



ALRIGHT!

CONGRATS!!

# SQL TUTORIAL



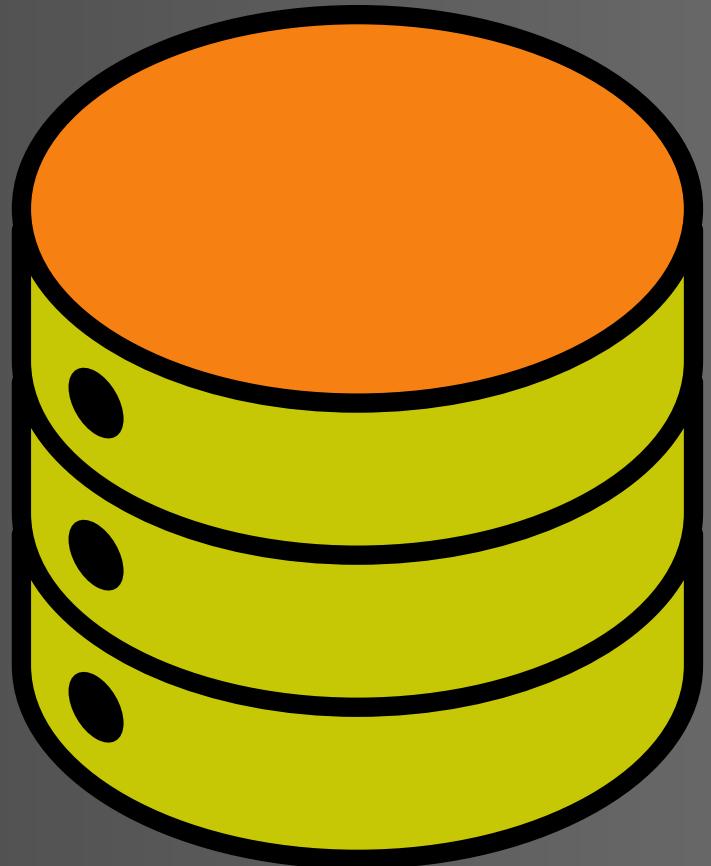
Microsoft SQL Server

***ZERO TO HERO***

# Section - 1

- What is database?
- Database vs DBMS
- DBMS vs RDBMS
- Other available databases
- SQL vs MS SQL
- Installation

# What is a Database?



# What is a Database?

A structured collection of data stored in a computer system that can be easily accessed and managed.

# Store details of students

**id**

**name**

**age**

**grade**



<b>Student_ID</b>	<b>Name</b>	<b>Age</b>	<b>Grade</b>
101	Raju	10	5
102	Sham	12	7
103	Baburao	14	9

# **DATABASE**

**vs**

# **DBMS**

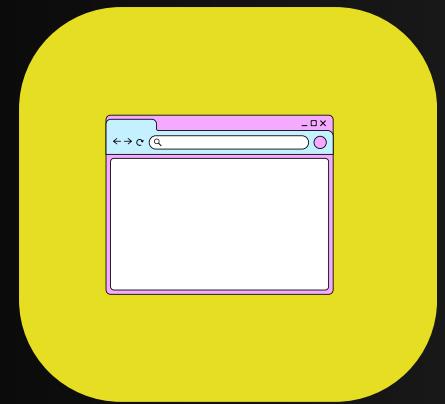
# DBMS

A DBMS (Database Management System) is software that is used to store, manage, and retrieve data efficiently and securely in a structured way.

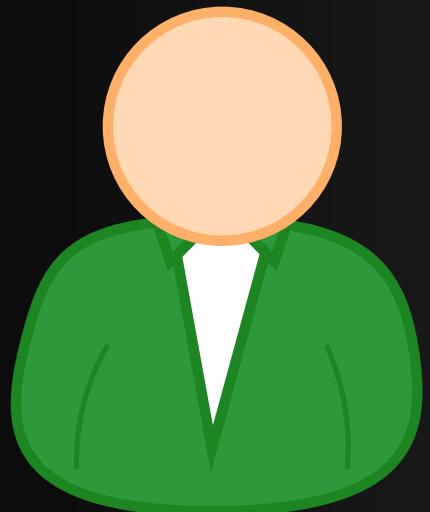




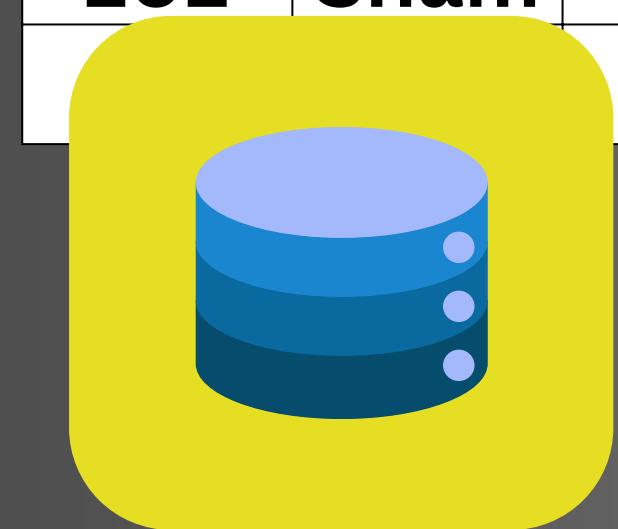
App



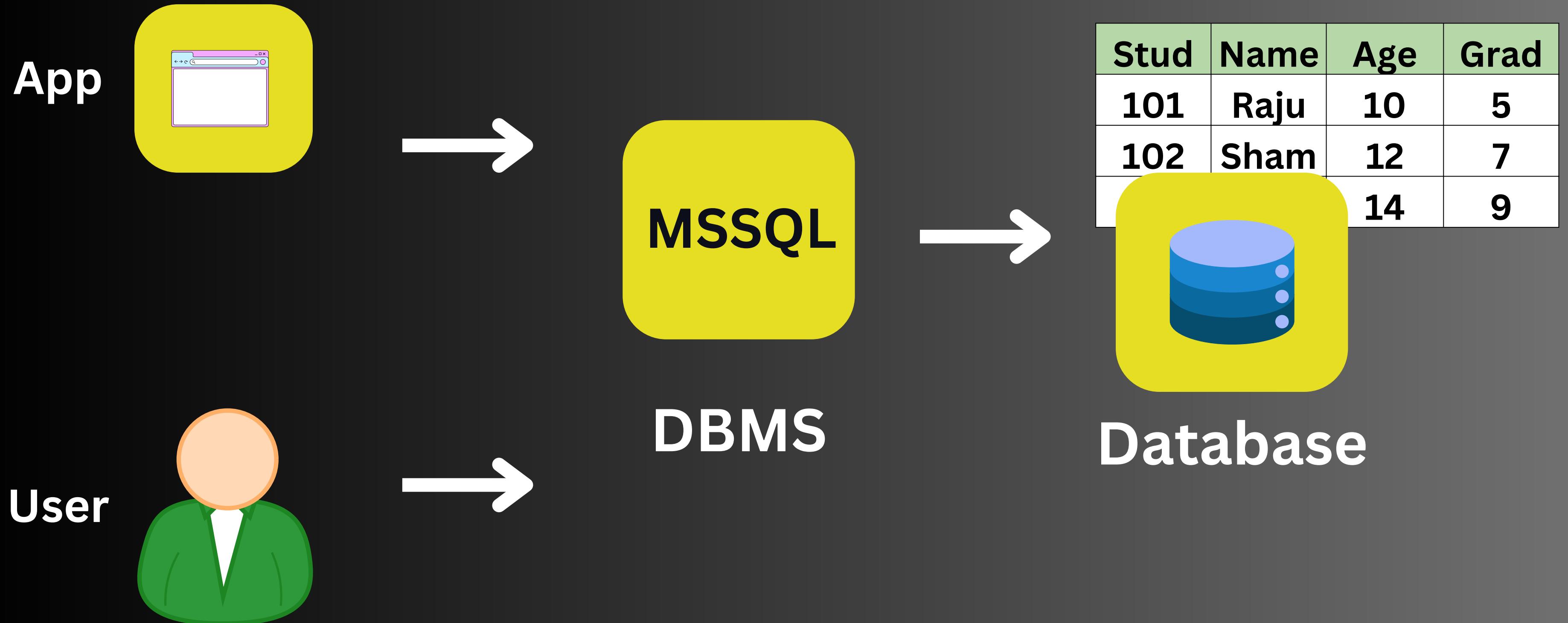
User

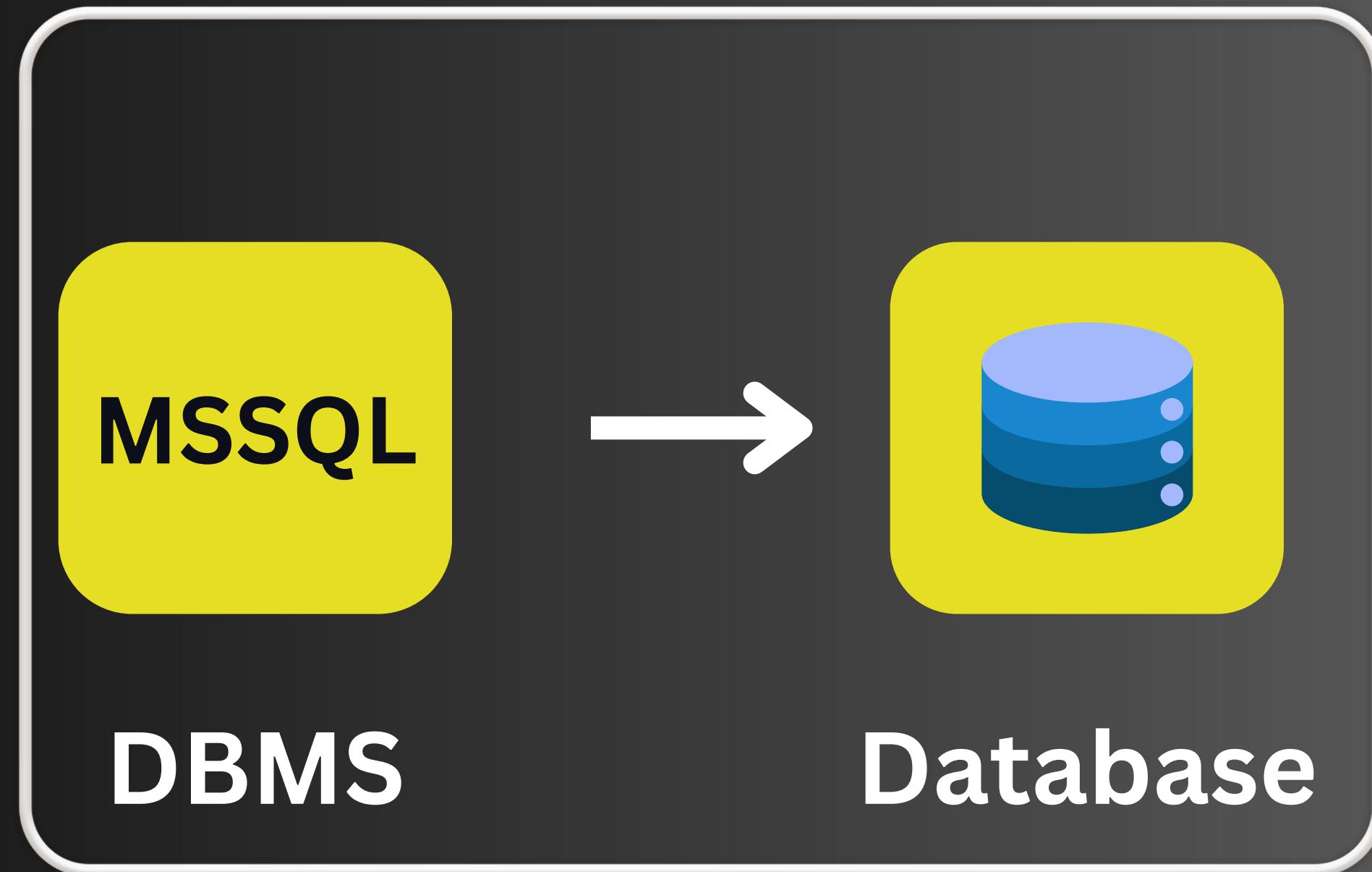


Stud	Name	Age	Grad
101	Raju	10	5
102	Sham	12	7
		14	9



Database





PostGresql, MySQL, **MSSQL** etc

# **What is RDBMS?**

# **What is RDBMS?**

A type of database system that stores data in structured tables (using rows and columns) and uses SQL for managing and querying data.

<b>Student_ID</b>	<b>Name</b>	<b>Age</b>	<b>Grade</b>
101	Raju	10	5
102	Sham	12	7
103	Baburao	14	9

What if we want to store the subjects each student is enrolled in and the marks they got?

Student_ID	Name	Age	Grade	Subject	Marks
101	Raju	10	5	Maths	95
101	Raju	10	5	Science	88
102	Sham	12	7	Maths	76
103	Baburao	14	9	History	91
103	Baburao	14	9	Maths	85

# marks

Subject	Marks	Student_ID
Maths	95	101
Science	88	101
Maths	76	102
History	91	103
Maths	85	103

# students

Student_ID	Name	Age	Grade
101	Raju	10	5
102	Sham	12	7
103	Baburao	14	9

# Some Other Databases are:

- Oracle
- MySQL
- PostgreSQL
- Firebird
- MongoDB
- Redis

# SQL vs MSSQL

# SQL

**Structured Query Language**  
**Which is used to talk to our databases.**

**Example: SELECT \* FROM person\_db;**



**SELECT \* FROM person\_db**

# Installation

**Download from the link**

**<https://www.microsoft.com/en-us/sql-server/sql-server-downloads>**

**Install SQL Server Management Studio (SSMS)**

# Section - 2

- Database
  - Creating, connect, listing, droping
- CRUD
  - Create - New Table
    - Inserting data
  - Read - How to read data
  - Update data
  - Delete data

# Databases

# List down existing databases

- **SELECT name FROM sys.databases;**

**EXEC sp\_databases;**



## Default System Databases in SQL Server – In Short

- **master** : Keeps track of all system-level info — databases, logins, settings. If it's down, SQL Server won't start.
- **model** : Template for new databases — any changes here apply to all newly created databases.
- **msdb** : Used by SQL Server Agent to store jobs, alerts, and backup history — helps automate tasks.
- **tempdb** : A temporary workspace for sorting, joins, and temp tables — it's reset every time the server restarts.

# **Creating a new Database**

**CREATE DATABASE <db\_name>;**

## Change or Use a Database

- **use <db\_name>;**

**SELECT DB\_NAME();**

# **Deleting a Database**

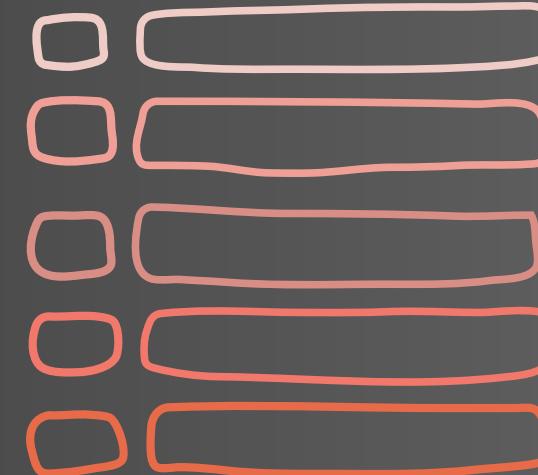
**DROP DATABASE <db\_name>;**

# CRUD

- CREATE
- READ
- UPDATE
- DELETE



UPDATE

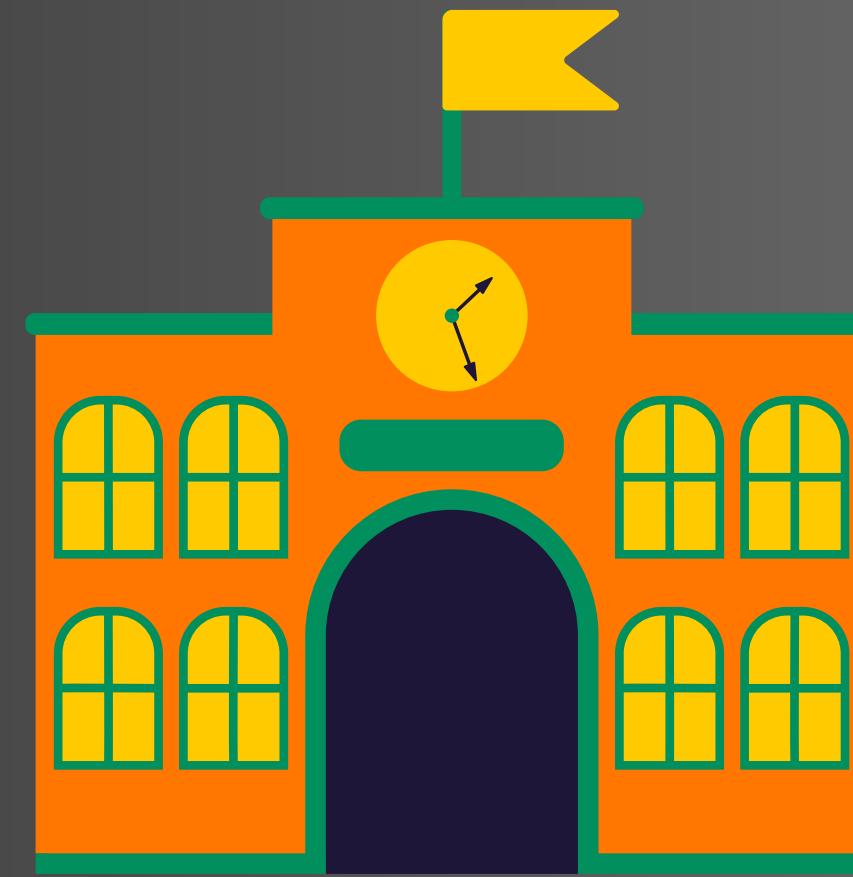
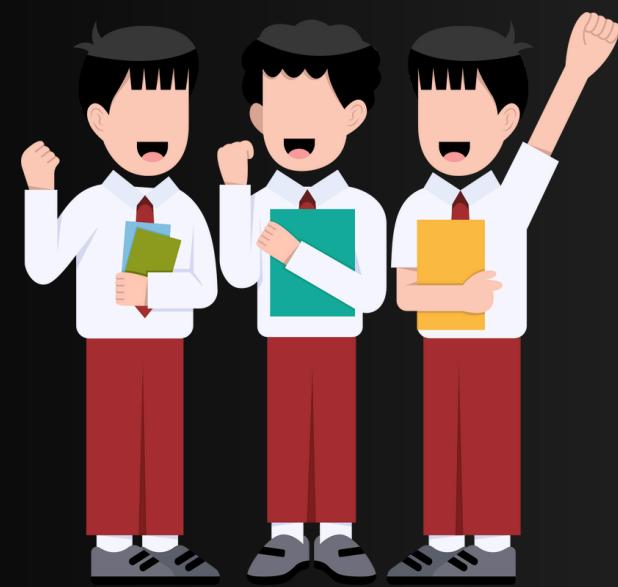


# CREATING Tables

# Table

A **table** is a collection of related data held in a table format within a database.

Student_ID	Name	Age	Grade
101	Raju	10	5
102	Sham	12	7
103	Baburao	14	9



# Creating a new Table

Student_ID	Name	Age	Grade
101	Raju	10	5
102	Sham	12	7
103	Baburao	14	9



Store details of  
students

# Creating a new Table

```
CREATE TABLE students (
    student_id INT,
    name VARCHAR(100),
    age INT,
    grade INT
);
```

```
CREATE TABLE students (id INT, name VARCHAR(100), city
VARCHAR(50));
```

# Checking your table

```
EXEC sp_help 'users';
```

# **INSERTING Data**

# Adding data into a Table

```
INSERT INTO students(student_id, name, age, grade)
VALUES (101, 'Raju', 10, 5);
```

```
INSERT INTO students VALUES (102, 'Sham', 12, 7)
```

# **READING DATA**

# Reading data from a Table

```
SELECT * FROM <table_name>
```

```
SELECT <column_name> from students
```

# UPDATING DATA

# Modify/Update data from a Table

```
UPDATE users  
SET city='London'  
WHERE id=102;
```

# **DELETING DATA**

# **DELETE data from a Table**

```
DELETE FROM users
WHERE name='Raju';
```

# TRUNCATE

**TRUNCATE TABLE employees;**

# **Exercise:**

- Write a query to change Grade of Raju from 5 to 6.
- Add a new student to the table:
  - Student\_ID = 104, Name = 'Alex', Age = 11, Grade = 6.
- Write a query to remove 'Baburao' from the table.
- Write a query to retrieve only the details for the student named 'Sham'.
- Write a query to print/get age of Raju

**WHERE**

- UPDATE students SET grade=12 WHERE student\_id=102;
- DELETE FROM students WHERE name='Raju';
- SELECT \* FROM students WHERE name='Raju'
- SELECT age FROM students WHERE name='Sham'

# Section - 3

- Datatypes
- Constraint

# DataTypes

An attribute that defines the kind of data a column in a database table can hold, such as numbers, text, dates, or boolean values.

```
CREATE TABLE students (
    student_id INT,
    name VARCHAR(100),
    age INT,
    grade INT
);
```

**Can we store this number in  
a column with INT  
datatype?**

**4,735,892,104,326**



**What will happen when  
we store values like  
15.35?**



# Most widely used are

- Numeric - INT | BIGINT | FLOAT | DECIMAL/NUMERIC
- String - VARCHAR | CHAR
- Date - DATE
- Date Time - DateTime
- Boolean - BIT (0/1)

<b>Data type</b>	<b>Range expression</b>	<b>Range</b>	<b>Storage</b>
<b>bigint</b>	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	-2^63 to 2^63-1	8 bytes
<b>int</b>	-2,147,483,648 to 2,147,483,647	-2^31 to 2^31-1	4 bytes
<b>smallint</b>	-32,768 to 32,767	-2^15 to 2^15-1	2 bytes
<b>tinyint</b>	0 to 255	2^0-1 to 2^8-1	1 byte

# DATA TYPES

DECIMAL(5,2)

Digits after decimal

Total digit

# **DECIMAL(5,2) DATATYPES**

**Example: 155.38**

- |        |   |
|--------|---|
| 119.12 | ✓ |
| 28.15  | ✓ |
| 1150.1 | ✗ |

# Constraint

A constraint decides what kind of data is allowed in a column.

- PRIMARY KEY
- NOT NULL
- DEFAULT
- IDENTITY
- UNIQUE

**Let's First Understand the Problems with  
Current Table Structure**

# Primary Key



- The PRIMARY KEY constraint uniquely identifies each record in a table.
- Primary keys must contain UNIQUE values, and cannot contain NULL values.
- A table can have only ONE primary key.

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL
);
```

```
CREATE TABLE Students (
    StudentID INT NOT NULL,
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    CONSTRAINT PK_Students PRIMARY KEY (StudentID)
);
```

If you need to use two or more columns to uniquely identify a record

```
CREATE TABLE OrderDetails (
    OrderID INT NOT NULL,
    ProductID INT NOT NULL,
    Quantity INT,
    CONSTRAINT PK_OrderDetails PRIMARY KEY (OrderID, ProductID)
);
```

# UNIQUE

- ◆ UNIQUE constraint makes sure that no two rows in a table have the same value in a column.
- 👉 It helps to prevent duplicate data, like the same email or phone number being used twice.
- ❗ However, NULL is allowed – but only once (because NULL is treated as “unknown”, and SQL allows one unknown value in a unique column).



# NOT NULL

```
CREATE TABLE customers
(
    id INT NOT NULL,
    name VARCHAR(100) NOT NULL
);
```

# DEFAULT Value

<b>id</b>	<b>name</b>	<b>email</b>	<b>created_at</b>
1	raju	raju@email.com	2025-06-22 18:06:59
2	sham	sham@email.com	2025-06-22 18:07:08
3	baburao	baburao@email.com	2025-06-22 18:07:19

```
Create table account (
    id INT identity(1,1) Primary key,
    name VARCHAR(100) Not null,
    email VARCHAR(100) Unique,
    created_at DATETIME DEFAULT GETDATE()
)
```

# IDENTITY

It is used to automatically generate unique numbers for a column when new rows are inserted into a table.

It works like auto-increment, usually for primary key columns.



What `IDENTITY(1,1)` means:

Value	Meaning
1	Start at 1
1	Increase by 1 each time

**TASK**

Creating New Table

# employees

emp_id	fname	lname	email	job_title	department	salary	hire_date	city
101	Aarav	Sharma	aarav.sharma@example.com	Director	Management	180000	2019-02-10	Mumbai
102	Diya	Patel	diya.patel@example.com	Lead Engineer	Tech	120000	2020-08-15	Bengaluru
103	Rohan	Mehra	rohan.mehra@example.com	Software Engineer	Tech	85000	2022-05-20	Bengaluru
104	Priya	Singh	priya.singh@example.com	HR Manager	Human Resources	95000	2019-11-05	Mumbai
105	Arjun	Kumar	arjun.kumar@example.com	Data Scientist	Tech	110000	2021-07-12	Hyderabad
106	Ananya	Gupta	ananya.gupta@example.com	Marketing Lead	Marketing	90000	2020-03-01	Delhi
107	Vikram	Reddy	vikram.reddy@example.com	Sales Executive	Sales	75000	2023-01-30	Mumbai
108	Sameera	Rao	sameera.rao@example.com	Software Engineer	Tech	88000	2023-06-25	Pune
109	Ishaan	Verma	ishaan.verma@example.com	Recruiter	Human Resources	65000	2022-09-01	Mumbai
110	Kavya	Joshi	kavya.joshi@example.com	Product Designer	Design	92000	2021-04-18	Bengaluru
111	Zain	Khan	zain.khan@example.com	Sales Manager	Sales	115000	2019-09-14	Delhi
112	Nisha	Desai	nisha.desai@example.com	Jr. Data Analyst	Tech	70000	2024-02-01	Hyderabad
113	Aditya	Nair	aditya.nair@example.com	Marketing Analyst	Marketing	68000	2022-10-10	Delhi
114	Fatima	Ali	fatima.ali@example.com	Sales Executive	Sales	78000	2022-11-22	Mumbai
115	Kabir	Shah	kabir.shah@example.com	DevOps Engineer	Tech	105000	2020-12-01	Pune

## **Requirement:**

- **emp\_id** set as primary key and its value should be auto-increment by 1 starting from 101.
- Email should be Unique
- Null value should not be allowed in
  - **fname, lname, email, job\_title**
- **Salary column** - default set to 30,000 if not provided
- **Hire\_date** - default set to today's date

```
CREATE TABLE employees (
    emp_id INT IDENTITY(101,1) PRIMARY KEY,
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    job_title VARCHAR(50) NOT NULL,
    department VARCHAR(50),
    salary DECIMAL(10,2) DEFAULT 30000.00,
    hire_date DATE NOT NULL DEFAULT CONVERT(date, GETDATE()),
    city VARCHAR(50)
);
```

```
INSERT INTO employees
(fname, lname, email, job_title, department, salary, hire_date, city)
VALUES
('Aarav', 'Sharma', 'aarav.sharma@example.com', 'Director', 'Management', 180000, '2019-02-10', 'Mumbai'),
('Diya', 'Patel', 'diya.patel@example.com', 'Lead Engineer', 'Tech', 120000, '2020-08-15', 'Bengaluru'),
('Rohan', 'Mehra', 'rohan.mehra@example.com', 'Software Engineer', 'Tech', 85000, '2022-05-20', 'Bengaluru'),
('Priya', 'Singh', 'priya.singh@example.com', 'HR Manager', 'Human Resources', 95000, '2019-11-05', 'Mumbai'),
('Arjun', 'Kumar', 'arjun.kumar@example.com', 'Data Scientist', 'Tech', 110000, '2021-07-12', 'Hyderabad'),
('Ananya', 'Gupta', 'ananya.gupta@example.com', 'Marketing Lead', 'Marketing', 90000, '2020-03-01', 'Delhi'),
('Vikram', 'Reddy', 'vikram.reddy@example.com', 'Sales Executive', 'Sales', 75000, '2023-01-30', 'Mumbai'),
('Sameera', 'Rao', 'sameera.rao@example.com', 'Software Engineer', 'Tech', 88000, '2023-06-25', 'Pune'),
('Ishaan', 'Verma', 'ishaan.verma@example.com', 'Recruiter', 'Human Resources', 65000, '2022-09-01', 'Mumbai'),
('Kavya', 'Joshi', 'kavya.joshi@example.com', 'Product Designer', 'Design', 92000, '2021-04-18', 'Bengaluru'),
('Zain', 'Khan', 'zain.khan@example.com', 'Sales Manager', 'Sales', 115000, '2019-09-14', 'Delhi'),
('Nisha', 'Desai', 'nisha.desai@example.com', 'Jr. Data Analyst', 'Tech', 70000, '2024-02-01', 'Hyderabad'),
('Aditya', 'Nair', 'aditya.nair@example.com', 'Marketing Analyst', 'Marketing', 68000, '2022-10-10', 'Delhi'),
('Fatima', 'Ali', 'fatima.ali@example.com', 'Sales Executive', 'Sales', 78000, '2022-11-22', 'Mumbai'),
('Kabir', 'Shah', 'kabir.shah@example.com', 'DevOps Engineer', 'Tech', 105000, '2020-12-01', 'Pune');
```

# Section - 4

# Getting the Data You Really Need

# Section - 4

- 4.1 - Where | Distinct | Order By | Like | TOP
- 4.2 - Logical Operators
- 4.3 - IN | NOT IN | BETWEEN

# Section - 4.1

# Clauses

A SQL clause is a part of a SQL statement that defines specific actions or conditions for querying, filtering, or organizing data in a database.

**Where | Distinct | Order By | Like | TOP**

Clause	What It Does	How to Use (Example)
WHERE	Filters rows based on a condition	<code>SELECT * FROM employees WHERE dept = 'IT';</code>
DISTINCT	Removes duplicate rows from the result set	<code>SELECT DISTINCT dept FROM employees;</code>
ORDER BY	Sorts results by one or more columns	<code>SELECT * FROM employees ORDER BY salary DESC;</code>
LIKE	Finds patterns in text (wildcards: %, _)	<code>SELECT * FROM employees WHERE fname LIKE 'A%';</code>
TOP	Limits number of rows returned (MSSQL only)	<code>SELECT TOP 5 * FROM employees;</code>

emp_id	fname	lname	email	job_title	department	salary	hire_date	city
101	Aarav	Sharma	aarav.sharma@example.com	Director	Management	180000	2019-02-10	Mumbai
102	Diya	Patel	diya.patel@example.com	Lead Engineer	Tech	120000	2020-08-15	Bengaluru
103	Rohan	Mehra	rohan.mehra@example.com	Software Engineer	Tech	85000	2022-05-20	Bengaluru
104	Priya	Singh	priya.singh@example.com	HR Manager	Human Resources	95000	2019-11-05	Mumbai
105	Arjun	Kumar	arjun.kumar@example.com	Data Scientist	Tech	110000	2021-07-12	Hyderabad
106	Ananya	Gupta	ananya.gupta@example.com	Marketing Lead	Marketing	90000	2020-03-01	Delhi
107	Vikram	Reddy	vikram.reddy@example.com	Sales Executive	Sales	75000	2023-01-30	Mumbai
108	Sameera	Rao	sameera.rao@example.com	Software Engineer	Tech	88000	2023-06-25	Pune
109	Ishaan	Verma	ishaan.verma@example.com	Recruiter	Human Resources	65000	2022-09-01	Mumbai
110	Kavya	Joshi	kavya.joshi@example.com	Product Designer	Design	92000	2021-04-18	Bengaluru
111	Zain	Khan	zain.khan@example.com	Sales Manager	Sales	115000	2019-09-14	Delhi
112	Nisha	Desai	nisha.desai@example.com	Jr. Data Analyst	Tech	70000	2024-02-01	Hyderabad
113	Aditya	Nair	aditya.nair@example.com	Marketing Analyst	Marketing	68000	2022-10-10	Delhi
114	Fatima	Ali	fatima.ali@example.com	Sales Executive	Sales	78000	2022-11-22	Mumbai
115	Kabir	Shah	kabir.shah@example.com	DevOps Engineer	Tech	105000	2020-12-01	Pune



## WHERE

- Find employees in the **IT** department
- Find employees with **salary above 50,000**
- Find employees hired **after 2020**
- Find employees whose **department is not HR**

# Relational Operators



# We have relational operators

Operators	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal to
!=	Not equal to

# DISTINCT

```
SELECT DISTINCT fname FROM employees;
```

# ORDER BY

```
SELECT * FROM employees ORDER BY fname;
```



## ORDER BY

- List employees by salary (high to low)
- Sort employees by hire date (earliest to latest)
- Order employees alphabetically by first name
- Order employees by department then by last name

# LIKE

```
Select * FROM employees  
WHERE dept LIKE "%Acc%";
```



## LIKE

- Find employees whose **first name starts with 'A'**
- Find employees whose **last name ends with 'a'**
- Find emails that contain **'gupta'**
- Find departments with names **starting with 'M'**

- Starts with 'A': **LIKE 'A%'**
- Starts with 'A' or B: **LIKE '[AB]%**'
- Not starts with A: **LIKE '[^A]%**'
- Ends with 'A': **LIKE '%A'**
- Contains 'A': **LIKE '%A%**'
- Second character is 'A': **LIKE '\_A%**'

 **TOP**

- Show **top 3 highest-paid employees**
- Display **top 5 most recently hired employees**
- Get the **top employee** from Marketing
- Show **first 2 alphabetically sorted employees**

# Exercise

DISTINCT, ORDER BY, LIKE and TOP

**1: Find Different type of departments in database?**

**2: Display records with High-low salary**

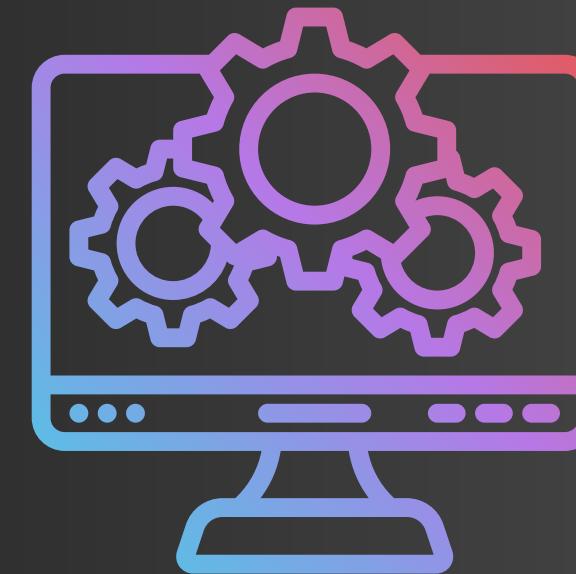
**3: How to see only top 3 records from a table?**

**4: Show records where first name start with letter 'A'**

**5: Show records where length of the lname is 4 characters**

# Section - 4.2

# Logical Operators



**AND**

**OR**

Condition 1

**AND**

Condition 2

When both the conditions are true

salary = 75000

AND

dept = Sales

Condition 1

OR

Condition 2

When either of the condition is true

**city = Mumbai**

**OR**

**dept = 'Tech'**

# Section - 4.3

**IN | NOT IN | BETWEEN**

# Find employees From following department

Tech  
Sales

Marketing



```
SELECT * FROM employees  
WHERE department = 'Marketing'  
    OR department = 'Sales'  
    OR department = 'Tech';
```

```
SELECT * FROM employees
WHERE department IN ('Marketing', 'Sales', 'Tech');
```

**BETWEEN**

# Find employees whose salary is more than 60000 and Less than 65000

>40000

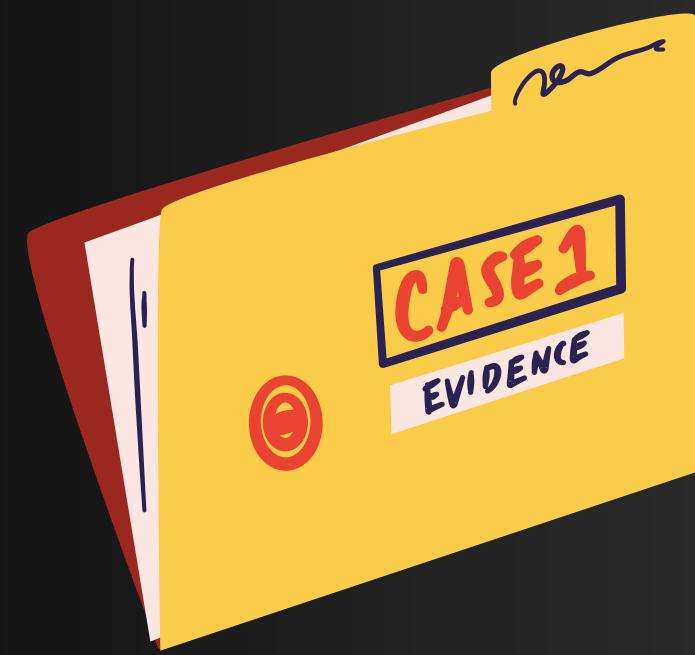
<65000



```
SELECT * FROM employees  
WHERE  
salary >=40000 AND salary <=65000;
```

```
SELECT * FROM employees  
WHERE  
salary BETWEEN 55000 AND 65000;
```

# Additional Topics



# CASE



fname	lname	salary	salary_band
Aarav	Sharma	180000	High Earner
Diya	Patel	120000	High Earner
Rohan	Mehra	85000	Medium Earner
Priya	Singh	95000	Medium Earner
Vikram	Reddy	75000	Standard Earner

```
SELECT
    fname,
    lname,
    salary,
    CASE
        WHEN salary > 100000 THEN 'High Earner'
        WHEN salary >= 80000 AND salary <= 100000 THEN 'Medium Earner'
        ELSE 'Standard Earner'
    END AS salary_band
FROM
    employees;
```

**Calculate a bonus amount. Sales and Marketing get a 10% bonus, Tech gets a 12% bonus, and everyone else gets a standard 5% bonus.**

fname	lname	department	salary	bonus_amount
Diya	Patel	Tech	120000	14400.00
Arjun	Kumar	Tech	110000	13200.00
Ananya	Gupta	Marketing	90000	9000.00
Zain	Khan	Sales	115000	11500.00
Priya	Singh	Human Resources	95000	4750.00

```
SELECT
    fname,
    lname,
    department,
    salary,
    CASE
        WHEN department IN ('Sales', 'Marketing') THEN salary * 0.10
        WHEN department = 'Tech' THEN salary * 0.12
        ELSE salary * 0.05
    END AS bonus_amount
FROM
    employees;
```

# Task

bonus		count
High		2
Mid		5
Low		3

```
SELECT
CASE
    WHEN salary > 55000 THEN 'High'
    WHEN salary BETWEEN 50000 AND 55000 THEN 'Mid'
    ELSE 'Low'
END AS bonus,
COUNT(emp_id)
FROM
employees
GROUP BY
CASE
    WHEN salary > 55000 THEN 'High'
    WHEN salary BETWEEN 50000 AND 55000 THEN 'Mid'
    ELSE 'Low'
END;
```

**IS NULL**

**IS NULL**

```
SELECT * FROM employees
WHERE fname IS NULL;
```

NOT LIKE

**IS NULL**

```
SELECT * FROM employees
WHERE fname NOT LIKE 'A%';
```

# SECTION - 5

- 5.1 Aggregate Functions
- 5.2 Group By

## Section - 5.1

# Aggregate functions

- How to find total no. of employees?
- Employee with Max or Min salary
- Average salary of employees
- Sum/total salary paid

- COUNT
- SUM
- AVG
- MIN
- MAX

# COUNT

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

# MAX & MIN

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

# SUM & AVG

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

```
SELECT emp_id, fname, salary FROM employees
WHERE
salary = (SELECT MAX(salary) FROM employees);
```

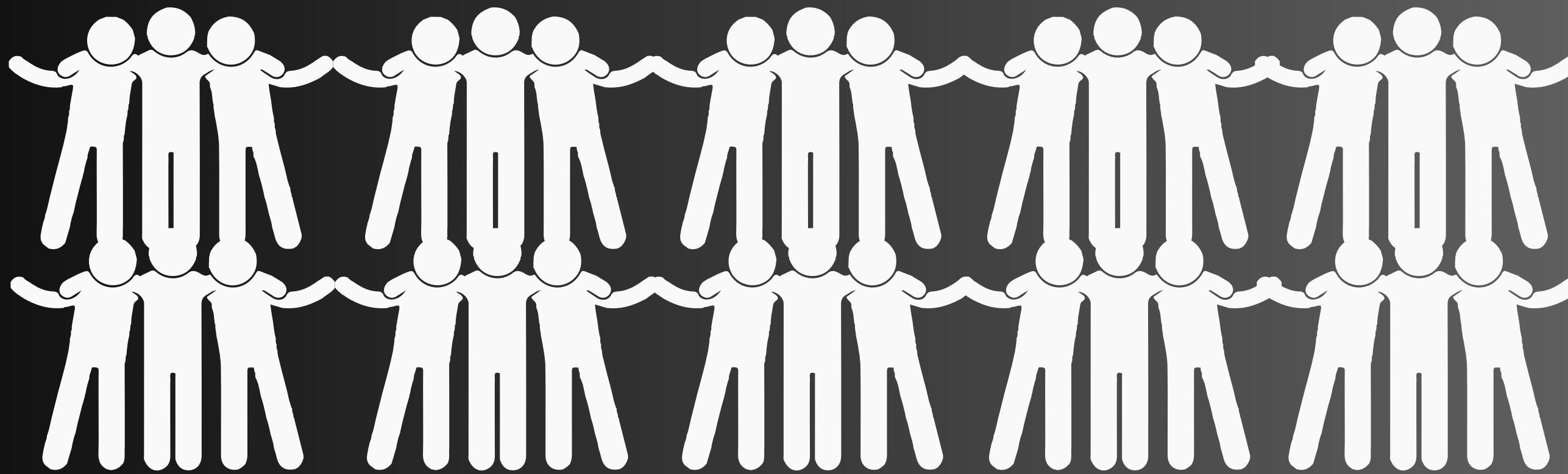
Section - 5.2

**GROUP BY**



# No. of employees in each department

dept	count
Marketing	2
Finance	2
IT	4
HR	2



HR

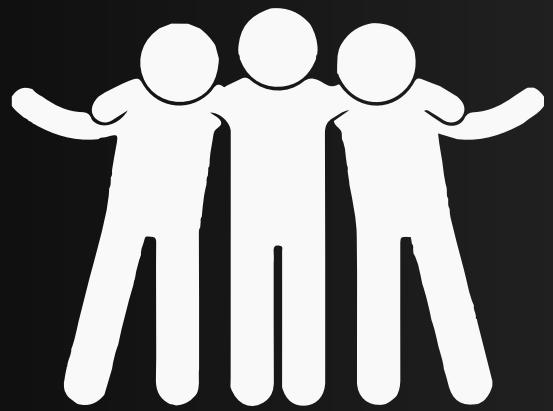
IT

Finance

Deposit

Marketing

BANK



HR



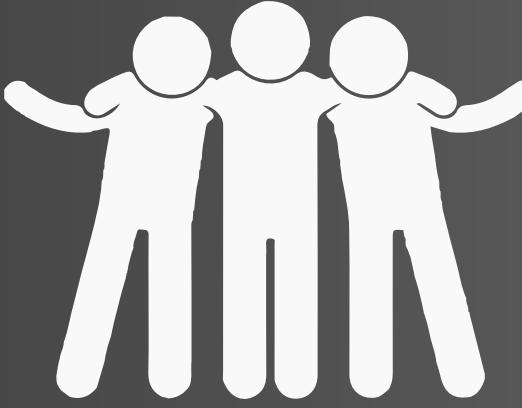
Finance



Marketing



IT



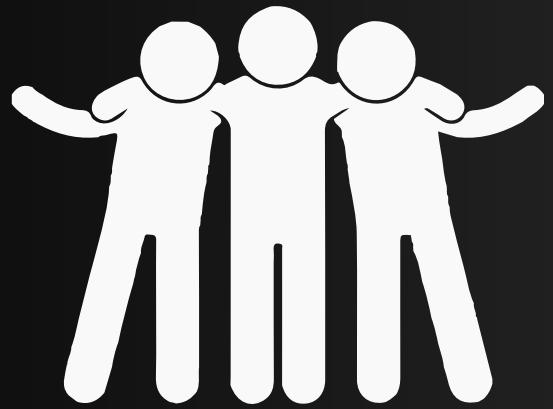
Deposit

```
SELECT dept FROM employees GROUP BY dept;
```

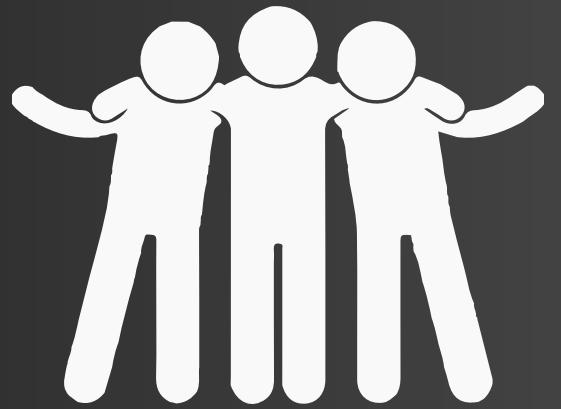
```
SELECT dept, COUNT(fname) FROM employees GROUP  
BY dept;
```

- Find number of employees in each department
- Find number of employees in each city
- Find average salary in each department

# Multi-Level Grouping



HR



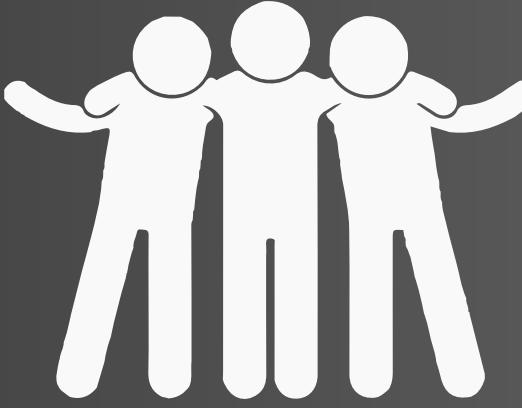
Finance



Marketing



Tech



Deposit



Tech

Mumbai 2

Pune 3

Hyderabad 3

Sales	Delhi	1
Sales	Mumbai	2
Tech	Bengaluru	2
Tech	Hyderabad	2
Tech	Pune	2

# HAVING clause

- Find Departments with More Than 2 Employees
- Find Job Titles with an Average Salary Above 90000
- Find department with Total Salary Above 300000

```
SELECT  
    department,  
    COUNT(emp_id) AS number_of_employees  
FROM  
    employees  
GROUP BY  
    department  
HAVING  
    COUNT(emp_id) > 2;
```

# GROUP BY ROLLUP

**GROUP BY ROLLUP** is an extension of the  
GROUP BY clause that generates **subtotals** and  
a **grand total** for a set of columns.

# Use case

- Employee Headcount by City and Department.
  - You want a report showing the number of employees for each city within each department, a subtotal for each department, and a grand total for the entire company.

```
SELECT  
    city,  
    department,  
    COUNT(emp_id) AS number_of_employees  
FROM  
    employees  
GROUP BY  
    ROLLUP(city, department)  
ORDER BY  
    city, department;
```

# Exercise

COUNT, GROUP BY, MIN, MAX and SUM and AVG

**1: Find Total no. of employees in database?**

**2: Find no. of employees in each department.**

**3: Find lowest salary paying**

**4: Find highest salary paying**

**5: Find total salary paying in Loan department?**

**6: Average salary paying in each department**

# Section 5.3

## SUB-QUERIES

## Usecases:

- Find Employees Earning More Than the Company Average
- Find Employees Who Work in the Same City as a Specific Person (ex: aarav sharma)
- Find the Highest-Paid Employee name.
- Find the Highest-Paid Employee in Each Department

A **SubQuery** (also called an inner query or nested query) is a query inside another query.

- ◆ The subquery runs first and gives a result.
- ◆ The main query (outer query) then uses that result.

```
SELECT emp_id, fname, lname, salary  
FROM employees  
WHERE salary > (  
    SELECT AVG(salary)  
    FROM employees  
) ;
```

# Types of SubQueries

Type	Description	Usually used in	Returns
Single-row	Returns one value	WHERE , HAVING	One row, one column
Multiple-row	Returns many rows (one column)	IN , ANY , ALL	Multiple rows
Correlated	Depends on outer query	WHERE , SELECT	Varies
Inline View	Inside FROM , acts like temporary table	FROM	Table-like result set

# Find employees who work in departments with at least one person in Mumbai

👉 Multi-row subquery.

```
SELECT fname, lname, department  
FROM employees  
WHERE department IN (  
    SELECT department  
    FROM employees  
    WHERE city = 'Mumbai'  
);
```

# Find employees with the highest salary in each department

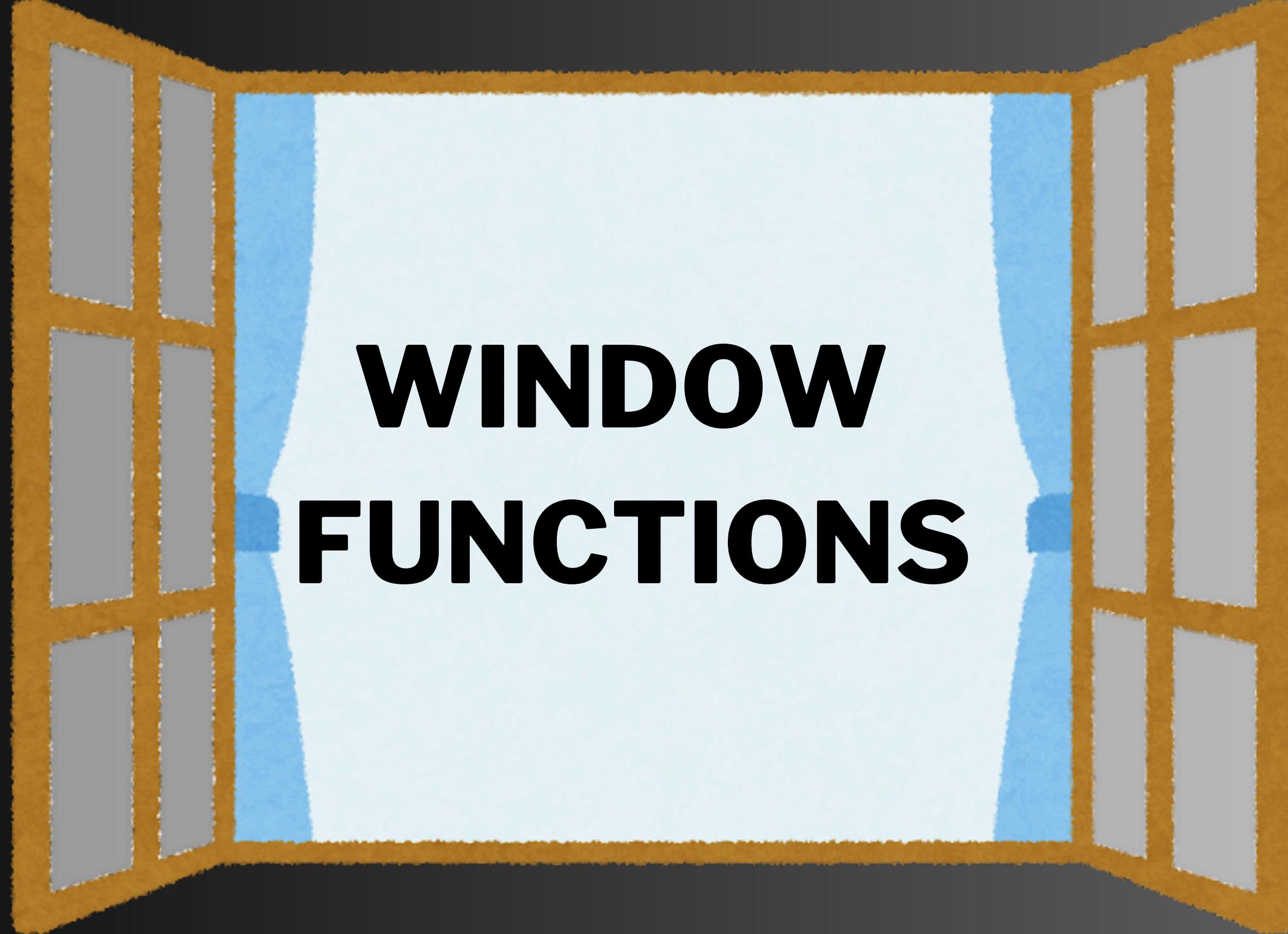
👉 Correlated subquery.

```
SELECT emp_id, fname, lname, department, salary  
FROM employees e1  
WHERE salary = (  
    SELECT MAX(salary)  
    FROM employees e2  
    WHERE e2.department = e1.department  
);
```

If we want to find departments whose average salary is above 90,000.

👉 Inline View subquery.

```
SELECT department, avg_salary
FROM (
    SELECT department, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY department
) AS dept_avg
WHERE avg_salary > 90000;
```



# **WINDOW FUNCTIONS**

**Window functions**, also known as analytic functions allow you to perform calculations across a set of rows related to the current row.

Defined by an **OVER()** clause.

# Compare each employee's salary to total salary

fname	department	salary	total_salary	pct_of_total
Aarav	Management	180000	1515000	11.88%
Diya	Tech	120000	1515000	7.92%

**Compare each employee's salary to total salary in each department**

fname	department	salary	tot_dept_sal
Nisha	Tech	70000.00	578000.00
Sameera	Tech	88000.00	578000.00
Diya	Tech	120000.00	578000.00
Rohan	Tech	85000.00	578000.00
Arjun	Tech	110000.00	578000.00
Kabir	Tech	105000.00	578000.00
Fatima	Sales	78000.00	268000.00
Vikram	Sales	75000.00	268000.00
Zain	Sales	115000.00	268000.00
Ananya	Marketing	90000.00	158000.00
Aditya	Marketing	68000.00	158000.00

`OVER (PARTITION BY  
col)`

The window is the **group** of rows with the same `col` value. (e.g., average salary *per department*).

`OVER (ORDER BY col)`

The window is all rows from the **start up to the current row**, ordered by `col`. (e.g., a *running total* for the whole table).

`OVER (PARTITION BY col1  
ORDER BY col2)`

The window is all rows in the group (`col1`) from its **start up to the current row**, ordered by `col2`. (e.g., a *running total per department*).

# Add row number for each row in table.

row_no	fname	department	salary
1	Aarav	Management	180000.00
2	Aarav	Management	30000.00
3	Aditya	Marketing	68000.00
4	Alex	Trainee	10000.00
5	Ananya	Marketing	90000.00
6	Arjun	Tech	110000.00
7	Diya	Tech	120000.00
8	Fatima	Sales	78000.00

```
select fname, dept, salary,  
    ROW_NUMBER()  
    OVER(PARTITION BY dept ORDER BY salary DESC)  
        AS row_num  
from employees;
```

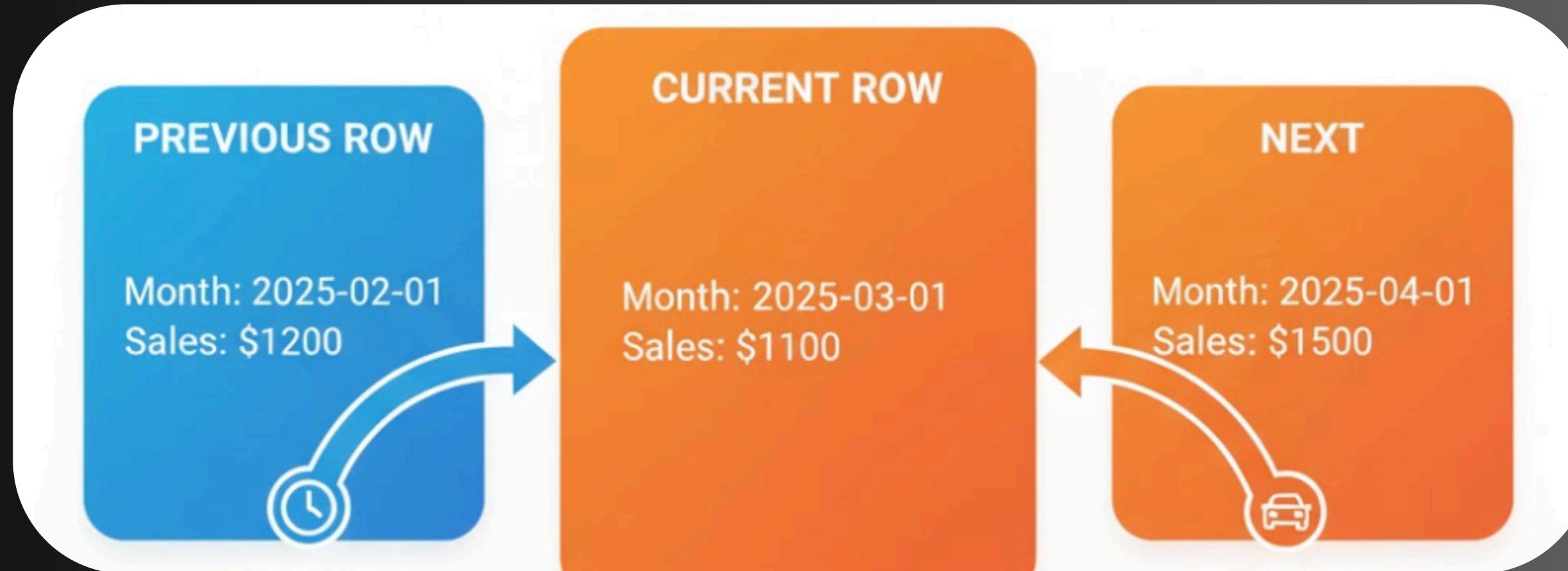
- **ROW\_NUMBER()**
- **RANK()**
- **DENSE\_RANK()**
- **LAG()**
- **LEAD()**

# Rank all employees based on salary from High to Low

fname	department	salary	rank
Aarav	Management	180000.00	1
Diya	Tech	120000.00	2
Zain	Sales	115000.00	3
Arjun	Tech	110000.00	4
Kabir	Tech	105000.00	5
Priya	Human Resources	95000.00	6
Kavya	Design	92000.00	7

# LAG

# LEAD



department	salary
Management	180000.00
Tech	120000.00
Sales	115000.00
Tech	110000.00
Tech	105000.00
Human Resources	95000.00
Design	92000.00
Marketing	90000.00

A diagram illustrating the context of the current row in a table. Three arrows point from the right side of the table towards the text labels:

- A teal curved arrow points upwards to the word **Lag**.
- An orange straight arrow points horizontally to the words **Current Row**.
- A teal curved arrow points downwards to the word **Lead**.

# Compare an Employee's Salary to the Previous Hire

fname	department	hire_date	salary	previous_hire_salary
Diya	Tech	2020-08-15	120000	0
Kabir	Tech	2020-12-01	105000	120000
Arjun	Tech	2021-07-12	110000	105000

## Usecases:

- Rank Employees Within Each Department by Salary.
- Calculate a Running Total of Salary Budget in each department.
- Compare an Employee's Salary to the Previous Hire

# Rank Employees Within Each Department by Salary.

fname	department	salary	salary_rank
Priya	Human Resources	95000	1
Ishaan	Human Resources	65000	2
Ananya	Marketing	90000	1
Aditya	Marketing	68000	2
Diya	Tech	120000	1
Arjun	Tech	110000	2

## Calculate a Running Total of Salary Budget in each department.

fname	department	hire_date	salary	running_salary_total
Zain	Sales	2019-09-14	115000	115000
Fatima	Sales	2022-11-22	78000	193000
Vikram	Sales	2023-01-30	75000	268000

# Benefits of Window Functions

- **Advanced Analytics:** They enable complex calculations like running totals, moving averages, rank calculations, and cumulative distributions.
- **Non-Aggregating:** Unlike aggregate functions, window functions do not collapse rows. This means you can calculate aggregates while retaining individual row details.
- **Flexibility:** They can be used in various clauses of SQL, such as SELECT, ORDER BY, and HAVING, providing a lot of flexibility in writing queries.

# **ROWS BETWEEN**

**ROWS BETWEEN** is a clause in a window function which tells that -

For the row I'm currently on, calculate the result using only this specific group of surrounding rows.

# Running Total of Salary

fname	department	salary	running_total
Aarav	Management	180000.00	360000.00
Alex	Trainee	180000.00	360000.00
Diya	Tech	120000.00	480000.00
Zain	Sales	115000.00	595000.00
Arjun	Tech	110000.00	705000.00
Kabir	Tech	105000.00	810000.00
Priya	Human Resources	95000.00	905000.00

Output

running_total
180000.00
360000.00
480000.00
595000.00
705000.00
810000.00
905000.00

Expected

# Find Running Total of Salary

```
SELECT
    name,
    hire_date,
    salary,
    SUM(salary) OVER (
        ORDER BY hire_date
        ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
    ) AS running_total_salary
FROM
    employees;
```

fname	department	salary	running_total
Aarav	Management	180000.00	180000.00
Diya	Tech	120000.00	300000.00
Rohan	Tech	85000.00	385000.00
Priya	Human Resources	95000.00	480000.00
Arjun	Tech	110000.00	590000.00
Ananya	Marketing	90000.00	680000.00



Current Row

**480000 = Current ROW + Sum of all preceding row  
 $= 95k + (85K + 120K + 180K)$**

# Calculate 3-Rows Moving Average

Calculate the average salary of the current employee, the one hired just before, and the one hired just after.

```
SELECT
    name,
    hire_date,
    salary,
    AVG(salary) OVER (
        ORDER BY hire_date
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS 3_month_moving_avg
FROM
    employees;
```

```
AGGREGATE_FUNCTION() OVER (
    PARTITION BY ...
    ORDER BY ...
    ROWS BETWEEN <start_boundary> AND <end_boundary>
)
```

The power of `ROWS BETWEEN` comes from defining the start and end of your frame. The most common boundaries are:

- `CURRENT ROW` : The row being processed right now. ↗
- `UNBOUNDED PRECEDING` : All rows *before* the current row in the partition.
- `UNBOUNDED FOLLOWING` : All rows *after* the current row in the partition.
- `n PRECEDING` : A specific number (`n`) of rows *before* the current row. ↗
- `n FOLLOWING` : A specific number (`n`) of rows *after* the current row.

- FIRST\_VALUE
- LAST\_VALUE
- NTILE

```
SELECT fname, department, salary,  
       LAST_VALUE(fname) OVER(  
           PARTITION BY department  
           ORDER BY fname  
           ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING  
       ) AS last_in_dept  
FROM employees;
```

Divide all employees into four groups (quartiles) based on their salary, from highest to lowest.

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

Find the top, middle, and bottom earners within each department.

```
SELECT
    name,
    department,
    salary,
    NTILE(3) OVER (PARTITION BY department ORDER BY salary DESC) AS dept_salary_tier
FROM
    employees;
```

# CTE

## Common Table Expression

**CTE (Common Table Expression)** is a temporary result set that you can define within a query to simplify complex SQL statements.

```
WITH cte_name (optional_column_list) AS (
    -- CTE query definition
    SELECT ...
)

-- Main query referencing the CTE
SELECT ...
FROM cte_name
WHERE ...;
```

## Use Cases - 1

- Finding Employees Who Earn More Than Their Department's Average Salary

emp_id	fname	department	salary	dept_avg
104	Priya	Human Resources	95000.00	80000.00
101	Aarav	Management	180000.00	80000.00
106	Ananya	Marketing	90000.00	79000.00
111	Zain	Sales	115000.00	89333.33
102	Diya	Tech	120000.00	96333.33
105	Arjun	Tech	110000.00	96333.33

```
WITH avgsal AS (
    SELECT
        department,
        AVG(salary) AS dept_avg
    FROM employees
    GROUP BY department
)
SELECT
    emp_id, fname, e.department, salary, a.dept_avg
    FROM employees e JOIN avgsal a
    ON e.department = a.department
    WHERE salary > a.dept_avg
```

## Use Cases - 2

- We want to find the highest-paid employee in each department.

emp_id	fname	department	salary	dept_max
123	Alex	Trainee	180000.00	180000.00
102	Diya	Tech	120000.00	120000.00
111	Zain	Sales	115000.00	115000.00
106	Ananya	Marketing	90000.00	90000.00
101	Aarav	Management	180000.00	180000.00
104	Priya	Human Resources	95000.00	95000.00
110	Kavya	Design	92000.00	92000.00

```
WITH maxsal AS (
    SELECT
        department,
        MAX(salary) AS dept_max
    FROM employees
    GROUP BY department
)
SELECT
    emp_id, fname, e.department, salary, m.dept_max
    FROM employees e JOIN maxsal m
    ON e.department = m.department
    WHERE salary = m.dept_max
```

## Points:

- Once CTE has been created it can only be used once. It will not be persisted.

# Section - 6

## String Functions

- CONCAT, CONCAT\_WS
- SUBSTRING
- LEFT, RIGHT
- LEN
- UPPER, LOWER
- TRIM, LTRIM, RTRIM
- REPLACE
- CHARINDEX

# CONCAT

`CONCAT(first_col, sec_col)`

`CONCAT(first_word, sec_word, ...)`

# CONCAT\_WS

**CONCAT\_WS**( '-' , fname , lname )

# SUBSTRING

```
SELECT SUBSTRING('Hey Buddy', 1, 4);
```

# SUBSTRING

```
SELECT SUBSTRING( 'Hey Buddy' , 1, 4);  
           1   4
```

Result: Hey

# REPLACE

Hey Buddy



Hello Buddy

**REPLACE(str, from\_str, to\_str)**

**REPLACE('Hey Buddy', 'Hey', 'Hello')**

# REVERSE

```
SELECT REVERSE('Hello World');
```

# LENGTH

Select LEN('Hello World');

# UPPER & LOWER

```
SELECT UPPER('Hello World');  
SELECT LOWER('Hello World');
```

# Other Functions

```
SELECT LEFT('Abcdefghij', 3);  
SELECT RIGHT('Abcdefghij', 4);
```

```
SELECT TRIM(' Alright! ' );  
SELECT CHARINDEX('OM', 'ThOMAS');
```

# Exercise

**Task 1:**

**101:Aarav:Sharma:Management**

**Task2:**

**102:Diya Patel:Tech:120000**

**Task3**

**104: Priya: HUMAN RESOURCES**

**Task4**

**H104 Priya**

**M101 Aarav**

# **DATE Functions**

- **GETDATE()**
- **DATEADD(interval, number, date)**
- **DATEDIFF(interval, start\_date, end\_date)**
- **DATEPART(interval, date) / YEAR(date), MONTH(date), DAY(date)**
- **FORMAT(date, format\_string)**
  - formats like '**MM/dd/yyyy**' or '**DD-MMM-yyyy**'

- Find out each employee's 5-year anniversary date.
- Find employees hired in March.
- Show the year, month, and day each employee was hired, separately.
- Display the hire date in a standard US format (MM/dd/yyyy)

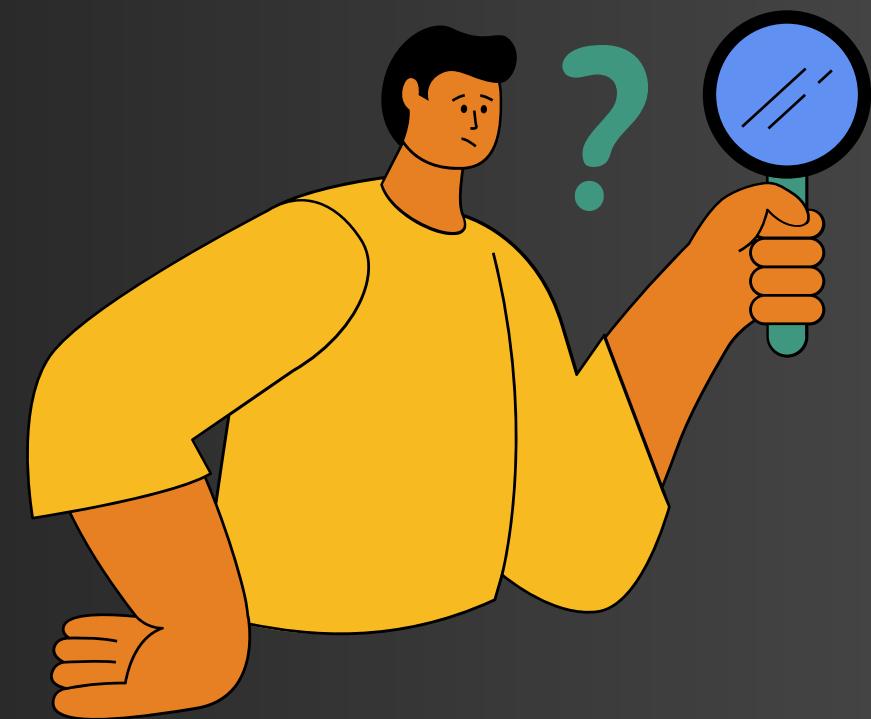


# Section - 7

# **ALTERING**

## **Tables**

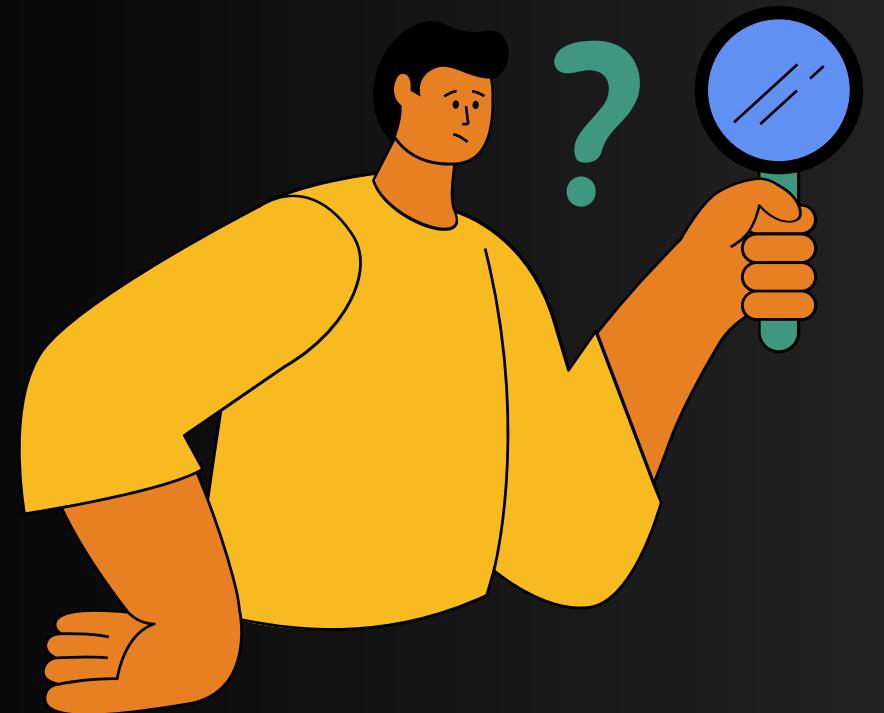
# How to add or remove a column?



```
ALTER TABLE employees  
ADD phone VARCHAR(15);
```

```
ALTER TABLE employees  
DROP COLUMN phone;
```

# How to modify a column?



Ex: Changing datatype

# How to change datatype of a column?

Ex: VARCHAR limit

**ALTER TABLE employees**

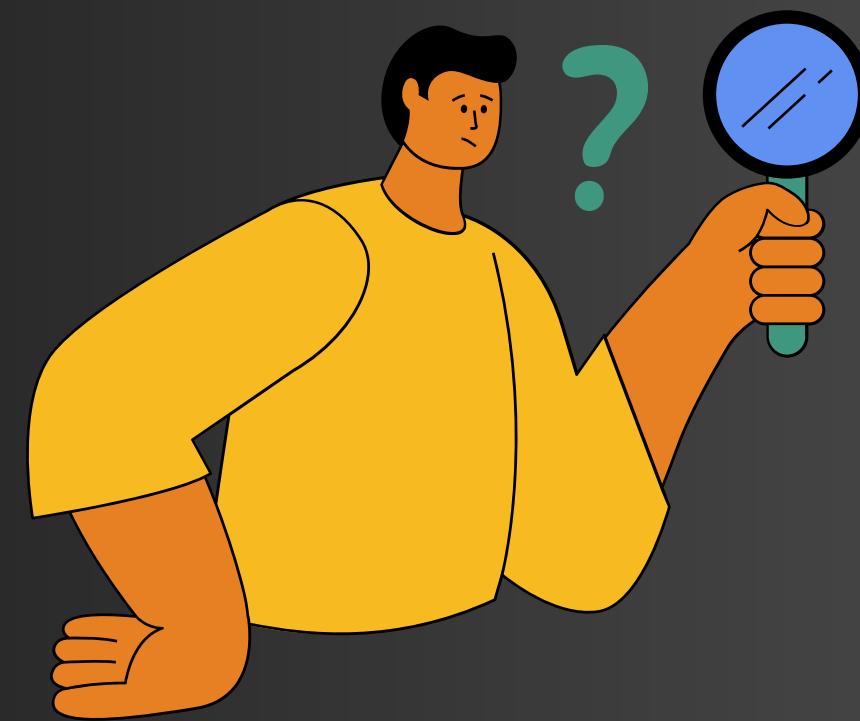
**ALTER COLUMN lname VARCHAR(100) NOT NULL;**

# How to set NOT NULL to a column?

ALTER TABLE employees

ALTER COLUMN email VARCHAR(100) NOT NULL;

# How to rename a column or table name?



# How to rename a column?

```
EXEC sp_rename  
'employees.fname', 'first_name', 'COLUMN';
```

# How to rename a table?

```
EXEC sp_rename  
'employees', 'staff';
```

# ADD/DROP Constraint

# Constraint

A constraint in SQL is a rule applied to a column or table to control the type of data that can be stored in it, ensuring accuracy, validity, and integrity of the data.

# How to set Default Value to a column?

```
ALTER TABLE employees  
ADD CONSTRAINT default_dept DEFAULT 'Trainee'  
FOR department;
```

# CHECK CONSTRAINT



We want to make sure  
salary of an employee is  
positive..

```
CREATE TABLE emp(  
    name varchar(50),  
    salary DECIMAL(10,2) CHECK (salary>0)  
)
```

# NAMED CONSTRAINT

```
CREATE TABLE contacts (
    name VARCHAR(50),
    salary DECIMAL(10,2),
    CONSTRAINT chk_emp_positive_salary CHECK (salary>0)
);
```

```
ALTER TABLE table_name  
ADD CONSTRAINT constraint_name  
CHECK (condition);
```

```
ALTER TABLE contacts
DROP CONSTRAINT mob_no_less_than_10digits;
```

# SECTION - 8

# RELATIONSHIP

A database relationship is a connection  
between two or more tables, established using a  
**primary key** and a **foreign key**.

# Employees

emp_id	fname	lname	email	job_title	department	salary	hire_date	city
101	Aarav	Sharma	aarav.sharma@example.com	Director	Management	180000	2019-02-10	Mumbai
102	Diya	Patel	diya.patel@example.com	Lead Engineer	Tech	120000	2020-08-15	Bengaluru
103	Rohan	Mehra	rohan.mehra@example.com	Software Engineer	Tech	85000	2022-05-20	Bengaluru
104	Priya	Singh	priya.singh@example.com	HR Manager	Human Resources	95000	2019-11-05	Mumbai
105	Arjun	Kumar	arjun.kumar@example.com	Data Scientist	Tech	110000	2021-07-12	Hyderabad
106	Ananya	Gupta	ananya.gupta@example.com	Marketing Lead	Marketing	90000	2020-03-01	Delhi
107	Vikram	Reddy	vikram.reddy@example.com	Sales Executive	Sales	75000	2023-01-30	Mumbai
108	Sameera	Rao	sameera.rao@example.com	Software Engineer	Tech	88000	2023-06-25	Pune
109	Ishaan	Verma	ishaan.verma@example.com	Recruiter	Human Resources	65000	2022-09-01	Mumbai
110	Kavya	Joshi	kavya.joshi@example.com	Product Designer	Design	92000	2021-04-18	Bengaluru
111	Zain	Khan	zain.khan@example.com	Sales Manager	Sales	115000	2019-09-14	Delhi
112	Nisha	Desai	nisha.desai@example.com	Jr. Data Analyst	Tech	70000	2024-02-01	Hyderabad
113	Aditya	Nair	aditya.nair@example.com	Marketing Analyst	Marketing	68000	2022-10-10	Delhi
114	Fatima	Ali	fatima.ali@example.com	Sales Executive	Sales	78000	2022-11-22	Mumbai
115	Kabir	Shah	kabir.shah@example.com	DevOps Engineer	Tech	105000	2020-12-01	Pune

**Let's understand problems with our  
current database**

emp_id	fname	department	job_title	city
101	Aarav	Management	Director	Mumbai
102	Diya	Tech	Lead Engineer	Bengaluru
103	Rohan	Tech	Software Engineer	Bengaluru
104	Priya	Human Resources	HR Manager	Mumbai
105	Arjun	Tech	Data Scientist	Hyderabad
106	Ananya	Marketing	Marketing Lead	Delhi
107	Vikram	Sales	Sales Executive	Mumbai
108	Sameera	Tech	Software Engineer	Pune
109	Ishaan	Human Resources	Recruiter	Mumbai
110	Kavya	Design	Product Designer	Bengaluru
111	Zain	Sales	Sales Manager	Delhi

- 1. Data Redundancy (Duplication)**
- 2. What if I want to add a Finance department & its related details?**
- 3. What if Kayva leaves the company which was only one in Design department?**
- 4. What if there is a typo in department column?**

# department

	department_id (PK)	department_name
1		Management
2		Tech
3		Human Resources
4		Marketing
5		Sales
6		Design

# employees

	emp_id	fname	department_id (FK)	job_title
	101	Aarav	1	Director
	102	Diya	2	Lead Engineer
	103	Rohan	2	Software Engineer
	104	Priya	3	HR Manager
	105	Arjun	2	Data Scientist
	106	Ananya	4	Marketing Lead
	107	Vikram	5	Sales Executive

# Foreign Key

A **foreign key** is a column in one table that links to the **primary key** of another table.

# department

department_id (PK)	department_name
1	Management
2	Tech
3	Human Resources
4	Marketing
5	Sales
6	Design

# employees

emp_id	fname	department_id (FK)	job_title
101	Aarav	1	Director
102	Diya	2	Lead Engineer
103	Rohan	2	Software Engineer
104	Priya	3	HR Manager
105	Arjun	2	Data Scientist
106	Ananya	4	Marketing Lead
107	Vikram	5	Sales Executive

Primary Key

Foreign Key

**Salary**

**Attendance**

**Employees**

**requests**

**offices**

**task**

# Types of Relationship

- One to One
- One to Many
- Many to Many

1 : 1

## Employees



emp_id (PK)	fname	lname
101	Aarav	Sharma
102	Diya	Patel

## Employee Bank Details

emp_id (PK, FK)	bank_account_no	emergency_contact
101	xxxx-xxxx-1234'	Rhea Sharma'
102	xxxx-xxxx-5678'	Anil Patel'

**1 : MANY**

## department

department_id (PK)	department_name
1	Management
2	Tech
3	Sales

## employees



emp_id (PK)	fname	Iname	department_id (FK)
101	Aarav	Sharma	1
102	Diya	Patel	2
103	Rohan	Mehra	2
105	Arjun	Kumar	1

# Many : Many



**Books**



**Authors**



**Book A**



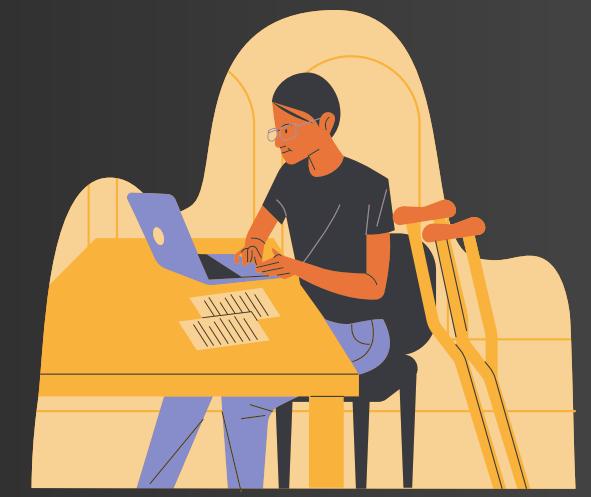
**Author A**



**Author B**



**Book A**



**Author A**



**Author B**



**Book B**



**Book C**



**Book D**

# Projects



employee



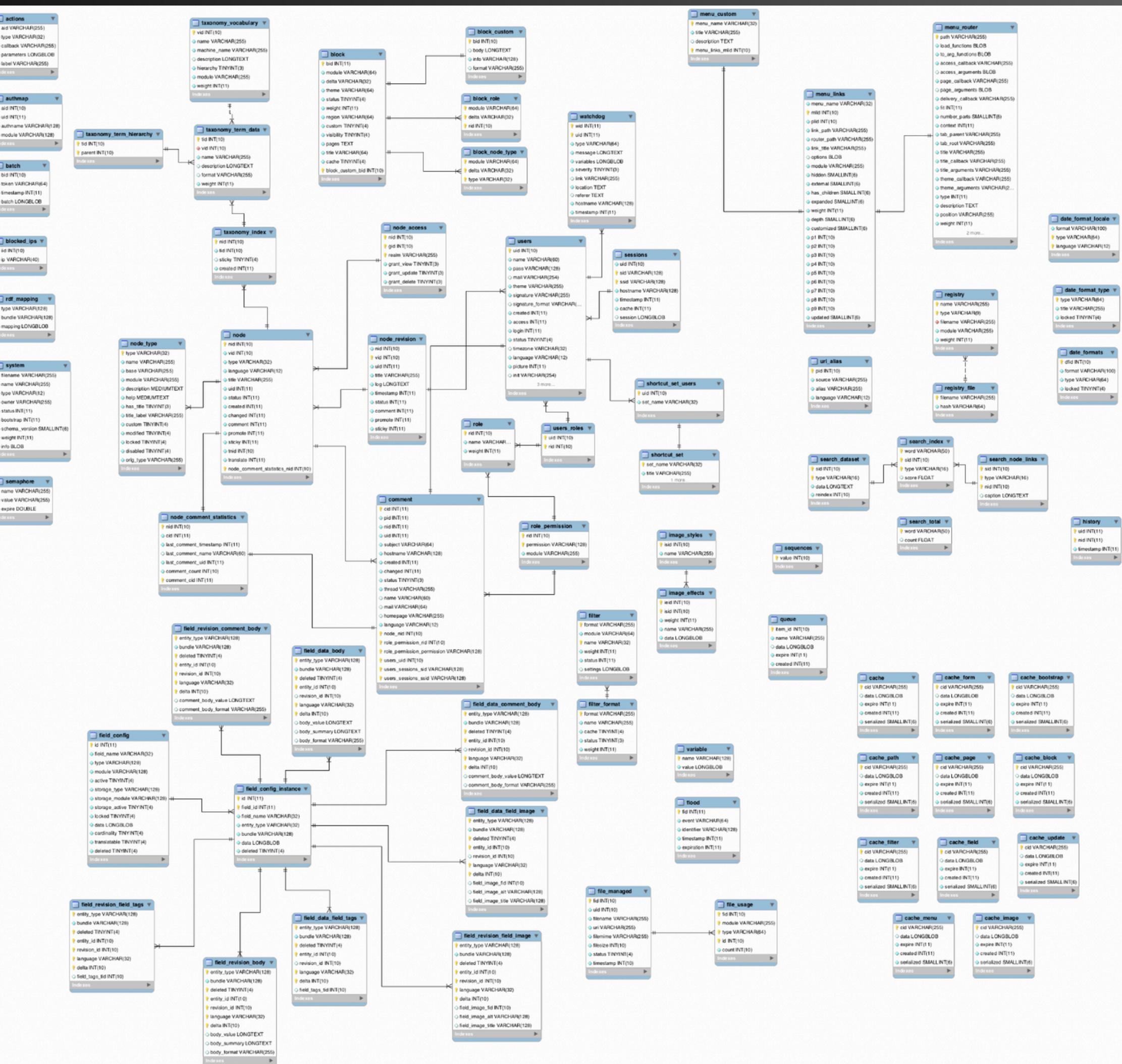
# Projects



employee



employees



# PRACTICAL

## 1: Many





**Suppose we need to store the following data**

- customer name
- customer email
- order date
- order price

customer_id	customer_name	email	order_id	order_date	total_amount
100	Raju	raju@example.com	500	2025-09-15	1500.00
101	Sham	sham@example.com	501	2025-09-28	800.00
100	Raju	raju@example.com	502	2025-10-05	2200.00
102	Baburao	baburao@example.com	503	2025-10-12	500.00
101	Sham	sham@example.com	504	2025-10-17	1200.00

## **Customers**

**cust\_id**

**cust\_name**

**cust\_email**

## **Orders**

**order\_id**

**order\_date**

**order\_amount**

## Customers

**cust\_id**  
**cust\_name**  
**cust\_email**

## Orders

**order\_id**  
**order\_date**  
**order\_amount**  
**cust\_id**

## Customers

customer_id (PK)	customer_name	email
101	Raju	raju@example.com
102	Sham	sham@example.com
103	Baburao	baburao@example.com

## Orders

order_id (PK)	order_date	total_amount	customer_id (FK)
5001	2025-09-15	1500.00	101
5002	2025-09-28	800.00	102
5003	2025-10-05	2200.00	101
5004	2025-10-12	500.00	103

Let's work practically  
with Foreign Key..

1-Many

Customers

```
CREATE TABLE Customers (
    customer_id INT IDENTITY(100,1) PRIMARY KEY,
    customer_name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE
);
```

Orders

```
CREATE TABLE Orders (
    order_id INT IDENTITY(500,1) PRIMARY KEY,
    order_date DATE NOT NULL,
    total_amount DECIMAL(10, 2),
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id)
);
```

```
INSERT INTO Customers (customer_name, email)
VALUES
('Raju', 'raju@example.com'),
('Sham', 'sham@example.com'),
('Baburao', 'baburao@example.com');
```

---

```
INSERT INTO Orders (order_date, total_amount, customer_id)
VALUES
('2025-09-15', 1500.00, 100), -- This links to Raju (customer_id 100)
('2025-09-28', 800.00, 101), -- This links to Sham (customer_id 101)
('2025-10-05', 2200.00, 100), -- This links to Raju (customer_id 100)
('2025-10-12', 500.00, 102), -- This links to Baburao (customer_id 102)
('2025-10-17', 1200.00, 101); -- New order for Sham (customer_id 101)
```

# JOINS

**JOIN** operation is used to combine rows from two or more tables based on a related column between them.

# Types of Join

- Cross Join
- Inner Join
- Left Join
- Right Join
- Full Join

# Cross Join

**Every row from one table is combined with  
every row from another table.**

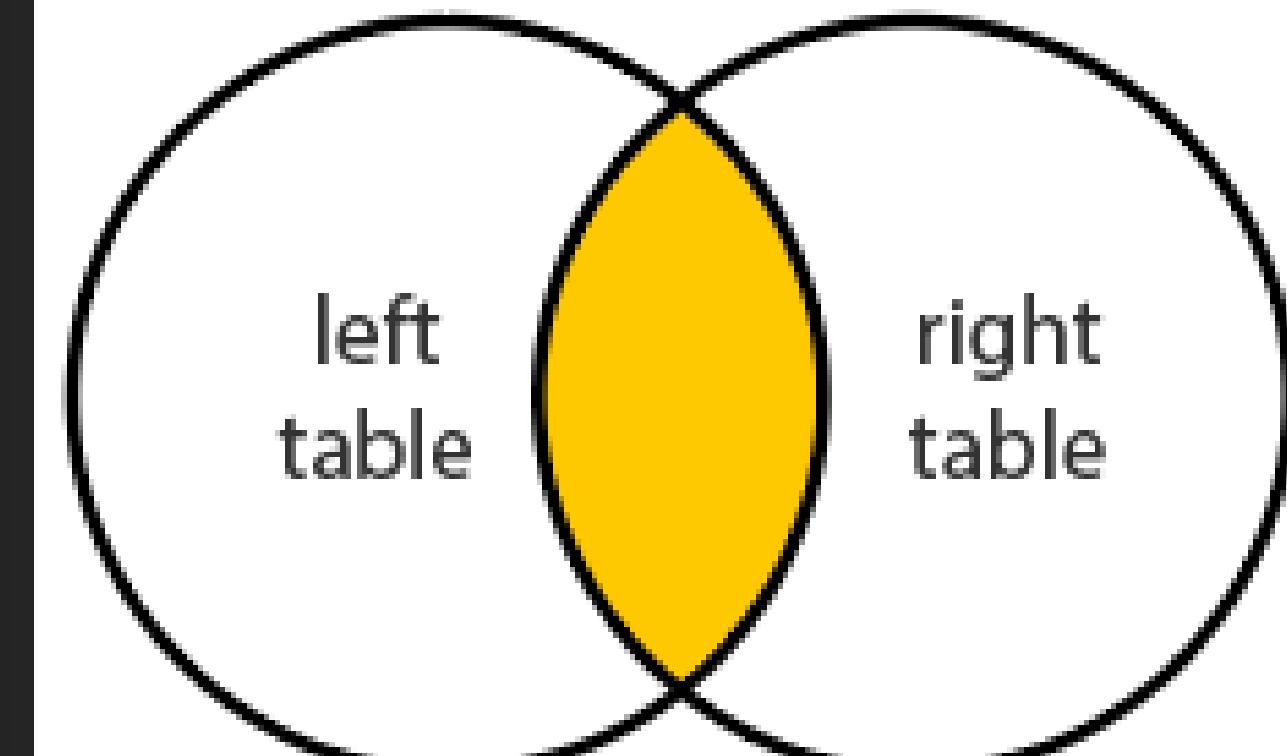
```
SELECT ...  
FROM table1 t1  
CROSS JOIN table2 t2;
```

# Inner Join

Returns only the rows where there is a match between the specified columns in both the left (or first) and right (or second) tables.

```
SELECT ...
FROM table1 t1
INNER JOIN table2 t2
    ON t1.common_column = t2.common_column;
```

## INNER JOIN



# Inner Join with Group By

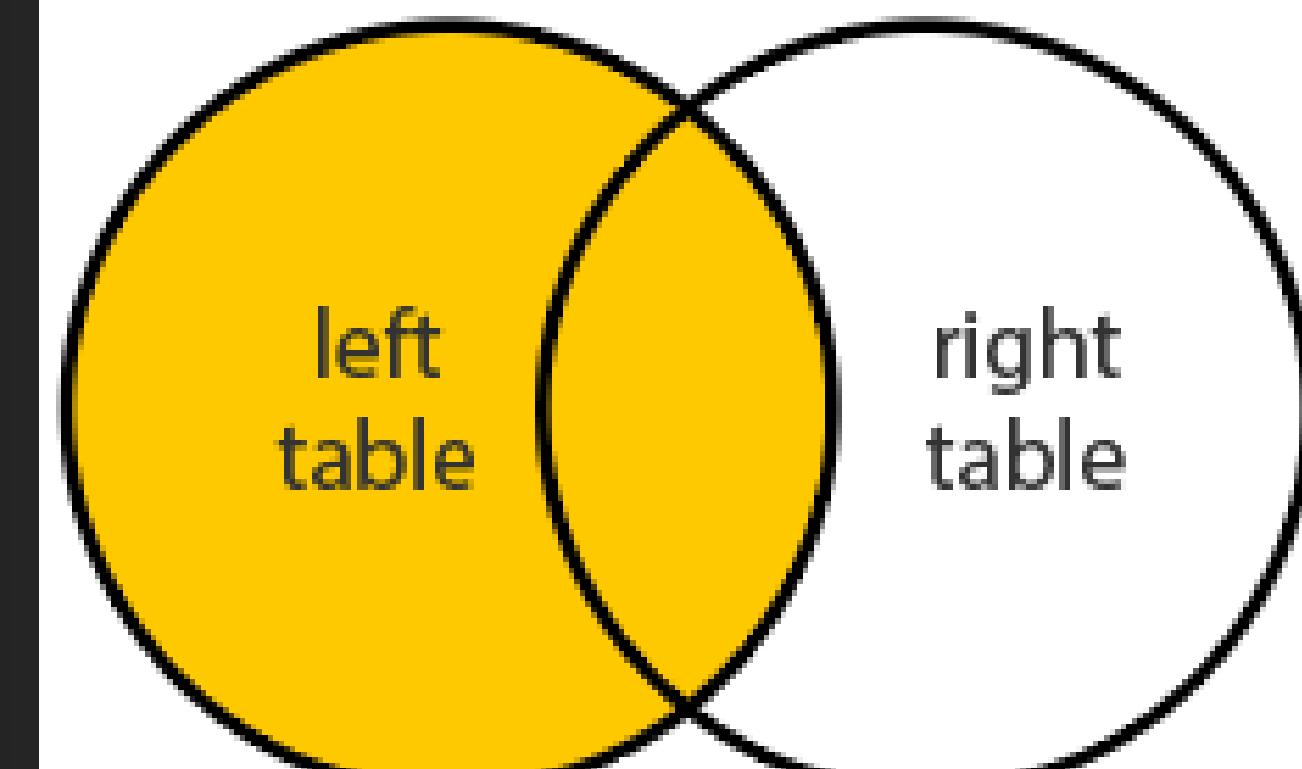
```
SELECT name FROM customers
INNER JOIN orders
ON orders.cust_id=customers.cust_id
GROUP BY name;
```

# Left Join

Returns all rows from the left (or first) table and the matching rows from the right (or second) table.

```
SELECT ...
FROM table1 t1
LEFT JOIN table2 t2
    ON t1.common_column = t2.common_column;
```

## LEFT JOIN

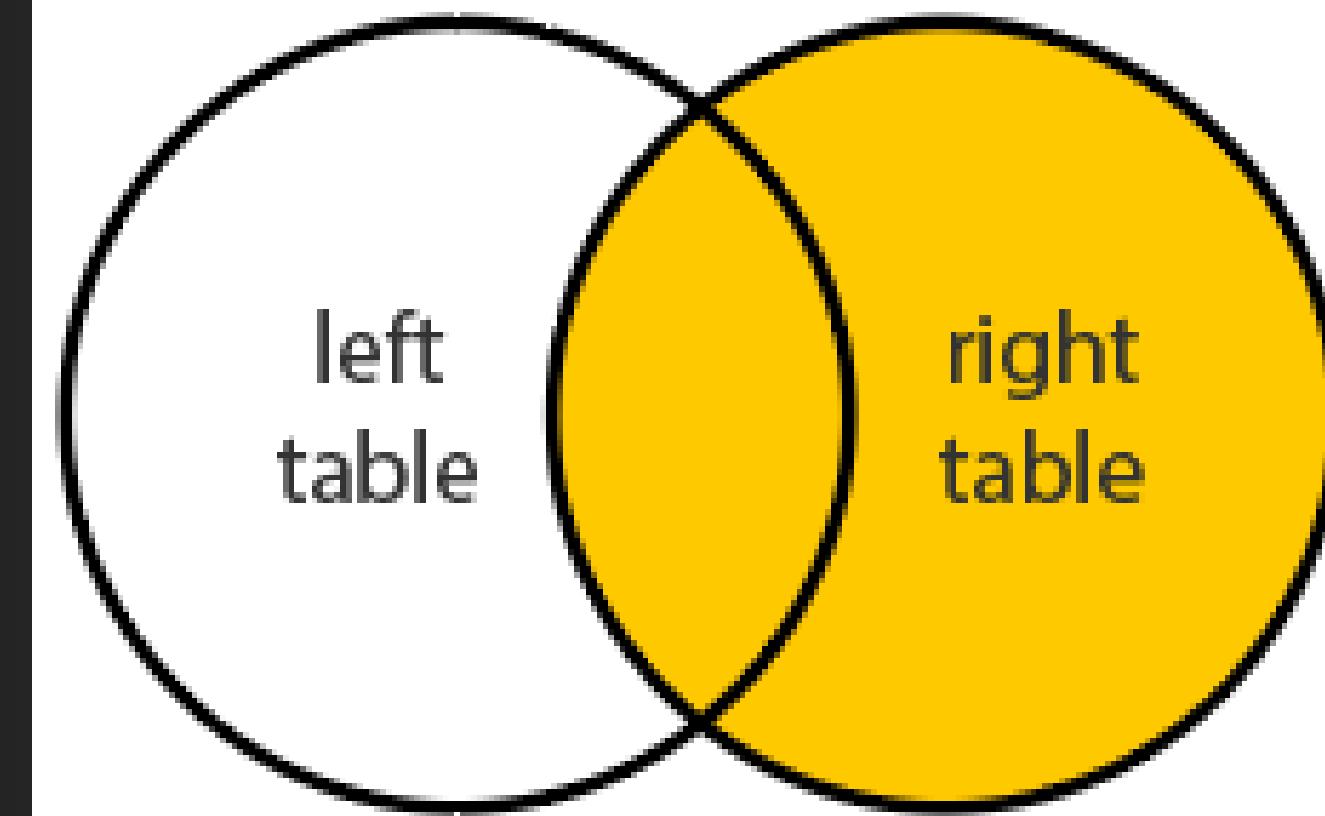


# Right Join

Returns all rows from the right (or second) table  
and the matching rows from the left (or first) table.

```
SELECT ...
FROM table1 t1
RIGHT JOIN table2 t2
ON t1.common_column = t2.common_column;
```

## RIGHT JOIN



# Full Outer Join

Returns all rows when there is a match in either the left or right table.

```
SELECT ...  
FROM table1 t1  
FULL OUTER JOIN table2 t2  
ON t1.common_column = t2.common_column;
```

# OUTER APPLY

OUTER APPLY is used to join each row from one table (the left table) to the results of a table-valued function or subquery (the right side).

# Usecase of OUTER APPLY

For each customer, show their most recent order  
(if they have one).

If they have no orders, still show the customer.

```
SELECT  
    c.customer_id,  
    c.customer_name,  
    o.order_id,  
    o.order_date,  
    o.total_amount  
FROM Customers AS c  
  
OUTER APPLY (  
    SELECT TOP 1 *  
    FROM Orders AS o  
    WHERE o.customer_id = c.customer_id  
    ORDER BY o.order_date DESC  
) AS o;
```



## What happens here:

- For each customer (c), SQL Server runs the subquery on the right.
- The subquery selects the latest order (because of ORDER BY ... DESC and TOP 1).
- If the customer has no orders, it returns NULL values for order columns – similar to a LEFT JOIN.

# CROSS APPLY

**CROSS APPLY** is used to join each row from one table (the left table) to the results of a table-valued function or subquery (the right side).

It behaves like an **INNER JOIN**, meaning it only returns rows where the right-side subquery produces a result.

# **UNION**

**UNION** is used to combine the results of  
two or more **SELECT** statements into a  
single result set.

combines data vertically (adds rows, same structure).

## Requirements:

1. Each SELECT must have the same number of columns.
2. Corresponding columns must have compatible data types.
3. Column names are taken from the first SELECT.

## Union vs Union ALL

Union removes the duplicates from combined result

```
SELECT column_list FROM table1  
UNION  
SELECT column_list FROM table2;
```

## Mumbai

EmployeeID	Name	Department
101	Amit Sharma	Sales
102	Priya Singh	Marketing
103	Rohan Gupta	IT

## Delhi

EmployeeID	Name	Department
201	Sonia Verma	Sales
202	Karan Mehta	Finance
103	Rohan Gupta	IT

## UNION Mum & Delhi

EmployeeID	Name	Department
101	Amit Sharma	Sales
102	Priya Singh	Marketing
103	Rohan Gupta	IT
201	Sonia Verma	Sales
202	Karan Mehta	Finance
103	Rohan Gupta	IT

# EXCEPT

EXCEPT returns rows from the first query  
that do not exist in the second query.

```
SELECT column_list FROM TableA  
EXCEPT  
SELECT column_list FROM TableB;
```



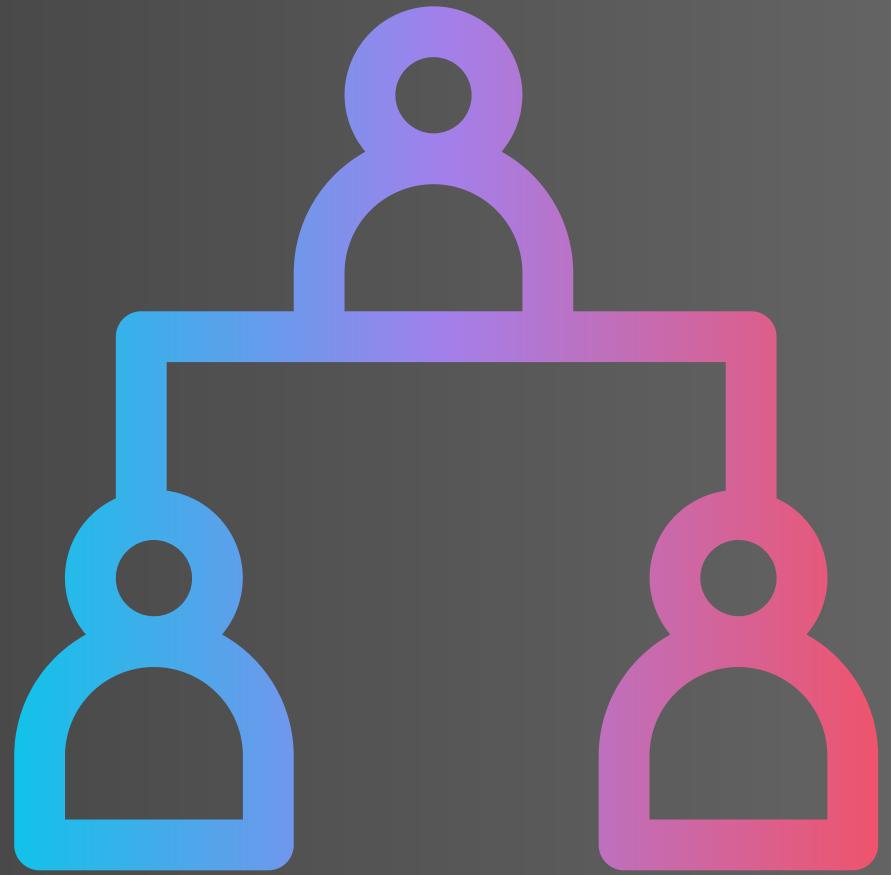
# **SELF JOIN**

**A self join is a standard SQL join where a table is joined to itself.**

**It's used when rows in a table are related to other rows in the same table.**



CEO  
Emp\_ID - 1



Emp\_ID - 2      Emp\_ID - 3  
Manager\_ID - 1    Manager\_ID - 1

Let's create a new table

```
CREATE TABLE CompanyHierarchy (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(100),
    ManagerID INT
);
```

```
INSERT INTO CompanyHierarchy (EmployeeID, Name, ManagerID)
VALUES
(1, 'Sonia Verma', NULL), -- The CEO
(2, 'Rohan Gupta', 1),   -- Reports to Sonia
(3, 'Amit Sharma', 2),   -- Reports to Rohan
(4, 'Priya Singh', 1),   -- Reports to Sonia
(5, 'Kabir Shah', 2);   -- Reports to Rohan
```

```
SELECT  
    e.Name AS EmployeeName,  
    m.Name AS ManagerName  
FROM  
    CompanyHierarchy AS e  
LEFT JOIN  
    CompanyHierarchy AS m  
ON e.ManagerID = m.EmployeeID;
```



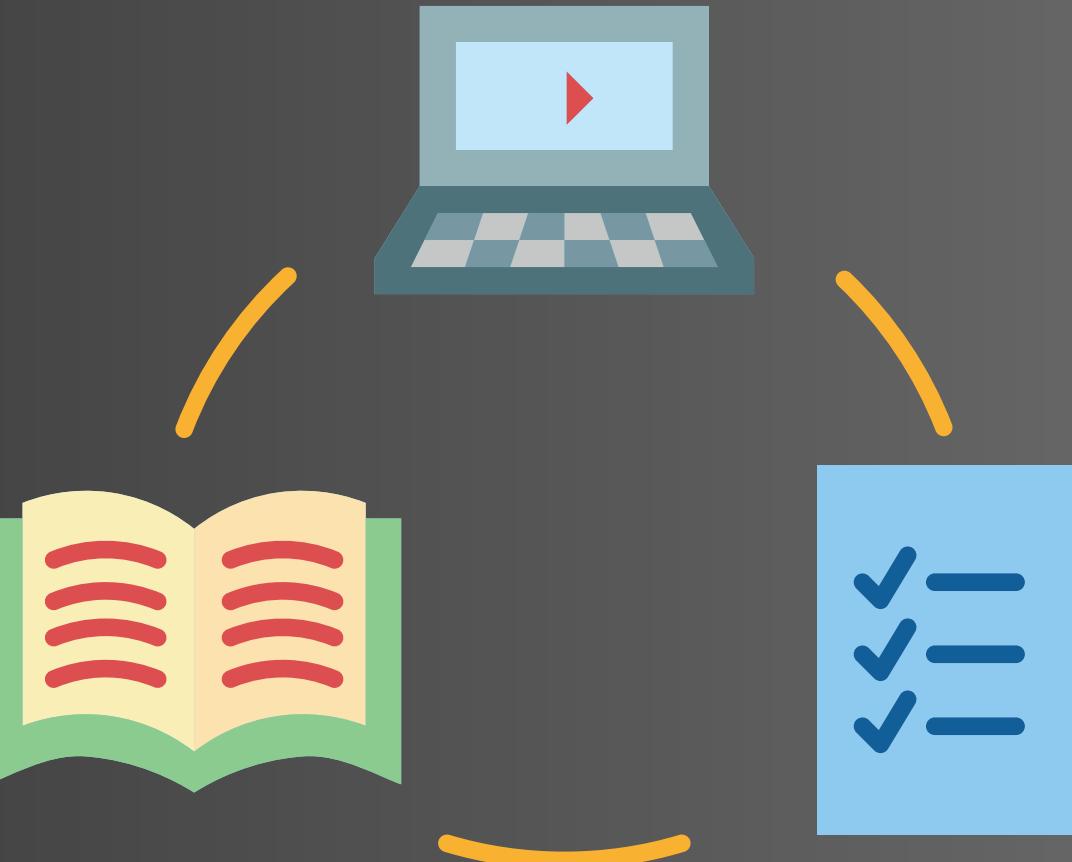
**Many : Many**

Let's Understand a Use-Case of  
Many : Many

# Students



# Courses



**Student A**



**Course A**



**Course B**



**Course C**

# Course A



**Student A**



**Student B**



**Student C**



# students

- **id**
- **student\_name**

# courses

- **id**
- **course\_name**
- **fees**

- A single student can enroll in many courses (like Math, History, and Art).
- A single course (like Math) can have many students enrolled in it.

## students

- id
- student\_name

## courses

- id
- course\_name
- fees

## enrollment

- student\_id
- course\_id

## students

- id
- student\_name

## courses

- id
- course\_name
- fees

## enrollment

- student\_id
- course\_id



# TASK





# e-store

Create a one-to-many and many-to-many relationship in a shopping store context using four tables:

- **customers**
- **orders**
- **products**
- **order\_items**

Include a price column in the **products** table and display the relationship between customers and their orders, along with the details of the products in each order.

## Customers

cust\_id  
cust\_name

## Orders

ord\_id  
ord\_date  
cust\_id

## Products

p\_id  
p\_name  
price

## ord\_items

items\_id  
ord\_id  
p\_id  
quantity

# End Result

cust_name	ord_id	ord_date	p_name	quantity	price	total_price
Sham	4	2024-04-04	Keyboard	1	800.00	800.00
Raju	1	2024-01-01	Laptop	1	55000.00	55000.00
Raju	1	2024-01-01	Cable	2	250.00	500.00
Sham	2	2024-02-01	Laptop	1	55000.00	55000.00
Paul	3	2024-03-01	Mouse	1	500	500
Paul	3	2024-03-01	Cable	3	250.00	750.00
Sham	4	2024-04-04	Keyboard	1	800.00	800.00

# VIEWS

**A view in MS SQL Server is a virtual table that shows data from a saved query.**

**It doesn't store data, just displays it from other tables.**

# How to check existing views

```
SELECT  
    TABLE_SCHEMA,  
    TABLE_NAME  
FROM  
    INFORMATION_SCHEMA.VIEWS;
```

# How to check code of views

```
sp_helptext 'YourViewName';
```

# Delete a View

```
DROP VIEW YourViewName;  
DROP VIEW IF EXISTS YourViewName;
```



# **STORED ROUTINE**

# **STORED Routine**

An SQL statement or a set of SQL Statement  
that can be stored on database server  
which can be call no. of times.



**Order of Pizza**  
**Receipt:**  
**Prepare BASE**  
**Add topping**  
**Bake**



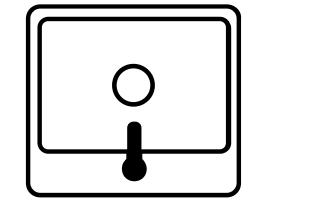
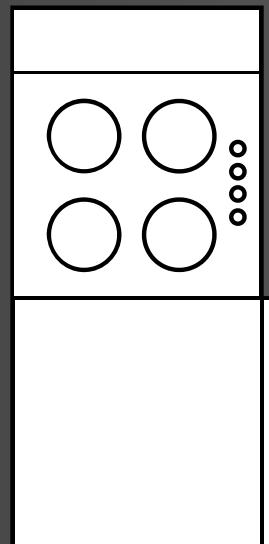


**Order of Pizza**

pizza



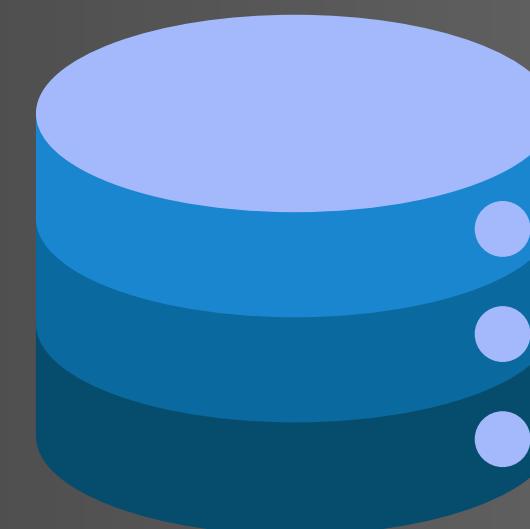
burger



**sp\_employees\_get**



**sp\_employees\_get**



# Types of STORED Routine

- STORED Procedure
- User defined Functions

# STORED PROCEDURE

# **STORED PROCEDURE**

**Set of SQL statements &  
Procedural Logic that can perform operations  
such as  
INSERT, UPDATE, DELETE, and QUERING data.**

**1. SP without parameters**

**2. SP with INPUT parameters**

**3. SP with INPUT & OUTPUT parameters**

```
CREATE PROCEDURE procedure_name  
    @parameter_name datatype,  
    ...  
AS  
BEGIN  
    -- procedural code here  
END;
```

# How to Check Existing SP

```
SELECT  
    ROUTINE_NAME  
FROM  
    INFORMATION_SCHEMA.ROUTINES  
WHERE  
    ROUTINE_TYPE = 'PROCEDURE'  
ORDER BY  
    ROUTINE_NAME;
```

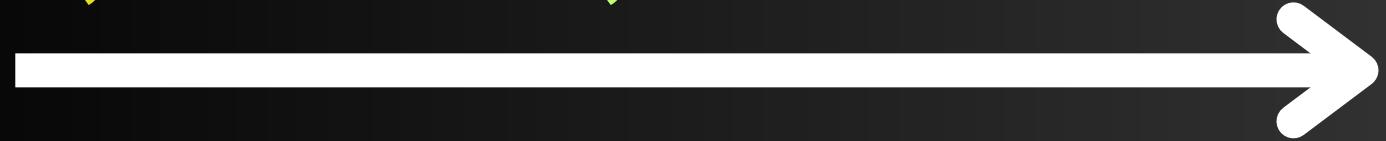
```
EXEC sp_helptext 'YourProcedureName';
```

```
CREATE PROCEDURE update_emp_salary
    @p_employee_id INT,
    @p_new_salary NUMERIC(10, 2)
AS
BEGIN
    UPDATE employees
    SET salary = @p_new_salary
    WHERE emp_id = @p_employee_id;
END;
```

## How to execute

- EXEC update\_emp\_salary  
    @p\_employee\_id = 102,  
    @p\_new\_salary = 125000.00;
- EXEC update\_emp\_salary 102, 125000.00;

```
EXEC update_emp_salary  
102, 125000.00;
```



```
CREATE PROCEDURE update_emp_salary  
    @p_employee_id INT = 102,  
    @p_new_salary NUMERIC(10, 2) = 125000  
AS  
BEGIN  
    UPDATE employees  
    SET salary = 102  
    WHERE emp_id = 125000;  
END;
```

```
CREATE PROCEDURE add_employee
```

```
    @p_fname VARCHAR(50),  
    @p_lname VARCHAR(50),  
    @p_email VARCHAR(100),  
    @p_job_title VARCHAR(50),  
    @p_department VARCHAR(50),  
    @p_salary NUMERIC(10, 2),  
    @p_city VARCHAR(50)
```

```
AS
```

```
BEGIN
```

```
    INSERT INTO employees (fname, lname, email, job_title, department, salary, city)  
    VALUES (@p_fname, @p_lname, @p_email, @p_job_title, @p_department,  
           @p_salary, @city);
```

```
END;
```

# How to Modify or Delete a Procedure

- **ALTER PROCEDURE sp\_GetAllTechEmployees ...**
- **DROP PROCEDURE sp\_GetAllTechEmployees;**

# Procedural logic

Procedural logic in a stored procedure means

The ability to write SQL code that follows a step-by-step (imperative) flow, just like in a programming language.

## Variables

DECLARE @x INT

Store temporary values

## Conditions

IF ... ELSE

Decision-making

## Loops

WHILE ...

Repeat logic until a condition is met

## Error handling

TRY...CATCH

Handle exceptions

## Flow control

RETURN , BREAK , CONTINUE

Control execution flow

# USECASE

Create a procedure to update an employee's salary, but  
only if the new salary is a raise  
(i.e., greater than the current salary).  
We also want to return a message about what happened.

## Steps:

- Get emp\_id and new\_salary as input
- Check if emp\_id exist and valid
- Compare new\_salary with current salary to check if new salary is more than current.
  - If yes, update the salary
  - If No, give error message

```
CREATE PROCEDURE sp_SafelyUpdateSalary
    -- Input parameters
    @p_employee_id INT,
    @p_new_salary NUMERIC(10, 2),

    -- Output parameter for our message
    @p_message VARCHAR(200) OUTPUT
AS
BEGIN
    SET NOCOUNT ON; -- Stops "1 row(s) affected" messages

    -- 1. Declare a variable
    DECLARE @current_salary NUMERIC(10, 2);

    -- 2. Check if the employee exists
    IF NOT EXISTS (SELECT 1 FROM employees WHERE emp_id = @p_employee_id)
    BEGIN
        SET @p_message = 'Error: Employee ID does not exist.';
        RETURN; -- This exits the procedure immediately
    END

    -- 3. Get the current salary
    SELECT @current_salary = salary
    FROM employees
    WHERE emp_id = @p_employee_id;

    -- 4. This is the procedural logic!
    IF @p_new_salary > @current_salary
    BEGIN
        -- 5. Logic passed: Update the salary
        UPDATE employees
        SET salary = @p_new_salary
        WHERE emp_id = @p_employee_id;

        SET @p_message = 'Success: Salary updated.';
    END
    ELSE
    BEGIN
        -- 6. Logic failed: Do not update
        SET @p_message = 'Error: New salary must be greater than the current salary.';
    END
END;
```

Benefit	Explanation
<b>Security</b> 	This is a huge benefit. You can grant a user <code>EXECUTE</code> permission on the procedure <b>without</b> giving them <code>SELECT</code> or <code>UPDATE</code> permission on the actual tables. The user can <i>run the recipe</i> ( <code>sp_UpdateSalary</code> ) but can't see or touch the raw ingredients (the <code>employees</code> table).
<b>Performance</b> 	When you run an SP for the first time, SQL Server creates an "execution plan" (the fastest way to run the query) and saves it. The next time, it skips the planning step and is much faster. It's "pre-compiled."
<b>Reusability</b> 	You write the complex logic <i>once</i> (e.g., a 50-line query to add a new employee with validation). Now, your website, mobile app, and reports can all just call <code>EXEC sp_AddNewEmployee</code> . If the logic changes, you only fix it in one place!
<b>Simplicity</b> 	It hides complex logic. An application developer doesn't need to know the 10 tables involved in a query. They just call <code>EXEC sp_GetCustomerOrderHistory @customerID = 123</code> .
<b>Reduced Network Traffic</b>	The app sends one short line ( <code>EXEC sp_GetCustomerOrderHistory 123</code> ) instead of a giant 500-line SQL script over the network.

# USER DEFINED FUNCTIONS

custom function created by the user  
to perform specific operations and  
return a value.

## Scalar Function

```
CREATE FUNCTION function_name
(
    @param1 INT,
    @param2 VARCHAR(50)
)
RETURNS return_data_type (INT, VARCHAR, DATE etc..)
AS
BEGIN
    DECLARE @result INT;
    -- Example logic
    SET @result = @param1 * 2;
    RETURN @result;
END;
```

**Usecase:**

**Create a function which returns double the salary**

```
CREATE FUNCTION fn_GetYearsOfService (
    @p_hire_date DATE
)
RETURNS INT
AS
BEGIN
    DECLARE @YearsOfService INT;
    SET @YearsOfService = DATEDIFF(YEAR, @p_hire_date, GETDATE());

    -- This complex part is what makes it a good function!
    IF (DATEADD(YEAR, @YearsOfService, @p_hire_date) > GETDATE())
        BEGIN
            SET @YearsOfService = @YearsOfService - 1;
        END

    RETURN @YearsOfService;
END;
```

## Inline Table-Valued Function (ITVF)

```
CREATE FUNCTION function_name
(
    @param1 INT,
    @param2 VARCHAR(50)
)
RETURNS TABLE
AS
RETURN (
    SELECT column1, column2
    FROM your_table
    WHERE some_column = @parameter1
)
```

**Find name of the employees in each department having maximum salary.**

```
CREATE FUNCTION dept_max_sal_emp1 (@dept_name VARCHAR(100))
RETURNS TABLE
AS
RETURN
(
    SELECT e.emp_id, e.fname, e.salary
    FROM
        employees e
    WHERE
        e.dept = @dept_name
    AND e.salary = (
        SELECT MAX(emp.salary)
        FROM employees emp
        WHERE emp.dept = @dept_name
    )
);
```





# TRIGGERS

Triggers are special procedures in a database that automatically execute predefined actions in response to certain events on a specified table or view.

# Use Case

Audit Salary Changes (Track Old vs  
New Values)

Employees Salary

SalaryAudit

# Trigger



# Database

```
CREATE TRIGGER trigger_name
ON table_name
AFTER INSERT, UPDATE, DELETE -- OR INSTEAD OF INSERT/UPDATE/DELETE
AS
BEGIN
    -- Trigger logic here
END;
```

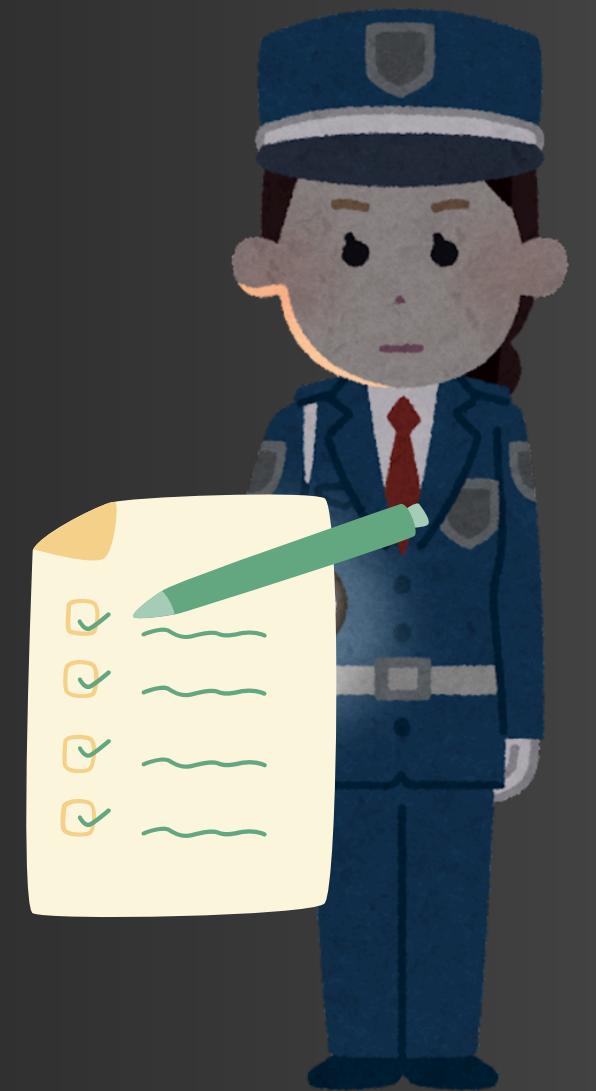
```
CREATE TABLE SalaryAudit (
    AuditID INT IDENTITY PRIMARY KEY,
    EmplID INT,
    OldSalary DECIMAL(10,2),
    NewSalary DECIMAL(10,2),
    ChangedDate DATETIME DEFAULT GETDATE()
);
```

```
CREATE TRIGGER trg_AuditSalaryChange
ON Employees
AFTER UPDATE
AS
BEGIN
    IF UPDATE(Salary)
    BEGIN
        INSERT INTO SalaryAudit (EmpID, OldSalary, NewSalary)
        SELECT
            d.EmpID,
            d.Salary AS OldSalary,
            i.Salary AS NewSalary
        FROM deleted d
        INNER JOIN inserted i ON d.EmpID = i.EmpID;
    END
END;
```

**deleted**



**inserted**



**Trigger**



**Database**

# Use Case

Prevent accidental removal of  
employee is a specific department.

Ex: Management

```
CREATE TRIGGER trg_PreventManagementDeletion
ON Employees
INSTEAD OF DELETE
AS
BEGIN
    -- Prevent deletion of Management employees
    IF EXISTS (SELECT 1 FROM deleted WHERE Department = 'Management')
        BEGIN
            RAISERROR('Deletion not allowed for Management employees.', 16, 1);
            ROLLBACK TRANSACTION;
            RETURN;
        END
    -- Allow deletion for others
    DELETE FROM Employees
    WHERE EmpID IN (SELECT EmpID FROM deleted);
END;
```

# **Random Data for Testing**

```
CREATE TABLE Employees (
    EmployeeID INT IDENTITY PRIMARY KEY,
    FirstName NVARCHAR(50),
    LastName NVARCHAR(50),
    Department NVARCHAR(50),
    Salary INT
);
```

```
INSERT INTO Employees (FirstName, LastName, Department, Salary)
SELECT TOP (500000)
    LEFT(NEWID(), 8), -- random first name
    LEFT(NEWID(), 8), -- random last name
    CASE ABS(CHECKSUM(NEWID())) % 5
        WHEN 0 THEN 'IT'
        WHEN 1 THEN 'HR'
        WHEN 2 THEN 'Finance'
        WHEN 3 THEN 'Marketing'
        ELSE 'Sales'
    END,
    ABS(CHECKSUM(NEWID())) % 100000 + 30000
FROM sys.objects a
CROSS JOIN sys.objects b;
```

# INDEXES

**Creating and using indexes in SQL Server is  
a powerful way to improve database  
performance.**



## How to add index?

- **CREATE INDEX i\_name  
ON employees(salary, emp\_id);**

## How to see index?

- **EXEC sp\_helpindex 'employees';**



## Step 2: Enable Query Statistics

Before running any queries, enable these options:

```
sql
```

```
SET STATISTICS TIME ON;
```

```
SET STATISTICS IO ON;
```

```
GO
```

## How to remove index?

- **DROP INDEX i\_name ON employees;**

- Clustered Index
- Non-Clustered Index

## Performance Comparison

**Without Index:** Full table scan, time complexity  $O(n)$ .

**With Index:** Indexed search, time complexity  $O(\log n)$ .

# Normalization

**Normalization is the process of organizing  
data in a database efficiently**

**i.e. reduce redundancy and improve data integrity.**

## First Normal Form (1NF)

- Each column contains only atomic (indivisible) values.  
→ No arrays, lists, or sets inside a single cell.
- Each column stores values of a single data type.  
→ Example: You shouldn't mix text and numbers in the same column.
- Each record (row) is unique.  
→ Usually ensured by a primary key.

**Example:**

 courses = "Math, Physics"

◆ 1NF – First Normal Form (Atomic values only)

StudentID	Name	Courses
1	Raju	Math, Physics
2	Sham	Chemistry

✗ Problem:

- Courses column has multiple values → not atomic.

Fix (Convert to 1NF):

StudentID	Name	Course
1	Raju	Math
1	Raju	Physics
2	Sham	Chemistry

## Second Normal Form (2NF)

- Rule: Be in 1NF + every non-key column depends on the whole primary key.
- Applies to: Composite keys.

### Example:

- If table has (student\_id, course\_id) as key, then attributes like student\_name should not be there (it depends only on student\_id).

```
CREATE TABLE Enrollment (
    student_id INT,
    course_id INT,
    enrollment_date DATE,
    PRIMARY KEY (student_id, course_id)
);
```

- ◆ **2NF – Second Normal Form (Remove partial dependency)**

Now let's assume a composite key like `(StudentID, Course)` is used:

StudentID	Course	StudentName
1	Math	Raju
1	Physics	Raju
2	Chemistry	Sham

✖ **Problem:**

- `StudentName` depends **only** on `StudentID`, not on the full key (`StudentID + Course`).

## Third Normal Form (3NF)

- Rule: Be in 2NF + no transitive dependency  
(non-key → non-key).

**Does any non-key attribute depend on  
another non-key attribute?**

### Example:

-  employee(city, zip, state)
-  Move zip → city, state to a separate  
zip\_codes table.

employee_id (PK)	city	zip	state
101	Mumbai	40001	MH
102	Pune	41101	MH
103	Mumbai	40002	MH

## Problem:

- `city` depends on `zip`.
- `state` depends on `zip`.
- `zip` depends on `employee_id` (where the employee lives).
- Since `city` and `state` depend on `zip` (a non-key attribute), and `zip` depends on `employee_id` (the key), you have **transitive dependencies**:
  - `employee_id -> zip -> city`
  - `employee_id -> zip -> state`
- This violates 3NF. It leads to redundancy (MH is stored multiple times) and update anomalies (if a zip code changes its city, you have to update many employee records).

# CASCADE ON DELETE

## Primary Key

Customers

cust_id	name	email
101	Raju	raju@email.com
102	Sham	sham@email.com
103	Baburao	babu@email.com

## Primary Key

Orders

## Foreign Key

order_id	date	amount	cust_id
ord-1	2023-05-15	200	101
ord-2	2023-04-28	500	102
ord-3	2023-05-14	1000	101

```
CREATE TABLE orders (
    ord_id INT IDENTITY(1,1) PRIMARY KEY, -- Use IDENTITY(1,1)
    date DATE,
    amount DECIMAL(10, 2),
    cust_id INT,
    FOREIGN KEY (cust_id) REFERENCES customers(cust_id) ON DELETE CASCADE
);
```

# **IMPORT / EXPORT**

# **Database**

# Taking Backup

```
BACKUP DATABASE YourDatabaseName  
TO DISK = 'C:\Path\To\Your\Backup\YourDatabaseName.bak'  
WITH FORMAT, -- Overwrites existing backup file if it exists, creates new if not  
      NAME = 'Full Backup of YourDatabaseName'; -- A description for the backup
```

# Restoring Database

```
RESTORE DATABASE YourDatabaseName -- Or a NewDatabaseName  
FROM DISK = 'C:\Path\To\Your\Backup\YourDatabaseName.bak'  
WITH REPLACE, -- Overwrites the existing database if it exists  
NOUNLOAD,  
STATS = 5; -- Shows progress updates
```

# Import data from CSV

```
BULK INSERT YourTableName
FROM 'C:\Path\To\Your\File\data.csv'
WITH (
    FORMAT = 'CSV', -- Specify the format
    FIELDTERMINATOR = ',', -- Specify the column delimiter (comma for CSV)
    ROWTERMINATOR = '\n', -- Specify the row delimiter (newline)
    FIRSTROW = 2 -- Skip the header row if it exists
);
```