Module 4: Structured Query Language (SQL) - Part 1

Welcome to Module 4, where we begin our journey into **Structured Query Language (SQL)**. If the previous modules provided you with the theoretical understanding of how databases are designed (ER Model) and organized (Relational Model), SQL is where theory meets practice. SQL is the universal language that almost all relational database systems understand. It's the tool you'll use to create your database structure, add data to it, retrieve specific information, update records, and even delete them.

In this first part of our SQL exploration, we will focus on the fundamental commands that allow you to define the very shape of your database (**Data Definition Language - DDL**) and manipulate the data stored within it (**Data Manipulation Language - DML**). By the end of this module, you will be able to construct basic SQL statements to build tables, enforce rules, insert new data, query for information, and modify existing records.

Chapter 4: SQL Fundamentals (DDL & DML)

4.1 Introduction to SQL: History and Standards

What is SQL?

Structured Query Language (SQL) is the standard language used to communicate with and manage data in relational database management systems (RDBMS). It is a powerful, high-level language that allows users to interact with databases by telling the system *what* they want to achieve, rather than *how* to achieve it. This makes SQL a **declarative language** – you declare your desired result, and the RDBMS figures out the steps to get there.

SQL is used for:

- **Creating and modifying database structures:** Defining tables, specifying rules (constraints), and setting up relationships.
- **Manipulating data:** Inserting new records, updating existing ones, deleting records, and retrieving information.
- **Controlling access:** Granting or revoking permissions to users.

History of SQL:

SQL's origins trace back to the early 1970s at IBM.

- It was initially developed by Donald D. Chamberlin and Raymond F. Boyce at IBM's San Jose Research Laboratory.
- Their original language was called **SEQUEL** (Structured English Query Language), designed for IBM's System R experimental RDBMS.
- The name was later shortened to **SQL** due to a trademark conflict.
- SQL quickly gained popularity, leading to commercial implementations such as IBM's SQL/DS and DB2.

- Its success spurred other database vendors to adopt similar languages, eventually leading to the need for standardization.

SQL Standards:

To ensure interoperability and portability across different database systems, SQL has been standardized by major organizations:

- The **American National Standards Institute (ANSI)** first published an SQL standard in 1986.
- The **International Organization for Standardization (ISO)** also publishes SQL standards, often in conjunction with ANSI.
- Key standard versions include **SQL-92** (a significant milestone), SQL:1999, SQL:2003, SQL:2008, SQL:2011, and SQL:2016.
- **Importance of Standards:** While different RDBMS (like MySQL, PostgreSQL, Oracle Database, Microsoft SQL Server) have their own variations or "dialects" of SQL, they largely adhere to the core ANSI/ISO SQL standards. This adherence means that a basic SQL query written for one database system will often work with minor or no modifications on another, making SQL a truly universal language for relational databases.

4.2 Data Definition Language (DDL)

The **Data Definition Language (DDL)** portion of SQL is used to define, modify, and delete the structure (or schema) of your database. Think of DDL commands as the tools you use to build the "empty boxes" (tables) in your database and set the rules for what kind of data can go into them. DDL statements affect the database schema, not the data itself.

The primary DDL commands we will cover are `CREATE TABLE`, `ALTER TABLE`, and `DROP TABLE`.

4.2.1 `CREATE TABLE` Statement

The `CREATE TABLE` statement is used to create a new table in your database. When you create a table, you specify its name, the names of all its columns, the data type for each column, and any constraints (rules) that apply to those columns or the table as a whole.

General Syntax:

```
SQL
CREATE TABLE table_name (
    column1_name  data_type  [column_constraint],
    column2_name  data_type  [column_constraint],
    ...
    [table_constraint]
);
```

- `CREATE TABLE`: The keywords that initiate the creation of a new table.

- `table_name`: The unique name you choose for your new table (e.g., `Students`, `Courses`).
- `column_name`: The unique name for each column within that table (e.g., `StudentID`, `FirstName`).
- `data_type`: Specifies the type of data that can be stored in that column (e.g., `INTEGER`, `VARCHAR(100)`, `DATE`). We'll discuss basic data types in Section 4.4.
- `[column_constraint]`: An optional rule that applies to a single column (e.g., `NOT NULL`, `UNIQUE`). These are placed immediately after the column's data type.
- `[table_constraint]`: An optional rule that applies to the table as a whole, or involves multiple columns (e.g., `PRIMARY KEY (col1, col2)`, `FOREIGN KEY`). These are typically placed after all column definitions.

Example of `CREATE TABLE`:

Let's create a `Students` table and a `Departments` table:

```
SQL
CREATE TABLE Departments (
    DeptID      INTEGER     PRIMARY KEY,
    DeptName    VARCHAR(50) NOT NULL UNIQUE,
    Location    VARCHAR(100)
);

CREATE TABLE Students (
    StudentID       INTEGER       PRIMARY KEY,
    FirstName       VARCHAR(50)   NOT NULL,
    LastName        VARCHAR(50)   NOT NULL,
    DateOfBirth     DATE,
    Email           VARCHAR(100)  UNIQUE,
    MajorDeptID     INTEGER,
    EnrollmentDate DATE           DEFAULT CURRENT_DATE,
    CHECK (DateOfBirth < '2007-01-01'), -- Students must be at least 18 (roughly)
    FOREIGN KEY (MajorDeptID) REFERENCES Departments(DeptID)
);
```

4.2.2 Defining Constraints

Constraints are rules that are enforced on data columns or tables to maintain data integrity and consistency, as discussed in Module 2. They ensure that data conforms to specific business rules and prevent invalid data from being inserted, updated, or deleted.

Constraints can be defined in two ways:

- **Inline (Column-level):** Defined immediately after the column definition to which they apply.

- **Out-of-line (Table-level):** Defined separately after all column definitions, typically for constraints that involve multiple columns or for better readability.

Let's detail the most common constraints:

- **`PRIMARY KEY` Constraint:**
    - **Purpose:** Uniquely identifies each row (tuple) in a table. It ensures that no two rows have the same value for the primary key column(s), and that no part of the primary key is NULL. A table can have only one primary key.

Syntax (Inline - Single Column PK): `column_name data_type PRIMARY KEY`
SQL
StudentID INTEGER PRIMARY KEY

    -

Syntax (Out-of-line - Single or Composite PK): `PRIMARY KEY (column1_name, [column2_name, ...])`
SQL
PRIMARY KEY (StudentID) -- For a single column PK
PRIMARY KEY (CourseID, StudentID) -- For a composite PK (e.g., in an enrollment table)

    -
    - **Example from `Students` table:** `StudentID INTEGER PRIMARY KEY`
- **`FOREIGN KEY` Constraint:**
    - **Purpose:** Links a column (or set of columns) in one table (the "child" or referencing table) to the `PRIMARY KEY` (or `UNIQUE` key) in another table (the "parent" or referenced table). It enforces **referential integrity**, ensuring that a foreign key value always refers to an existing primary key value in the referenced table, or is NULL (if allowed).
    - **Syntax (Out-of-line - Most Common):** `FOREIGN KEY (referencing_column1, ...) REFERENCES referenced_table(referenced_pk_column1, ...) [ON DELETE action] [ON UPDATE action]`
    - `REFERENCES`: Keyword followed by the parent table name and the parent table's primary/unique key column(s) in parentheses.
    - `ON DELETE action`: Specifies what happens to child rows when the parent row is deleted. Common actions:
        - `NO ACTION` or `RESTRICT`: Prevents deletion of the parent row if child rows exist. (Default in many systems).
        - `CASCADE`: Deletes child rows when the parent row is deleted.
        - `SET NULL`: Sets the foreign key value in child rows to NULL when the parent row is deleted. (Requires the foreign key column to be nullable).
    - `ON UPDATE action`: Specifies what happens to child rows when the parent's primary key is updated. Actions are similar to `ON DELETE`.

        - **Example from `Students` table:** `FOREIGN KEY (MajorDeptID) REFERENCES Departments(DeptID) ON DELETE SET NULL ON UPDATE CASCADE`
            - This means if a department is deleted, students previously majoring in that department will have their `MajorDeptID` set to NULL.
            - If a `DeptID` in the `Departments` table changes, all `MajorDeptID` values in `Students` referring to it will automatically update to the new `DeptID`.
- **`NOT NULL` Constraint:**
    - **Purpose:** Ensures that a column cannot store NULL values. Every row must have a definite value for this column.
    - **Syntax (Inline):** `column_name data_type NOT NULL`
    - **Example from `Students` table:** `FirstName VARCHAR(50) NOT NULL`
- **`UNIQUE` Constraint:**
    - **Purpose:** Ensures that all values in a specific column (or set of columns) are unique across all rows in the table. While similar to `PRIMARY KEY`, a `UNIQUE` constraint can allow **one** NULL value for that column (if the column is not also `NOT NULL`).

Syntax (Inline - Single Column): `column_name data_type UNIQUE`
SQL
Email VARCHAR(100) UNIQUE

    -

Syntax (Out-of-line - Single or Composite): `UNIQUE (column1_name, [column2_name, ...])`
SQL
UNIQUE (DeptName) -- From Departments table example
UNIQUE (CourseID, ProfessorID) -- Example composite unique key

    -
    - **Example from `Students` table:** `Email VARCHAR(100) UNIQUE`
- **`CHECK` Constraint:**
    - **Purpose:** Enforces a specific condition on the values in a column or set of columns. If a value violates the condition, the insertion or update operation is rejected.

Syntax (Inline): `column_name data_type CHECK (condition)`
SQL
Age INTEGER CHECK (Age >= 18 AND Age <= 100)

    -

Syntax (Out-of-line - For complex conditions or multiple columns): `CHECK (condition)`

SQL
CHECK (Salary > 0)
CHECK (StartDate < EndDate)

    -
    - **Example from `Students` table:** `CHECK (DateOfBirth < '2007-01-01')`
- **`DEFAULT` Constraint:**
    - **Purpose:** Specifies a default value for a column. If no value is explicitly provided for this column during an `INSERT` operation, the default value will be automatically inserted.
    - **Syntax (Inline):** `column_name data_type DEFAULT default_value`
    - **Example from `Students` table:** `EnrollmentDate DATE DEFAULT CURRENT_DATE`
        - `CURRENT_DATE` is a function that returns the current system date.

4.2.3 `ALTER TABLE` Statement

The `ALTER TABLE` statement is used to modify the structure of an existing table. You can use it to add, drop, or modify columns, or to add and remove constraints.

Common `ALTER TABLE` operations:

Adding a New Column:
SQL
ALTER TABLE table_name
ADD COLUMN new_column_name data_type [constraint];

-
    - *Example:* `ALTER TABLE Students ADD COLUMN PhoneNumber VARCHAR(20);`

Dropping an Existing Column:
SQL
ALTER TABLE table_name
DROP COLUMN column_name;

-
    - *Caution:* This will permanently delete the column and all its data.
    - *Example:* `ALTER TABLE Students DROP COLUMN PhoneNumber;`

Adding a Constraint (out-of-line syntax only for existing tables):
SQL
ALTER TABLE table_name
ADD CONSTRAINT constraint_name constraint_definition;

-

    - `constraint_name`: An optional name for the constraint, useful for dropping it later.
    - *Example:* `ALTER TABLE Students ADD CONSTRAINT UQ_Email UNIQUE (Email);`

Dropping a Constraint:
SQL
ALTER TABLE table_name
DROP CONSTRAINT constraint_name; -- For named constraints
-- OR (depending on DBMS, for unnamed PRIMARY KEY/UNIQUE)
-- DROP PRIMARY KEY;
-- DROP UNIQUE (column_name);

-
    - *Example:* `ALTER TABLE Students DROP CONSTRAINT UQ_Email;`
- **Modifying a Column's Data Type or Properties (DBMS-specific syntax):**
    - Syntax for modifying columns can vary significantly between RDBMS (e.g., `ALTER COLUMN` in SQL Server, `MODIFY COLUMN` in Oracle, `ALTER COLUMN` in PostgreSQL/MySQL for specific changes). Always consult your specific DBMS documentation.

Example (Conceptual for changing data type, actual syntax varies):
SQL
ALTER TABLE Students
ALTER COLUMN FirstName TYPE VARCHAR(100); -- PostgreSQL/MySQL style

    -

4.2.4 `DROP TABLE` Statement

The `DROP TABLE` statement is used to completely remove an existing table from the database. This command deletes both the table's structure (schema) and all the data within it.

General Syntax:

SQL
DROP TABLE table_name;

- `DROP TABLE`: Keywords to initiate the table deletion.
- `table_name`: The name of the table to be deleted.

Caution: This is an irreversible operation! Once a table is dropped, its data and structure are lost unless you have a backup.

- *Example:* `DROP TABLE Students;`

- **Important:** If other tables have foreign keys referencing the table you are trying to drop, the `DROP TABLE` command might fail due to referential integrity constraints. You might need to drop the referencing foreign keys first, or use a `CASCADE` option (if supported and desired by your DBMS) to automatically drop dependent objects.
    - `DROP TABLE Departments CASCADE;` (If supported, this would drop `Departments` and automatically drop the foreign key `MajorDeptID` in `Students` that references `Departments`.)

4.3 Data Manipulation Language (DML)

The **Data Manipulation Language (DML)** portion of SQL is used for managing and manipulating the actual data stored within the database tables. DML commands allow you to insert new rows, retrieve existing data, update values in rows, and delete rows. These statements affect the content of the database, not its structure.

The primary DML commands we will cover are `INSERT`, `SELECT`, `UPDATE`, and `DELETE`.

4.3.1 `INSERT` Statements

The `INSERT` statement is used to add new rows (tuples) of data into an existing table.

Syntax Option 1: Inserting values into specific columns (Recommended): This is the safest and most common way to insert data, as it explicitly lists the columns you are providing values for.

SQL
```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

- `INSERT INTO table_name`: Specifies the table where data will be added.
- `(column1, column2, ...)`: An optional (but recommended) list of columns you are providing values for. The order here *must match* the order of values in the `VALUES` clause. If a column is omitted, it will either get its `DEFAULT` value or be `NULL` (if `NOT NULL` is not specified).
- `VALUES (value1, value2, ...)`: The list of actual values to be inserted. Each value must match the data type and constraints of its corresponding column. Text values are enclosed in single quotes (`'`).

Example 1 (Inserting into all specified columns):
SQL
```
INSERT INTO Departments (DeptID, DeptName, Location)
VALUES (10, 'Computer Science', 'Main Campus');
```

-

Example 2 (Inserting into specific columns, letting others default/be NULL):
SQL
INSERT INTO Students (StudentID, FirstName, LastName, MajorDeptID)
VALUES (1, 'Alice', 'Smith', 10);
-- DateOfBirth, Email, EnrollmentDate would be NULL or use their DEFAULTs

-

Syntax Option 2: Inserting values into all columns (Order-dependent - Use with Caution): If you omit the column list, you *must* provide values for *all* columns in the exact order they were defined in the `CREATE TABLE` statement. This is less robust as table structure changes can break your inserts.

SQL
INSERT INTO table_name
VALUES (value1, value2, ...);

Example: (Assuming `Students` columns are `StudentID, FirstName, LastName, DateOfBirth, Email, MajorDeptID, EnrollmentDate`)
SQL
INSERT INTO Students
VALUES (2, 'Bob', 'Johnson', '2005-02-20', 'bob@example.com', 10, '2023-09-01');

-

Syntax Option 3: Inserting data from another table (INSERT SELECT): You can insert rows into a table by selecting data from another table (or the same table).

SQL
INSERT INTO target_table (column1, column2, ...)
SELECT source_column1, source_column2, ...
FROM source_table
WHERE condition;

Example: Imagine a `NewStudents` table with temporary data.
SQL
INSERT INTO Students (StudentID, FirstName, LastName, Email)
SELECT NewStudentID, NewFirstName, NewLastName, NewEmail
FROM NewStudents
WHERE AdmissionStatus = 'Admitted';

-

4.3.2 `SELECT` Statement: Basic Queries

The `SELECT` statement is the most frequently used DML command. It is used to retrieve data from one or more tables in your database. The result of a `SELECT` query is always a new, temporary table called a **result set**.

Basic Syntax:

SQL
SELECT [DISTINCT] column1, column2, ... (or *)
FROM table_name
[WHERE condition];

- `SELECT`: The keyword that begins a data retrieval query.
- `[DISTINCT]`: An optional keyword. If included, it ensures that only unique (non-duplicate) rows are returned in the result set. If omitted, all matching rows, including duplicates, are returned.
- `column1, column2, ...`: A comma-separated list of the specific column names you want to retrieve.
- `*`: A wildcard character that means "select all columns" from the specified table(s).
- `FROM table_name`: Specifies the table (or tables) from which to retrieve the data.
- `[WHERE condition]`: An optional clause that filters the rows. Only rows that satisfy the specified condition(s) will be included in the result set.

Examples of Basic `SELECT` Queries:

Select all columns from a table:
SQL
SELECT *
FROM Students;

- - *Result:* All columns and all rows from the `Students` table.

Select specific columns from a table:
SQL
SELECT FirstName, LastName, Email
FROM Students;

- - *Result:* Only `FirstName`, `LastName`, and `Email` columns for all students.

Select distinct values from a column:
SQL
SELECT DISTINCT MajorDeptID
FROM Students;

- - *Result:* A list of unique `MajorDeptID`s found in the `Students` table (e.g., 10, 20, NULL).

Using `AS` for Column Aliases: You can rename columns in the result set for readability.
SQL
SELECT FirstName AS StudentFirstName, LastName AS StudentLastName
FROM Students;

-

The `FROM` Clause:

- **Purpose:** The `FROM` clause indicates which table(s) the data should be retrieved from. It's mandatory for `SELECT` queries that access data from tables.
- *Example:* `FROM Employees` tells the database to look for data in the `Employees` table.

The `WHERE` Clause:

- **Purpose:** The `WHERE` clause is used to filter rows based on a specified condition. Only rows that meet the condition(s) are included in the result. It's applied *before* any other operations like sorting or aggregation.
    - **Comparison Operators:** Used to compare values.
        - `=` (Equal to)
        - `!=` or `<>` (Not equal to)
        - `>` (Greater than)
        - `<` (Less than)
        - `>=` (Greater than or equal to)
        - `<=` (Less than or equal to)

Example:
SQL
SELECT StudentID, FirstName, LastName
FROM Students
WHERE MajorDeptID = 10; -- Students in DeptID 10
SQL
SELECT DeptName, Location
FROM Departments
WHERE Location != 'Main Campus'; -- Departments not on Main Campus

        -
    - **Logical Operators (Boolean Operators):** Used to combine multiple conditions.
        - `AND`: Both conditions must be true.
        - `OR`: At least one condition must be true.
        - `NOT`: Negates a condition (reverses its truth value).

Example:
SQL
SELECT FirstName, LastName, DateOfBirth

```
FROM Students
WHERE MajorDeptID = 10 AND DateOfBirth < '2005-01-01';
-- Students in Dept 10 AND born before Jan 1, 2005
SQL
SELECT StudentID, Email
FROM Students
WHERE MajorDeptID = 20 OR Email IS NULL;
-- Students in Dept 20 OR those with no email
SQL
SELECT DeptName
FROM Departments
WHERE NOT (Location = 'East Campus');
-- Departments not located on East Campus
```

- **Other Useful Operators in `WHERE` Clause (Fundamentals):**
  - **`BETWEEN value1 AND value2`:** Checks if a value is within a specified range (inclusive).
    - *Example:* `WHERE Salary BETWEEN 40000 AND 60000;`
  - **`IN (value1, value2, ...)`:** Checks if a value matches any value in a list.
    - *Example:* `WHERE MajorDeptID IN (10, 30, 50);`
  - **`LIKE pattern`:** Used for pattern matching with string values.
    - `%`: Matches any sequence of zero or more characters.
    - `_`: Matches any single character.
    - *Example:* `WHERE LastName LIKE 'Smi%';` (Starts with "Smi")
    - *Example:* `WHERE FirstName LIKE '_a%';` (Second letter is 'a')
  - **`IS NULL` / `IS NOT NULL`:** Checks for NULL values. You cannot use `=` or `!=` with NULL.
    - *Example:* `WHERE Email IS NULL;`
    - *Example:* `WHERE MajorDeptID IS NOT NULL;`

4.3.3 `UPDATE` Statements

The `UPDATE` statement is used to modify existing data in one or more rows of a table.

General Syntax:

```
SQL
UPDATE table_name
SET column1 = new_value1, column2 = new_value2, ...
[WHERE condition];
```

- `UPDATE table_name`: Specifies the table whose data will be updated.
- `SET column1 = new_value1, ...`: Lists the columns to be modified and their new values. You can update one or many columns in a single `UPDATE` statement.
- `[WHERE condition]`: This clause is **extremely important**. It specifies which rows will be updated. If the `WHERE` clause is omitted, the `UPDATE` statement will affect *all* rows in the table!

Example 1 (Updating specific rows):
SQL
UPDATE Students
SET Email = 'alice.smith@newdomain.com'
WHERE StudentID = 1;
-- Changes Alice Smith's email if StudentID is 1

-

Example 2 (Updating multiple columns for specific rows):
SQL
UPDATE Departments
SET Location = 'North Campus', DeptName = 'Computer Science & Engineering'
WHERE DeptID = 10;
-- Changes location and name for Department ID 10

-

Example 3 (Updating all rows - Use with Extreme Caution!):
SQL
UPDATE Employees
SET Salary = Salary * 1.05; -- Gives a 5% raise to ALL employees

-

4.3.4 `DELETE` Statements

The `DELETE` statement is used to remove one or more existing rows (tuples) from a table.

General Syntax:

SQL
DELETE FROM table_name
[WHERE condition];

- `DELETE FROM table_name`: Specifies the table from which rows will be deleted.
- `[WHERE condition]`: This clause is **extremely important**. It specifies which rows will be deleted. If the `WHERE` clause is omitted, the `DELETE` statement will remove *all* rows from the table!

Example 1 (Deleting specific rows):
SQL
DELETE FROM Students
WHERE StudentID = 2;
-- Deletes the student with StudentID 2

-

Example 2 (Deleting rows based on a condition):
SQL
DELETE FROM Departments
WHERE Location = 'Old Building';
-- Deletes all departments located in 'Old Building'

-

Example 3 (Deleting all rows - Use with Extreme Caution!):
SQL
DELETE FROM Employees; -- This will remove ALL records from the Employees table!

-
    - *Note:* While `DELETE FROM table_name;` removes all rows, it does *not* remove the table structure itself. The table will still exist, but it will be empty. To remove both structure and data, use `DROP TABLE`.

4.4 Basic SQL Data Types

When you define columns in your tables using `CREATE TABLE` or `ALTER TABLE`, you must specify a **data type** for each column. Data types tell the database what kind of values a column can hold (e.g., numbers, text, dates) and how much storage space to allocate. They are crucial for data validation and for performing correct operations.

Here are some of the most common and widely supported basic SQL data types. Keep in mind that exact names and variations might differ slightly between different RDBMS (e.g., MySQL, PostgreSQL, Oracle, SQL Server).

- **Numeric Types:** Used for storing numbers.
    - **`INTEGER` (or `INT`):** Stores whole numbers (integers) without any decimal places.
        - *Example:* `StudentID INTEGER`, `Age INT`.
    - **`DECIMAL(p, s)` (or `NUMERIC(p, s)`):** Stores exact decimal numbers.
        - `p` (precision): The total number of digits (before and after the decimal point).
        - `s` (scale): The number of digits after the decimal point.
        - *Example:* `Salary DECIMAL(10, 2)` (can store numbers up to 99,999,999.99).

    - **`FLOAT` (or `REAL`):** Stores approximate floating-point numbers (numbers with decimal places, but precision might vary). Use for scientific calculations where exact precision isn't paramount.
        - *Example:* `GPA FLOAT`.
- **String Types (Character Types):** Used for storing text.
    - **`VARCHAR(length)` (or `VARCHAR2`, `NVARCHAR`):** Stores variable-length strings. The `length` specifies the maximum number of characters allowed. It uses only the space needed for the actual data. `NVARCHAR` variations typically support Unicode characters.
        - *Example:* `FirstName VARCHAR(50)`, `Email VARCHAR(100)`.
    - **`CHAR(length)` (or `NCHAR`):** Stores fixed-length strings. If the actual string is shorter than `length`, it's padded with spaces. Uses fixed space regardless of content. Less common than `VARCHAR` for general text.
        - *Example:* `Gender CHAR(1)` (for 'M' or 'F').
    - **`TEXT` (or `LONGTEXT`, `CLOB`):** Used for very long strings or large blocks of text. The maximum length varies significantly by DBMS.
        - *Example:* `CourseDescription TEXT`.
- **Date/Time Types:** Used for storing dates, times, or both.
    - **`DATE`:** Stores a date (year, month, day).
        - *Example:* `DateOfBirth DATE`.
    - **`TIME`:** Stores a time (hour, minute, second).
        - *Example:* `ClassStartTime TIME`.
    - **`DATETIME` (or `TIMESTAMP`):** Stores both a date and a time. `TIMESTAMP` often includes time zone information or automatic update capabilities.
        - *Example:* `EnrollmentDateTime DATETIME`, `LastLogin TIMESTAMP`.
- **Boolean Types:** Used for storing true/false values.
    - **`BOOLEAN`:** Stores `TRUE` or `FALSE`. (Directly supported in some DBMS like PostgreSQL).
    - **`TINYINT` or `BIT`:** In some DBMS (like MySQL, SQL Server), `BOOLEAN` is often implemented as a numeric type (e.g., `TINYINT(1)` where 0 means false, 1 means true).
        - *Example:* `IsActive BOOLEAN` or `IsActive TINYINT(1)`.

Choosing the correct data type for each column is a crucial part of good database design, as it impacts storage efficiency, data integrity, and query performance.

Module Summary

In this Module 4, we embarked on our practical journey into **Structured Query Language (SQL)**, the universal language for interacting with relational databases. We started with an introduction to SQL's history and its importance as a standardized, declarative language for database management.

We then thoroughly explored the **Data Definition Language (DDL)** commands, which are used to define and manage the database's structure (its schema). You learned how to:

- Use `CREATE TABLE` to build new tables, specifying their columns and initial properties.
- Implement various **constraints** (`PRIMARY KEY`, `FOREIGN KEY`, `NOT NULL`, `UNIQUE`, `CHECK`, `DEFAULT`) to enforce data integrity rules directly within your table definitions, ensuring data quality and consistency.
- Modify existing table structures using `ALTER TABLE` to add or drop columns and constraints.
- Completely remove tables and their data using `DROP TABLE`.

Following DDL, we transitioned to the **Data Manipulation Language (DML)** commands, which are used to manage the actual data within your tables. You gained proficiency in:

- `INSERT` statements to add new rows of data into tables.
- The fundamental `SELECT` statement, the most common SQL command, for retrieving data. You learned to select specific columns or all columns, use `DISTINCT` for unique results, and crucially, apply the `WHERE` clause with various **comparison** and **logical operators** (`AND`, `OR`, `NOT`, `BETWEEN`, `IN`, `LIKE`, `IS NULL`) to filter rows based on specific conditions.
- `UPDATE` statements to modify existing data in one or more rows.
- `DELETE` statements to remove specific rows from a table.

Finally, we covered the **Basic SQL Data Types**, understanding their role in defining column characteristics and ensuring appropriate storage and validation for numeric, string, date/time, and boolean values.