL

Window Functions:

Function	Purpose
ROW_NUMBER()	Sequential row numbering
RANK()	Ranking with gaps
DENSE_RANK()	Ranking without gaps
NTILE(n)	Divides rows into n buckets
SUM(), AVG(), MIN(), MAX(), COUNT()	Aggregates without collapsing rows
LAG()	Value from previous row
LEAD()	Value from next row
FIRST_VALUE()	First value in window
LAST_VALUE()	Last value in window

24. INDEXES

Indexes in PostgreSQL are data structures that speed up searching, filtering, and joining by reducing the amount of data scanned. Think of them like the index of a book — instead of scanning all pages, you jump straight to the right section.

B-Tree Index

- Most common index in PostgreSQL.

- Works well for equality (=) and range queries (<, >, BETWEEN).
- Balanced tree structure → log(n) lookup time.

```
In [ ]: -- Example:

-- Create B-Tree index on salary column
CREATE INDEX idx_salary ON employees(salary);

-- Query uses the index
SELECT * FROM employees WHERE salary > 5000;
```

Hash Index

- Optimized for equality lookups (=).
- Faster than B-Tree for equality, but not for ranges.
- Often used for exact match searches.

```
In [ ]: -- Example:

-- Create a hash index on name column
CREATE INDEX idx_employee_name_hash ON employees USING HASH(name);

-- Query uses the hash index
SELECT * FROM employees WHERE name = 'Sara';
```

GIN (Generalized Inverted Index)

- Best for full-text search and array/jsonb columns.
- Quickly finds rows where values exist inside complex data.

```
In [ ]: -- Example:

-- Array column example
CREATE INDEX idx_tags_gin ON articles USING GIN(tags);

-- Query uses the GIN index
SELECT * FROM articles WHERE tags @> '{postgresql}';
```

4) GiST (Generalized Search Tree)

- Flexible index type.
- Used for geometric data, full-text search, and range types.

- Example: spatial queries, searching inside a range.

```
In [ ]: -- Example:

-- For geometric data (points)
CREATE INDEX idx_location_gist ON places USING GIST(location);

-- For range queries
CREATE INDEX idx_date_range_gist ON reservations USING GIST(daterange);
```

Summary:

- Use B-Tree for most queries.
- Use Hash for exact lookups only.
- Use GIN for text search, arrays, JSONB.
- Use GiST for ranges, geometric, custom search.

25. USER / ROLES MANAGEMENT

PostgreSQL doesn't separate users and groups → everything is a **role**.

A role can:

- Log in (like a user) → `LOGIN` privilege.
- Own objects (tables, databases, schemas).
- Be granted to other roles (works like groups).

Privileges

Control what actions a role can perform.

- `SELECT` , `INSERT` , `UPDATE` , `DELETE` , `CREATE` , `CONNECT` .

Authentication Methods

- Password-based (`md5` , `scram-sha-256`).
- OS-based (`peer`).
- Certificate-based, etc.

Configured in the file → `pg_hba.conf` .

Best Practices

- Use least privilege principle (give only necessary access).

- Avoid using the default postgres superuser for daily tasks.
- Enforce strong password policies.

1. Create User / Role

```
In [ ]: -- Create a login role (user)
CREATE ROLE data_analyst WITH LOGIN PASSWORD 'securePass123';

-- Create a group role
CREATE ROLE read_only;
```

2. Grant Role Membership

```
In [ ]: -- Make user part of group
GRANT read_only TO data_analyst;
```

3. Grant Privileges

```
In [ ]: -- Grant access to a database
GRANT CONNECT ON DATABASE sales_db TO data_analyst;

-- Grant table-level permissions
GRANT SELECT, INSERT ON employees TO data_analyst;

-- Grant schema-level permissions
GRANT USAGE ON SCHEMA public TO read_only;
```

4. Revoke Privileges

```
In [ ]: REVOKE INSERT ON employees FROM data_analyst;
```

5. Alter User

```
In [ ]: -- Change password
ALTER ROLE data_analyst WITH PASSWORD 'newSecurePass456';

-- Remove login capability
ALTER ROLE read_only NOLOGIN;
```

6. Drop User / Role

```
In [ ]: DROP ROLE data_analyst;
```

26. CONNECTING PostgreSQL TO PYTHON

Why connect Python with PostgreSQL?

- Automate data extraction for analysis.
- Perform ETL (Extract, Transform, Load) tasks.
- Integrate SQL queries with Pandas, NumPy, and Scikit-learn pipelines.
- Build AI/ML models directly from database data.

Libraries commonly used:

- **psycopg2** → Low-level PostgreSQL adapter (most popular).
- **SQLAlchemy** → High-level ORM (Object Relational Mapper) for easier queries.
- **pandas** → `read_sql()` can load query results into DataFrames.

Workflow:

1. Install required library: `pip install psycopg2-binary sqlalchemy pandas`
2. Connect to the PostgreSQL server with `host` , `dbname` , `user` , `password` , `port` .
3. Run queries → fetch results → use in Python.

```
In [ ]: from sqlalchemy import create_engine
import pandas as pd

# Create engine (replace with your credentials)
engine = create_engine("postgresql://postgres:your_password@localhost:5432/s

# Query with pandas
df = pd.read_sql("SELECT name, salary FROM employees WHERE salary > 50000;",
print(df)
```

pgAdmin GUI Steps (to prepare for Python connection)

Before Python can connect, ensure:


- **Enable database & user access**
 - Open **pgAdmin** → Expand **Databases** → Ensure your database (e.g., `sales_db`) exists.
 - Expand **Login/Group Roles** → Ensure your Python user (e.g., `postgres` or `data_analyst`) exists and has:
 - `CONNECT` privilege on the database.
 - `SELECT` privilege on required tables.


- **Check connection parameters**
 - Database name (`sales_db`).
 - Username (`postgres` or custom user).
 - Password (set in pgAdmin).
 - Host (`localhost` if local).
 - Port (`5432` by default).
- **Optional: Allow remote connections**
 - Edit `pg_hba.conf` to allow remote IPs.
 - Restart PostgreSQL service.

WHAT NEXT...

You've now learned how to connect PostgreSQL with Python.
But to unlock the full power of **database + Python**, you should also learn:

- **SQLAlchemy** → A modern Python toolkit & ORM for working with databases.
 - Learn how to create models, insert/update/delete records, and manage relationships.

 Check out my [SQLAlchemy Cheatsheet](#)
- **Pandas** → The most popular Python library for data analysis.
 - Learn how to query PostgreSQL data directly into DataFrames and perform advanced analysis.

 Explore my [Pandas Cheatsheet](#)