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.

Ans. DDL, DML, and DQL are distinct categories of SQL commands serving different purposes in database management.

1. DDL (Data Definition Language):

- **Purpose:** Used to define, modify, or delete the structure of database objects (tables, indexes, views, etc.). It deals with the schema of the database.
- **Impact:** Affects the database's structure, not the data itself. DDL operations are typically irreversible without manual reconstruction.
- **Examples of DDL Commands:** `CREATE`, `ALTER`, `DROP`, `TRUNCATE`

2. DML (Data Manipulation Language):

- **Purpose:** Used to manipulate data within the database objects defined by DDL. It focuses on inserting, updating, and deleting records.
- **Impact:** Affects the data stored within the tables, not the table structure. DML operations are typically transactional and can be rolled back**.**
- **Examples of DML Commands:** `INSERT`, `UPDATE`, `DELETE`

3. DQL (Data Query Language):

- **Purpose:** Used to retrieve data from the database. It allows users to query and extract information based on specified criteria.
- **Impact:** Does not modify the database structure or the data itself; it only retrieves a subset of data for viewing.
- **Example:**
- **SELECT FirstName, LastName**

  FROM Students

  WHERE Age > 18;

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.

Ans. SQL constraints are rules enforced on the data columns of a table to limit the type of data that can be entered. Their purpose is to ensure the accuracy, reliability, and integrity of the data within a database. If a data manipulation operation (e.g., insert, update) violates a defined constraint, the operation is aborted, preventing invalid or inconsistent data from being stored.

Here are three common types of constraints:

1. **NOT NULL Constraint:**

- **Description:** This constraint ensures that a column cannot contain `NULL` values. Every row in the table must have a value for the column with this constraint.
- **Scenario:** In a `Customers` table, the `FirstName` and `LastName` columns should always have a value. Applying a `NOT NULL` constraint to these columns would prevent entries where a customer's name is missing.
- CREATE TABLE Customers (
  CustomerID INT PRIMARY KEY,
  FirstName VARCHAR(50) NOT NULL,
  LastName VARCHAR(50) NOT NULL,
  Email VARCHAR(100)
);

2. **UNIQUE Constraint:**

- **Description:** This constraint ensures that all values in a specified column (or set of columns) are distinct. No two rows can have the same value in a `UNIQUE` column, though `NULL` values are typically allowed (and are considered distinct from other `NULL` values).
- **Scenario:** In a `Products` table, the `SKU` (Stock Keeping Unit) column should uniquely identify each product. A `UNIQUE` constraint on the `SKU` column would prevent the accidental entry of duplicate product SKUs.
- CREATE TABLE Products (
  ProductID INT PRIMARY KEY,
  ProductName VARCHAR(100),
  SKU VARCHAR(20) UNIQUE,
  Price DECIMAL(10, 2)

);

3. **PRIMARY KEY Constraint:**

- **Description:** A `PRIMARY KEY` is a special type of `UNIQUE` and `NOT NULL` constraint. It uniquely identifies each record in a table. A table can have only one `PRIMARY KEY`, which can consist of one or more columns.
- **Scenario:** In an `Orders` table, the `OrderID` column serves as the unique identifier for each order. Designating `OrderID` as the `PRIMARY KEY` ensures that each order has a distinct and non-null identifier, facilitating efficient data retrieval and relationships with other tables.
- ```
  CREATE TABLE Orders (
      OrderID INT PRIMARY KEY,
      OrderDate DATE,
      CustomerID INT,
      TotalAmount DECIMAL(10, 2)
  );
  ```

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?

Ans. SQL provides the **LIMIT** and **OFFSET** clauses to control the number of rows returned in a query, especially when working with large datasets or implementing pagination features.

1. LIMIT Clause (What it does)

Definition:

The **LIMIT** clause is used to **restrict the number of rows** returned by a SQL query.

Purpose:

- Helps retrieve only a portion of records instead of the entire table
- Improves query performance
- Used commonly for pagination, sampling, and previews

Example:

```
SELECT * FROM Employees
LIMIT 5;
```
```

This returns **only the first 5 rows** from the Employees table.

2. OFFSET Clause (What it does)

Definition:

The **OFFSET** clause tells SQL to **skip a specific number of rows** before it begins to return results.

Purpose:

- Useful when browsing results page by page
- Allows you to skip previously viewed rows
- Often combined with LIMIT
- Prevents fetching all data at once

Example:

```
SELECT * FROM Employees
OFFSET 10;
```

This skips the **first 10 rows** and begins returning data from the 11th row onward.

3. Using LIMIT and OFFSET Together

When combined, **OFFSET determines where the results start**, and **LIMIT determines how many results to fetch**.

Why combine them?

To create **pagination**, such as:

- Page 1 → Records 1–10
- Page 2 → Records 11–20
- Page 3 → Records 21–30

Retrieving the Third Page of Results (10 records per page)

Given:

- Page number = **3**
- Records per page = **10**

Formula for OFFSET:

OFFSET=(Page Number−1)×Records Per Page\text{OFFSET} = (\text{Page Number} - 1) \times \text{Records Per Page}OFFSET=(Page Number−1)×Records Per Page

Apply the values:

OFFSET=(3−1)×10=20\text{OFFSET} = (3 - 1) \times 10 = 20OFFSET=(3−1)×10=20

This means you skip the **first 20 records** (records 1–20).

LIMIT Value:

LIMIT=10\text{LIMIT} = 10LIMIT=10

You want exactly 10 records on the page.

Final SQL Query:

```
SELECT *

FROM table_name

LIMIT 10

OFFSET 20;
```

Explanation:

- **OFFSET 20** → Skip the first 20 records
- **LIMIT 10** → Return the next 10 records

This gives you **records 21 to 30**, which correspond to the **3rd page**.

Summary

- **LIMIT** controls **how many** rows to retrieve.
- **OFFSET** controls **from where** to start retrieval.
- To fetch the 3rd page with 10 records per page:
    - **OFFSET = 20**, **LIMIT = 10**
    - SQL: `SELECT * FROM table_name LIMIT 10 OFFSET 20;`

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.

Ans. Common Table Expression (CTE) in SQL is a temporary, named result set that you can reference within a single `SELECT`, `INSERT`, `UPDATE`, `MERGE`, or `DELETE` statement. It is defined using the `WITH` clause and provides a way to break down complex queries into more readable and manageable components.

Main Benefits of CTEs:

- **Improved Readability and Organization:** CTEs allow you to structure complex queries logically, making them easier to understand and maintain, especially when dealing with multiple subqueries or intricate logic.
- **Reusability:** You can define a CTE once and reference it multiple times within the same query, avoiding redundant code and ensuring consistency.
- **Simplification of Complex Logic:** CTEs can simplify complex calculations or data transformations by breaking them into smaller, more manageable steps.
- **Recursive Queries:** CTEs are essential for creating recursive queries, which are used to traverse hierarchical data structures like organizational charts or bill of materials.
- **Alternative to Derived Tables:** CTEs offer a more structured and often more readable alternative to deeply nested derived tables (subqueries in the `FROM` clause)

Simple SQL Example of a CTE

Example: Retrieve employees with salary above the average salary

```
WITH AvgSalary AS (
    SELECT AVG(Salary) AS AverageSal
```

```
    FROM Employees
)
SELECT Name, Salary
FROM Employees, AvgSalary
WHERE Employees.Salary > AvgSalary.AverageSal;
```

Explanation:

- The CTE `AvgSalary` calculates the **average salary** of all employees.
- The main query then selects only those employees whose salary is **higher than the average**.

Summary

- A **CTE** is a named temporary result set created using the **WITH** clause.
- It improves **readability**, allows **reuse**, and supports **recursive queries**.
- It makes complex queries modular and easier to maintain.

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

Ans. SQL Normalization is a systematic process of organizing the columns and tables of a relational database to minimize data redundancy and improve data integrity.

Primary Goals of Normalization:

- **Reduce Data Redundancy:** Eliminate duplicate data storage, saving space and preventing inconsistencies.
- **Improve Data Integrity:** Ensure data accuracy and consistency by storing data in a single, authoritative location.
- **Prevent Anomalies:** Avoid insertion, update, and deletion anomalies that can occur in unnormalized databases.

- **Enhance Database Flexibility and Scalability:** Make the database easier to modify, extend, and adapt to changing requirements.

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

1. **First Normal Form (1NF):**
    - Requires that each column in a table contains only atomic (single) values. There should be no repeating groups of columns.
    - Each row must be unique and identifiable by a primary key.

StudentID	Name	PhoneNumber
101	Aisha	99999
101	Aisha	88888

    - Now each cell contains a single (atomic) value.
2. **Second Normal Form (2NF):**
    - Must satisfy the requirements of 1NF.
    - All non-key attributes must be fully functionally dependent on the entire primary key. This means no non-key attribute should depend only on a part of a composite primary key.
    - Example of NOT 2NF:
    - Suppose a table has a composite key: (StudentID, CourseID)

StudentID	CourseID	CourseName
101	C01	Python

Here, **CourseName depends only on CourseID**, not on the full key (StudentID + CourseID). This causes **partial dependency**, violating 2NF.

2NF Corrected:

- **StudentsCourses** (StudentID, CourseID)
- **Courses** (CourseID, CourseName)

Now CourseName depends only on CourseID, which is correct.

3. **Third Normal Form (3NF):**
    - Must satisfy the requirements of 2NF.
    - Eliminates transitive dependencies, meaning no non-key attribute should be functionally dependent on another non-key attribute. In other words, every non-key attribute must directly depend on the primary key and nothing else.
    - Example of NOT 3NF:

StudentID	StudentName	City	Pincode
101	Aisha	Delhi	110001

Here, City → Pincode (Pincode depends on City, not directly on StudentID).
This is a **transitive dependency**, violating 3NF.

3NF Corrected:

- **Students** (StudentID, StudentName, City)
- **CityInfo** (City, Pincode)

Now, each non-key attribute depends only on the primary key.