Question 1: Explain the fundamental differences between DDL, DML, and DQL commands in SQL. Provide one example for each type of command.
Answer:
SQL (Structured Query Language) is divided into different categories based on the type of operations performed on the database. The three main categories are **DDL**, **DML**, and **DQL**.
1. DDL (Data Definition Language)

- **Purpose:** Defines and manages the structure of the database (tables, schemas, indexes, etc.).
- **Key Operations:** CREATE, ALTER, DROP, TRUNCATE.
- **Effect:** Changes the schema of the database permanently.
- **Auto-commit:** Yes (changes are saved automatically).

Example:
```
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   StudentName VARCHAR(100),
   Age INT
);
```
This command creates a new table named Students.
2. DML (Data Manipulation Language)

- **Purpose:** Deals with manipulation of data stored in database tables.
- **Key Operations:** INSERT, UPDATE, DELETE.
- **Effect:** Changes the actual data in the tables, not the structure.
- **Auto-commit:** No (requires explicit COMMIT to save changes).

Example:
```
INSERT INTO Students (StudentID, StudentName, Age)
VALUES (1, 'Alice', 21);
```
This command inserts a new record into the Students table.
3. DQL (Data Query Language)

- **Purpose:** Used to query and retrieve data from the database.
- **Key Operation:** SELECT.
- **Effect:** Only reads data; does not modify it.

Example:
```
SELECT * FROM Students;
```
This command retrieves all the data from the Students table.

Question 2: What is the purpose of SQL constraints? Name and describe three common types of constraints, providing a simple scenario where each would be useful.
Answer:
Purpose of SQL Constraints:
SQL **constraints** are rules applied to table columns to ensure the **accuracy, reliability, and integrity** of the data stored in a database.
They help maintain consistent and valid data by restricting invalid or duplicate entries.
Common Types of Constraints:
1. PRIMARY KEY Constraint

- **Definition:**
  Ensures that each record in a table is **unique** and **cannot contain NULL values**.
  A table can have only one primary key, which uniquely identifies each row.
- **Example Scenario:**
  In a Customers table, the CustomerID can be set as the primary key to uniquely identify every customer.
- **Example Command:**

```
CREATE TABLE Customers (
   CustomerID INT PRIMARY KEY,
   CustomerName VARCHAR(100),
   Email VARCHAR(100)
);
```

2. FOREIGN KEY Constraint

- **Definition:**
  Ensures **referential integrity** by linking a column in one table to a primary key in another table.
  It prevents actions that would break relationships between tables.
- **Example Scenario:**
  In an Orders table, the CustomerID acts as a foreign key that references Customers(CustomerID) — ensuring that an order cannot exist without a valid customer.
- **Example Command:**

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

3. UNIQUE Constraint

- **Definition:**
  Ensures that all values in a column are **distinct**, meaning no duplicate entries are allowed.
  Unlike the primary key, multiple UNIQUE constraints can exist in one table.
- **Example Scenario:**
  In a Users table, the Email field must be unique for each user to prevent duplicate accounts.
- **Example Command:**

```
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    Username VARCHAR(50) UNIQUE,
    Email VARCHAR(100) UNIQUE
);
```

Question 3: Explain the difference between LIMIT and OFFSET clauses in SQL. How would you use them together to retrieve the third page of results, assuming each page has 10 records?

Answer:

LIMIT Clause:

- The **LIMIT** clause in SQL is used to **restrict the number of rows** returned by a query.
- It is commonly used when you only need a specific number of records — for example, showing the first 10 or 20 records in a result set.

Example:

```
SELECT * FROM Products
LIMIT 10;
```

This query retrieves only the **first 10 rows** from the Products table.

OFFSET Clause:

- The **OFFSET** clause is used to **skip a specific number of rows** before starting to return the results.
- It's often used together with LIMIT to implement **pagination** (displaying data in pages).

Example:

```
SELECT * FROM Products
LIMIT 10 OFFSET 10;
```

This skips the **first 10 rows** and retrieves the **next 10 rows** (i.e., page 2).

Using LIMIT and OFFSET Together for Pagination:

To display the **third page of results** where **each page has 10 records**,

- Page 1 → records 1–10 (OFFSET 0)
- Page 2 → records 11–20 (OFFSET 10)
- **Page 3 → records 21–30 (OFFSET 20)**

SQL Query:

```
SELECT * FROM Products
LIMIT 10 OFFSET 20;
```

This query **skips the first 20 records** and retrieves **records 21–30**, effectively showing the **third page**.

Question 4: What is a Common Table Expression (CTE) in SQL, and what are its main benefits? Provide a simple SQL example demonstrating its usage.
Answer:
Definition:
A **Common Table Expression (CTE)** is a **temporary result set** in SQL that exists only during the execution of a single query.
It is defined using the WITH keyword and can be referenced later within a SELECT, INSERT, UPDATE, or DELETE statement.
CTEs are often used to simplify **complex queries**, especially those involving subqueries or recursive logic.
Main Benefits of CTEs:

1. **Improves Readability:**
   Makes long and complex SQL queries easier to understand by breaking them into logical parts.
2. **Supports Recursion:**
   Allows writing recursive queries, such as for hierarchical data (e.g., employee–manager relationships).
3. **Reusability:**
   The same CTE result can be referenced multiple times within the same query.
4. **Simplifies Maintenance:**
   Easier to modify and debug than nested subqueries.

Simple Example:
Suppose we want to find all products priced above the **average product price**.
SQL Query:

```
WITH AveragePrice AS (
   SELECT AVG(Price) AS AvgPrice
   FROM Products
)
SELECT ProductName, Price
FROM Products, AveragePrice
WHERE Products.Price > AveragePrice.AvgPrice;
```

Explanation:

- The CTE AveragePrice first calculates the **average price** of all products.
- The main query then retrieves all products that have a **price greater than the average**.

Example Output (Assuming Sample Data):

ProductName	Price
Laptop Pro	1200.00
Wireless Earbuds	150.00
Blender X	120.00

Question 5: Describe the concept of SQL Normalization and its primary goals. Briefly explain the first three normal forms (1NF, 2NF, 3NF).
Answer:
SQL Normalization is the process of organizing data in a database to **reduce redundancy** and **improve data integrity**.
It involves dividing large, complex tables into smaller, related tables and defining relationships between them using **keys (Primary and Foreign Keys)**.
Primary Goals of Normalization:

1. **Eliminate Data Redundancy:**
   Prevents storing the same piece of data in multiple places.
2. **Ensure Data Integrity:**
   Ensures that updates, deletions, and insertions do not cause data inconsistencies.
3. **Improve Database Efficiency:**
   Reduces storage space and makes queries faster and more reliable.
4. **Simplify Relationships:**
   Makes it easier to manage relationships between tables.

The First Three Normal Forms (1NF, 2NF, 3NF):

1. First Normal Form (1NF):

- **Rule:**
  Each table cell should contain **only one value** (atomic data), and each record must be **unique**.
- **Violation Example (Not 1NF):**

StudentID	Name	Subjects
1	Alice	Math, Science

- **Normalized (1NF) Table:**

StudentID	Name	Subject
1	Alice	Math
1	Alice	Science

2. Second Normal Form (2NF):

- **Rule:**
  The table must be in **1NF** and all **non-key attributes** must depend on the **entire primary key**, not just part of it.
  (Applies mainly to tables with composite keys.)
- **Example:**
  In a table with a composite key (StudentID, CourseID), if StudentName depends only on StudentID, it violates 2NF.
- **Fix:**
  Split the table into two — one for student details and another for course enrollment.

3. Third Normal Form (3NF):

- **Rule:**
  The table must be in **2NF** and **no non-key attribute should depend on another non-key attribute** (no transitive dependency).
- **Example (Not 3NF):**

StudentID	StudentName	Department	DeptHead
1	Alice	Science	Dr. Smith

- Here, DeptHead depends on Department, not directly on StudentID.
- **Fix (3NF):**
  Create a separate Department table:
  - Students(StudentID, StudentName, Department)
  - Departments(Department, DeptHead)

Question 6: Create a database named ECommerceDB and perform the following tasks:
1. Create the following tables with appropriate data types and constraints:

- Categories
  - CategoryID (INT, PRIMARY KEY)
  - CategoryName (VARCHAR(50), NOT NULL, UNIQUE)
- Products
  - ProductID (INT, PRIMARY KEY)
  - ProductName (VARCHAR(100), NOT NULL, UNIQUE)

    - CategoryID (INT, FOREIGN KEY → Categories)
    - Price (DECIMAL(10,2), NOT NULL)
    - StockQuantity (INT)
- Customers
    - CustomerID (INT, PRIMARY KEY)
    - CustomerName (VARCHAR(100), NOT NULL)
    - Email (VARCHAR(100), UNIQUE)
    - JoinDate (DATE)
- Orders
    - OrderID (INT, PRIMARY KEY)
    - CustomerID (INT, FOREIGN KEY → Customers)
    - OrderDate (DATE, NOT NULL)
    - TotalAmount (DECIMAL(10,2))

Answer:

Step 1: Create the Database

```
CREATE DATABASE ECommerceDB;
USE ECommerceDB;
```

Step 2: Create Tables

(a) Categories Table

```
CREATE TABLE Categories (
   CategoryID INT PRIMARY KEY,
   CategoryName VARCHAR(50) NOT NULL UNIQUE
);
```

(b) Products Table

```
CREATE TABLE Products (
   ProductID INT PRIMARY KEY,
   ProductName VARCHAR(100) NOT NULL UNIQUE,
   CategoryID INT,
   Price DECIMAL(10,2) NOT NULL,
   StockQuantity INT,
   FOREIGN KEY (CategoryID) REFERENCES Categories(CategoryID)
);
```

(c) Customers Table

```
CREATE TABLE Customers (
   CustomerID INT PRIMARY KEY,
   CustomerName VARCHAR(100) NOT NULL,
   Email VARCHAR(100) UNIQUE,
   JoinDate DATE
);
```

(d) Orders Table

```
CREATE TABLE Orders (
   OrderID INT PRIMARY KEY,
   CustomerID INT,
   OrderDate DATE NOT NULL,
   TotalAmount DECIMAL(10,2),
   FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
```

Step 3: Insert Data

(a) Categories

```
INSERT INTO Categories (CategoryID, CategoryName) VALUES
```

```
(1, 'Electronics'),
(2, 'Books'),
(3, 'Home Goods'),
(4, 'Apparel');
```

(b) Products

```
INSERT INTO Products (ProductID, ProductName, CategoryID, Price, StockQuantity) VALUES
(101, 'Laptop Pro', 1, 1200.00, 50),
(102, 'SQL Handbook', 2, 45.50, 200),
(103, 'Smart Speaker', 1, 99.99, 150),
(104, 'Coffee Maker', 3, 75.00, 80),
(105, 'Novel: The Great SQL', 2, 25.00, 120),
(106, 'Wireless Earbuds', 1, 150.00, 100),
(107, 'Blender X', 3, 120.00, 60),
(108, 'T-Shirt Casual', 4, 20.00, 300);
```

(c) Customers

```
INSERT INTO Customers (CustomerID, CustomerName, Email, JoinDate) VALUES
(1, 'Alice Wonderland', 'alice@example.com', '2023-01-10'),
(2, 'Bob the Builder', 'bob@example.com', '2022-11-25'),
(3, 'Charlie Chaplin', 'charlie@example.com', '2023-03-01'),
(4, 'Diana Prince', 'diana@example.com', '2021-04-26');
```

(d) Orders

```
INSERT INTO Orders (OrderID, CustomerID, OrderDate, TotalAmount) VALUES
(1001, 1, '2023-04-26', 1245.50),
(1002, 2, '2023-10-12', 99.99),
(1003, 1, '2023-07-01', 145.00),
(1004, 3, '2023-01-14', 150.00),
(1005, 2, '2023-09-24', 120.00),
(1006, 1, '2023-06-19', 20.00);
```

Question 7: Generate a report showing CustomerName, Email, and the TotalNumberofOrders for each customer. Include customers who have not placed any orders, in which case their TotalNumberofOrders should be 0. Order the results by CustomerName.

Answer:

```
SELECT
   c.CustomerName,
   c.Email,
   COUNT(o.OrderID) AS TotalNumberOfOrders
FROM Customers c
LEFT JOIN Orders o ON c.CustomerID = o.CustomerID
GROUP BY c.CustomerName, c.Email
ORDER BY c.CustomerName;
```

Question 8 : Retrieve Product Information with Category: Write a SQL query to display the ProductName, Price, StockQuantity, and CategoryName for all products. Order the results by CategoryName and then ProductName alphabetically.

Answer:

```
SELECT
   p.ProductName,
   p.Price,
   p.StockQuantity,
   c.CategoryName
FROM Products p
```

```
INNER JOIN Categories c
   ON p.CategoryID = c.CategoryID
ORDER BY c.CategoryName, p.ProductName;
```

Question 9: Write a SQL query that uses a Common Table Expression (CTE) and a Window Function (specifically ROW_NUMBER() or RANK()) to display the CategoryName, ProductName, and Price for the top 2 most expensive products in each CategoryName.
Answer:

```
WITH RankedProducts AS (
   SELECT
      c.CategoryName,
      p.ProductName,
      p.Price,
      ROW_NUMBER() OVER (
         PARTITION BY c.CategoryName
         ORDER BY p.Price DESC
      ) AS RankNo
   FROM Products p
   INNER JOIN Categories c
      ON p.CategoryID = c.CategoryID
)
SELECT
   CategoryName,
   ProductName,
   Price
FROM RankedProducts
WHERE RankNo <= 2
ORDER BY CategoryName, Price DESC;
```

Question 10: You are hired as a data analyst by Sakila Video Rentals, a global movie rental company. The management team is looking to improve decision-making by analyzing existing customer, rental, and inventory data.
Using the Sakila database, answer the following business questions to support key strategic initiatives.
Tasks & Questions:
1. Identify the top 5 customers based on the total amount they've spent. Include customer name, email, and total amount spent.
Answer:

```
SELECT
   c.first_name AS FirstName,
   c.last_name AS LastName,
   c.email AS Email,
   SUM(p.amount) AS TotalAmountSpent
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id, c.first_name, c.last_name, c.email
ORDER BY TotalAmountSpent DESC
LIMIT 5;
```

2. Which 3 movie categories have the highest rental counts? Display the category name and number of times movies from that category were rented.
Answer:

```
SELECT
```

```
    cat.name AS CategoryName,
    COUNT(r.rental_id) AS RentalCount
FROM rental r
JOIN inventory i ON r.inventory_id = i.inventory_id
JOIN film_category fc ON i.film_id = fc.film_id
JOIN category cat ON fc.category_id = cat.category_id
GROUP BY cat.name
ORDER BY RentalCount DESC
LIMIT 3;
```

3. Calculate how many films are available at each store and how many of those have never been rented.
Answer:

```
SELECT
    s.store_id AS StoreID,
    COUNT(i.inventory_id) AS TotalFilms,
    SUM(CASE WHEN r.rental_id IS NULL THEN 1 ELSE 0 END) AS NeverRented
FROM store s
JOIN inventory i ON s.store_id = i.store_id
LEFT JOIN rental r ON i.inventory_id = r.inventory_id
GROUP BY s.store_id;
```

4. Show the total revenue per month for the year 2023 to analyze business seasonality.
Answer:

```
SELECT
    DATE_FORMAT(payment_date, '%Y-%m') AS Month,
    SUM(amount) AS TotalRevenue
FROM payment
WHERE YEAR(payment_date) = 2023
GROUP BY Month
ORDER BY Month;
```

5. Identify customers who have rented more than 10 times in the last 6 months.
Answer:

```
SELECT
    c.first_name AS FirstName,
    c.last_name AS LastName,
    COUNT(r.rental_id) AS TotalRentals
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
WHERE r.rental_date >= DATE_SUB(CURDATE(), INTERVAL 6 MONTH)
GROUP BY c.customer_id, c.first_name, c.last_name
HAVING TotalRentals > 10
ORDER BY TotalRentals DESC;
```