



ALRIGHT!

# SQL TUTORIAL



# PostgreSQL

***ZERO TO HERO***

# Section - 1

- **What is database?**
- **Database vs DBMS**
- **DBMS vs RDBMS**
- **Other available databases**
- **SQL vs PostgreSQL**

# What is a Database?



# **What is a Database?**

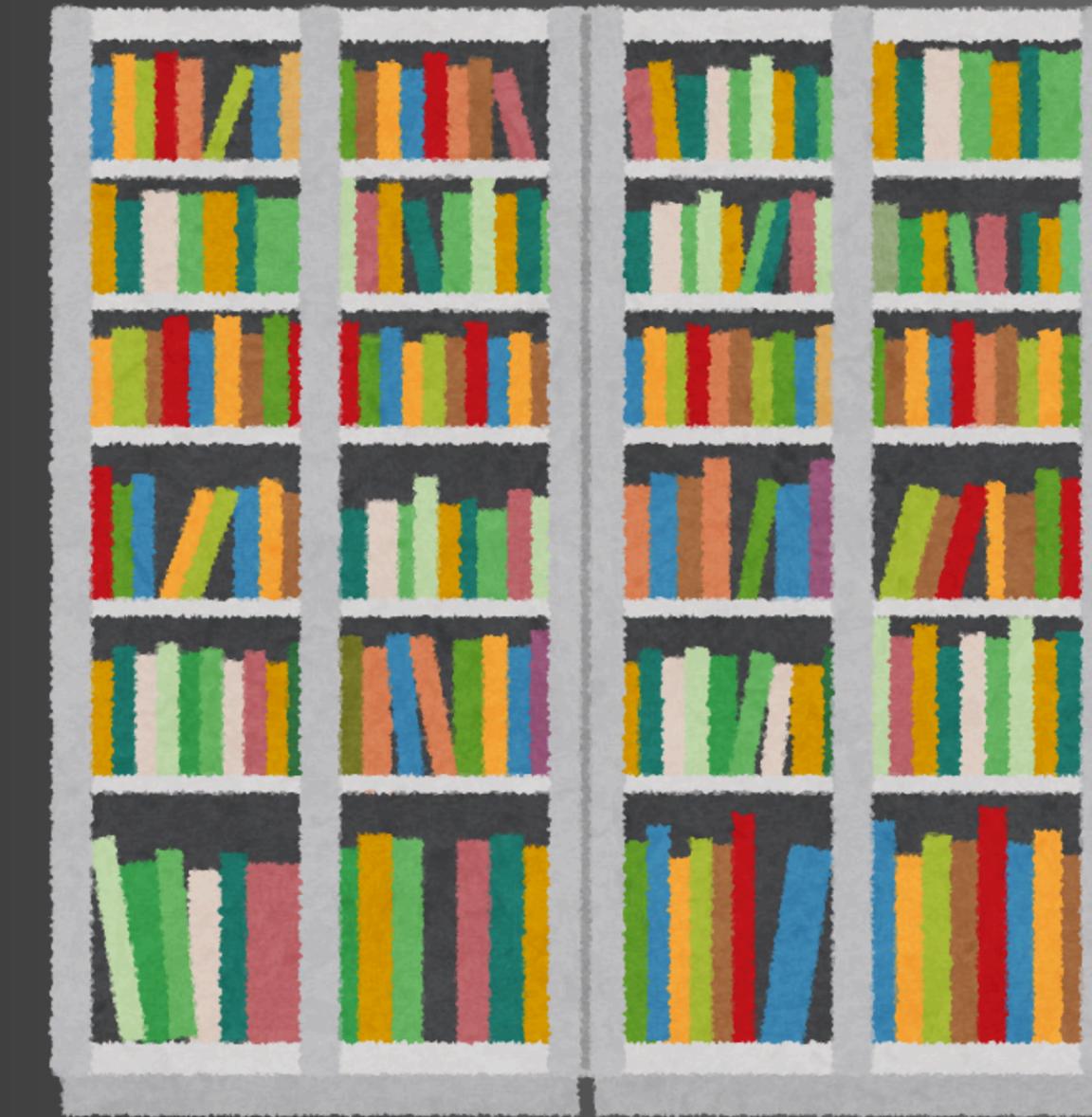
**An organised collection of data.**

**A method to manipulate and access the data.**

# Unorganized



# Organized

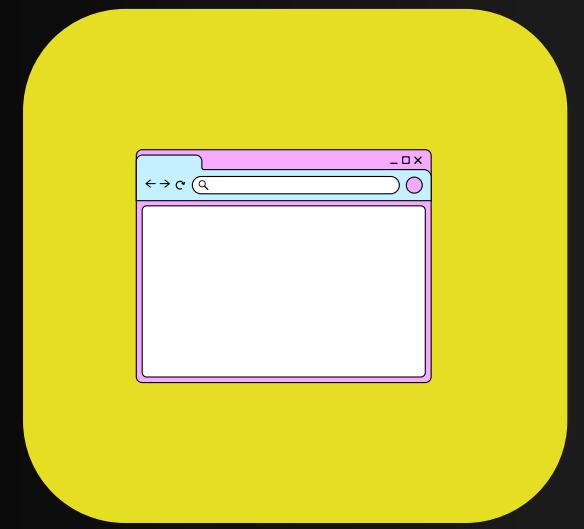


Firstname	Lastname	City	Contact
Paul	Philips	London	39899829
Raju	Sharma	Ranchi	90890288
Keto	Leri	Tokyo	50505005
Sham	Sha	Delhi	602020

# **Database**

**vs**

# **DBMS**



App



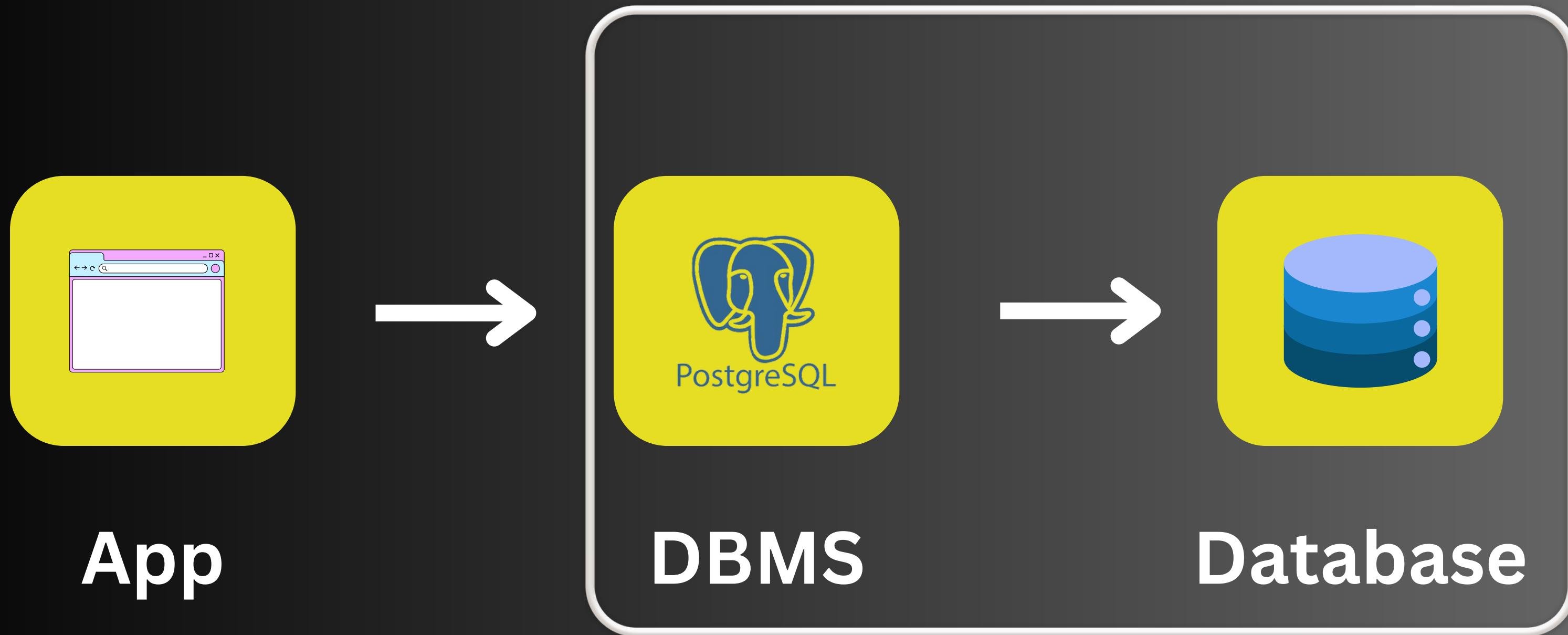
Database



App

DBMS

Database



PostGresql, MySQL, Oracle 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.

## Some Other Databases are:

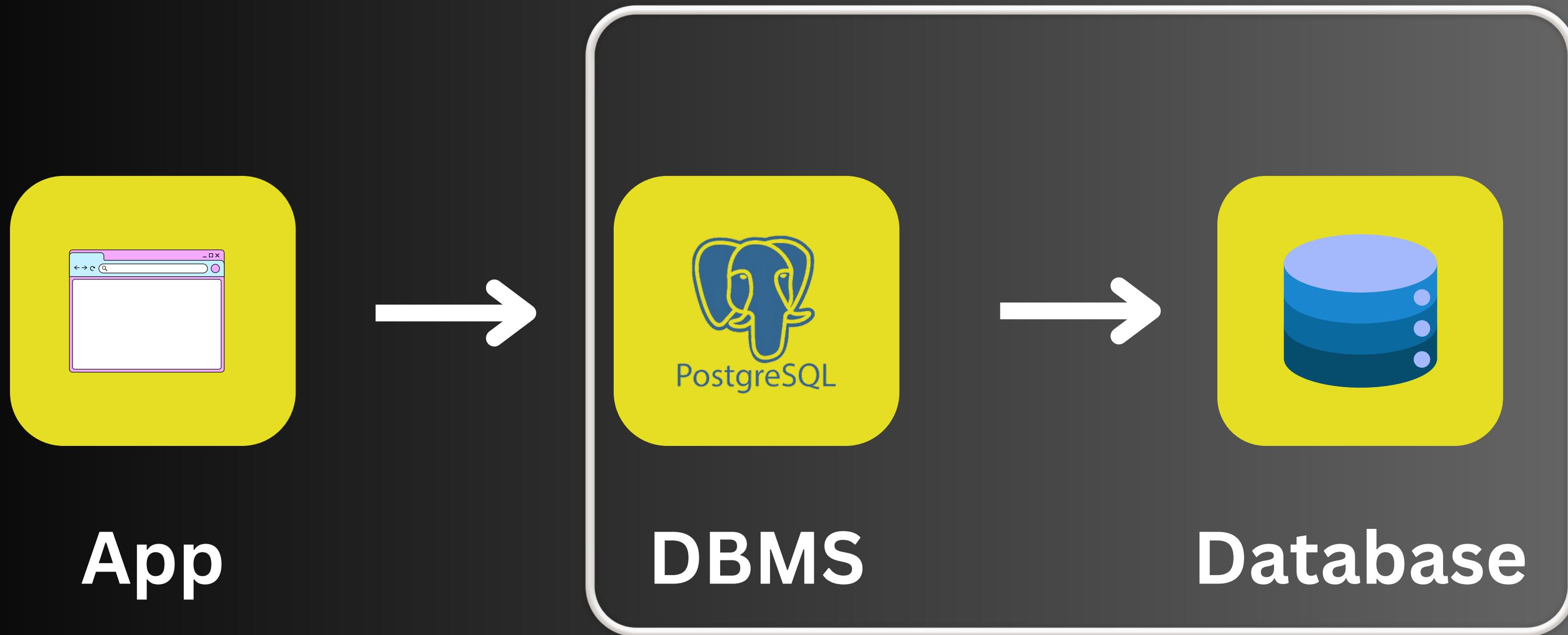
- MongoDB
- Oracle
- MySQL
- SQLite
- PostgreSQL
- MaxDB
- Firebird
- Redis

# SQL vs PostgreSQL

# SQL

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

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



PostGresql, MySQL, Oracle etc

# Installation

# Database

vs

# Schema

vs

# Table

# Section - 2

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

# Databases

# List down existing databases

- **SELECT datname FROM pg\_database;**
- \l
-

# Some Important Queries in starting

- `\l` - list all databases
- `\q` - quit
- `\c db_name` - connect to database
- `\dt` - list all tables
- `\d tb_name` - desc a table
- `\d+ tb_name` - list all column
- `\! cls.` - list all column

# **Creating a new Database**

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

# Change a Database

```
\c <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.

<b>id</b>	<b>name</b>	<b>city</b>
101	Raju	Delhi
102	Sham	Mumbai
103	Paul	Chennai
104	Alex	Pune

# Creating a new Table

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

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

# Checking your table

```
\d TABLE_NAME
```

# **INSERTING Data**

# Adding data into a Table

```
INSERT INTO person(id, name, city)  
VALUES (101, 'Raju', 'Delhi');
```

```
INSERT INTO students VALUES (101, "Rahul")
```

# READING DATA

# Reading data from a Table

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

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

# UPDATING DATA

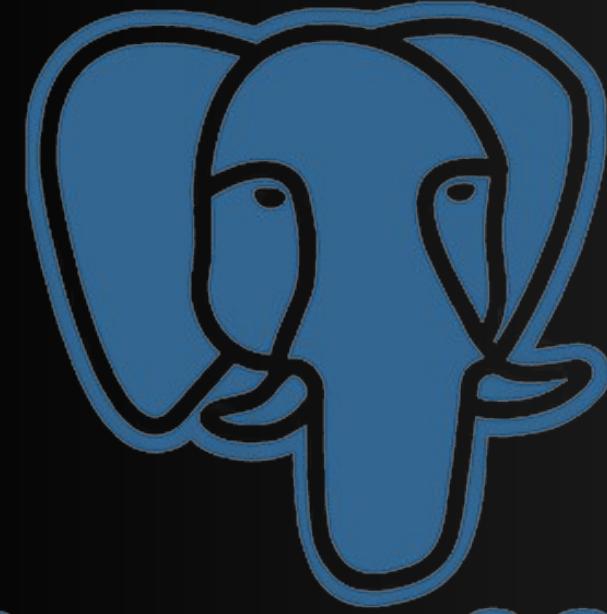
# Modify/Update data from a Table

```
UPDATE person  
SET city='London'  
WHERE id=2;
```

# **DELETING DATA**

# **DELETE data from a Table**

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



PostgreSQL

WHERE

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

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

```
UPDATE students SET contact=12345 WHERE id=104;
```

```
DELETE FROM students WHERE id=104;
```

# Section - 3

- **Datatypes**
- **Constraint**
  - Primary Key
  - NOT NULL
  - Default
  - Serial
  - Unique

# DataTypes

An attribute that specifies the type of data in a column of our database - table.

# Most widely used are

- Numeric - INT DOUBLE FLOAT DECIMAL
- String - VARCHAR
- Date - DATE
- Boolean - BOOLEAN

# **DATATYPES**

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

# DATA TYPES

DECIMAL(5,2)

Digits after decimal

Total digit

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

**Example: 155.38**

119.12



28.15



1150.15



# Constraint

A constraint in PostgreSQL is a rule applied to a column.

# 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.



# NOT NULL

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

# DEFAULT Value

```
CREATE TABLE customers
(
    acc_no INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    acc_type VARCHAR(50) NOT NULL DEFAULT 'Savings'
);
```

acc_no	name	acc_type
101	Raju	Savings
102	Sham	Savings
103	Paul	Current
104	Alex	Savings

# AUTO\_INCREMENT

```
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    firstname VARCHAR(50),
    lastname VARCHAR(50)
);
```

# TASK 1:

## Creating New Table

emp_id	fname	Iname	email	dept	salary	hire_date
1	Raj	Sharma	raj.sharma@example.com	IT	50000	2020-01-15
2	Priya	Singh	priya.singh@example.com	HR	45000	2019-03-22
3	Arjun	Verma	arjun.verma@example.com	IT	55000	2021-06-01
4	Suman	Patel	suman.patel@example.com	Finance	60000	2018-07-30
5	Kavita	Rao	kavita.rao@example.com	HR	47000	2020-11-10
6	Amit	Gupta	amit.gupta@example.com	Marketing	52000	2020-09-25
7	Neha	Desai	neha.desai@example.com	IT	48000	2019-05-18
8	Rahul	Kumar	rahul.kumar@example.com	IT	53000	2021-02-14
9	Anjali	Mehta	anjali.mehta@example.com	Finance	61000	2018-12-03
10	Vijay	Nair	vijay.nair@example.com	Marketing	50000	2020-04-19

```
CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    fname VARCHAR(50) NOT NULL,
    lname VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    dept VARCHAR(50),
    salary DECIMAL(10,2) DEFAULT 30000.00,
    hire_date DATE NOT NULL DEFAULT CURRENT_DATE
);
```

```
INSERT INTO employees (emp_id, fname, lname, email, dept, salary, hire_date)
VALUES
(1, 'Raj', 'Sharma', 'raj.sharma@example.com', 'IT', 50000.00, '2020-01-15'),
(2, 'Priya', 'Singh', 'priya.singh@example.com', 'HR', 45000.00, '2019-03-22'),
(3, 'Arjun', 'Verma', 'arjun.verma@example.com', 'IT', 55000.00, '2021-06-01'),
(4, 'Suman', 'Patel', 'suman.patel@example.com', 'Finance', 60000.00, '2018-07-30'),
(5, 'Kavita', 'Rao', 'kavita.rao@example.com', 'HR', 47000.00, '2020-11-10'),
(6, 'Amit', 'Gupta', 'amit.gupta@example.com', 'Marketing', 52000.00, '2020-09-25'),
(7, 'Neha', 'Desai', 'neha.desai@example.com', 'IT', 48000.00, '2019-05-18'),
(8, 'Rahul', 'Kumar', 'rahul.kumar@example.com', 'IT', 53000.00, '2021-02-14'),
(9, 'Anjali', 'Mehta', 'anjali.mehta@example.com', 'Finance', 61000.00, '2018-12-03'),
(10, 'Vijay', 'Nair', 'vijay.nair@example.com', 'Marketing', 50000.00, '2020-04-19');
```

# Section - 4

# Clauses

- Where
- Distinct
- Order By
- Limit
- Like

# Relational Operators



# Find employees whose salary is more than 65000



# 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

=

```
SELECT * FROM employees  
WHERE salary > 65000;
```

# Logical Operators



**AND**

**OR**

Condition 1

**AND**

Condition 2

When both the conditions are true

salary=25000 **AND** dept=Loan

Condition 1

OR

Condition 2

When either of the condition is true

salary=65000

OR

desig='Lead'

**IN, NOT IN**

# Find employees From following department

- 

- 

- 

IT

Finance

HR



```
SELECT * FROM employees  
WHERE dept = 'IT'  
    OR dept = 'HR'  
    OR dept = 'Finance';
```

```
SELECT * FROM employees
WHERE dept IN ('IT', 'HR', 'Finance');
```

**BETWEEN**

# Find employees whose salary is more than 40000 and Less than 65000

>40000

<65000



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

```
SELECT * FROM employees  
WHERE  
salary BETWEEN 40000 AND 65000;
```

# DISTINCT

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

# ORDER BY

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

# LIMIT

```
SELECT * FROM employees LIMIT 3;
```

# LIKE

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

- Starts with 'A': LIKE 'A%'
- Ends with 'A': LIKE '%A'
- Contains 'A': LIKE '%A%'
- Second character is 'A': LIKE '\_A%
- Case-insensitive contains 'john': ILIKE '%john%

# Section - 5

- How to find total no. of employees?
- Employee with Max salary
- Average salary of employees

- COUNT
- SUM
- AVG
- MIN
- MAX

# COUNT

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

# MAX & MIN

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

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

# SUM & AVG

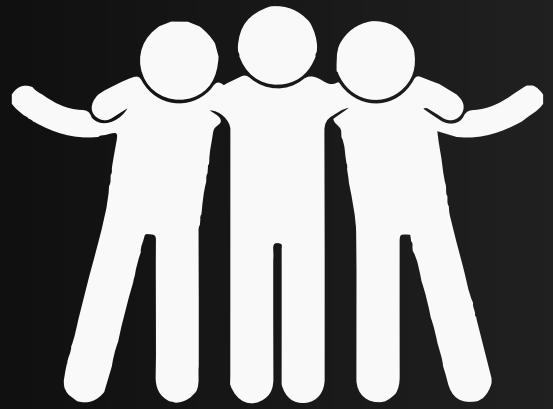
```
SELECT SUM(salary) FROM employees;  
SELECT AVG(salary) FROM employees;
```



# GROUP BY

## No. of employees in each department

dept		count
Marketing		2
Finance		2
IT		4
HR		2



HR



Finance



Marketing



IT



Deposit

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

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

# Section - 6

## String Functions

- CONCAT, CONCAT\_WS
- SUBSTR
- LEFT, RIGHT
- LENGTH
- UPPER, LOWER
- TRIM, LTRIM, RTRIM
- REPLACE
- POSITION

# 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

Hey → Hello

Hello Buddy

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

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

# REVERSE

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

# LENGTH

Select LENGTH( '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 POSITION('OM' in 'Thomas');
```



# Exercise

**Task 1:**

**1:Raj:Sharma:IT**

**Task2:**

**1:Raju Sharma:IT:50000**

# **Task3**

## **4:Suman:FINANCE**

# **Task4**

**I1 Raju**

**H2 Priya**

# **Exercise**

## DISTINCT, ORDER BY, LIKE and LIMIT

**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**

# 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?**

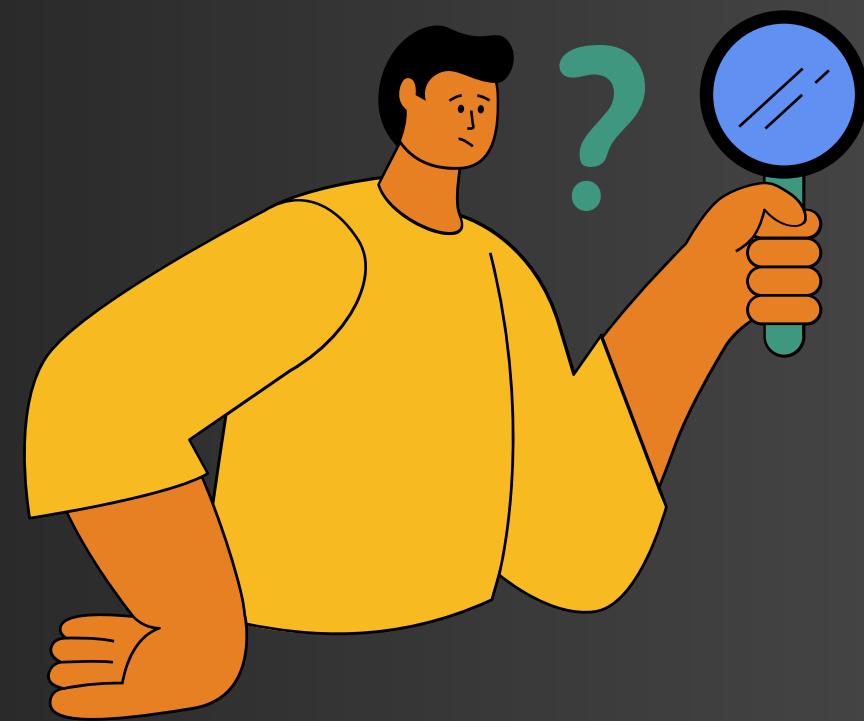
**6: Average salary paying in each department**

# Section - 7

# **ALTERING**

## **Tables**

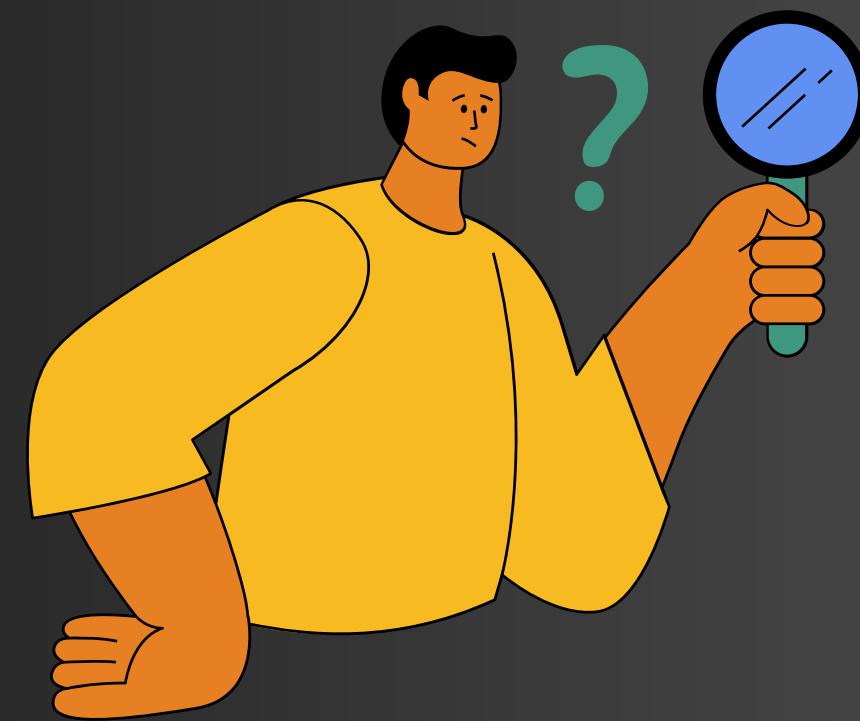
# How to add or remove a column?



```
ALTER TABLE contacts  
ADD COLUMN city VARCHAR(50);|
```

```
ALTER TABLE contacts  
DROP COLUMN city;
```

# How to rename a column or table name?



# How to rename a column?

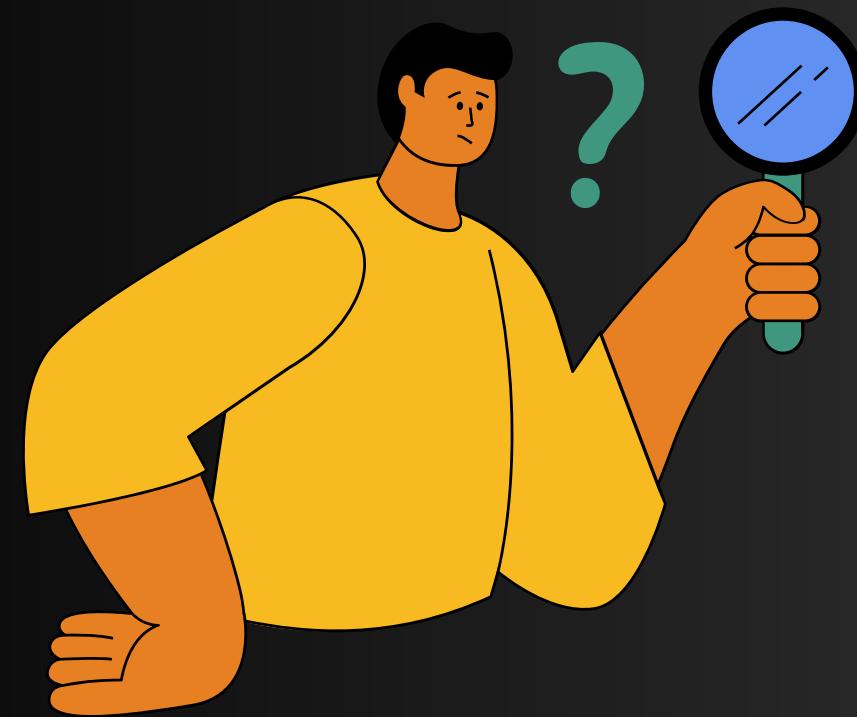
```
ALTER TABLE contacts  
RENAME COLUMN name TO full_name;
```

# How to rename a table name?

```
ALTER TABLE contacts  
RENAME TO mycontacts;
```

```
RENAME TABLE contacts TO mycontacts;
```

# How to modify a column?



Ex: Changing datatype  
or adding Default values etc

# How to add DEFAULT value to a column?

```
ALTER TABLE person  
ALTER COLUMN fname  
SET DATA TYPE VARCHAR(200);
```

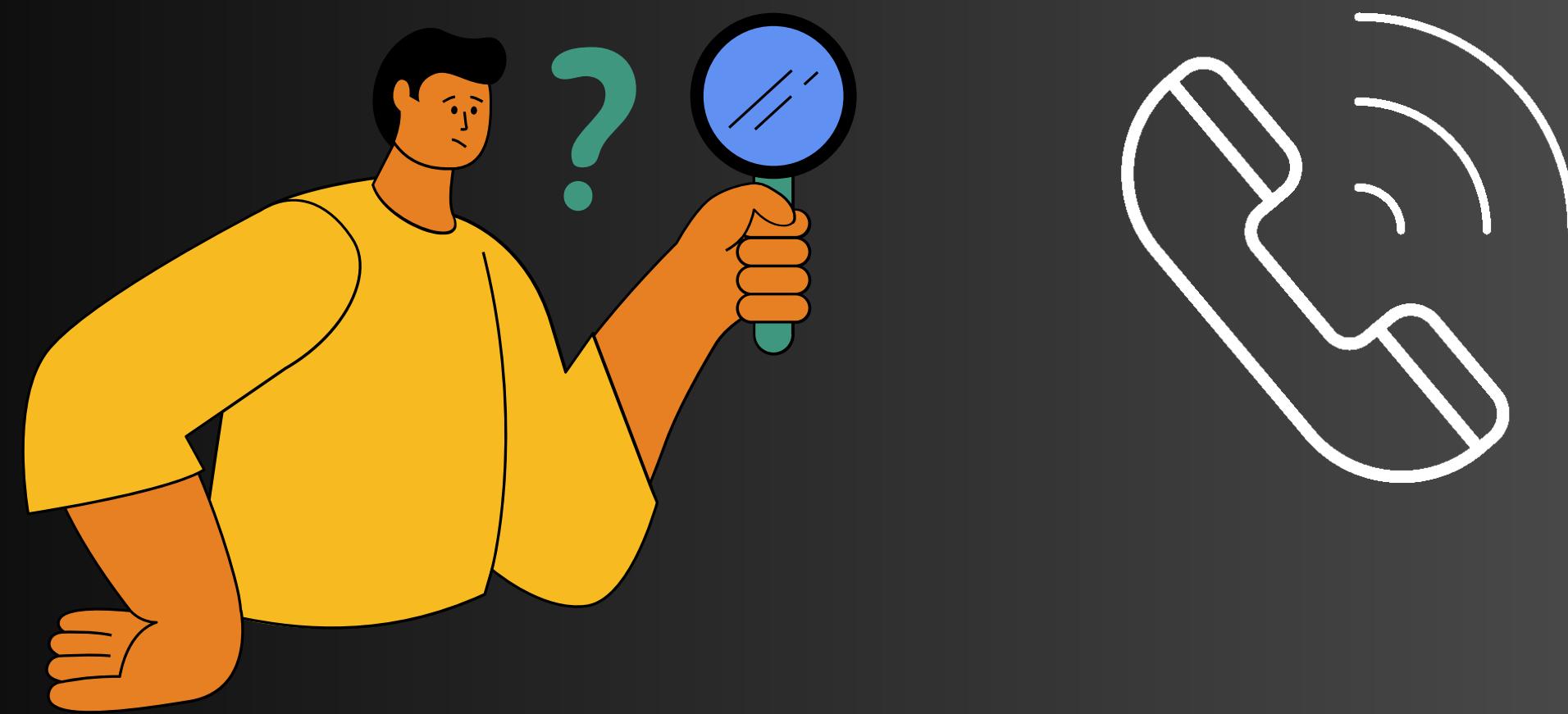
```
ALTER TABLE person  
ALTER COLUMN fname  
SET DEFAULT 'unknown';
```

# How to set NOT NULL?

```
ALTER TABLE person  
ALTER COLUMN fname  
SET NOT NULL;
```

**CHECK  
CONSTRAINT**

We want to make sure phone no. is  
atleast 10 digits...

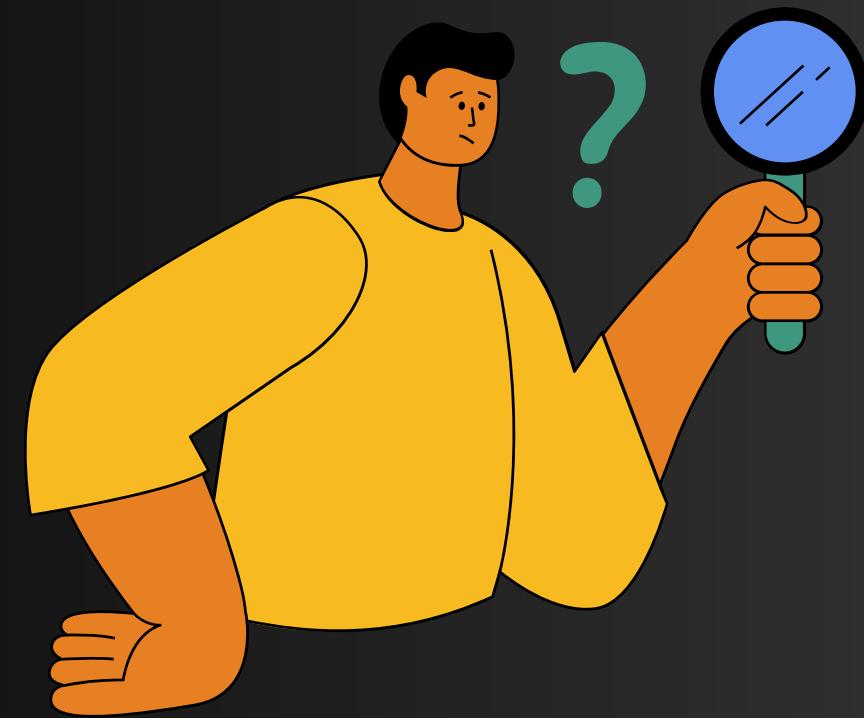


```
CREATE TABLE contacts(  
    name VARCHAR(50),  
    mob VARCHAR(15) UNIQUE CHECK (LENGTH(mob) >= 10)  
);
```

# NAMED CONSTRAINT

```
CREATE TABLE contacts(
    name VARCHAR(50),
    mob VARCHAR(15) UNIQUE,
    CONSTRAINT mob_no_less_than_10digits CHECK (LENGTH(mob) >= 10)
);
```

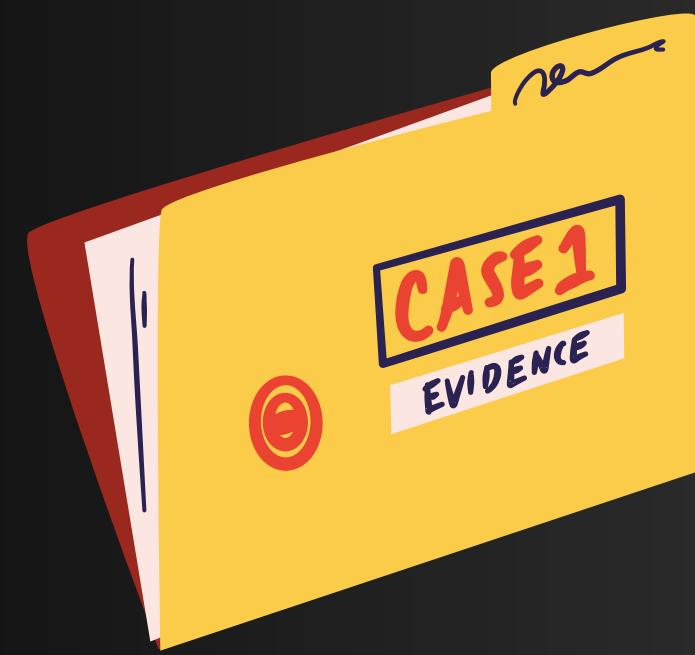
# How to Add or Drop Constraints?



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

```
ALTER TABLE contacts  
ADD CONSTRAINT mob_not_null CHECK (mob != null);
```

# Additional Topics



# CASE



fname	salary	Salary Category
Raju	37000	Low Salary
Sham	32000	Low Salary
Baburao	25000	Low Salary
Paul	45000	Low Salary
Alex	35000	Low Salary
Rick	65000	Higher Salary
Leena	25000	Low Salary
John	75000	Higher Salary
Alex	40000	Low Salary

```
select fname, salary,  
CASE  
    WHEN salary >= 50000 THEN 'High'  
    ELSE 'Low'  
END AS sal_cat  
FROM  
    employees;
```

```
SELECT fname, salary,  
CASE  
    WHEN salary >= 50000 THEN 'High'  
    WHEN salary >= 40000 AND  
        salary < 50000 THEN 'Mid'  
    ELSE 'Low'  
END AS sal_cat  
FROM  
employees;
```

# Task

fname	salary	bonus
Raj	50000 . 00	5000
Priya	45000 . 00	4500
Arjun	55000 . 00	5500
Suman	60000 . 00	6000
Kavita	47000 . 00	4700
Amit	52000 . 00	5200
Neha	48000 . 00	4800
Rahul	53000 . 00	5300
Anjali	61000 . 00	6100
Vijay	50000 . 00	5000

```
SELECT fname, salary,  
CASE  
    WHEN salary > 0 THEN Round(salary*.10)  
END AS bonus  
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 - 8

# RELATIONSHIP

**Employees**

**Salary**

**Attendance**

**Employees**

**requests**

**offices**

**task**

# Types of Relationship

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

1 : 1

## Employees

emp_id	name	dept
101	Raju	IT
102	Sham	Finance

## Employee Details

emp_id	addr	City	phone	title
101	CP	Delhi	909090909	Manager
102	Bhandup	Mumbai	902020200	Accountant

**1 : MANY**

## Employees

emp_id	name	dept
101	Raju	IT
102	Sham	Finance

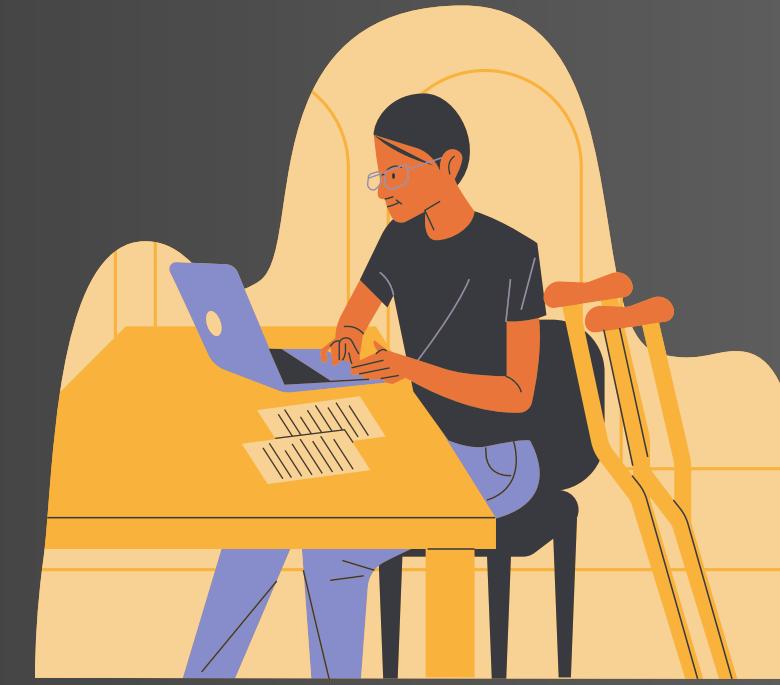
## Employee Task

emp_id	task_no	task_detail
101	TS-1	Opening account for Ram
102	TS-2	Closing account for Neru
101	TS-3	Loan sanction

**Many : Many**



**Books**



**Authors**



**Book A**



**Author A**



**Author B**



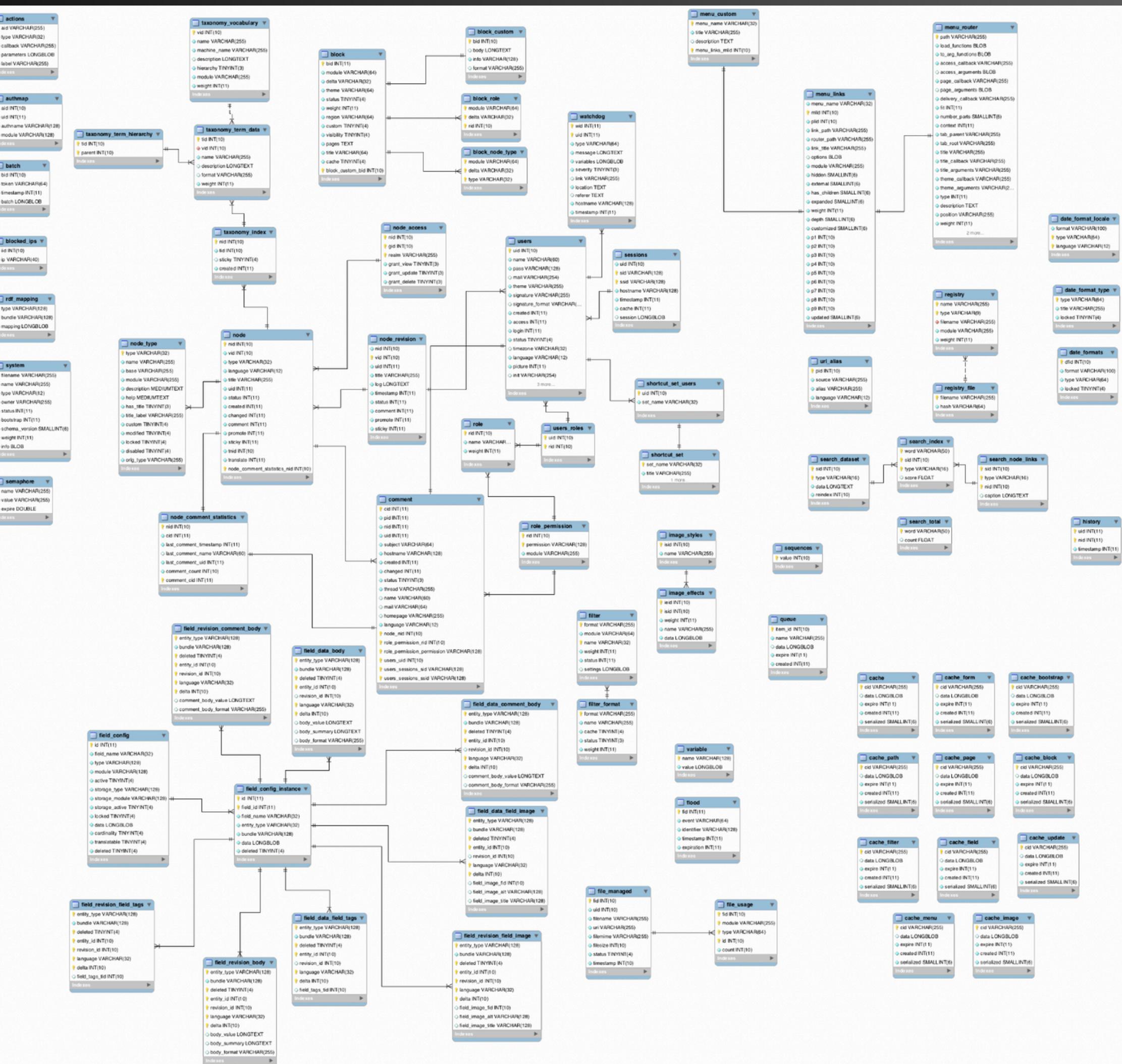
**Book B**



**Book C**



**Book D**



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





Suppose we need to store the following data

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

cust_name	email	order_daate	amount
Raju	raju@email.com	2023-05-15	200
Sham	sham@email.com	2023-04-28	500
Raju	raju@email.com	2023-05-14	1000
Baburao	babu@email.com	NULL	NULL
Sham	sham@email.com	2023-03-15	800

## **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

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

## Orders

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

# Foreign Key

## Customers

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

## Orders

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

## Primary Key

Customers

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

Orders

## Primary Key

## 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

Let's work practically  
with Foreign Key..

Customers

```
CREATE TABLE customers (
    cust_id SERIAL PRIMARY KEY,
    cust_name VARCHAR(100) NOT NULL
);
```

Orders

```
CREATE TABLE orders (
    ord_id SERIAL PRIMARY KEY,
    ord_date DATE NOT NULL,
    price NUMERIC NOT NULL,
    cust_id INTEGER NOT NULL,
    FOREIGN KEY (cust_id) REFERENCES
    customers (cust_id)
);
```

## Customers

```
INSERT INTO customers (cust_name)
VALUES
('Raju'), ('Sham'), ('Paul'), ('Alex');
```

## Orders

```
INSERT INTO orders (ord_date, cust_id, price)
VALUES
('2024-01-01', 1, 250.00),
('2024-01-15', 1, 300.00),
('2024-02-01', 2, 150.00),
('2024-03-01', 3, 450.00),
('2024-04-04', 2, 550.00);
```



# JOINS

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

## Customers

### Primary Key

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

## Orders

### Primary Key

### 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

# Types of Join

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

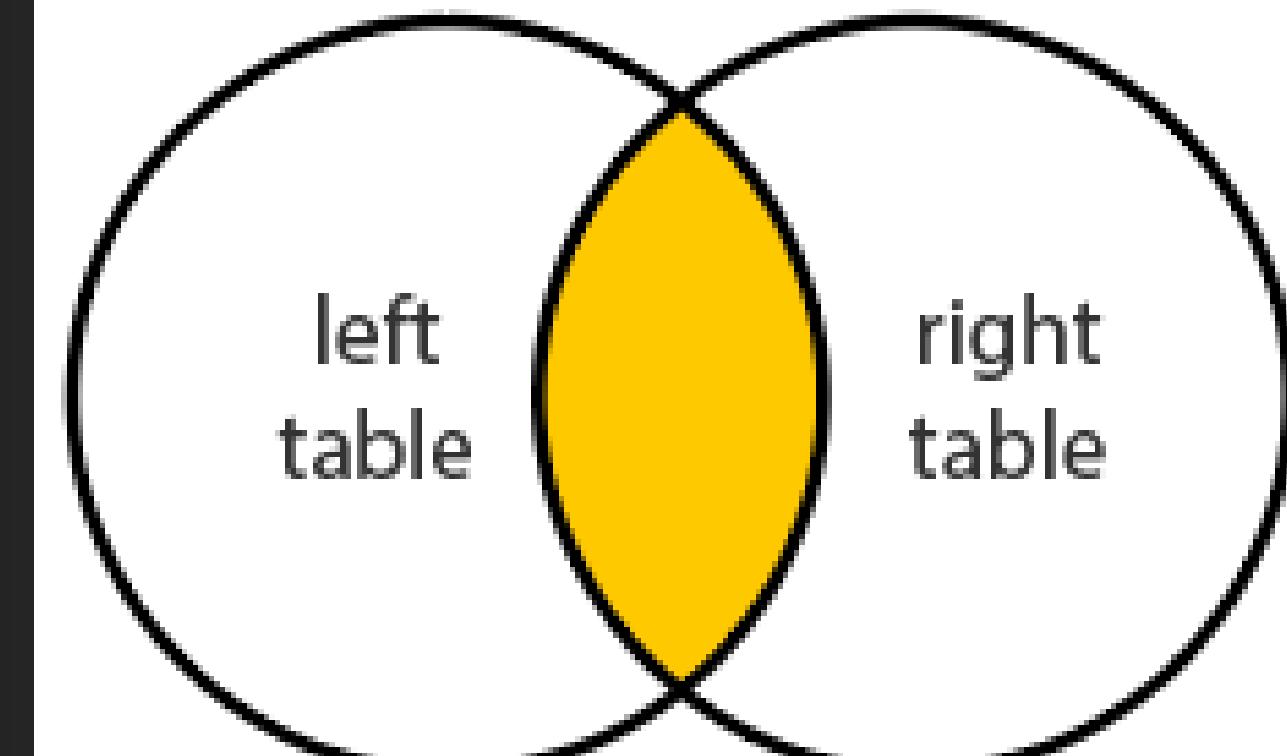
# Cross Join

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

# 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.

## INNER JOIN



```
SELECT * FROM customers
INNER JOIN orders
ON orders.cust_id=customers.cust_id;
```

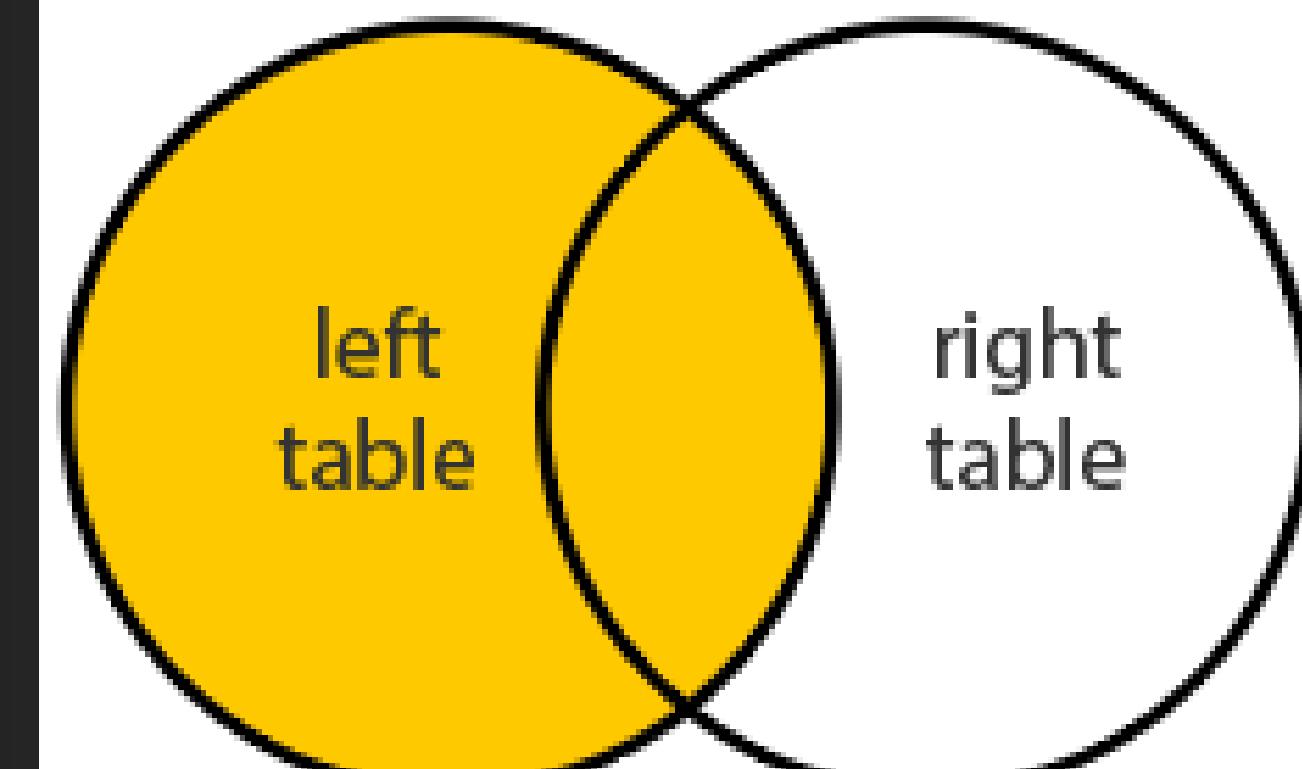
# Inner Join with Group By

```
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.

## LEFT JOIN

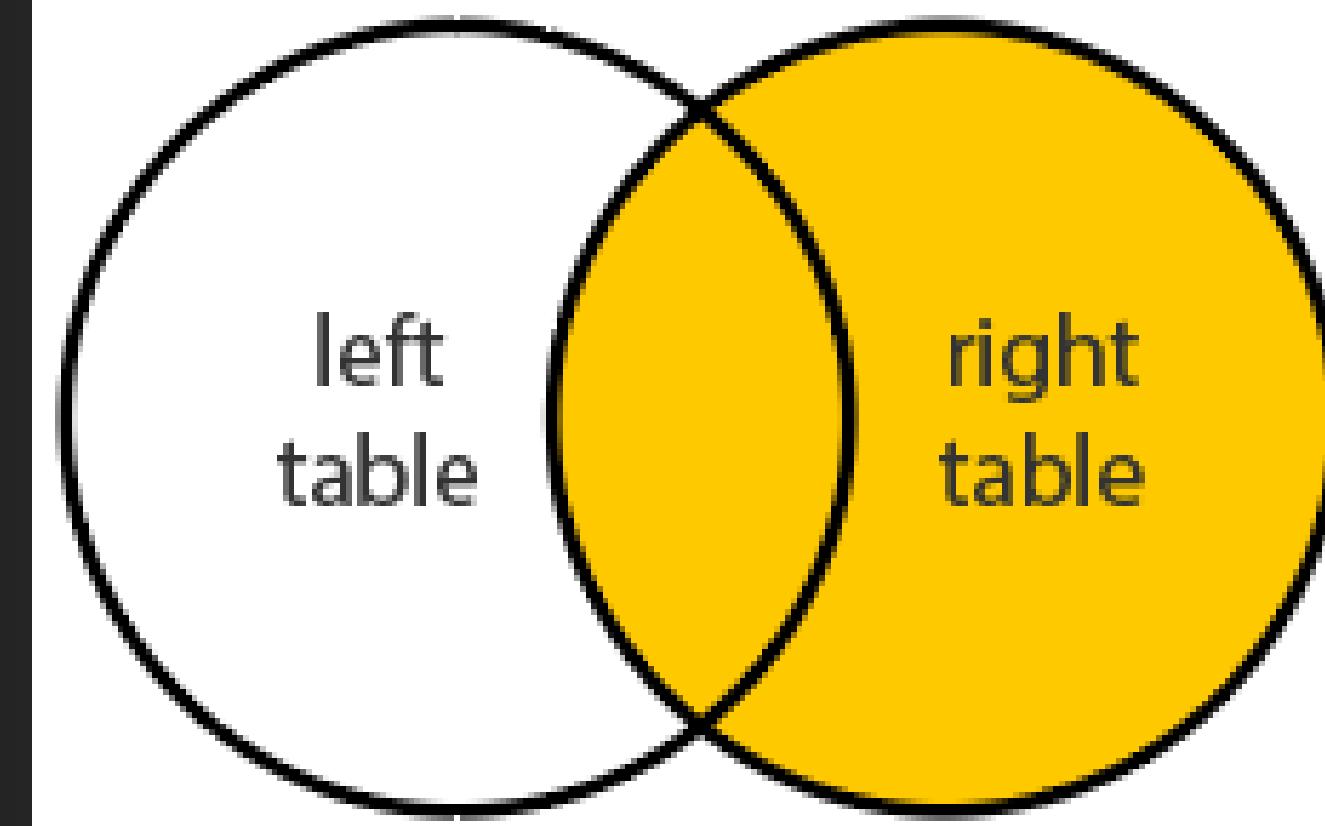


```
SELECT * FROM customers
LEFT JOIN orders
ON orders.cust_id=customers.cust_id;
```

# Right Join

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

## RIGHT JOIN



```
SELECT * FROM customers  
RIGHT JOIN orders  
ON orders.cust_id=customers.cust_id;
```

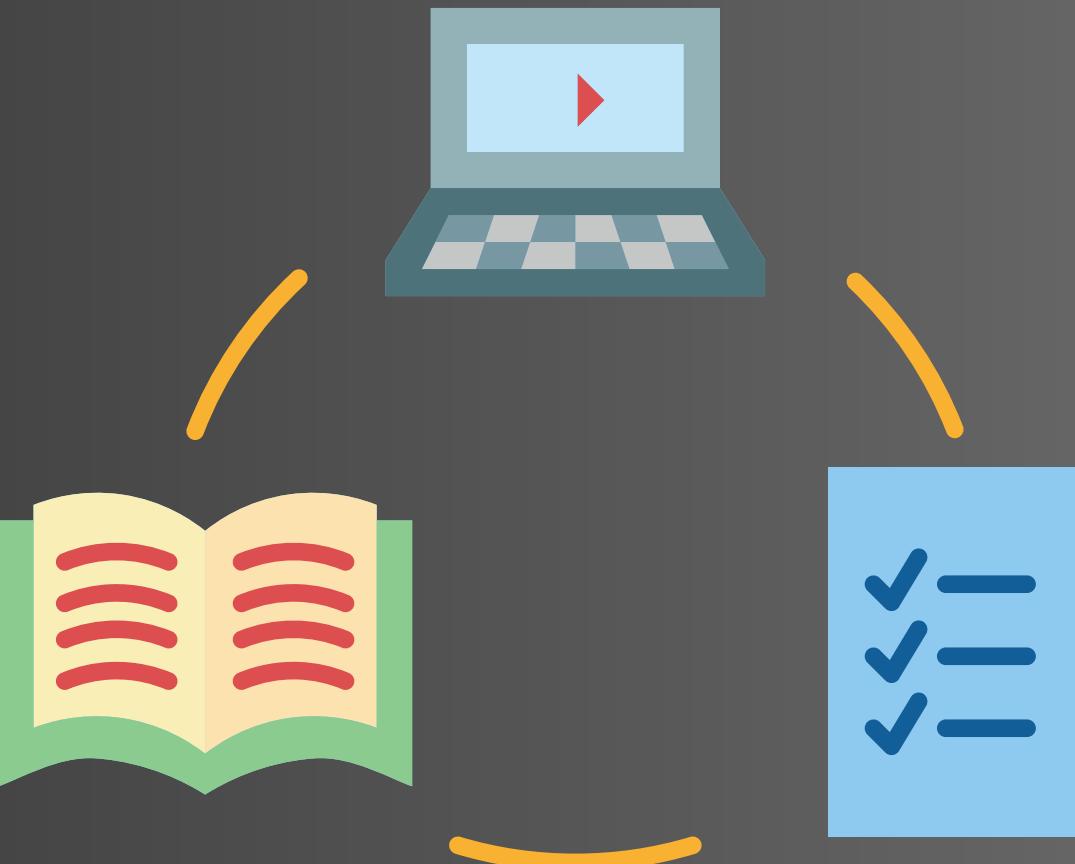
**Many : Many**

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

# **students**

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

# **courses**

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

# **student\_course**

- **student\_id**
- **course\_id**

## students

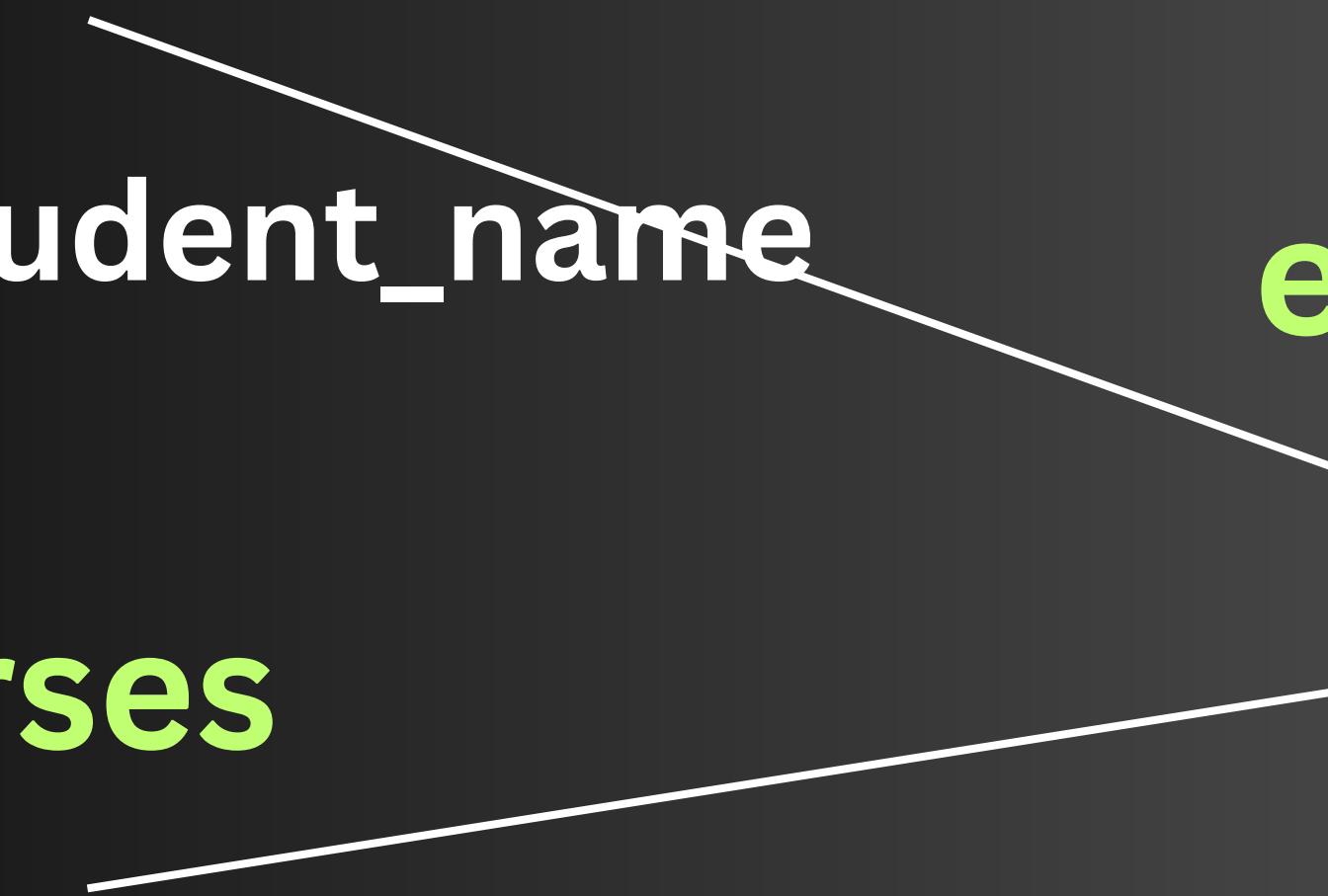
- id
- student\_name

## courses

- id
- course\_name
- fees

## enrollment

- student\_id
- course\_id





# TASK



**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

```
CREATE view billing_info AS
SELECT
    c.cust_name,
    o.ord_id,
    o.ord_date,
    p.p_name,
    oi.quantity,
    p.price,
    (oi.quantity * p.price) AS total_price
FROM
    customers c
JOIN
    orders o ON c.cust_id = o.cust_id
JOIN
    order_items oi ON o.ord_id = oi.ord_id
JOIN
    products p ON oi.p_id = p.p_id;
```

# HAVING clause

```
SELECT  
    p_name,  
    SUM(total_price) as Amount  
FROM billing_info  
GROUP BY p_name  
HAVING SUM(total_price) > 1500;
```

# GROUP BY ROLLUP

```
SELECT  
    COALESCE(p_name, 'Total'),  
    SUM(total_price) as Amount  
FROM billing_info  
    GROUP BY  
    ROLLUP(p_name) ORDER BY amount;
```

3	Keyboard	1600.00
4	Laptop	110000.00
5	Total	113350.00

# **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.

# **Types of STORED Routine**

- **STORED Procedure**
- **User defined Functions**

# **STORED Procedure**

# STORED PROCEDURE

Set of SQL statements and procedural logic  
that can perform operations such as  
inserting, updating, deleting, and querying  
data.

```
CREATE OR REPLACE PROCEDURE procedure_name (parameter_name parameter_type, ...)  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    -- procedural code here  
END;  
$$;
```

```
CREATE OR REPLACE PROCEDURE update_emp_salary(  
    p_employee_id INT,  
    p_new_salary NUMERIC  
)  
LANGUAGE plpgsql  
AS $$  
BEGIN  
    UPDATE employees  
    SET salary = p_new_salary  
    WHERE emp_id = p_employee_id;  
END;  
$$;
```

```
CREATE OR REPLACE PROCEDURE add_employee(
    p_fname VARCHAR,
    p_lname VARCHAR,
    p_email VARCHAR,
    p_dept VARCHAR,
    p_salary NUMERIC
)
LANGUAGE plpgsql
AS $$

BEGIN
    INSERT INTO employees (fname, lname, email, dept, salary)
    VALUES (p_fname, p_lname, p_email, p_dept, p_salary);
END;
$$;
```

# USER DEFINED FUNCTIONS

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

```
CREATE OR REPLACE FUNCTION function_name(parameters)
RETURNS return_type AS $$

BEGIN

    -- Function body (SQL statements)

    RETURN some_value;    -- For scalar functions

END;

$$ LANGUAGE plpgsql;
```

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

```
CREATE OR REPLACE FUNCTION dept_max_sal_emp1(dept_name VARCHAR)
RETURNS TABLE(emp_id INT, fname VARCHAR, salary NUMERIC)
AS $$

BEGIN

    RETURN QUERY
    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
        );
END;
$$ LANGUAGE plpgsql;
```

# 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.

# 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.

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

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

- We want to calculate the average salary per department and then find all employees whose salary is above the average salary of their department.

emp_id	fname	dept	salary	avg_salary
101	Raju	Loan	37000	34000.0000
102	Sham	Cash	32000	28500.0000
106	Rick	Account	65000	55000.0000
109	Alex	Loan	40000	34000.0000

```
WITH AvgSal AS (
    SELECT
        dept, AVG(salary) AS avg_salary
    FROM employees
    GROUP BY
        dept
)
SELECT
    e.emp_id, e.fname, e.dept, e.salary,
    a.avg_salary
FROM
    employees e
JOIN
    AvgSal a ON e.dept = a.dept
WHERE
    e.salary > a.avg_salary;
```

## Use Cases - 2

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

emp_id	fname	lname	desig	dept	salary
102	Sham	Mohan	Cashier	Cash	32000
105	Alex	Watt	Associate	Deposit	35000
106	Rick	Watt	Manager	Account	65000
108	John	Paul	Manager	IT	75000
109	Alex	Watt	Probation	Loan	40000

```
WITH HighestPaid AS (
    SELECT
        dept,
        MAX(salary) AS max_salary
    FROM
        employees
    GROUP BY
        dept
)
SELECT
    e.emp_id,
    e.fname,
    e.lname,
    e.desig,
    e.dept,
    e.salary
FROM
    employees e
JOIN
    HighestPaid h ON e.dept = h.dept AND e.salary = h.max_salary;
```

## Points:

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



# TRIGGERS

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

```
CREATE TRIGGER trigger_name
{ BEFORE | AFTER | INSTEAD OF } { INSERT | UPDATE | DELETE | TRUNCATE }
ON table_name
FOR EACH { ROW | STATEMENT }
EXECUTE FUNCTION trigger_function_name();
```

```
CREATE OR REPLACE FUNCTION trigger_function_name()
RETURNS TRIGGER AS $$

BEGIN
    -- Trigger logic here
    RETURN NEW;

END;

$$ LANGUAGE plpgsql;
```

# Use Case

Create a Trigger so that  
If we insert/update negative salary in a table,  
it will be triggered and set it to 0.

## Step 2: Create the Trigger Function

sql

 Copy code

```
CREATE OR REPLACE FUNCTION check_salary()
RETURNS TRIGGER AS $$

BEGIN

    IF NEW.salary < 0 THEN
        NEW.salary := 0;
    END IF;
    RETURN NEW;
END;

$$ LANGUAGE plpgsql;
```

## Step 3: Create the Trigger

sql

 Copy code

```
CREATE TRIGGER before_insert_salary
BEFORE INSERT ON employees
FOR EACH ROW
EXECUTE FUNCTION check_salary();
```



# CASCADE ON DELETE

## Customers

### Primary Key

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

## Orders

### Primary Key

### 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 AUTO_INCREMENT PRIMARY KEY,
    date DATE,
    amount DECIMAL(10, 2),
    cust_id INT,
    FOREIGN KEY (cust_id) REFERENCES customers(cust_id) ON DELETE CASCADE
)
```