# DAY 1

1. **Explain all the algorithm basics in brief and compare**

Algorithm Basics (Brief + Comparison)

An algorithm is a clear set of steps to solve a problem.

Good algorithms are: definite, finite, correct, effective, and take input/output.

Types of algorithms:

* Searching: Find an item (e.g., Linear, Binary Search).
* Sorting: Arrange items (e.g., Bubble, Quick Sort).
* Greedy: Take best option at each step (e.g., Kruskal’s).
* Divide & Conquer: Break into sub-problems (e.g., Merge Sort).
* Dynamic Programming: Reuse solved sub-problems (e.g., Fibonacci).
* Backtracking: Try all options, backtrack if needed (e.g., Sudoku Solver).

Algorithm Approach Comparison Table

|  |  |  |  |
| --- | --- | --- | --- |
| Approach | Speed | Use Case Example | Best When |
| Brute Force | Slow | Password cracking | Input size is small |
| Heuristic | Moderate | Guessing book section | Exact answer not needed |
| Greedy | Fast | Coin change, scheduling | Greedy logic fits the problem |
| Divide and Conquer | Faster | Sorting, searching | Large datasets |
| Dynamic Programming | Fastest | Fibonacci, Knapsack | Overlapping subproblems |

1. **Compare all sorting algorithm and choose any two best according to you and why**

Sorting Algorithms Compared + Best Two

* Bubble Sort: Simple, slow; compares and swaps; O(n²).
* Insertion Sort: Good for nearly-sorted data; O(n²).
* Selection Sort: Picks min repeatedly; O(n²); not stable.
* Merge Sort: Always O(n log n), stable, uses extra space.
* Quick Sort: Very fast in practice, O(n log n) average, not stable.
* Heap Sort: O(n log n), no extra space, not stable.

Top 2 choices:

* Merge Sort: Stable, reliable performance.
* Quick Sort: Fastest on average, space-efficient.

Sorting Algorithm Comparison Table

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Algorithm | Time Complexity | Space | Stable | Best Use Case |
| Bubble Sort | O(n²) | O(1) | Yes | Very small data, learning purposes |
| Insertion Sort | O(n²) | O(1) | Yes | Small or nearly sorted lists |
| Selection Sort | O(n²) | O(1) | No | Memory-limited cases |
| Merge Sort | O(n log n) | O(n) | Yes | Large data where stability is required |
| Quick Sort | O(n log n)\* | O(log n) | No | Best average performance |

1. **compare searching algorithm**

Searching Algorithms Compared

* Linear Search: Check one by one, works on any list, slow (O(n)).
* Binary Search: Fast (O(log n)), needs sorted data.
* Jump Search: Jumps ahead by steps, better than linear, sorted data needed.
* Interpolation Search: Works well on uniformly distributed sorted data, best O(log log n).

Summary:

* Use Linear for small/unsorted data.
* Use Binary for fast search on sorted data.

Searching Algorithm Comparison Table

|  |  |  |  |
| --- | --- | --- | --- |
| Algorithm | Time Complexity | Requires Sorted Data | Best For |
| Linear Search | O(n) | No | Small or unsorted data |
| Binary Search | O(log n) | Yes | Fast lookup in sorted datasets |

1. **why we use BST and what is the need of AVL and difference between BST and AVL tree**

Why BST? Why AVL? Difference

BST (Binary Search Tree):

* Left < Root < Right.
* Fast operations if balanced.
* Can degrade to linked list if unbalanced → O(n).
* AVL Tree:
* Self-balancing BST.
* Maintains height ≈ log(n) → faster operations.
* Performs rotations to stay balanced.

Difference:

* BST: Easy to implement, can be slow if unbalanced.
* AVL: Always fast, needs more maintenance (rotations).

Comparison Table: BST vs AVL Tree

|  |  |  |
| --- | --- | --- |
| Feature | BST | AVL Tree |
| Balancing | Not guaranteed | Always balanced |
| Performance | Can degrade to O(n) | Maintains O(log n) consistently |
| Rotations | Not required | Required to maintain balance |
| Complexity | Simpler to implement | Slightly more complex due to balancing |

# DAY-3

**SQL Server Practical Assignment (30 Minutes)**

**Section A: Managing Databases (10 mins)**

**1. List all system databases in SQL Server.**

SELECT name FROM sys.databases WHERE database\_id < 5;

**2. List physical file paths for all databases.**

SELECT name, physical\_name FROM sys.master\_files;

**3. Create a new user-defined database named TeamDB.**

CREATE DATABASE TeamDB;

**4. Rename the database TeamDB to ProjectDB**.

ALTER DATABASE TeamDB MODIFY NAME = ProjectDB;

**5. Drop the ProjectDB database.**

DROP DATABASE ProjectDB;

**Section B: Managing Tables (10 mins)**

**1. Create a table Employees with the following columns:**

EmpID INT (Primary Key)

Name VARCHAR(50)

Department VARCHAR(30)

JoiningDate DATE

IsActive BIT

Salary DECIMAL(10,2)

CREATE TABLE Employees (

EmpID INT PRIMARY KEY,

Name VARCHAR(50),

Department VARCHAR(30),

JoiningDate DATE,

IsActive BIT,

Salary DECIMAL(10,2)

);

**2. Add a column Salary (DECIMAL) to the table.**

ALTER TABLE Employees ADD Salary DECIMAL(10,2);

**3. Rename table Employees to TeamMembers.**

EXEC sp\_rename 'Employees', 'TeamMembers';

**4. Drop the table TeamMembers.**

DROP TABLE TeamMembers;

**Section C: DML Operations (10 mins)**

**1. Insert three rows into Employees.**

INSERT INTO Employees VALUES

(1, 'Amit', 'HR', '2022-01-01', 1, 50000),

(2, 'Sneha', 'IT', '2021-06-15', 1, 75000),

(3, 'John', 'Finance', '2020-10-10', 0, 65000);

**2. Update salary of 'Sneha' to 80000.**

UPDATE Employees SET Salary = 80000 WHERE Name = 'Sneha';

**3. Delete employee with IsActive = 0.**

DELETE FROM Employees WHERE IsActive = 0;

**4. Retrieve names and departments of all employees.**

SELECT Name, Department FROM Employees;

**5. Fetch employees from 'IT' department with salary above 70000.**

SELECT \* FROM Employees WHERE Department = 'IT' AND Salary > 70000;

**6. Apply filtering using LIKE, BETWEEN, and IN.**

SELECT \* FROM Employees WHERE Name LIKE 'S%';

SELECT \* FROM Employees WHERE Salary BETWEEN 60000 AND 80000;

SELECT \* FROM Employees WHERE Department IN ('IT', 'Finance');

# DAY 4

1. **Insert and Update with Integrity: Create a 'students' table with constraints (NOT NULL, UNIQUE). Insert 5 records. Then, update a student's marks ensuring data integrity is maintained.**

create database day4;

use day4;

create table students(

id int primary key not null,

name varchar(100) not null,

email varchar(100) unique not null,

marks int);

insert into students values

(1, 'Aarthi', 'aarthi@example.com', 88),

(2, 'John', 'john.doe@example.com', 76),

(3, 'Sara', 'sara@example.com', 91),

(4, 'David', 'david123@example.com', 67),

(5, 'Priya', 'priya.k@example.com', 83);

update students set name='Aarthi Sharma' where id=1;

update students set name='John Doe' where id=2;

update students set name='Sara Khan' where id=3;

update students set name='David Lee' where id=4;

update students set name='Priya Kapoor' where id=5;

select \* from students;

**2. String Function Challenge: Given a 'customers' table with a 'full\_name' column, write a query to display:**

**- First name**

**- Last name**

**- Length of each name**

select

name,

substring\_index(name, ' ',1) as first\_name,

substring\_index(name, ' ', -1) as last\_name,

LENGTH(substring\_index(name, ' ', 1)) as first\_name\_length,

LENGTH(substring\_index(name, ' ', -1)) as last\_name\_length,

length(name) as name\_length

from students;

alter table students add column dob date;

UPDATE students SET dob = '2004-03-03' WHERE id = 1;

UPDATE students SET dob = '2004-04-29' WHERE id = 2;

UPDATE students SET dob = '2004-12-13' WHERE id = 3;

UPDATE students SET dob = '2004-05-08' WHERE id = 4;

UPDATE students SET dob = '2004-07-09' WHERE id = 5;

**3. Date Function Usage:**

**From a 'sales' table with a 'sale\_date' column, write a query to:**

**- Extract the month name and year**

**- Display how many days ago the sale happened**

create table sales(

sale\_id int primary key auto\_increment,

category varchar(50),

amount decimal(10, 2),

sale\_date date

);

insert into sales (category, amount, sale\_date) values

('Electronics', 10000.00, '2025-06-01'),

('Clothing', 2500.00, '2025-06-02'),

('Groceries', 1200.00, '2025-06-03'),

('Furniture', 18000.00, '2025-06-04');

select sale\_date,

monthname(sale\_date) as month,

year(sale\_date) as year,

datediff(curdate(), sale\_date) as days\_ago

from sales;

**4. Mathematical Functions on Salary:**

**In an 'employees' table, calculate:**

**- Salary after a 10% hike**

**- Round the salary to the nearest hundred**

select

employee\_id,

name,

salary as original\_salary,

round(salary \* 1.10, -2) as salary\_after\_hike

from employees;

**5. System Function Check:**

**Retrieve:**

**- Current date and time**

**- Database name and logged-in user**

select

now() AS current\_datetime,

database() AS current\_database,

current\_user() AS logged\_in\_user;

**6. Demo: Custom Result Set:**

**From the 'products' table, write a query that:**

**- Returns product name in uppercase**

**- Replaces any NULL prices with 'Not Available'**

select

upper(product\_name) AS product\_name\_uppercase,

ifnull(price, 'Not Available') AS price\_status

from products;

**7. Aggregate Functions Practice:**

**From a 'transactions' table, get:**

**- Total sales**

**- Average sale value**

**- Maximum and minimum sale on a single transaction**

create table transactions (

transaction\_id int primary key auto\_increment,

customer\_name varchar(100),

amount decimal(10,2) not null,

transaction\_date date

);

insert into transactions (customer\_name, amount, transaction\_date) values

('Aarthi', 1500.00, '2025-06-10'),

('John', 3200.50, '2025-06-11'),

('Sara', 850.75, '2025-06-12');

select

sum(amount) as total\_sales,

avg(amount) as average\_sale,

max(amount) as max\_sale,

min(amount) as min\_sale

from transactions;

**8. Grouping with Aggregation:**

**From a 'sales' table:**

**- Group by product category**

**- Show total sales and number of transactions in each category**

select

category,

sum(amount) AS total\_sales,

count(\*) AS number\_of\_transactions

from sales

group by category;

**9. Inner Join for Orders and Customers:**

**Join 'orders' and 'customers' to show:**

**- Customer name**

**- Order amount**

**- Only for customers who made orders**

CREATE TABLE customers (

customer\_id INT PRIMARY KEY,

customer\_name VARCHAR(100)

);

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

customer\_id INT,

product\_id INT,

amount DECIMAL(10, 2),

order\_date DATE,

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id),

FOREIGN KEY (product\_id) REFERENCES products(product\_id)

);

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

price DECIMAL(10, 2)

);

INSERT INTO customers (customer\_id, customer\_name) VALUES

(1, 'Aarthi'),

(2, 'Rahul'),

(3, 'Meena'),

(4, 'John'),

(5, 'Sara');

INSERT INTO products (product\_id, product\_name, price) VALUES

(1, 'Laptop', 60000.00),

(2, 'Mobile', 25000.00),

(3, 'Tablet', 15000.00),

(4, 'Headphones', 3000.00),

(5, 'Smartwatch', 8000.00);

INSERT INTO orders (order\_id,product\_id, customer\_id, amount, order\_date) VALUES

(101,2, 1, 1500.00, '2025-06-01'),

(102,1, 2, 2300.00, '2025-06-02'),

(103,1, 1, 1200.00, '2025-06-03'),

(104,5, 4, 3400.00, '2025-06-04'),

(105,4, 2, 500.00, '2025-06-05');

SELECT c.customer\_name,o.amount FROM customers c

INNER JOIN orders o ON c.customer\_id = o.customer\_id;

**10. Left Join for Products with or without Orders:**

**Show all products with:**

**- Their order details (if available)**

**- Use LEFT JOIN**

SELECT p.product\_name, p.price, o.order\_id, o.order\_date, o.customer\_id FROM products p

LEFT JOIN orders o ON p.product\_id = o.product\_id;

**11. Right Join for Customer Contacts:**

**Use a RIGHT JOIN between 'contacts' and 'customers' to display:**

**- All customers, even if they don't have contact info**

CREATE TABLE contacts (

contact\_id INT PRIMARY KEY,

customer\_id INT,

phone VARCHAR(15),

email VARCHAR(100),

FOREIGN KEY (customer\_id) REFERENCES customers(customer\_id)

);

INSERT INTO contacts (contact\_id, customer\_id, phone, email) VALUES

(201, 1, '9876543210', 'aarthi@example.com'),

(202, 2, '9123456780', 'rahul@example.com'),

(203, 4, '9988776655', 'john@example.com');

SELECT c.customer\_name, ct.phone, ct.email FROM contacts ct

RIGHT JOIN customers c ON ct.customer\_id = c.customer\_id;

**12. Full Outer Join for Suppliers and Products:**

**Use a FULL OUTER JOIN to list:**

**- All suppliers and products**

**- Match supplier to product, or show NULLs where not available**

CREATE TABLE suppliers (

supplier\_id INT PRIMARY KEY,

supplier\_name VARCHAR(100)

);

alter table products add column supplier\_id int;

INSERT INTO suppliers (supplier\_id, supplier\_name) VALUES

(1, 'Acme Corp'),

(2, 'Tech Distributors'),

(3, 'Global Supplies');

update products set supplier\_id=1 where product\_id=1;

update products set supplier\_id=1 where product\_id=2;

update products set supplier\_id=2 where product\_id=3;

update products set supplier\_id=3 where product\_id=4;

update products set supplier\_id=3 where product\_id=5;

SELECT s.supplier\_name, p.product\_name FROM suppliers s

LEFT JOIN products p ON s.supplier\_id = p.supplier\_id

UNION

SELECT s.supplier\_name, p.product\_name FROM suppliers s

RIGHT JOIN products p ON s.supplier\_id = p.supplier\_id;

**13. Cross Join for Offers:**

**Suppose you have tables 'products' and 'offers'.**

**Write a CROSS JOIN to show:**

**- All possible combinations of products and offers**

CREATE TABLE offers (

offer\_id INT PRIMARY KEY,

offer\_description VARCHAR(100)

);

INSERT INTO offers (offer\_id, offer\_description) VALUES

(101, '10% Off'),

(102, 'Buy 1 Get 1'),

(103, 'Free Shipping');

SELECT p.product\_name, o.offer\_description FROM products p

CROSS JOIN offers o;

**14. Join with Aggregation:**

**Join 'orders' and 'products', then group by product category and:**

**- Show total quantity sold and average price per category**

alter table products add column category varchar(50);

alter table orders add column quantity int;

update products set category = 'Computing' where product\_id=1;

update products set category = 'Mobile Devices' where product\_id=2;

update products set category = 'Mobile Devices' where product\_id=3;

update products set category = 'Accessories' where product\_id=4;

update products set category = 'Accessories' where product\_id=5;

update orders set quantity=3 where order\_id=101;

update orders set quantity=5 where order\_id=102;

update orders set quantity=2 where order\_id=103;

update orders set quantity=4 where order\_id=104;

update orders set quantity=6 where order\_id=105;

SELECT p.category, SUM(o.quantity) AS total\_quantity\_sold, AVG(p.price) AS average\_price

FROM orders o

JOIN products p ON o.product\_id = p.product\_id

GROUP BY p.category;

**15. Demo: Join with Grouping and Filter:**

**Join 'students' and 'marks' tables.**

**Display:**

**- Student name**

**- Average marks**

**- Filter to show only students with average marks > 75**

CREATE TABLE marks (

mark\_id INT PRIMARY KEY,

student\_id INT,

subject VARCHAR(50),

mark INT,

FOREIGN KEY (student\_id) REFERENCES students(id)

);

INSERT INTO marks VALUES

(1, 1, 'Math', 95),(2, 1, 'Science', 85),

(3, 2, 'Math', 60),(4, 2, 'Science', 70),

(5, 3, 'Math', 75),(6, 3, 'Science', 95),

(7, 4, 'Math', 50),(8, 4, 'Science', 75),

(9, 5, 'Math', 40),(10, 5, 'Science', 80);

SELECT s.name, AVG(m.mark) AS average\_marks FROM students s

JOIN marks m ON s.id = m.student\_id

GROUP BY s.id, s.name

HAVING AVG(m.mark) > 75;

# DAY 5 - Assignment(1)

1. **Differentiate between SQL and NoSQL. Provide two advantages and two disadvantages of each with real-world examples.**
2. SQL (Structured Query Language) databases are relational. They store data in tables with rows and columns (like Excel). You define the structure (schema) before inserting data.

****Examples****: MySQL, PostgreSQL, Oracle, Microsoft SQL Server

* Advantages:
* Structured & Reliable: Perfect when data follows a clear structure (like student records or invoices).
* Powerful Querying: You can write complex queries using JOINs, filters, and aggregations.
* Disadvantages:
* Less Flexible: Hard to adapt if your data model changes frequently.
* Scalability is Vertical: Needs a more powerful server instead of just adding more machines.

1. NoSQL (Not Only SQL) databases are non-relational. They store data in flexible formats like JSON, documents, key-value pairs, or graphs. You don’t need to define a rigid schema upfront.

****Examples****: MongoDB, Firebase, Cassandra, Redis

* Advantages:
* Schema-less & Flexible: Great for rapidly changing or unstructured data.
* Horizontally Scalable: Easily handles massive data by adding more servers.
* Disadvantages:
* Lacks Complex Joins: Not ideal for multi-table relational logic.
* Less Mature for Transactions: Not always reliable for financial-like operations requiring strong consistency.

| **Feature** | **SQL** | **NoSQL** |
| --- | --- | --- |
| Structure | Table-based, fixed schema | Document/Key-Value/Graph-based |
| Best For | Structured, consistent data | Unstructured, flexible data |
| Example Use Case | Bank, ERP system | Social media, real-time chat app |
| Advantage 1 | Strong data integrity | Flexible schema |
| Advantage 2 | Complex query capabilities | Scalable across many servers |
| Disadvantage 1 | Hard to change structure | Weaker on relational logic |
| Disadvantage 2 | Not ideal for big data scaling | Not ideal for transactions |

1. **Given the below unnormalized data, convert it to 1NF, 2NF, and 3NF: Student (StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone) :**

**1NF:**

* Each field must contain **atomic (indivisible)** values.
* Each row should only contain one StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone.
* Assuming that Now each row represents **one student in one course**, with **no multi-valued attributes**

****2NF:****

* All non-key attributes must depend on the **entire primary key** (no partial dependency).
* In the current table, the **composite key** is: (StudentID, CourseID)
* But Name depends only on StudentID,
* and CourseName, InstructorName, InstructorPhone depend only on CourseID.

So we have to decompose the table into:

1. Student (StudentID, Name)
2. Course (CourseID, CourseName,InstructorName, InstructorPhone)
3. Enrollment (StudentID, CourseID)

* Now, every non-key attribute in each table depends on **the whole primary key** of that table.

**3NF:**

* Non-key attributes must not depend on other non-key attributes.
* In Course, InstructorPhone depends on InstructorName, not directly on CourseID.
* So we separate **Instructor** info into its own table.

So we have to decompose the table into:

1. Student (StudentID, Name)
2. Course (CourseID, CourseName)
3. Instructor(InstructorName, InstructorPhone)
4. Enrollment (StudentID, CourseID)
5. **5 marks)**
6. **Create a database named StudentDB.**

CREATE DATABASE StudentDB;

USE StudentDB;

1. **Create a table Students with fields: StudentID, Name, DOB, Email.**

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Email VARCHAR(100));

1. **Rename the table to Student\_Info.**

RENAME TABLE Students TO Student\_Info;

1. **Add a column PhoneNumber.**

ALTER TABLE Student\_Info

ADD PhoneNumber VARCHAR(15);

1. **Drop the table.**

DROP TABLE Student\_Info;

1. **(5 marks)**

CREATE TABLE Student\_Info (

StudentID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Email VARCHAR(100),

PhoneNumber VARCHAR(15));

1. **Insert 3 student records into Student\_Info.**

INSERT INTO Student\_Info VALUES

(1, 'Aarthi', '2001-04-15', 'aarthi@gmail.com', '9876543210'),

(2, 'Dhivya', '1999-11-23', 'dhiv23@gmail.com', '9123456780'),

(3, 'Charulatha', '2003-06-10', 'charu10latha@yahoo.com', '9988776655');

1. **Update one student's phone number.**

UPDATE Student\_Info SET PhoneNumber = '9000000001' WHERE StudentID = 1;

1. **Delete one student whose email ends with @gmail.com.**

DELETE FROM Student\_Info WHERE Email LIKE ['%@gmail.com'](mailto:'%@gmail.com') LIMIT 1;

1. **Retrieve only names and emails of students born after the year 2000.**

SELECT Name, Email FROM Student\_Info WHERE YEAR(DOB) > 2000;

1. **Retrieve distinct domain names from the email column.**

SELECT DISTINCT SUBSTRING\_INDEX(Email, '@', -1) AS Domain

FROM Student\_Info;

1. **(5 marks)**
2. **Retrieve students with names starting with 'A'.**

SELECT \* FROM Student\_Info WHERE Name LIKE 'A%';

1. **Retrieve students with phone number between 9000000000 and 9999999999.**

SELECT \* FROM Student\_Info

WHERE PhoneNumber BETWEEN '9000000000' AND '9999999999';

1. **Retrieve students using IN operator on city names.**

SELECT \* FROM Student\_Info WHERE City IN ('Delhi', 'Chennai', 'Mumbai');

1. **Use AND, OR to filter students based on age and email provider.**

SELECT \* FROM Student\_Info

WHERE TIMESTAMPDIFF(YEAR, DOB, CURDATE()) < 25

AND (Email LIKE '%@gmail.com' OR Email LIKE '%@yahoo.com');

1. **Use table and column aliasing in a query to get all student names and DOBs.**

SELECT s.Name AS StudentName, s.DOB AS DateOfBirth

FROM Student\_Info AS s;

1. **(5 marks)**

**Create a new table Marks(StudentID, Subject, Marks). Insert at least 3 rows.**

CREATE TABLE Marks (

StudentID INT,

Subject VARCHAR(100),

Marks INT

);

INSERT INTO Marks (StudentID, Subject, Marks) VALUES

(1, 'Math', 85),

(2, 'Science', 72),

(3, 'History', 65),

(1, 'Science', 78),

(2, 'Math', 55),

(3, 'Science', 92);

1. **Display student IDs and their subjects where marks > 70.**

SELECT StudentID, Subject FROM Marks

WHERE Marks > 70;

1. **Display subjects with average marks.**

SELECT Subject, AVG(Marks) AS AverageMarks FROM Marks

GROUP BY Subject;

**c) Filter subjects with average marks between 60 and 90.**

SELECT Subject, AVG(Marks) AS AverageMarks FROM Marks

GROUP BY Subject HAVING AVG(Marks) BETWEEN 60 AND 90;

1. **(5 marks)**
2. **Get the current date and format it as "YYYY-MM-DD".**

SELECT DATE\_FORMAT(CURDATE(), '%Y-%m-%d') AS FormattedDate;

1. **Extract month and year from a DOB column.**

SELECT MONTH(DOB) AS BirthMonth, YEAR(DOB) AS BirthYear

FROM Student\_Info;

1. **Convert a student's name to uppercase.**

SELECT UPPER(Name) AS UppercaseName FROM Student\_Info;

1. **Round off marks to 2 decimal places.**

SELECT StudentID, Subject, ROUND(Marks, 2) AS RoundedMarks

FROM Marks;

**e) Use system function to return user name or current database.**

SELECT USER() AS CurrentUser;

SELECT DATABASE() AS CurrentDatabase;

1. **(5 marks)**
2. **Display total marks of each student.**

SELECT StudentID, SUM(Marks) AS TotalMark FROM Marks

GROUP BY StudentID;

1. **Display subject-wise highest mark.**

SELECT Subject, MAX(Marks) AS HighestMark FROM Marks

GROUP BY Subject;

1. **Use GROUP BY and HAVING to display subjects with average marks > 75.**

SELECT Subject, AVG(Marks) AS AverageMarks FROM Marks

GROUP BY Subject HAVING AVG(Marks) > 75;

1. **(5 marks)**

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(100));

CREATE TABLE Courses (

CourseID INT PRIMARY KEY,

CourseName VARCHAR(100));

INSERT INTO Students (StudentID, Name) VALUES

(1, 'Aarthi'),

(2, 'Dhiya'),

(3, 'Charulatha');

INSERT INTO Courses (CourseID, CourseName) VALUES

(101, 'Math'),

(102, 'Science'),

(103, 'History');

1. **Inner Join to retrieve students and their courses.**

SELECT s.StudentID, s.Name, c.CourseID, c.CourseName FROM Students s

INNER JOIN Courses c ON s.StudentID = c.CourseID;

1. **Left Join to get all students even if not enrolled.**

SELECT s.StudentID, s.Name, c.CourseName FROM Students s

LEFT JOIN Courses c ON s.StudentID = c.CourseID;

1. **Right Join to get all courses even if no students.**

SELECT s.Name, c.CourseID, c.CourseName FROM Students s

RIGHT JOIN Courses c ON s.StudentID = c.CourseID;

1. **Full Outer Join equivalent using UNION.**

SELECT s.StudentID, s.Name, c.CourseName FROM Students s

LEFT JOIN Courses c ON s.StudentID = c.CourseID

UNION

SELECT s.StudentID, s.Name, c.CourseName FROM Students s

RIGHT JOIN Courses c ON s.StudentID = c.CourseID;

1. **Cross Join to show all combinations.**

SELECT s.StudentID, s.Name, c.CourseID, c.CourseName FROM Students s

CROSS JOIN Courses c;

1. **(5 marks)**
2. **Students who scored more than average in 'Maths'.**

SELECT s.StudentID, s.Name

FROM Students s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Subject = 'Maths'

AND m.Marks > (

SELECT AVG(Marks) FROM Marks WHERE Subject = 'Maths'

);

1. **Students not in the Marks table.**

SELECT s.StudentID, s.Name

FROM Students s

LEFT JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.StudentID IS NULL;

1. **Use EXISTS to get students with at least one subject.**

SELECT s.StudentID, s.Name FROM Students s

WHERE EXISTS (SELECT 1 FROM Marks m WHERE m.StudentID = s.StudentID);

1. **Use ALL to find those scoring more than all in 'Science'.**

SELECT s.StudentID, s.Name FROM Students s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Subject = 'Science' AND m.Marks > ALL(SELECT Marks FROM Marks WHERE Subject = 'Science' AND m.StudentID != StudentID );

1. **Use ANY for students scoring better than some in 'English'.**

INSERT INTO Marks VALUES

(1, 'English', 80),

(2, 'English', 65),

(3, 'English', 70);

SELECT s.StudentID, s.Name

FROM Students s

JOIN Marks m ON s.StudentID = m.StudentID

WHERE m.Subject = 'English'

AND m.Marks > ANY (

SELECT Marks FROM Marks WHERE Subject = 'English' AND m.StudentID != StudentID

);

1. **(5 marks)**
2. **UNION of student names from two tables.**

SELECT Name FROM Students

UNION

SELECT Name FROM Student\_Info;

1. **INTERSECT to find common students.**

SELECT Name FROM Students

WHERE Name IN (SELECT Name FROM Student\_Info);

1. **EXCEPT to list students in Students but not in Marks.**

SELECT s.Name FROM Students s

LEFT JOIN Student\_Info si ON s.Name = si.Name

WHERE si.Name IS NULL;

1. **MERGE concept or simulate with UPDATE and INSERT.**

UPDATE Students SET Name = 'Dhivya Updated' WHERE StudentID = 2;

INSERT INTO Students (StudentID, Name)

SELECT 4, 'NewStudent' WHERE NOT EXISTS ( SELECT 1 FROM Students WHERE StudentID = 4);

**e) Correlated subquery to list students with above average per subject.**

SELECT StudentID, Name, DOB FROM Student\_Info s

WHERE DOB > ( SELECT DATE(AVG(DATE\_FORMAT(DOB, '%Y%m%d') + 0)) FROM Student\_Info );

# DAY 5- Assignment(2)

1. **Explain with examples the scenarios where NoSQL is preferred over SQL. Discuss types of NoSQL databases and suggest a real-time application for each.**

NoSQL over SQL:

* Flexible Schema: No fixed table structure.

E.g., Product catalog where items have different attributes.

* High Scalability: Easily handles huge data by adding more servers.

E.g., Netflix storing millions of viewing logs.

* Fast Reads/Writes: Optimized for real-time performance.

E.g., WhatsApp chat messages.

* Frequent Schema Changes: Ideal for agile/rapidly evolving apps.

E.g., Startups iterating on features.

### **Types of NoSQL & Real-Time Uses**

| Type | Example DB | Best Used For | Real Example |
| --- | --- | --- | --- |
| **Document** | MongoDB | User profiles, dynamic content | Pinterest, Instagram |
| **Key-Value** | Redis | Caching, session management | Amazon cart sessions |
| **Column** | Cassandra | High-speed logging, time-series data | IoT devices, analytics |
| **Graph** | Neo4j | Relationship-heavy data | LinkedIn friend networks |

1. **Customer (CustomerID, Name, Orders (OrderID, ProductID, Quantity, ProductName)). Normalize the data up to BCNF with appropriate table structures.**

**1NF:**

* This table contains **repeating groups(**multiple orders inside a single row for a customer).
* 1NF remove repeating groups ( each order becomes a separate row).

CREATE TABLE Customer (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100));

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

ProductID INT,

Quantity INT,

ProductName VARCHAR(100),

FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID));

**2NF:**

* 2NF removes **partial dependencies (**values depending only on part of the primary key).
* In Orders, ProductName depends only on ProductID, not on the full row.

CREATE TABLE Product (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100));

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

ProductID INT,

Quantity INT,

FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID),

FOREIGN KEY (ProductID) REFERENCES Product(ProductID));

**3NF:**

* Already in 3NF — no **transitive dependencies** exist now (i.e., all non-key attributes depend only on primary keys).

**BCNF:**

* Also satisfies BCNF — every determinant is a candidate key in all tables.

1. **(5 marks)**
2. **Create a database RetailDB and design a schema for Customers, Orders, and Products with primary and foreign keys.**

CREATE DATABASE RetailDB;

USE RetailDB;

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY AUTO\_INCREMENT,

Name VARCHAR(100),

Email VARCHAR(100),

ReferredBy INT,

FOREIGN KEY (ReferredBy) REFERENCES Customers(CustomerID));

CREATE TABLE Products (

ProductID INT PRIMARY KEY AUTO\_INCREMENT,

ProductName VARCHAR(100),

Category VARCHAR(50),

Price DECIMAL(10,2));

CREATE TABLE Orders (

OrderID INT PRIMARY KEY AUTO\_INCREMENT,

CustomerID INT,

ProductID INT,

OrderDate DATE,

Quantity INT,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID));

INSERT INTO Customers (Name, Email) VALUES

('Aarthi', 'aarthi@gmail.com'),

('Dhivya', 'dhivya@gmail.com'),

('Charulatha', ['charu@gmail.com');](mailto:'charu@example.com');)

INSERT INTO Products (ProductName, Category, Price) VALUES

('Laptop', 'Electronics', 70000),

('Book', 'Stationery', 500),

('Smartphone', 'Electronics', 30000),

('Pen', 'Stationery', 20),

('Headphones', 'Electronics', 2000);

1. **Implement a check constraint on Quantity (>0) in Orders.**

ALTER TABLE Orders ADD CONSTRAINT Check\_Quality

CHECK (Quantity>0);

1. **Alter the Products table to add 'Discount' column and update some values.**

ALTER TABLE Products ADD COLUMN Discount DECIMAL(5,2);

UPDATE Products SET Discount = 10.0 WHERE ProductID = 1;

UPDATE Products SET Discount = 5.0 WHERE ProductID = 2;

UPDATE Products SET Discount = 0.0 WHERE ProductID = 3;

UPDATE Products SET Discount = 10.0 WHERE ProductID = 4;

UPDATE Products SET Discount = 5.0 WHERE ProductID = 5;

1. **Using the above schema:**
2. **Insert 3 sample orders per customer.**

INSERT INTO Orders (CustomerID, ProductID, OrderDate, Quantity) VALUES

(1, 1, CURDATE(), 2),

(1, 2, CURDATE(), 3),

(1, 3, CURDATE(), 1),

(2, 2, CURDATE(), 6),

(2, 4, CURDATE(), 1),

(2, 5, CURDATE(), 2),

(3, 3, CURDATE(), 5),

(3, 1, CURDATE(), 1),

(3, 4, CURDATE(), 7);

1. **Update prices with 10% increase where quantity sold > 5.**

UPDATE Products SET Price = Price \* 1.10

WHERE ProductID IN ( SELECT ProductID FROM Orders

GROUP BY ProductID HAVING SUM(Quantity) > 5);

**c) Delete orders where the product has never been sold.**

DELETE FROM Products

WHERE ProductID NOT IN ( SELECT DISTINCT ProductID FROM Orders);

1. **Retrieve the following:**
2. **Customers who ordered more than 3 different products.**

SELECT o.CustomerID, COUNT(DISTINCT o.ProductID) AS ProductCount FROM Orders o

GROUP BY o.CustomerID

HAVING COUNT(DISTINCT o.ProductID) > 3;

1. **Products not ordered by any customer.**

SELECT p.ProductID, p.ProductName

FROM Products p

LEFT JOIN Orders o ON p.ProductID = o.ProductID

WHERE o.ProductID IS NULL;

1. **Count of orders placed by each customer in the last 30 days.**

SELECT o.CustomerID, COUNT(\*) AS OrderCount

FROM Orders o

WHERE o.OrderDate >= CURDATE() - INTERVAL 30 DAY

GROUP BY o.CustomerID;

1. **(5 marks)**
2. **Use string functions to standardize and extract parts from customer email IDs.**

SELECT CustomerID, Email, LOWER(Email) AS LowercaseEmail, SUBSTRING\_INDEX(Email, '@', -1) AS Domain FROM Customers;

1. **Use date functions to compute days between order date and today.**

SELECT OrderID, OrderDate, DATEDIFF(CURDATE(), OrderDate) AS DaysSinceOrder FROM Orders;

1. **Use system functions to return current user and host.**

SELECT CURRENT\_USER() AS CurrentUser, USER() AS SessionUser, DATABASE() AS CurrentDatabase;

1. **Use nested functions to format a customer greeting string.**

SELECT CustomerID, CONCAT('Hello ', UPPER(Name), '! Your email is ', Email) AS Greeting FROM Customers;

1. **(5 marks)**
2. **Aggregate total revenue by product category.**

SELECT p.Category, SUM(p.Price \* o.Quantity) AS TotalRevenue FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category;

1. **Use GROUP BY with ROLLUP to compute subtotal and grand total sales.**

SELECT p.Category, SUM(p.Price \* o.Quantity) AS Revenue FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category WITH ROLLUP;

1. **Use HAVING clause to filter categories with revenue > 100000.**

SELECT p.Category, SUM(p.Price \* o.Quantity) AS Revenue FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category

HAVING SUM(p.Price \* o.Quantity) > 100000;

1. **(5 marks)**

UPDATE Customers SET ReferredBy = 1 WHERE CustomerID = 2;

UPDATE Customers SET ReferredBy = 2 WHERE CustomerID = 3;

1. **Self join to list customers referred by other customers.**

SELECT C1.Name AS ReferredCustomer, C2.Name AS Referrer

FROM Customers C1

JOIN Customers C2 ON C1.ReferredBy = C2.CustomerID;

1. **Equi join across Orders and Products.**

SELECT O.OrderID, C.Name AS CustomerName, P.ProductName, O.Quantity, P.Price, (O.Quantity \* P.Price) AS TotalAmount FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

JOIN Customers C ON O.CustomerID = C.CustomerID;

1. **Join Customers and Orders to display top 3 spenders using window function.**

SELECT \* FROM (

SELECT C.CustomerID, C.Name, SUM(O.Quantity \* P.Price) AS TotalSpent, RANK() OVER (ORDER BY SUM(O.Quantity \* P.Price) DESC) AS RankPos FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

JOIN Customers C ON O.CustomerID = C.CustomerID

GROUP BY C.CustomerID, C.Name

) RankedSpenders

WHERE RankPos <= 3;

1. **LEFT OUTER JOIN with WHERE NULL to identify inactive customers.**

SELECT C.CustomerID, C.Name FROM Customers C LEFT JOIN Orders O ON C.CustomerID = O.CustomerID

WHERE O.OrderID IS NULL;

1. **Cross join for all product combinations in a bundle offer.**

SELECT P1.ProductName AS Product1, P2.ProductName AS Product2 FROM Products P1

CROSS JOIN Products P2

WHERE P1.ProductID < P2.ProductID;

1. **(5 marks)**
2. **Correlated subquery to get customers whose order amount exceeds their average.**

SELECT DISTINCT C.CustomerID, C.Name FROM Customers C

JOIN Orders O1 ON C.CustomerID = O1.CustomerID

JOIN Products P1 ON O1.ProductID = P1.ProductID

WHERE (O1.Quantity \* P1.Price) >(

SELECT AVG(O2.Quantity \* P2.Price) FROM Orders O2

JOIN Products P2 ON O2.ProductID = P2.ProductID

WHERE O2.CustomerID = C.CustomerID);

1. **Subquery using EXISTS to find customers with at least 2 different products.**

SELECT C.CustomerID, C.Name FROM Customers C

WHERE EXISTS ( SELECT 1 FROM Orders O

WHERE O.CustomerID = C.CustomerID

GROUP BY O.CustomerID

HAVING COUNT(DISTINCT O.ProductID) >= 2 );

1. **Use ALL to find customers who ordered more than every other customer.**

SELECT C.CustomerID, C.Name

FROM Customers C

WHERE (

SELECT COUNT(\*)

FROM Orders O

WHERE O.CustomerID = C.CustomerID

) > ALL (

SELECT COUNT(\*)

FROM Orders O

GROUP BY O.CustomerID

HAVING O.CustomerID != C.CustomerID

);

1. **Use ANY to find products costlier than some in category 'Electronics'.**

SELECT ProductName, Price FROM Products

WHERE Price > ANY ( SELECT Price FROM Products WHERE Category = 'Electronics');

1. **Nested subquery to list top 3 best-selling products.**

SELECT ProductID, ProductName, TotalSold FROM (SELECT P.ProductID, P.ProductName, SUM(O.Quantity) AS TotalSold, RANK() OVER (ORDER BY SUM(O.Quantity) DESC) AS rnk FROM Orders O

JOIN Products P ON O.ProductID = P.ProductID

GROUP BY P.ProductID, P.ProductName

) RankedProducts

WHERE rnk <= 3;

1. **Simulate INTERSECT using INNER JOIN on two customer segments.**

SELECT Name FROM Customers WHERE Email LIKE '%gmail.com'

INTERSECT

SELECT Name FROM Student\_Info WHERE Email LIKE '%gmail.com';

1. **Use EXCEPT to find products in inventory not yet ordered.**

SELECT P.ProductID, P.ProductName FROM Products P

LEFT JOIN Orders O ON P.ProductID = O.ProductID

WHERE O.ProductID IS NULL;

1. **Simulate MERGE: If customer exists, update; else insert.**

INSERT INTO Customers (CustomerID, Name, Email) VALUES

(1, 'Aarthi Updated', 'aarthi\_new@gmail.com')

ON DUPLICATE KEY UPDATE

Name = 'Aarthi Updated', Email = 'aarthi\_new@gmail.com';

1. **Use UNION to combine two regional customer tables.**

SELECT Name, Email FROM Customers WHERE Email LIKE '%gmail.com'

UNION

SELECT Name, Email FROM Student\_Info WHERE Email LIKE '%yahoo.com';

**e) Write a WITH CTE that ranks customers by total spend and filters top 5.**

WITH CustomerSpending AS (

SELECT C.CustomerID, C.Name, SUM(O.Quantity \* P.Price) AS TotalSpend,

RANK() OVER (ORDER BY SUM(O.Quantity \* P.Price) DESC) AS rnk

FROM Customers C

JOIN Orders O ON C.CustomerID = O.CustomerID

JOIN Products P ON O.ProductID = P.ProductID

GROUP BY C.CustomerID, C.Name

)

SELECT \* FROM CustomerSpending

WHERE rnk <= 5;

# DAY 5 - Assignment(3)

**Querying Data by Using Subqueries - Examples**

QUESTIONS

1) Querying Data by Using Subqueries

2) Querying Data by Using Subqueries Using the EXISTS,

3) Querying Data by Using Subqueries using ANY,

4) Querying Data by Using Subqueries using ALL Keywords

5) Querying Data by Using Subqueries using Using Nested Subqueries

6) Querying Data by Using Subqueries Using Correlated Subqueries

7) Querying Data by Using Subqueries Using UNION,

8) Querying Data by Using Subqueries using INTERSECT,

9) Querying Data by Using Subqueries using EXCEPT,

10)Querying Data by Using Subqueries using MERGE"

ANSWER QUERIES:

Create and use the database

CREATE DATABASE IF NOT EXISTS CompanyDB;

USE CompanyDB;

Drop old tables

DROP TABLE IF EXISTS employees, departments, job\_history, new\_employees, locations, countries, suppliers, products, categories;

Create schema

CREATE TABLE departments (

department\_id INT PRIMARY KEY,

department\_name VARCHAR(50),

location\_id INT

);

CREATE TABLE employees (

employee\_id INT PRIMARY KEY,

name VARCHAR(50),

salary INT,

department\_id INT,

job\_id VARCHAR(10),

FOREIGN KEY (department\_id) REFERENCES departments(department\_id)

);

CREATE TABLE job\_history (

employee\_id INT,

department\_id INT,

job\_id VARCHAR(10)

);

CREATE TABLE locations (

location\_id INT PRIMARY KEY,

city VARCHAR(50)

);

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(50),

category\_id INT,

supplier\_id INT

);

CREATE TABLE categories (

category\_id INT PRIMARY KEY,

category\_name VARCHAR(50)

);

CREATE TABLE suppliers (

supplier\_id INT PRIMARY KEY,

country\_id INT

);

CREATE TABLE countries (

country\_id INT PRIMARY KEY,

country\_name VARCHAR(50)

);

CREATE TABLE new\_employees (

employee\_id INT,

name VARCHAR(50),

salary INT,

department\_id INT

);

Insert sample data

INSERT INTO departments VALUES

(1, 'HR', 100),

(2, 'Engineering', 101),

(3, 'Sales', 102);

INSERT INTO locations VALUES

(100, 'New York'),

(101, 'Tokyo'),

(102, 'London');

INSERT INTO employees VALUES

(101, 'Alice', 80000, 2, 'ENG'),

(102, 'Bob', 50000, 2, 'ENG'),

(103, 'Charlie', 90000, 1, 'HR'),

(104, 'David', 60000, 3, 'SAL');

INSERT INTO job\_history VALUES

(101, 2, 'ENG'),

(103, 1, 'HR');

INSERT INTO categories VALUES

(1, 'Electronics'),

(2, 'Furniture');

INSERT INTO countries VALUES

(1, 'Japan'),

(2, 'Germany');

INSERT INTO suppliers VALUES

(1, 1),

(2, 2);

INSERT INTO products VALUES

(1, 'Laptop', 1, 1),

(2, 'TV', 1, 2);

INSERT INTO new\_employees VALUES

(101, 'Alice Johnson', 85000, 2),

(105, 'Eva', 70000, 3);

1. Basic Subquery: Get employees in HR department

SELECT name

FROM employees

WHERE department\_id = (

SELECT department\_id

FROM departments

WHERE department\_name = 'HR'

);

2. EXISTS: Get employees who have job history

SELECT name

FROM employees e

WHERE EXISTS (

SELECT 1

FROM job\_history j

WHERE j.employee\_id = e.employee\_id

);

3. ANY: Get employees earning more than someone in HR

SELECT name

FROM employees

WHERE salary > ANY (

SELECT salary

FROM employees

WHERE department\_id = 1

);

4. ALL: Get employees earning more than everyone in Sales

SELECT name

FROM employees

WHERE salary > ALL (

SELECT salary

FROM employees

WHERE department\_id = 3

);

1. Nested Subquery: Employees in NY-based department

SELECT name

FROM employees

WHERE department\_id = (

SELECT department\_id

FROM departments

WHERE location\_id = (

SELECT location\_id

FROM locations

WHERE city = 'New York'

)

);

1. Correlated Subquery: Earn more than dept avg

SELECT name

FROM employees e

WHERE salary > (

SELECT AVG(salary)

FROM employees

WHERE department\_id = e.department\_id

);

7. UNION: Combine employees from dept 1 and 2

(SELECT name FROM employees WHERE department\_id = 1)

UNION

(SELECT name FROM employees WHERE department\_id = 2);

8. INTERSECT (Conceptual only; not in MySQL)

SELECT e.employee\_id

FROM employees e

INNER JOIN job\_history j ON e.employee\_id = j.employee\_id

WHERE e.department\_id = 1;

9. EXCEPT (MySQL Alternative using NOT IN)

SELECT employee\_id

FROM employees

WHERE employee\_id NOT IN (

SELECT employee\_id

FROM job\_history

);

10. MERGE (Simulated using INSERT...ON DUPLICATE KEY)

INSERT INTO employees (employee\_id, name, salary, department\_id, job\_id)

VALUES

(101, 'Alice Johnson', 85000, 2, 'ENG'),

(105, 'Eva', 70000, 3, 'SAL') AS new\_vals

ON DUPLICATE KEY UPDATE

name = new\_vals.name,

salary = new\_vals.salary,

department\_id = new\_vals.department\_id;