**DAY 1 – 09.06.2025**

**ASSIGNMENT 1**1) Explain all the algorithm basics in brief and compare   
  
2) Compare all sorting algorithm and choose any two best according to you and why  
  
3) compare searching algorithm  
  
4) why we use BST and what is the need of AVL and difference between BST and AVL tree

**1) Algorithm Approaches – Basics with Real-Life Examples**

a) Brute Force and Heuristic

Brute Force

* Tries all possible solutions
* Example: Opening a combination lock by checking every code from 000 to 999
* Best used when: Problem size is small
* Drawback: Very inefficient for large datasets

Heuristic

* Makes educated guesses instead of checking all options
* Example: Searching a book in the “Science” section without a catalog
* Best used when: Exact solution is not feasible, and approximation is acceptable
* Drawback: May not guarantee correct answer

b) Greedy Approach

* Makes the best choice at each step
* Example: Making ₹43 using the largest available coins first
* Best used when: Local optimal choices lead to global solution
* Drawback: Fails when local best does not lead to overall best

c) Divide and Conquer

* Divide: Split problem into subproblems
* Conquer: Solve each recursively
* Combine: Merge the results
* Example: Sorting papers by splitting, sorting each part, and combining
* Used in: Merge Sort, Quick Sort, Binary Search
* Efficient for large datasets

d) Dynamic Programming

* Stores results of subproblems (memoization/tabulation)
* Example: Counting ways to climb stairs using stored results of previous steps
* Best used when: Problem has overlapping subproblems and optimal substructure
* Efficient but may be harder to implement initially

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 |

**2) Sorting Algorithms – Comparison and Examples**

Real-Life Examples

* Bubble Sort: Swapping books by height repeatedly until sorted
* Insertion Sort: Arranging playing cards one by one
* Selection Sort: Finding the smallest item and placing it at the beginning
* Merge Sort: Two people sort halves of a deck and merge
* Quick Sort: Picking a pivot height, dividing into smaller and greater

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 |

Best Two Sorting Algorithms

1. Merge Sort – Best for large datasets when stability is needed
2. Quick Sort – Fastest on average, memory efficient

**3) Searching Algorithms – Comparison and Examples**

Real-Life Examples

* Linear Search: Finding a person in a crowd
* Binary Search: Searching a word in a dictionary

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 |

**4) Binary Search Tree (BST) vs AVL Tree**

Real-Life Examples

* BST: Contact list arranged alphabetically
* AVL Tree: Music library with balanced indexing

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 2 – 10.06.2025**

**ASSIGNMENT – 2**

**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**
2. **Which system database stores all login accounts and configuration settings?**  
   a) tempdb  
   b) model  
   **c) master**d) msdb
3. **What is the purpose of the model database in SQL Server?**  
   a) Backup  
   b) Log storage  
   **c) Template for new databases**  
   d) System configuration
4. **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
5. **Which SQL command is used to create a new database?**  
   a) MAKE DATABASE  
   b) NEW DATABASE  
   **c) CREATE DATABASE**  
   d) INIT DATABASE
6. **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
7. **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

**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
2. **What command is used to create a table?**  
   a) MAKE TABLE  
   b) INSERT TABLE  
   **c) CREATE TABLE**d) DEFINE TABLE
3. **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
4. **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
5. **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

**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
2. **Which clause is used to update data in a table?**  
   a) MODIFY  
   **b) UPDATE**  
   c) CHANGE  
   d) SET TABLE
3. **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
4. **Which clause is used to filter rows in a SELECT statement?**  
   a) HAVING  
   b) SELECT  
   **c) WHERE**  
   d) ORDER BY
5. **Which keyword ensures no duplicate records are returned?**  
   a) UNIQUE  
   b) NO\_REPEAT  
   **c) DISTINCT**d) ONLY
6. **What does the LIKE keyword do in SQL?**  
   a) Finds exact matches  
   **b) Finds pattern-based matches**c) Sorts records  
   d) Deletes matches
7. **Which operator is used to combine multiple conditions in a WHERE clause?**  
   a) TO  
   b) WITH  
   **c) AND / OR**  
   d) IF / ELSE
8. **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

**DAY 3 – 11.06.2025**

**ASSIGNMENT – 3**

**SECTION A**

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

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

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

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

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

CREATE DATABASE TeamDB;

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

ALTER DATABASE TeamDB MODIFY NAME = ProjectDB;

1. **Drop the ProjectDB database.**

DROP DATABASE ProjectDB

**SECTION B – Managing Tables**

DROP DATABASE IF EXISTS AssignmentDB;

CREATE DATABASE AssignmentDB;

USE AssignmentDB;

**1. Create Employees table**

**2. Add column Salary**

CREATE TABLE Employees (

EmpID INT PRIMARY KEY,

Name VARCHAR(50),

Department VARCHAR(30),

JoiningDate DATE,

IsActive BOOLEAN,

Salary DECIMAL(10,2)

);

**3. Rename table Employees to TeamMembers**

RENAME TABLE Employees TO TeamMembers;

**4. Drop the TeamMembers table**

DROP TABLE TeamMembers;

**SECTION C – DML operations**

CREATE TABLE Employees (

EmpID INT PRIMARY KEY,

Name VARCHAR(50),

Department VARCHAR(30),

JoiningDate DATE,

IsActive BOOLEAN,

Salary DECIMAL(10,2)

);

**1. Insert three rows**

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;

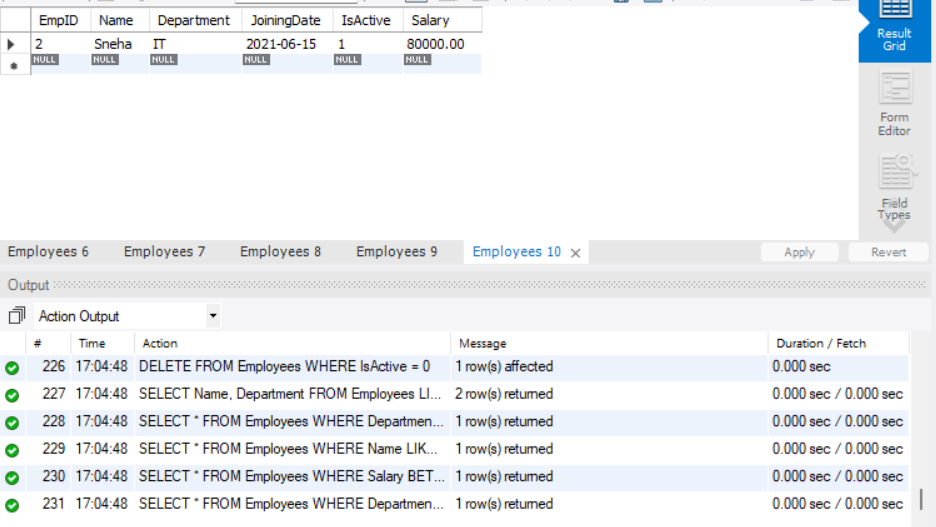
**6. Filtering using LIKE, BETWEEN, IN**

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

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

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

**OUTPUT**

****

**DAY 4 – 12.06.2025**

**ASSIGNMENT – 4**

DROP DATABASE IF EXISTS AssignmentDay4;

CREATE DATABASE AssignmentDay4;

USE AssignmentDay4;

**Q1: Students Table, Insert and Update**

CREATE TABLE students (

student\_id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

marks INT NOT NULL,

email VARCHAR(100) NOT NULL UNIQUE

);

INSERT INTO students (name, marks, email) VALUES

('Anu', 78, 'anu@example.com'),

('Ravi', 85, 'ravi@example.com'),

('Sneha', 92, 'sneha@example.com'),

('Arjun', 67, 'arjun@example.com'),

('Divya', 88, 'divya@example.com');

UPDATE students SET marks = 90 WHERE name = 'Arjun';

**Q2: Customers Table and String Functions**

CREATE TABLE customers (

customer\_id INT AUTO\_INCREMENT PRIMARY KEY,

full\_name VARCHAR(100)

);

INSERT INTO customers (full\_name) VALUES

('Anu Sharma'),

('Ravi Kumar'),

('Sneha Rao');

SELECT

full\_name,

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

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

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

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

FROM customers;

**Q3: Sales Table and Date Functions**

CREATE TABLE sales (

sale\_id INT AUTO\_INCREMENT PRIMARY KEY,

sale\_amount DECIMAL(8,2),

sale\_date DATE,

product\_category VARCHAR(50)

);

INSERT INTO sales (sale\_amount, sale\_date, product\_category) VALUES

(500.00, '2025-06-10', 'Cakes'),

(200.00, '2025-06-08', 'Pastries'),

(300.00, '2025-06-01', 'Beverages');

SELECT

sale\_date,

MONTHNAME(sale\_date) AS month\_name,

YEAR(sale\_date) AS year,

DATEDIFF(CURDATE(), sale\_date) AS days\_ago

FROM sales;

**Q4: Employees Table and Math Functions**

CREATE TABLE employees (

emp\_id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100),

salary DECIMAL(10,2)

);

INSERT INTO employees (name, salary) VALUES

('Priya', 24500),

('Rahul', 31500),

('Meena', 41200);

SELECT

name,

salary,

salary \* 1.10 AS salary\_after\_hike,

ROUND(salary, -2) AS rounded\_salary

FROM employees;

**Q5: System Functions**

SELECT

NOW() AS Current\_datetime,

DATABASE() AS current\_database,

CURRENT\_USER() AS logged\_in\_user;

**Q6: Products Table and Custom Result Set**

CREATE TABLE products (

product\_id INT AUTO\_INCREMENT PRIMARY KEY,

product\_name VARCHAR(100),

price DECIMAL(8,2)

);

INSERT INTO products (product\_name, price) VALUES

('Chocolate Cake', 350),

('Vanilla Pastry', NULL),

('Cold Coffee', 150);

SELECT

UPPER(product\_name) AS product\_name\_upper,

IFNULL(price, 'Not Available') AS price\_display

FROM products;

**Q7: Transactions Table and Aggregates**

CREATE TABLE transactions (

transaction\_id INT AUTO\_INCREMENT PRIMARY KEY,

sale\_amount DECIMAL(10,2)

);

INSERT INTO transactions (sale\_amount) VALUES

(500), (650), (200), (850);

SELECT

SUM(sale\_amount) AS total\_sales,

AVG(sale\_amount) AS average\_sale,

MAX(sale\_amount) AS max\_sale,

MIN(sale\_amount) AS min\_sale

FROM transactions;

**Q8: Grouping in Sales Table**

SELECT

product\_category,

SUM(sale\_amount) AS total\_sales,

COUNT(\*) AS transactions

FROM sales

GROUP BY product\_category;

**Q9: Inner Join Orders and Customers**

CREATE TABLE orders (

order\_id INT AUTO\_INCREMENT PRIMARY KEY,

customer\_id INT,

product\_id INT,

order\_amount DECIMAL(8,2),

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

);

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

(1, 1, 350),

(2, 2, 150);

SELECT

c.full\_name AS customer\_name,

o.order\_amount

FROM orders o

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

**Q10: Left Join with Products**

SELECT

p.product\_name,

o.order\_id,

o.order\_amount

FROM products p

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

**Q11: Right Join Customers and Contacts**

CREATE TABLE contacts (

contact\_id INT AUTO\_INCREMENT PRIMARY KEY,

customer\_id INT,

phone VARCHAR(15),

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

);

INSERT INTO contacts (customer\_id, phone) VALUES

(1, '9876543210'),

(2, '9123456789');

SELECT

c.customer\_id,

c.full\_name,

con.phone

FROM contacts con

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

**Q12: Full Outer Join with Suppliers (via UNION)**

CREATE TABLE suppliers (

supplier\_id INT AUTO\_INCREMENT PRIMARY KEY,

supplier\_name VARCHAR(100)

);

ALTER TABLE products ADD supplier\_id INT;

UPDATE products SET supplier\_id = 1 WHERE product\_id = 1;

UPDATE products SET supplier\_id = 2 WHERE product\_id = 2;

INSERT INTO suppliers (supplier\_name) VALUES

('Bakery Supplies Inc.'),

('Fresh Goods Ltd.'),

('No Match Supplier');

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;

**Q13: Cross Join with Offers**

CREATE TABLE offers (

offer\_id INT AUTO\_INCREMENT PRIMARY KEY,

offer\_name VARCHAR(100)

);

INSERT INTO offers (offer\_name) VALUES

('Buy 1 Get 1 Free'),

('20% Off'),

('Free Delivery');

SELECT

p.product\_name,

o.offer\_name

FROM products p

CROSS JOIN offers o;

**Q14: Join with Aggregation (Orders + Products) - FIXED**

ALTER TABLE products ADD category VARCHAR(100);

UPDATE products

SET category = 'Cakes' WHERE product\_name = 'Chocolate Cake';

UPDATE products

SET category = 'Pastries' WHERE product\_name = 'Vanilla Pastry';

UPDATE products

SET category = 'Beverages' WHERE product\_name = 'Cold Coffee';

SELECT

p.category AS product\_category,

SUM(o.order\_amount) AS total\_sold,

AVG(p.price) AS avg\_price

FROM orders o

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

GROUP BY p.category;

**Q15: Join Students + Marks with Filter**

CREATE TABLE marks (

mark\_id INT AUTO\_INCREMENT PRIMARY KEY,

student\_id INT,

mark INT,

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

);

INSERT INTO marks (student\_id, mark) VALUES

(1, 85), (1, 75),

(2, 90),

(3, 78),

(4, 82),

(5, 91);

SELECT

s.name AS student\_name,

AVG(m.mark) AS average\_marks

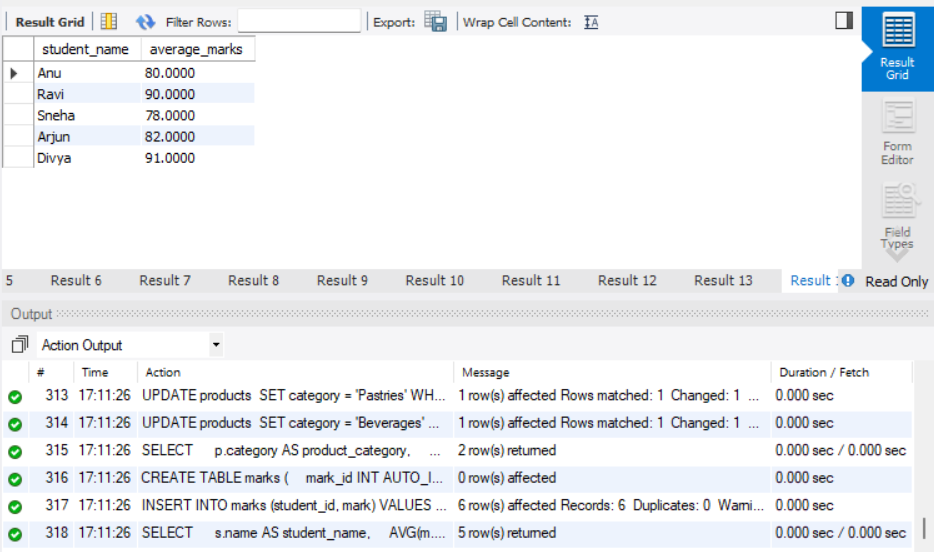
FROM students s

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

GROUP BY s.name

HAVING AVG(m.mark) > 75;

**OUTPUT**



**DAY 5 – 13.06.2025**

**ASSIGNMENT 5 – QUESTION PAPER 1**

**Section A: Basics & Data Definition (10 Marks)**

**Q1. Difference Between SQL and NoSQL (3 marks)**

| **Feature** | **SQL (Relational)** | **NoSQL (Non-Relational)** |
| --- | --- | --- |
| Data Format | Tables (rows & columns) | Documents, key-value pairs, graphs, wide-columns |
| Schema | Fixed schema | Dynamic/Schema-less |
| ACID Compliance | Strong ACID compliance | BASE properties (eventual consistency) |
| Example | MySQL, PostgreSQL | MongoDB, Cassandra |

**Advantages:**

* **SQL:**
  1. Structured & reliable transactions
  2. Excellent for complex queries (JOINs, subqueries)
* **NoSQL:**
  1. Flexible schema for unstructured data
  2. Better horizontal scalability

**Disadvantages:**

* **SQL:**
  1. Schema rigidity
  2. Scaling vertically is costly
* **NoSQL:**
  1. Less reliable for multi-row transactions
  2. Limited support for complex queries

**Q2. Normalization (2 marks)**

**Unnormalized:**

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

**1NF (Remove multivalued fields):**

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

**2NF (Remove partial dependencies):**  
Split into:

* Student(StudentID, Name)
* Course(CourseID, CourseName, InstructorName, InstructorPhone)
* Enrollment(StudentID, CourseID)

**3NF (Remove transitive dependencies):**  
Split:

* Instructor(InstructorName, InstructorPhone)
* Course(CourseID, CourseName, InstructorName)

Final:

* Student(StudentID, Name)
* Instructor(InstructorName, InstructorPhone)
* Course(CourseID, CourseName, InstructorName)
* Enrollment(StudentID, CourseID)

**Q3 a) Create Database**

DROP DATABASE IF EXISTS StudentDB;

CREATE DATABASE StudentDB;

USE StudentDB;

**Q3 b) Create Table**

CREATE TABLE Students (

StudentID INT PRIMARY KEY,

Name VARCHAR(100),

DOB DATE,

Email VARCHAR(100)

);

**Q3 c) Rename Table**

RENAME TABLE Students TO Student\_Info;

**Q3 d) Add Column**

ALTER TABLE Student\_Info ADD PhoneNumber BIGINT;

**SECTION B: DML & Filtering Data**

**Q4 a) Insert 3 Records**

INSERT INTO Student\_Info VALUES

(1, 'Arjun', '2001-06-21', 'arjun@gmail.com', 9876543210),

(2, 'Meera', '1998-12-01', 'meera@yahoo.com', 9123456789),

(3, 'Ravi', '2003-04-15', 'ravi@outlook.com', 9988776655);

**Q4 b) Update one student’s phone number**

UPDATE Student\_Info SET PhoneNumber = 9000012345 WHERE StudentID = 2;

**Q4 c) Delete one student whose email ends with @gmail.com**

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

**Q4 d) Retrieve names and emails of students born after 2000**

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

**Q4 e) Retrieve distinct domain names from email**

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

**Q5: Filtering Data**

**Q5 a) Retrieve students with names starting with 'A'**

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

**Q5 b) Retrieve students with phone number between 9000000000 and 9999999999**

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

**Q5 c) Use IN operator on city names**

ALTER TABLE Student\_Info ADD City VARCHAR(50);

UPDATE Student\_Info SET City = 'Chennai' WHERE StudentID = 2;

UPDATE Student\_Info SET City = 'Bangalore' WHERE StudentID = 3;

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

**Q5 d) Use AND, OR to filter students based on age and email provider**

SELECT \* FROM Student\_Info

WHERE (YEAR(CURDATE()) - YEAR(DOB)) > 22 AND Email LIKE '%@yahoo.com';

**Q5 e) Use table and column aliasing**

SELECT s.Name AS Student\_Name, s.DOB AS BirthDate FROM Student\_Info s;

**Q6: Create Marks Table and Queries**

CREATE TABLE Marks (

StudentID INT,

Subject VARCHAR(50),

Marks FLOAT

);

INSERT INTO Marks VALUES

(2, 'Maths', 85.50),

(3, 'Science', 78.75),

(3, 'English', 64.80);

**Q6 a) Display student IDs and their subjects where marks > 70**

SELECT StudentID, Subject FROM Marks WHERE Marks > 70;

**Q6 b) Display subjects with average marks**

SELECT Subject, AVG(Marks) AS AvgMarks FROM Marks GROUP BY Subject;

**Q6 c) Filter subjects with average marks between 60 and 90**

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

**SECTION C: Functions & Grouping**

**Q7: Built-in Functions**

**Q7 a) Get current date formatted**

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

**Q7 b) Extract month and year from DOB**

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

**Q7 c) Convert student name to uppercase**

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

**Q7 d) Round off marks to 2 decimal places**

SELECT ROUND(Marks, 2) AS RoundedMarks FROM Marks;

**Q7 e) Use system function to return username or database**

SELECT USER() AS LoggedInUser, DATABASE() AS CurrentDB;

**Q8: Grouping**

**Q8 a) Display total marks of each student**

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

**Q8 b) Display subject-wise highest mark**

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

**Q8 c) GROUP BY and HAVING to show subjects with avg > 75**

SELECT Subject, AVG(Marks) AS AvgMarks

FROM Marks

GROUP BY Subject

HAVING AVG(Marks) > 75;

**SECTION D: Joins & Subqueries**

**Q9: Joins**

CREATE TABLE Course (

CourseID INT PRIMARY KEY,

CourseName VARCHAR(100)

);

CREATE TABLE Enrollment (

StudentID INT,

CourseID INT

);

INSERT INTO Course VALUES (1, 'Maths'), (2, 'Science');

INSERT INTO Enrollment VALUES (2, 1), (3, 2);

**Q9 a) Inner Join: Students and their courses**

SELECT s.Name, c.CourseName

FROM Student\_Info s

JOIN Enrollment e ON s.StudentID = e.StudentID

JOIN Course c ON e.CourseID = c.CourseID;

**Q9 b) Left Join: 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;

**Q9 c) Right Join: All courses, even if no students**

SELECT s.Name, c.CourseName

FROM Course c

RIGHT JOIN Enrollment e ON c.CourseID = e.CourseID

RIGHT JOIN Student\_Info s ON e.StudentID = s.StudentID;

**Q9 d) Full Outer Join 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 Course c

LEFT JOIN Enrollment e ON c.CourseID = e.CourseID

LEFT JOIN Student\_Info s ON e.StudentID = s.StudentID;

**Q9 e) Cross Join**

SELECT s.Name, c.CourseName

FROM Student\_Info s

CROSS JOIN Course c;

**Q10: Subqueries**

**Q10 a) Students who scored more than average in Maths**

SELECT \* FROM Marks

WHERE Subject = 'Maths'

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

**Q10 b) Students not in Marks table**

SELECT \* FROM Student\_Info

WHERE StudentID NOT IN (SELECT DISTINCT StudentID FROM Marks);

**Q10 c) Use EXISTS to get students with at least one subject**

SELECT \* FROM Student\_Info s

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

**Q10 d) Use ALL to find those scoring more than all in Science**

SELECT \* FROM Marks

WHERE Subject = 'Science'

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

**Q10 e) Use ANY for students scoring better than some in English**

SELECT \* FROM Marks WHERE Subject = 'English'

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

**Q11: Set Operators and Correlated Subquery**

**Q11 a) UNION of student names from two tables (simulate)**

SELECT Name FROM Student\_Info

UNION

SELECT Name FROM Student\_Info;

**Q11 b) INTERSECT simulation using INNER JOIN**

SELECT DISTINCT s1.Name

FROM Student\_Info s1

INNER JOIN Student\_Info s2 ON s1.Name = s2.Name;

**Q11 c) EXCEPT simulation using NOT IN**

SELECT Name FROM Student\_Info

WHERE StudentID NOT IN (SELECT StudentID FROM Marks);

**Q11 d) MERGE simulation: insert/update (UPSERT)**

INSERT INTO Marks (StudentID, Subject, Marks)

VALUES (2, 'English', 70)

ON DUPLICATE KEY UPDATE Marks = 70;

**Q11 e) Correlated subquery: above average per subject**

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

FROM Marks m1

WHERE m1.Marks > (

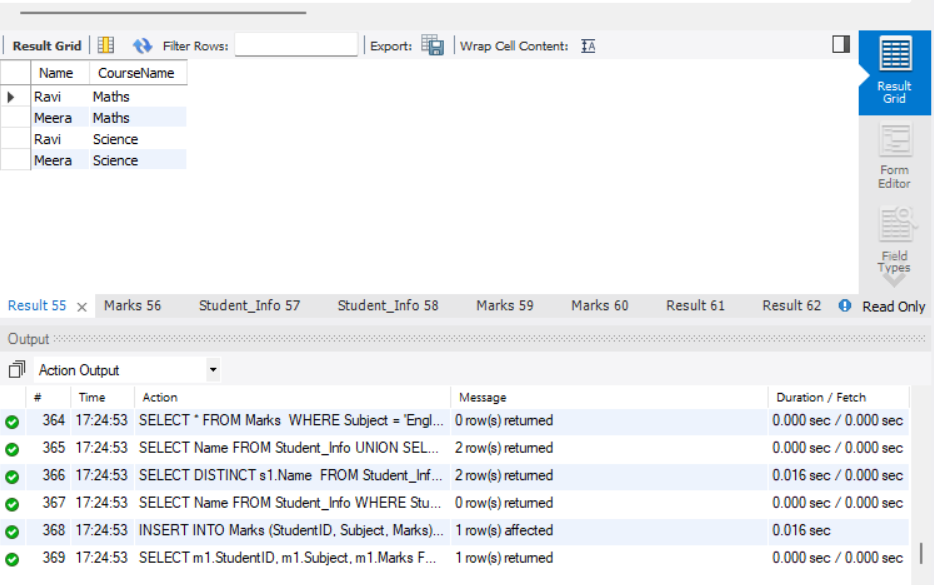
SELECT AVG(m2.Marks)

FROM Marks m2

WHERE m2.Subject = m1.Subject

);

**OUTPUT**

****

**ASSIGNMENT 5 – QUESTION PAPER 2**

**Section A: Advanced Concepts & Schema Design**

**Q1. NoSQL Use Cases and Types**

Scenarios where NoSQL is preferred:

* Flexible Schema: Useful for rapidly changing data models.
* Scalability: Ideal for large-scale distributed systems.
* Unstructured/Semi-structured data: e.g., JSON, XML.

| **Type of NoSQL** | **Description** | **Real-time Example** |
| --- | --- | --- |
| **Document Store** | **Stores JSON-like documents** | **MongoDB → Content Management System** |
| **Key-Value Store** | **Stores data as key-value pairs** | **Redis → Caching in e-commerce sites** |
| **Column Store** | **Columnar format, highly scalable** | **Cassandra → Time-series IoT data** |
| **Graph Store** | **Nodes & edges for relationships** | **Neo4j → Social media connections** |

**Q2. Normalization to BCNF**

**Unnormalized Table:**

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

**1NF (Flatten multivalued fields):**

CustomerOrders (CustomerID, Name, OrderID, ProductID, Quantity, ProductName)

**2NF** (Remove partial dependency):

Tables:

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

**3NF** (Remove transitive dependencies):

* No transitive dependency as ProductName depends on ProductID.

**BCNF** (Ensure candidate keys only have dependencies on superkeys):

All tables above already satisfy BCNF.

**Section B: Complex DDL and DML**

**Q3 a) Create database and schema**

DROP DATABASE IF EXISTS RetailDB;

CREATE DATABASE RetailDB;

USE RetailDB;

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

Name VARCHAR(100),

Email VARCHAR(100)

);

CREATE TABLE Products (

ProductID INT PRIMARY KEY,

ProductName VARCHAR(100),

Category VARCHAR(50),

Price DECIMAL(10, 2)

);

**Q3 b) Add check constraint on Quantity > 0**

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

ProductID INT,

Quantity INT CHECK (Quantity > 0),

OrderDate DATE,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

FOREIGN KEY (ProductID) REFERENCES Products(ProductID)

);

**Q3 c) Alter Products table to add Discount column and update values**

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

UPDATE Products SET Discount = 5 WHERE Category = 'Electronics';

**Q4 a) Insert 3 sample orders per customer**

INSERT INTO Customers VALUES

(1, 'Ankit', 'ankit@example.com'),

(2, 'Riya', 'riya@example.com');

INSERT INTO Products VALUES

(101, 'Phone', 'Electronics', 20000, 10),

(102, 'Book', 'Stationery', 300, 0),

(103, 'Bag', 'Accessories', 1500, 5);

INSERT INTO Orders VALUES

(1, 1, 101, 3, CURDATE()),

(2, 1, 102, 2, CURDATE()),

(3, 1, 103, 6, CURDATE()),

(4, 2, 101, 1, CURDATE()),

(5, 2, 103, 7, CURDATE()),

(6, 2, 102, 4, CURDATE());

**Q4 b) Update prices with 10% increase where quantity sold > 5**

UPDATE Products

SET Price = Price \* 1.10

WHERE ProductID IN (

SELECT ProductID FROM Orders WHERE Quantity > 5

);

**Q4 c) Delete orders where the product has never been sold**

DELETE FROM Orders

WHERE ProductID NOT IN (

SELECT ProductID FROM (

SELECT DISTINCT ProductID FROM Orders

) AS temp

);

**Q5 a) Customers who ordered more than 3 different products**

SELECT CustomerID

FROM Orders

GROUP BY CustomerID

HAVING COUNT(DISTINCT ProductID) > 3;

**Q5 b) Products not ordered by any customer**

SELECT \* FROM Products

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

**Q5 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;

**Section C: Advanced Functions and Aggregations**

**Q6 a) Use string functions to standardize and extract parts from email IDs**

SELECT LOWER(Email),

SUBSTRING\_INDEX(Email, '@', 1),

SUBSTRING\_INDEX(Email, '@', -1)

FROM Customers;

**Q6 b) Use date functions to compute days between order date and today**

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

**Q6 c) Use system functions to return current user and host**

SELECT USER(), SYSTEM\_USER();

**Q6 d) Use nested functions to format a customer greeting string**

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

**Q7 a) Aggregate total revenue by product category**

SELECT Category, SUM(Price \* o.Quantity)

FROM Products p

JOIN Orders o ON p.ProductID = o.ProductID

GROUP BY Category;

**Q7 b) Use GROUP BY with ROLLUP to compute subtotal and grand total**

SELECT Category, SUM(Price \* o.Quantity)

FROM Products p

JOIN Orders o ON p.ProductID = o.ProductID

GROUP BY Category WITH ROLLUP;

**Q7 c) Use HAVING to filter categories with revenue > 100000**

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

FROM Products p

JOIN Orders o ON p.ProductID = o.ProductID

GROUP BY Category

HAVING Revenue > 100000;

**Section D: Complex Joins, Subqueries, and Set Ops**

**Q8 a) Self join to list customers referred by other customers**

ALTER TABLE Customers ADD ReferredBy INT;

SELECT c1.Name, c2.Name

FROM Customers c1

JOIN Customers c2 ON c1.ReferredBy = c2.CustomerID;

**Q8 b) Equi join across Orders and Products**

SELECT o.OrderID, p.ProductName, o.Quantity

FROM Orders o

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

**Q8 c) Join Customers and Orders to display top 3 spenders using window function**

SELECT CustomerID, SUM(p.Price \* o.Quantity),

RANK() OVER (ORDER BY SUM(p.Price \* o.Quantity) DESC)

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY CustomerID;

**Q8 d) LEFT OUTER JOIN with WHERE NULL to identify inactive customers**

SELECT c.\*

FROM Customers c

LEFT JOIN Orders o ON c.CustomerID = o.CustomerID

WHERE o.OrderID IS NULL;

**Q8 e) Cross join for all product combinations in a bundle offer**

SELECT p1.ProductName, p2.ProductName

FROM Products p1

CROSS JOIN Products p2

WHERE p1.ProductID < p2.ProductID;

**Q9 a) Correlated subquery to get customers whose order amount exceeds their average**

SELECT DISTINCT o.CustomerID

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

WHERE p.Price \* o.Quantity > (

SELECT AVG(p2.Price \* o2.Quantity)

FROM Orders o2

JOIN Products p2 ON o2.ProductID = p2.ProductID

WHERE o2.CustomerID = o.CustomerID

);

**Q9 b) Subquery using EXISTS to find customers with at least 2 different products**

SELECT \* 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

);

**Q9 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

);

**Q9 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'

);

**Q9 e) Nested subquery to list top 3 best-selling products**

SELECT ProductID, SUM(Quantity)

FROM Orders

GROUP BY ProductID

ORDER BY SUM(Quantity) DESC

LIMIT 3;

**Q10 a) Simulate INTERSECT using INNER JOIN**

CREATE TABLE IF NOT EXISTS SegmentA (

CustomerID INT,

Name VARCHAR(100),

Email VARCHAR(100)

);

CREATE TABLE IF NOT EXISTS SegmentB (

CustomerID INT,

Name VARCHAR(100),

Email VARCHAR(100)

);

INSERT INTO SegmentA VALUES (1, 'Ankit', 'ankit@example.com');

INSERT INTO SegmentB VALUES (1, 'Ankit', 'ankit@example.com'), (3, 'NewUser', 'new@example.com');

SELECT a.CustomerID, a.Name, a.Email

FROM SegmentA a

INNER JOIN SegmentB b ON a.CustomerID = b.CustomerID;

**Q10 b) Use EXCEPT to find products in inventory not yet ordered**

SELECT ProductID FROM Products

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

**Q10 c) Simulate MERGE: If customer exists, update; else insert**

INSERT INTO Customers (CustomerID, Name, Email)

VALUES (3, 'NewUser', 'new@example.com')

ON DUPLICATE KEY UPDATE Name = 'UpdatedUser';

**Q10 d) Use UNION to combine two regional customer tables**

CREATE TABLE IF NOT EXISTS RegionNorth (

CustomerID INT,

Name VARCHAR(100),

Email VARCHAR(100)

);

CREATE TABLE IF NOT EXISTS RegionSouth (

CustomerID INT,

Name VARCHAR(100),

Email VARCHAR(100)

);

INSERT INTO RegionNorth VALUES (101, 'Dev', 'dev@north.com');

INSERT INTO RegionSouth VALUES (201, 'Priya', 'priya@south.com');

SELECT \* FROM RegionNorth

UNION

SELECT \* FROM RegionSouth;

**Q10 e) WITH CTE that ranks customers by total spend and filters top 5**

WITH RankedCustomers AS (

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

RANK() OVER (ORDER BY SUM(p.Price \* o.Quantity) DESC) AS Rnk

FROM Orders o

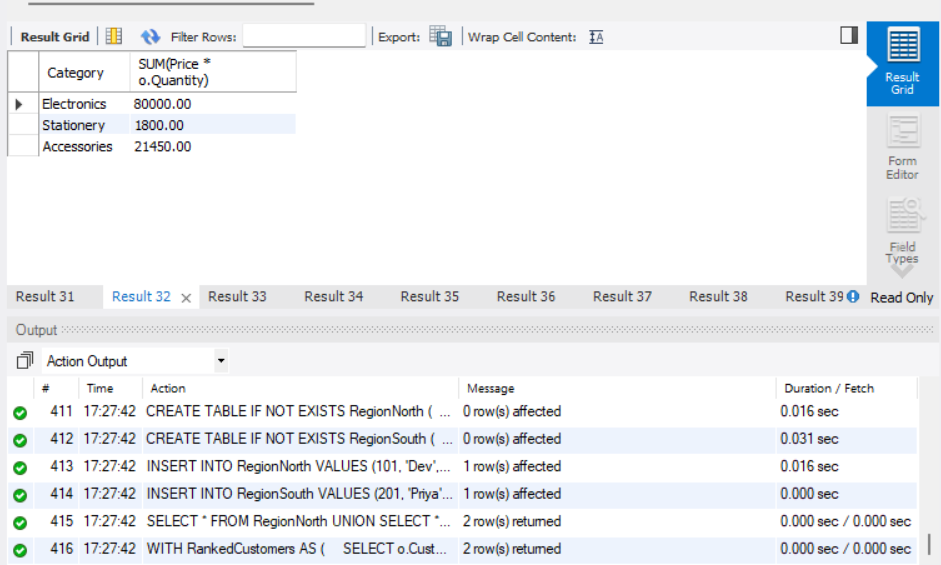
JOIN Products p ON o.ProductID = p.ProductID

GROUP BY o.CustomerID

)

SELECT \* FROM RankedCustomers WHERE Rnk <= 5;

**OUTPUT**

****