What we'll cover today?

✅ Section 1: SQL Basics – Getting Started

🔹 **Topics:**

- What is SQL & where is it used in real-world businesses
- Understanding Structured Databases and SQL Data Types
- Common SQL Commands: DDL, DML, DQL
- SQL Query Structure: SELECT, FROM, WHERE
- Filtering data using conditions: =, !=, <, >, AND, OR

✅ Section 2: Aggregations, Grouping, Sorting, Limiting & Overview of Suquery

🔹 **Topics:**

- Aggregate functions: COUNT(), SUM(), AVG(), MIN(), MAX()
- Grouping records using GROUP BY
- Filtering grouped data with HAVING
- Difference between WHERE and HAVING
- Sorting data using ORDER BY with ASC and DESC

- Limiting rows in output using LIMIT
- What are Subqueries?

✅ Section 3: Test Your Knowledge Section

🔹 **Topics:**

- **Solve Top 5 SQL - Real Interview Question to cover Section 1&2**

✅ Section 4: Understanding JOINs, Subqueries & CASE Statements

🔹 **Topics:**

- INNER JOIN – Combining data from two tables with matching keys
- LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN – Key differences and use cases
- When to use different types of joins in real-life scenarios
- Writing subqueries inside SELECT, FROM, and WHERE
- Using CASE statements for conditional logic

✅ Section 5: Solve Company based Interview Q&A (Take Home Tasks)

🔹 **Topics:**

- Solve more than 10 SQL interview Question & Doubt clearing session

---

✅ Section 6: Window Functions – RANK, DENSE_RANK, ROW_NUMBER, LAG, LEAD

🔹 Definitions of all 5 major window functions

🔹 Ranking students based on marks – RANK vs DENSE_RANK vs ROW_NUMBER

🔹 Comparing current row with previous/next using LAG & LEAD

🔹 Practice Questions using Window Functions

✅ Section 7: Common Table Expressions (CTEs)

🔹 What is a CTE and when to use it

🔹 Writing step-by-step layered queries using WITH clause

🔹 Practice Examples using CTEs

✅ Section 1: SQL Basics – Getting Started

🔹 Introduction to SQL & Real-World Use Cases

SQL stands for **Structured Query Language**. It is used to manage and query structured data stored in relational databases.

Just like **you and I are talking in Hindi / English**, we use a language to **communicate**.

Similarly, **SQL (Structured Query Language)** is the **language we use to talk to databases**.

It works only with **Structured Databases** — meaning data is stored in rows and columns like tables. And, this structure is known as **RDBMS** (Relational Database Management System).

Example: Imagine you run an **online store** with thousands of customers and daily transactions. To understand how your business is doing, you may want to:

- Find out the top-selling products.
- Check which customers bought the most.
- Update customer information.
- Delete discontinued products.

All of this can be done easily using **SQL**.

🔹 Why Do We Need a Structured Database?

Think of tracking customer orders in WhatsApp or a messy notepad. It becomes hard to **search**, **compare**, or **update** any information.

That's where a **structured database** helps — it organizes data in **rows and columns**. Let's visualize it with real life example -

Here’s a simple table from an online store:

Sample Table: customers

customer_id	name	email	city	signup_date
1	Priya	priya@gmail.com	Mumbai	2023-06-01
2	Rahul	rahul@hotmail.com	Delhi	2023-07-12
3	Ananya	ananya@yahoo.com	Bangalore	2023-08-05
4	Siddharth	sid@store.com	Kolkata	2023-08-20

SQL makes it easy to:

- ✅ **Search**: "Find all customers from Mumbai"
- ✅ **Update**: "Update email of customer_id = 3"
- ✅ **Compare**: "See which city has the most customers"

🔹 SQL Data Types

Data types define what kind of data can be stored in each column.

Data Type	Description	Example
VARCHAR	Stores text/characters	‘Priya’, 'Mumbai'
INTEGER	Stores whole numbers	2, 1, 4
DATE	Stores calendar dates	2023-05-01
BOOLEAN	Stores True/False values	TRUE, FALSE

🔹 Types of SQL Commands

SQL is categorized into different command types based on what they do:

✏️ 1. DDL (Data Definition Language)

Used to define and modify the structure of database objects (tables, schema, etc.)

- CREATE – creates a new table or database
- ALTER – modifies an existing object
- DROP – deletes tables or databases
- TRUNCATE – removes all rows from a table quickly

📝 2. DML (Data Manipulation Language)

Used to manipulate data stored in database objects.

- INSERT – adds new records to a table
- UPDATE – updates existing records
- DELETE – removes records

🌐 3. DQL (Data Query Language)

Used to query data from the database.

- SELECT – fetches data from one or more tables

🔑 4. DCL (Data Control Language) *(optional for now)*

- GRANT, REVOKE – controls access to data (used more in admin roles)

✅ SQL Commands – End-to-End Example

🧾 Use Case: Managing an Online Food Delivery Service

We'll work with a simple table named customers to demonstrate major SQL commands step-by-step.

🔹 Table Structure:

Table Name: customers
Columns:

- customer_id (INT)
- name (VARCHAR)
- city (VARCHAR)

🔹 1. CREATE – Create the customers table

```
CREATE TABLE customers (

  customer_id INT PRIMARY KEY,
  name VARCHAR(100),
  city VARCHAR(50)
);
```

🔹 2. INSERT – Add new customers

```
INSERT INTO customers (customer_id, name, city) VALUES
(1, 'Aman', 'Delhi'),
(2, 'Priya', 'Mumbai');
```

🔹 3. UPDATE – Customer moved to another city

```
UPDATE customers
SET city = 'Bangalore'
WHERE customer_id = 2;
```

🔹 4. ALTER – Add a new column for phone number

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

🔹 5. DELETE – Remove a customer who deleted their account

```
DELETE FROM customers
```

WHERE customer_id = 1;

🔹 6. SELECT – View all current customers

SELECT * FROM customers;

🔹 7. TRUNCATE – Clear all customer data (e.g., for system reset)

TRUNCATE TABLE customers;

🔹 8. DROP – Permanently remove the table

DROP TABLE customers;

🔹 What are SELECT, FROM, WHERE?

Let’s say you have a big table full of information — names, age, salary, city, etc.

Here’s how these basic keywords work:

Keyword	Meaning
SELECT	What data do you want? (Columns)
FROM	From which table?
WHERE	Apply conditions to filter data

Example in simple words:
“SELECT name, salary FROM employees WHERE city = 'Mumbai';”
Means → *Get names and salaries of employees who live in Mumbai*

🔹 Sample Table to Create & Insert

```
CREATE TABLE employees (
  emp_id INT,
  name VARCHAR(50),
  city VARCHAR(50),
  salary INT
);

INSERT INTO employees (emp_id, name, city, salary) VALUES
(1, 'Ankit', 'Delhi', 55000),
(2, 'Riya', 'Mumbai', 62000),
(3, 'Sohail', 'Delhi', 48000),
(4, 'Priya', 'Mumbai', 75000),
(5, 'Raj', 'Bangalore', 53000),
(6, 'Meena', 'Delhi', 61000),
(7, 'Karan', 'Bangalore', 72000),
(8, 'Sneha', 'Mumbai', 57000),
(9, 'Amit', 'Delhi', 65000),
(10, 'Neha', 'Bangalore', 49000),
(11, 'Pooja', 'Kolkata', 56000),
(12, 'Aditya', 'Kolkata', 58000),
(13, 'Farhan', 'Mumbai', 54000),
(14, 'Rohit', 'Delhi', 60000),
(15, 'Divya', 'Bangalore', 71000);
```

🔹 Basic SQL Operators to Know

These help in writing conditions inside WHERE

Operator	Use
=	Equal to

!= or <>	Not equal to
<, >, <=, >=	Less than, Greater than, etc.
AND	Combine 2 or more conditions
OR	At least one condition is true
IN	Match values from a list
BETWEEN	Match within a range
LIKE	Pattern match (used with %)

✅ Practice Questions – Based on Section 1 - For Understanding

Q1. Get names and cities of all employees who earn more than ₹50,000

Q2. Find all employees who live in either 'Delhi' or 'Mumbai'

Q3. Show employees whose names start with the letter 'P'

✅ **Section 2: Aggregations, Grouping, Sorting & Limiting**

🔹 **What are Aggregate Functions?**

Definition: Aggregate functions perform a calculation on a set of values and return a single result.

They help you summarize data, like finding totals, averages, counts, etc.

Example:
Imagine you're an HR Manager, and you have a list of 1000 employees in Excel.
 You want to know:

- How many total employees? → use COUNT()
- What's the average salary in your company? → use AVG()
- What's the highest salary? → use MAX()

In SQL, we use aggregate functions to answer such questions with just a few lines of code.

🔹 Common Aggregate Functions

Function	What it does
COUNT()	Counts number of rows
SUM()	Adds all values
AVG()	Calculates average value
MIN() / MAX()	Finds lowest / highest value

Example Queries -
How many total employees are there ? - Select count(emp_id) from employees

What's the average salary in your company? → Select Avg(salary) from employees

---

🔹 What is GROUP BY?

Definition: GROUP BY is used to group rows that have the same values into summary rows. It is often used with aggregate functions.

Example:
Let's say your employee table has a city column. As a manager, you want to know:

> "How many employees are there in each city?"

So instead of counting manually, SQL groups employees based on city and gives a count for each group.

It's like making a pivot table in Excel — you group data by one column and summarize it with another.

Query - Select city, count(emp_id) from employees

Group By city

🔹 What is HAVING?

Definition: HAVING is used to filter groups based on the result of aggregate functions.

Example:
You got the total number of employees in each city using GROUP BY. But now you only want to see:

> "Cities where the number of employees is more than 2."

Since you're filtering the result **after grouping**, you must use HAVING.

Query -

Select city, count(emp_id) from employees

Group by city

Having count(emp_id) > 2

🔹 What is ORDER BY?

Definition: ORDER BY is used to sort the result based on one or more columns.

Example:

Suppose, after finding the total employees in each city, you want to list cities **from highest to lowest number of employees**.

This is where ORDER BY helps:

Query - Select city, count(emp_id) from employees

group by city

order by count(emp_id) desc

🔹 What is LIMIT?

Definition: LIMIT is used to restrict the number of rows returned in the output.

Example:

Once you sort the cities by total employees, you may want to show **only the top 2 cities**. Use LIMIT:

Query -

SELECT city, COUNT(*) AS total_employees

FROM employees

GROUP BY city

ORDER BY total_employees DESC

LIMIT 2;

🔹 **What are Subqueries?**

Definition: A Subquery is a query nested inside another SQL query. It can be used inside SELECT, FROM, or WHERE clauses.

Example: "Show all employees whose salary is more than the average salary of all employees."

To solve this, you'll first calculate the average salary — and then compare each employee's salary to that. That inner query is your subquery.

Query -

```
SELECT name, city, salary

FROM employees

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

✅ Section 3: Test Your Knowledge Section

Q1. Write a query to get the second highest salary from the employees table.

```
Select max(salary) as second_highest_salary
From employees
Where salary < (Select max(salary) from employees)
```

Q2. You have a sales table with columns (region, month, revenue). Find the region that had the highest average revenue across all months.

```
CREATE TABLE sales (
  region VARCHAR(50),
  month VARCHAR(20),
  revenue INT
);
```

```
INSERT INTO sales (region, month, revenue) VALUES
('North', 'January', 100000),
('South', 'January', 85000),
('East', 'January', 92000),
('North', 'February', 110000),
('South', 'February', 87000),
('East', 'February', 95000);
```

Query -

```
SELECT region, AVG(revenue) AS avg_revenue
FROM sales
GROUP BY region
ORDER BY AVG(revenue) DESC
LIMIT 1;
```

Q3. Given a students table with (student_id, course_id, score), write a query to find all course_ids where more than 10 students have scored above 80.

```
CREATE TABLE students (
  student_id INT,
  course_id VARCHAR(10),
  score INT
);

INSERT INTO students (student_id, course_id, score) VALUES
(1, 'C101', 85),
(2, 'C101', 91),
(3, 'C102', 76),
(4, 'C101', 88),
(5, 'C102', 93),
(6, 'C103', 67),
(7, 'C101', 81),
(8, 'C103', 89),
(9, 'C101', 82),
(10, 'C102', 84),
(11, 'C101', 90);
```

Query -

```
SELECT course_id
FROM students
WHERE score > 80
GROUP BY course_id
HAVING COUNT(*) > 10;
```

Q4. In a products table with (category, price), write a query to find the top 3 categories with the highest total price value.

```
CREATE TABLE products (
  product_id INT,
  category VARCHAR(50),
  price INT
);

INSERT INTO products (product_id, category, price) VALUES
(1, 'Electronics', 45000),
(2, 'Home', 15000),
(3, 'Electronics', 30000),
(4, 'Fashion', 7000),
(5, 'Home', 12000),
(6, 'Fashion', 9000),
(7, 'Electronics', 60000);
```

Query -

```
SELECT category, SUM(price) AS total_value
FROM products
GROUP BY category
ORDER BY total_value DESC
LIMIT 3;
```

Q5. What is the difference between COUNT(*) and COUNT(column_name)?

COUNT(*):

- Counts **all rows** in the table — including rows where the column values are NULL.
- It doesn't care about specific columns; it just counts rows.

COUNT(column_name):

- Counts only **non-NULL values** in the specified column.
- If the column contains NULL values, those rows will be **excluded** from the count.

Example:

Suppose this is your employees table

emp_id	name	salary
1	Ankit	55000
2	NULL	51000
3	Sohail	48000

- SELECT COUNT(*) FROM employees; → Returns **3**

- SELECT COUNT(name) FROM employees; → Returns **2**

Because the second row has a NULL salary, and COUNT(salary) ignores it.

✅ Section 4: Understanding JOINs, Subqueries & CASE Statements

🔹 What is a JOIN?

Definition: A JOIN is used to combine rows from two or more tables based on a related column between them. It helps you answer questions that involve pulling related data from different tables.

🔹 **Types of JOINs**

INNER JOIN – returns only the rows with matching keys in both tables.

LEFT JOIN – returns all rows from the left table and matched rows from the right table.

RIGHT JOIN – returns all rows from the right table and matched rows from the left table.

FULL OUTER JOIN – returns all rows /basically right join + left join.

Example: Imagine you have two tables - emp_1 & emp_2

emp_1

emp_id
1
2
3
5
7
8

emp_2

emp_id
1
2
4
6
7
8

This requires you to fetch data from **both** tables — that's where JOIN comes in.

Let's write query for all these types

Create Syntax for Both the tables -

CREATE TABLE emp_1 (emp_id INT);

INSERT INTO emp_1 (emp_id) VALUES (1), (2), (3), (5), (7), (8);

CREATE TABLE emp_2 (emp_id INT);

INSERT INTO emp_2 (emp_id) VALUES (1), (2), (4), (6), (7), (8);

INNER JOIN

Query - select * **from** emp_1 as e1 **inner join** emp_2 **as** e2 **on** e1.emp_id = e2.emp_id

LEFT JOIN

Query - select * **from** emp_1 as e1 **left join** emp_2 **as** e2 **on** e1.emp_id = e2.emp_id

RIGHT JOIN

Query - select * **from** emp_1 as e1 **right join** emp_2 **as** e2 **on** e1.emp_id = e2.emp_id

FULL OUTER JOIN

Query - select * **from** emp_1 as e1 **full outer join** emp_2 **as** e2 **on** e1.emp_id = e2.emp_id

---

🔹 **What is a CASE Statement?**

Definition: CASE is used to apply conditional logic in SQL. It works like IF-ELSE in programming.

Example:

"Write a query to add a column in the result that says `'High Earner'` if salary is more than 60,000 — otherwise show `'Regular'`"

That's exactly where you use CASE.

Let's write query for it -

SELECT name, city, salary,

CASE WHEN salary > 60000 THEN 'High Earner'

ELSE 'Regular' END AS earning_status

FROM employees;

✅ Section 5: Take Home Task-

Practice Session For - Company based Interview Q&A

✅ **1.** Find Developers Who Earn More Than Their Leads

Write a solution to find the developers who earn more than their team leads.

Return the result table in **any order**.

dev_id	dev_name	salary	lead_id
1	Aakash	70000	3
2	Meera	80000	4
3	Rahul	60000	NULL
4	Tanya	90000	NULL

Create & Insert Statement-

```
CREATE TABLE developers (

   dev_id     INT PRIMARY KEY,

   dev_name   VARCHAR(50),

   salary     INT,
    lead_id    INT
);
INSERT INTO developers (dev_id, dev_name, salary, lead_id) VALUES
(1, 'Aakash', 70000, 3),
(2, 'Meera', 80000, 4),
(3, 'Rahul', 60000, NULL),
(4, 'Tanya', 90000, NULL);
```

Solution:-

```
SELECT d.dev_id, d.dev_name, d.salary, d.lead_id
FROM developers d
JOIN developers l
  ON d.lead_id = l.dev_id
WHERE d.salary > l.salary;
```

✅ 2. List Subscribers Who Never Watched Any Video

Write a solution to find all subscribers who never streamed any content.

Return the result table in **any order**.

📦 **Tables:**

Subscribers Table

id	**subscriber_name**
1	Arjun

2	Nisha
3	Ravi
4	Priya

StreamingLogs Table

id	sub_id
1	3
2	1

Output:

Subscribers
Nisha
Priya

🧠 **Explanation:** Nisha and Priya never streamed any content.

✅ 3. List Dates With Higher COVID Cases Than Previous Day

Write a solution to find all dates with more reported cases than the previous day.

Return the result table in **any order**.

📦 **Table:**

id	reportDate	case_count
1	2023-01-01	100
2	2023-01-02	250
3	2023-01-03	200

4	2023-01-04	300

Output:

id
2
4

🧠 **Explanation:** Case count on Day 2 and Day 4 is higher than the previous day.

✅ 4. Find Courses With Zero Enrollment

Write a query to return course IDs of all online courses that have zero enrollments.

Return result in **ascending order**.

📦 **Tables:**

Courses Table

course_id	course_title
101	Data Science Basics
102	Python for Beginners
103	SQL Mastery

Enrollments Table

user_id	course_id	enrolled_date
1	101	2022-06-01 0:00:00
2	102	2022-06-03 0:00:00

Output:

course_id
103

🧠 **Explanation:** No one enrolled in the course with ID 103.

✅ 5. Flip 'Active' and 'Inactive' Status in a Single Update

Write a solution to flip all Active values to Inactive and vice versa using a single update query.

Do not use intermediate tables or multiple updates.

📦 **Table:**

Vendors Table

id	vendor_name	status	rating
1	Alpha	Active	4.5
2	Beta	Inactive	3.8
3	Gamma	Active	4.9
4	Delta	Inactive	2.5

Output:

id	vendor_name	status	rating
1	Alpha	Inactive	4.5
2	Beta	Active	3.8
3	Gamma	Inactive	4.9
4	Delta	Active	2.5

🧠 **Explanation:** Status flipped using conditional logic inside a single UPDATE.

✅ Section 6: Window Functions – RANK, DENSE_RANK, ROW_NUMBER, LAG, LEAD

🔹 What are Window Functions?

Definition:
A **window function** is like a special calculator in SQL that can **look at a group of rows (a "window" of data)** while still keeping each individual row in the result.

It doesn't collapse rows like `GROUP BY` does — instead, it adds extra information **side-by-side** with your existing data.

Imagine you're in a classroom with 10 students:

- Normally, if you ask "What's the class average score?" — you'll get **one answer for the whole class** (like `GROUP BY`).

- But with a window function, you can say — "For each student, show their score **and** the class average."
  Everyone gets to see their own marks *plus* the class average in the same row.

🔹 Types of Window Functions – One-Line Definitions

- **RANK()** – Assigns a rank to each row within a partition, with the **same rank for ties** and **skips in ranking** after ties.

- **DENSE_RANK()** – Assigns a rank to each row within a partition, with the **same rank for ties** and **no skips** in ranking.

- **ROW_NUMBER()** – Assigns a **unique sequential number** to each row within a partition, even if values tie.

- **LAG(column, offset, default)** – Returns the value from the **previous row** in the same partition, based on ordering.
- **LEAD(column, offset, default)** – Returns the value from the **next row** in the same partition, based on ordering.

🔹 Example

Imagine you are a teacher preparing a **rank list of students** in each class based on marks. You want to:

- Assign positions in the class.
- Decide how to handle ties (skip rank numbers or not).
- Optionally assign a unique number regardless of ties.
- Compare a student's marks with the **previous or next exam**.

🔹 Sample Table – Students

```
CREATE TABLE students (

  student_id INT,

  name VARCHAR(50),

  class VARCHAR(10),

  marks INT

);

INSERT INTO students (student_id, name, class, marks) VALUES

(1, 'Aman',    '10A', 95),

(2, 'Priya',   '10A', 88),

(3, 'Rohit',   '10A', 95),
```

(4, 'Meera',   '10A', 80),

(5, 'Karan',   '10A', 75),

(1, 'Riya',   '10B', 95),

(2, 'Siya',   '10B', 88),

(3, 'Jiya',   '10B', 95),

(4, 'Ram',   '10B', 80),

(5, 'Sam',   '10B', 75);

RANK() Example

SELECT name, class, marks,

    RANK() OVER (PARTITION BY class ORDER BY marks DESC) AS rnk

FROM students;

Explanation:

- Aman and Rohit both have 95 marks → **Rank 1**.
- Priya gets **Rank 3** (Rank 2 is skipped)

name	class	marks	rnk
Aman	10A	95	1
Rohit	10A	95	1
Priya	10A	88	3
Meera	10A	80	4
Karan	10A	75	5

DENSE_RANK() Example

```
SELECT name, class, marks,

    DENSE_RANK() OVER (PARTITION BY class ORDER BY marks DESC) AS dense_rnk

FROM students;
```

Explanation:

- Aman and Rohit both have 95 marks → **Rank 1**.
- Priya gets **Rank 2** (no skip).

name	class	marks	dense_rnk
Aman	10A	95	1
Rohit	10A	95	1
Priya	10A	88	2
Meera	10A	80	3
Karan	10A	75	4

ROW_NUMBER() Example

```
SELECT name, class, marks,

    ROW_NUMBER() OVER (PARTITION BY class ORDER BY marks DESC) AS row_num

FROM students;
```

Explanation:

- Every student gets a **unique number**, even if marks are the same

name	class	marks	row_num
Aman	10A	95	1
Rohit	10A	95	2
Priya	10A	88	3
Meera	10A	80	4
Karan	10A	75	5

LAG & LEAD Example – Marks Change

```
CREATE TABLE exam_scores (
  student_id INT,
  exam_month DATE,
  marks INT
);


INSERT INTO exam_scores VALUES
(1, '2024-01-01', 80),
(1, '2024-02-01', 85),
(1, '2024-03-01', 78);
```

Query-

```
SELECT student_id, exam_month, marks,
```

    LAG(marks, 1, 0) OVER (PARTITION BY student_id ORDER BY exam_month) AS prev_marks,

    marks - LAG(marks, 1, 0) OVER (PARTITION BY student_id ORDER BY exam_month) AS diff,

    LEAD(marks, 1, 0) OVER (PARTITION BY student_id ORDER BY exam_month) AS next_marks

FROM exam_scores;

✅ Practice Questions – Window Functions

Q1. From the students table, list the **top 2 students by marks in each class**.

```
WITH ranked AS (

  SELECT name, class, marks,

         DENSE_RANK() OVER (PARTITION BY class ORDER BY marks DESC) AS rnk

  FROM students

)

SELECT * FROM ranked WHERE rnk <= 2;
```

Q2. ¬.

```
SELECT student_id, exam_month, marks,

       marks - LAG(marks, 1, 0) OVER (PARTITION BY student_id ORDER BY exam_month) AS change

FROM exam_scores;
```

✅ Section 7: Common Table Expressions (CTEs)

🔹 What is a CTE?

Definition:
A CTE is a **temporary named result set** defined using WITH that exists **only within the query**. It is mainly used to break down complex queries into **smaller, readable steps**.

🔹 Example

Imagine you have sales data for the year and you want to:

1. Get **total sales per salesperson**.
2. Find the **top salesperson in each region**.

You can do this step-by-step using CTEs.

🔹 Syntax:

```
WITH cte_name AS (

  SELECT ...

)

SELECT * FROM cte_name;
```

🔹 Example – Top Salesperson Per Region

```
CREATE TABLE sales (

  emp_id INT,

  emp_name VARCHAR(50),

  region VARCHAR(20),

  revenue INT
);

INSERT INTO sales VALUES

(1, 'Aman',  'North', 120000),

(2, 'Priya', 'North', 110000),

(3, 'Rohit', 'South', 150000),

(4, 'Meera', 'South', 150000),

(5, 'Karan', 'East',  90000);
```

Query -

```
WITH emp_totals AS (

   SELECT emp_id, emp_name, region, SUM(revenue) AS total_rev

   FROM sales

   GROUP BY emp_id, emp_name, region

),

ranked AS (

   SELECT *,

          DENSE_RANK() OVER (PARTITION BY region ORDER BY total_rev
DESC) AS rnk

   FROM emp_totals

)

SELECT emp_name, region, total_rev

FROM ranked

WHERE rnk = 1;
```

Stay Connected ❤️

📩 **Email for Collaboration:** shakrashamim09@gmail.com

📌 **LinkedIn:** https://www.linkedin.com/in/shakra-shamim/

📌 **YouTube:** https://www.youtube.com/@ShakraShamim

📌 **Instagram:** https://www.instagram.com/shakra.shamim