

# ★ DAY 5 — DATABASES

## ◇ PART A — INTRO TO DATABASES

By the end of this part, students should **clearly understand WHY databases exist** and **why files are not enough**.

### ⌚ 1 WHAT IS A DATABASE?

#### 🔧 Definition

A **database** is a **place to store data permanently** so that:

- Data does not disappear
- Data can be searched easily
- Data can be updated safely
- Many users can use it at the same time



“A database is like a **smart digital cupboard** for your data.”

### ⌚ Real-Life Analogy

Imagine:

- You write student details on paper
- You keep papers in a bag

Problems:

- ✗ Papers can tear
- ✗ Hard to find one student

- ✗ No backup
  - ✗ Multiple people can't edit
- 📌 Database solves all these problems.

## 📁 2 WHY FILES (TXT, JSON, CSV) ARE NOT ENOUGH

### ⌚ Question:

“Why can't we just use files to store data?”

### ✗ Problems with Files

Problem	Example
Data loss	File deleted accidentally
No structure	Everyone writes differently
Slow search	Reading entire file
No security	Anyone can open
No multi-user	Two people overwrite data
No relationships	Hard to connect data



“Files are okay for practice, but **real apps will break** if we use files.”

## 💾 3 WHAT PROBLEM DATABASES SOLVE

### Databases give us:

- ✓ Permanent storage
- ✓ Fast search
- ✓ Organized data
- ✓ Safe updates

- ✓ Multiple users at once
- ✓ Security
- ✓ Backup & recovery

📌 Example:

Instagram:

- Millions of users
- Millions of posts
- Likes, comments, messages

👉 Impossible without a database



## PERSISTENT STORAGE (VERY IMPORTANT)

❓ What does "Persistent" mean?

Persistent = Data stays even after app/server is stopped

Example:

- Backend stopped ✗
- Laptop restarted 🔋
- Data still exists ✓



"If data disappears after refresh, it's not a database."

### ⌚ Example Comparison

Storage Type	Data after restart
Variable	✗ Lost

Array	 Lost
File	 Risky
Database	 Safe

## 5 TYPES OF DATA (STRUCTURED vs UNSTRUCTURED)

### Structured Data

Data with **fixed format**

Example:

id	name	age
1	Alex	21

- ✓ Easy to search
- ✓ Fixed columns

Used in:

- Banks
- Schools
- Finance apps

### Unstructured Data

Data with **no fixed format**

Examples:

- Images
- Videos
- Messages
- JSON documents

Used in:

- Social media
- Chat apps
- Logs



“Not all data fits in tables.”



6

## TYPES OF DATABASES (HIGH LEVEL)

### Two Main Types (IMPORTANT)

Type	Name
SQL	Relational Database
NoSQL	Non-Relational Database

We will learn:

- ✓ PostgreSQL (SQL)
- ✓ MongoDB (NoSQL)



7

## SQL DATABASES (RELATIONAL)

### Key Idea:

Data is stored in **tables**

Example:

#### Students Table

id name email

## Features:

- ✓ Fixed structure
- ✓ Strong rules
- ✓ Relationships between tables

Used when:

- Data is structured
- Rules are strict
- Financial accuracy is needed

 Example Apps:

- Banking apps
- College systems
- Payroll

## 8 NOSQL DATABASES (MONGODB)

### Key Idea:

Data is stored as **documents**

Example (JSON-like):

```
{  
  "name": "Alex",  
  "email": "alex@mail.com",  
  "skills": ["React", "FastAPI"]  
}
```

## Features:

- ✓ Flexible structure
- ✓ Easy to scale
- ✓ Faster for large apps

Used when:

- Data changes often
- App grows fast
- Flexible data is needed

👤 Example Apps:

- Instagram
- Netflix
- Uber

## ⚖️ 9 SQL vs NoSQL (VERY IMPORTANT)

Feature	SQL	NoSQL
Structure	Fixed	Flexible
Schema	Required	Optional
Scaling	Vertical	Horizontal
Best for	Accuracy	Speed & scale
Example	PostgreSQL	MongoDB

👤 Tell students:

“There is NO best database — only best choice for the problem.”



10

## ACID vs FLEXIBILITY (SIMPLE LANGUAGE)

### ACID (SQL Databases)

ACID = Rules to protect data

- ✓ Data is always correct
- ✓ Transactions are safe
- ✓ No partial updates

Example:

- Money transfer
- Exam results

### Flexibility (NoSQL)

- ✓ Faster writes
- ✓ Flexible structure
- ✓ Handles large data

Example:

- Social posts
- Chat messages

👤 Simple line:

“SQL = Safety first

NoSQL = Speed first”

## 1 1 SCALING IDEA (BASIC)

### Vertical Scaling

Add more power to one machine

 Example:

- More RAM
- Better CPU

 Expensive

 Limited

### Horizontal Scaling

Add more machines

 Example:

- Multiple servers
- Load sharing

✓ Cheap

✓ Unlimited growth

 MongoDB is great at this.

## MINI CHEAT SHEET — PART A

Concept	Meaning
Database	Permanent data storage
Persistent	Data survives restart
SQL	Table-based, strict

NoSQL	Document-based, flexible
ACID	Data safety rules
Vertical scaling	Bigger machine
Horizontal scaling	More machines



## FINAL MINDSET (VERY IMPORTANT)

Tell students:

“Backend without database is incomplete.  
Database is the memory of your application.”

## ◊ PART-B — MongoDB Setup & Verification



Goal:

By the end of this part:

- MongoDB is installed correctly
- MongoDB is running
- They can talk to MongoDB
- They understand *what is happening*, not just commands



### 1 Before We Touch MongoDB



“MongoDB is NOT a programming language.  
MongoDB is a **database server** that runs in the background.”

Think of it like:

- WhatsApp app → frontend
- WhatsApp servers → backend
- **MongoDB → place where data is stored permanently**



**2**

## What Needs to Be Installed (Checklist)

On **Mac**, for MongoDB work we need:

Item	Why needed
MongoDB Server	To store data
MongoDB Shell (mongosh)	To talk to MongoDB
Terminal	To run commands
VS Code	To write backend code
Python	For FastAPI (already done)

👉 We will **VERIFY**



**3**

## Step 1 — Check if MongoDB is Installed

⌚ Open Terminal and run:

```
mongod --version
```



If MongoDB is installed, you'll see:

Something like:

```
db version v7.0.x
```

✓ This means **MongoDB Server exists**

## If command not found:

MongoDB server is **NOT installed**

## Step 2 — Check MongoDB Shell

MongoDB is useless if we can't talk to it.

**Run:**

```
mongosh
```

## Expected output:

You'll see something like:

```
Current Mongosh Log ID: ...
Connecting to: mongodb://127.0.0.1:27017
>
```

That > symbol means:

 **You are inside MongoDB**

 :

“This is like opening Python REPL or Node console —  
now we are talking directly to MongoDB.”

## If mongosh not found:

Shell missing → install required



## 5 What is Running Right Now?

Let's clarify **confusion students always have**:

Thing	Meaning
mongod	MongoDB server (background)
mongosh	MongoDB shell (your chat window)
Port 27017	Default MongoDB door



Visualization:

Your Mac

```
|-- MongoDB Server (mongod)  ← running silently
  └-- Terminal (mongosh)      ← you typing commands
```



## 6 Verify MongoDB is Actually Running

Inside mongosh, type:

```
db
```

**Output:**

```
test
```

This means:

- MongoDB is alive
- You are connected

- Default database is test
- ✓ Good sign



7

## Create a Test Database (NO DATA YET)

Run:

```
use school
```

Output:

```
switched to db school
```



Explain:

“MongoDB creates databases **lazily**.  
Database exists only when we insert data.”



8

## What is a Database? (MongoDB Style)

Term	Meaning
Database	Big container
Collection	Like a table
Document	Like a row (JSON)



Visualization:

```
Database: school
  └ Collection: students
    └ Document: { name, age, email }
```

## ✍️ 9 Create First Collection + Document

### Insert one student ↗

```
db.students.insertOne({  
  name: "Alex",  
  
  age: 20,  
  email: "alex@mail.com"  
})
```

### Output:

```
acknowledged: true  
insertedId: ObjectId(...)
```

🎉 MongoDB is WORKING

## 🔍 1 0 Read Data (Important)

```
db.students.find()
```

You will see:

```
[  
{  
  _id: ObjectId("..."),  
  name: "Alex",  
  age: 20,  
  email: "alex@mail.com"  
}  
]
```

💡 Explain:

- `_id` is auto-generated
- MongoDB stores **JSON-like documents**
- No schema forced

## 📋 1 1 MINI CHEAT SHEET — MongoDB Basics

Command	Meaning
<code>mongosh</code>	Open MongoDB shell
<code>use dbName</code>	Switch DB
<code>db</code>	Current DB
<code>show dbs</code>	List databases
<code>db.collection.insertOne()</code>	Insert
<code>db.collection.find()</code>	Read

## ⚠ 1 2 Important MongoDB Characteristics (Explain Slowly)

### ✗ No fixed schema

```
{ name: "Alex" }  
{ name: "Sam", age: 22 }
```

✓ Both valid

### ✗ No tables

✓ Collections instead

## No joins by default

✓ Designed for flexibility + speed



**1**

**3**

## MongoDB vs SQL (Student Friendly)

MongoDB	SQL
Flexible	Strict
JSON documents	Tables
Schema-less	Schema-based
Scales horizontally	Mostly vertical



“MongoDB is like a flexible notebook  
SQL is like a ruled register.”

## Let's Practise some MongoDB



## LAYER 1 — MongoDB MENTAL MODEL

Read this slowly.

### ? What MongoDB REALLY is

MongoDB is:

- a **JSON document storage engine**
- running as a **background service**
- optimized for **real-world app data**

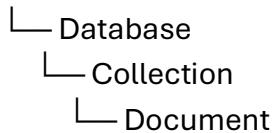
It is **NOT**:

- Excel
- SQL tables
- rows & columns thinking



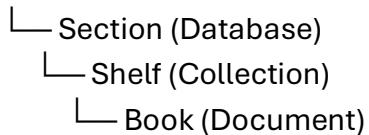
## MongoDB Structure (Burn this into your brain)

MongoDB Server



**Real-life analogy:**

Library



👉 MongoDB doesn't care:

- what fields a book has
- how many pages
- what order

That's the **power and danger**.



## What is a DOCUMENT really?

A MongoDB document is just:

```
{  
  name: "Rahul",  
  age: 21,
```

```
skills: ["Python", "MongoDB"]  
}
```

It's:

- NOT a class
- NOT a table row
- Just **stored JSON**

MongoDB **does not ask permission**.



## KEY DIFFERENCE THAT BLOCKS CONFIDENCE

### SQL mindset X

“First define structure, then insert data”

### MongoDB mindset ✓

“Insert data first, structure evolves naturally”

This shift takes time — **you're normal**.



## LAYER 2 — CORE COMMANDS (DEEP COMFORT)

We will now go **very slow** and explain *why*, not just *how*.

## ◊ 1. Switching Databases

```
use college
```

🧠 Meaning:

“Hey MongoDB, from now on, keep my data in a box called college.”

⚠ Database is NOT created yet.

## ◊ 2. Collections Are Created Automatically

```
db.students.insertOne({  
  name: "Ananya",  
  age: 22  
})
```

🧠 What MongoDB did:

1. Created college database
2. Created students collection
3. Inserted document
4. Added \_id

SQL people find this scary — MongoDB people love it.

## ◊ 3. Why \_id Exists (Important)

Every document has:

```
_id: ObjectId("...")
```

Why?

- MongoDB needs **unique identity**
- Like Aadhaar for documents

You don't manage it → MongoDB does.

## ◊ 4. Reading Data Slowly

```
db.students.find()
```

Means:

“Give me **all documents** from students collection”

MongoDB does **not** care about column order.

## ◊ 5. Filtering (THIS BUILDS CONFIDENCE)

```
db.students.find({ age: 22 })
```

Meaning:

“Only give documents where age is 22”

### Multiple conditions:

```
db.students.find({  
    age: 22,  
    name: "Ananya"  
})
```

This is **AND** by default.

## ◊ 6. Projection (Selective Fields)

```
db.students.find(  
  { age: 22 },  
  { name: 1, _id: 0 }  
)
```

💡 Meaning:

- 1 → include
- 0 → exclude

MongoDB forces `_id` unless you hide it.

## ◊ 7. Update (THIS IS WHERE FEAR COMES)

**Update ONE document:**

```
db.students.updateOne(  
  { name: "Ananya" },  
  { $set: { age: 23 } }  
)
```

💡 `$set` means:

“Only update this field — don’t destroy others”

⚠ Without `$set`, MongoDB **replaces the entire document**.

This single rule causes most beginner disasters.

## ◊ 8. Delete (Safe First)

```
db.students.deleteOne({ name: "Ananya" })
```

One match → delete one.

## ◊ 9. Schema Flexibility (See the Magic)

```
db.students.insertOne({  
  name: "Rohit",  
  skills: ["Java", "Node"],  
  
  active: true  
})
```

Same collection.

Different structure.

Still valid.



## LAYER 2 PRACTICE (MANDATORY)

Before Part-C, you must be able to do this **without thinking**:

- ✓ Create DB
- ✓ Insert different shaped documents
- ✓ Find with filter
- ✓ Update using \$set
- ✓ Delete safely

## ◇ PART-C — CONNECT FASTAPI WITH MONGODB

⌚ Goal:

You should clearly understand **how FastAPI talks to MongoDB**, how data flows, and how real APIs store data permanently.



### 1 THEORY — WHAT ARE WE DOING?

#### Real-World Story (IMPORTANT)

Think like this 👇

```
React → FastAPI → MongoDB  
(UI)       (Brain)      (Memory)
```

- React: asks for data
- FastAPI: processes request
- MongoDB: stores & returns data permanently

➡️ **FastAPI itself does NOT store data**

If server restarts → memory is gone

So we need **MongoDB = permanent storage**

### ❓ Why we cannot use MongoDB shell directly in backend?

Mongo shell is for:

- humans
- manual testing

Backend needs:

- automatic
- async
- non-blocking
- production-safe access

👉 So we use a **MongoDB DRIVER**



## 2 IMPORTANT CONCEPT — DRIVER & ASYNC

**MongoDB + FastAPI uses:**

Motor (Async MongoDB Driver)

Why async?

- FastAPI is async
- MongoDB queries can be slow
- Async = server doesn't freeze

📌 Motor = bridge between FastAPI and MongoDB

## 3 BACKEND PROJECT STRUCTURE (VERY IMPORTANT)

We now slightly upgrade backend structure 👈

```
backend/
├── main.py
├── database.py      ➔ NEW (MongoDB connection)
└── routers/
    └── students.py
└── models/
    └── student.py    ➔ NEW (data structure)
```

👉 This is **industry-standard**, not random.

## 🔗 4 DATABASE CONNECTION (CORE FILE)

### 📁 backend/database.py

```
from motor.motor_asyncio import AsyncIOMotorClient

MONGO_URL = "mongodb://localhost:27017"

client = AsyncIOMotorClient(MONGO_URL)

database = client.college

student_collection = database.students
```

#### 🌐 Understand this properly:

- AsyncIOMotorClient → connects to MongoDB service
- college → database name
- students → collection name

 MongoDB creates DB & collection **automatically**

## Why separate database.py?

- ✓ Clean code
- ✓ Reusable
- ✓ Easy to change DB later
- ✓ Professional backend practice

## **5 DATA MODEL (HOW DATA SHOULD LOOK)**

### backend/models/student.py

```
from pydantic import BaseModel, EmailStr
from typing import Optional

class Student(BaseModel):
    name: str
    email: EmailStr
    age: int
```

### **IMPORTANT CLARITY**

This model:

- does NOT create DB
- does NOT insert data
- only **validates incoming data**

👉 MongoDB stores JSON

👉 Pydantic ensures correctness

## 🔗 6 ROUTER — REAL CRUD WITH MONGODB

### 📁 backend/routers/students.py

```
from fastapi import APIRouter, HTTPException
from models.student import Student
from database import student_collection

router = APIRouter(prefix="/students", tags=["Students"])

@router.post("/")
async def create_student(student: Student):
    student_dict = student.dict()
    result = await student_collection.insert_one(student_dict)
    return {
        "message": "Student created",
        "id": str(result.inserted_id)
    }

@router.get("/")
async def get_students():
    students = []
    cursor = student_collection.find()

    async for student in cursor:
        student["_id"] = str(student["_id"])

    return {"students": students}
```

```
students.append(student)

return students
```



## LET'S UNDERSTAND THE FLOW (VERY IMPORTANT)

### POST /students

- 1 React sends JSON
- 2 FastAPI validates using Student
- 3 Converted to dict
- 4 Inserted into MongoDB
- 5 MongoDB returns \_id

📌 Permanent storage achieved

### GET /students

- 1 MongoDB returns cursor
- 2 Cursor iterated asynchronously
- 3 \_id converted to string
- 4 JSON returned to frontend

⚠️ MongoDB \_id is not JSON-serializable → must convert

## 7 MAIN ENTRY FILE (YOU WARNED ME — SO HERE IT IS)

### backend/main.py

```
from fastapi import FastAPI  
from routers import students  
  
app = FastAPI(title="Student API")  
  
app.include_router(students.router)
```

- ✓ Router connected
- ✓ Backend complete
- ✓ Clean startup

## 8 TEST FLOW (YOU MUST DO THIS)

### Step 1 — Start MongoDB

```
mongod
```

### Step 2 — Start FastAPI

```
uvicorn main:app --reload
```

### Step 3 — Open Swagger

<http://localhost:8000/docs>

- ✓ Create student
- ✓ Fetch students
- ✓ Check MongoDB shell

## MINI CHEAT SHEET (MEMORIZE THIS)

Concept	Meaning
Motor	Async MongoDB driver
Collection	Like table (but flexible)
Document	JSON object
Pydantic	Validation layer
Async	Non-blocking
Cursor	MongoDB iterator
_id	Unique identifier

## 1 COMMON BEGINNER MISTAKES (READ CAREFULLY)

- ✗ Forgetting async/await
- ✗ Returning ObjectId directly
- ✗ Mixing shell syntax in Python
- ✗ Writing DB logic inside main.py
- ✗ Skipping data validation

You avoided all of them 

## WHERE YOU ARE NOW (IMPORTANT)

You can now confidently say:

- ✓ “I know MongoDB shell”
  - ✓ “I know how FastAPI stores data”
  - ✓ “I understand backend → DB flow”
  - ✓ “I’m not scared of MongoDB anymore”
- .

## MINI ASSIGNMENT — USER REGISTRATION SYSTEM (MongoDB + FastAPI)

### What we are building

A **real backend flow** that every company uses:

- ✓ Register user
- ✓ Prevent duplicate emails
- ✓ Hash password
- ✓ Store user in MongoDB
- ✓ List users (without password)



# MENTAL MODEL (VERY IMPORTANT)

Flow in simple words:

Client (Swagger / React)

↓

FastAPI Endpoint

↓

Pydantic Validation

↓

Password Hashing

↓

MongoDB (users collection)

↓

Response (safe data)

MongoDB stores **raw data**, FastAPI controls **logic & safety**.



## PROJECT STRUCTURE

```
backend/
|
├── main.py
├── database.py
├── models/
│   └── user_model.py
└── routers/
    └── users.py
└── requirements.txt
```

## 1 INSTALL REQUIRED PACKAGES

Run this once:

```
pip install fastapi uvicorn motor python-dotenv passlib[bcrypt]
```

Why each?

- motor → async MongoDB
- passlib → password hashing
- dotenv → environment variables later

## 2 database.py (MongoDB Connection)

```
# database.py

from motor.motor_asyncio import AsyncIOMotorClient

MONGO_URL = "mongodb://localhost:27017"

client = AsyncIOMotorClient(MONGO_URL)

db = client["user_db"]

user_collection = db["users"]
```

🧠 Notes:

- Database name = user\_db
- Collection = users
- MongoDB auto-creates both

## 3 User Model (Validation Layer)

 models/user\_model.py

```
from pydantic import BaseModel, EmailStr
from typing import Optional

class UserCreate(BaseModel):
    name: str
    email: EmailStr
    password: str

class UserResponse(BaseModel):
    id: str
    name: str
    email: EmailStr
```

 Why this matters:

- UserCreate → input
- UserResponse → output (NO password)

## 4 Password Hashing Logic (SECURITY)

 routers/users.py

```
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")
def hash_password(password: str) -> str:
    return pwd_context.hash(password)
```

 Rule:

**Never store raw passwords. EVER.**

MongoDB will store:

\$2b\$12\$K9kE... (hashed)

## 5 User Router (FULL CRUD LOGIC)

 routers/users.py

```
from fastapi import APIRouter, HTTPException
from database import user_collection
from models.user_model import UserCreate, UserResponse
from bson import ObjectId
from passlib.context import CryptContext

router = APIRouter()

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def hash_password(password: str):
    return pwd_context.hash(password)

def user_serializer(user) -> dict:
    return {
        "id": str(user["_id"]),
        "name": user["name"],
        "email": user["email"]
    }

# REGISTER USER
@router.post("/register", response_model=UserResponse)
async def register_user(user: UserCreate):

    existing_user = await user_collection.find_one(
```

```

        {"email": user.email}
    )

if existing_user:
    raise HTTPException(
        status_code=400,
        detail="Email already registered"
    )

user_dict = user.dict()
user_dict["password"] = hash_password(user.password)

result = await user_collection.insert_one(user_dict)

new_user = await user_collection.find_one(
    {"_id": result.inserted_id}
)

return user_serializer(new_user)

# LIST USERS
@router.get("/users")
async def list_users():
    users = []
    async for user in user_collection.find():
        users.append(user_serializer(user))
    return users

```

### WHAT YOU LEARNED HERE:

- Duplicate email check
- Password hashing
- MongoDB insert
- MongoDB find
- Safe response

## 6 main.py (APP ENTRY POINT — IMPORTANT)

```
from fastapi import FastAPI  
from routers import users  
  
app = FastAPI()  
  
app.include_router(users.router, prefix="/api")
```

📌 You previously noticed when this was missing — good instinct.

## 7 TEST FLOW (DO THIS STEP-BY-STEP)

### Step 1 — MongoDB (already running ✓)

```
brew services list
```

### Step 2 — Start FastAPI

```
uvicorn main:app --reload
```

### Step 3 — Swagger

Open:

```
http://127.0.0.1:8000/docs
```

## Test APIs

### ◊ Register User

```
POST /api/register
{
  "name": "Yashas",
  "email": "yashas@mail.com",
  "password": "secret123"
}
```

### ◊ List Users

```
GET /api/users
```

- ✓ Password NOT visible
- ✓ Email unique
- ✓ Clean response

8

## VERIFY IN MONGODB SHELL

```
mongosh
use user_db
db.users.find().pretty()
```

You will see:

- \_id
- name
- email
- hashed password



## MINI CHEAT SHEET (SAVE THIS)

### MongoDB

```
db.users.find()  
db.users.findOne({email: "x@mail.com"})  
db.users.deleteMany({})
```

### Motor

```
insert_one()  
find_one()  
find()
```

### Security Rule

- Plain password
- Hashed password

## PostgreSQL: SQL Basics (Theory + CLI)

### Goal of Day 5

By the end of today, students should:

- Understand **how SQL databases think**
- Be comfortable using **PostgreSQL CLI**
- Write **basic SQL queries confidently**
- Clearly see the difference between **MongoDB vs SQL**

## 1. SQL Mental Model (VERY IMPORTANT)

### MongoDB vs PostgreSQL (Quick Comparison)

MongoDB	PostgreSQL
Database	Database
Collection	Table
Document	Row
Field	Column
Schema-less	Fixed Schema
JSON-like	Structured (Rows & Columns)

 PostgreSQL is like an Excel sheet with rules

## 2. Core SQL Concepts (Theory)

### Table

A **table** stores data in a structured format.

Example: students table

id	name	age	email
----	------	-----	-------

### Row

A **row** = one complete record

(1, "Rahul", 21, "[rahul@gmail.com](mailto:rahul@gmail.com)")

## Column

A **column** = one type of data

name → only names

age → only numbers

## 3. Primary Key (PK)

### What is a Primary Key?

- Uniquely identifies each row
- Cannot be NULL
- Cannot be duplicated

Example:

```
id SERIAL PRIMARY KEY
```

✓ SERIAL → auto-increment

✓ PRIMARY KEY → unique identity

## 4. Foreign Key (FK)

### What is a Foreign Key?

- Creates a **relationship** between tables
- Points to another table's primary key

Example:

- One **department**
- Many **students**

```
department_id INT REFERENCES departments(id)
```

👉 This enforces **data integrity**

## ▀ 5. PostgreSQL CLI (Hands-On)

### Step 1: Enter PostgreSQL

```
psql postgres
```

--or

```
psql -U postgres
```

### Step 2: Create a Database

```
CREATE DATABASE college;
```

### Step 3: Use the Database

```
\c college
```

### Step 4: List Databases

```
\l
```

## Step 5: List Tables

```
\dt
```

## 6. CREATE TABLE

### Create students table

```
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    email VARCHAR(150)
);
```

#### Explanation:

- VARCHAR(100) → string with max 100 chars
- INT → integer
- SERIAL → auto increment

## 7. INSERT (Add Data)

### Insert one row

```
INSERT INTO students (name, age, email)
VALUES ('Aman', 20, 'aman@gmail.com');
```

### Insert multiple rows

```
INSERT INTO students (name, age, email) VALUES
('Riya', 21, 'riya@gmail.com'),
```

```
('Kunal', 22, 'kunal@gmail.com'),  
('Neha', 19, 'neha@gmail.com');
```

## 8. SELECT (Read Data)

### Get all data

```
SELECT * FROM students;
```

### Get specific columns

```
SELECT name, age FROM students;
```

## 9. WHERE (Filter Data)

### Students older than 20

```
SELECT * FROM students  
WHERE age > 20;
```

### Student with specific name

```
SELECT * FROM students  
WHERE name = 'Riya';
```

### Combine conditions

```
SELECT * FROM students  
WHERE age >= 20 AND age <= 22;
```

## 10. ORDER BY (Sorting)

Sort by age (ascending)

```
SELECT * FROM students  
ORDER BY age;
```

Sort by age (descending)

```
SELECT * FROM students  
ORDER BY age DESC;
```

## 11. LIMIT (Restrict Results)

Get only first 2 students

```
SELECT * FROM students  
LIMIT 2;
```

Top 2 oldest students

```
SELECT * FROM students  
ORDER BY age DESC  
LIMIT 2;
```

## 12. Foreign Key Example (Mini)

 First: Forget SQL for 1 minute

Think about a **college** in real life.

## Real-life rules:

- A college has departments  
→ CSE, ECE, MECH
- A student must belong to ONE department
- A student cannot belong to a department that doesn't exist

 This rule is what **Foreign Key** enforces.

## Step 1: Departments table (Parent table)

```
CREATE TABLE departments (
    id SERIAL PRIMARY KEY,
    dept_name VARCHAR(100)
);
```

**What this means:**

Column	Meaning
id	Unique ID for each department
dept_name	Department name

Example data:

```
INSERT INTO departments (dept_name)
VALUES ('CSE'), ('ECE');
```

Now the table looks like:

id	dept_name
1	CSE
2	ECE

### Important

- PostgreSQL automatically creates id

- id is the **identity** of the department



## Step 2: Students table (Child table)

```
CREATE TABLE college_students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    department_id INT REFERENCES departments(id)
);
```

### What is department\_id?

👉 It points to departments.id

This is the **Foreign Key**.



## How tables are connected

departments	
id	dept_name
1	CSE
2	ECE

college_students		
id	name	department_id

## Meaning:

- department\_id = 1 → CSE
- department\_id = 2 → ECE

## Insert VALID student (Allowed)

```
INSERT INTO college_students (name, department_id)
VALUES ('Aman', 1);
```

Why allowed?

- ✓ Department with id = 1 exists (CSE)

```
INSERT INTO college_students (name, department_id)
VALUES ('Riya', 2);
```

- ✓ ECE exists → allowed

## Insert INVALID student (Blocked by PostgreSQL)

```
INSERT INTO college_students (name, department_id)
VALUES ('Kunal', 5);
```

## What happens?

 ERROR:

insert or update on table "college\_students"  
violates foreign key constraint

Why?

- 🚫 There is **NO** department with id = 5



“You cannot assign a student to a department that doesn’t exist.”

## ⌚ One-to-Many (Very Important)

**One department → many students**

Example:

Student	department_id
Aman	1
Neha	1
Riya	2

- CSE (1) → Aman, Neha
- ECE (2) → Riya

- ✓ ONE department
- ✓ MANY students

This is called:

ONE → MANY relationship

## 🔒 What Foreign Key REALLY enforces

Rule	Who enforces it
Student must belong to a valid department	PostgreSQL
No fake department IDs allowed	PostgreSQL
Data consistency	PostgreSQL

Without Foreign Key ✗:

- Students can point to random numbers

- Data becomes dirty

With Foreign Key  :

- Clean data
- Real-world logic enforced



## One Line Definition (Exam / Interview)

A **Foreign Key** is a column that **references the Primary Key of another table** to maintain data integrity.



## Mini Test (Check your understanding)

Answer mentally:

**1** Can two students have same department\_id?

YES

**2** Can one student have two departments here?

NO

**3** Can a student belong to non-existing department?

NO



## Why this matters later

You'll use this in:

- SQLAlchemy relationships
- Backend APIs
- Authentication (users → roles)

- Real projects

## Mini Practice

Ask to write:

1. Create a teachers table
2. Insert 3 teachers
3. Fetch teachers older than 30
4. Sort teachers by name
5. Get only top 1 result

# COMPLETE SQL ROADMAP (A → Z)

## LEVEL 1 — EASY (FOUNDATION)

👉 Goal:

- ✓ Be confident in PostgreSQL CLI
- ✓ Read & write basic queries without fear

### **1 Database & CLI Basics**

- psql login
- \l – list databases
- \c database\_name – connect database
- \dt – list tables
- \d table\_name – describe table
- \q – exit

### **2 Data Types**

- INT, SERIAL

- VARCHAR, TEXT
- BOOLEAN
- DATE, TIMESTAMP
- FLOAT, NUMERIC

## 3 Table Operations (DDL)

- CREATE TABLE
- DROP TABLE
- ALTER TABLE
- RENAME TABLE
- ADD COLUMN
- DROP COLUMN

## 4 Basic CRUD Queries

- INSERT
- SELECT
- UPDATE
- DELETE

## 5 Filtering Data

- WHERE
- AND, OR, NOT
- Comparison operators: =, !=, >, <, >=, <=
- IN
- BETWEEN
- LIKE
- ILIKE
- IS NULL

## 6 Sorting & Limiting

- ORDER BY
- ASC, DESC
- LIMIT

- OFFSET



## LEVEL 2 — INTERMEDIATE (REAL BACKEND SQL)

👉 Goal:

- ✓ Think like a backend developer
- ✓ Write meaningful business queries

### 7 Constraints

- PRIMARY KEY
- FOREIGN KEY
- UNIQUE
- NOT NULL
- DEFAULT
- CHECK

### 8 Relationships

- One-to-Many
- REFERENCES
- Foreign key behavior

### 9 Joins (VERY IMPORTANT)

- INNER JOIN
- LEFT JOIN
- RIGHT JOIN
- FULL JOIN
- JOIN ... ON

### 10 Aggregate Functions

- COUNT()
- SUM()

- AVG()
- MIN()
- MAX()

## 1 1 Grouping Data

- GROUP BY
- HAVING

## 1 2 Subqueries

- Subquery in SELECT
- Subquery in WHERE
- IN (SELECT ...)

## 1 3 Indexes

- CREATE INDEX
- DROP INDEX
- Why indexes exist (performance)

# LEVEL 3 — ADVANCED

👉 Goal:

- ✓ Handle complex data
- ✓ Think like DB designer

## 1 4 Transactions

- BEGIN
- COMMIT
- ROLLBACK

- ACID properties

## 1 5 Views

- CREATE VIEW
- DROP VIEW
- Why views are useful

## 1 6 Functions

- Built-in functions
- NOW(), CURRENT\_DATE
- String functions
- Date functions

## 1 7 Case Statements

- CASE WHEN THEN ELSE END

## 1 8 Advanced Joins & Logic

- Self join
- Conditional joins

## 1 9 Performance Basics

- Query planning (conceptual)
- When queries are slow

## 2 0 Real-World Patterns

- Pagination (LIMIT + OFFSET)
- Soft delete
- Unique email constraint
- Auditing (created\_at)



# LEVEL 1 — SQL FOUNDATIONS

## 🎯 Goal after this

You should be able to:

- Sit in psql comfortably
- Create tables without fear
- Insert, read, update, delete data
- Write basic filters confidently
- Understand *what you're doing*, not memorizing

## 1 PostgreSQL CLI BASICS

### Enter PostgreSQL

```
psql postgres
```

You see:

```
postgres=#
```

That means:

- You are inside PostgreSQL
- Connected to database: postgres

### List databases

```
\l
```

## Create database

```
CREATE DATABASE college;
```

## Connect to database

```
\c college
```

Now prompt becomes:

```
college=#
```

If you see this →  success

## List tables (initially empty)

```
\dt
```

## Exit psql

```
\q
```

## ② DATA TYPES (YOU MUST KNOW THESE)

These decide **what kind of data a column can store.**

Data Type	Meaning	Example
INT	Whole number	21
SERIAL	Auto-increment ID	1, 2, 3
VARCHAR(100)	Short text	name
TEXT	Long text	description

BOOLEAN	true / false	is_active
DATE	Date only	2026-01-22
TIMESTAMP	Date + time	created_at

👉 Don't overthink. We'll use **INT**, **SERIAL**, **VARCHAR**, **BOOLEAN** mostly.

## 3 CREATE TABLE (DDL — STRUCTURE)

### Create students table

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

📌 Meaning:

- id → auto number (1,2,3...)
- PRIMARY KEY → unique identity
- Columns define **shape of data**

### Verify table

```
\dt
```

### Describe table

```
\d students
```

## INSERT DATA (CREATE ROWS)

### Insert one row

```
INSERT INTO students (name, age, email)  
VALUES ('Aman', 21, 'aman@gmail.com');
```

### Insert multiple rows

```
INSERT INTO students (name, age, email) VALUES  
('Riya', 20, 'riya@gmail.com'),  
('Kunal', 22, 'kunal@gmail.com'),  
('Neha', 19, 'neha@gmail.com');
```

 Every row = one student

## SELECT DATA (MOST IMPORTANT)

### Get everything

```
SELECT * FROM students;
```

### Select specific columns

```
SELECT name, email FROM students;
```

### Select single row

```
SELECT * FROM students WHERE id = 1;
```

## 6 WHERE CLAUSE (FILTERING DATA)

### Equality

```
SELECT * FROM students WHERE age = 21;
```

### Greater / Less

```
SELECT * FROM students WHERE age > 20;
```

### AND

```
SELECT * FROM students  
WHERE age > 20 AND name = 'Kunal';
```

### OR

```
SELECT * FROM students  
WHERE age = 19 OR age = 20;
```

### NOT

```
SELECT * FROM students WHERE age != 21;
```

### IN

```
SELECT * FROM students WHERE age IN (19, 21);
```

## BETWEEN

```
SELECT * FROM students WHERE age BETWEEN 20 AND 22;
```

## LIKE (pattern search)

```
SELECT * FROM students WHERE name LIKE 'A%';
```

Starts with A

## ILIKE (case-insensitive)

```
SELECT * FROM students WHERE email ILIKE '%gmail%';
```

## NULL check

```
SELECT * FROM students WHERE email IS NULL;
```

## 7 ORDER BY (SORTING)

### Ascending (default)

```
SELECT * FROM students ORDER BY age;
```

### Descending

```
SELECT * FROM students ORDER BY age DESC;
```

## Order by name

```
SELECT * FROM students ORDER BY name;
```

## 8 LIMIT & OFFSET (PAGINATION BASICS)

### First 2 rows

```
SELECT * FROM students LIMIT 2;
```

### Skip first row, get next 2

```
SELECT * FROM students LIMIT 2 OFFSET 1;
```

💡 Used in pagination in backend APIs

## 9 UPDATE DATA

### Update one row

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

⚠️ **ALWAYS use WHERE**

Without WHERE → all rows update ❌

## Verify

```
SELECT * FROM students;
```

## 10 DELETE DATA

### Delete one row

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

### Delete all rows (danger)

```
DELETE FROM students;
```

## 1 1 DROP TABLE (DANGEROUS)

```
DROP TABLE students;
```

Deletes structure + data permanently.

## LEVEL 1 — FINAL CHECKLIST (ALL DONE)

- ✓ PostgreSQL CLI
- ✓ CREATE DATABASE
- ✓ CREATE TABLE
- ✓ INSERT
- ✓ SELECT

- ✓ WHERE
- ✓ AND / OR / NOT
- ✓ IN / BETWEEN
- ✓ LIKE / ILIKE
- ✓ ORDER BY
- ✓ LIMIT / OFFSET
- ✓ UPDATE
- ✓ DELETE



## Level 2: SQL INTERMEDIATE LEVEL



### PART 1 — CONSTRAINTS & DATA INTEGRITY



#### CONCEPT (VERY SIMPLE)

**Constraint = Rule**

Database says:

“I will not allow bad data.”

Just like:

- Exam hall rules
- College ID rules

#### ◊ 1. NOT NULL

##### ? Why?

Some data is **mandatory**.

## Example:

Student **must** have a name.

### Create table

```
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    age INT,
    email VARCHAR(100)
);
```

### Try inserting NULL

```
INSERT INTO students (age, email)
VALUES (20, 'test@gmail.com');
```

✖ Error → GOOD (database saved you)

## ◊ 2. UNIQUE

### ? Why?

Email should not repeat.

## Example

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    email VARCHAR(100) UNIQUE,
    password TEXT
);
```

## Test

```
INSERT INTO users (email, password)
VALUES ('a@gmail.com', '123');
```

```
INSERT INTO users (email, password)
VALUES ('a@gmail.com', '456');
```

✗ Duplicate blocked

## ◊ 3. DEFAULT

### ? Why?

If user doesn't give value → database sets it.

### Example

```
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    is_active BOOLEAN DEFAULT true
);
```

Insert:

```
INSERT INTO students (name)
VALUES ('Riya');
```

is\_active → true automatically

## ◊ 4. CHECK

### ? Why?

Ensure logical values.

#### Example:

Age cannot be negative.

```
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    age INT CHECK (age >= 0)
);
```

Try:

```
INSERT INTO students (name, age)
VALUES ('Test', -5);
```

✗ Rejected

## ◊ 5. PRIMARY KEY (Quick Reminder)

- Unique
- Not NULL
- Identifies row

```
id SERIAL PRIMARY KEY
```

## ◊ 6. FOREIGN KEY (EASY EXPLANATION)

### Story

- One **Department**
- Many **Students**

Student belongs to **one department**.

### Create departments

```
CREATE TABLE departments (
    id SERIAL PRIMARY KEY,
    dept_name VARCHAR(100)
);
```

Insert:

```
INSERT INTO departments (dept_name)
VALUES ('CSE'), ('ECE');
```

### Create students with department\_id

```
CREATE TABLE college_students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    department_id INT REFERENCES departments(id)
);
```

 Meaning:

- department\_id must exist in departments table

- No invalid department allowed

## Try invalid insert

```
INSERT INTO college_students (name, department_id)  
VALUES ('Aman', 99);
```

✗ Blocked → database protecting data

## 📋 MINI CHEAT SHEET — PART 1

Constraint	Purpose
NOT NULL	Value required
UNIQUE	No duplicates
DEFAULT	Auto value
CHECK	Condition
PRIMARY KEY	Row identity
FOREIGN KEY	Table relation



## PART 2 — AGGREGATE FUNCTIONS

This is where SQL stops being “just data storage”  
and starts becoming **analytics + reporting**.



## MENTAL MODEL

Think like this 👇

👉 **Rows = raw data**

👉 **Aggregate functions = summary of data**

Example:

- 100 students →
- “How many students?”
- “Average age?”
- “Youngest student?”

That's **aggregate functions**.

## ◇ WHAT ARE AGGREGATE FUNCTIONS?

Aggregate functions:

- Take **many rows**
- Return **ONE value**

## 🌐 AGGREGATE FUNCTIONS WE WILL COVER

Function	Meaning
COUNT	How many rows
SUM	Total
AVG	Average
MIN	Smallest value
MAX	Largest value

## 🔧 SETUP (USE THIS TABLE)

Make sure you have a table like this 👉

```
CREATE TABLE students (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100),
    age INT,
    department VARCHAR(50)
);
```

Insert data:

```
INSERT INTO students (name, age, department) VALUES
('Aman', 21, 'CSE'),
('Riya', 22, 'CSE'),
('Kunal', 20, 'ECE'),
('Neha', 19, 'ECE'),
('Rahul', 23, 'CSE');
```

## ◊ 1. COUNT()

### ❓ What does COUNT do?

Counts **number of rows**

#### Count all students

```
SELECT COUNT(*) FROM students;
```

✓ Output: total students

#### Count students with age

```
SELECT COUNT(age) FROM students;
```

📌 Ignores NULL values

## Count CSE students

```
SELECT COUNT(*)  
FROM students  
WHERE department = 'CSE';
```

## ◊ 2. SUM()

### ? What does SUM do?

Adds numeric values

## Total age of all students

```
SELECT SUM(age) FROM students;
```

## Total age of CSE students

```
SELECT SUM(age)  
FROM students  
WHERE department = 'CSE';
```

📌 Only works on **numbers**

## ◊ 3. AVG()

### ? What does AVG do?

Average (mean)

## Average age of students

```
SELECT AVG(age) FROM students;
```

❖ Result is **decimal**

## Average age of ECE students

```
SELECT AVG(age)  
FROM students  
WHERE department = 'ECE';
```

## ◊ 4. MIN()

### ? What does MIN do?

Finds smallest value

## Youngest student age

```
SELECT MIN(age) FROM students;
```

## Youngest CSE student

```
SELECT MIN(age)  
FROM students  
WHERE department = 'CSE';
```

## ◊ 5. MAX()

### ? What does MAX do?

Finds largest value

## Oldest student

```
SELECT MAX(age) FROM students;
```

## Oldest ECE student

```
SELECT MAX(age)  
FROM students  
WHERE department = 'ECE';
```



## IMPORTANT RULES

### ! Aggregate functions:

- Return **one row**
- Ignore NULL values (except COUNT\*)



## PRACTICE QUESTIONS (DO THESE)

Try without looking

- 1 Total number of students
- 2 Average age of all students
- 3 Count students in ECE
- 4 Youngest student age
- 5 Oldest student in CSE

(Answer using COUNT, AVG, MIN, MAX)

## MINI CHEAT SHEET — PART 2

Function	Example
COUNT	COUNT(*)
SUM	SUM(age)
AVG	AVG(age)
MIN	MIN(age)
MAX	MAX(age)

## WHY THIS MATTERS

These are used in:

- Dashboards
- Reports
- Analytics
- Admin panels
- Business decisions

## PART 3 — GROUP BY + HAVING

## MENTAL MODEL

Think like this 

- **WHERE** → filters rows
- **GROUP BY** → creates groups
- **AGGREGATE FUNCTIONS** → summarize each group
- **HAVING** → filters groups

❖ HAVING = WHERE for groups

## 🔑 SAME TABLE (IMPORTANT)

We'll use the same students table:

```
SELECT * FROM students;
```

id	name	age	department
1	Aman	21	CSE
2	Riya	22	CSE
3	Kunal	20	ECE
4	Neha	19	ECE
5	Rahul	23	CSE

## ◊ 1. GROUP BY (BASICS)

### ❓ What does GROUP BY do?

It **groups rows** that have the **same value**

**Example: Count students per department**

```
SELECT department, COUNT(*)
FROM students
GROUP BY department;
```

**Output:**

department	count
CSE	3
ECE	2

✓ One row per department

## RULE

If you use GROUP BY:

### 👉 Every selected column must be

- Either in GROUP BY
- Or inside an aggregate function

### ✗ WRONG:

```
SELECT name, COUNT(*)  
FROM students  
GROUP BY department;
```

### ✓ RIGHT:

```
SELECT department, COUNT(*)  
FROM students  
GROUP BY department;
```

## ◊ 2. GROUP BY + AVG()

### Average age per department

```
SELECT department, AVG(age)  
FROM students  
GROUP BY department;
```

## ◊ 3. GROUP BY + MIN / MAX

**Youngest student in each department**

```
SELECT department, MIN(age)
FROM students
GROUP BY department;
```

**Oldest student in each department**

```
SELECT department, MAX(age)
FROM students
GROUP BY department;
```

## ◊ 4. WHERE + GROUP BY (ORDER MATTERS)

**Count students older than 20 per department**

```
SELECT department, COUNT(*)
FROM students
WHERE age > 20
GROUP BY department;
```

📌 WHERE filters rows **before grouping**

## ◊ 5. HAVING (MOST CONFUSING PART)

### ? Why HAVING?

Because WHERE **cannot** use aggregate functions

✗ WRONG:

```
SELECT department, COUNT(*)  
  
FROM students  
WHERE COUNT(*) > 2  
GROUP BY department;
```

✓ RIGHT:

```
SELECT department, COUNT(*)  
FROM students  
GROUP BY department  
HAVING COUNT(*) > 2;
```

**Output:**

department	count
CSE	3

## ◊ 6. WHERE vs HAVING (CLEAR DIFFERENCE)

WHERE	HAVING
Filters rows	Filters groups

Used before GROUP BY	Used after GROUP BY
Cannot use COUNT, AVG	Uses aggregate functions

## ❖ 7. Combined Example (REAL LIFE)

**Departments with average age > 21**

```
SELECT department, AVG(age)
FROM students
GROUP BY department
HAVING AVG(age) > 21;
```



## PRACTICE QUESTIONS (IMPORTANT)

Try these 👇

- 1 Count students per department
- 2 Average age per department
- 3 Departments with more than 2 students
- 4 Departments where average age > 20
- 5 Count students older than 21 per department



## MINI CHEAT SHEET — PART 3

```
SELECT column, AGG_FUNC(column)
FROM table
WHERE condition
GROUP BY column
HAVING AGG_FUNC(column) condition;
```

## COMMON INTERVIEW QUESTION

❓ Difference between WHERE and HAVING?

 Answer:

- WHERE filters rows before grouping
- HAVING filters groups after aggregation

## PART 4 — JOINS

### MENTAL MODEL

Think like this 

- Tables are **separate**
- JOINS are used to **connect tables**
- They connect using a **common column (usually FK → PK)**

 **No JOIN = isolated data**

 **JOIN = meaningful data**

## TABLE SETUP (USE THIS)

### **departments**

```
SELECT * FROM departments;
```

<b>id</b>	<b>dept_name</b>
1	CSE
2	ECE

## students

```
SELECT * FROM students;
```

<b>id</b>	<b>name</b>	<b>age</b>	<b>department_id</b>
1	Aman	21	1
2	Riya	22	1
3	Kunal	20	2
4	Neha	19	2
5	Rahul	23	1

## ◊ 1. INNER JOIN (MOST USED)

### ? What is INNER JOIN?

✓ Returns rows **only when match exists in both tables**

### Example: Student name + Department name

```
SELECT students.name, departments.dept_name  
FROM students  
INNER JOIN departments  
ON students.department_id = departments.id;
```

### Output:

<b>name</b>	<b>dept_name</b>
Aman	CSE
Riya	CSE
Rahul	CSE
Kunal	ECE
Neha	ECE



```
students n departments
```

Only common records.

## ◊ 2. LEFT JOIN

### ? What is LEFT JOIN?

- ✓ All rows from **left table**
- ✓ Matching rows from right table
- ✓ Non-matching → NULL

### Example: All students even if no department

```
SELECT students.name, departments.dept_name  
FROM students  
LEFT JOIN departments  
ON students.department_id = departments.id;
```

If a student has no department → dept\_name = NULL

## ◊ 3. RIGHT JOIN

### ? What is RIGHT JOIN?

- ✓ All rows from **right table**
- ✓ Matching rows from left table

```
SELECT students.name, departments.dept_name  
FROM students  
RIGHT JOIN departments  
ON students.department_id = departments.id;
```

✖ Rarely used (LEFT JOIN preferred)

## ◊ 4. FULL OUTER JOIN

### ? What is FULL JOIN?

- ✓ All rows from both tables
- ✓ Matching + non-matching

```
SELECT students.name, departments.dept_name  
FROM students  
FULL OUTER JOIN departments  
ON students.department_id = departments.id;
```

NULL appears where no match exists.

## ◊ 5. JOIN + WHERE

### Students from CSE only

```
SELECT students.name  
FROM students  
JOIN departments  
ON students.department_id = departments.id  
WHERE departments.dept_name = 'CSE';
```

## ◊ 6. JOIN + GROUP BY (REAL LIFE)

Count students per department

```
SELECT departments.dept_name, COUNT(students.id)
FROM departments
LEFT JOIN students
ON students.department_id = departments.id
GROUP BY departments.dept_name;
```

✓ Even departments with **zero students appear**

## ◊ 7. ALIAS (MAKE QUERIES CLEAN)

```
SELECT s.name, d.dept_name
FROM students s
JOIN departments d
ON s.department_id = d.id;
```

📌 Always use aliases in real projects

## 📝 PRACTICE QUESTIONS

- 1 List student names with department names
- 2 Find students in ECE department
- 3 Count students per department
- 4 Show departments even if no students
- 5 Find students without a department

## MINI CHEAT SHEET — PART 4

```
-- INNER JOIN
```

```
FROM A
```

```
JOIN B ON A.col = B.col
```

```
-- LEFT JOIN
```

```
FROM A
```

```
LEFT JOIN B ON A.col = B.col
```

```
-- RIGHT JOIN
```

```
FROM A
```

```
RIGHT JOIN B ON A.col = B.col
```

```
-- FULL JOIN
```

```
FROM A
```

```
FULL JOIN B ON A.col = B.col
```



## COMMON INTERVIEW QUESTION

 Difference between INNER JOIN and LEFT JOIN?

 Answer:

- INNER → only matching rows
- LEFT → all left + matching right



## PART 5 — SUBQUERIES & NESTED QUERIES

## FIRST: WHAT IS A SUBQUERY?

Simple definition:

👉 A query inside another query

```
SELECT *  
FROM students  
WHERE age > (  
    SELECT AVG(age) FROM students  
)
```

📌 Inner query runs **first**

📌 Outer query uses its result

## REAL-LIFE ANALOGY

“Show students older than the **average age**”

You first calculate average → then compare

### ◊ 1. SUBQUERY IN WHERE

**Example: Students older than average age**

```
SELECT name, age  
FROM students  
WHERE age > (  
    SELECT AVG(age)  
    FROM students  
)
```

### Example: Students from CSE department

```
SELECT name
FROM students
WHERE department_id = (
    SELECT id
    FROM departments
    WHERE dept_name = 'CSE'
);
```

- ✓ Very common
- ✓ Clean logic

## ◊ 2. SUBQUERY WITH IN

### ? When to use IN?

- ✓ When subquery returns **multiple rows**

### Example: Students in CSE or ECE

```
SELECT name
FROM students
WHERE department_id IN (
    SELECT id
    FROM departments
);
```

### Example: Students older than any 22-year-old

```
SELECT name, age
FROM students
WHERE age > ANY (
    SELECT age
```

```
FROM students  
WHERE age = 22  
);
```

## ◊ 3. SUBQUERY IN SELECT (CALCULATED COLUMN)

**Example:** Show average age with every student

```
SELECT name, age,  
       (SELECT AVG(age) FROM students) AS avg_age  
FROM students;
```

- 📌 Subquery runs once
- 📌 Result reused

## ◊ 4. SUBQUERY IN FROM (DERIVED TABLE)

**Example:** Find students older than 20

```
SELECT *  
FROM (  
    SELECT name, age  
    FROM students  
    WHERE age > 20  
) AS temp;
```

- 📌 Subquery behaves like a **temporary table**

## ◊ 5. EXISTS (VERY IMPORTANT)

### ? What is EXISTS?

- ✓ Checks **presence**, not value
- ✓ Faster than IN in large tables

### Example: Students with valid department

```
SELECT name
FROM students s
WHERE EXISTS (
    SELECT 1
    FROM departments d
    WHERE d.id = s.department_id
);
```

### Example: Departments having students

```
SELECT dept_name
FROM departments d
WHERE EXISTS (
    SELECT 1
    FROM students s
    WHERE s.department_id = d.id
);
```

## ◊ 6. NOT EXISTS

### Departments with NO students

```
SELECT dept_name
FROM departments d
WHERE NOT EXISTS (
    SELECT 1
    FROM students s
    WHERE s.department_id = d.id
);
```

## ◊ 7. CORRELATED SUBQUERY (ADVANCED INTERMEDIATE)

### ? What is correlated?

- ✓ Inner query depends on outer query
- ✓ Runs **for each row**

### Example: Students older than department average

```
SELECT name, age
FROM students s
WHERE age > (
    SELECT AVG(age)
    FROM students
    WHERE department_id = s.department_id
);
```

- ➔ Powerful
- ➔ Slower if misused

## PRACTICE QUESTIONS

- 1 Find students older than average
- 2 Departments having more than 2 students
- 3 Students belonging to CSE using subquery
- 4 Students without department
- 5 Students older than their department average

## MINI CHEAT SHEET — PART 5

-- Single value

WHERE col = (SELECT ...)

-- Multiple values

WHERE col IN (SELECT ...)

-- Check existence

WHERE EXISTS (SELECT 1 ...)

-- Derived table

FROM (SELECT ...) AS alias

-- Correlated

WHERE col > (SELECT ... WHERE t.col = outer.col)

## VERY IMPORTANT INTERVIEW TIP

 IN vs EXISTS?

 Answer:

- IN → compares values
- EXISTS → checks row existence
- EXISTS is faster for large datasets

## PART 6 — INDEXES & PERFORMANCE BASICS

Goal:

Understand **why queries become slow** and **how indexes fix them**

### FIRST: WHY DATABASES BECOME SLOW

Imagine this table:

students (1 million rows)

Query:

```
SELECT * FROM students WHERE email = 'abc@gmail.com';
```

Without index 

 PostgreSQL checks **every row** (full table scan)

With index 

 PostgreSQL jumps directly to the row

## ◊ 1. WHAT IS AN INDEX?

**Simple definition:**

👉 An index is like a book's index

- Table = Book
- Rows = Pages
- Index = Alphabetical index

📌 Faster reads

📌 Slightly slower writes

## ◊ 2. CREATE INDEX

**Index on email**

```
CREATE INDEX idx_students_email  
ON students(email);
```

Now this becomes fast:

```
SELECT * FROM students WHERE email = 'aman@gmail.com';
```

## ◊ 3. UNIQUE INDEX

**Prevent duplicate values**

```
CREATE UNIQUE INDEX idx_students_email_unique  
ON students(email);
```

✗ Duplicate emails now fail

## ◊ 4. WHEN TO USE INDEX (VERY IMPORTANT)

### Good candidates:

- ✓ WHERE
- ✓ JOIN
- ✓ ORDER BY
- ✓ GROUP BY

### Bad candidates:

- ✗ Small tables  
✗ Columns with few unique values (gender, status)

## ◊ 5. MULTI-COLUMN INDEX

### Query example:

```
SELECT *  
FROM students  
WHERE department_id = 1 AND age = 21;
```

### Index:

```
CREATE INDEX idx_students_dept_age  
ON students(department_id, age);
```

📌 Order matters!

## ◊ 6. INDEX ORDER MATTERS

Index:

(department\_id, age)

✓ Works:

```
WHERE department_id = 1  
WHERE department_id = 1 AND age = 21
```

✗ Does NOT work:

WHERE age = 21

## ◊ 7. DROP INDEX

```
DROP INDEX idx_students_email;
```

## ◊ 8. CHECK QUERY PERFORMANCE (INTRO)

**Explain query plan**

```
EXPLAIN SELECT * FROM students WHERE email = 'aman@gmail.com';
```

With index:

Index Scan using idx\_students\_email

Without index:

Seq Scan on students

- ✖ Seq Scan = Slow for big tables
- ✓ Index Scan = Fast

## ◊ 9. EXPLAIN ANALYZE (REAL EXECUTION)

EXPLAIN ANALYZE

```
SELECT * FROM students WHERE email = 'aman@gmail.com';
```

Shows:

- ✓ Time taken
- ✓ Rows scanned
- ✓ Index used or not

## ◊ 10. FOREIGN KEY & INDEX (IMPORTANT)

Foreign keys should be indexed!

```
CREATE INDEX idx_students_dept  
ON students(department_id);
```

Why?

- JOINs become fast
- Deletes/updates safer

## PRACTICE TASKS

- 1** Create index on email
- 2** Create composite index on (department\_id, age)
- 3** Run EXPLAIN before & after index
- 4** Drop index and observe plan
- 5** Add index to foreign key column



## MINI CHEAT SHEET — PART 6

```
CREATE INDEX idx_name ON table(col);
CREATE UNIQUE INDEX idx_name ON table(col);
CREATE INDEX idx_name ON table(col1, col2);

DROP INDEX idx_name;

EXPLAIN SELECT ...
EXPLAIN ANALYZE SELECT ...
```

## INTERVIEW GOLD ANSWER

 Why not index everything?

 Answer:

Indexes speed up reads but slow down inserts/updates and consume memory.

## ⌚ WHERE YOU STAND NOW

You can now:

- ✓ Write optimized SQL
- ✓ Understand slow queries
- ✓ Read execution plans
- ✓ Design better schemas

## ➊ PART 7 — TRANSACTIONS (ACID IN ACTION)

### ➊ What is a Transaction?

A **transaction** is a group of SQL operations that must all succeed together.

- 👉 Either **everything happens**
- 👉 Or **nothing happens**

There is **NO half-done state**.

### ➋ Real-World Example (Bank 📈)

You transfer ₹1000 from **Account A** → **Account B**

**Steps involved:**

1. Subtract ₹1000 from A
2. Add ₹1000 to B

❓ What if step 1 succeeds but step 2 fails?

👉 Money disappears 💰

💡 Transactions prevent this.

## 3 ACID Properties (Very Important)

### ◊ A — Atomicity

All or nothing

If one step fails → everything rolls back.

### ◊ C — Consistency

Database rules are never broken

Balances can't go negative (if rule exists).

### ◊ I — Isolation

Multiple users don't disturb each other

Two people transferring money at the same time won't mess balances.

### ◊ D — Durability

Once committed, data is permanent

Even if the server crashes → data stays.

## 4 SQL Commands for Transactions

### ◊ BEGIN

Start a transaction

### ◊ COMMIT

Save everything permanently

### ◊ ROLLBACK

Undo everything

## 5 Hands-On Practice (DO THIS IN psql)

### Step 1: Create table

```
CREATE TABLE accounts (
    id SERIAL PRIMARY KEY,
    name VARCHAR(50),
    balance INT
);
```

### Step 2: Insert data

```
INSERT INTO accounts (name, balance)
VALUES ('Aman', 5000), ('Riya', 3000);
```

Check:

```
SELECT * FROM accounts;
```

## 6 Transaction Example (Correct Way)

Transfer ₹1000 from Aman → Riya

```
BEGIN;  
  
UPDATE accounts  
SET balance = balance - 1000  
WHERE name = 'Aman';  
  
UPDATE accounts  
SET balance = balance + 1000  
WHERE name = 'Riya';  
  
COMMIT;
```

- Both updates succeed
- Money is safe

## 7 What If Something Goes Wrong?

Example: Mistake happens

```
BEGIN;  
  
UPDATE accounts  
SET balance = balance - 1000  
WHERE name = 'Aman';  
  
-- Oops! Wrong table / error  
UPDATE accountsss  
SET balance = balance + 1000  
WHERE name = 'Riya';
```

```
ROLLBACK;
```

### Result:

- ✓ Aman's balance restored
- ✓ No money lost

## 8 WITHOUT Transactions (DANGEROUS ✗)

```
UPDATE accounts  
SET balance = balance - 1000  
WHERE name = 'Aman';  
  
-- crash happens here 🤦
```

- ✗ Aman lost money
- ✗ Riya didn't receive
- ✗ Database corrupted logically

## 9 Where Transactions Are USED (REAL LIFE)

- ✓ Banking systems
- ✓ Payments (UPI, cards)
- ✓ E-commerce orders
- ✓ Ticket booking
- ✓ Wallet systems
- ✓ Stock trading

## MINI CHEAT SHEET (SAVE THIS)

```
BEGIN; -- start transaction  
COMMIT; -- save changes  
ROLLBACK; -- undo changes
```

Rule:

**Multiple dependent queries → ALWAYS use transactions**



## PART 8 — VIEWS (VIRTUAL TABLES)

### **1** What is a VIEW?

A **VIEW** is a **saved SELECT query**.



Think of it as:

A **window** to see data  
NOT a real table  
NO data stored separately

### Simple analogy

- **Table** → Actual data stored
- **View** → Filtered / formatted way to look at that data

### **2** Why Do We Need Views?

#### Without view

You write long queries again and again:

```
SELECT name, email  
FROM students  
WHERE age > 18  
ORDER BY name;
```

### With view

Write once → reuse forever

## 3 Create a Sample Table

```
CREATE TABLE students (  
    id SERIAL PRIMARY KEY,  
    name VARCHAR(50),  
    age INT,  
    email VARCHAR(100)  
);
```

Insert data:

```
INSERT INTO students (name, age, email)  
VALUES  
    ('Aman', 21, 'aman@gmail.com'),  
    ('Riya', 19, 'riya@gmail.com'),  
    ('Neha', 17, 'neha@gmail.com');
```

## 4 CREATE VIEW (Core Concept)

### Create a view of adult students

```
CREATE VIEW adult_students AS  
SELECT name, age, email
```

```
FROM students  
WHERE age >= 18;
```

- View created
- No new table created
- No data duplicated

## 5 Use the View (Like a Table)

```
SELECT * FROM adult_students;
```

Output:

name	age	email
Aman	21	aman@gmail.com
Riya	19	riya@gmail.com

- 👉 Looks like a table
- 👉 Acts like a table
- 👉 But is NOT a table

## 6 Update Data Through View (IMPORTANT)

Can we INSERT/UPDATE using views?

- YES — if view is SIMPLE

This works:

```
UPDATE adult_students  
SET age = 20  
WHERE name = 'Riya';
```

Because:

- ✓ Single table
- ✓ No joins
- ✓ No group by

## ✗ NOT allowed in complex views

Example (not updatable):

```
CREATE VIEW student_count AS  
SELECT age, COUNT(*)  
FROM students  
GROUP BY age;
```

Why ✗?

- Aggregation
- Grouping
- PostgreSQL doesn't know which row to update

## 7 Why Companies Use Views (REAL WORLD)

### ◊ Security

Hide sensitive columns

```
CREATE VIEW public_students AS  
SELECT name, age  
FROM students;
```

Users can't see emails/passwords.

### ❖ Simplicity 🎯

Frontend developers use views instead of writing joins.

### ❖ Consistency ✏️

Everyone uses the same logic → no mistakes.

## 8 Read-Only vs Updatable Views

Type	Description
Read-only	Uses joins, group by, aggregate
Updatable	Single table, simple SELECT

## 9 DROP VIEW

```
DROP VIEW adult_students;
```

## ⚡ MINI CHEAT SHEET

```
CREATE VIEW view_name AS SELECT ...
SELECT * FROM view_name
DROP VIEW view_name
```

Rule:

Views store **queries**, not **data**

# PART 9 — CONSTRAINTS IN REAL SYSTEMS

We'll focus on:

- ✓ ON DELETE CASCADE
- ✓ ON UPDATE CASCADE
- ✓ RESTRICT
- ✓ Why companies care

## **1 First: What is a Constraint? (Simple)**

A **constraint** is a **rule** that protects your data.

 Think:

“Database police 

## **2 Base Tables (We'll Reuse This Everywhere)**

### **Departments table (Parent)**

```
CREATE TABLE departments (
    id SERIAL PRIMARY KEY,
    dept_name VARCHAR(50)
);
```

Insert:

```
INSERT INTO departments (dept_name)
VALUES ('CSE'), ('ECE');
```

## Students table (Child)

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

Insert students:

```
INSERT INTO students (name, department_id)
VALUES
('Aman', 1),
('Riya', 1),
('Neha', 2);
```

## 3 Problem Without Rules (WHY constraints exist)

If we delete a department:

```
DELETE FROM departments WHERE id = 1;
```

✗ What about students of CSE?

- They still exist
- But their department doesn't
- **Broken data**

This is called:

👉 **Orphan records**

## 4 ON DELETE CASCADE

### Meaning:

“If parent is deleted → delete children automatically”

### Create tables with CASCADE

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

### What happens now?

```
DELETE FROM departments WHERE id = 1;
```

- ✓ Department CSE deleted
  - ✓ All CSE students deleted automatically
-  PostgreSQL does the cleanup for you

## 5 ON UPDATE CASCADE

### Meaning:

“If parent key changes → update child automatically”

Example (rare but important):

```
REFERENCES departments(id)
ON UPDATE CASCADE
```

If department ID changes:

```
UPDATE departments SET id = 10 WHERE id = 2;
```

- ✓ Students' department\_id updates automatically

## 6 RESTRICT (Strict Mode ✖)

**Meaning:**

“Don’t allow delete/update if children exist”

**Example:**

```
REFERENCES departments(id)
ON DELETE RESTRICT
```

Now:

```
DELETE FROM departments WHERE id = 2;
```

✖ ERROR:

cannot delete department — students exist

- ✓ Forces admin to handle students first

## 7 Comparison Table (VERY IMPORTANT)

Rule	What it does
CASCADE	Auto delete/update child
RESTRICT	Block delete/update
NO ACTION	Similar to RESTRICT
SET NULL	Set FK to NULL

## 8 Real-World Usage 🏢

### Banking

- Delete account → delete transactions? ❌
- Use **RESTRICT**

### E-commerce

- Delete order → delete order\_items ✅
- Use **CASCADE**

### College System

- Delete department → delete students? ❌
- Usually **RESTRICT**

## 9 When to Use What?

Situation	Best choice
Temporary child data	CASCADE

Critical historical data	RESTRICT
Optional relation	SET NULL

## MINI CHEAT SHEET

REFERENCES table(id)

ON DELETE CASCADE

ON UPDATE CASCADE

ON DELETE RESTRICT

## PART 10 — CASE & CONDITIONAL LOGIC (SQL IF-ELSE)

Think of **CASE** as:

SQL's **decision-making tool**

Just like:

```
if condition:  
    do this  
else:  
    do that
```

### Why CASE Exists (Intuition First)

SQL can:

- Store data

- Filter data

But SQL **cannot think** unless we use **CASE**.

CASE allows SQL to:

- Categorize data
- Assign labels
- Apply business rules
- Make decisions per row

## 2 Basic CASE Syntax (Must Memorize)

```
CASE
WHEN condition THEN result
WHEN condition THEN result
ELSE result
END
```

- ✓ Evaluated **row by row**
- ✓ Returns **one value**

## 3 Sample Table (Mental Model)

students

<b>id</b>	<b>name</b>	<b>age</b>	<b>marks</b>
1	Aman	21	85
2	Riya	17	92
3	Neha	19	55

## 4 CASE in SELECT (Most Common)

### Example: Adult or Minor

```
SELECT name, age,  
CASE  
    WHEN age >= 18 THEN 'Adult'  
    ELSE 'Minor'  
END AS status  
FROM students;
```

### Output:

name	age	status
Aman	21	Adult
Riya	17	Minor
Neha	19	Adult

- ✓ No table modification
- ✓ Dynamic column

## 5 Multiple Conditions (Grades Example 🎓)

```
SELECT name, marks,  
CASE  
    WHEN marks >= 90 THEN 'A'  
    WHEN marks >= 75 THEN 'B'  
    WHEN marks >= 60 THEN 'C'  
    ELSE 'Fail'  
END AS grade  
FROM students;
```

### Important rule

SQL checks **top to bottom**, first match wins.

## 6 CASE in UPDATE (Real Systems Use This 💧)

### Scenario:

Give scholarship based on marks.

```
ALTER TABLE students ADD COLUMN scholarship VARCHAR(10);
```

```
UPDATE students
SET scholarship =
CASE
    WHEN marks >= 80 THEN 'YES'
    ELSE 'NO'
END;
```

- ✓ Updates all rows in one query
- ✓ Used heavily in production systems

## 7 CASE in ORDER BY (Very Powerful)

### Example: Adults first, minors later

```
SELECT name, age
FROM students
ORDER BY
CASE
    WHEN age >= 18 THEN 1
    ELSE 2
END;
```

- ✓ Custom sorting logic
- ✓ Frequently asked in interviews

## 8 CASE in WHERE (Rare but Valid)

```
SELECT *
FROM students
WHERE
CASE
    WHEN marks >= 60 THEN true
    ELSE false
END;
```

⚠ Usually better to use normal WHERE  
CASE shines most in **SELECT & UPDATE**

## 9 Real-World Applications 🌎

Area	Use
Reports	Pass/Fail, Grades
Dashboards	Status labels
APIs	Flags & categories
Analytics	Bucketing data
Finance	Risk levels

## 🔟 MINI CHEAT SHEET (Save This)

```
CASE
WHEN condition THEN value
```

```
ELSE value  
END
```

Can be used in:

- ✓ SELECT
- ✓ UPDATE
- ✓ ORDER BY
- ⚠ WHERE



## LEVEL 3 — ADVANCED SQL



### LEVEL 3 ROADMAP (What we'll cover)



#### PART 1 — Subqueries (Nested Queries)

- Subquery in WHERE
- Subquery in SELECT
- Subquery in FROM
- Correlated subqueries



#### PART 2 — Common Table Expressions (CTEs)

- WITH clause
- Why CTEs > subqueries
- Multiple CTEs
- Recursive CTE (intro)



#### PART 3 — Window Functions (MOST IMPORTANT 💧 )

- OVER()
- ROW\_NUMBER
- RANK, DENSE\_RANK
- SUM() OVER

- PARTITION BY
- ORDER BY in windows

## PART 4 — Indexes & Performance

- What indexes really are
- When indexes help
- When indexes hurt
- EXPLAIN

## PART 5 — Advanced Joins & Data Analysis

- Self join
- Anti join
- Semi join
- Finding duplicates
- Gaps & islands problems

## PART 6 — Real Interview & Production Patterns

- Nth highest salary
- Top-N per group
- Running totals
- Time-based queries

## VERY IMPORTANT NOTE

Level 3 concepts:

- Are **heavily tested in interviews**
- Are used **daily by backend engineers**
- Separate juniors from strong developers



# PART 1 — SUBQUERIES (START HERE)

## 1 What is a Subquery? (Easy words)

A **subquery** is:

A query inside another query

Think like:

“First find something → then use it”

## 2 Basic Subquery Example

**Question:**

Find students who scored **more than average marks**

```
SELECT *  
FROM students  
WHERE marks > (  
    SELECT AVG(marks)  
    FROM students  
)
```

📌 Execution order:

1. Inner query runs first → average marks
2. Outer query uses that value

## 3 Subquery in SELECT

```
SELECT name,  
       (SELECT AVG(marks) FROM students) AS avg_marks  
  FROM students;
```

- ✓ Same value shown for every row
- ✓ Useful for reports

## 4 Subquery in FROM (Derived Table)

```
SELECT *  
FROM (  
        SELECT name, marks  
          FROM students  
         WHERE marks >= 60  
) AS passed_students;
```

📌 Inner query behaves like a **temporary table**

## 5 Correlated Subquery (Important 💧 )

Runs **once per row**

**Example:**

Students who scored more than **their department average**

```
SELECT s1.*  
  FROM students s1  
 WHERE marks > (  
           SELECT AVG(s2.marks)
```

```
FROM students s2  
WHERE s2.department_id = s1.department_id  
);
```

- 📌 Used in analytics & business logic
- 📌 Slower than joins (important later)

## PART 1 MINI CHEAT SHEET

```
-- WHERE  
WHERE column > (SELECT ...)  
  
-- SELECT  
SELECT col, (SELECT ...)  
  
-- FROM  
FROM (SELECT ...) alias
```

## PART 2 — CTEs (Common Table Expressions)

### 1 What is a CTE? (Very simple words)

A **CTE** is:

A named temporary result that exists **only for one query**

Think of it as:

“Create a readable temporary table → use it below”

- 📌 Syntax keyword: WITH

## 2 Why CTEs exist (Problem they solve)

### ✗ Without CTE (ugly subquery)

```
SELECT *
FROM (
    SELECT department_id, AVG(marks) AS avg_marks
    FROM students
    GROUP BY department_id
) t
WHERE avg_marks > 70;
```

Hard to read ✗

### ✓ With CTE (clean & readable)

```
WITH dept_avg AS (
    SELECT department_id, AVG(marks) AS avg_marks
    FROM students
    GROUP BY department_id
)
SELECT *
FROM dept_avg
WHERE avg_marks > 70;
```

- ✓ Clear
- ✓ Reusable
- ✓ Interview-friendly

## 3 CTE vs Subquery (Important)

Feature	Subquery	CTE
---------	----------	-----

Readability	<span style="color:red">✗</span> Hard	<span style="color:green">✓</span> Clean
Reusability	<span style="color:red">✗</span> No	<span style="color:green">✓</span> Yes
Debugging	<span style="color:red">✗</span> Pain	<span style="color:green">✓</span> Easy
Complex logic	<span style="color:red">✗</span> Messy	<span style="color:green">✓</span> Best

📌 **Rule:**

If query is more than 1 subquery → use CTE

## ↳ Multiple CTEs in one query

```
WITH dept_avg AS (
    SELECT department_id, AVG(marks) AS avg_marks
    FROM students
    GROUP BY department_id
),
top_students AS (
    SELECT *
    FROM students
    WHERE marks > 80
)
SELECT t.name, d.avg_marks
FROM top_students t
JOIN dept_avg d
ON t.department_id = d.department_id;
```

- 📌 Each CTE builds on logic
- 📌 Used heavily in analytics

## 5 CTE with INSERT / UPDATE / DELETE

### Insert using CTE

```
WITH new_students AS (
    SELECT 'Rahul' AS name, 20 AS age
)
INSERT INTO students (name, age)
SELECT name, age FROM new_students;
```

### Update using CTE

```
WITH low_scorers AS (
    SELECT id FROM students WHERE marks < 40
)
UPDATE students
SET status = 'FAIL'
WHERE id IN (SELECT id FROM low_scorers);
```

## 6 Recursive CTE (Intro — Powerful 💦)

Used for:

- Trees
- Hierarchies
- Org charts
- Folder structures

### Example: Employee → Manager hierarchy

```
WITH RECURSIVE emp_tree AS (
    SELECT id, name, manager_id
    FROM employees
    WHERE manager_id IS NULL
```

```
UNION ALL

SELECT e.id, e.name, e.manager_id
FROM employees e
JOIN emp_tree t
ON e.manager_id = t.id
)
SELECT * FROM emp_tree;
```

 Asked in **senior interviews**

## PART 2 MINI CHEAT SHEET

```
WITH cte_name AS (
    SELECT ...
)
SELECT * FROM cte_name;
```

```
WITH cte1 AS (...),
      cte2 AS (...)

SELECT ...
```

```
WITH RECURSIVE cte AS (...)
```

# PART 3 — WINDOW FUNCTIONS (Deep + Practical)

## 1 What is a Window Function? (Very simple)

A **window function**:

Performs calculations **across related rows**  
**WITHOUT collapsing rows**

📌 Difference from GROUP BY:

- GROUP BY → reduces rows ❌
- Window functions → keep all rows ✓

## 2 Basic Syntax (MUST MEMORIZE)

```
function_name(...) OVER (  
    PARTITION BY column  
    ORDER BY column  
)
```

📌 OVER() = this is a window function

## 3 Why window functions exist (Real pain)

### X GROUP BY problem

```
SELECT department_id, AVG(salary)  
FROM employees  
GROUP BY department_id;
```

You lose individual employee rows ✗

### ✓ Window function solution

```
SELECT  
    name,  
    department_id,  
    salary,  
    AVG(salary) OVER (PARTITION BY department_id) AS dept_avg  
FROM employees;
```

- ✓ Every employee stays
- ✓ Department average shown per row

## 4 MOST IMPORTANT WINDOW FUNCTIONS (Core set)

### 1. ROW\_NUMBER()

Gives unique number per row

```
SELECT  
    name,  
    department_id,  
    ROW_NUMBER() OVER (
```

```
PARTITION BY department_id  
ORDER BY salary DESC  
) AS rn  
FROM employees;
```

📌 Use case:

- Top N per group
- Pagination

## 2. RANK()

Handles **ties**, skips numbers

```
SELECT  
    name,  
    salary,  
    RANK() OVER (ORDER BY salary DESC) AS rank  
FROM employees;
```

Salaries: 100, 100, 90

Ranks: 1, 1, 3

## 3. DENSE\_RANK()

Handles ties **without skipping**

```
SELECT  
    name,  
    salary,  
    DENSE_RANK() OVER (ORDER BY salary DESC) AS rank  
FROM employees;
```

Ranks: 1, 1, 2

## 4. NTILE(n)

Divides rows into **n equal buckets**

```
SELECT
    name,
    salary,
    NTILE(4) OVER (ORDER BY salary DESC) AS quartile
FROM employees;
```

📌 Used in:

- Percentiles
- Grading systems

## 5 Aggregate Window Functions

Yes, **aggregates + OVER()** 🔥

```
SELECT
    name,
    salary,
    SUM(salary) OVER () AS total_salary,
    AVG(salary) OVER () AS avg_salary
FROM employees;
```

✓ Total shown on every row

✓ No GROUP BY needed

## 6 ORDER BY inside window (Running totals)

```
SELECT
    name,
    salary,
    SUM(salary) OVER (
        ORDER BY salary
    ) AS running_total
FROM employees;
```

📌 This is **cumulative sum**

## 7 PARTITION + ORDER together (Very common)

```
SELECT
    name,
    department_id,
    salary,
    SUM(salary) OVER (
        PARTITION BY department_id
        ORDER BY salary
    ) AS dept_running_total
FROM employees;
```

📌 Used in finance, reports, dashboards

## 8 LAG() and LEAD() (Time-based logic)

### LAG() → previous row

```
SELECT
    date,
    revenue,
    revenue - LAG(revenue) OVER (ORDER BY date) AS growth
FROM sales;
```

📍 Month-over-month growth

### LEAD() → next row

```
SELECT
    date,
    revenue,
    LEAD(revenue) OVER (ORDER BY date) AS next_month
FROM sales;
```

## 9 REAL INTERVIEW QUESTION 💧

### ? Get top 2 salaries per department

```
SELECT *
FROM (
    SELECT
        name,
        department_id,
        salary,
        DENSE_RANK() OVER (
            PARTITION BY department_id
            ORDER BY salary DESC
)
```

```
) AS rnk  
FROM employees  
) t  
WHERE rnk <= 2;
```

 This is asked everywhere.

## PART 3 MINI CHEAT SHEET

```
ROW_NUMBER() OVER ()
```

```
RANK() OVER ()
```

```
DENSE_RANK() OVER ()
```

```
NTILE(n) OVER ()
```

```
SUM() OVER ()
```

```
AVG() OVER ()
```

```
LAG(col) OVER (ORDER BY col)
```

```
LEAD(col) OVER (ORDER BY col)
```

## LEVEL 3 — PART 4

### Window Functions (Most Important for Interviews & Real Systems)

This is a **game-changer topic**.

Many people know SQL but **fail here**.

## 1 What is a Window Function? (Easy words)

- 👉 A **window function** lets you perform calculations **across rows**
- 👉 **WITHOUT** grouping them **into one row**

### Difference from GROUP BY

GROUP BY	Window Function
Collapses rows	Keeps all rows
One row per group	One row per record
Loses detail	Keeps detail

## 2 Simple Table Example

Employees

id	name	department	salary
1	Aman	CSE	50000
2	Riya	CSE	60000
3	Neha	ECE	55000
4	Kunal	CSE	45000
5	Arjun	ECE	65000

## 3 ROW\_NUMBER()

- 👉 Assigns **unique number** to each row

### Example

```
SELECT  
    name,
```

```
department,  
salary,  
ROW_NUMBER() OVER () AS row_num  
FROM employees;
```

📌 Output:

Aman → 1

Riya → 2

Neha → 3

## Row number per department

```
SELECT  
name,  
department,  
salary,  
ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rank  
FROM employees;
```

📌 Meaning:

- Restart numbering **for each department**
- Highest salary = rank 1

## ◀ RANK() vs DENSE\_RANK()

### Salary ranking (ties allowed)

```
SELECT  
name,  
department,  
salary,
```

```
RANK() OVER (PARTITION BY department ORDER BY salary DESC) AS rank  
FROM employees;
```

## Difference

Function	Behavior
RANK	Skips numbers
DENSE_RANK	No gaps

## Example salaries

60000 → Rank 1  
60000 → Rank 1  
50000 → Rank 3 (RANK)  
50000 → Rank 2 (DENSE\_RANK)

## 5 NTILE() — Divide into Buckets

👉 Used for grading, percentiles

```
SELECT  
    name,  
    salary,  
    NTILE(4) OVER (ORDER BY salary DESC) AS quartile  
FROM employees;
```

📌 Divides employees into **4 equal salary groups**

## 6 Aggregate as Window Function

Average salary per department (without GROUP BY)

```
SELECT
    name,
    department,
    salary,
    AVG(salary) OVER (PARTITION BY department) AS dept_avg
FROM employees;
```

- ✓ Every row still visible
- ✓ Aggregate value attached

## 7 Running Total (Cumulative Sum)

```
SELECT
    name,
    salary,
    SUM(salary) OVER (ORDER BY salary) AS running_total
FROM employees;
```

📌 Used in:

- Bank balances
- Sales dashboards
- Financial reports

## 8 LEAD() & LAG() (Previous / Next Row)

Compare with previous salary

```
SELECT
    name,
    salary,
    salary - LAG(salary) OVER (ORDER BY salary) AS diff
FROM employees;
```

📌 Used in:

- Growth analysis
- Month-on-month reports
- Trend detection

## 9 Real Interview Question 💧

Find highest paid employee in each department

```
SELECT *
FROM (
    SELECT
        name,
        department,
        salary,
        ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS rn
    FROM employees
) t
WHERE rn = 1;
```

- ✓ VERY common interview question
- ✓ Uses **window + subquery**

## 10 Mini Cheat Sheet 🍳

ROW\_NUMBER() OVER ()  
RANK() OVER ()  
DENSE\_RANK() OVER ()  
NTILE(n) OVER ()

AVG() OVER ()  
SUM() OVER ()  
COUNT() OVER ()

LAG(column) OVER ()  
LEAD(column) OVER ()

PARTITION BY column  
ORDER BY column

## ✳️ When to Use Window Functions

- ✓ Rankings
- ✓ Running totals
- ✓ Percentiles
- ✓ Analytics
- ✓ Reports
- ✓ Interview problems

# PART 5 — Advanced Joins & Data Analysis

(This is where SQL starts feeling like real engineering)

## 1 Self Join

### What is it?

A table joined with itself.

Used when rows in the same table are related to other rows.

### Real-life intuition

Employees table:

emp_id	name	manager_id
1	CEO	NULL
2	Alice	1
3	Bob	2

You want:

 Employee name + Manager name

### ◊ SQL (Self Join)

```
SELECT e.name AS employee,  
       m.name AS manager  
  FROM employees e
```

```
LEFT JOIN employees m  
ON e.manager_id = m.emp_id;
```

## Where used?

- Employee–manager hierarchy
- Friend recommendations
- Parent–child relationships
- Organizational charts

## Anti Join

### What is it?

Find **records in table A that do NOT exist in table B.**

### Real-life intuition

- Customers **who never placed an order**
- Students **not enrolled in any course**

### ◊ Method 1: LEFT JOIN + IS NULL

```
SELECT c.*  
FROM customers c  
LEFT JOIN orders o  
ON c.id = o.customer_id  
WHERE o.customer_id IS NULL;
```

## ◊ Method 2: NOT EXISTS (preferred)

```
SELECT *
FROM customers c
WHERE NOT EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.id
);
```

### Where used?

- Inactive users
- Missing records
- Data quality checks

## 3 Semi Join

### What is it?

Return rows **from table A that HAVE a match in table B, without returning table B's columns.**

### Real-life intuition

- Customers **who have placed at least one order**
- Students **who passed any exam**

## ◊ Using EXISTS (classic semi-join)

```
SELECT *
FROM customers c
WHERE EXISTS (
    SELECT 1
```

```
FROM orders o
WHERE o.customer_id = c.id
);
```

## ◊ Using IN (simple but less optimal)

```
SELECT *
FROM customers
WHERE id IN (
    SELECT customer_id FROM orders
);
```

## ⚠ Difference from JOIN?

- JOIN may duplicate rows
- SEMI JOIN returns each row **only once**

## ↳ Finding Duplicates

### ⌚ Very common interview + production task

#### 🌐 Example problem

Find duplicate emails.

## ◊ Step 1: Identify duplicates

```
SELECT email, COUNT(*)
FROM users
GROUP BY email
HAVING COUNT(*) > 1;
```

## ◊ Step 2: Fetch full duplicate rows

```
SELECT *
FROM users
WHERE email IN (
    SELECT email
    FROM users
    GROUP BY email
    HAVING COUNT(*) > 1
);
```

## ◊ Step 3: Remove duplicates (keep one)

```
DELETE FROM users
WHERE id NOT IN (
    SELECT MIN(id)
    FROM users
    GROUP BY email
);
```

### Where used?

- Cleaning messy data
- Enforcing uniqueness
- Analytics accuracy

## 5 Gaps & Islands Problem

(This separates beginners from advanced SQL devs)

## ⌚ What is it?

Find **continuous ranges (islands)** or **missing values (gaps)** in ordered data.

## ⌚ Example

Login days:

user_id	login_date
1	2024-01-01
1	2024-01-02
1	2024-01-04
1	2024-01-05

👉 Gap on **2024-01-03**

### ❖ Island detection using ROW\_NUMBER()

```
SELECT user_id,
       MIN(login_date) AS start_date,
       MAX(login_date) AS end_date
  FROM (
    SELECT * ,
           login_date - ROW_NUMBER() OVER (
             PARTITION BY user_id ORDER BY login_date
           ) AS grp
      FROM logins
    ) t
 GROUP BY user_id, grp;
```

## ⌚ Why this works?

- Consecutive dates keep same difference
- Breaks create a new group

## Where used?

- Attendance streaks
- Subscription continuity
- Sensor data analysis
- Fraud detection

## Summary — PART 5

Concept	Purpose
Self Join	Relationship within same table
Anti Join	Find missing data
Semi Join	Check existence
Duplicates	Data cleanup
Gaps & Islands	Time-series analysis

# PART 6 — Real Interview & Production SQL Patterns

## 1 Nth Highest Salary

### Problem

Find the **Nth highest salary** from an employees table.

## Human thinking

- Remove duplicates
- Sort salaries
- Pick the Nth one

## Example table

id	name	salary
1	A	50000
2	B	60000
3	C	60000
4	D	80000

## Solution 1: Using DISTINCT + LIMIT/OFFSET

```
SELECT DISTINCT salary
FROM employees
ORDER BY salary DESC
OFFSET 1
LIMIT 1;
```

 (2nd highest salary)

## Solution 2: Using DENSE\_RANK (INTERVIEW FAVORITE)

```
SELECT salary
FROM (
  SELECT salary,
    DENSE_RANK() OVER (ORDER BY salary DESC) AS rnk
  FROM employees
```

```
) t  
WHERE rnk = 2;
```

## Why DENSE\_RANK?

Function	Duplicate behavior
ROW_NUMBER	Counts duplicates
RANK	Skips ranks
DENSE_RANK	No gaps <input checked="" type="checkbox"/>

## Mini Cheat Sheet

```
Nth highest → DENSE_RANK  
Unique salary → DISTINCT
```

## Top-N Per Group

### Problem

Find **top 2 highest-paid employees per department**.

### Real-world use

- Top sellers per region
- Top scorers per team
- Best students per class

## Example

name	dept	salary
A	CSE	90
B	CSE	80
C	CSE	70
D	ECE	85
E	ECE	75

## Correct solution (Window Function)

```
SELECT *
FROM (
  SELECT *,  
    DENSE_RANK() OVER (  
      PARTITION BY dept  
      ORDER BY salary DESC  
    ) AS rnk  
  FROM employees  
) t  
WHERE rnk <= 2;
```

## Why GROUP BY fails?

GROUP BY **loses row-level details.**

## Mini Cheat Sheet

Top-N per group → PARTITION BY + ORDER BY

## 3 Running Totals

### ⌚ Problem

Calculate **cumulative sales over time**.

### ⌚ Real-world use

- Bank balances
- Daily revenue tracking
- User activity count

### 💡 Example

date	sales
Jan-01	100
Jan-02	200
Jan-03	150

### ✅ SQL (Running Total)

```
SELECT date,  
       sales,  
       SUM(sales) OVER (  
           ORDER BY date  
           ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW  
       ) AS running_total  
FROM sales;
```

### ⌚ Key idea

Each row “remembers” all previous rows.

## Mini Cheat Sheet

Running total → `SUM() OVER (ORDER BY)`

## Time-Based Queries

### Problem

Analyze **time differences, trends, gaps.**

### Use cases

- User inactivity
- Late payments
- Session duration

### Example (Login gaps)

user_id	login_date
1	2024-01-01
1	2024-01-05

### Find days between logins

```
SELECT user_id,
       login_date,
       login_date - LAG(login_date) OVER (
           PARTITION BY user_id ORDER BY login_date
       ) AS days_gap
  FROM logins;
```

Filter users inactive > 7 days

```
SELECT *
FROM (
  SELECT user_id,
    login_date - LAG(login_date) OVER (
      PARTITION BY user_id ORDER BY login_date
    ) AS gap
  FROM logins
) t
WHERE gap > 7;
```

 Mini Cheat Sheet

```
Previous row → LAG()
Next row → LEAD()
Time gap → date - LAG(date)
```

 Final Mental Model (IMPORTANT)

Pattern	Tool
Ranking	DENSE_RANK
Group Top-N	PARTITION BY
Accumulation	SUM OVER
Time difference	LAG

## Where you stand now

- Beginner SQL — DONE
- Intermediate SQL — DONE
- Advanced SQL — DONE
- Interview-level SQL — DONE

You are now **production-ready in SQL**.

## DAY 6 — PART 1

This is the **most important backend skill** you'll learn.

### BIG PICTURE

Till now:

PostgreSQL → you used via **CLI**

- FastAPI → you used with **in-memory data / MongoDB**
- Now we connect:

FastAPI → SQLAlchemy ORM → PostgreSQL

### Real-world analogy

- PostgreSQL = Warehouse
- SQLAlchemy = **Manager who talks SQL**
- FastAPI = **Receptionist / API**
- Client (React / Swagger) = **Customer**
- Client never talks to DB directly.

Everything goes through **FastAPI** → **SQLAlchemy**.



## WHAT WE WILL BUILD (CLEAR GOAL)

A **Student Database API** with:

- Create student
- Get all students
- Get student by ID
- Update student
- Delete student
- Filter students
- Department → Students (One-to-Many)



## PROJECT STRUCTURE

```
backend/
|
├── main.py
├── database.py
└── models/
    └── student.py
├── schemas/
    └── student.py
└── routers/
    └── students.py
└── requirements.txt
```

## STEP 1 — Install Required Packages

### Packages we need

```
pip install fastapi uvicorn sqlalchemy psycopg2-binary
```

### Why these?

Package	Why
fastapi	API framework
uvicorn	Server
sqlalchemy	ORM (talks to DB)
psycopg2-binary	PostgreSQL driver

### Mini Cheat Sheet

```
SQLAlchemy = ORM  
psycopg2 = PostgreSQL connector
```

Here ORM stands for **Object–Relational Mapping**.

### In simple words

ORM is a technique that lets you interact with a **database using objects and classes** instead of writing raw SQL queries.

### Breakdown of the term

- **Object** → Python classes & objects
- **Relational** → Tables, rows, columns in SQL databases
- **Mapping** → Connecting class fields ↔ table columns

### Example (without ORM – raw SQL)

```
SELECT * FROM students WHERE id = 1;
```

### Same thing (with ORM – SQLAlchemy)

```
student = db.query(Student).filter(Student.id == 1).first()
```

Here:

- Student → Python class
- students → Database table
- student.id → Table column

### Why ORM is used

- ✓ No need to write SQL again and again
- ✓ Cleaner, readable code
- ✓ Prevents SQL injection
- ✓ Works across different databases
- ✓ Easy integration with frameworks like **FastAPI**, **Django**, **Flask**

## Popular ORMs

- **SQLAlchemy** (FastAPI favorite)
- Django ORM
- Hibernate (Java)
- Sequelize (Node.js)

## One-line interview answer

**ORM (Object–Relational Mapping) is a technique that maps database tables to programming language classes, allowing developers to interact with the database using objects instead of SQL queries.**

## STEP 2 — Database Connection (`database.py`)

## database.py

```
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker, declarative_base

DATABASE_URL = "postgresql://postgres:password@localhost:5432/college"

engine = create_engine(DATABASE_URL)

SessionLocal = sessionmaker(
    autocommit=False,
    autoflush=False,
    bind=engine
)

Base = declarative_base()
```



## Explain line by line

### create\_engine

- Creates **connection channel** to PostgreSQL
- Does NOT connect immediately
- Connects when first query is made

### SessionLocal

- Represents **one DB transaction**
- Every API request gets **its own session**

### Base

- Parent class for **all models**
- SQLAlchemy uses this to create tables

### Mini Cheat Sheet

Engine = connection

Session = transaction

Base = table blueprint

## STEP 3 — SQL Model (Table Definition)

### **models/student.py**

```
from sqlalchemy import Column, Integer, String
from database import Base

class Student(Base):
    __tablename__ = "students"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, nullable=False)
    age = Column(Integer)
    email = Column(String, unique=True, index=True)
```

### **VERY IMPORTANT CONCEPT**

**This class = SQL TABLE**

Python

PostgreSQL

class	table
Column	column
Integer	INT
String	VARCHAR

### tablename

This is **actual DB table name.**

### **primary\_key=True**

- Uniquely identifies each row
- Auto indexed

### **Mini Cheat Sheet**

ORM model = table

Column = DB column

## **STEP 4 — Create Tables in DB**

## main.py

```
from fastapi import FastAPI
from database import engine
from models.student import Student

app = FastAPI()

Student.metadata.create_all(bind=engine)
```

## What happens here?

SQLAlchemy checks:

- Does students table exist?
- If not → creates it

 You do NOT write CREATE TABLE manually

## Mini Cheat Sheet

metadata.create\_all = create tables

## STEP 5 — Pydantic Schemas (API Validation)

## schemas/student.py

```
from pydantic import BaseModel

class StudentCreate(BaseModel):
    name: str
    age: int
    email: str

class StudentResponse(BaseModel):
    id: int
    name: str
    age: int
    email: str

class Config:
    from_attributes = True
```

## Why schemas?

Purpose	Reason
Input validation	Prevent bad data
Output control	Hide DB internals
Docs	Swagger UI

 ORM ≠ API schema

## Mini Cheat Sheet

Schema = API data shape

Model = DB structure

## ■ STEP 6 — Dependency: Database Session

### 📁 database.py (add this)

```
from sqlalchemy.orm import Session
```

```
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```



### Why this?

- Opens DB session per request
- Closes automatically
- Prevents connection leaks



### Mini Cheat Sheet

Dependency = shared logic

## ■ STEP 7 — CRUD APIs (students.py)

### 📁 routers/students.py

```
from fastapi import APIRouter, Depends, HTTPException
from sqlalchemy.orm import Session
from database import get_db
from models.student import Student
from schemas.student import StudentCreate, StudentResponse

router = APIRouter(prefix="/students", tags=["Students"])
```

## ● CREATE STUDENT

```
@router.post("/", response_model=StudentResponse)
def create_student(student: StudentCreate, db: Session = Depends(get_db)):
    new_student = Student(**student.dict())
    db.add(new_student)
    db.commit()
    db.refresh(new_student)
    return new_student
```

### 🧠 Key steps

Create ORM object

1. Add to session
2. Commit transaction
3. Refresh to get ID
- 4.

## GET ALL STUDENTS

```
@router.get("/", response_model=list[StudentResponse])
def get_students(db: Session = Depends(get_db)):
    return db.query(Student).all()
```

## GET BY ID

```
@router.get("/{student_id}")
def get_student(student_id: int, db: Session = Depends(get_db)):
    student = db.query(Student).filter(Student.id == student_id).first()
    if not student:
        raise HTTPException(404, "Student not found")
    return student
```

## UPDATE

```
@router.put("/{student_id}")
def update_student(student_id: int, data: StudentCreate, db: Session = Depends(get_db)):
    student = db.query(Student).filter(Student.id == student_id).first()
    if not student:
        raise HTTPException(404, "Student not found")

    for key, value in data.dict().items():
        setattr(student, key, value)

    db.commit()
    return student
```

## ● DELETE

```
@router.delete("/{student_id}")
def delete_student(student_id: int, db: Session = Depends(get_db)):
    student = db.query(Student).filter(Student.id == student_id).first()
    if not student:
        raise HTTPException(404, "Student not found")

    db.delete(student)
    db.commit()
    return {"message": "Deleted"}
```



## CRUD Cheat Sheet

```
add → insert
query → select
commit → save
delete → remove
```



## STEP 8 — Register Router

### 📁 main.py

```
from routers import students

app.include_router(students.router)
```

## ■ STEP 9 — One-to-Many Relationship (Concept Preview)

Example:

Department → Students

One department has many students

You'll learn:

ForeignKey

- relationship()
- back\_populates
-  We'll do this **next**, step-by-step.

## ✓ WHAT YOU HAVE COMPLETED NOW

- ✓ PostgreSQL + FastAPI connection
- ✓ SQLAlchemy ORM basics
- ✓ CRUD APIs
- ✓ Dependency injection
- ✓ Production folder structure

---

## PostgreSQL + FastAPI (FULL PRACTICAL, IN-DEPTH)

Goal:

By the end, you should be able to **design, build, explain, and debug** a FastAPI + PostgreSQL backend using SQLAlchemy ORM confidently.

# STEP 0 — THE BIG PICTURE (VERY IMPORTANT)

Before touching code, you must understand **WHAT TALKS TO WHAT**.

**In your system there are 4 layers:**

Client (Swagger / React)

↓

FastAPI (Routes & Validation)

↓

SQLAlchemy ORM (Python ↔ SQL translator)

↓

PostgreSQL (Actual Database)

## Key rule

- FastAPI  does NOT talk SQL
- PostgreSQL  does NOT understand Python
- **SQLAlchemy ORM is the bridge**

# STEP 1 — WHAT IS SQLALCHEMY (MENTAL MODEL)

**ORM = Object Relational Mapping**

Instead of writing this:

```
INSERT INTO students (name, age) VALUES ('Aman', 21);
```

You write this:

```
student = Student(name="Aman", age=21)
db.add(student)
db.commit()
```

👉 SQLAlchemy **converts Python objects → SQL** internally.

## 🧠 How ORM thinks

Database	ORM
Table	Class
Row	Object
Column	Attribute
Foreign Key	Relationship

## 🧱 STEP 2 — PROJECT STRUCTURE (INDUSTRY STANDARD)

We do **clean backend**, not messy scripts.

```
backend/
|
├── main.py
├── database.py
└── models/
    ├── __init__.py
    ├── student.py
    └── department.py
```

```
|  
|   └── schemas/  
|       ├── student.py  
|       └── department.py  
|  
└── routers/  
    └── student.py  
  
└── venv/
```

📌 This structure scales to production.

## 🔌 STEP 3 — DATABASE CONNECTION (CORE FOUNDATION)

### database.py

This file answers ONE question:

“How does FastAPI talk to PostgreSQL?”

```
from sqlalchemy import create_engine  
from sqlalchemy.orm import sessionmaker, declarative_base
```

#### 1 Connection URL

```
DATABASE_URL = "postgresql://username:password@localhost:5432/college"
```

Meaning:

postgresql://USER:PASSWORD@HOST:PORT/DATABASE

## 2 Engine (SQL Connector)

```
engine = create_engine(DATABASE_URL)
```

📌 Engine = **physical connection pipe**

## 3 Session (Transaction Manager)

```
SessionLocal = sessionmaker(  
    autocommit=False,  
    autoflush=False,  
    bind=engine  
)
```

📌 Session =

- Opens DB connection
- Executes queries
- Commits / Rollbacks

## 4 Base (ORM Foundation)

```
Base = declarative_base()
```

📌 Every ORM model **must inherit from Base**

## 5 Dependency (FastAPI magic)

```
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```



Why?

- One request = one DB session
- Auto close after response



## MINI CHEAT SHEET (DATABASE)

Component	Purpose
engine	Connects to DB
SessionLocal	Executes queries
Base	Parent for models
get_db	Dependency injection

# STEP 4 — ORM MODELS (MOST IMPORTANT PART)

**Student table → Python class**

**models/student.py**

```
from sqlalchemy import Column, Integer, String
from database import Base

class Student(Base):
    __tablename__ = "students"

    id = Column(Integer, primary_key=True, index=True)
    name = Column(String)
    age = Column(Integer)
    email = Column(String, unique=True)
```

## How this maps

Python	PostgreSQL
class	students
Student	table
id	primary key
name	VARCHAR
email	UNIQUE

## STEP 5 — CREATE TABLES (AUTOMATIC)

### main.py

```
from fastapi import FastAPI
from database import engine
from models import student

student.Base.metadata.create_all(bind=engine)

app = FastAPI()
```

💡 What happens:

- SQLAlchemy scans all models
- Generates SQL
- Creates tables if not exists

## THIS IS HUGE

You **DO NOT** write **CREATE TABLE SQL** manually anymore.

ORM does it.

# STEP 6 — SCHEMAS (FASTAPI ⇄ ORM SAFETY)

## Why schemas exist?

Problem	Solution
Accept request data	Pydantic schema
Hide passwords	Response schema
Validate input	Schema

## schemas/student.py

```
from pydantic import BaseModel

class StudentCreate(BaseModel):
    name: str
    age: int
    email: str

class StudentResponse(BaseModel):
    id: int
    name: str
    age: int
    email: str

class Config:
    orm_mode = True
```

 orm\_mode = True  
→ allows SQLAlchemy objects → JSON



## MINI CHEAT SHEET (SCHEMAS)

Schema	Used for
StudentCreate	POST request
StudentResponse	API response



## STEP 7 — CRUD APIs (REAL WORK)

### routers/student.py

```
from fastapi import APIRouter, Depends
from sqlalchemy.orm import Session
from database import get_db
from models.student import Student
from schemas.student import StudentCreate, StudentResponse
```

### ✚ CREATE STUDENT

```
@router.post("/students", response_model=StudentResponse)
def create_student(student: StudentCreate, db: Session = Depends(get_db)):
    db_student = Student(**student.dict())
    db.add(db_student)
    db.commit()
    db.refresh(db_student)
    return db_student
```

🧠 What happens:

1. Convert request → object
2. Add to session
3. Commit transaction

4. Refresh to get ID

## READ STUDENTS

```
@router.get("/students")
def get_students(db: Session = Depends(get_db)):
    return db.query(Student).all()
```

## CRUD CHEAT SHEET

Operation	ORM
Insert	db.add()
Save	db.commit()
Read	db.query()
Delete	db.delete()

## STEP 8 — ONE-TO-MANY RELATIONSHIP (CRITICAL)

### Department → Students

```
class Department(Base):
    __tablename__ = "departments"

    id = Column(Integer, primary_key=True)
    name = Column(String)

    students = relationship("Student", back_populates="department")
```

```
class Student(Base):
    department_id = Column(Integer, ForeignKey("departments.id"))
    department = relationship("Department", back_populates="students")
```

- 📌 ORM now understands relationships automatically.

## 🎯 WHERE WE ARE NOW

### Completed in this explanation:

- ✓ SQLAlchemy mental model
- ✓ Database connection
- ✓ ORM models
- ✓ Table creation
- ✓ CRUD APIs
- ✓ Relationships
- ✓ Schemas