SQL Assessment

Section A: Basics & Data Definition (10 Marks)

Q1. (3 marks) Differentiate between SQL and NoSQL. Provide two advantages and two disadvantages of each with real-world examples.

SQL uses structured table with predefined schema. It is vertically scalable.It uses ACID principles.

Advantages- It has strong data integrity and consistency.It also supports complex queries.

Disadvantages-It is harder to scale horizontally and fixed schema can be restrictive.

NOSQL uses flexible schemas like document,graph based.It is horizontally scalable. It uses BASE principle.

Advantages- It handles large data very well.It flexible schema.

Disadvantages-It has weaker consistency and fixed support for complex joins.

Q2. (2 marks) Given the below unnormalized data, convert it to 1NF, 2NF, and 3NF: Student (StudentID, Name, CourseID, CourseName, InstructorName, InstructorPhone)

1NF - Student (StudentID, Name, CourseID, CourseName, InstructorID)

Instructor (InstructorID, InstructorName, InstructorPhone)

2NF - Student (StudentID, Name)

Enrollment (StudentID, CourseID)

Course (CourseID, CourseName, InstructorID)

Instructor (InstructorID, InstructorName, InstructorPhone)

3NF - Student (StudentID, Name)

Enrollment (StudentID, CourseID)

Course (CourseID, CourseName, InstructorID)

Instructor (InstructorID, Name, PhoneNumber)

Q3. (5 marks) a) Create a database named StudentDB. b) Create a table Students with fields: StudentID, Name, DOB, Email. c) Rename the table to Student\_Info. d) Add a column PhoneNumber. e) Drop the table

a) Create database StudentDB;

b) create table Students (

StudentID int primary key,

Name varchar(50),

DOB date,

Email varchar(100)

);

c) alter table Students rename to Student\_Info;

d) alter table Student\_Info add PhoneNumber varchar(50);

e)drop table Student\_Info;

Section B: DML & Filtering Data (15 Marks)

Q4. (5 marks) a) Insert 3 student records into Student\_Info. b) Update one student's phone number. c) Delete one student whose email ends with @gmail.com. d) Retrieve only names and emails of students born after the year 2000. e) Retrieve distinct domain names from the email column.

a)Insert into Student\_Info (StudentID, Name, DOB, Email, PhoneNumber) values

(1, 'A', '2001-05-12', 'a@gail.com', '9876543210'),

(2, 'B', '1999-09-25', 'b@gmil.com', '9123456789'),

(3, 'C', '2002-01-15', 'c@gmail.com', '9234567890');

b) Update Student\_Info set PhoneNumber = '9000000001' where Name = 'Meera';

c)Delete from Student\_Info where Email LIKE ['% @gmail.com';](mailto:'%@gmail.com';)

d)select Name, Email from Student\_Info where year(DOB) > 2000;

e)select distinct substring\_index(Email, '@', -1) as DomainName from Student\_Info;

Q5. (5 marks) a) Retrieve students with names starting with 'A'. b) Retrieve students with phone number between 9000000000 and 9999999999. c) Retrieve students using IN operator on city names. d) Use AND, OR to filter students based on age and email providere)e) Use table and column aliasing in a query to get all student names and DOBs.

1. select \* from Student\_Info where Name like 'A%';
2. select \* from Student\_Info where PhoneNumber between 9000000000 and 9999999999;

C)select \* from Student\_Info where City in ('Chennai', 'Mumbai', 'Delhi');

d)select \* from Student\_Info where year(DOB) < 2000 and Email like '%@yahoo.com'

or PhoneNumber between 9100000000 and 9199999999;

e)select s.Name as Student\_Name, s.DOB as Birth\_Date from Student\_Info s;

Q6. (5 marks) Create a new table Marks(StudentID, Subject, Marks). Insert at least 3 rows. a) Display student IDs and their subjects where marks > 70. b) Display subjects with average marks. c) Filter subjects with average marks between 60 and 90

A)create table Marks ( StudentID int, Subject varchar(50), Marks int);

Insert into Marks (StudentID, Subject, Marks) values

(1, 'Maths', 85),(2, 'Science', 72),(3, 'English', 65);

1. select StudentID, Subject from Marks where Marks > 70;
2. select Subject, avg(Marks) as AverageMarks from Marks Group by Subject;
3. select Subject, avg(Marks) as AverageMarks from Marks group by Subject Having Avg(Marks) between 60 and 90;

Section C: Functions & Grouping (10 Marks) Q7. (5 marks) a) Get the current date and format it as "YYYY-MM-DD". b) Extract month and year from a DOB column. c) Convert a student's name to uppercase. d) Round off marks to 2 decimal places. e) Use system function to return user name or current database.

1. select date\_format(curdate(), '%Y-%m-%d') as CurrentDate;
2. select Name, MONTH(DOB) as Month, year(DOB) as Year from Student\_Info;
3. select upper(Name) as UpperCaseName from Student\_Info;
4. select StudentID, Subject, round(Marks, 2) as RoundedMarks from Marks;
5. select user() as CurrentUser, database() as CurrentDatabase;

Q8. (5 marks) a) Display total marks of each student. b) Display subject-wise highest mark. c) Use GROUP BY and HAVING to display subjects with average marks > 75.

1. select StudentID, sum(Marks) as TotalMarks from Marks group by StudentID;
2. select Subject, max(Marks) as HighestMark from Marks group by Subject;
3. select Subject, avg(Marks) as AverageMarks from Marks group by Subject

having avg(Marks) > 75;

Q9. (5 marks) a) Inner Join to retrieve students and their courses. b) Left Join to get all students even if not enrolled. c) Right Join to get all courses even if no students. d) Full Outer Join equivalent using UNION. e) Cross Join to show all combinations

1. select s.Name, c.CourseName from Students s

inner join Courses c on s.CourseID = c.CourseID;

B)select s.Name, c.CourseName from Students s

left JOIN Courses c on s.CourseID = c.CourseID;

c)select s.Name, c.CourseName from Students s

right join Courses c on s.CourseID = c.CourseID;

1. select s.Name, c.CourseName from Students s

left join Courses c on s.CourseID = c.CourseID

Union

select s.Name, c.CourseName from Students s

right join Courses c on s.CourseID = c.CourseID;

1. select s.Name, c.CourseName from Students s

cross join Courses c;

Q10. (5 marks) a) Students who scored more than average in 'Maths'. b) Students not in the Marks table. c) Use EXISTS to get students with at least one subject. d) Use ALL to find those scoring more than all in 'Science'. e) Use ANY for students scoring better than some in 'English'.

1. select StudentID, Name from Marks where Subject = 'Maths'

and Marks > (select avg(Marks) from Marks where Subject = 'Maths');

B select StudentID, Name from Students

where StudentID not in (select distinct StudentID from Marks);

C select Name from Students s where

exists (select 1 from Marks m where s.StudentID = m.StudentID);

D select StudentID, Name from Marks where Subject = 'Science' and

Marks > all (select Marks from Marks where Subject = 'Science');

E select StudentID, Name from Marks where Subject = 'English' and Marks > any (selecT Marks from Marks where Subject = 'English');

Q11. (5 marks) a) UNION of student names from two tables. b) INTERSECT to find common students. c) EXCEPT to list students in Students but not in Marks. d) MERGE concept or simulate with UPDATE and INSERT. e) Correlated subquery to list students with above average per subject

A select Name from Students

Union select Name from Marks;

B select Name from Students intersect select Name from Marks;

C select Name from Students except select Name from Marks;

eselect StudentID, Subject, Marks from Marks m1

where Marks > (select avg(Marks) from Marks m2 where m1.Subject = m2.Subject);

SQL assessment - 2

Q1

High scalability & performance: NoSQL databases like MongoDB are used in systems handling massive real-time data (e.g., sensor networks, social media).

Unstructured or semi-structured data: When data formats are dynamic, such as JSON logs or user-generated content.

Types of NoSQL databases with real-time examples:

1.Document-oriented (e.g., MongoDB):

2.Key-Value Store (e.g., Redis):

3.Column-family Store (e.g., Apache Cassandra):

4.Graph Databases (e.g., Neo4j):

Types of NoSQL databases with real-time examples:

Section B: Complex DDL and DML (15 Marks)

Q3. (5 marks) a) Create a database RetailDB and design a schema for Customers, Orders, and Products with primary and foreign keys. b) Implement a check constraint on Quantity (>0) in Orders. c) Alter the Products table to add 'Discount' column and update some values.

a)

create database RetailDB;

use RetailDB;

B)

create table Customers (CustomerID Int primary key, Name varchar(100));

create table Products (ProductID int primary key, ProductName varchar(100), Price int);

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));

c) alter table Products ADD Discount int;

update Products SET Discount = 5 where ProductID = 1;

Q4. (5 marks) Using the above schema: a) Insert 3 sample orders per customer. b) Update prices with 10% increase where quantity sold > 5. c) Delete orders where the product has never been sold.

A insert into Orders values (101, 1, 1, 2), (102, 1, 2, 3), (103, 1, 3, 1);

b) update Products set Price = Price \* 1.10 where ProductID IN (select ProductID

from Orders group by ProductID having sum(Quantity) > 5 );

c) delete from Orders where ProductID not in (select distinct ProductID from Orders);

Q5 Retrieve the following: a) Customers who ordered more than 3 different products. b) Products not ordered by any customer. c) Count of orders placed by each customer in the last 30 days

A.select CustomerID from Orders group by CustomerID having count(distinct ProductID) > 3;

b)select productid from products where productid not in (select distinct ProductID from Orders);

Q6. (5 marks)

A.select lower(Email) AS StandardizedEmail, substring\_index(Email, '@', 1) AS Username,

substring\_index(Email, '@', -1) as Domain from Customers;

B.select OrderID, datediff(CURDATE(), OrderDate) as DaysSinceOrder

from Orders;

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

select current\_user() AS User,

host\_name() AS Host;

select system\_user();

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

select

concat('Hello ', upper(Name), '! Your email is ', Email) as Greeting

from Customers;

Q7. (5 marks)

1. Aggregate total revenue by product category.

select

p.Category,

sum(o.Quantity \* p.Price) 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(o.Quantity \* p.Price) AS Revenue

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category

HAVING Revenue > 100000;

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

SELECT

p.Category,

SUM(o.Quantity \* p.Price) AS Revenue

FROM Orders o

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY p.Category

HAVING Revenue > 100000;

# Section D: Complex Joins, Subqueries, and Set Ops (25 Marks)

Q8. (5 marks)

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

SELECT c1.Name AS Customer, c2.Name AS ReferredBy

FROM Customers c1

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

1. Equi join across Orders and Products.

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

FROM Orders o

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

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 Rank

FROM Customers c

JOIN Orders o ON c.CustomerID = o.CustomerID

JOIN Products p ON o.ProductID = p.ProductID

GROUP BY c.CustomerID

) AS Ranked

WHERE Rank <= 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 ProductA,

p2.ProductName AS ProductB

FROM Products p1

CROSS JOIN Products p2

WHERE p1.ProductID < p2.ProductID;

Q9. (5 marks)

1. Correlated subquery to get customers whose order amount exceeds their average.

SELECT \*

FROM Orders o1

WHERE o1.Quantity \* (SELECT Price FROM Products WHERE ProductID = o1.ProductID) >

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

FROM Orders o2

JOIN Products p2 ON o2.ProductID = p2.ProductID

WHERE o2.CustomerID = o1.CustomerID);

1. 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.ProductID

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 o2 WHERE o2.CustomerID != c.CustomerID GROUP BY o2.CustomerID

);

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

SELECT \*

FROM Products

WHERE Price > ANY (

SELECT Price

FROM Products

WHERE Category = 'Electronics'

);

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

SELECT \* FROM (

SELECT p.ProductID, p.ProductName, SUM(o.Quantity) AS TotalSold,

RANK() OVER (ORDER BY SUM(o.Quantity) DESC) AS rnk

FROM Products p

JOIN Orders o ON p.ProductID = o.ProductID

GROUP BY p.ProductID

) AS RankedProducts

WHERE rnk <= 3;

Q10. (5 marks)

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

SELECT pc.CustomerID, pc.Name

FROM PremiumCustomers pc

INNER JOIN LoyalCustomers lc ON pc.CustomerID = lc.CustomerID;

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 (101, 'John Doe', '[john@example.com](mailto:john@example.com)')

ON DUPLICATE KEY UPDATE

Name = VALUES(Name),

Email = VALUES(Email);

1. Use UNION to combine two regional customer tables.

SELECT CustomerID, Name, Email FROM EastRegionCustomers

UNION

SELECT CustomerID, Name, Email FROM WestRegionCustomers;

1. 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 TotalSpent,

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

)

SELECT CustomerID, Name, TotalSpent

FROM CustomerSpending

WHERE rnk <= 5;

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');

Querying Data by Using Subqueries - Examples

Sample Table: Employees

CREATE TABLE Employees (EmpID INT, Name VARCHAR(50), Department VARCHAR(50), Salary INT);

INSERT INTO Employees VALUES (1, 'Alice', 'HR', 5000);

INSERT INTO Employees VALUES (2, 'Bob', 'IT', 7000);

INSERT INTO Employees VALUES (3, 'Charlie', 'Finance', 6000);

INSERT INTO Employees VALUES (4, 'David', 'IT', 8000);

INSERT INTO Employees VALUES (5, 'Eva', 'HR', 5500);

INSERT INTO Employees VALUES (6, 'Frank', 'Finance', 6200);

Querying Data by Using Subqueries

Query:

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

Querying Data by Using Subqueries Using the EXISTS

Query:

SELECT Name FROM Employees e WHERE EXISTS (SELECT 1 FROM Employees WHERE Department = 'IT' AND e.Department = Department);

Querying Data by Using Subqueries using ANY

Query:

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

Querying Data by Using Subqueries using ALL Keywords

Query:

SELECT Name FROM Employees WHERE Salary > ALL (SELECT Salary FROM Employees WHERE Department = 'HR');

Querying Data by Using Subqueries using Nested Subqueries

Query:

SELECT Name FROM Employees WHERE Salary = (SELECT MAX(Salary) FROM Employees WHERE Department = (SELECT Department FROM Employees WHERE Name = 'Charlie'));

Querying Data by Using Subqueries Using Correlated Subqueries

Query:

SELECT Name FROM Employees e1 WHERE Salary > (SELECT AVG(Salary) FROM Employees e2 WHERE e1.Department = e2.Department);

Querying Data by Using Subqueries Using UNION

Query:

SELECT Name FROM Employees WHERE Department = 'HR' UNION SELECT Name FROM Employees WHERE Salary > 7000;

Querying Data by Using Subqueries using INTERSECT

Query:

SELECT Name FROM Employees WHERE Department = 'IT' INTERSECT SELECT Name FROM Employees WHERE Salary > 7000;

Querying Data by Using Subqueries using EXCEPT

Query:

SELECT Name FROM Employees WHERE Department = 'IT' EXCEPT SELECT Name FROM Employees WHERE Salary > 7000;

Querying Data by Using Subqueries using MERGE

Query:

MERGE INTO Employees AS target USING (SELECT 2 AS EmpID, 'Bob' AS Name) AS source ON target.EmpID = source.EmpID WHEN MATCHED THEN UPDATE SET Salary = 7500;

**Medium-Level Practical SQL Questions**

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 TABLE students (

student\_id INT PRIMARY KEY,

name VARCHAR(100) NOT NULL UNIQUE,

marks INT NOT NULL

);

INSERT INTO students VALUES

(1, 'Alice', 85),

(2, 'Bob', 78),

(3, 'Charlie', 92),

(4, 'David', 88),

(5, 'Eva', 90);

UPDATE students SET marks = 95 WHERE name = 'Bob';

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

-- Create the customers table

CREATE TABLE customers (

customer\_id INTEGER PRIMARY KEY AUTOINCREMENT,

full\_name TEXT NOT NULL

);

-- Insert sample data

INSERT INTO customers (full\_name) VALUES

('John Doe'),

('Jane Smith'),

('Emily Johnson');

-- String function query to extract first and last names and their lengths

SELECT

full\_name,

SUBSTR(full\_name, 1, INSTR(full\_name, ' ') - 1) AS first\_name,

SUBSTR(full\_name, INSTR(full\_name, ' ') + 1) AS last\_name,

LENGTH(SUBSTR(full\_name, 1, INSTR(full\_name, ' ') - 1)) AS first\_name\_length,

LENGTH(SUBSTR(full\_name, INSTR(full\_name, ' ') + 1)) AS last\_name\_length

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

CREATE TABLE sales (

sale\_id INT AUTO\_INCREMENT PRIMARY KEY,

sale\_date DATE NOT NULL

);

INSERT INTO sales (sale\_date) VALUES

('2025-06-10'),

('2025-05-20'),

('2025-04-01');

SELECT

sale\_id,

sale\_date,

MONTHNAME(sale\_date) AS month\_name,

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

CREATE TABLE employees (

emp\_id INT AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100) NOT NULL,

salary DECIMAL(10, 2) NOT NULL

);

INSERT INTO employees (name, salary) VALUES

('Alice', 48350.75),

('Bob', 55990.00),

('Charlie', 61240.25),

('Diana', 70000.00);

SELECT

emp\_id,

name,

salary,

ROUND(salary \* 1.10, 2) AS salary\_after\_10\_percent\_hike,

ROUND(salary, -2) AS rounded\_to\_nearest\_100

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,

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'

CREATE TABLE products (

product\_id INT AUTO\_INCREMENT PRIMARY KEY,

product\_name VARCHAR(100) NOT NULL,

price DECIMAL(10, 2) DEFAULT NULL

);

INSERT INTO products (product\_name, price) VALUES

('Laptop', 75000.00),

('Tablet', NULL),

('Smartphone', 35000.00),

('Headphones', NULL);

SELECT

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

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

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 AUTO\_INCREMENT PRIMARY KEY,

amount DECIMAL(10, 2)

);

INSERT INTO transactions (amount) VALUES

(250.00), (499.99), (120.75), (780.50), (350.25);

SELECT

SUM(amount) AS total\_sales,

AVG(amount) AS average\_sale\_value,

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

CREATE TABLE sales (

sale\_id INT AUTO\_INCREMENT PRIMARY KEY,

product\_category VARCHAR(50),

sale\_amount DECIMAL(10, 2)

);

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

('Electronics', 1200.50),

('Clothing', 750.00),

('Electronics', 500.00),

('Clothing', 300.25),

('Furniture', 1500.00);

SELECT

product\_category,

SUM(sale\_amount) AS total\_sales,

COUNT(\*) AS number\_of\_transactions

FROM sales

GROUP BY product\_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 AUTO\_INCREMENT PRIMARY KEY,

name VARCHAR(100)

);

CREATE TABLE orders (

order\_id INT AUTO\_INCREMENT PRIMARY KEY,

customer\_id INT,

order\_amount DECIMAL(10, 2),

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

);

INSERT INTO customers (name) VALUES

('Alice'), ('Bob'), ('Charlie');

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

(1, 500.00),

(2, 1200.75),

(1, 300.00); -- Charlie didn't order

10. Left Join for Products with or without Orders:

Show all products with:

- Their order details (if available)

- Use LEFT JOIN

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100)

);

CREATE TABLE orders (

order\_id INT PRIMARY KEY,

product\_id INT,

quantity INT,

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

);

INSERT INTO products VALUES

(1, 'Laptop'), (2, 'Phone'), (3, 'Tablet');

INSERT INTO orders VALUES

(101, 1, 2), (102, 2, 1); -- No order for Tablet

SELECT

p.product\_name,

o.order\_id,

o.quantity

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 customers (

customer\_id INT PRIMARY KEY,

name VARCHAR(100)

);

CREATE TABLE contacts (

contact\_id INT PRIMARY KEY,

customer\_id INT,

email VARCHAR(100),

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

);

INSERT INTO customers VALUES

(1, 'Alice'), (2, 'Bob'), (3, 'Charlie');

INSERT INTO contacts VALUES

(201, 1, 'alice@mail.com'), (202, 2, 'bob@mail.com'); -- Charlie has no contact

SELECT

c.customer\_id,

c.name,

ct.email

FROM contacts ct

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

);

CREATE TABLE products (

product\_id INT PRIMARY KEY,

product\_name VARCHAR(100),

supplier\_id INT

);

INSERT INTO suppliers VALUES

(1, 'Supplier A'), (2, 'Supplier B');

INSERT INTO products VALUES

(10, 'Laptop', 1), (11, 'Monitor', NULL);

-- Left join

SELECT

s.supplier\_name,

p.product\_name

FROM suppliers s

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

UNION

-- Right join

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\_name VARCHAR(50)

);

-- Assume 'products' table already exists

INSERT INTO offers VALUES

(1, '10% Off'), (2, 'Buy 1 Get 1');

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

ALTER TABLE products ADD COLUMN category VARCHAR(50);

UPDATE products

SET category = CASE

WHEN product\_name = 'Laptop' THEN 'Electronics'

WHEN product\_name = 'Phone' THEN 'Electronics'

WHEN product\_name = 'Tablet' THEN 'Electronics'

ELSE 'General'

END;

SELECT

p.category,

SUM(o.quantity) AS total\_quantity\_sold,

AVG(p\_price.price) AS average\_price

FROM orders o

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

JOIN (

SELECT product\_id, 50000 AS price FROM products

) AS p\_price ON o.product\_id = p\_price.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 students (

student\_id INT PRIMARY KEY,

name VARCHAR(100)

);

CREATE TABLE marks (

mark\_id INT PRIMARY KEY,

student\_id INT,

subject VARCHAR(50),

score INT,

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

);

INSERT INTO students VALUES

(1, 'Ravi'), (2, 'Neha'), (3, 'John');

INSERT INTO marks VALUES

(1, 1, 'Math', 80), (2, 1, 'Science', 90),

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

(5, 3, 'Math', 95), (6, 3, 'Science', 85);

SELECT

s.name AS student\_name,

AVG(m.score) AS average\_marks

FROM students s

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

GROUP BY s.student\_id

HAVING AVG(m.score) > 75;