# Section-I(MCQ)

**1**. What is the purpose of the SQL keyword "WHERE" in a query?

Answer:

**(c).to filter rows based on a condition**

2. In a retail database, whenever a new product is added to the "Products" table, the system automatically updates the inventory count in the "Inventory" table. Which SQL feature can be used to implement this automatic update process?

Answer:

**(c). Trigger**

3. In a retail database, the sales manager used SQL to analyze product sales performance. The manager applied the ORDER BY clause in the SELECT query to _____.

**(d). Sort the result set based on specified columns.**

4. In a government agency's database, the security team used a command from DCL to remove specific access rights from a former employee, preventing unauthorized data access after their departure. Which SQL command is used to revoke access privileges from users?

Answer:
**(a). Revoke**

6. In a database system, which of the following SQL-related statements is true?

**(c). The TRUNCATE command is used to delete the information present in the table, but not the table structure.**

8. What is the purpose of the HAVING clause in SQL?

**(b). To filter the rows based on specific conditions**

# Section - II (Short Answer Questions)

**1. Implement a trigger to update the EmployeeCount column in the "Departments" table whenever a new employee is added. Consider the following structure of tables while writing the query –Employees (EmployeeID,FirstName, LastName, DepartmentID), Departments (DepartmentID , DepartmentName, EmployeeCount)**

Answer:

```
CREATE TRIGGER UpdateEmployeeCount

AFTER INSERT ON Employees

FOR EACH ROW

BEGIN

   UPDATE Departments

   SET EmployeeCount = EmployeeCount + 1

   WHERE DepartmentID = NEW.DepartmentID;

END;
```

**2. What is the difference between a primary key and a foreign key?**

| Primary key | Foreign key |
|---|---|
| It uniquely identifies rows or records in a table or database objects. | It is a database concept that establishes a relationship between two tables. |
| Primary key does not have null vales. | Foreign key can have null values. |
| Primary key ensures no duplicate values. | Foreign key can have duplicate values. |
| Primary key constraints is applied only once in a table. | Foreign key constraints can be applied multiple times within a single table. |
| Automatically indexed for fast searching. | Not automatically indexed; indexing is optional but often recommended. |

**3.What is the difference between a CREATE TABLE statement and an ALTER TABLE statement?**

| CREATE | ALTER |
|---|---|
| Creates a new table in the database with specified columns, data types, and constraints. | Modifies the structure of an existing table, including adding, removing, or changing columns or constraints. |
| CREATE TABLE table_name (column1 datatype, column2 datatype, ... constraint); | ALTER TABLE table_name ADD column_name datatype;<br>ALTER TABLE table_name MODIFY column_name datatype;<br>ALTER TABLE table_name DROP column_name; |

| | |
|---|---|
| Defines the entire table structure from scratch, including columns, data types, primary keys, and other constraints. | Alters only specific parts of an existing table, such as adding a new column, modifying data types, or dropping columns or constraints. |
| Creates a new, empty table; no data exists until rows are inserted after creation. | Changes the structure without impacting existing data (except if a column is dropped, then data in that column is lost). |
| Not applicable as this statement is used to create, not delete tables. | Can be used to delete columns from an existing table using DROP COLUMN. |
| The table can be created without dependencies or references to other tables. | Any change to a table's structure might impact queries, views, or other objects depending on it. |

**4. What is the difference between the WHERE and HAVING clauses in a SELECT statement?**

| WHERE | HAVING |
|---|---|
| Filters rows based on specified conditions before any grouping occurs. | Filters groups created by the GROUP BY clause, based on aggregate functions or conditions. |
| Used with SELECT, UPDATE, and DELETE statements to filter individual rows. | Used with GROUP BY in SELECT statements to filter grouped results. |
| Cannot use aggregate functions (like SUM, COUNT, AVG) directly in conditions. | Can use aggregate functions in conditions (e.g., HAVING COUNT(*) > 5). |
| Executes before GROUP BY and HAVING clauses. | Executes after WHERE and GROUP BY clauses. |
| Example:<br>SELECT * FROM Employees WHERE Age > 30; | Example:<br>SELECT DepartmentID, COUNT(*) FROM Employees GROUP BY DepartmentID HAVING COUNT(*) > 10; |

**5. Differentiate between SQL INNER JOIN and SQL LEFT JOIN. Include an example for each**.

| **INNER JOIN** | **LEFT JOIN** |
|---|---|
| Returns only the rows that have matching values in both tables. | Returns all rows from the left (first) table, and matching rows from the right (second) table. If no match, NULL values are returned for the right table columns. |
| Only includes rows with a match in both tables. | Includes all rows from the left table, regardless of whether there is a match in the right table. |
| Useful when you only want records that exist in both tables. | Useful when you need all records from the left table and matching or null records from the right table. |
| Doesn't return NULLs for unmatched rows, as only matching rows are included. | Returns NULLs for columns in the right table when there is no matching row. |
| Finding employees who have been assigned to a department | Finding all employees, along with their department if assigned, or NULL if unassigned. |
| Generally faster, as it only processes matching rows. | Slightly slower than INNER JOIN when the left table has many unmatched rows to fill with NULLs. |
| Example<br>SELECT Employees.EmployeeID, Employees.EmployeeName, Departments.DepartmentName<br>FROM Employees | Example<br>SELECT Employees.EmployeeID, Employees.EmployeeName, Departments.DepartmentName<br>FROM Employees |

| | |
|---|---|
| INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID; | INNER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID; |

**8. Explain the purpose of a FROM subquery and how it's used.**

**Answer:**

1.Simplifies Complex Queries: Allows you to structure complicated queries by creating intermediate results.

2. Reduces Redundancy: Avoids repeating the same subquery or calculation in multiple parts of a query.

3. Enables Further Filtering or Aggregation: Allows you to filter, join, or aggregate results from a subquery before using them in the main query.

Uses:

1. Define the Subquery: Write a subquery that returns a result set. This subquery can include SELECT, WHERE, GROUP BY, and other SQL clauses.

2.Use the Subquery in the FROM Clause: Incorporate the subquery within the FROM clause of your main query. Assign an alias to the subquery for reference.

3.Reference the Alias: Use the alias to reference the columns of the subquery in the main query.

4.Add Conditions: You can add WHERE clauses, join conditions, or other filters in the main query to refine the results further.

# Section - III (Long Answer Questions)

**1. Explain the concept of a foreign key in SQL. Provide an example of two tables with a foreign key relationship and demonstrate how to establish this relationship.**

Answer:

Concept of a foreign key:

A column or group of columns in one table that uniquely identifies a row in another table is called a foreign key. It creates a connection between the two tables and is essential to a relational database's referential integrity. The definition of a foreign key guarantees that the values in the foreign key column or columns must either be null or match the values in the referenced primary key column or columns of another table.

Example of Two Tables with a Foreign Key Relationship

Let's consider two tables: Customers and Orders.

Customers Table: Stores information about customers.

Columns: CustomerID, CustomerName, ContactNumber

Orders Table: Stores information about orders placed by customers.

Columns: OrderID, OrderDate, CustomerID (foreign key referencing Customers)

```
CREATE TABLE Customers (
 CustomerID INT PRIMARY KEY,
CustomerName VARCHAR(100) NOT NULL,
ContactNumber VARCHAR(15)
);

CREATE TABLE Orders (
OrderID INT PRIMARY KEY,
OrderDate DATE NOT NULL,
 CustomerID INT,
 FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Inserting Data into the Tables:

```
INSERT INTO Customers (CustomerID, CustomerName, ContactNumber) VALUES (1, 'John Doe', '555-1234');
INSERT INTO Orders (OrderID, OrderDate, CustomerID) VALUES (101, '2024-10-26', 1);
```

A foreign key in SQL establishes a relationship between two tables, enforcing referential integrity. In the example provided, the Orders table contains a foreign key (CustomerID) that references the Customers table, ensuring that any order must be associated with a valid customer. This setup allows for organized data management and prevents data anomalies.

**2. Explain the use of SQL triggers and their significance in database management. Provide an example scenario where a trigger could be used to automate data validation or auditing tasks.**

**Answer:**

**Uses of sql triggers:**

Data validation: By enforcing rules on data inputs, triggers can make sure that information entered into the database satisfies predetermined standards before being committed.

Auditing Changes: Triggers provide the ability to automatically record adjustments to a table, creating an audit trail that may be used to monitor changes over time.

Preserving Referential Integrity: By making sure that activities on one table are accurately reflected in another, triggers can assist in preserving consistency across related tables.

Automating System Tasks: Triggers can reduce manual intervention by starting processes or carrying out calculations automatically when specific criteria are met.

Decoupling Logic from Applications: By allowing specific business logic to exist in the database and be separate from application code, triggers can make it easier to design and maintain applications.

**Scenario**

Suppose you have a table called Employees that stores employee details, including their salary. To ensure data integrity, you want to enforce a rule where no employee can have a salary below the minimum wage, which is set to $15 per hour. Additionally, you want to log any salary changes for auditing purposes in a separate table called SalaryAudit.

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    Salary DECIMAL(10, 2) NOT NULL
);

CREATE TABLE SalaryAudit (
    AuditID INT PRIMARY KEY AUTO_INCREMENT,
    EmployeeID INT,
    OldSalary DECIMAL(10, 2),
```

```
    NewSalary DECIMAL(10, 2),
    ChangeDate DATETIME DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
);
```

Trigger for Salary Validation and Auditing

You can create a trigger that executes before any UPDATE on the Employees table to enforce the salary rule and log any changes.

```
CREATE TRIGGER ValidateAndAuditSalary
BEFORE UPDATE ON Employees
FOR EACH ROW
BEGIN
    -- Validate that the new salary is not below the minimum wage
    IF NEW.Salary < 15 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = 'Salary cannot be below the minimum wage of $15.';
    END IF;

    -- Log the salary change in SalaryAudit table
    IF OLD.Salary != NEW.Salary THEN
        INSERT INTO SalaryAudit (EmployeeID, OldSalary, NewSalary)
        VALUES (OLD.EmployeeID, OLD.Salary, NEW.Salary);
    END IF;
END;
```

SQL triggers are powerful tools for enforcing rules, automating tasks, and ensuring data integrity within a database. The above example demonstrates how a trigger can validate data and create an audit trail for changes, helping maintain a robust and reliable data management system.

**3. Explain the concept of operator precedence in SQL. How does operator precedence affect the order of operations in SQL expressions?**

Answer:

Concept of Operator Precedence in SQL:

The rules that specify the sequence in which operators in a SQL expression are evaluated are known as operator precedence in SQL. Like mathematical expressions, some operators are evaluated first because they have a higher precedence than others. Writing precise SQL queries and making sure that expressions provide the intended results require an understanding of operator precedence.

Correctness of Results: The results of expressions are influenced by operator precedence. It could have unanticipated consequences if not correctly understood.
Complex Expressions: Understanding which actions are carried out first helps prevent logical mistakes in complex SQL queries with numerous operators.

Fundamental Guidelines for Operator Precedence
The general hierarchy of operator precedence in SQL is as follows, arranged from highest to lowest:

Unary operators (for example, unary plus/minus, +, -)
Exponentiation (in certain SQL languages, for example, ^
Division and Multiplication (e.g., *, /)
Subtraction and addition (e.g., +, -)
Operators for comparison (such as =, <, >, <=, >=, <>)
Operators of logic (such as AND, OR, and NOT)

How Operator Precedence Affects Order of Operations

When evaluating an SQL expression, SQL evaluates operators according to their precedence. Operators with higher precedence are evaluated before those with lower precedence.

Example of Operator Precedence

Consider the following SQL expression:

SELECT 10 + 5 * 2;

Evaluation Order:

1. The multiplication operator (*) has higher precedence than addition (+).
2. Therefore, the multiplication is performed first: 5 * 2 results in 10.
3. Then the addition is performed: 10 + 10 results in 20.

Evaluation Order:

The expression inside the parentheses is evaluated first: 10 + 5 results in 15.

Then the multiplication is performed: 15 * 2 results in 30.

**4. Explain the advantages of using stored procedures in SQL database systems. Provide an example scenario where a stored procedure could be utilized to streamline complex data manipulation tasks.**

**Advantages of using stored procedures:**

**Improved Performance**: Stored procedures are precompiled and stored in parsed form on the server, which enables faster execution compared to running individual SQL statements, especially for complex operations.

**Enhanced Security**: Stored procedures allow access control, so users can execute them without direct access to underlying tables, protecting sensitive data and enforcing permissions.

**Reduced Network Traffic**: Only the procedure call and parameters are sent over the network, minimizing the amount of data transmitted compared to long SQL statements.

**Code Reusability and Centralization**: Complex logic can be centralized within stored procedures, reducing redundancy, making it reusable across applications, and simplifying maintenance.

**Easier Maintenance**: Updating logic in a stored procedure only requires a single change in one place, which keeps the database consistent and allows easier updates without modifying client applications.

Example Scenario

Scenario: In an e-commerce system, after each purchase, various operations need to be performed, such as updating the inventory, recording the sales transaction, updating customer points, and logging the transaction for audit purposes. Manually executing these operations would be complex, error-prone, and resource-intensive.

Stored Procedure Solution: A stored procedure can be created to streamline these operations by automating the entire process within a single transaction.

```
CREATE PROCEDURE ProcessOrder (
    IN OrderID INT,
    IN CustomerID INT,
    IN ProductID INT,
    IN Quantity INT,
    IN SaleAmount DECIMAL(10, 2)
)
BEGIN
    -- Start Transaction
    START TRANSACTION;

    UPDATE Inventory
    SET StockLevel = StockLevel - Quantity
    WHERE ProductID = ProductID;
```

```
INSERT INTO Sales (OrderID, ProductID, Quantity, SaleAmount, SaleDate)
VALUES (OrderID, ProductID, Quantity, SaleAmount, NOW());

UPDATE Customers
SET Points = Points + FLOOR(SaleAmount / 10)
WHERE CustomerID = CustomerID;

INSERT INTO TransactionLog (OrderID, LogDate, Action)
VALUES (OrderID, NOW(), 'Order Processed');

END;
```

**5. Explain the difference between scalar functions and aggregate functions in SQL. Provide examples of each type of function.**

| scalar | aggregate |
|---|---|
| Operate on single values, returning one result per row. | Operate on a set of values, returning a single result for a group of rows. |
| Perform operations on individual data points (e.g., format, modify). | Calculate summaries or aggregate data for a group of rows. |
| Transforming or formatting data, such as converting text to uppercase. | Summarizing data, such as finding totals, averages, or counts. |
| UPPER(), LOWER(), ROUND(), LENGTH() | SUM(), COUNT(), AVG(), MAX(), MIN() |
| Usually used in the SELECT clause or WHERE clause. | Often used with GROUP BY in SELECT statements. |

**Scalar Function Example:**

**A scalar function performs operations on a single value and returns one output per row.**

**SELECT EmployeeID, UPPER(FirstName) AS UpperFirstName, ROUND(Salary, 2) AS RoundedSalary**

**FROM Employees;**

**Explanation: The UPPER() function converts the FirstName to uppercase for each row, and ROUND() rounds the Salary to two decimal places.**

**Aggregate Function Example:**

**An aggregate function performs a calculation on a set of values and returns a single result.**

**SELECT DepartmentID, AVG(Salary) AS AverageSalary, COUNT(EmployeeID) AS EmployeeCount**

**FROM Employees**

**GROUP BY DepartmentID;**

**Summary**

- **Scalar functions operate on individual values, modifying or formatting data one row at a time.**
- **Aggregate functions summarize data across multiple rows, typically used for totals, averages, or counts in data analysis.**

# Section - IV (Scenario Based Questions)

**1. Write a SQL query to get the department names and the corresponding number of employees in each department. Consider the following structure of tables while writing the query – Employees (EmployeeID,FirstName, LastName, DepartmentID),Departments (DepartmentId , DepartmentName)**

**Query:**

```
SELECT d.DepartmentName, COUNT(e.EmployeeID) AS EmployeeCount

FROM Departments d

LEFT JOIN Employees e ON d.DepartmentID = e.DepartmentID

GROUP BY d.DepartmentName;
```

**Output:**

| DepartmentName | EmployeeCount |
|---|---|
| Marketing | 2 |
| Sales | 1 |
| Finance | 1 |
| Human Resources | 1 |

**2. Write a SQL query to get the last names of all employees who work in the marketing department. Consider the following structure of tables while writing the query – Employees (EmployeeID,FirstName, LastName, DepartmentID), Departments (DepartmentId , DepartmentName)**

**Query:**

```
SELECT e.LastName

FROM Employees e

JOIN Departments d ON e.DepartmentID = d.DepartmentID

WHERE d.DepartmentName = 'Marketing';
```

**Output:**

| LastName |
|---|
| Doe |
| Johnson |

**3. Write a SQL query to get the first names of all employees who have a salary greater than $50,000. Consider the following structure of table while writing the query – Employees (EmployeeID,FirstName, LastName, Salary)**

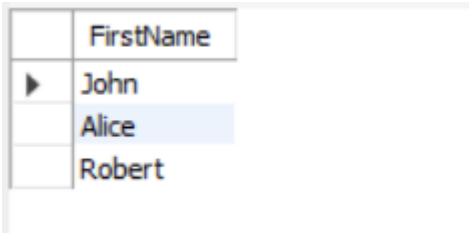
**Query:**

```
SELECT FirstName
```

**FROM Employees**

**WHERE Salary > 50000;**

**Output:**



| | FirstName |
|---|---|
| ▶ | John |
| | Alice |
| | Robert |

**4. Design a trigger that automatically updates the count of customers in each city whenever a new customer is added or an existing customer's city is modified. Consider the following structure of table while writing the query – Customers (CustomerID,CustomerName, City), CityCustomerCount (City, CustomerCount) .**

**Query:**

```
DELIMITER //
CREATE TRIGGER UpdateCityCountAfterInsert
AFTER INSERT ON Customers
FOR EACH ROW
BEGIN
    IF EXISTS (SELECT * FROM CityCustomerCount WHERE City = NEW.City) THEN
        UPDATE CityCustomerCount
        SET CustomerCount = CustomerCount + 1
        WHERE City = NEW.City;
    ELSE
        INSERT INTO CityCustomerCount (City, CustomerCount)
        VALUES (NEW.City, 1);
    END IF;
END;
//


DELIMITER ;
```

**-- Trigger to update count on customer city update**

```
DELIMITER //

CREATE TRIGGER `UpdateCityCountAfterUpdate`
AFTER UPDATE ON `Customers`
FOR EACH ROW
BEGIN
   IF OLD.City <> NEW.City THEN
      -- Decrement count in old city
      UPDATE CityCustomerCount
      SET CustomerCount = CustomerCount - 1
      WHERE City = OLD.City;

      -- Increment count in new city
      IF EXISTS (SELECT * FROM CityCustomerCount WHERE City = NEW.City) THEN
         UPDATE CityCustomerCount
         SET CustomerCount = CustomerCount + 1
         WHERE City = NEW.City;
      ELSE
         INSERT INTO CityCustomerCount (City, CustomerCount)
         VALUES (NEW.City, 1);
      END IF;
   END IF;
END;
//
```

**DELIMITER ;**

**Output:**

```
INSERT INTO Customers (CustomerID, CustomerName, City)
VALUES (206, 'CustomerF', 'New York');
SELECT * FROM CityCustomerCount;
```

| City | CustomerCount |
|---|---|
| Chicago | 1 |
| Los Angeles | 2 |
| New York | 3 |
| NULL | NULL |

**UPDATE Customers**

SET City = 'Los Angeles'

WHERE CustomerID = 201;

SELECT * FROM CityCustomerCount;

| City | CustomerCount |
|---|---|
| Chicago | 1 |
| Los Angeles | 3 |
| New York | 2 |
| NULL | NULL |

**5. Write a SQL query to get the employees and the corresponding departments for each employee.**

Query:

SELECT e.EmployeeID, e.FirstName, e.LastName, d.DepartmentName

FROM Employees e

JOIN Departments d ON e.DepartmentID = d.DepartmentID;

**Output:**

| EmployeeID | FirstName | LastName | DepartmentName |
|---|---|---|---|
| 101 | John | Doe | Marketing |
| 103 | Alice | Johnson | Marketing |
| 102 | Jane | Smith | Sales |
| 104 | Robert | Brown | Finance |
| 105 | Emily | Davis | Human Resources |

Result 12