Question 1: Explain the fundamental differences between DDL, DML, and DQL commands in SQL. Provide one example for each type of command.

```
1️⃣DDL (Data Definition Language)
------------------------------------------------------------
→ Used to define and modify database structures (tables, schemas).
→ Examples: CREATE, ALTER, DROP, TRUNCATE
============================================================
*/
-- Example: Create a table (DDL)
CREATE TABLE employees (
emp_id INT PRIMARY KEY,
emp_name VARCHAR(50),
department VARCHAR(50),
salary DECIMAL(10, 2)
);
-- Example: Alter the table (add a new column)
ALTER TABLE employees ADD COLUMN hire_date DATE;

/*
============================================================
2️⃣DML (Data Manipulation Language)
------------------------------------------------------------
→ Used to insert, update, or delete records from a table.
→ Examples: INSERT, UPDATE, DELETE
============================================================
*/
-- Example: Insert data (DML)
INSERT INTO employees (emp_id, emp_name, department, salary, hire_date)
VALUES (1, 'John Doe', 'HR', 55000.00, '2023-06-15');
-- Example: Update data (DML)
UPDATE employees
SET salary = 60000.00

WHERE emp_id = 1;
-- Example: Delete data (DML)
DELETE FROM employees
WHERE emp_id = 1;

/*
============================================================
3️⃣DQL (Data Query Language)
------------------------------------------------------------
→ Used to retrieve data from the database.
```

→ Main command: SELECT
==========
*/
-- Example: Retrieve records (DQL)
SELECT emp_id, emp_name, department, salary
FROM employees
WHERE department = 'HR';

/*
==========
✅ Summary:

DDL → Defines database structure (CREATE, ALTER, DROP)
DML → Manipulates data inside tables (INSERT, UPDATE, DELETE)
DQL → Retrieves data using SELECT queries
==========

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.

==========
📘 What are SQL Constraints?

→ SQL constraints are rules applied to table columns to ensure
the accuracy, integrity, and reliability of data in a database.
→ They help enforce conditions automatically —
so invalid or inconsistent data cannot be entered.
==========
Common SQL Constraints:
1️⃣PRIMARY KEY
2️⃣FOREIGN KEY
3️⃣CHECK
==========
*/
-- ==========
--1️⃣PRIMARY KEY Constraint
-- ----------
-- Ensures that each record in a table is unique and not NULL.
-- Example Scenario: Every employee must have a unique ID.
-- ==========
CREATE TABLE employees (

```
emp_id INT PRIMARY KEY, -- Unique identifier
emp_name VARCHAR(50) NOT NULL,
department VARCHAR(50)
);
-- Trying to insert duplicate emp_id will cause an error
INSERT INTO employees VALUES (1, 'Alice', 'HR');
-- This will fail:
-- INSERT INTO employees VALUES (1, 'Bob', 'Finance');

-- ============================================================
-- 2 FOREIGN KEY Constraint
-- ------------------------------------------------------------
-- Maintains referential integrity between two related tables.
-- Example Scenario: Each employee belongs to a valid department.
-- ============================================================
CREATE TABLE departments (
dept_id INT PRIMARY KEY,
dept_name VARCHAR(50)
);

CREATE TABLE emp_details (
emp_id INT PRIMARY KEY,
emp_name VARCHAR(50),
dept_id INT,
FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
-- Insert department first
INSERT INTO departments VALUES (10, 'Finance'), (20, 'HR');
-- Insert valid employee (works fine)
INSERT INTO emp_details VALUES (1, 'John Doe', 10);
-- This will fail because dept_id 99 doesn't exist
-- INSERT INTO emp_details VALUES (2, 'Jane', 99);

-- ============================================================
-- 3 CHECK Constraint
-- ------------------------------------------------------------
-- Ensures that values meet specific conditions.
-- Example Scenario: Employee salary must be greater than 0.
-- ============================================================
CREATE TABLE payroll (
emp_id INT PRIMARY KEY,
emp_name VARCHAR(50),
salary DECIMAL(10, 2) CHECK (salary > 0)
);
```

```
-- Valid entry
INSERT INTO payroll VALUES (1, 'Ravi', 50000.00);
-- Invalid entry (will fail CHECK constraint)
-- INSERT INTO payroll VALUES (2, 'Sita', -1000.00);

-- ============================================================
-- ✅ Summary:
-- ------------------------------------------------------------
-- PRIMARY KEY → Ensures uniqueness of each record.
-- FOREIGN KEY → Ensures data consistency between related tables.
-- CHECK → Ensures values meet certain conditions.

-- ============================================================

/* ============================================================
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?
============================================================ */
/*
============================================================
📘 LIMIT and OFFSET Clauses
------------------------------------------------------------
→ Both clauses are used to control the number of records
returned by a SQL query — particularly useful for pagination.
============================================================
1️⃣ LIMIT → Specifies how many records to return.
2️⃣ OFFSET → Specifies how many records to skip before starting to return rows.
*/
-- Example table
CREATE TABLE products (
product_id INT PRIMARY KEY,
product_name VARCHAR(50),
price DECIMAL(10, 2)
);
-- Insert sample data (for demonstration)
INSERT INTO products VALUES
(1, 'Laptop', 65000.00),
(2, 'Mouse', 800.00),
(3, 'Keyboard', 1200.00),
(4, 'Monitor', 15000.00),
(5, 'Printer', 8500.00),
(6, 'Scanner', 9500.00),
```

```
(7, 'Webcam', 4000.00),
(8, 'Router', 2500.00),
(9, 'SSD', 6000.00),
(10, 'HDD', 3500.00),
(11, 'Smartphone', 20000.00),
(12, 'Tablet', 18000.00),
(13, 'Headphones', 2500.00),

(14, 'Speaker', 5000.00),
(15, 'Microphone', 3500.00),
(16, 'Projector', 30000.00),
(17, 'Charger', 900.00),
(18, 'Cable', 400.00),
(19, 'Adapter', 700.00),
(20, 'Camera', 28000.00),
(21, 'Smartwatch', 12000.00),
(22, 'Flash Drive', 1000.00),
(23, 'Memory Card', 1500.00),
(24, 'Cooling Pad', 1800.00),
(25, 'Tripod', 2200.00),
(26, 'TV', 45000.00),
(27, 'Fan', 1800.00),
(28, 'Lamp', 1200.00),
(29, 'VR Headset', 25000.00),
(30, 'Game Console', 40000.00);

/*
============================================================
🔍 Retrieving Data Using LIMIT and OFFSET
------------------------------------------------------------
Scenario: Each page shows 10 records.
We want to display the **third page** of results.
------------------------------------------------------------
Page 1 → OFFSET 0 LIMIT 10
Page 2 → OFFSET 10 LIMIT 10
Page 3 → OFFSET 20 LIMIT 10 ✅ (this one)
============================================================
*/
-- Retrieve the 3rd page of results (records 21–30)
SELECT *
FROM products
ORDER BY product_id
LIMIT 10 OFFSET 20;
-- ✅ Explanation:
```

-- LIMIT 10 → returns 10 rows per page
-- OFFSET 20 → skips the first 20 rows (2 pages × 10 rows)
-- So, this query returns the 21st to 30th records.

/*
==========
✅ Summary:

LIMIT → Controls how many rows to display.
OFFSET → Skips a specified number of rows.
Together, they are ideal for pagination (page navigation).
==========
*/

/* ==========
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.
========== */
/*
==========
📘 What is a CTE (Common Table Expression)?

→ A CTE is a **temporary, named result set** defined within
the execution scope of a single SQL query.
→ It is created using the `WITH` keyword and can be referenced
just like a regular table or view.
==========
🔹 Syntax:

WITH cte_name AS (
SELECT ...
)
SELECT * FROM cte_name;
==========
🔹 Benefits of Using CTEs:

1️⃣ Improves readability and organization of complex queries.
2️⃣ Allows reusing intermediate query results.
3️⃣ Helps avoid subquery repetition.
4️⃣ Enables recursive queries (like hierarchical data traversal).
==========

```
*/
-- Example: Create a sample table
CREATE TABLE sales (
sale_id INT PRIMARY KEY,
emp_name VARCHAR(50),
region VARCHAR(20),
total_sale DECIMAL(10,2)
);
-- Insert sample data
INSERT INTO sales VALUES
(1, 'Alice', 'North', 15000.00),
(2, 'Bob', 'South', 12000.00),
(3, 'Charlie', 'North', 18000.00),
(4, 'David', 'West', 10000.00),
(5, 'Eve', 'South', 22000.00);
-- ============================================================
-- Example: Using a CTE to calculate average sales by region
-- ============================================================
WITH regional_avg AS (
SELECT region, AVG(total_sale) AS avg_sales
FROM sales
GROUP BY region
)
SELECT s.emp_name, s.region, s.total_sale, r.avg_sales
FROM sales s
JOIN regional_avg r
ON s.region = r.region
WHERE s.total_sale > r.avg_sales;
-- ✅ Explanation:
--1️⃣The CTE `regional_avg` computes the average sale per region.
--2️⃣The main query compares each employee's sales with their regional average.
--3️⃣Only employees whose sales exceed the average are displayed.
-- ============================================================
-- ✅ Summary:
------------------------------------------------------------
CTEs help break complex SQL logic into readable, modular sections.
They improve query maintainability and enable advanced operations
like recursion, aggregation, and filtering on intermediate results.

============================================================
/* ============================================================
Question 5:
Describe the concept of SQL Normalization and its primary goals.
Briefly explain the first three normal forms (1NF, 2NF, 3NF).
```

========== */
/*
==========
📘 What is SQL Normalization?

→ Normalization is the process of organizing data in a database
to reduce redundancy (duplicate data) and improve data integrity.
→ It divides large tables into smaller, related tables and
establishes relationships using foreign keys.
==========
🎯 Primary Goals of Normalization:

1️⃣ Eliminate redundant data (avoid data duplication)
2️⃣ Ensure logical data storage (data dependency makes sense)
3️⃣ Improve data consistency and efficiency
==========
*/

/*
==========
1️⃣ FIRST NORMAL FORM (1NF)

✅ Rule:
- Each cell must contain **atomic (indivisible)** values.
- Each record must be unique.

❌ Example (Not in 1NF):
----
student_id	student_name	subjects
1	Alice	Math, Science
2	Bob	English, History

✅ Convert to 1NF:

----
student_id	student_name	subject
1	Alice	Math
1	Alice	Science
2	Bob	English
2	Bob	History
==========
*/

```
CREATE TABLE student_subjects_1NF (
student_id INT,
student_name VARCHAR(50),
subject VARCHAR(50)
);

/*
============================================================
2️⃣ SECOND NORMAL FORM (2NF)
------------------------------------------------------------
✅ Rule:
- Table must be in 1NF.
- All non-key columns must depend on the entire primary key
(no partial dependency).
------------------------------------------------------------
❌ Example (Not in 2NF):
------------------------------------------------------------
student_id | course_id | student_name | course_name
------------------------------------------------------------
✅ Convert to 2NF:
------------------------------------------------------------
-- Separate into two tables:
------------------------------------------------------------
Table: students
student_id | student_name
Table: courses
course_id | course_name
Table: enrollments
student_id | course_id
------------------------------------------------------------
*/

CREATE TABLE students (
student_id INT PRIMARY KEY,
student_name VARCHAR(50)
);
CREATE TABLE courses (
course_id INT PRIMARY KEY,
course_name VARCHAR(50)
);
CREATE TABLE enrollments (
student_id INT,
course_id INT,
FOREIGN KEY (student_id) REFERENCES students(student_id),
```

```
FOREIGN KEY (course_id) REFERENCES courses(course_id)
);

/*
============================================================
3️⃣ THIRD NORMAL FORM (3NF)
------------------------------------------------------------
✅ Rule:
- Table must be in 2NF.
- There must be **no transitive dependency**,
meaning non-key columns should not depend on other non-key columns.
------------------------------------------------------------
❌ Example (Not in 3NF):
------------------------------------------------------------
student_id | student_name | city | zipcode
------------------------------------------------------------
(city → zipcode) creates a transitive dependency.
------------------------------------------------------------
✅ Convert to 3NF:
------------------------------------------------------------
Table: students
student_id | student_name | city_id
Table: cities
city_id | city | zipcode
------------------------------------------------------------
*/

CREATE TABLE cities (
city_id INT PRIMARY KEY,
city VARCHAR(50),
zipcode VARCHAR(10)
);
ALTER TABLE students ADD COLUMN city_id INT;
ALTER TABLE students
ADD FOREIGN KEY (city_id) REFERENCES cities(city_id);

/*
============================================================
✅ Summary:
------------------------------------------------------------
1NF → Eliminate repeating groups and ensure atomicity.
2NF → Eliminate partial dependencies on composite keys.
3NF → Eliminate transitive dependencies between non-key attributes.
============================================================
```

*/

SQL ASSIGNMENT (ECommerceDB)

Assignment Code: DA-AG-014
Topic: Introduction to SQL and Advanced Functions | Assignment

```
-=====================================- ====
-- SQL ASSIGNMENT (ECommerceDB)
-- Assignment Code: DA-AG-014
-- Introduction to SQL and Advanced Functions | Assignment
-- ========================================
-- ========================================
-- QUESTION 6:
-- Create a database named ECommerceDB and perform the following tasks:
-- (a) Create tables
-- (b) Insert records
-- ========================================
CREATE DATABASE ECommerceDB;
USE ECommerceDB;
-- ---- Create Tables ----
CREATE TABLE Categories (
CategoryID INT PRIMARY KEY,
CategoryName VARCHAR(50) NOT NULL UNIQUE
);
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)
);
CREATE TABLE Customers (
CustomerID INT PRIMARY KEY,
CustomerName VARCHAR(100) NOT NULL,
Email VARCHAR(100) UNIQUE,
JoinDate DATE
);
CREATE TABLE Orders (
OrderID INT PRIMARY KEY,
CustomerID INT,
OrderDate DATE NOT NULL,
TotalAmount DECIMAL(10,2),
```

```
FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)
);
-- ---- Insert Data ----
INSERT INTO Categories VALUES
(1, 'Electronics'),
(2, 'Books'),
(3, 'Home Goods'),
(4, 'Apparel');
INSERT INTO Products 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);
INSERT INTO Customers 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');
INSERT INTO Orders 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 (including customers with 0 orders).
-- ========================================
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.CustomerID
ORDER BY c.CustomerName;
-- ========================================
```

```
-- QUESTION 8:
-- Retrieve Product Information with Category:
-- Display ProductName, Price, StockQuantity, and CategoryName
-- ========================================
SELECT
p.ProductName,
p.Price,
p.StockQuantity,
c.CategoryName
FROM Products p
JOIN Categories c ON p.CategoryID = c.CategoryID
ORDER BY c.CategoryName, p.ProductName;
-- ========================================
-- QUESTION 9:
-- Use a CTE and RANK() to display CategoryName, ProductName, and Price
-- for the top 2 most expensive products in each category.
-- ========================================
WITH RankedProducts AS (
SELECT
c.CategoryName,
p.ProductName,
p.Price,
RANK() OVER (PARTITION BY c.CategoryName ORDER BY p.Price DESC) AS rnk
FROM Products p
JOIN Categories c ON p.CategoryID = c.CategoryID
)
SELECT CategoryName, ProductName, Price
FROM RankedProducts
WHERE rnk <= 2;
-- ========================================
-- QUESTION 10:
-- Sakila Video Rentals Analysis
-- Database: sakila
-- ========================================
USE sakila;
-- 10.1 Top 5 customers based on total amount spent
SELECT
CONCAT(c.first_name, ' ', c.last_name) AS CustomerName,
c.email,
SUM(p.amount) AS TotalAmountSpent
FROM customer c
JOIN payment p ON c.customer_id = p.customer_id
GROUP BY c.customer_id
ORDER BY TotalAmountSpent DESC
```

```
LIMIT 5;
-- 10.2 Top 3 movie categories by rental count
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.category_id
ORDER BY RentalCount DESC
LIMIT 3;
-- 10.3 Films per store and never rented films
SELECT
s.store_id,
COUNT(DISTINCT i.film_id) AS TotalFilms,
COUNT(DISTINCT i.film_id) - COUNT(DISTINCT r.inventory_id) AS NeverRentedFilms
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;
-- 10.4 Monthly revenue for 2023
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;
-- 10.5 Customers with more than 10 rentals in last 6 months
SELECT
CONCAT(c.first_name, ' ', c.last_name) AS CustomerName,
c.email,
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
HAVING TotalRentals > 10
ORDER BY TotalRentals DESC;
```