**DAY 1 – ASSIGNMENT**

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

An algorithm is a finite step-by-step procedure to solve a problem or perform a computation.

Comparison of algorithms:

| **Feature** | **Brute Force** | **Divide & Conquer** | **Greedy** | **Dynamic Prog.** | **Backtracking** |
| --- | --- | --- | --- | --- | --- |
| Efficiency | Low | Moderate | High (for optimal cases) | High | Varies |
| Memory Use | Low | Moderate | Low | High (due to memoization) | Moderate |
| Suitable for | Simple problems | Complex recursive tasks | Optimization problems | Overlapping subproblems | Constraint-based problems |

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

| **Algorithm** | **Working Principle** | **Best Case** | **Average Case** | **Worst Case** | **Space Complexity** | **Use Case** |
| --- | --- | --- | --- | --- | --- | --- |
| **Bubble Sort** | Repeatedly swaps adjacent elements if they are in the wrong order | O(n) | O(n²) | O(n²) | O(1) | Educational use; simple but inefficient |
| **Selection Sort** | Selects the minimum and swaps it to the front in each iteration | O(n²) | O(n²) | O(n²) | O(1) | When memory is limited; small datasets |
| **Insertion Sort** | Builds sorted array by inserting elements at correct position | O(n) | O(n²) | O(n²) | O(1) | Small or nearly sorted arrays |
| **Merge Sort** | Recursively divides array, sorts halves, and merges them | O(n log n) | O(n log n) | O(n log n) | O(n) | Stable sort for large datasets, linked lists |
| **Quick Sort** | Divides array based on pivot, recursively sorts partitions | O(n log n) | O(n log n) | O(n²) | O(log n) | Best average-case; fast in practice |

Two best Sorting algorithms:

1. **Bubble Sort**

Repeatedly compares adjacent elements and swaps them if they are in the wrong order. Largest elements “bubble up” to the end.

EX: Educational purposes; inefficient for large datasets.

1. **Selection Sort**

Selects the minimum element from the unsorted part and swaps it with the element at the beginning.

1. Compare searching algorithm.

| **Algorithm** | **Description** | **Best Case** | **Average Case** | **Worst Case** | **Time Complexity** | **Use Case** |
| --- | --- | --- | --- | --- | --- | --- |
| **Linear Search** | Scans each element one by one until the target is found or end is reached. | O(1) | O(n) | O(n) | O(n) | Small or unsorted datasets |
| **Binary Search** | Repeatedly divides the sorted list and searches the appropriate half. | O(1) | O(log n) | O(log n) | O(log n) | Large, sorted datasets |
| **Jump Search** | Jumps ahead by fixed steps, then performs linear search within a block. | O(1) | O(√n) | O(√n) | O(√n) | Faster than linear on sorted arrays |
| **Interpolation Search** | Estimates the position of the target based on value range (like binary search). | O(1) | O(log log n) | O(n) | O(log log n) | Numerical data with uniform distribution |
| **Exponential Search** | Finds range by exponential steps, then uses binary search. | O(log i) | O(log i) | O(log n) | O(log n) | Infinite or large sorted arrays |
| **Ternary Search** | Divides array into 3 parts and recursively searches the correct third. | O(1) | O(log₃ n) | O(log n) | O(log n) | Similar to binary but less practical |

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

A **Binary Search Tree** is a binary tree where:

* Left child < parent
* Right child > parent

### Use of BST

* **Efficient Searching**: Average time complexity O(log n)
* **Efficient Insertion/Deletion**
* **Sorted Data Traversal**:Inorder traversal gives sorted order
* **Dynamic Dataset Handling**: Unlike arrays, size can grow/shrink

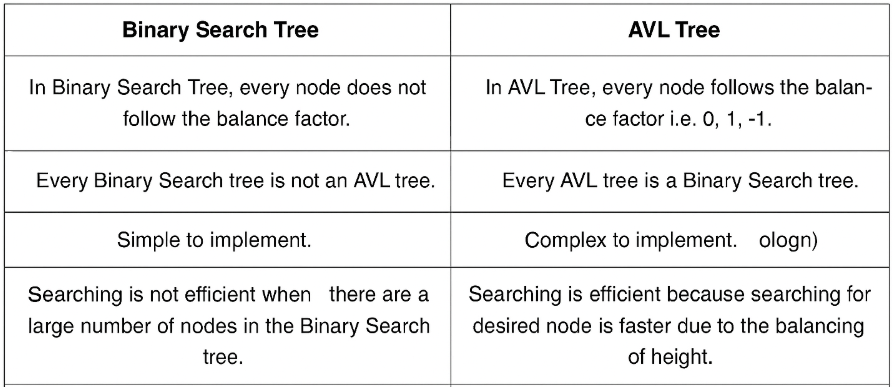
### AVL Tree

* **Self-balancing Binary Search Tree**
* Maintains **balance factor** (difference in heights of left and right subtrees) between -1 and +1
* Automatically **rotates** nodes to stay balanced after insert/delete

**Need for AVL Tree (Self-Balancing BST)**

 Can become unbalanced, resembling a linked list  
(e.g., inserting 1 → 2 → 3 → 4 → ... forms a right-skewed tree)

 Time complexity degrades to O(n)



**DAY 2 – ASSIGNMENT**

**Section 1: Managing Databases**

1. **Which of the following is NOT a system database in SQL Server?**  
   a) master  
   b) model  
   c) tempdb  
   d) userdb

**ANS:**- d) userdb

1. **Which system database stores all login accounts and configuration settings?**  
   a) tempdb  
   b) model  
   c) master  
   d) msdb

**ANS:**- c) master

1. **What is the purpose of the model database in SQL Server?**  
   a) Backup  
   b) Log storage  
   c) Template for new databases  
   d) System configuration

**ANS:**- c)Template for new databases

1. **What are the two main types of database files in SQL Server?**  
   a) MDF and NDF  
   b) LDF and MDF  
   c) NDF and BAK  
   d) BAK and TRN

**ANS:**- c) NDF and BAK

1. **Which SQL command is used to create a new database?**  
   a) MAKE DATABASE  
   b) NEW DATABASE  
   c) CREATE DATABASE  
   d) INIT DATABASE

**ANS:**- c)CREATEDATABASE

1. **What happens when you execute DROP DATABASE SalesDB?**  
   a) SalesDB is backed up  
   b) SalesDB is renamed  
   c) SalesDB is deleted permanently  
   d) SalesDB is restored

**ANS:**- c) SalesDB is deleted permanently

1. **Which command renames a database in SQL Server?**  
   a) RENAME DATABASE old\_name TO new\_name  
   b) ALTER DATABASE old\_name MODIFY NAME = new\_name  
   c) UPDATE DATABASE NAME  
   d) SET DATABASE NAME

**ANS:**- b) ALTER DATABASE old\_name MODIFY NAME = new\_name

**Section 2: Managing Tables**

1. **Which data type should be used to store a date of birth?**  
   a) VARCHAR  
   b) DATE  
   c) INT  
   d) TEXT

**ANS:**- b) DATE

1. **What command is used to create a table?**  
   a) MAKE TABLE  
   b) INSERT TABLE  
   c) CREATE TABLE  
   d) DEFINE TABLE

**ANS:**- c) CREATE TABLE

1. **How do you add a new column to an existing table?**  
   a) ALTER TABLE table\_name ADD column\_name datatype  
   b) MODIFY TABLE table\_name ADD column\_name  
   c) UPDATE TABLE table\_name ADD column\_name  
   d) APPEND column\_name TO table\_name

**ANS:**- a) ALTER TABLE table\_name ADD column\_name datatype

1. **Which command is used to rename a table?**  
   a) RENAME TABLE old\_name TO new\_name  
   b) ALTER TABLE old\_name RENAME TO new\_name  
   c) EXEC sp\_rename 'old\_name', 'new\_name'  
   d) MODIFY TABLE RENAME

**ANS:**- b) ALTER TABLE old\_name RENAME TO new\_name

1. **What is the command to delete a table permanently?**  
   a) DELETE TABLE table\_name  
   b) ERASE TABLE table\_name  
   c) DROP TABLE table\_name  
   d) REMOVE TABLE table\_name

**ANS:**- c) DROP TABLE table\_name

**Section 3: DML - Manipulating Data**

1. **Which command adds data into a table?**  
   a) INSERT INTO  
   b) ADD ROW  
   c) CREATE DATA  
   d) APPEND TO

**ANS:**- a) INSERT INTO

1. **Which clause is used to update data in a table?**  
   a) MODIFY  
   b) UPDATE  
   c) CHANGE  
   d) SET TABLE

**ANS:**- b) UPDATE

1. **What does the DELETE statement do?**  
   a) Removes a column  
   b) Removes all data from a table  
   c) Removes specific rows  
   d) Deletes the table schema

**ANS:**- b) Removes all data from a table

1. **Which clause is used to filter rows in a SELECT statement?**  
   a) HAVING  
   b) SELECT  
   c) WHERE  
   d) ORDER BY

**ANS:**- b) SELECT

1. **Which keyword ensures no duplicate records are returned?**  
   a) UNIQUE  
   b) NO\_REPEAT  
   c) DISTINCT  
   d) ONLY

**ANS:**- c) DISTINCT

1. **What does the LIKE keyword do in SQL?**  
   a) Finds exact matches  
   b) Finds pattern-based matches  
   c) Sorts records  
   d) Deletes matches

**ANS:**- b) Finds pattern-based matches

1. **Which operator is used to combine multiple conditions in a WHERE clause?**  
   a) TO  
   b) WITH  
   c) AND / OR  
   d) IF / ELSE

**ANS:**- c) AND / OR

1. **What does the BETWEEN operator do?**  
   a) Compares text fields  
   b) Finds rows outside a range  
   c) Filters values within a range  
   d) Joins tables

**ANS:**- c) Filters values within a range

**DAY 3 – ASSIGNMENT**

* 1. Create database HexaAirlines

CREATE DATABASE HexaAirlines;

USE HexaAirlines;

* 1. Create tables flight, pilots, airhostess , foodTeam, customers.

CREATE TABLE Flight (

FlightID INT PRIMARY KEY,

FlightName VARCHAR(50),

Destination VARCHAR(50)

);

CREATE TABLE Pilots (

PilotID INT PRIMARY KEY,

PilotName VARCHAR(50),

FlightID INT,

FOREIGN KEY (FlightID) REFERENCES Flight(FlightID)

);

CREATE TABLE AirHostess (

HostessID INT PRIMARY KEY,

HostessName VARCHAR(50),

FlightID INT,

FOREIGN KEY (FlightID) REFERENCES Flight(FlightID)

);

CREATE TABLE FoodTeam (

FoodID INT PRIMARY KEY,

FoodType VARCHAR(20)

);

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

CustomerName VARCHAR (50),

FlightID INT,

FoodID INT,

FOREIGN KEY (FlightID) REFERENCES Flight(FlightID),

FOREIGN KEY (FoodID) REFERENCES FoodTeam(FoodID)

);

* 1. Fill 6 to 7 rows in these and have primary and foreign keys.

INSERT INTO Flight VALUES

(1, 'IndiGo', 'Bangalore'),

(2, 'JetAirways', 'Mumbai'),

(3, 'AirIndia', 'Goa'),

(4, 'IndiGo', 'Chennai'),

(5, 'JetAirways', 'Delhi'),

(6, 'AirIndia', 'Kolkata');

INSERT INTO Pilots VALUES

(101, 'Captain Rao', 1),

(102, 'Captain Sharma', 2),

(103, 'Captain Iyer', 3),

(104, 'Captain Khan', 4),

(105, 'Captain Das', 5),

(106, 'Captain Singh', 6);

INSERT INTO AirHostess VALUES

(201, 'Anjali', 1),

(202, 'Divya', 2),

(203, 'Rekha', 3),

(204, 'Priya', 4),

(205, 'Neha', 5),

(206, 'Sara', 6);

INSERT INTO FoodTeam VALUES

(1, 'Veg'),

(2, 'Non-Veg');

INSERT INTO Customers VALUES

(301, 'Arun', 1, 1),

(302, 'Bala', 2, 2),

(303, 'Chitra', 3, 1),

(304, 'Deepak', 1, 2),

(305, 'Esha', 2, 1),

(306, 'Farhan', 3, 2),

(307, 'Gita', 3, 1);

* 1. Select customers going to Bangalore in indigo.

SELECT c.CustomerName

FROM Customers c

JOIN Flight f ON c.FlightID = f.FlightID

WHERE f.Destination = 'Bangalore' AND f.FlightName = 'IndiGo';

* 1. Select customers going to mumbai in jetairways.

SELECT c.CustomerName

FROM Customers c

JOIN Flight f ON c.FlightID = f.FlightID

WHERE f.Destination = 'Mumbai' AND f.FlightName = 'JetAirways';

* 1. Select customers going to goa in airindia and having veg food.

SELECT c.CustomerName

FROM Customers c

JOIN Flight f ON c.FlightID = f.FlightID

JOIN FoodTeam ft ON c.FoodID = ft.FoodID

WHERE f.Destination = 'Goa' AND f.FlightName = 'AirIndia' AND ft.FoodType = 'Veg';

**DAY 4 – ASSIGNMENT**

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.

ANS:-

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

Email VARCHAR(100) NOT NULL UNIQUE,

Marks INT CHECK (Marks >= 0 AND Marks <= 100)

);

INSERT INTO Students VALUES

(1, 'Arun', 'arun@example.com', 85),

(2, 'Bala', 'bala@example.com', 90),

(3, 'Chitra', 'chitra@example.com', 78),

(4, 'Deepa', 'deepa@example.com', 92),

(5, 'Eshan', 'eshan@example.com', 88);

UPDATE Students

SET Marks = 95

WHERE StudentID = 2;

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

ANS:-

SELECT

full\_name,

SUBSTRING\_INDEX(full\_name, ' ', 1) AS FirstName,

SUBSTRING\_INDEX(full\_name, ' ', -1) AS LastName,

LENGTH(SUBSTRING\_INDEX(full\_name, ' ', 1)) AS FirstNameLength,

LENGTH(SUBSTRING\_INDEX(full\_name, ' ', -1)) AS LastNameLength

FROM customers;

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

ANS:-

SELECT

sale\_date,

MONTHNAME(sale\_date) AS SaleMonth,

YEAR(sale\_date) AS SaleYear,

DATEDIFF(CURDATE(), sale\_date) AS DaysAgo

FROM sales;

4. Mathematical Functions on Salary:

In an 'employees' table, calculate:

- Salary after a 10% hike

- Round the salary to the nearest hundred

ANS:-

SELECT

emp\_id,

salary,

salary \* 1.10 AS HikedSalary,

ROUND(salary, -2) AS RoundedSalary

FROM employees;

5. System Function Check:

Retrieve:

- Current date and time

- Database name and logged-in user

ANS:-

SELECT

NOW() AS CurrentDateTime,

DATABASE() AS CurrentDatabase,

USER() AS LoggedInUser;

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 ProductName,

IFNULL(price, 'Not Available') AS PriceStatus

FROM products;

7. Aggregate Functions Practice:

From a 'transactions' table, get:

- Total sales

- Average sale value

- Maximum and minimum sale on a single transaction

ANS:-

SELECT

SUM(amount) AS TotalSales,

AVG(amount) AS AverageSale,

MAX(amount) AS MaxSale,

MIN(amount) AS MinSale

FROM transactions;

8. Grouping with Aggregation:

From a 'sales' table:

- Group by product category

- Show total sales and number of transactions in each category

ANS:-

SELECT

category,

SUM(sale\_amount) AS TotalSales,

COUNT(\*) AS TransactionCount

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

ANS:-

SELECT

c.CustomerName, o.OrderAmount

FROM orders o

INNER JOIN customers c ON o.CustomerID = c.CustomerID;

10. Left Join for Products with or without Orders:

Show all products with:

- Their order details (if available)

- Use LEFT JOIN

ANS:-

SELECT

p.product\_name, o.order\_id, o.order\_amount

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

ANS:-

SELECT

c.CustomerID, c.CustomerName, con.ContactNumber

FROM contacts con

RIGHT JOIN customers c ON c.CustomerID = con.CustomerID;

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

ANS:-

SELECT s.supplier\_id, s.supplier\_name, p.product\_name

FROM suppliers s

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

UNION

SELECT s.supplier\_id, 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

ANS:-

SELECT

p.product\_name, o.offer\_name

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

ANS:-

SELECT

p.category,

SUM(o.quantity) AS TotalQuantity,

AVG(p.price) AS AvgPrice

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

SELECT

s.Name,

AVG(m.Marks) AS AverageMarks

FROM students s

JOIN marks m ON s.StudentID = m.StudentID

GROUP BY s.StudentID

HAVING AVG(m.Marks) > 75;

**DAY 5 – ASSIGNMENT**

* + 1. Querying Data by Using Subqueries.

SELECT CustomerName FROM Customers

WHERE FlightID = (

SELECT FlightID FROM Customers

GROUP BY FlightID

ORDER BY COUNT(\*) DESC LIMIT 1

);

* + 1. Querying Data by Using Subqueries Using the EXISTS.

SELECT c.CustomerName FROM Customers c

WHERE EXISTS (

SELECT 1 FROM Customers c2

WHERE c.FlightID = c2.FlightID AND c2.FoodID = 1

);

* + 1. Querying Data by Using Subqueries using ANY.

SELECT CustomerName FROM Customers

WHERE FlightID = ANY (

SELECT FlightID FROM Customers WHERE FoodID = 2

);

* + 1. Querying Data by Using Subqueries using ALL Keywords.

SELECT CustomerName FROM Customers

WHERE FlightID = ALL (

SELECT FlightID FROM Flight

WHERE Destination != 'Delhi'

);

* + 1. Querying Data by Using Subqueries using Nested Subqueries.

SELECT CustomerName FROM Customers

WHERE FlightID IN (

SELECT FlightID FROM Flight WHERE Destination = (

SELECT Destination FROM (

SELECT Destination, COUNT(\*) AS dest\_count FROM Flight

GROUP BY Destination

ORDER BY dest\_count DESC LIMIT 1 OFFSET 1

) AS Sub

)

);

1. Querying Data by Using Subqueries Using Correlated Subqueries.

SELECT c.CustomerName FROM Customers c

WHERE c.FoodID = (

SELECT FoodID FROM Customers c2

WHERE c2.FlightID = c.FlightID

GROUP BY FoodID

ORDER BY COUNT(\*) DESC LIMIT 1

);

* 1. Querying Data by Using Subqueries Using UNION.

SELECT CustomerName FROM Customers WHERE FoodID = 1

UNION

SELECT CustomerName FROM Customers WHERE FoodID = 2;

* 1. Querying Data by Using Subqueries using INTERSECT.

SELECT CustomerName FROM Customers WHERE FlightID IN (SELECT FlightID FROM Flight WHERE FlightName = 'AirIndia')

AND CustomerID IN (

SELECT CustomerID FROM Customers WHERE FlightID IN (SELECT FlightID FROM Flight WHERE FlightName = 'IndiGo')

);

* 1. Querying Data by Using Subqueries using EXCEPT.

SELECT CustomerName FROM Customers WHERE FlightID IN (SELECT FlightID FROM Flight WHERE FlightName = 'AirIndia')

AND CustomerID NOT IN (

SELECT CustomerID FROM Customers WHERE FlightID IN (SELECT FlightID FROM Flight WHERE FlightName = 'JetAirways')

);

* 1. Querying Data by Using Subqueries using MERGE.

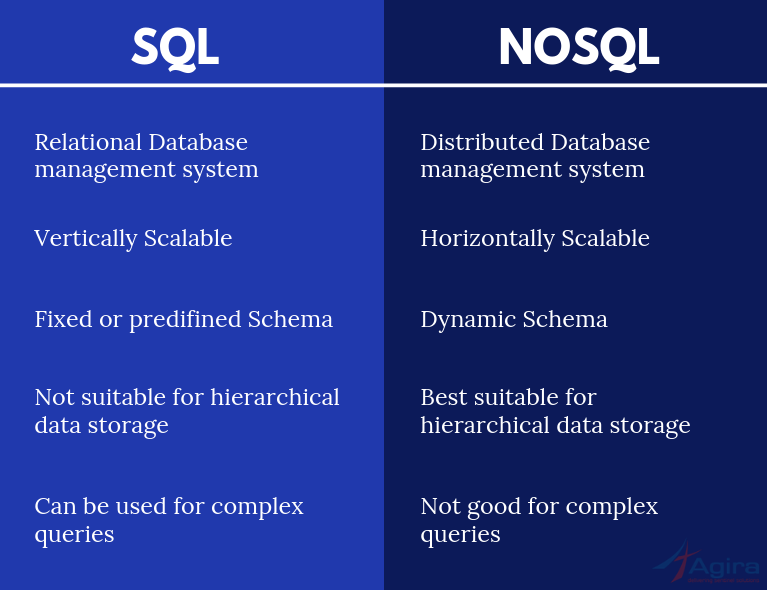
INSERT INTO Customers (CustomerID, CustomerName, FlightID, FoodID)

VALUES (308, 'Hari', 3, 1)

ON DUPLICATE KEY UPDATE FlightID = 3, FoodID = 1;

**ASSESSMENT 1:**

* + 1. Differentiate between SQL and NoSQL. Provide two advantages and two disadvantages of each with real-world examples.



|  |  |
| --- | --- |
| ADVANTAGES & DISADVANTAGES | |
| SQL | NoSQL |
| SQL allows for multiple data view | SQL has a difficult interface |
| Easy to learn and understand | Some versions of SQL are costly |
| SQL offers a uniform platform | Partial control is given to the database |
| Example: Used in banking systems. | Example: Used in social media, IoT platforms. |

* + 1. Given the below unnormalized data, convert it to 1NF, 2NF, and 3NF:

Student (StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)

**1NF**:  
Break repeating groups —> separate each course into a new row.

**2NF**:  
Remove partial dependencies —> create separate **Student** and **Course** tables.

**3NF**:  
Remove transitive dependencies —> create separate **Instructor** table.

|  |  |
| --- | --- |
| StudentID | Name |
| 1 | Aarthi |
| 2 | Ajay |

|  |  |
| --- | --- |
| CourseID | CourseName |
| 212 | Electronics |
| 106 | Computer science |

|  |  |
| --- | --- |
| InstructorName | InstructorPhone |
| Prakash | 84823XXXXX |
| Venkat | 98436XXXXX |

* + 1. a) Create a database named StudentDB:

CREATE DATABASE StudentDB;

b) Create a table Students with fields: StudentID, Name, DOB, Email.

Students(StudentID int auto\_increment primary key, Name varchar(60), DOB date, Email varchar(60))

c) Rename the table to Student\_Info:

ALTER TABLE Students RENAME TO Student\_Info;

d) Add a column PhoneNumber:

ALTER TABLE Student\_Info ADD PhoneNumber VARCHAR(15);

e) Drop the table.

DROP TABLE Student\_Info;

* + 1. a) Insert 3 student records into Student\_Info:

INSERT INTO Student\_Info VALUES (1, 'Ram', '1999-05-12', 'ram12@gmail.com', '9876543210');

INSERT INTO Student\_Info VALUES (2, 'ravi', '2004-02-14', 'raviraj@gmail.com', '9884596420'), (3, 'Priya', '2003-11-13', 'priyavetri@ yahoo.com', '8754832991')

b) Update one student's phone number:

UPDATE Student\_Info SET PhoneNumber = '8482991238' WHERE StudentID = 2;

c) Delete one student whose email ends with @gmail.com:

DELETE FROM Student\_Info WHERE Email LIKE '%raj@gmail.com';

d) Retrieve only names and emails of students born after the year 2000:

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

e) Retrieve distinct domain names from the email column:

SELECT DISTINCT SUBSTRING\_INDEX(Email, '@', -1) AS Domain FROM Student\_Info;

* + 1. a) Retrieve students with names starting with 'A':

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

b) Retrieve students with phone number between 9000000000 and 9999999999:

SELECT \* FROM Student\_Info WHERE PhoneNumber BETWEEN '9000000000' AND '9999999999';

c) Retrieve students using IN operator on city names:

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

d) Use AND, OR to filter students based on age and email provider.

SELECT \* FROM Student\_Info WHERE Age > 20 AND Email LIKE '%@yahoo.com';

e) Use table and column aliasing in a query to get all student names and DOBs:

SELECT Name AS StudentName, DOB AS BirthDate FROM Student\_Info;

* + 1. Create a new table Marks(StudentID, Subject, Marks). Insert at least 3 rows:

CREATE TABLE Marks (StudentID INT, Subject VARCHAR(50), Marks INT);

INSERT INTO Marks VALUES (1, 'Maths', 85), (2, 'Science', 78), (3, 'English', 90);

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) FROM Marks GROUP BY Subject;

1. Filter subjects with average marks between 60 and 90.

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

* + 1. a) Get the current date and format it as "YYYY-MM-DD":

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

b) Extract month and year from a DOB column:

SELECT MONTH(DOB), YEAR(DOB) FROM Student\_Info;

1. Convert a student's name to uppercase:

SELECT UPPER(Name) FROM Student\_Info;

1. Round off marks to 2 decimal places:

SELECT ROUND(Marks, 2) FROM Marks;

1. Use system function to return user name or current database:

SELECT USER(), DATABASE();

* + 1. a) Display total marks of each student:

SELECT StudentID, SUM(Marks) FROM Marks GROUP BY StudentID;

b) Display subject-wise highest mark:

SELECT Subject, MAX(Marks) FROM Marks GROUP BY Subject;

c) Use GROUP BY and HAVING to display subjects with average marks > 75:

SELECT Subject FROM Marks GROUP BY Subject HAVING AVG(Marks) > 75;

* + 1. a) Inner Join to retrieve students and their courses:

SELECT s.Name, c.CourseName FROM Student\_Info s INNER JOIN Enrollment e ON s.StudentID = e.StudentID INNER JOIN Course c ON e.CourseID = c.CourseID;

b) Left Join to get all students even if not enrolled.

SELECT s.Name, c.CourseName FROM Student\_Info s LEFT JOIN Enrollment e ON s.StudentID = e.StudentID LEFT JOIN Course c ON e.CourseID = c.CourseID;

c) Right Join to get all courses even if no students.

SELECT s.Name, c.CourseName FROM Student\_Info s RIGHT JOIN Enrollment e ON s.StudentID = e.StudentID RIGHT JOIN Course c ON e.CourseID = c.CourseID;

d) Full Outer Join equivalent using UNION.

SELECT s.Name, c.CourseName FROM Student\_Info s LEFT JOIN Enrollment e ON s.StudentID = e.StudentID LEFT JOIN Course c ON e.CourseID = c.CourseID

UNION

SELECT s.Name, c.CourseName FROM Student\_Info s RIGHT JOIN Enrollment e ON s.StudentID = e.StudentID RIGHT JOIN Course c ON e.CourseID = c.CourseID;

e) Cross Join to show all combinations.

SELECT s.Name, c.CourseName FROM Student\_Info s CROSS JOIN Course c;

* + 1. a) Students who scored more than average in 'Maths'.

SELECT StudentID FROM Marks WHERE Subject = 'Maths' AND Marks > (SELECT AVG(Marks) FROM Marks WHERE Subject = 'Maths');

b) Students not in the Marks table.

SELECT StudentID FROM Student\_Info WHERE StudentID NOT IN (SELECT DISTINCT StudentID FROM Marks);

c) Use EXISTS to get students with at least one subject.

SELECT \* FROM Student\_Info s WHERE EXISTS (SELECT 1 FROM Marks m WHERE s.StudentID = m.StudentID);

d) Use ALL to find those scoring more than all in 'Science'.

SELECT StudentID FROM Marks WHERE Marks > ALL (SELECT Marks FROM Marks WHERE Subject = 'Science');

e) Use ANY for students scoring better than some in 'English'.

SELECT StudentID FROM Marks WHERE Marks > ANY (SELECT Marks FROM Marks WHERE Subject = 'English');

* 1. a) UNION of student names from two tables.

SELECT Name FROM Student\_Info

UNION

SELECT Name FROM Alumni\_Info;

b) INTERSECT to find common students.

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

c) EXCEPT to list students in Students but not in Marks.

SELECT Name FROM Student\_Info WHERE Name NOT IN (SELECT Name FROM Marks);

d) MERGE concept or simulate with UPDATE and INSERT.

UPDATE Marks SET Marks = 85 WHERE StudentID = 1 AND Subject = 'Science';

INSERT INTO Marks(StudentID, Subject, Marks)

SELECT 1, 'Science', 85 FROM DUAL

WHERE NOT EXISTS (SELECT 1 FROM Marks WHERE StudentID = 1 AND Subject = 'Science');

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

SELECT m1.StudentID, m1.Subject, m1.Marks

FROM Marks m1

WHERE Marks > (SELECT AVG(Marks) FROM Marks m2 WHERE m2.Subject = m1.Subject);

**ASSESSMENT 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.**
* For big data, high scalability, real-time analytics, or flexible schema designs.
* Example: Social media feeds, IoT data, and personalized recommendations.

**Types of NoSQL**:

* **Document Store** (e.g., MongoDB) – Ideal for content management systems.
* **Key-Value Store** (e.g., Redis) – Useful for session caching.
* **Column Store** (e.g., Cassandra) – Best for IoT and analytics.
* **Graph Database** (e.g., Neo4j) – For relationship-heavy data like social networks.
  + 1. **A retail store keeps the following unnormalized record:**

**Customer (CustomerID, Name, Orders (OrderID, ProductID, Quantity, ProductName))**

**Normalize the data up to BCNF with appropriate table structures.**

**1NF** – Remove nested data:

* Customers (CustomerID, Name)
* Orders (OrderID, CustomerID, ProductID, Quantity, ProductName)

**2NF** – Remove partial dependencies:

* Products (ProductID, ProductName)
* OrderDetails (OrderID, ProductID, Quantity)

**3NF/BCNF** – Ensure no transitive dependencies:

* Customers(CustomerID, Name)
* Orders(OrderID, CustomerID)
* Products(ProductID, ProductName)
* OrderDetails(OrderID, ProductID, Quantity)
  + 1. a) 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,

Name VARCHAR(100)

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100),

Price DECIMAL(10,2)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

ProductID INT,

Quantity INT CHECK (Quantity > 0),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

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

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

c) Alter the Products table to add 'Discount' column and update some values.

UPDATE Products SET Discount = 10 WHERE ProductID = 1;

* + 1. Using the above schema: a) Insert 3 sample orders per customer.

Orders(OrderID, CustomerID, ProductID, Quantity, OrderDate)

Orders for Customer 1

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

(1, 1, 101, 2, '2024-06-01'),

(2, 1, 102, 1, '2024-06-02'),

(3, 1, 103, 3, '2024-06-03');

Orders for Customer 2

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

(4, 2, 101, 1, '2024-06-01'),

(5, 2, 103, 4, '2024-06-02'),

(6, 2, 102, 2, '2024-06-03');

Orders for Customer 3

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

(7, 3, 102, 5, '2024-06-01'),

(8, 3, 101, 1, '2024-06-02'),

(9, 3, 103, 2, '2024-06-03');

b) 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: a) Customers who ordered more than 3 different products.

SELECT CustomerID FROM Orders

GROUP BY CustomerID

HAVING COUNT(DISTINCT ProductID) > 3;

b) Products not ordered by any customer.

SELECT \* FROM Products

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

c) Count of orders placed by each customer in the last 30 days.

SELECT CustomerID, COUNT(\*) AS OrderCount

FROM Orders

WHERE OrderDate >= CURDATE() - INTERVAL 30 DAY

GROUP BY CustomerID;

* + 1. a) Use string functions to standardize and extract parts from customer email IDs.

SELECT Email, SUBSTRING\_INDEX(Email, '@', 1) AS Username FROM Customers;

b) Use date functions to compute days between order date and today.

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

c) Use system functions to return current user and host.

SELECT USER(), CURRENT\_USER();

d) Use nested functions to format a customer greeting string.

SELECT CONCAT('Hello ', UPPER(Name), '!') AS Greeting FROM Customers;

* + 1. a) Aggregate total revenue by product category.

SELECT Category, SUM(Price \* Quantity) AS Revenue

FROM Products p JOIN Orders o ON p.ProductID = o.ProductID

GROUP BY Category;

b) Use GROUP BY with ROLLUP to compute subtotal and grand total sales.

SELECT Category, SUM(Price \* Quantity) AS Total

FROM Products p JOIN Orders o ON p.ProductID = o.ProductID

GROUP BY Category WITH ROLLUP;

c) Use HAVING clause to filter categories with revenue > 100000.

SELECT Category, SUM(Price \* Quantity) AS Revenue

FROM Products p JOIN Orders o ON p.ProductID = o.ProductID

GROUP BY Category

HAVING Revenue > 100000;

* + 1. a) Self join to list customers referred by other customers.

SELECT A.Name AS Customer, B.Name AS ReferredBy

FROM Customers A

JOIN Customers B ON A.ReferredBy = B.CustomerID;

b) Equi join across Orders and Products.

SELECT o.OrderID, p.ProductName

FROM Orders o JOIN Products p ON o.ProductID = p.ProductID;

c) Join Customers and Orders to display top 3 spenders using window function.

SELECT CustomerID, SUM(Price \* Quantity) AS TotalSpent,

RANK() OVER (ORDER BY SUM(Price \* Quantity) DESC) AS Rank

FROM Orders o JOIN Products p ON o.ProductID = p.ProductID

GROUP BY CustomerID

LIMIT 3;

d) LEFT OUTER JOIN with WHERE NULL to identify inactive customers.

SELECT \* FROM Customers

LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID

WHERE Orders.OrderID IS NULL;

e) Cross join for all product combinations in a bundle offer.

SELECT A.ProductName, B.ProductName

FROM Products A CROSS JOIN Products B;

* + 1. a) Correlated subquery to get customers whose order amount exceeds their average.

SELECT \* FROM Orders o1

WHERE Quantity > (

SELECT AVG(Quantity)

FROM Orders o2

WHERE o1.CustomerID = o2.CustomerID

);

b) Subquery using EXISTS to find customers with at least 2 different products.

SELECT CustomerID FROM Orders

GROUP BY CustomerID

HAVING COUNT(DISTINCT ProductID) >= 2;

c) Use ALL to find customers who ordered more than every other customer.

SELECT CustomerID FROM Orders

GROUP BY CustomerID

HAVING COUNT(\*) > ALL (

SELECT COUNT(\*) FROM Orders GROUP BY CustomerID

);

d) Use ANY to find products costlier than some in category 'Electronics'.

SELECT \* FROM Products

WHERE Price > ANY (

SELECT Price FROM Products WHERE Category = 'Electronics'

);

e) Nested subquery to list top 3 best-selling products.

SELECT ProductID, SUM(Quantity) AS TotalSold

FROM Orders

GROUP BY ProductID

ORDER BY TotalSold DESC

LIMIT 3;

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

SELECT c1.CustomerID FROM SegmentA c1

JOIN SegmentB c2 ON c1.CustomerID = c2.CustomerID;

b) Use EXCEPT to find products in inventory not yet ordered.

SELECT ProductID FROM Inventory

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

c) Simulate MERGE: If customer exists, update; else insert.

INSERT INTO Customers (CustomerID, Name)

VALUES (101, 'John Doe')

ON DUPLICATE KEY UPDATE Name = 'John Doe';

d) Use UNION to combine two regional customer tables.

SELECT \* FROM Region1\_Customers

UNION

SELECT \* FROM Region2\_Customers;

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

WITH RankedCustomers AS (

SELECT CustomerID, SUM(Price \* Quantity) AS TotalSpend,

RANK() OVER (ORDER BY SUM(Price \* Quantity) DESC) AS SpendRank

FROM Orders o JOIN Products p ON o.ProductID = p.ProductID

GROUP BY CustomerID

)

SELECT \* FROM RankedCustomers WHERE SpendRank <= 5;

**CASE STUDY – CRIME REPORTING SYSTEM**

CREATE DATABASE IF NOT EXISTS CrimeDB;

USE CrimeDB;

CREATE TABLE IF NOT EXISTS Victims (

VictimID INT PRIMARY KEY,

FirstName VARCHAR(50) NOT NULL,

LastName VARCHAR(50) NOT NULL,

DateOfBirth DATE,

Gender VARCHAR(10),

ContactInfo VARCHAR(100)

);

CREATE TABLE IF NOT EXISTS Suspects (

SuspectID INT PRIMARY KEY,

FirstName VARCHAR(50) NOT NULL,

LastName VARCHAR(50) NOT NULL,

DateOfBirth DATE,

Gender VARCHAR(10),

ContactInfo VARCHAR(100)

);

CREATE TABLE IF NOT EXISTS Agencies (

AgencyID INT PRIMARY KEY,

AgencyName VARCHAR(100) NOT NULL,

Jurisdiction VARCHAR(100),

ContactInfo VARCHAR(100)

);

CREATE TABLE IF NOT EXISTS Officers (

OfficerID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

BadgeNumber VARCHAR(20),

Rank VARCHAR(50),

ContactInfo VARCHAR(100),

AgencyID INT,

FOREIGN KEY (AgencyID) REFERENCES Agencies(AgencyID)

);

CREATE TABLE IF NOT EXISTS Incidents (

IncidentID INT PRIMARY KEY,

IncidentType VARCHAR(50) NOT NULL,

IncidentDate DATE NOT NULL,

Latitude DECIMAL(10, 6),

Longitude DECIMAL(10, 6),

Location VARCHAR(100),

Description TEXT,

Status VARCHAR(30),

VictimID INT,

SuspectID INT,

AgencyID INT,

FOREIGN KEY (VictimID) REFERENCES Victims(VictimID),

FOREIGN KEY (SuspectID) REFERENCES Suspects(SuspectID),

FOREIGN KEY (AgencyID) REFERENCES Agencies(AgencyID)

);

CREATE TABLE IF NOT EXISTS Evidence (

EvidenceID INT PRIMARY KEY,

Description TEXT,

LocationFound VARCHAR(100),

IncidentID INT,

FOREIGN KEY (IncidentID) REFERENCES Incidents(IncidentID)

);

CREATE TABLE IF NOT EXISTS Reports (

ReportID INT PRIMARY KEY,

IncidentID INT,

ReportingOfficer INT,

ReportDate DATE,

ReportDetails TEXT,

Status VARCHAR(30),

FOREIGN KEY (IncidentID) REFERENCES Incidents(IncidentID),

FOREIGN KEY (ReportingOfficer) REFERENCES Officers(OfficerID)

);

INSERT INTO Victims VALUES

(1, 'Rahul', 'Mehta', '1990-04-15', 'Male', '123 Street, Delhi, 9876543210'),

(2, 'Neha', 'Sharma', '1988-08-25', 'Female', '456 Avenue, Mumbai, 9123456780');

INSERT INTO Suspects VALUES

(1, 'Rohan', 'Kumar', '1985-02-10', 'Male', '789 Lane, Delhi, 8899776655'),

(2, 'Simran', 'Kaur', '1992-11-05', 'Female', '321 Boulevard, Mumbai, 9988776655');

INSERT INTO Agencies VALUES

(1, 'Delhi Police', 'Delhi Region', 'contact@delhipolice.in'),

(2, 'Mumbai Police', 'Mumbai Region', 'help@mumbaipolice.in');

INSERT INTO Officers VALUES

(1, 'Ajay', 'Verma', 'D101', 'Inspector', 'ajay@delhipolice.in', 1),

(2, 'Priya', 'Joshi', 'M202', 'Sub-Inspector', 'priya@mumbaipolice.in', 2);

INSERT INTO Incidents VALUES

(101, 'Robbery', '2024-06-01', 28.613939, 77.209023, 'Connaught Place, Delhi', 'Stolen bag near metro station', 'Open', 1, 1, 1),

(102, 'Theft', '2024-06-03', 19.076090, 72.877426, 'Churchgate, Mumbai', 'Wallet stolen during rush hour', 'Under Investigation', 2, 2, 2);

INSERT INTO Evidence VALUES

(1, 'Security camera footage', 'Connaught Place CCTV Booth', 101),

(2, 'Recovered wallet', 'Churchgate platform 3', 102);

INSERT INTO Reports VALUES

(1, 101, 1, '2024-06-02', 'Suspect identified via CCTV footage', 'Draft'),

(2, 102, 2, '2024-06-04', 'Witness interview conducted and report submitted', 'Finalized');