What is SQL?

SQL stands for **Structured Query Language**.

It is a standard programming language used to **store, manage, and retrieve data** from **relational databases** like MySQL, PostgreSQL, Oracle, and SQL Server.

What Can SQL do?

SQL (Structured Query Language) is a standard language used for managing and manipulating relational databases. Its primary uses include
SQL lets you design the structure of your database (tables, columns, keys)

What is Database?

A database is a structured collection of data, typically stored electronically in a computer system, that allows for efficient storage, retrieval, and management of information. It's designed to hold large amounts of organized data, often accessed by multiple users or programs simultaneously.

Or

A **Database** is an organized collection of **data** that can be **stored, managed, and retrieved** easily using a computer system.

A **Database = Data + Structure + Access**

Stores **information** like customer details, product catalogs, financial transactions, etc.

Can be **small (Excel file)** or **huge (Amazon, Google, Facebook data)**.

Managed by a **DBMS (Database Management System)** such as **MySQL, Oracle, SQL Server, MongoDB**.

Types of Databases

1. **Relational Database (RDBMS)**
   - Stores data in **tables (rows & columns)**.

- Example: **MySQL, PostgreSQL, Oracle, SQL Server**
- Query language: **SQL**

2. **NoSQL Database**
   - Stores unstructured data like JSON, documents, graphs.
   - Example: **MongoDB, Cassandra, Firebase**
3. **Cloud Databases**
   - Hosted on the cloud, scalable.
   - Example: **AWS RDS, Google Cloud Firestore, Azure SQL**

Working of Databases?

A **Database** works like a **smart system** that **stores, organizes, and delivers data** efficiently whenever needed.

1. Data Storage

- Data is stored in **tables (RDBMS)** or **documents/collections (NoSQL)**.
- Example: In a **Bank Database**:
  - `Customers` table → Name, Account No, Phone
  - `Transactions` table → Amount, Date, Type

2. Data Input (Insert)

- When you **add new information**, the database saves it.
- Example: A new customer opens an account → record stored in `Customers` table.

```
INSERT INTO Customers (id, name, balance) VALUES (1,
'Vijay', 5000);
```

3. Data Processing (Queries)

- You send a **query** (request) using **SQL** or API.

- The DBMS **searches, filters, and processes** the data.
- Example: Checking your account balance.

```
SELECT balance FROM Customers WHERE name='Vijay';
```

4. Data Output (Results)

- The database **returns the result** in seconds.
- Example: *Balance = ₹5,000* is shown on your banking app.

What is a database in SQL?

In **SQL**, a **Database** is a **container** that stores all the data in an organized way using **tables (rows & columns)**.
It is the **highest-level structure** in SQL where all the information like tables, views, procedures, and indexes are kept.

🔑 Key Points

- A **Database** is like a **digital folder** 📂.
- Inside it, you can create **tables**, store records (rows), and organize data.
- You use **SQL commands** to create, manage, and interact with it.

🏗️ Example in SQL

1. **Create a Database**

```
CREATE DATABASE company;
```

2. **Use the Database**

```
USE company;
```

3. **Create a Table inside Database**

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    salary DECIMAL(10,2)
);
```

4. **Insert Data**

```
INSERT INTO employees (emp_id, name, salary)
VALUES (1, 'Vijay', 25000.00);
```

5. **Query Data**

```
SELECT * FROM employees;
```

What Is Sql Syntax?

SQL Syntax refers to the **rules and structure** of writing commands in SQL (Structured Query Language).
Just like English has grammar, **SQL has syntax** that tells the database what action to perform.

🔑 Basic SQL Syntax Rules

1. **SQL Keywords** (like `SELECT`, `INSERT`, `UPDATE`, `DELETE`) are **not case-sensitive**.
   - `SELECT` = `select` = `Select`
     (but best practice = always write in UPPERCASE)
2. **Statements end with a semicolon** (;)
   - Example: `SELECT * FROM employees;`
3. **Strings are written inside single quotes** (' ')
   - Example: `'Vijay'`
4. **Identifiers** (like table names, column names) should not have spaces.

🏗️ Common SQL Syntax Examples

1. Create a Database

```
CREATE DATABASE company;
```

2. Use a Database

```
USE company;
```

3. Create a Table

```
CREATE TABLE employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    salary DECIMAL(10,2)
);
```

4. Insert Data

```
INSERT INTO employees (emp_id, name, salary)
VALUES (1, 'Vijay', 25000.00);
```

5. Select Data

```
SELECT name, salary FROM employees WHERE salary > 20000;
```

6. Update Data

```
UPDATE employees SET salary = 28000 WHERE emp_id = 1;
```

7. Delete Data

```
DELETE FROM employees WHERE emp_id = 1;
```

✅ **In short:**
SQL syntax = **the grammar of SQL** → how you write commands so the database understands what you want to do (create, insert, update, delete, query).

Types of DBMS?

📌 Types of DBMS (Database Management Systems)

A **DBMS** is software that helps to **store, organize, and manage data** in a database. There are different **types of DBMS** based on how data is structured and managed.

🔑 1. Hierarchical DBMS

- Data is stored in a **tree-like structure** (parent → child).
- Each child record has **only one parent**.
- Fast for searching hierarchical data.

📌 Example: **IBM IMS (Information Management System)**

🔑 2. Network DBMS

- Data stored as **records (nodes)** connected by **links (edges)** like a graph.
- One record can have **multiple parent and child records** (many-to-many).

📌 Example: **IDMS (Integrated Data Management System)**

🔑 3. Relational DBMS (RDBMS)

- Data stored in **tables (rows & columns)**.
- Most common type, uses **SQL** for queries.
- Supports **relationships** (primary key, foreign key).

📌 Examples: **MySQL, Oracle, PostgreSQL, SQL Server**

🔑 4. Object-Oriented DBMS (OODBMS)

- Data stored as **objects** (like in programming languages such as Java, C++).
- Supports features like **inheritance, polymorphism**.

📌 Example: **db4o, ObjectDB**

🔑 5. NoSQL DBMS

- Stores **unstructured/semi-structured data**.
- Used for **big data & real-time apps**.
- Types of NoSQL:
  - Document-based (MongoDB, CouchDB)
  - Key-Value (Redis, DynamoDB)
  - Column-based (Cassandra, HBase)
  - Graph-based (Neo4j)

📌 Examples: **MongoDB, Cassandra, Redis**

✅ Summary Table

Type	Structure	Examples
Hierarchical DBMS	Tree (Parent-Child)	IBM IMS
Network DBMS	Graph (Many-to-Many)	IDMS
Relational DBMS (RDBMS)	Tables (Rows/Cols)	MySQL, Oracle, PostgreSQL
Object-Oriented DBMS	Objects	db4o, ObjectDB
NoSQL DBMS	Documents, Graphs	MongoDB, Cassandra, Redis, Neo4j

✅ **In short:**
Most companies today use **RDBMS (SQL)** and **NoSQL DBMS** depending on whether the data is structured or unstructured

DDL?/DML/DCL/DQL/TCL?

1. DDL – Data Definition Language

👉 Used to **define and manage database structure** (tables, schemas, etc.).

- Affects the **structure** of the database, not the data inside.

Commands:

- `CREATE` → Create database, table, index
- `ALTER` → Modify table (add/remove columns)
- `DROP` → Delete database or table
- `TRUNCATE` → Remove all data from table (structure remains)

📌 Example:

```
CREATE TABLE Students (
   id INT PRIMARY KEY,
   name VARCHAR(50),
   age INT
);
```

📌 2. DML – Data Manipulation Language

👉 Used to **manipulate (change) data** in tables.

Commands:

- `INSERT` → Add new data
- `UPDATE` → Modify existing data
- `DELETE` → Remove data

📌 Example:

```
INSERT INTO Students (id, name, age) VALUES (1, 'Vijay',
24);
```

📌 3. DCL – Data Control Language

👉 Used to **control access/permissions** to the database.

Commands:

- `GRANT` → Give access
- `REVOKE` → Take back access

📌 Example:

```
GRANT SELECT, INSERT ON Students TO user1;
```

📌 4. DQL – Data Query Language

👉 Used to **query (fetch) data** from the database.

Command:

- `SELECT` → Retrieve data

📌 Example:

```
SELECT name, age FROM Students WHERE age > 20;
```

📌 5. TCL – Transaction Control Language

👉 Used to **manage transactions** in a database (mainly in banking, financial apps).

Commands:

- `COMMIT` → Save changes permanently

- `ROLLBACK` → Undo changes
- `SAVEPOINT` → Mark a point in a transaction
- `SET TRANSACTION` → Define a transaction

📌 Example:

```
BEGIN;
UPDATE Students SET age = 25 WHERE id = 1;
COMMIT;
```

✅ Quick Summary Table

Category	Full Form	Purpose	Examples
DDL	Data Definition Language	Define database structure	CREATE, ALTER, DROP, TRUNCATE
DML	Data Manipulation Language	Change data in tables	INSERT, UPDATE, DELETE
DCL	Data Control Language	Control access/ permissions	GRANT, REVOKE
DQL	Data Query Language	Retrieve/query data	SELECT
TCL	Transaction Control Language	Manage transactions	COMMIT, ROLLBACK, SAVEPOINT

✅ In short:

- **DDL** → Structure
- **DML** → Data changes
- **DCL** → Permissions
- **DQL** → Fetch data
- **TCL** → Transactions

SQL Database?

An **SQL Database** is a **relational database** that stores and manages data in **tables (rows & columns)** and uses **SQL (Structured Query Language)** to interact with the data.

👉 It is also called an **RDBMS (Relational Database Management System).**

🔑 Key Features of an SQL Database

1. **Table-Based Structure** → Data is stored in tables with rows (records) and columns (fields).
2. **SQL Language** → You use SQL commands to **Create, Read, Update, Delete (CRUD)** data.
3. **Relationships** → Different tables can be linked using **Primary Keys** & **Foreign Keys**.
4. **Data Integrity** → Ensures accuracy with constraints (NOT NULL, UNIQUE, CHECK, etc.).
5. **ACID Properties** → Transactions are reliable:
    - **Atomicity** – All or nothing
    - **Consistency** – Data must be valid
    - **Isolation** – Transactions don't affect each other
    - **Durability** – Data is saved permanently

🏗️ Example of an SQL Database

Let's say we create a **School Database**:

Create a Database

```
CREATE DATABASE SchoolDB;
```

Create a Table

```
CREATE TABLE Students (
```

```
    student_id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT,
    grade VARCHAR(10)
);
```

Insert Data

```
INSERT INTO Students (student_id, name, age, grade)
VALUES (1, 'Vijay', 24, 'A');
```

Fetch Data

```
SELECT * FROM Students;
```

🔎 Popular SQL Databases

- **MySQL** (open-source, widely used in web apps)
- **PostgreSQL** (advanced open-source database)
- **Oracle Database** (enterprise-level, paid)
- **SQL Server (by Microsoft)**
- **MariaDB** (MySQL fork)

✅ **In short:**
An **SQL Database** is a **relational database** that uses **SQL** to store, manage, and retrieve structured data efficiently.

SQL Tables?

In an **SQL Database**, a **Table** is the **main structure** used to store data in a **tabular format (rows & columns)**.

👉 Think of a table like an **Excel sheet**:

- **Columns** → Fields (Name, Age, Salary, etc.)
- **Rows** → Records (actual data of each person/item)

🔑 Key Points about SQL Tables

1. **Each table has a unique name** inside a database.
2. **Columns define the type of data** (INTEGER, VARCHAR, DATE, etc.).
3. **Rows store the actual data values.**
4. Tables can be linked together using **Primary Keys** and **Foreign Keys**.

🏗️ Example of a Table in SQL

Create a Table

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    age INT,
    salary DECIMAL(10,2),
    department VARCHAR(50)
);
```

Insert Data into Table

```
INSERT INTO Employees (emp_id, name, age, salary,
department)
VALUES (1, 'Vijay', 24, 25000.00, 'Operations');
```

Select Data from Table

```
SELECT * FROM Employees;
```

📌 Result:

emp_id	name	age	salary	department
1	Vijay	24	25000.00	Operations

🔎 Types of Tables in SQL

1. **Permanent Tables** – Standard tables in the database.
2. **Temporary Tables** – Exist only during the session/query.

```
CREATE TEMPORARY TABLE temp_students (...);
```

3.

4. **System Tables** – Created by DBMS to store metadata.
5. **Partitioned Tables** – Split large data into smaller sections for performance.

✅ **In short:**
An **SQL Table** is where your data lives — rows (records) and columns (fields). Almost everything in SQL starts with creating and working with tables.

SQL Queries?

An **SQL Query** is a **request (command)** you send to the database to **create, read, update, or delete** data.

- Written using **SQL syntax**.
- The database engine processes the query and returns results.

Think of it like:
🗣️ *"Hey database, show me all employees with salary > 20,000."*
➡️ The query would be:

```
SELECT * FROM employees WHERE salary > 20000;
```

🔑 Types of SQL Queries

1. Data Definition Queries (DDL)

- Define **database structure**.

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

2. Data Manipulation Queries (DML)

- Change the **data inside tables**.

```
INSERT INTO students (id, name, age) VALUES (1, 'Vijay',
24);
UPDATE students SET age = 25 WHERE id = 1;
DELETE FROM students WHERE id = 1;
```

3. Data Query Language (DQL)

- **Retrieve (fetch) data**.

```
SELECT name, age FROM students WHERE age > 20;
```

4. Data Control Queries (DCL)

- Manage **user permissions**.

```
GRANT SELECT ON students TO user1;
```

5. Transaction Control Queries (TCL)

- Manage **transactions** (banking, finance apps).

```
BEGIN;
UPDATE students SET age = 26 WHERE id = 1;
COMMIT;
```

🏗️ Example Queries

1. **Show all employees**

```
SELECT * FROM employees;
```

2. **Find employees in "Operations" department**

```
SELECT name, salary FROM employees WHERE department =
'Operations';
```

3. **Increase salary by 10% for all employees**

```
UPDATE employees SET salary = salary * 1.10;
```

4. **Delete all records of employees younger than 20**

```
DELETE FROM employees WHERE age < 20;
```

✅ **In short:**
SQL Queries are the way you **talk to a database** → ask questions, insert, update, delete, or manage data.

Difference between Primary Key and Foreign Key

Primary Key vs Foreign Key in SQL

🔑 Primary Key

- A **unique identifier** for each record in a table.
- Ensures **no duplicate** and **no NULL values**.
- A table can have **only one Primary Key** (but it can consist of multiple columns → Composite Key).

✅ Example:

```
CREATE TABLE Students (
   student_id INT PRIMARY KEY,
   name VARCHAR(50),
   age INT
);
```

Here, `student_id` uniquely identifies each student.

🔑 Foreign Key

- A **reference** to the Primary Key in another table.
- Creates a **relationship** between two tables.
- Ensures **referential integrity** (a record in one table must match a valid record in another).

✅ Example:

```
CREATE TABLE Courses (
   course_id INT PRIMARY KEY,
   course_name VARCHAR(50)
);

CREATE TABLE Students (
   student_id INT PRIMARY KEY,
   name VARCHAR(50),
   course_id INT,
   FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);
```

Here, `course_id` in **Students** is a **Foreign Key** → it must exist in **Courses**.

⚡ Key Differences

Feature	Primary Key 🗝	Foreign Key 🔗
Purpose	Uniquely identifies each record in a table	Establishes a relationship between two tables
Uniqueness	Must be unique	Can have duplicate values

NULL values	Not allowed	Allowed (unless specified as NOT NULL)
Number per Table	Only one Primary Key	Multiple Foreign Keys allowed
Defined On	Single table	References another table's Primary Key
Example	`student_id` in Students table	`course_id` in Students table referencing Courses table

✅ Simple Real-Life Example

- **Primary Key** → Aadhaar Number (unique for each person).
- **Foreign Key** → Aadhaar Number linked in a Bank Account table (reference to identify person).

✅ **In short:**

- **Primary Key = Unique ID inside a table.**
- **Foreign Key = A link between two tables (points to another table's Primary Key).**

SQL Clauses?

What are SQL Clauses?

👉 An **SQL Clause** is a **keyword or condition** that is used inside an SQL query to **filter, organize, or control** the data.
They are like **rules** you add to your queries.

For example:

```
SELECT name, salary
FROM employees
WHERE salary > 20000
ORDER BY salary DESC;
```

Here:

- `WHERE` and `ORDER BY` are **SQL Clauses**.

🔑 Important SQL Clauses

1. WHERE Clause

👉 Filters rows based on a condition.

```
SELECT * FROM employees WHERE department = 'IT';
```

2. ORDER BY Clause

👉 Sorts data in ascending (ASC) or descending (DESC) order.

```
SELECT * FROM employees ORDER BY salary DESC;
```

3. GROUP BY Clause

👉 Groups rows that have the same values into summary rows.

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

4. HAVING Clause

👉 Works like `WHERE` but for groups (used with `GROUP BY`).

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

5. LIMIT / TOP Clause

👉 Restricts the number of rows returned.

- In **MySQL / PostgreSQL**:

```
SELECT * FROM employees LIMIT 5;
```

- In **SQL Server**:

```
SELECT TOP 5 * FROM employees;
```

6. DISTINCT Clause

👉 Removes duplicate values.

```
SELECT DISTINCT department FROM employees;
```

✅ Quick Summary Table

Clause	Purpose
WHERE	Filter rows by condition
ORDER BY	Sort results (ASC/DESC)
GROUP BY	Group rows for aggregation
HAVING	Filter grouped data
LIMIT/ TOP	Restrict number of rows
DISTINCT	Remove duplicate values

✅ **In short:**
SQL Clauses are extra conditions we attach to queries to **filter, sort, group, and limit** data.

SQL Operators?

What are SQL Operators?

👉 **SQL Operators** are **symbols or keywords** used in SQL queries to perform operations on data, such as **comparison, arithmetic, logical checks, or pattern matching**.

They help us **filter and manipulate data** inside `WHERE`, `HAVING`, and other clauses.

🔑 Types of SQL Operators

1. Arithmetic Operators

👉 Used for mathematical calculations.

Operator	Use	Example
`+`	Addition	`SELECT salary + 500 FROM employees;`
`-`	Subtraction	`SELECT salary - 1000 FROM employees;`
`*`	Multiplication	`SELECT salary * 1.1 FROM employees;`
`/`	Division	`SELECT salary / 12 FROM employees;`
`%`	Modulus (Remainder)	`SELECT 10 % 3; -- Output = 1`

2. Comparison Operators

👉 Compare values (returns `TRUE` or `FALSE`).

Operator	Meaning	Example
`=`	Equal	`WHERE salary = 25000`
`<>` or `!=`	Not Equal	`WHERE department <> 'HR'`
`>`	Greater Than	`WHERE salary > 20000`
`<`	Less Than	`WHERE age < 30`
`>=`	Greater or Equal	`WHERE age >= 18`
`<=`	Less or Equal	`WHERE salary <= 50000`

3. Logical Operators

👉 Combine multiple conditions.

Operator	Meaning	Example
`AND`	All conditions must be true	`WHERE department='IT' AND salary > 20000`
`OR`	At least one condition true	`WHERE department='IT' OR department='HR'`
`NOT`	Negates condition	`WHERE NOT department='HR'`

4. Special Operators

👉 SQL has some operators for special filtering.

Operato	Use	Example
`BETWEEN`	Range check	`WHERE salary BETWEEN 20000 AND 40000`
`IN`	Match against list	`WHERE department IN ('IT','HR')`
`LIKE`	Pattern match	`WHERE name LIKE 'V%'` (names starting
`IS NULL`	Check for NULL values	`WHERE email IS NULL`
`EXISTS`	Check if subquery returns results	`WHERE EXISTS (SELECT * FROM orders)`

5. Set Operators

👉 Combine results of multiple queries.

Operator	Meaning	Example
`UNION`	Combine results,	`SELECT name FROM emp1 UNION`
`UNION ALL`	Combine results, keep duplicates	`SELECT name FROM emp1 UNION ALL SELECT name FROM emp2;`
`INTERSECT`	Common rows only	`SELECT name FROM emp1 INTERSECT SELECT name FROM emp2;`

`MINUS` (or `EXCEPT`)	Rows in first query but not in second	`SELECT name FROM emp1 MINUS SELECT name FROM emp2;`

✅ Quick Example

```
SELECT name, salary
FROM employees
WHERE salary > 20000 AND department IN ('IT', 'Finance')
ORDER BY salary DESC;
```

✅ **In short:**
SQL Operators are the **tools** you use inside queries to **calculate, compare, filter, and combine** data.

SQL Aggregate Functions ?

SQL Aggregate Functions

👉 **Aggregate Functions** in SQL are used to **perform calculations on a set of rows** and return a **single value**.
They are often used with **`GROUP BY`** to summarize data.

🔑 Main SQL Aggregate Functions

1. COUNT()

👉 Returns the **number of rows**.

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

📌 *Counts all employees.*

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

📌 *Counts employees in each department.*

2. SUM()

👉 Returns the **total sum of a numeric column**.

```
SELECT SUM(salary) FROM employees;
```

📌 *Total salary of all employees.*

3. AVG()

👉 Returns the **average value**.

```
SELECT AVG(salary) FROM employees;
```

📌 *Average salary of employees.*

4. MIN()

👉 Returns the **minimum value**.

```
SELECT MIN(salary) FROM employees;
```

📌 *Lowest salary.*

5. MAX()

👉 Returns the **maximum value**.

```
SELECT MAX(salary) FROM employees;
```

📌 *Highest salary.*

🏗️ Example with GROUP BY

Suppose we have an `employees` table:

emp_id	name	department	salary
1	Vijay	IT	25000
2	Amit	HR	20000
3	Neha	IT	30000
4	Rahul	Finance	28000

Query:

```
SELECT department, COUNT(*) AS total_employees,
AVG(salary) AS avg_salary
FROM employees
GROUP BY department;
```

Result:

department	total_employees	avg_salary
IT	2	27500
HR	1	20000
Finance	1	28000

✅ Quick Summary

Function	Purpose
COUNT()	Count rows
SUM()	Total sum
AVG()	Average value
MIN()	Minimum value
MAX()	Maximum value

✅ **In short:**
SQL Aggregate Functions are used to **summarize data** (total, average, min, max, count).
They are super powerful when combined with **GROUP BY**.

Data Constraints?

What are Data Constraints in SQL?

👉 **Constraints** in SQL are **rules applied to table columns** to ensure the **accuracy, reliability, and integrity** of the data.
They prevent invalid data from being inserted into the database.

Think of them as **rules/locks** for your table. 🔒

🔑 Types of Data Constraints

1. NOT NULL

👉 Ensures that a column **cannot have NULL values**.

```
CREATE TABLE students (
   id INT NOT NULL,
   name VARCHAR(50) NOT NULL
);
```

📌 *You must provide `id` and `name`, they cannot be empty.*

2. UNIQUE

👉 Ensures all values in a column are **unique** (no duplicates).

```
CREATE TABLE employees (
   email VARCHAR(100) UNIQUE
);
```

📌 *No two employees can have the same email.*

3. PRIMARY KEY

👉 Combines **NOT NULL + UNIQUE**.

- Identifies each row uniquely.

```
CREATE TABLE employees (
   emp_id INT PRIMARY KEY,
   name VARCHAR(50)
);
```

📌 *Each `emp_id` must be unique & not null.*

4. FOREIGN KEY

👉 Creates a relationship between **two tables**.

```
CREATE TABLE orders (
   order_id INT PRIMARY KEY,
   emp_id INT,
   FOREIGN KEY (emp_id) REFERENCES employees(emp_id)
);
```

📌 *`emp_id` in `orders` must exist in `employees`.*

5. CHECK

👉 Ensures values meet a specific condition.

```
CREATE TABLE employees (
   emp_id INT PRIMARY KEY,
   age INT CHECK (age >= 18)
);
```

📌 *Employee age must be 18 or above.*

6. DEFAULT

👉 Assigns a default value if none is provided.

```
CREATE TABLE employees (
   emp_id INT PRIMARY KEY,
   salary DECIMAL(10,2) DEFAULT 20000
);
```

📌 *If no salary is entered, it defaults to 20,000.*

7. AUTO_INCREMENT / IDENTITY (MySQL / SQL Server)

👉 Automatically generates unique values.

```
CREATE TABLE employees (
   emp_id INT PRIMARY KEY AUTO_INCREMENT,
   name VARCHAR(50)
);
```

📌 *Automatically increases emp_id: 1,2,3…*

✅ Quick Summary Table

Constraint	Purpose
NOT NULL	Prevents NULL values
UNIQUE	No duplicate values
PRIMARY KEY	Unique + Not Null, identifies a record
FOREIGN KEY	Links two tables
CHECK	Ensures condition is true
DEFAULT	Assigns default value
AUTO_INCREMENT	Auto-generates IDs

✅ **In short:**
Data Constraints = Rules on columns to keep data **valid, accurate, and consistent.**

SQL Joins?

What are SQL Joins?

👉 **SQL JOIN** is used to **combine data from two or more tables** based on a related column (usually a **Primary Key** and a **Foreign Key**).

Think of it like:

- Table A = Employees
- Table B = Departments
- A JOIN links them together to show which employee belongs to which department.

🔑 Types of SQL Joins

1. INNER JOIN

👉 Returns **only matching records** from both tables.

```
SELECT e.name, d.department_name
FROM employees e
INNER JOIN departments d
ON e.dept_id = d.dept_id;
```

📌 *Shows only employees who are assigned to a department.*

2. LEFT JOIN (or LEFT OUTER JOIN)

Mishra Tech Hub

👉 Returns **all records from the left table**, and matching records from the right table.
If no match → `NULL`.

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

```
LEFT JOIN departments d
ON e.dept_id = d.dept_id;
```

📌 *Shows all employees, even if they don't have a department.*

3. RIGHT JOIN (or RIGHT OUTER JOIN)

👉 Returns **all records from the right table**, and matching from the left table.

```
SELECT e.name, d.department_name
FROM employees e
RIGHT JOIN departments d
ON e.dept_id = d.dept_id;
```

📌 *Shows all departments, even those with no employees.*

4. FULL JOIN (or FULL OUTER JOIN)

👉 Returns **all records from both tables**.

- If no match, fills with `NULL`.

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

📌 *Shows all employees and all departments, even if unmatched.*

(Note: MySQL doesn't support FULL JOIN directly, you use `UNION` of LEFT + RIGHT join.)

5. CROSS JOIN

👉 Returns the **Cartesian product** (all possible combinations).

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

```
CROSS JOIN departments d;
```
📌 *If 5 employees × 3 departments = 15 rows.*

6. SELF JOIN

👉 A table joins **with itself**.

```
SELECT a.name AS Employee, b.name AS Manager
FROM employees a
INNER JOIN employees b
ON a.manager_id = b.emp_id;
```
📌 *Shows employees and their managers from the same table.*

✅ Quick Visual (Summary)

Join Type	Result
INNER JOIN	Only matching rows
LEFT JOIN	All rows from left + matches from right
RIGHT JOIN	All rows from right + matches from left
FULL JOIN	All rows from both tables
CROSS JOIN	All combinations
SELF JOIN	Join table with itself

✅ **In short:**
SQL Joins allow you to **combine data from multiple tables** into a single result set, making it powerful for relational databases.

SQL Views?

What is a SQL View?

👉 A **View** in SQL is a **virtual table** created using a query.

- It looks like a table but **doesn't store data itself**.
- Instead, it **stores a SQL query** and shows results whenever you call it.

Think of it like a **saved shortcut query**.

🔑 Key Features of Views

- A view is created using the `CREATE VIEW` statement.
- You can query a view just like a normal table.
- Views simplify **complex queries**.
- They provide **security** by showing only selected columns/rows.
- Views are always **up-to-date** because they pull data from the underlying tables.

🏗️ Example

Suppose we have a table `employees`:

emp_id	name	department	salary
1	Vijay	IT	25000
2	Neha	HR	20000
3	Amit	IT	30000

Create a View

```
CREATE VIEW IT_Employees AS
SELECT name, salary
FROM employees
WHERE department = 'IT';
```

Use the View

```
SELECT * FROM IT_Employees;
```

📌 Output:

name	salary
Vijay	25000
Amit	30000

🔎 Types of Views

1. **Simple View** → Based on a single table.
2. **Complex View** → Based on multiple tables with joins.
3. **Materialized View** → Actually stores data (used in Oracle, PostgreSQL). Faster but needs refresh.

✅ Advantages of Views

- Simplifies complex queries.
- Provides **data security** (users see only required data).
- Ensures **data independence** (changes in table don't affect users).
- Easier to maintain frequently used queries.

❌ Limitations

- Cannot always perform `INSERT`, `UPDATE`, `DELETE` on views (especially complex ones).
- Performance may be slower for large joins (since it runs the underlying query each time).

✅ **In short:**
A **SQL View** is like a **saved query** that acts as a **virtual table** for easier access, security, and simplified reporting.

SQL Indexes?

What is an SQL Index?

👉 An **Index** in SQL is like an **index in a book**.

- Instead of reading the entire book (table), the DBMS uses the index to **quickly find data**.
- An index is created on **columns** to speed up **searching, filtering, and sorting**.

📌 Without Index → Full Table Scan (slow on big tables).

📌 With Index → Quick Lookup (fast).

🔑 Creating an Index

```
CREATE INDEX idx_employee_name
ON employees(name);
```

👉 Now, searching by name is much faster:

```
SELECT * FROM employees WHERE name = 'Vijay';
```

🔎 Types of Indexes in SQL

1. Single-Column Index

- Created on one column.

```
CREATE INDEX idx_salary ON employees(salary);
```

2. Composite (Multi-Column) Index

- Created on two or more columns.

```
CREATE INDEX idx_dept_salary
ON employees(department, salary);
```

3. Unique Index

- Ensures all values in a column are **unique** (like UNIQUE constraint).

```
CREATE UNIQUE INDEX idx_email ON employees(email);
```

4. Clustered Index

- Sorts and stores rows in **physical order**.
- Only **one clustered index per table** (usually on Primary Key).

```
CREATE CLUSTERED INDEX idx_empid ON employees(emp_id);
```

5. Non-Clustered Index

- Does **not** change table order.
- Can have **multiple non-clustered indexes**.

```
CREATE NONCLUSTERED INDEX idx_name ON employees(name);
```

6. Full-Text Index

- Used for searching text efficiently (LIKE, keywords).

✅ Advantages of Indexes

- 🚀 Speeds up `SELECT` queries (search, filter, sort, join).
- Reduces disk I/O.
- Improves performance for large databases.

❌ Disadvantages of Indexes

- Slows down `INSERT`, `UPDATE`, `DELETE` (because index also updates).
- Takes extra storage space.

🏗 Example

Table: `Employees`

emp_id	name	department	salary
1	Vijay	IT	25000
2	Neha	HR	20000
3	Amit	IT	30000

If we create an index on `department`,

```
CREATE INDEX idx_dept ON employees(department);
```
Query:

```
SELECT * FROM employees WHERE department = 'IT';
```
👉 Runs much faster because it uses the index instead of scanning the whole table.

✅ **In short:**
An **Index** in SQL is a performance tool that helps the database **find data faster**, like a book index.

SQL Subquery ?

What is a Subquery in SQL?

👉 A **Subquery** is a query **inside another query**.

- It is enclosed in parentheses `( )`.
- The result of the subquery is used by the main query.

Think of it like:

- **Inner query** = helper query
- **Outer query** = main query

🔑 Types of Subqueries

1. Single-Row Subquery

👉 Returns only **one value**.

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

📌 *Shows employees who earn more than the average salary.*

2. Multi-Row Subquery

👉 Returns **multiple values** (used with `IN`, `ANY`, `ALL`).

```
SELECT name
FROM employees
WHERE department_id IN (SELECT department_id FROM
departments WHERE location = 'Noida');
```

📌 *Find employees who work in departments located in Noida.*

3. Correlated Subquery

👉 The **inner query depends on the outer query** (runs for each row).

```
SELECT e1.name, e1.salary
FROM employees e1
WHERE salary > (SELECT AVG(e2.salary)
                FROM employees e2
                WHERE e1.department_id =
e2.department_id);
```

📌 *Shows employees earning more than their department average.*

4. Nested Subquery in SELECT

👉 Used in `SELECT` clause.

```
SELECT name,
       (SELECT department_name
        FROM departments d
        WHERE e.department_id = d.department_id) AS
dept_name
FROM employees e;
```

5. Subquery with EXISTS

👉 Checks if subquery returns any result (`TRUE/FALSE`).

```
SELECT name
FROM employees e
WHERE EXISTS (SELECT * FROM projects p WHERE e.emp_id =
p.emp_id);
```

📌 *Find employees who are assigned to at least one project.*

✅ Advantages of Subqueries

- Simplifies complex queries.
- Makes queries easier to read.
- Useful for filtering, comparisons, and calculations.

❌ Disadvantages

- Can be **slower** than `JOIN` in large datasets.
- Not all subqueries can be optimized well.

🏗 Example

Employees Table

emp_id	name	dept_id	salary
1	Vijay	101	25000
2	Neha	102	20000
3	Amit	101	30000

Departments Table

dept_id	dept_name
101	IT
102	HR

Query:

```
SELECT name
FROM employees
WHERE dept_id = (SELECT dept_id FROM departments WHERE
dept_name = 'IT');
```

📌 Result: Vijay, Amit

✅ **In short:**
A **Subquery** = Query inside another query.
It can return **single values, multiple values, or be correlated** with the outer query.

Database Design and Modeling?

Database Design and Modeling

👉 **Database Design & Modeling** is the process of **planning and structuring** how data will be stored, related, and accessed in a database.
It's like creating the **blueprint (architecture)** of your database before building it.

🔑 Steps in Database Design

1. Requirement Analysis

- Understand the business needs.
- Example: In a **College System**, you need data about Students, Courses, Teachers, Fees.

2. Identify Entities and Attributes

- **Entities** = Things you want to store data about (Students, Teachers, Courses).
- **Attributes** = Properties of those entities (Student Name, Age, Roll No, etc.).

Example:

- Student → (student_id, name, age, course_id)
- Course → (course_id, course_name, credits)

3. Define Relationships

- How entities are connected.
- Example:
  - A Student **enrolls in** a Course.

- A Teacher **teaches** a Course.

4. Create an ER Diagram (Entity-Relationship Model)

📌 Example ERD:

```
STUDENT (student_id, name, age, course_id)
   |
   |-- enrolls in
   |
COURSE (course_id, course_name, credits)
   |
   |-- taught by
   |
TEACHER (teacher_id, name, subject)
```

5. Normalization

👉 Organize data into multiple related tables to **remove redundancy**.

- **1NF** – Each column has atomic values.
- **2NF** – Every non-key column depends on the whole primary key.
- **3NF** – No transitive dependency (non-key depending on non-key).

📌 Example: Instead of storing teacher details in the course table → create a separate Teacher table.

6. Choose Keys

- **Primary Key**: Uniquely identifies each record (e.g., student_id).
- **Foreign Key**: Links between tables (student.course_id → course.course_id).

7. Implement Tables

👉 Convert the model into SQL tables.

```
CREATE TABLE Students (
   student_id INT PRIMARY KEY,
   name VARCHAR(50),
   age INT,
   course_id INT,
   FOREIGN KEY (course_id) REFERENCES Courses(course_id)
);

CREATE TABLE Courses (
   course_id INT PRIMARY KEY,
   course_name VARCHAR(50),
   credits INT
);
```

8. Test the Design

- Insert sample data.
- Run queries to check if design works for real needs.

✅ Benefits of Good Database Design

- 🚀 Fast queries (optimized).
- 🔒 Data consistency & accuracy.
- 📦 Less redundancy (no duplicate data).
- 🔗 Easy to scale and maintain.

⚡ Example: E-commerce Database Design

Entities:

- Customers (customer_id, name, email)
- Orders (order_id, date, amount, customer_id)

- Products (product_id, name, price)
- Order_Items (order_id, product_id, quantity)

Relationships:

- A Customer places many Orders.
- An Order contains many Products.

✅ **In short:**
Database Design & Modeling = creating a **blueprint** (ER model, normalization, relationships) to ensure data is **organized, consistent, and efficient** before implementation.

Database Security?

Database Security

👉 **Database Security** means protecting a database from:

- Unauthorized access 🚫
- Data breaches 🛡️
- Data loss or corruption 💾
- Misuse by internal or external threats 👨‍💻

It includes **tools, processes, and controls** to keep the data **safe, private, and reliable**.

🔑 Key Threats to Database Security

1. **Unauthorized Access** – Hackers or unapproved users accessing data.
2. **SQL Injection Attacks** – Malicious queries that manipulate your database.
3. **Data Loss/Corruption** – Due to hardware failure, system crash, or human error.
4. **Malware & Ransomware** – Attackers encrypt or steal sensitive data.

5. **Insider Threats** – Employees misusing data.

🛡️ Best Practices for Database Security

1. User Authentication & Access Control

- Use **strong passwords**, 2FA (two-factor authentication).
- Apply **Principle of Least Privilege** → Give users only the permissions they need.

```
GRANT SELECT, INSERT ON employees TO user1;
```

2. Encryption

- **Data-at-Rest Encryption** → Secure stored data.
- **Data-in-Transit Encryption (SSL/TLS)** → Protect data moving over networks.

3. Regular Backups

- Schedule **automatic backups**.
- Store backups in **secure, separate locations**.

4. SQL Injection Prevention

- Always use **Prepared Statements / Parameterized Queries**.
  ❌ Unsafe:

```
"SELECT * FROM users WHERE name = '" + userInput + "'";
```

✅ Safe:

```
SELECT * FROM users WHERE name = ?;
```

5. Database Monitoring

- Track **login attempts, query patterns, unusual activity**.

- Use tools like **SIEM (Security Information and Event Management)**.

6. Patching and Updates

- Keep DBMS software **up-to-date** (MySQL, SQL Server, Oracle, PostgreSQL).

7. Firewalls & Network Security

- Restrict database access to **trusted IPs** only.
- Place databases behind **application servers**, not exposed directly.

8. Auditing & Logging

- Maintain logs of who accessed what data and when.
- Helps in detecting breaches and compliance.

✅ Example: Banking Database Security

- Encrypt customer account details.
- Limit access → Teller can only view transactions, not modify accounts.
- Automatic daily backups.
- Regular security audits.

⚡ Quick Summary

Security Measure	Purpose
Authentication & Roles	Control access
Encryption	Protect data (stored & in transit)
Backups	Prevent data loss
SQL Injection Prevention	Stop hacking attacks

Monitoring & Logging	Detect suspicious activity
Updates & Patches	Fix vulnerabilities
Firewalls	Block unauthorized access

✅ **In short:**
Database Security = Protecting data's confidentiality, integrity, and availability (CIA).

Database Connectivity?

What is Database Connectivity?

👉 **Database Connectivity** means **connecting an application (like Python, Java, Web App, etc.) to a database (MySQL, SQL Server, PostgreSQL, etc.)** so the app can:

- **Insert** data
- **Retrieve** data
- **Update** data
- **Delete** data

📌 Example: When you sign up on a website → your **Name, Email, Password** are saved into a database using **database connectivity**.

🔑 Steps for Database Connectivity

1. **Install Database & Driver**
   - Example: MySQL, PostgreSQL, Oracle.
   - Install connector/driver (e.g., MySQL Connector for Python).
2. **Establish Connection**
   - Use a connection string (host, username, password, database name).
3. **Execute SQL Queries**
   - Send queries (INSERT, SELECT, UPDATE, DELETE).

4. **Fetch Results**
    - Get data from the database.
5. **Close the Connection**
    - Always close to avoid memory leaks.

🏗 Examples of Database Connectivity

🔷 1. Python with MySQL

```
import mysql.connector

# Step 1: Connect to Database
conn = mysql.connector.connect(
    host="localhost",
    user="root",
    password="yourpassword",
    database="testdb"
)

cursor = conn.cursor()

# Step 2: Run Query
cursor.execute("SELECT * FROM employees")

# Step 3: Fetch Data
for row in cursor.fetchall():
    print(row)

# Step 4: Close Connection
conn.close()
```

🔷 2. Java with JDBC (MySQL)

```
import java.sql.*;
```

```
class DBConnect {
    public static void main(String args[]) {
        try {
            // Step 1: Load Driver
            Class.forName("com.mysql.cj.jdbc.Driver");

            // Step 2: Connect
            Connection con = DriverManager.getConnection(
                "jdbc:mysql://localhost:3306/testdb",
"root", "yourpassword");

            // Step 3: Execute Query
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT *
FROM employees");

            // Step 4: Print Results
            while (rs.next()) {
                System.out.println(rs.getInt(1) + "  " +
rs.getString(2));
            }

            // Step 5: Close
            con.close();

        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

🔷 3. Web Application (PHP + MySQL)

```
<?php
$conn = new mysqli("localhost", "root", "yourpassword",
"testdb");

if ($conn->connect_error) {
```

```
    die("Connection failed: " . $conn->connect_error);
}

$result = $conn->query("SELECT * FROM employees");

while($row = $result->fetch_assoc()) {
    echo $row["id"]. " - " . $row["name"]. "<br>";
}

$conn->close();
?>
```

✅ Why Database Connectivity is Important?

- Real-time apps (banking, e-commerce, social media).
- Dynamic websites (fetch & show user data).
- Enterprise apps (store employee & customer info).

⚡ Quick Summary

- **Database Connectivity** = Bridge between application ↔ Database.
- Done using **connectors/drivers**.
- Works with languages: **Python, Java, PHP, Node.js, C#, etc.**

✅ **In short:**
Database Connectivity lets apps talk to databases — insert, fetch, update, delete data — making apps dynamic and interactive.

Why do we need Normalization?

Why do we need Normalization in Databases?

👉 **Normalization** is the process of **organizing data in a database** to:

- **Reduce redundancy** (no duplicate data)
- **Improve data integrity** (accuracy & consistency)
- **Make queries efficient**

Think of it like **cleaning and organizing your cupboard** → everything is in the right place, no duplicates, and easy to find. 🗂️

🔑 Problems Without Normalization

Suppose we have a table `Students`:

StudentID	Name	Course	Teacher
1	Vijay	SQL	Mr. Sharma
2	Neha	Python	Ms. Gupta
3	Amit	SQL	Mr. Sharma

Issues ❌

1. **Redundancy** → "Mr. Sharma" is repeated.
2. **Update Anomaly** → If Mr. Sharma's name changes, we must update it in multiple rows.
3. **Insert Anomaly** → Cannot add a new teacher unless a student is enrolled.
4. **Delete Anomaly** → If Vijay is deleted, we also lose Mr. Sharma's record.

✅ How Normalization Fixes This

We split into **two tables**:

Students Table

StudentID	Name	CourseID
1	Vijay	101
2	Neha	102
3	Amit	101

Courses Table

CourseID	Course	Teacher
101	SQL	Mr. Sharma
102	Python	Ms. Gupta

Benefits ✔️

1. **No redundancy** → "Mr. Sharma" stored only once.
2. **Easy updates** → If Mr. Sharma changes, update only one row.
3. **Independent insertions** → We can add a course without students.
4. **Data consistency** → No conflicting info.

🔎 Reasons Why We Need Normalization

1. 📦 **Remove Data Redundancy** → No duplicate data.
2. 🔒 **Ensure Data Integrity** → Accurate & consistent data.
3. 🚀 **Optimize Queries** → Faster joins and smaller storage.
4. 🛠️ **Ease of Maintenance** → Easier to insert, update, delete.
5. 📊 **Better Organization** → Clear structure, scalable for growth.

✅ **In short:**
We need **Normalization** to keep databases **clean, efficient, consistent, and scalable** by removing redundancy and anomalies.

What is the Denormalization?
What is Denormalization in SQL?

👉 **Denormalization** is the process of **introducing redundancy back into a database** to improve **query performance**.

- In **Normalization**, we split tables into smaller related ones to remove redundancy.
- In **Denormalization**, we sometimes **combine tables or duplicate data** to make queries **faster** (especially in reporting, analytics, big databases).

📌 **Think of it like this:**

- Normalization = A clean, well-organized cupboard (everything separate, no duplicates).
- Denormalization = Bringing some items together again so they're faster to access.

🔑 Why Denormalization is Used?

1. 🚀 **Improve Performance** → Fewer joins, faster queries.
2. 📊 **Reporting & Analytics** → Complex queries run faster.
3. 💾 **Reduce Joins** → Avoid performance overhead in large datasets.
4. 🔗 **Better for Read-Heavy Systems** (like Data Warehouses).

🏗️ Example

Normalized (Separate Tables)

Students Table

StudentID	Name	CourseID
1	Vijay	101
2	Neha	102

Courses Table

CourseID	CourseName
101	SQL
102	Python

👉 To get student + course name → we need a `JOIN`.

```
SELECT s.Name, c.CourseName
FROM Students s
JOIN Courses c ON s.CourseID = c.CourseID;
```

Denormalized (Combined Table)

StudentCourses Table

StudentID	Name	CourseID	CourseName
1	Vijay	101	SQL
2	Neha	102	Python

👉 No need for JOINs — faster to query:

```
SELECT Name, CourseName FROM StudentCourses;
```

✅ Faster reads, but data redundancy exists (CourseName stored multiple times).

✅ Advantages of Denormalization

- 🚀 Faster query performance.
- ✅ Reduces complex joins.
- 📊 Useful for reporting & analytics.
- 🔗 Better for read-heavy applications.

❌ Disadvantages of Denormalization

- ❌ More redundancy (duplicate data).
- ❌ Update Anomalies (if "SQL" course name changes, update in many rows).
- ❌ More storage space needed.
- ❌ Harder to maintain consistency.

⚡ Quick Difference: Normalization vs Denormalization

Aspect	Normalization	Denormalization
Goal	Reduce redundancy, improve integrity	Improve speed, reduce joins
Storage	Efficient, less data	More storage (duplicate data)
Query Speed	Slower (joins needed)	Faster (less joins)
Use Case	Transaction systems (OLTP)	Reporting/Analytics (OLAP)

✅ **In short:**
Denormalization = adding redundancy back into a database to improve **performance and query speed**, at the cost of storage and consistency.

SQL Subquery?

What is a SQL Subquery?

👉 A **Subquery** is a query **inside another query**.

- Also called **nested query** or **inner query**.
- The result of the subquery is used by the **main (outer) query**.
- Subqueries are enclosed in **parentheses ( )**.

🔑 Types of Subqueries

1. Single-Row Subquery

👉 Returns **only one value**.

```
SELECT name, salary
FROM employees
WHERE salary > (SELECT AVG(salary) FROM employees);
```

✅ Finds employees who earn **more than the average salary**.

2. Multi-Row Subquery

👉 Returns **multiple values** (use with `IN`, `ANY`, `ALL`).

```
SELECT name
FROM employees
WHERE department_id IN (SELECT department_id FROM
departments WHERE location = 'Noida');
```

✅ Finds employees who work in **departments located in Noida**.

3. Correlated Subquery

👉 The **inner query depends on the outer query** (runs row by row).

```
SELECT e1.name, e1.salary
FROM employees e1
WHERE salary > (SELECT AVG(e2.salary)
                FROM employees e2
                WHERE e1.department_id =
e2.department_id);
```

✅ Finds employees earning **more than their department's average**.

4. Subquery in SELECT Clause

👉 Used to fetch extra calculated values.

```
SELECT name,
       (SELECT department_name
        FROM departments d
        WHERE e.department_id = d.department_id) AS
dept_name
FROM employees e;
```

✅ Shows employee name with their **department name**.

5. Subquery with EXISTS

👉 Checks if the subquery returns any result (`TRUE/FALSE`).

```
SELECT name
FROM employees e
WHERE EXISTS (SELECT * FROM projects p WHERE e.emp_id =
p.emp_id);
```

✅ Finds employees who are assigned to **at least one project**.

✅ Advantages of Subqueries

- Makes queries **simpler** and more **readable**.

- Useful for **filtering, comparisons, and calculations**.
- Helps when joins are complex.

❌ Disadvantages

- Can be **slower** than JOINs on large datasets.
- May require optimization.

⚡ Quick Example

Employees Table

emp_id	name	dept_id	salary
1	Vijay	101	25000
2	Neha	102	20000
3	Amit	101	30000

Departments Table

dept_id	dept_name
101	IT
102	HR

Query:

```
SELECT name
FROM employees
WHERE dept_id = (SELECT dept_id FROM departments WHERE
dept_name = 'IT');
```

✅ Result: Vijay, Amit

✅ **In short:**
A **Subquery = Query inside another query** used for filtering, comparison, or calculations.

SQL Stored Procedures

What is a Stored Procedure in SQL?

👉 A **Stored Procedure** is a **set of SQL statements** that are **precompiled and stored** in the database.

- You can **call** it whenever needed.
- It helps **automate tasks**, **reuse code**, and **improve performance**.

📌 Think of it as a **function in programming**, but inside the database.

🔑 Why use Stored Procedures?

1. 🚀 **Performance** – Precompiled, runs faster.
2. 🔒 **Security** – Limit direct access to tables, allow procedure execution.
3. 🔁 **Reusability** – Write once, call many times.
4. 🛠️ **Maintainability** – Easy to update logic in one place.

🏗️ Creating a Stored Procedure

Example 1: Simple Procedure

```
CREATE PROCEDURE GetAllEmployees()
AS
BEGIN
    SELECT * FROM Employees;
END;
```

📌 Usage:

```
EXEC GetAllEmployees;
```

Example 2: Procedure with Parameters

```
CREATE PROCEDURE GetEmployeeByDept
   @DeptID INT
AS
BEGIN
   SELECT name, salary
   FROM Employees
   WHERE department_id = @DeptID;
END;
```

📌 Usage:

```
EXEC GetEmployeeByDept @DeptID = 101;
```

Example 3: Insert Data Using Procedure

```
CREATE PROCEDURE AddEmployee
   @Name VARCHAR(50),
   @Age INT,
   @Salary DECIMAL(10,2),
   @DeptID INT
AS
BEGIN
   INSERT INTO Employees (name, age, salary,
department_id)
   VALUES (@Name, @Age, @Salary, @DeptID);
END;
```

📌 Usage:

```
EXEC AddEmployee 'Vijay', 24, 25000, 101;
```

Example 4: Procedure with Output Parameter

```
CREATE PROCEDURE GetEmployeeCount
    @DeptID INT,
    @EmpCount INT OUTPUT
AS
BEGIN
    SELECT @EmpCount = COUNT(*)
    FROM Employees
    WHERE department_id = @DeptID;
END;
```

📌 Usage:

```
DECLARE @Count INT;
EXEC GetEmployeeCount 101, @EmpCount = @Count OUTPUT;
PRINT @Count;
```

✅ Advantages of Stored Procedures

- Precompiled → faster execution.
- Code reuse → no repetition.
- Security → hide SQL logic from users.
- Reduce network traffic (send procedure call instead of long queries).

❌ Disadvantages

- More complex to maintain if too many procedures.
- Debugging can be harder compared to application code.
- Database dependent (syntax may vary between MySQL, SQL Server, Oracle, etc.).

⚡ Quick Real-Life Example

In a **Banking System**:

- A stored procedure can be written to **transfer money** between two accounts:

- Deduct from one account.
- Add to another.
- Ensure transaction is safe.

✅ **In short:**
Stored Procedure = Predefined SQL code stored in the database, reusable, secure, and faster.

-