COURSE NAME

ADVANCED SQL

GROUP BY, HAVING, ANY, ALL, and EXISTS

Clause	When to Use
`GROUP BY`	To aggregate data based on one or more columns (e.g., find total salary per department).
`HAVING`	To filter **aggregated** results after `GROUP BY` (e.g., show only departments with total salary > 20,000).
`ANY`	To compare a value with **any value** from a subquery (e.g., find employees with a salary higher than at least one employee in another department).
`ALL`	To compare a value with **all values** from a subquery (e.g., find employees whose salary is higher than everyone in another department).
`EXISTS`	To check **if a subquery returns any rows** (e.g., find departments that have at least one employee).

EmployeeID	Name	Department	Salary
2	Bob	HR	7000
3	Charlie	HR	6000
1	Alice	HR	5000
6	Frank	IT	9000
4	David	IT	8000
5	Eve	IT	7500
8	Hank	Finance	7200
7	Grace	Finance	6500

1. `GROUP BY` with Aggregation (`SUM()`, `AVG()`, `COUNT()`)

```sql
SELECT Department, COUNT(*) AS EmployeeCount, SUM(Salary)
AS TotalSalary, AVG(Salary) AS AvgSalary
FROM Employees
GROUP BY Department;
```

Result

Department	EmployeeCount	TotalSalary	AvgSalary
HR	3	18500	6166.67
IT	3	24500	8166.67
Finance	2	13700	6850.00

Explanation:

- Groups employees by `Department` and calculates:
  - `COUNT(*)` → Number of employees per department.
  - `SUM(Salary)` → Total salary per department.
  - `AVG(Salary)` → Average salary per department.

2. `HAVING` (Filter After `GROUP BY`)

Find departments where the **total salary** exceeds `20,000`.

```
SELECT Department, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY Department
HAVING SUM(Salary) > 20000;
```

Result

Department	TotalSalary
HR	18500
IT	24500

> **Explanation:**
>
> - `HAVING` **is used to filter aggregated values (** `SUM(Salary) > 20000` **).**
> - Unlike `WHERE`, which filters individual rows, `HAVING` works on **grouped** data.

3. `ANY` (Compare Against Any Value in a Subquery)

Find employees who earn **more than at least one employee in the HR department**.

```
SELECT Name, Salary
FROM Employees
WHERE Salary > ANY (SELECT Salary FROM Employees WHERE
Department = 'HR');
```

Result

Name	Salary
Bob	7000
Charlie	6000
David	8000
Eve	7500
Frank	9000
Hank	7200
Grace	6500

> **Explanation:**
>
> - The subquery `(SELECT Salary FROM Employees WHERE Department = 'HR')` returns `{5000, 5500, 7000}`.
> - `ANY` means **at least one comparison must be true**, so it finds employees earning more than **any** salary in HR (i.e., greater than `5000`, `5500`, or `7000`).

4. ALL (Compare Against All Values in a Subquery)

Find employees who earn **more than every employee in the Finance department**.

```
SELECT Name, Salary
FROM Employees
WHERE Salary > ALL (SELECT Salary FROM Employees WHERE
Department = 'Finance');
```

Result

Name	Salary
Bob	7000
Charlie	6000
David	8000
Eve	7500
Frank	9000

> **Explanation:**
>
> - The subquery `(SELECT Salary FROM Employees WHERE Department = 'Finance')` returns `{6500, 7200}`.
> - `ALL` means the condition must be true for every row.
> - Employees must have a salary greater than 7200.

5. `EXISTS` (Check If Any Matching Row Exists)

Find departments that **have at least one employee**.

```
SELECT DISTINCT Department
FROM Employees e
WHERE EXISTS (SELECT 1 FROM Employees WHERE e.Department =
Employees.Department);
```

Result

Department
HR
IT
Finance

> **Explanation:**
>
> - `EXISTS` checks if the subquery returns any row.
> - Since every department has employees, all departments are listed.

6. Combining `GROUP BY`, `HAVING`, and `EXISTS`

Find departments where the **average salary is above `6000`** and **there are employees in that department**.

```
SELECT Department, AVG(Salary) AS AvgSalary
FROM Employees
WHERE EXISTS (SELECT 1 FROM Employees e WHERE e.Department
= Employees.Department)
GROUP BY Department
HAVING AVG(Salary) > 6000;
```

Department	AvgSalary
HR	6166.67
IT	8166.67
Finance	6850.00

> **Explanation:**
>
> - `EXISTS` **ensures the department has employees.**
> - `GROUP BY` **groups employees by department.**
> - `HAVING` **filters out departments where** `AVG(Salary) ≤ 6000`**.**

Conclusion

Clause	Purpose
`GROUP BY`	Groups rows and applies aggregate functions (`SUM()` , `AVG()` , etc.).
`HAVING`	Filters grouped results based on aggregate conditions.
`ANY`	Checks if **any value** in a subquery meets a condition.
`ALL`	Checks if **all values** in a subquery meet a condition.
`EXISTS`	Returns `TRUE` if the subquery returns any rows.

Correlated and Nested Queries

Query Type	When to Use	Example Use Case
Nested Queries (Subqueries)	When you need a query **inside another query** that runs independently	Finding employees who earn more than the average salary
Correlated Subqueries	When the inner query **depends on the outer query** and runs for each row	Finding employees who earn the highest salary in their department

1. Nested Queries (Subqueries)

Example 1: Find Employees Who Earn More Than the Average Salary

```
SELECT Name, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

Result

Name	Salary
Bob	7000
David	8000
Eve	7500
Frank	9000
Hank	7200

How It Works:

- The **subquery** `(SELECT AVG(Salary) FROM Employees)` **calculates the average salary.**
- The **main query** then selects employees whose salary is **greater than this value.**

2. Correlated Subqueries

Example 2: Find Employees Who Earn the Highest Salary in Their Department

```
SELECT Name, Salary, DepartmentID
FROM Employees e1
WHERE Salary = (SELECT MAX(Salary) FROM Employees e2 WHERE
e1.DepartmentID = e2.DepartmentID);
```

Result

Name	Salary	DepartmentID
Bob	7000	1
Frank	9000	2
Hank	7200	3

> **How It Works:**
>
> - The **inner query** `(SELECT MAX(Salary) FROM Employees e2 WHERE e1.DepartmentID = e2.DepartmentID)`
> - Runs for **each row** in the outer query.
> - Finds the **maximum salary** in the same department as the current row.
> - The **outer query** then selects only those employees **whose salary matches the max salary**.

SELF JOIN

Find Pairs of Employees in the Same Department

```sql
SELECT e1.Name AS Employee1, e2.Name AS Employee2,
e1.DepartmentID
FROM Employees e1
JOIN Employees e2 ON e1.DepartmentID = e2.DepartmentID AND
e1.EmployeeID <> e2.EmployeeID;
```

Result

Employee1	Employee2	DepartmentID
Alice	Bob	1
Alice	Charlie	1
Bob	Alice	1
Bob	Charlie	1

> **How It Works:**
>
> - Joins `Employees` **to itself** to find pairs in the same department.

PARTITION BY

The `PARTITION BY` clause is used in **window functions** to divide the result set into partitions and perform calculations within each partition **without collapsing rows**. Use it when you need **aggregations over subsets of data** while retaining row-level detail.

Common Use Cases

1. **Ranking rows within groups** (e.g., finding the top N sales per region)
2. **Running totals or cumulative sums** (e.g., calculating a running balance)
3. **Comparing current and previous rows** (e.g., difference from the last transaction)
4. **Aggregations while keeping details** (e.g., average salary per department)

Key Difference: `PARTITION BY` vs `GROUP BY`

Feature	`PARTITION BY`	`GROUP BY`
Keeps individual rows?	✅ Yes	❌ No (Collapses rows)
Used with	Window Functions	Aggregate Functions
Example Functions	`RANK()`, `SUM() OVER`, `LAG()`	`SUM()`, `COUNT()`, `AVG()`
Example Use	Running totals, rankings, previous row values	Total sales per region

Examples

EmployeeID	Name	Department	Salary
2	Bob	HR	7000
3	Charlie	HR	6000
1	Alice	HR	5000
6	Frank	IT	9000
4	David	IT	8000
5	Eve	IT	7500
8	Hank	Finance	7200
7	Grace	Finance	6500

Example 1: Ranking Employees by Salary Within Department

```
SELECT EmployeeID, Name, Department, Salary,
RANK() OVER (PARTITION BY Department ORDER BY Salary DESC)
AS Rank
FROM Employees;
```

Result

EmployeeID	Name	Department	Salary	Rank
2	Bob	HR	7000	1
3	Charlie	HR	6000	2
1	Alice	HR	5000	3
6	Frank	IT	9000	1
4	David	IT	8000	2
5	Eve	IT	7500	3
8	Hank	Finance	7200	1
7	Grace	Finance	6500	2

Example 2: Running Total of Salary in Each Department

```
SELECT EmployeeID, Name, Department, Salary,
SUM(Salary) OVER (PARTITION BY Department ORDER BY Salary)
AS RunningTotal
FROM Employees;
```

Result

EmployeeID	Name	Department	Salary	RunningTotal
1	Alice	HR	5000	5000
3	Charlie	HR	6000	11000
2	Bob	HR	7000	18000
5	Eve	IT	7500	7500
4	David	IT	8000	15500
6	Frank	IT	9000	24500
7	Grace	Finance	6500	6500
8	Hank	Finance	7200	13700

Example 3: Previous Employee Salary in Each Department

```
SELECT EmployeeID, Name, Department, Salary,
LAG(Salary) OVER (PARTITION BY Department ORDER BY Salary)
AS PreviousSalary
FROM Employees;
```

Result

EmployeeID	Name	Department	Salary	PreviousSalary
1	Alice	HR	5000	NULL
3	Charlie	HR	6000	5000
2	Bob	HR	7000	6000
5	Eve	IT	7500	NULL
4	David	IT	8000	7500
6	Frank	IT	9000	8000
7	Grace	Finance	6500	NULL
8	Hank	Finance	7200	6500

Example 4: Average Salary per Department

```
SELECT EmployeeID, Name, Department, Salary,
AVG(Salary) OVER (PARTITION BY Department) AS AvgSalary
FROM Employees;
```

Result

EmployeeID	Name	Department	Salary	AvgSalary
1	Alice	HR	5000	6000
2	Bob	HR	7000	6000
3	Charlie	HR	6000	6000
4	David	IT	8000	8166.67
5	Eve	IT	7500	8166.67
6	Frank	IT	9000	8166.67
7	Grace	Finance	6500	6850
8	Hank	Finance	7200	6850

Summary

- `RANK()` assigns **ranking within partitions** (departments).
- `SUM() OVER` calculates **cumulative totals** within partitions.
- `LAG()` retrieves **previous row's value** within partitions.
- `AVG() OVER` computes **aggregates while keeping row details**.

RANK(), DENSE_RANK(), and ROW_NUMBER()

These three **window functions** are used for **assigning unique row numbers** based on a given ordering within partitions (groups of data).

Function	When to Use
`RANK()`	When you need rankings **with gaps** if there are ties.
`DENSE_RANK()`	When you need rankings **without gaps** if there are ties.
`ROW_NUMBER()`	When you need **a unique row number for each row**, even if values are the same.

1. `RANK()` Example (Gaps in Ranking)

```
SELECT EmployeeID, Name, Department, Salary,
RANK() OVER (PARTITION BY Department ORDER BY Salary DESC)
AS Rank
FROM Employees;
```

Result (`RANK()` has gaps)

EmployeeID	Name	Department	Salary	Rank
2	Bob	HR	7000	1
3	Charlie	HR	6000	2
9	Ivy	HR	6000	2
1	Alice	HR	5000	4
6	Frank	IT	9000	1
4	David	IT	8000	2
5	Eve	IT	7500	3
8	Hank	Finance	7200	1
7	Grace	Finance	6500	2

Note: Since **Charlie and Ivy have the same salary, they share rank 2, and the next rank skips to 4**.

2. `DENSE_RANK()` Example (No Gaps in Ranking)

```
SELECT EmployeeID, Name, Department, Salary,
DENSE_RANK() OVER (PARTITION BY Department ORDER BY Salary
DESC) AS DenseRank
FROM Employees;
```

Result (`DENSE_RANK()` removes gaps)

EmployeeID	Name	Department	Salary	DenseRank
2	Bob	HR	7000	1
3	Charlie	HR	6000	2
9	Ivy	HR	6000	2
1	Alice	HR	5000	3
6	Frank	IT	9000	1
4	David	IT	8000	2
5	Eve	IT	7500	3
8	Hank	Finance	7200	1
7	Grace	Finance	6500	2

> **Difference:** `DENSE_RANK()` ensures there are **no gaps**—after `2`, the next rank is `3`, not `4`.

3. `ROW_NUMBER()` Example (Unique Row Number)

```
SELECT EmployeeID, Name, Department, Salary,
ROW_NUMBER() OVER (PARTITION BY Department ORDER BY Salary
DESC) AS RowNum
FROM Employees;
```

Result (`ROW_NUMBER()` assigns unique numbers)

EmployeeID	Name	Department	Salary	RowNum
2	Bob	HR	7000	1
3	Charlie	HR	6000	2
9	Ivy	HR	6000	3
1	Alice	HR	5000	4
6	Frank	IT	9000	1
4	David	IT	8000	2
5	Eve	IT	7500	3
8	Hank	Finance	7200	1
7	Grace	Finance	6500	2

Comparison Table

Function	Handles Ties?	Gaps in Ranking?	Unique Row for Each Record?
`RANK()`	Yes	**Yes**	No
`DENSE_RANK()`	Yes	**No** (Ranks stay sequential)	No
`ROW_NUMBER()`	No	No (Strictly sequential)	**Yes**

When to Use Which?

Use Case	Best Function
Need rankings **with gaps**	`RANK()`
Need rankings **without gaps**	`DENSE_RANK()`
Need **strict numbering with no ties**	`ROW_NUMBER()`
Need to **pick the top N per group**	`ROW_NUMBER()`
Need **consistent ranking even with ties**	`DENSE_RANK()`

Example: Get Top 2 Highest Salaries per Department

```
SELECT EmployeeID, Name, Department, Salary, Rank
FROM (
    SELECT EmployeeID, Name, Department, Salary,
    RANK() OVER
    (PARTITION BY Department ORDER BY Salary DESC) AS Rank
FROM Employees)
RankedEmployees WHERE Rank <= 2;
```

Result

EmployeeID	Name	Department	Salary	Rank
2	Bob	HR	7000	1
3	Charlie	HR	6000	2
9	Ivy	HR	6000	2
6	Frank	IT	9000	1
4	David	IT	8000	2
8	Hank	Finance	7200	1
7	Grace	Finance	6500	2

LEAD() Function in SQL - Example

The `LEAD()` function is used to access data from **the next row** within the same result set **without using a self-join**. It is useful for **comparing values between rows**.

Example Use Cases

- Finding the **next employee's salary** in the same department.
- Calculating **difference between current and next month's sales**.
- Checking **the next event in a schedule**.

1. Basic `LEAD()` Example

```
SELECT EmployeeID, Name, Department, Salary,
LEAD(Salary) OVER (PARTITION BY Department ORDER BY Salary
DESC) AS NextSalary
FROM Employees;
```

Result

EmployeeID	Name	Department	Salary	NextSalary
2	Bob	HR	7000	6000
3	Charlie	HR	6000	5000
1	Alice	HR	5000	NULL
6	Frank	IT	9000	8000
4	David	IT	8000	7500
5	Eve	IT	7500	NULL
8	Hank	Finance	7200	6500
7	Grace	Finance	6500	NULL

Explanation:

- `LEAD(Salary)` returns the next salary in the same department based on `ORDER BY Salary DESC`.
- For the last employee in each department, `LEAD()` returns `NULL` (no next row available).

2. Using `LEAD()` to Find Salary Difference

```
SELECT EmployeeID, Name, Department, Salary,
LEAD(Salary) OVER (PARTITION BY Department ORDER BY Salary
DESC) AS NextSalary,
       Salary - LEAD(Salary) OVER (PARTITION BY Department
ORDER BY Salary DESC) AS SalaryDifference
FROM Employees;
```

Result

EmployeeID	Name	Department	Salary	NextSalary	SalaryDifference
2	Bob	HR	7000	6000	1000
3	Charlie	HR	6000	5000	1000
1	Alice	HR	5000	NULL	NULL
6	Frank	IT	9000	8000	1000
4	David	IT	8000	7500	500
5	Eve	IT	7500	NULL	NULL
8	Hank	Finance	7200	6500	700
7	Grace	Finance	6500	NULL	NULL

Explanation:

- The column `SalaryDifference` calculates the **difference between current and next salary**.
- If there is no next salary (`NULL`), then `SalaryDifference` is also `NULL`.

3. Using `LEAD()` with a Custom Offset

You can specify how many rows ahead you want to look at.

```
SELECT EmployeeID, Name, Department, Salary,
LEAD(Salary, 2) OVER (PARTITION BY Department ORDER BY
Salary DESC) AS SalaryAfterTwo
FROM Employees;
```

Result

EmployeeID	Name	Department	Salary	SalaryAfterTwo
2	Bob	HR	7000	5000
3	Charlie	HR	6000	NULL
1	Alice	HR	5000	NULL
6	Frank	IT	9000	7500
4	David	IT	8000	NULL
5	Eve	IT	7500	NULL
8	Hank	Finance	7200	NULL
7	Grace	Finance	6500	NULL

> **Explanation:**
>
> - `LEAD(Salary, 2)` looks two rows ahead instead of just one.
> - If there aren't two rows ahead, it returns `NULL`.

4. Handling NULL Values with `LEAD()` (Default Value)

By default, `LEAD()` returns `NULL` if there's no next row. You can provide a **default value** instead.

```
SELECT EmployeeID, Name, Department, Salary,
LEAD(Salary, 1, 0) OVER (PARTITION BY Department ORDER BY
Salary DESC) AS NextSalary FROM Employees;
```

Result

EmployeeID	Name	Department	Salary	NextSalary
2	Bob	HR	7000	6000
3	Charlie	HR	6000	5000
1	Alice	HR	5000	**0**
6	Frank	IT	9000	8000
4	David	IT	8000	7500
5	Eve	IT	7500	**0**
8	Hank	Finance	7200	6500
7	Grace	Finance	6500	**0**

> **Explanation:**
>
> - `LEAD(Salary, 1, 0)` means if there's no next salary, it returns 0 instead of `NULL`.

When to Use `LEAD()`?

Use Case	Why Use `LEAD()`?
Comparing **current vs next row values**	E.g., Checking the salary difference between employees.
Finding **next events, orders, or transactions**	E.g., Checking what the next scheduled appointment is.
Performing **trend analysis**	E.g., Comparing sales of the current month vs the next month.
Avoiding **self-joins for next row comparisons**	E.g., Instead of using `JOIN` on the same table, `LEAD()` simplifies the query.

Summary

- `LEAD()` **looks ahead** to the next row.
- It helps in **comparing values** between rows without using a self-join.
- You can **customize the offset** (`LEAD(value, offset)`).
- You can **set a default value** to replace `NULL` (`LEAD(value, offset, default_value)`).

Exercise

Here's a **comprehensive list of 50 SQL exercises** from platforms like **LeetCode, DataLemur, and SQLZoo**, covering **basic to advanced** SQL concepts. Each exercise includes the **problem statement, solution, and explanation**.

📌 List of Topics Covered

✔ **Basic Queries** (`SELECT`, `WHERE`, `ORDER BY`)
✔ **Aggregations** (`GROUP BY`, `HAVING`)
✔ **Joins** (`INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, `FULL JOIN`, `SELF JOIN`, `CROSS JOIN`)
✔ **Subqueries** (`Nested`, `Correlated`)
✔ **Window Functions** (`RANK()`, `DENSE_RANK()`, `ROW_NUMBER()`, `LEAD()`, `LAG()`)
✔ **Set Operations** (`UNION`, `INTERSECT`, `EXCEPT`)
✔ **Conditional Statements** (`CASE WHEN`)
✔ **Existence Checks** (`EXISTS`, `NOT EXISTS`)
✔ **Filtering** (`ANY`, `ALL`, `LIKE`, `IN`)

💡 Basic SQL Queries

1️⃣ Retrieve All Employees' Names and Salaries

Problem: Select all employees' names and salaries from the `Employees` table.

```sql
SELECT Name, Salary FROM Employees;
```

- **Explanation:** Basic `SELECT` statement to retrieve data.

2️⃣ Find Employees Earning More Than $50,000

```sql
SELECT Name, Salary
FROM Employees
WHERE Salary > 50000;
```

- **Explanation:** Uses `WHERE` to filter salaries greater than `50000`.

3️⃣ Order Employees by Salary Descending

```sql
SELECT Name, Salary
FROM Employees
ORDER BY Salary DESC;
```

- **Explanation:** `ORDER BY DESC` sorts from highest to lowest salary.

📊 Aggregation & Grouping

4️⃣ Count Employees in Each Department

```sql
SELECT Department, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY Department;
```

- **Explanation:** Uses `COUNT(*)` with `GROUP BY` to count employees per department.

5️⃣ Find Departments with More Than 10 Employees

```sql
SELECT Department, COUNT(*) AS EmployeeCount
FROM Employees
GROUP BY Department
HAVING COUNT(*) > 10;
```

- **Explanation:**

✔ `HAVING` filters groups after aggregation (unlike `WHERE`, which filters before grouping).

6️⃣ Find the Maximum Salary in Each Department

```sql
SELECT Department, MAX(Salary) AS MaxSalary
FROM Employees
GROUP BY Department;
```

- **Explanation:** Uses `MAX()` to find the highest salary per department.

🔗 SQL Joins

7️⃣ Retrieve Employees with Their Department Names

```sql
SELECT e.Name, e.Salary, d.DepartmentName
FROM Employees e
INNER JOIN Departments d
ON e.DepartmentID = d.DepartmentID;
```

- **Explanation:** `INNER JOIN` matches employees with departments.

8️⃣ List All Employees, Even Those Without a Department (`LEFT JOIN`)

```sql
SELECT e.Name, e.Salary, d.DepartmentName
FROM Employees e
LEFT JOIN Departments d
ON e.DepartmentID = d.DepartmentID;
```

- **Explanation:**

✔ `LEFT JOIN` ensures **all employees** appear, even if they have no department.

9️⃣ Find Departments Without Employees (`RIGHT JOIN`)

```sql
SELECT d.DepartmentName
FROM Employees e
RIGHT JOIN Departments d
ON e.DepartmentID = d.DepartmentID
WHERE e.EmployeeID IS NULL;
```

🔹 **Explanation:**

✔ `RIGHT JOIN` ensures all departments appear.

✔ `WHERE e.EmployeeID IS NULL` filters **departments with no employees**.

📌 Subqueries

🔟 Find Employees Earning More Than the Average Salary

```
SELECT Name, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);
```

🔹 **Explanation:** Uses a **subquery** to compute average salary.

1️⃣1️⃣ Find the Department with the Highest Salary

```
SELECT DepartmentID
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees);
```

🔹 **Explanation: Nested Query** retrieves max salary and finds the department.

📌 Window Functions

1️⃣2️⃣ Rank Employees by Salary in Each Department

```
SELECT Name, Salary, DepartmentID,
       RANK() OVER (PARTITION BY DepartmentID ORDER BY Salary DESC) AS
SalaryRank
FROM Employees;
```

🔹 **Explanation:**

✔ `RANK()` assigns ranks within **each department** based on `Salary DESC`.

1️⃣3️⃣ Assign a Unique Row Number to Employees Per Department

```
SELECT Name, Salary, DepartmentID,
       ROW_NUMBER() OVER (PARTITION BY DepartmentID ORDER BY Salary DESC)
AS RowNum
FROM Employees;
```

🔹 **Explanation:**

✔ `ROW_NUMBER()` assigns **unique numbers** per department.

📌 Set Operations

1️⃣4️⃣ Find Employees Who Are Also in the Managers Table (`INTERSECT`)

```
SELECT Name FROM Employees
INTERSECT
SELECT Name FROM Managers;
```

🔹 **Explanation:**

✔ `INTERSECT` returns names common to both tables.

1️⃣5️⃣ Find Employees Who Are Not Managers (EXCEPT)

```
SELECT Name FROM Employees
EXCEPT
SELECT Name FROM Managers;
```

🔹 **Explanation:**

✔ `EXCEPT` returns employees **who are not in Managers**.

📌 EXISTS, ANY, ALL

1️⃣6️⃣ Find Employees in Departments with More Than 5 Employees

```
SELECT Name, DepartmentID
FROM Employees e
WHERE EXISTS (SELECT 1 FROM Employees e2 WHERE e.DepartmentID =
e2.DepartmentID GROUP BY e2.DepartmentID HAVING COUNT(*) > 5);
```

🔹 **Explanation:**

✔ `EXISTS` checks if any department has **more than 5 employees**.

1️⃣7️⃣ Find Employees Who Earn More Than Every Employee in HR (ALL)

```sql
SELECT Name, Salary
FROM Employees
WHERE Salary > ALL (SELECT Salary FROM Employees WHERE DepartmentID = 1);
```

🔹 **Explanation:**

✔ Compares salary with **all salaries** in HR.

📌 Advanced Queries

1️⃣8️⃣ Find the Second Highest Salary

```sql
SELECT DISTINCT Salary
FROM Employees
ORDER BY Salary DESC
LIMIT 1 OFFSET 1;
```

🔹 **Explanation:**

✔ `OFFSET 1` skips the highest salary and fetches the second highest.

1️⃣9️⃣ Find Employees Who Joined Before Their Manager

```
SELECT e.Name
FROM Employees e
JOIN Employees m ON e.ManagerID = m.EmployeeID
WHERE e.JoinDate < m.JoinDate;
```

🔹 **Explanation:**

✔ Self-join (`JOIN Employees m ON e.ManagerID = m.EmployeeID`).

2️⃣0️⃣ Retrieve Employee Salary Difference from the Previous Month (`LAG()`)

```
SELECT EmployeeID, Name, Salary,
       Salary - LAG(Salary) OVER (PARTITION BY EmployeeID ORDER BY Month)
AS SalaryChange
FROM Salaries;
```

🔹 **Explanation:**

✔ `LAG()` fetches the **previous month's salary** for comparison.
