Data Definition Language (DDL)

Data Definition Language (DDL) is a category of SQL (Structured Query Language) commands used to define, modify, and remove the structure of database objects such as databases, tables, indexes, and views. DDL statements specify how data is stored, organized, and related within the database, but they do not manipulate the actual data within the tables.

Creating a Database

To create a new database in SQL (if it doesn't already exist):

```
CREATE DATABASE IF NOT EXISTS university;
```

CREATE DATABASE creates a new database.
IF NOT EXISTS prevents an error if it already exists.

Switching to the Database

Once our database exists, we need to tell the SQL server to use it for our next actions. Otherwise, we might accidentally create tables somewhere else! Note that when we use python connection, we identify the database name that we are working with to get the connection.

```
USE university;
```

The USE command tells SQL which database we're working in. After this, any new tables we make will go into the university database.

Creating Tables

Each relation in relational databases is stored in a table. We're going to create three tables: one for students, one for courses, and one to keep track of enrollments.

Student Table

The student table will store information about each student.

```
CREATE TABLE IF NOT EXISTS student (
    sId INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
```

sId INT PRIMARY KEY: Every student gets a unique ID number (an integer). The PRIMARY KEY makes sure there can't be two students with the same ID.

name VARCHAR(255) NOT NULL: Each student must have a name. VARCHAR(255) means it can be up to 255 characters, and NOT NULL means a name is required (no blanks allowed).

Supported MySQL Data Types

For each attribute like name or sId we have to select appropriate data type. The following types are supported by MySQL:

- INT: Standard integer (whole numbers), commonly used for IDs and counts.
- SMALLINT: Smaller-range integer (saves space for small values).
- BIGINT: Large-range integer (for numbers too big for INT).

- FLOAT: Single-precision floating-point number (approximate decimals).
- DOUBLE: Double-precision floating-point number (more precision than FLOAT).
- DECIMAL(p, s): Exact numeric type for fixed-point numbers, with total digits (p) and decimals (s). Ideal for money or quantities needing exact precision.
- CHAR(n): Fixed-length string. Always stores exactly n characters.
- VARCHAR(n): Variable-length string, up to n characters. Most common for names and general text fields.
- TEXT: Long variable-length string (up to 65,535 characters), for large blocks of text.
- DATE: Stores dates (year, month, day), e.g., '2025-07-13'.
- TIME: Stores time of day, e.g., '15:30:00'.
- DATETIME: Stores both date and time, e.g., '2025-07-13 15:30:00'.
- TIMESTAMP: Like DATETIME, but often auto-updated for record creation/modification times.
- TINYINT(1): Very small integer, often used as a boolean (0 for false, 1 for true).
- BLOB: Binary Large Object. Stores binary data such as images, files, etc.
- ENUM('value1', 'value2', ...): Only allows one value from a list of predefined strings.

Course Table

Now, let's go back to university database. The course table holds the information about each course.

```
CREATE TABLE IF NOT EXISTS course (
    cId INT PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);
```

Enrolled Table

We also need to keep track of which students are taking which courses, and optionally their grade. This requires a table that links students to courses.

```
CREATE TABLE IF NOT EXISTS enrolled (
    sId INT,
    cId INT,
    grade FLOAT,
    PRIMARY KEY (sId, cId),
    FOREIGN KEY (sId) REFERENCES student(sId),
    FOREIGN KEY (cId) REFERENCES course(cId)
);
```

- sId INT, cId INT: These two columns will store the IDs of a student and a course.
- grade INT: The student's grade (this is optional, so it can be left blank if not known yet).

- PRIMARY KEY (sId, cId): Together, student ID and course ID make a unique combination. This means a student can only enroll in the same course once. When we define the primary key on a *separate line* (instead of directly after the column), it allows us to specify a **composite key** that involves multiple columns.

- FOREIGN KEY (sId) REFERENCES student(sId): Only allows enrollment for students that exist in the student table. This means that if you try to insert a row into `enrolled` with an `sId` that is not present in the `student` table, the database will reject the operation.
- FOREIGN KEY (cId) REFERENCES course(cId): Only allows enrollment for courses that exist in the course table. Similarly, you cannot enroll a student in a non-existent course. This ensures that the database remains consistent and prevents "orphaned" records in the `enrolled` table.

Foreign Keys and Referential Integrity

A **foreign key** is a column (or a set of columns) in one table that creates a link between data in two tables. It acts as a reference to the primary key in another table, ensuring that relationships between records remain consistent. The primary key uniquely identifies each row in its own table. For example, in a `student` table, `sId` might be the primary key. The foreign key in a different table must match a value of the primary key in the referenced table, or be `NULL` (if allowed). For example, in the `enrolled` table, `sId` is a foreign key referencing `student(sId)`, which indicates which row in the `student` table corresponds to the current row.

From a data engineering perspective, foreign keys are important because:

- They enforce **referential integrity**, preventing orphaned records (e.g., no enrollment without a valid student). Additionally, if a student is deleted from the `student` table, the database will either prevent the deletion (default behavior) or apply rules such as `CASCADE` or `SET NULL`, depending on how the foreign key is defined.
- They can introduce **performance trade-offs**: insert, update, and delete operations may be slower due to validation checks, especially on large tables.
- In very large databases, some teams choose to enforce these relationships through **ETL processes or application code** instead of database constraints, to improve scalability and flexibility.

Example Queries

We use the primary key-foreign key relationship to combine the information from different tables together, here student and enrolled tables. For example, if we want to see the name of student as well as their GPA:

```
SELECT student.name, AVG(enrolled.grade) AS GPA
FROM student
JOIN enrolled ON student.sId = enrolled.sId
GROUP BY student.sId;
```

Using the foreign key to the course table, we can also combine the information of enrolled and course. For example, we can collect the average grade in each course:

```
SELECT course.name, AVG(enrolled.grade) AS average_grade
FROM course
JOIN enrolled ON course.cId = enrolled.cId
GROUP BY course.cId;
```

Altering Tables

Sometimes we need to change a table after we've created it.

```
ALTER TABLE student ADD COLUMN email VARCHAR(255) UNIQUE;
```

This adds an `email` column to the `student` table. The `UNIQUE` constraint ensures that no two students can have the same email address, which helps maintain data consistency.

Similarly, we can remove columns:

```
ALTER TABLE student DROP COLUMN email;
```

This removes the `email` column from the table entirely. Removing a column should be done carefully, since once dropped, the data stored in that column is permanently lost.

From a data engineering perspective, altering tables is important because:

- It allows **schema evolution** without recreating the entire database.
- Adding columns with constraints (like `UNIQUE` or `NOT NULL`) helps enforce data quality.
- Frequent schema changes on very large tables can be **expensive**, as the database may need to rewrite data or rebuild indexes.
- For production systems, schema changes are often managed through **migration scripts** to keep track of how the database evolves over time.

Dropping and Truncating Tables

We can remove a table completely using `DROP`. This deletes both the table definition and all of its data permanently:

```
DROP TABLE enrolled;
```

If we only want to remove the data in the table but keep the table structure for future use, we can use `TRUNCATE`:

```
TRUNCATE TABLE enrolled;
```

From a data engineering perspective:

- `DROP TABLE` removes the table definition and all its data — the table no longer exists.
- `TRUNCATE TABLE` is faster than deleting rows one by one, since it resets the table without logging each deletion.
- `TRUNCATE` cannot be rolled back in some database systems unless wrapped in a transaction, so it must be used with care.
- Engineers often use `TRUNCATE` when reloading staging tables during ETL processes, while `DROP` is used when a table is no longer needed.

Database Schema

We demonstrate these tables and their relations using a **database schema**. A database schema is the **blueprint** or **map** of a database. It shows how data is organized, what tables exist, what columns are in each table, the data types of those columns, and how tables are related through keys. Importantly, the schema does not contain the actual data itself — just the design that structures the data.

Why the Schema Matters

- Think of a schema as the *floor plan of a building*: it shows where the rooms (tables) are, what each room contains (columns and data types), and how the rooms connect (relationships). The floor plan does not contain furniture (data), but it tells you how the furniture can be arranged.
- Without a schema, a database would just be a collection of unstructured data. The schema ensures that data is stored consistently and that relationships (such as students being enrolled in courses) make sense.
- The schema is what allows multiple developers, analysts, or engineers to share a common understanding of how the data is structured.

Key Components of a Schema

- **Tables**: The main entities (e.g., `student`, `course`, `enrolled`).
- **Columns**: The attributes of each table (e.g., `name`, `grade`).
- **Data types**: Define what kind of data each column can hold (e.g., `INT`, `VARCHAR(255)`).
- **Primary keys**: Unique identifiers for rows in a table (e.g., `sId` in `student`).
- **Foreign keys**: References to primary keys in other tables, creating links between them (e.g., `sId` in `enrolled` referencing `student(sId)`).
- **Relationships**: The logical connections between tables, often represented as lines in a schema diagram.

Example: University Schema

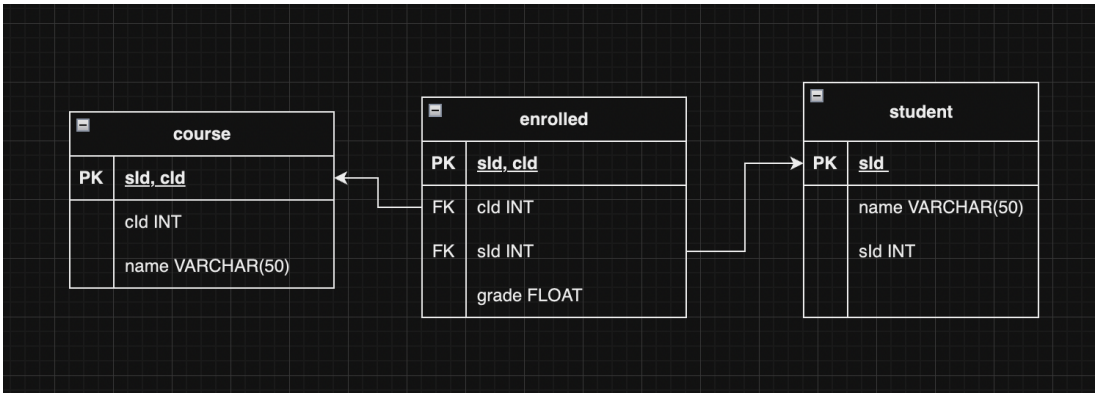
For our university database:

- The `student` table lists each student with their unique `sId` and name.
- The `course` table lists each course with a unique `cId` and course name.
- The `enrolled` table links students to the courses they are taking and optionally stores their grade.

How to Read the Schema Diagram

- Rectangles represent tables.
- Inside each rectangle are the column names and their data types.
- The primary key column is underlined or marked with a key symbol.
- Arrows between tables show foreign key relationships (e.g., `enrolled.sId` connects to `student.sId`).

Now we can draw the schema for the university database. This diagram is not the data itself, but the *design* that shows how the data is structured and how the tables connect.



Creating Schema is part of the design process, and often we first create the schema and then create tables based on the schema. (I used the draw.io, you need to draw the schema for your final project, and I recommend using the same tool)