Answer 01:

DDL – *Data Definition Language*

What it does:
These commands are used to **define or change the structure** of your database — like creating new tables, changing columns, or deleting tables.
Think of DDL as the “architect” of the database — it decides *how things are built*.

Example:

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

Result: This creates a new table called `Students`.
See the SQL Script on GitHub: SQL Assignment01.sql

2. DML – *Data Manipulation Language*

What it does:
These commands are used to **work with the data inside** the tables — adding, changing, or deleting rows.
Think of DML as the “worker” — it fills and updates the database.

Example:

```
INSERT INTO Students (StudentID, Name, Age)
VALUES (1, 'Alice', 20);
```

Result: This adds a new student record to the `Students` table.
See the SQL Script on GitHub: SQL Assignment01.sql

3. DQL – *Data Query Language*

What it does:
These commands are used to **get information** from the database — basically asking questions and reading data.
Think of DQL as the “detective” — it finds the information you need.

Example:

SELECT Name, Age FROM Students;

Result: This retrieves the names and ages of all students.
See the SQL Script on GitHub: SQL Assignment01.sql

Summary Table

Type	Full Form	Main Purpose	Example
DDL	Data Definition Language	Defines structure (tables, columns)	`CREATE TABLE`
DML	Data Manipulation Language	Changes data (insert, update, delete)	`INSERT INTO`
DQL	Data Query Language	Reads data (queries)	`SELECT`

In short:

- **DDL = Build it**
- **DML = Fill it**
- **DQL = Read it**

Answer 02:

Purpose of SQL Constraints

Constraints are rules you set on a database table to make sure the data stays **accurate, reliable, and meaningful.**

They help prevent mistakes — like duplicate IDs, missing names, or invalid references between tables.

Think of constraints as safety checks that protect your data.

Three Common Types of Constraints

1. PRIMARY KEY

What it does:	Ensures that each row in a table can be uniquely identified. It means the column must be unique (no duplicates) and not null (can't be empty).
Example Scenario:	In both your Students and Courses tables, every record needs a unique ID.

Query Example:

```
CREATE TABLE Students (
    StudentID INT PRIMARY KEY, -- The ID column cannot be duplicated or left empty.
    Name VARCHAR(50),
    Age INT
);
```

Result: This makes sure no two students have the same `StudentID`.
See the SQL Script on GitHub: SQL Assignment02.sql

2. FOREIGN KEY

What it does:	Ensures that a value in one table matches an existing value in another table. It's used to keep relationships between tables consistent (Referential Integrity).
Example Scenario:	Your Enrollment table needs to ensure that any StudentID or CourseID it uses actually exists in the Students and Courses tables, respectively.

Query Example:

```
CREATE TABLE Enrollment (
    EnrollmentID INT PRIMARY KEY,
    StudentID INT NOT NULL,
    CourseID INT NOT NULL,
    EnrollmentDate DATE,
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID), -- Links to the Students table
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)     -- Links to the Courses table
);
```

Result: This makes sure you can't enroll a student (or enroll them in a course) that doesn't exist.
See the SQL Script on GitHub: SQL Assignment02.sql

3. UNIQUE

What it does:	Ensures that all values in a column (or a set of columns) are different. Unlike a Primary Key, a table can have multiple UNIQUE constraints and the column can sometimes accept a single NULL value.
Example Scenario:	In your enrollment system, a student should only be able to enroll in the same course once.

Query Example:

```
CREATE TABLE Enrollment (
   EnrollmentID INT PRIMARY KEY,
   StudentID INT NOT NULL,
   CourseID INT NOT NULL,
   -- ... other columns …
   UNIQUE (StudentID, CourseID) -- The combination of student and course must be unique
);
```

Result: This is crucial for preventing redundant or impossible data. If you try to insert a second row with the combination **(StudentID 1, CourseID 101)**, the database will stop the operation.
See the SQL Script on GitHub: SQL Assignment02.sql

Quick Summary

Constraint	Purpose	Example Rule	Your Query Example
PRIMARY KEY	Makes each record uniquely identifiable	ID cannot be duplicated or missing	StudentID INT PRIMARY KEY
FOREIGN KEY	Keeps table relationships valid	Linking record must exist in the parent table	FOREIGN KEY (StudentID) REFERENCES Students(StudentID)

UNIQUE	Prevents duplication of specific data points	The combination of data must be one-of-a-kind	UNIQUE (StudentID, CourseID)

In short:

SQL constraints are like **rules of the road** for your database — they keep the data clean, consistent, and meaningful. By using these constraints (as demonstrated in your queries), you build a robust and reliable database structure.

Answer 03:

Difference Between LIMIT and OFFSET Clauses in SQL

Both `LIMIT` and `OFFSET` clauses are used for controlling the result set returned by a `SELECT` query, typically for pagination or sampling.

1. LIMIT Clause

- **Purpose:** Specifies the **maximum number of rows** the query should return.
- **Analogy:** If a full phone book has 1,000 entries, using `LIMIT 10` means you only get the first 10 entries.
- **Usage:** It is mandatory for pagination, as it defines the "page size."

2. OFFSET Clause

- **Purpose:** Specifies the **number of rows to skip** before starting to return the result rows.
- **Analogy:** If you use `OFFSET 20`, the database skips the first 20 entries in the sorted phone book and begins returning results from the 21st entry onward.
- **Requirement:** `OFFSET` is almost always used in conjunction with `ORDER BY` to ensure the skipped records are consistent, otherwise the skipped rows could change between queries.

Using LIMIT and OFFSET for Pagination

To implement pagination (showing results page by page), you use both clauses together.

The general formula for retrieving results for **Page N** with a size of **P** records per page is:

1. **LIMIT (P):** Set the page size.
2. **OFFSET ((N - 1) * P):** Calculate how many rows to skip based on the previous pages.

Scenario: Retrieving the Third Page (Page 3)

- **Page Number (N):** 3
- **Records per Page (P):** 10

Calculation:

- **Records to Skip (OFFSET):** `(3 - 1) * 10 = 20`
- **Records to Return (LIMIT):** `10`

Conclusion: To get the third page of 10 records, you must skip the first 20 records and then take the next 10.

SQL Example

You could use this logic to retrieve the third page of your top customers report, assuming the results are ordered consistently:

See the SQL Script on GitHub: SQL Assignment03.sql

Answer 04:

A Common Table Expression (CTE) is a powerful, temporary, named result set defined within the execution scope of a single SQL statement (such as a `SELECT`, `INSERT`, `UPDATE`, or `DELETE`).

Think of a CTE as a **temporary, virtual table** that only exists for the duration of the query it is a part of. It is defined using the `WITH` clause.

What is a Common Table Expression (CTE)?

A CTE is essentially a way to break down complex queries into simpler, more readable, and manageable parts. It acts as a subquery that is factored out and named at the beginning of the main statement.

Structure:

The basic structure of a CTE is:

```
WITH CTE_Name (Column1, Column2, ...) AS (
    -- The SELECT statement that defines the result set of the CTE
    SELECT …
```

```
    FROM …
    WHERE …
)


-- The main query that references the CTE_Name
SELECT …
FROM CTE_Name
WHERE ...;
```

Main Benefits of Using CTEs

CTEs are highly valued for improving query maintenance and performance in complex scenarios:

1. Readability and Maintainability (Simplicity)

- **Breaks Down Complexity:** Instead of nesting multiple subqueries (which can make a query very difficult to read from the inside out), a CTE allows you to define the intermediate steps clearly at the top. This makes the overall logic easier to follow.
- **Self-Documenting:** By giving the intermediate result set a meaningful name (e.g., `StudentsWithHighCredits`), you clearly signal the purpose of that data to anyone reading the query.

2. Recursive Queries

- CTEs are the standard and most efficient way to perform **recursive queries** in SQL.
- **Scenario:** This is essential for querying hierarchical or tree-like data, such as organizational charts (who reports to whom), bill-of-materials structures (what parts are needed to build a component), or navigating paths in a graph.

3. Non-Repetitive Code (DRY Principle)

- **Reusability within the Query:** A CTE can be referenced multiple times within the final main query. This avoids repeating the same lengthy subquery definition multiple times, simplifying the code and ensuring that any changes to the logic only need to be made in one place.

4. Improved Performance (Database Dependent)

- While not always guaranteed, many modern database engines (like SQL Server, PostgreSQL, and MySQL) can often optimize queries using CTEs better than complex subqueries, particularly when the same CTE is referenced multiple times.

Simple SQL Example Demonstrating CTE Usage

Let's use your `Students` and `Enrollment` tables to find the average age of students who are enrolled in **more than one course**.

This requires two steps: identifying the busy students, and then calculating their average age. A CTE is perfect for the first step.

Scenario: Calculate the average age of all students enrolled in at least two courses.

```
-- 1. Define the CTE named 'BusyStudents'

WITH BusyStudents AS (
    -- This query identifies StudentIDs who are enrolled in > 1 course
    SELECT
        StudentID
    FROM
        Enrollment
    GROUP BY
        StudentID
    HAVING
        COUNT(CourseID) > 1
)

-- 2. Use the CTE in the final query

SELECT
    AVG(S.Age) AS AverageAgeOfBusyStudents
FROM
    Students S
INNER JOIN
    BusyStudents B ON S.StudentID = B.StudentID;
```

Breakdown of the Example:

1. **`WITH BusyStudents AS (...)`**: This defines the temporary result set. We've named it logically (`BusyStudents`).
2. **Inner Query (The CTE):** It looks at the `Enrollment` table, groups the records by `StudentID`, and filters for only those students who have a count of courses greater than 1. This CTE produces a simple list of `StudentID` values (e.g., 1, 12, 15).
3. **Outer Query:** It treats `BusyStudents` just like a regular table, joining it to the `Students` table to retrieve the `Age` data, and then calculates the average age.

Answer 05:

The Concept of SQL Normalization

Normalization is a formal process, based on mathematical theory, used to test and refine a relational database schema (its structure) to ensure the tables are structurally sound.

Primary Goals of Normalization

The process of normalization aims to achieve two main goals:

1. **Minimize Data Redundancy (Duplication):**
    - **Goal:** Ensure that any piece of information is stored in only one place.
    - **Benefit:** Reduces the amount of storage space needed and, more importantly, prevents the same piece of information from being updated differently in multiple places, which leads to inconsistent data.
2. **Eliminate Anomalies (Data Inconsistencies):** Normalization eliminates three types of issues that arise from redundant data:
    - **Insertion Anomaly:** Being unable to add a new record to the table unless you have complete information for all columns (e.g., you can't add a new course until a student enrolls).
    - **Update Anomaly:** Having to update the same piece of information in multiple rows (e.g., if a professor changes their office number, you must update every course they teach).
    - **Deletion Anomaly:** Deleting a record causing the unintentional loss of unrelated data (e.g., deleting the last student in a course also deletes the course's only record).

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

Normalization is achieved through a progression of rules, or "Normal Forms." A table must satisfy the rules of the previous normal form before it can proceed to the next.

1. First Normal Form (1NF)

Rule: A table is in 1NF if and only if all attribute values (columns) are **atomic** (indivisible) and there are **no repeating groups** of columns.

- **In simple terms:**
    - Each cell in the table must contain a single value.
    - You cannot have multiple values stored in one column (e.g., a "PhoneNumbers" column containing "555-1212, 555-1213").

    - You cannot have repeating columns (e.g., `Course1`, `Course2`, `Course3`).

2. Second Normal Form (2NF)

Rule: A table is in 2NF if and only if it is in **1NF** and **all non-key attributes** (non-primary key columns) are **fully dependent on the entire Primary Key**.

This rule applies specifically to tables with a **composite primary key** (a key made up of two or more columns).

- **In simple terms:** If your primary key is (StudentID, CourseID), you cannot have a column like `StudentName` dependent only on `StudentID`. If it's not fully dependent on the *entire* key, you must move it to a new table.

3. Third Normal Form (3NF)

Rule: A table is in 3NF if and only if it is in **2NF** and **no non-key attribute** is **transitively dependent** on the Primary Key.

- **In simple terms:** Once you have the primary key, you should not be able to determine the value of a non-key column through **another** non-key column.

Answer 06:

See the Answer on GitHub: SQL Assignment_Answer06.sql

Answer 07:

See the Answer on GitHub: SQL Assignment_Answer07.sql

Answer 08:

See the Answer on GitHub: SQL Assignment_Answer08.sql

Answer 09:

See the Answer on GitHub: SQL Assignment_Answer09.sql

Answer 10:

See the Answer on GitHub: SQL Assignment_Answer10.sql