ASSIGNMENT : Introduction to SQL and Advanced Functions | Assignment

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

ANSWER 1 : In SQL, commands are categorized based on their purpose in managing and manipulating data. The three key types are **DDL (Data Definition Language)**, **DML (Data Manipulation Language)**, and **DQL (Data Query Language)**.

---

1. DDL – Data Definition Language

Purpose:
DDL commands are used to define or modify the structure of database objects such as tables, schemas, or indexes.
These commands affect the *schema* or *structure* of the database rather than the data itself.

Examples of DDL Commands: `CREATE`, `ALTER`, `DROP`, `TRUNCATE`

Example:

```
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   Name VARCHAR(50),
   Age INT
);
```

✅ *This command creates a new table named `Students`.*

---

2. DML – Data Manipulation Language

Purpose:
DML commands are used to manage and manipulate the data stored in the database.
They affect the *data* inside the tables, not the structure.

Examples of DML Commands: `INSERT`, `UPDATE`, `DELETE`

Example:

```
INSERT INTO Students (StudentID, Name, Age)
VALUES (1, 'Krushna Mahajan', 21);
```

✅ *This command inserts a new record into the `Students` table.*

---

3. DQL – Data Query Language

Purpose:
DQL commands are used to retrieve data from the database.
They are focused on *querying* and *reading* data, not changing it.

Example of DQL Command: `SELECT`

Example:

SELECT Name, Age FROM Students WHERE Age > 18;

✅ *This command retrieves the names and ages of students older than 18.*

Summary Table

Category	Full Form	Purpose	Common Commands	Example
DDL	Data Definition Language	Defines/modifies structure	CREATE, ALTER, DROP	`CREATE TABLE ...`
DML	Data Manipulation Language	Manages data	INSERT, UPDATE, DELETE	`INSERT INTO ...`
DQL	Data Query Language	Retrieves data	SELECT	`SELECT * FROM ...`

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 2 : **Purpose of SQL Constraints:**
SQL constraints are rules applied to table columns to **ensure the accuracy, consistency, and**

reliability of data in a database. They help maintain data integrity by restricting the type of data that can be inserted, updated, or deleted.

---

1. PRIMARY KEY Constraint

Description:
Ensures that each record in a table is **unique** and **not null**. It uniquely identifies each row in a table.

Example Scenario:
In a `Students` table, the `student_id` column is set as the **PRIMARY KEY** so that no two students can have the same ID.

```
CREATE TABLE Students (

    student_id INT PRIMARY KEY,

    name VARCHAR(50),

    age INT

);
```

---

2. FOREIGN KEY Constraint

Description:
Establishes a **relationship between two tables** by enforcing that a value in one table matches a value in another (referenced) table's primary key.

Example Scenario:
In an `Enrollments` table, the `student_id` should correspond to a valid student in the `Students` table.

```
CREATE TABLE Enrollments (

    enrollment_id INT PRIMARY KEY,

    student_id INT,

    course_name VARCHAR(50),
```

FOREIGN KEY (student_id) REFERENCES Students(student_id)

);

---

3. CHECK Constraint

Description:
Ensures that all values in a column satisfy a specific **logical condition**.

Example Scenario:
In an `Employees` table, ensure that the `salary` is always positive.

```
CREATE TABLE Employees (
    emp_id INT PRIMARY KEY,
    name VARCHAR(50),
    salary DECIMAL(10,2) CHECK (salary > 0)
);
```

✅ Summary Table

Constraint Type	Purpose	Example Use
PRIMARY KEY	Uniquely identifies each record	`student_id` in Students table
FOREIGN KEY	Maintains referential integrity between tables	`student_id` in Enrollments refers to Students

CHECK	Validates data based on a condition	Ensures `salary > 0`

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 3 : In SQL, **`LIMIT`** and **`OFFSET`** are clauses used to control the number of rows returned by a query — especially useful for **pagination** (displaying results page by page).

🔹 Difference between `LIMIT` and `OFFSET`:

Clause	Purpose	Example
`LIMIT`	Specifies **how many records** to return.	`LIMIT 10` → return 10 rows
`OFFSET`	Specifies **how many records to skip** before starting to return rows.	`OFFSET 20` → skip first 20 rows

---

🔹 Using `LIMIT` and `OFFSET` Together

They're often combined for **pagination** — to fetch specific "pages" of results.

Example:
Suppose each page shows **10 records**, and you want to retrieve the **third page**.

- **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 employees

LIMIT 10 OFFSET 20;
```

🔹 Formula for Pagination

To get page `n` with `m` records per page:

```
LIMIT m OFFSET (n - 1) * m
```

For example:
Third page (`n=3`), 10 records per page (`m=10`):

```
LIMIT 10 OFFSET (3 - 1) * 10  →  LIMIT 10 OFFSET 20
```

In summary:

- `LIMIT` → how many rows to fetch
- `OFFSET` → how many rows to skip
- Used together for **pagination** in SQL queries.

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 4 : A **Common Table Expression (CTE)** in SQL is a **temporary result set** that you can reference within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement.
It's defined using the `WITH` keyword and makes complex queries **easier to read, maintain, and reuse**.

🧩 Key Benefits of CTEs

1. **Improves Readability:** Breaks down complex queries into logical, easy-to-understand parts.
2. **Reusability:** The same CTE can be referenced multiple times within the main query.
3. **Recursion Support:** Enables writing recursive queries (e.g., hierarchical data like employee-manager structures).
4. **Simplifies Maintenance:** Easier to debug and modify than nested subqueries.

✅ Example: Using a CTE

Suppose we have a table `Employees`:

emp_id	name	department	salary
1	John	HR	50000
2	Alice	IT	60000

3	Bob	IT	70000
4	Carol	HR	55000

We want to find the employees whose salary is **above the average salary** of their department.

```
-- Define a CTE to calculate average salary by department
WITH DeptAvg AS (
   SELECT department, AVG(salary) AS avg_salary
   FROM Employees
   GROUP BY department
)
-- Use the CTE in the main query
SELECT e.name, e.department, e.salary
FROM Employees e
JOIN DeptAvg d
ON e.department = d.department
WHERE e.salary > d.avg_salary;
```

🧠 Explanation

- The CTE `DeptAvg` first computes the average salary for each department.
- The main query then joins this result with the `Employees` table to filter employees earning above their department's average.

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

ANSWER 5 : **SQL Normalization** is the process of organizing data in a database to reduce redundancy (duplicate data) and improve data integrity. It involves dividing large tables into smaller, related tables and defining relationships between them.

🎯 Primary Goals of Normalization

1. **Eliminate redundant data** – Avoid storing the same data in multiple places.
2. **Ensure data integrity** – Maintain data consistency and accuracy.
3. **Simplify data maintenance** – Make updates, inserts, and deletions more efficient and less error-prone.
4. **Improve query efficiency** – Streamline data retrieval by organizing it logically.

🧩 Normal Forms

1. First Normal Form (1NF)

- Ensures that each column contains **atomic (indivisible)** values.
- Each record (row) must be **unique**.
- No **repeating groups** or **arrays** allowed.

Example:

StudentID	Subjects
1	Math, Science

➡ **Not in 1NF** (multiple values in one column).
After conversion to **1NF**:

StudentID	Subject
1	Math
1	Science

---

2. Second Normal Form (2NF)

- Must first satisfy **1NF**.
- All **non-key attributes** must depend **entirely** on the **primary key** (no partial dependency).
- Applies mainly to tables with **composite primary keys**.

Example:

StudentID	CourseID	StudentName	CourseName

Here, `StudentName` depends only on `StudentID`, not on both keys.

➡ To achieve **2NF**, split into:
Students Table: (StudentID, StudentName)
Courses Table: (CourseID, CourseName)
Enrollment Table: (StudentID, CourseID)

---

3. Third Normal Form (3NF)

- Must satisfy **2NF**.
- No **transitive dependency** (non-key attribute depends on another non-key attribute).
- Every non-key attribute should depend **only on the primary key**.

Example:

StudentID	DepartmentID	DepartmentName

Here, `DepartmentName` depends on `DepartmentID`, not directly on `StudentID`.

➡ To achieve **3NF**, split into:
Students Table: (StudentID, DepartmentID)
Departments Table: (DepartmentID, DepartmentName)

✅ **Summary Table**

Normal Form	Key Requirement	Eliminates
1NF	Atomic values	Repeating groups
2NF	Full dependency on primary key	Partial dependency
3NF	No transitive dependency	Transitive dependency

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))

2. Insert the following records into each table

● Categories

CategoryID Category Name

1 Electronics

2 Books

3 Home Goods

4 Apparel

● Products

ProductID ProductName CategoryID Price StockQuantity

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

● Customers

CustomerID CustomerName Email Joining Date

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

- Orders

OrderID CustomerID OrderDate TotalAmount

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

ANSWER : 6 Here's the **complete SQL script** to create and populate the `ECommerceDB` database with the required tables, constraints, and records 👇

🧱 Step 1: Create Database

CREATE DATABASE ECommerceDB;

USE ECommerceDB;

🗂️ Step 2: Create Tables

1. Categories Table

CREATE TABLE Categories (

```sql
    CategoryID INT PRIMARY KEY,
    CategoryName VARCHAR(50) NOT NULL UNIQUE
);
```

2. Products Table

```sql
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)
);
```

3. Customers Table

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

4. Orders Table

```sql
CREATE TABLE Orders (
```

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

🧩 Step 3: Insert Records

1. Insert into `Categories`

```sql
INSERT INTO Categories (CategoryID, CategoryName) VALUES
(1, 'Electronics'),
(2, 'Books'),
(3, 'Home Goods'),
(4, 'Apparel');
```

2. Insert into `Products`

```sql
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);

3. Insert into `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');

4. Insert into `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);

✅ Step 4: Verify Data

You can confirm the insertions with:

SELECT * FROM Categories;

SELECT * FROM Products;

SELECT * FROM Customers;

SELECT * FROM Orders;

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 7 : Here’s the SQL query that generates the required report 👇

```
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;
```

🔍 Explanation:

- **LEFT JOIN**: Ensures all customers are included — even those with no orders.
- **COUNT(o.OrderID)**: Counts the number of orders per customer. For customers without orders, the count returns `0`.
- **GROUP BY**: Groups the result by each unique customer.
- **ORDER BY CustomerName**: Sorts the results alphabetically by customer name.

✅ **Result Example:**

CustomerName	Email	TotalNumberOfOrders
Alice Brown	alice@mail.com	3
Bob Martin	bob@mail.com	0
Charlie Singh	charlie@mail.com	1

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 8 : Here's the SQL query to retrieve the product information along with its category:

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

FROM

    Products p

JOIN

    Categories c

ON

    p.CategoryID = c.CategoryID

ORDER BY

    c.CategoryName ASC,

    p.ProductName ASC;

✅ Explanation:

- **JOIN** connects `Products` with `Categories` using the `CategoryID` foreign key.
- **SELECT** chooses the required columns:
  - `ProductName`, `Price`, `StockQuantity` (from `Products`)
  - `CategoryName` (from `Categories`)
- **ORDER BY** ensures sorting first by `CategoryName` and then alphabetically by `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 9 : Here's a SQL query that uses a **Common Table Expression (CTE)** and a **window function (`ROW_NUMBER()`)** to display the top 2 most expensive products in each category:

WITH RankedProducts AS (

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

🔍 Explanation:

- **CTE (`RankedProducts`)**:
  Joins the `Products` and `Categories` tables to get `CategoryName`, `ProductName`, and `Price`.

- **`ROW_NUMBER()` window function**:
 Assigns a unique ranking within each category (`PARTITION BY c.CategoryName`) based on product price in descending order (`ORDER BY p.Price DESC`).
- **Filtering (`WHERE RankInCategory <= 2`)**:
 Selects only the **top 2 most expensive products** per category.
- **Final ordering**:
 Displays results sorted by `CategoryName` and then `Price` descending.

✅ **Alternative using `RANK()`** (if you want to handle ties):

```
WITH RankedProducts AS (
    SELECT
        c.CategoryName,
        p.ProductName,
        p.Price,
        RANK() OVER (
            PARTITION BY c.CategoryName
            ORDER BY p.Price DESC
        ) AS RankInCategory
    FROM Products p
    INNER JOIN Categories c
        ON p.CategoryID = c.CategoryID
)
SELECT
    CategoryName,
    ProductName,
```

```
    Price
FROM RankedProducts
WHERE RankInCategory <= 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. 2. Which 3 movie categories have the highest rental counts? Display the category name and number of times movies from that category were rented. 3. Calculate how many films are available at each store and how many of those have never been rented. 4. Show the total revenue per month for the year 2023 to analyze business seasonality. 5. Identify customers who have rented more than 10 times in the last 6 months.

ANSWER 10 : Got it! I'll provide **SQL queries** for each of the business questions based on the **Sakila database schema**. I'll assume standard Sakila table relationships.

---

1. Top 5 customers by total amount spent

```
SELECT
    c.first_name || ' ' || c.last_name AS customer_name,
    c.email,
    SUM(p.amount) AS total_spent
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 total_spent DESC
LIMIT 5;
```

Explanation:

- `payment.amount` gives the amount each customer paid.
- Summing by customer gives total spent.
- Sorted descending to get top 5 spenders.

---

2. Top 3 movie categories by rental counts

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

Explanation:

- Links rentals to films via inventory.
- Then links films to categories.
- Counts total rentals per category.

3. Films available per store & number never rented

```
-- Films available per store

SELECT

    s.store_id,

    COUNT(i.inventory_id) AS total_films,

    SUM(CASE WHEN r.rental_id IS NULL THEN 1 ELSE 0 END) AS never_rented

FROM inventory i

JOIN store s ON i.store_id = s.store_id

LEFT JOIN rental r ON i.inventory_id = r.inventory_id

GROUP BY s.store_id;
```

Explanation:

- Counts total inventory per store.
- Uses `LEFT JOIN` to count inventory items with no rentals (`rental_id IS NULL`).

---

4. Total revenue per month for 2023

```
SELECT

    TO_CHAR(p.payment_date, 'YYYY-MM') AS month,

    SUM(p.amount) AS total_revenue

FROM payment p

WHERE EXTRACT(YEAR FROM p.payment_date) = 2023

GROUP BY TO_CHAR(p.payment_date, 'YYYY-MM')
```

ORDER BY month;

Explanation:

- Filters payments in 2023.
- Groups by month and sums revenue.
- Useful for seasonal analysis.

(Depending on your SQL dialect, `TO_CHAR` might be replaced with `DATE_FORMAT` in MySQL.)

---

5. Customers who rented more than 10 times in the last 6 months

```
SELECT
   c.customer_id,
   c.first_name || ' ' || c.last_name AS customer_name,
   c.email,
   COUNT(r.rental_id) AS rentals_last_6_months
FROM customer c
JOIN rental r ON c.customer_id = r.customer_id
WHERE r.rental_date >= CURRENT_DATE - INTERVAL '6 months'
GROUP BY c.customer_id, c.first_name, c.last_name, c.email
HAVING COUNT(r.rental_id) > 10
ORDER BY rentals_last_6_months DESC;
```

Explanation:

- Filters rentals in the last 6 months.

- Counts rentals per customer.
- Only includes customers with more than 10 rentals.
