**DAY 2 – Advanced SQL Concepts**

**SQL Key Concepts**

==============================================================

**🔹 1. WHERE Clause -**  
The WHERE clause is used to **filter rows** before any grouping or aggregation happens.

**Syntax:**

SELECT \*

FROM students

WHERE grade = 'A';

**Key Points:**

* Applies to **individual rows**.
* Can filter based on any column.
* Comes **before GROUP BY or HAVING**.

**Example:**

SELECT name, marks

FROM students

WHERE marks > 80;

==============================================================

**🔹 2. HAVING Clause -**  
HAVING is like a WHERE clause for **aggregated data**. It filters the results **after grouping**.

**Syntax:**

SELECT grade, COUNT(\*) AS num\_students

FROM students

GROUP BY grade

HAVING COUNT(\*) > 2;

**Key Points:**

* Use when filtering **aggregated results**.
* HAVING is evaluated **after GROUP BY**.
* Cannot be replaced by WHERE for aggregate filters.

**Example:**

-- Find subjects with average marks above 80

SELECT subject, AVG(marks) AS avg\_marks

FROM students

GROUP BY subject

HAVING AVG(marks) > 80;

==============================================================

**🔹 3. GROUP BY Clause -**  
GROUP BY groups rows that have the same values in specified columns into summary rows often used with aggregate functions like COUNT(), SUM(), AVG(), etc.

**Syntax:**

SELECT grade, COUNT(\*) AS num\_students

FROM students

GROUP BY grade;

**Key Points:**

* Works **after WHERE** and **before HAVING**.
* Each column in SELECT not part of an aggregate must be in the GROUP BY.

**Example:**

SELECT subject, AVG(marks) AS avg\_marks

FROM students

GROUP BY subject;

==============================================================

**🔹 4. Window Function -**  
A **window function** performs a calculation across a set of table rows that are somehow related to the current row, without collapsing the rows like GROUP BY does.

**Common window functions:**

* RANK()
* DENSE\_RANK()
* ROW\_NUMBER()
* SUM(), AVG(), COUNT() (as window functions)

**Syntax:**

SELECT

name,

marks,

RANK() OVER (ORDER BY marks DESC) AS rank

FROM

students;

**Key Elements:**

* OVER(...): Defines the "window" (set of rows to consider for the calculation).
* PARTITION BY: (Optional) Divides rows into groups.
* ORDER BY: Orders the rows in each group/window.

**Advantages:**

* Doesn’t reduce result set like GROUP BY.
* Can perform calculations **while keeping all rows**.
* Supports advanced analytics (ranking, moving averages, percentiles).

**🧾 Summary Table**

| **Concept** | **Purpose** | **Applied To** | **Comes Before/After** | **Example Use** |
| --- | --- | --- | --- | --- |
| **WHERE** | Filters raw rows | Individual rows | Before GROUP BY | WHERE marks > 70 |
| **GROUP BY** | Groups rows for aggregation | Sets of rows | After WHERE | GROUP BY grade |
| **HAVING** | Filters grouped/aggregated data | Aggregated groups | After GROUP BY | HAVING COUNT(\*) > 2 |
| **Window Funcs** | Row-wise calculations over groups | Rows (not collapsed) | Doesn't filter rows | RANK() OVER (ORDER BY marks DESC) |

**🔹 5. CTE (Common Table Expression)-**

**Make complex queries more readable to read & write**

**With student as(**

**Select \* from students where age < 20**

**)**

**SELECT name, grade from students;**

* Basic CTE
* Recursive CTE (Factorial)

| **SQL Concept** | **Definition** | **Functions Used** | **Example** |
| --- | --- | --- | --- |
|  |  |  |  |
| **Joins** | Combines rows from two or more tables based on related columns | INNER JOIN, LEFT JOIN, ON | Link orders with customers/products |
| **Aggregation** | Performs calculations across multiple rows | SUM(), COUNT(), AVG(), GROUP BY, HAVING | Total revenue per category, orders per customer |
| **String Functions** | Manipulate or analyze string/text values | UPPER(), LEN(), LIKE, CONCAT() | Search customers by name, format full name |
| **Date Functions** | Handle and compute date/time values | DATEDIFF(), FORMAT(), YEAR(), MONTH(), GETDATE() | Delivery time analysis, sign-up trends |
| **Math Functions** | Perform arithmetic operations and rounding | ROUND(), ABS(), basic math: +, -, \*, / | Round salaries, calculate differences |
| **Subqueries** | Query within a query to compute dynamic results | Scalar subquery, Correlated subquery | Compare salary to average, find max revenue |
| **Window Functions** | Compute values across rows related to the current row | ROW\_NUMBER(), RANK(), LAG(), LEAD(), PARTITION BY | Find first/last order, monthly revenue change |
| **CTEs (WITH)** | Temporary named result sets used within a single SQL statement | WITH, RECURSIVE | Cohort analysis, generate numbers 1 to 10 |
| **Case Expressions** | Return specific values based on conditions | CASE, WHEN, THEN, ELSE | Label stock as low/high, categorize salaries |
| **NULL Handling** | Manage missing or undefined values | IS NULL, IS NOT NULL, ISNULL(), COALESCE() | Identify or replace missing delivery dates or order totals |
|  |  |  |  |

Task 1 - Advanced SQL Problem Set with [Dataset](https://drive.google.com/file/d/1EtcD4npVKiSTn9C_t1hTp7rSysDy7gxJ/view?usp=drive_link)🔗

create database company;

use company;

create table departments(

department\_id INT PRIMARY KEY,

department\_name VARCHAR(50) NOT NULL

);

CREATE TABLE projects (

project\_id INT PRIMARY KEY,

project\_name VARCHAR(50) NOT NULL,

start\_date DATE,

end\_date DATE

);

CREATE TABLE employees (

emp\_id INT PRIMARY KEY,

first\_name VARCHAR(50) NOT NULL,

last\_name VARCHAR(50) NOT NULL,

department VARCHAR(50),

salary INT CHECK (salary >= 0),

join\_date DATE NOT NULL,

manager\_id INT NULL,

project\_id INT NULL,

CONSTRAINT FK\_Manager FOREIGN KEY (manager\_id) REFERENCES employees(emp\_id),

CONSTRAINT FK\_Project FOREIGN KEY (project\_id) REFERENCES projects(project\_id)

);

INSERT INTO departments (department\_id, department\_name) VALUES

(1, 'Engineering'),

(2, 'Human Resources'),

(3, 'Marketing'),

(4, 'Sales'),

(5, 'Finance');

INSERT INTO projects (project\_id, project\_name, start\_date, end\_date) VALUES

(101, 'Apollo Revamp', '2024-01-15', '2024-07-15'),

(102, 'Neon AI Suite', '2024-03-01', '2024-12-01'),

(103, 'Brand Blitz', '2024-02-10', '2024-08-10'),

(104, 'Sales Dashboard', '2024-04-01', '2024-10-01'),

(105, 'Finance Forecast', '2024-05-01', '2024-11-01');

INSERT INTO employees (emp\_id, first\_name, last\_name, department, salary, join\_date, manager\_id, project\_id) VALUES

(1, 'Alice', 'Smith', 'Engineering', 95000, '2023-01-10', NULL, 101),

(2, 'Bob', 'Johnson', 'Engineering', 85000, '2023-02-20', 1, 101),

(3, 'Carol', 'Williams', 'Human Resources', 70000, '2023-03-15', NULL, NULL),

(4, 'David', 'Brown', 'Marketing', 80000, '2022-11-01', NULL, 103),

(5, 'Eva', 'Davis', 'Sales', 75000, '2023-04-12', 4, 104),

(6, 'Frank', 'Miller', 'Finance', 88000, '2023-06-10', NULL, 105),

(7, 'Grace', 'Wilson', 'Engineering', 90000, '2023-07-05', 1, 102),

(8, 'Henry', 'Moore', 'Marketing', 77000, '2023-05-01', 4, 103),

(9, 'Ivy', 'Taylor', 'Sales', 73000, '2023-08-01', 5, 104),

(10, 'Jake', 'Anderson', 'Finance', 89000, '2023-09-15', 6, 105);

select \* from employees

select \* from departments

select \* from projects

--====String Functions===----

--1. Find employees whose last name starts with 'S'.

SELECT \*

FROM employees

WHERE last\_name LIKE 'S%';

--2. Display first\_name and last\_name concatenated as full\_name in uppercase.

SELECT UPPER(first\_name + ' ' + last\_name) AS full\_name

FROM employees;

--3. Show employees with a 5-character first name.

SELECT \*

FROM employees

WHERE LEN(first\_name) = 5;

---=== Date Functions ===---

--4. List employees who joined in the last 2 years.

SELECT \*

FROM employees

WHERE join\_date >= DATEADD(YEAR, -2, GETDATE());

--5. Show number of days since each employee joined.

SELECT emp\_id, first\_name, last\_name, DATEDIFF(DAY, join\_date, GETDATE()) AS days\_since\_joined

FROM employees;

--6. Find the month name and year from each employee's join\_date.

SELECT emp\_id, first\_name, last\_name,

DATENAME(MONTH, join\_date) AS join\_month,

YEAR(join\_date) AS join\_year

FROM employees;

---=== Maths Functions ===---

INSERT INTO employees (emp\_id, first\_name, last\_name, department, salary, join\_date, manager\_id, project\_id) VALUES

(11, 'Mark', 'Twain', 'Engineering', 95459, '2025-01-10', NULL, 104);

--7. Round off each employee's salary to the nearest thousand.

SELECT emp\_id, first\_name, last\_name,

salary, ROUND(salary, -3) AS rounded\_salary

FROM employees;

--8. Find employees whose salary is above the average salary.

SELECT \*

FROM employees

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

--9. Show absolute difference from company average salary.

SELECT AVG(salary) as average\_sal FROM employees;

SELECT emp\_id, first\_name, last\_name, salary,

ABS(salary - (SELECT AVG(salary) FROM employees)) AS salary\_diff\_from\_avg

FROM employees;

---==== Aggregate Functions ===----

--10. Find departments with more than 3 employees.

SELECT department, COUNT(\*) AS employee\_count

FROM employees

GROUP BY department

HAVING COUNT(\*) > 3;

--11. Show total and average salary per department with avg salary > 60000.

select department, sum(salary) as total\_sal, avg(salary) as avg\_sal

from employees

group by department

having avg(salary) > 60000;

---=== Subqueries ===---

--12. Find the employee(s) with the maximum salary.

select

max(salary) as max\_salary

from employees;

/\* SELECT \*

FROM employees

WHERE salary = (SELECT MAX(salary) FROM employees); \*/

--13. List employees earning more than avg salary in their department

SELECT \*

FROM employees e

WHERE salary > (

SELECT AVG(salary)

FROM employees

WHERE department = e.department

);

--14. Show employees who joined before the earliest join date in IT.

SELECT \*

FROM employees

WHERE join\_date < (

SELECT MIN(join\_date)

FROM employees

WHERE department = 'Engineering'

);

---=== Joins ===---

--15. Show each employee's name and manager's name.

SELECT e.first\_name + ' ' + e.last\_name AS employee\_name,

m.first\_name + ' ' + m.last\_name AS manager\_name

FROM employees e

LEFT JOIN employees m ON e.manager\_id = m.emp\_id;

--16. List employees with their department name.

SELECT e.first\_name, e.last\_name, d.department\_name

FROM employees e

JOIN departments d ON e.department = d.department\_name;

--17. List employees not assigned any project.

SELECT \*

FROM employees

WHERE project\_id IS NULL;

---=== Window Functions ===---

--18. Assign a row number to employees in each department based on salary.

SELECT salary, emp\_id, first\_name, last\_name, department,

ROW\_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) AS row\_num

FROM employees;

--19. Show running total salary within each department.

SELECT emp\_id, first\_name, department, salary,

SUM(salary) OVER (PARTITION BY department ORDER BY emp\_id) AS running\_total

FROM employees;

--20. Show difference in salary between employee and previous by join date.

SELECT emp\_id, first\_name, join\_date, salary,

salary - LAG(salary) OVER (ORDER BY join\_date) AS salary\_diff

FROM employees;

---=== Common Table Expressions (CTEs) ===---

--21. Use CTE to calculate total salary per department, filter total > 200000.

WITH DeptSalary AS (

SELECT department, SUM(salary) AS total\_salary

FROM employees

GROUP BY department

)

SELECT \*

FROM DeptSalary

WHERE total\_salary > 200000;

--22. Create a recursive CTE to generate numbers 1 to 10.

WITH Numbers AS (

SELECT 1 AS num

UNION ALL

SELECT num + 1 FROM Numbers WHERE num < 10

)

SELECT \* FROM Numbers;

---=== CASE Statements ===---

--24. Label employees as 'Junior', 'Mid', or 'Senior' based on salary.

SELECT first\_name, last\_name, salary,

CASE

WHEN salary < 10000 THEN 'Junior'

WHEN salary BETWEEN 20000 AND 90000 THEN 'Mid'

ELSE 'Senior'

END AS level

FROM employees;

--25. Count employees in salary categories using CASE.

SELECT

COUNT(CASE WHEN salary < 60000 THEN 1 END) AS junior\_count,

COUNT(CASE WHEN salary BETWEEN 60000 AND 90000 THEN 1 END) AS mid\_count,

COUNT(CASE WHEN salary > 90000 THEN 1 END) AS senior\_count

FROM employees;

---=== Null Functions ===---

--26. Replace NULL department values with 'Unknown'.

SELECT emp\_id, first\_name, last\_name,

ISNULL(department, 'Unknown') AS department

FROM employees;

--27. Show employees with no department.

SELECT \*

FROM employees

WHERE department IS NULL;

--28. Use COALESCE to provide default for missing projects.

SELECT emp\_id, first\_name, last\_name,

COALESCE(project\_id, 0) AS project\_id

FROM employees;