**Università Degli Studi Niccolò Cusano**

**FACULTY OF ENGINEERING**

**BASI DATI PROJECT (E-tivity 4)**



**Application design and implementation with database from a case study**

**Kaan Bilgili**

**ERASMUS 307**

**09.11.2001**

**1. Selected Case Study**

**Title:** High School Management System

**2. Case Study Description**

Designing a database to manage a high school's students, instructors, classes, and classrooms and the school itself is the goal of this project. The system will support course enrollments, teacher assignments, classroom management, and administrative staff data management.

**Key Requirements:**

A student may sign up for more than one course.

Multiple courses may be assigned to teachers.

Classes take place in specified classrooms.

Secretaries, accountants, and other administrative personnel manage a variety of school-related tasks.

Classrooms have floor information and capacities.

All employees' and students' personal information needs to be kept on file.

School information (name, address, contact) should be recorded.

**3. Requirements Analysis and Collection**

**General Requirements**

* Personal information of students, teachers and administrative staff must be managed.
* Course and registration procedures must be done.
* Teachers should be able to assigned to classes.
* Manages classrooms and their capacities.

**Specific Requirements**

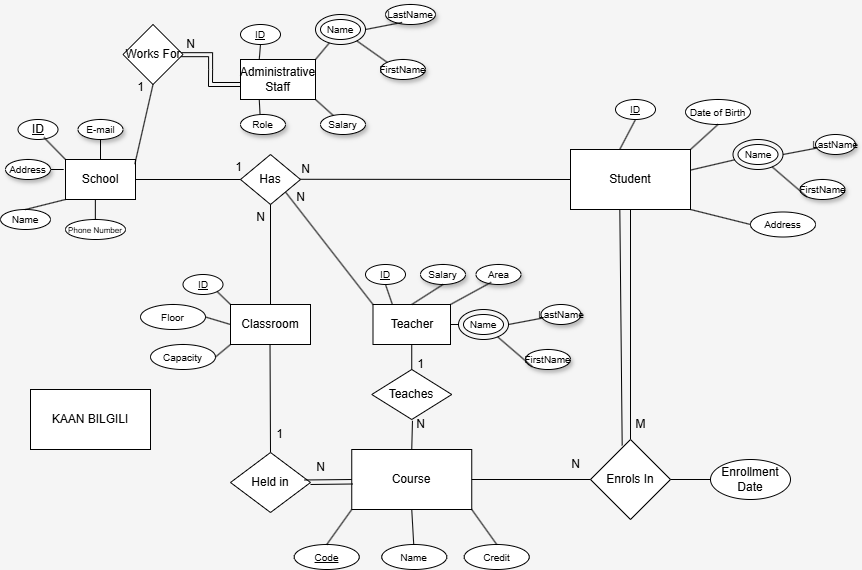
* **Student:** Student ID, First Name, Date of Birth, Address.
* **Teacher:** ID, Name, Area of Expertise, Salary.
* **Administrative Staff:** ID, Name, Role, Salary.
* **Course:** Course Code, Course Name, Number of Credits.
* **Classroom:** ID, Capacity, Floor Number.
* **School:** School ID, Name, Address, Phone Number, Email.
* **Enrollment**: Enrollment Date.

**4. Structuring Requirements into Homogeneous Sentence Groups**

* **Personal Information Management:**  
  Manage the data of students, teachers, and administrative staff.
* **Course Management:**  
  Handle course enrollments and teacher assignments.
* **Classroom Management:**  
  Manage classroom capacity and floor location information.
* **School Management:**  
  Store and manage general information about the school.

**5. Glossary of Terms**

|  |  |
| --- | --- |
| **Student** | A person enrolled in one or more classes at the institution. |
| **Teacher** | Someone in the position of educating one or more classes. |
| **Administrative Staff** | Individuals in charge of managing the school's administrative operations, such as accountants and secretaries. |
| **Course** | A course or curriculum that the school offers. |
| **Classroom** | Classes take place in a physical location that is identified by its floor number and capacity. |
| **Enrollment** | Enrollment: The process through which a student registers for a course. Includes additional information such as the Enrollment Date. |
| **School** | The institution being managed, containing all relevant information like name, address, and contact details. |
| **Course Assignment** | The process of assigning a teacher to a specific course. |

****  
**6. ER Model**

**7. Design Strategy Used**

For the design of the High School Management System database, a **Top-Down Design Approach** was adopted.

Initially, the system was analyzed at a high level by identifying the main entities involved: Student, Teacher, Administrative Staff, Course, Classroom, and School. This high-level conceptual understanding allowed the identification of the primary data requirements and functional areas such as personal information management, course management, and classroom management.

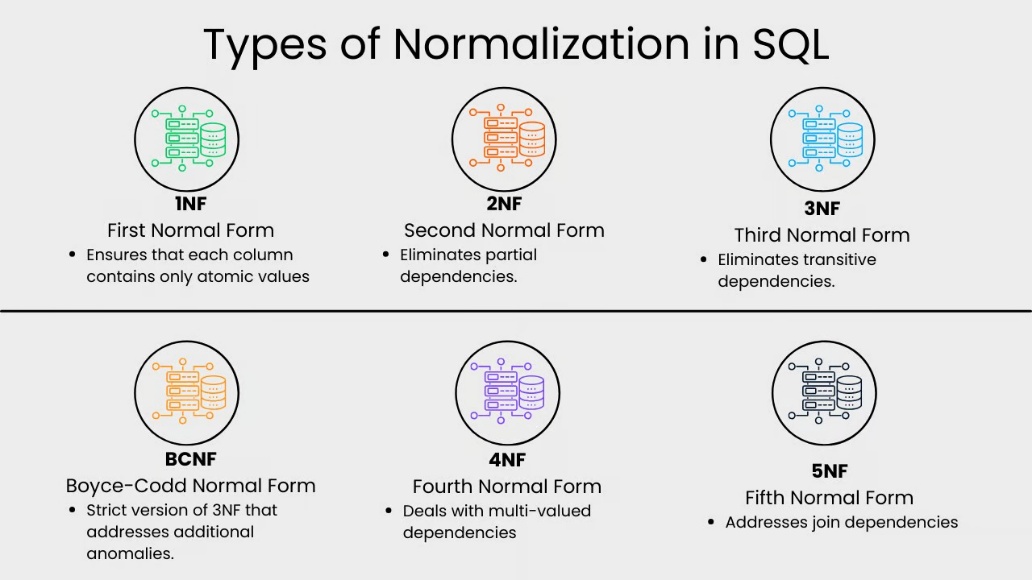
Subsequently, each entity was further detailed with its specific attributes and relationships. The system's core processes, such as Enrollment and Course Assignment, were also modeled to accurately represent the real-world operations within a high school environment.

This approach ensured a comprehensive view of the system before delving into finer details, allowing for better organization and integrity of the final E-R model.

**7. Logical Design and Normalization**

Normalization plays a crucial role in database design. If a table is not properly normalized and has data redundancy, it will not only take up extra data storage space but also make it difficult to handle and update the database. Normalization has many advantages. Redundancy is when the same information is stored multiple times, and a good way of avoiding this is by splitting data into smaller tables.You can perform faster query execution on smaller tables that have undergone normalization.With normalized tables, you can easily update data without affecting other records. It ensures that data remains consistent and accurate.

**7.1 Types of Normalization on SQL**



First Normal Form (1NF)

This normalization level ensures that each column in your data contains only atomic values. Atomic values in this context means that each entry in a column is indivisible. It is like saying that each cell in a spreadsheet should hold just one piece of information. 1NF ensures atomicity of data, with each column cell containing only a single value and each column having unique names.

Second Normal Form (2NF)

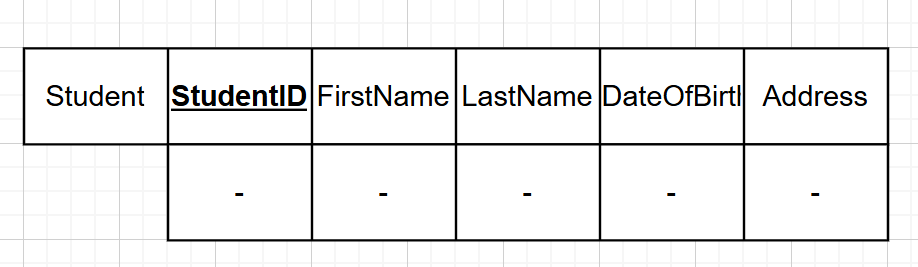
Eliminates partial dependencies by ensuring that non-key attributes depend only on the primary key. What this means, in essence, is that there should be a direct relationship between each column and the primary key, and not between other columns.

Third Normal Form (3NF)

Removes transitive dependencies by ensuring that non-key attributes depend only on the primary key. This level of normalization builds on 2NF.

**7.2 Normalization on Tables**

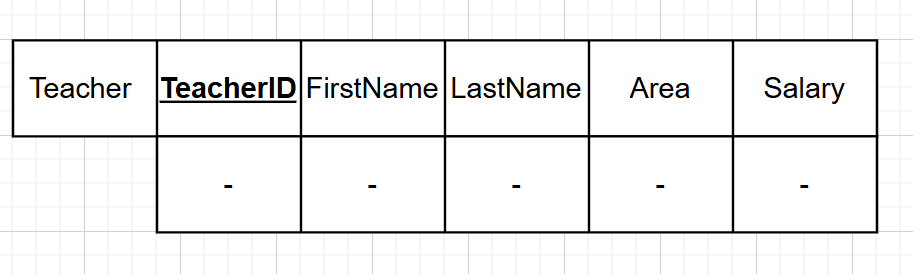
**1. Studen**t (StudentID(PK), FirstName, LastName, DateOfBirth, Address)



**1NF:** All values are atomic ad indivisible.

**2NF:** Non-key attributes fully depend on the primary key StudentID and no partial dependencies.

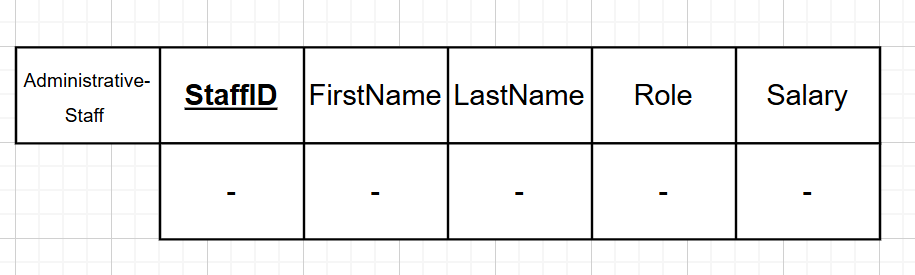
**3NF:** No transitive dependencies.

**2.Teacher** (TeacherID (PK), FirstName, LastName, Area, Salary)

**1NF:** All attributes contain atomic values.

**2NF:** TeacherID is the primary key, and all other attributes depend entirely on it.

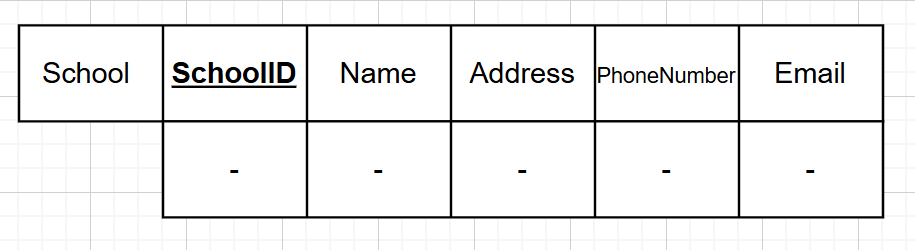
**3NF:** There are no dependencies between non-key attributes like Area and Salary.

**3.AdministrativeStaff** (StaffID (PK), FirstName, LastName, Role, Salary)

**1NF:** Attributes are atomic with no repeating groups

**2NF:** The primary key is StaffID, and all other fields fully depend on it.

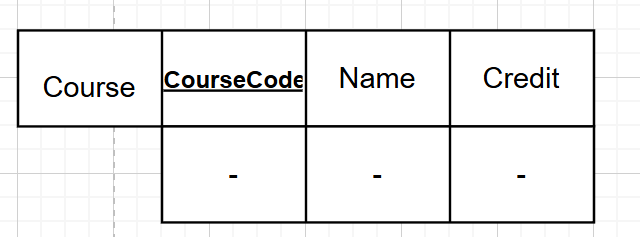
**3NF:** No transitive dependencies exist between non-key attributes.

**4.School** (SchoolID (PK), Name, Address, PhoneNumber, Email)

**1NF:** All attributes contain atomic values.

**2NF:** Each non-key attribute fully depends on SchoolID.

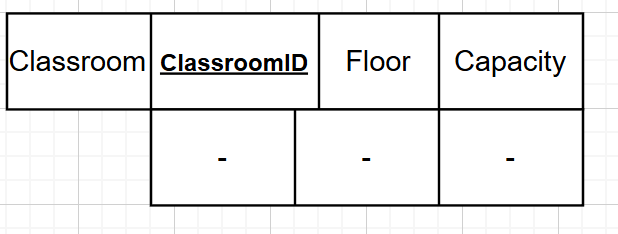
**3NF:** There are no transitive dependencies among non-key attributes.

**5.Course** (CourseCode (PK), Name, Credit)

**1NF:** Atomic values, no repeating groups.

**2NF:** CourseCode is the primary key, and other fields fully depend on it.

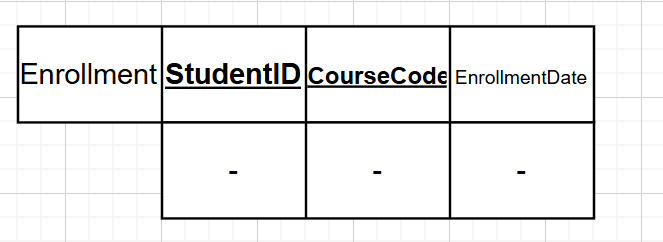
**3NF:** No transitive dependencies are present.

**6.Classroom** (ClassroomID (PK), Floor, Capacity)

**1NF:** All attribute values are atomic.

**2NF:** All non-key attributes directly depend on ClassroomID.

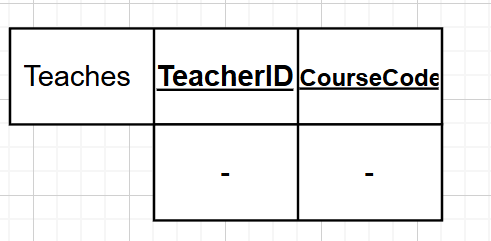
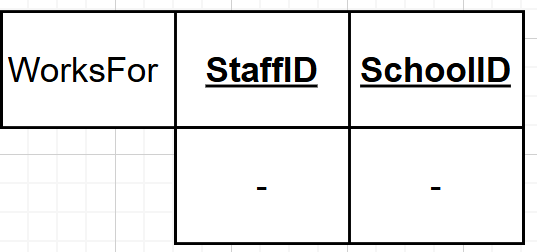
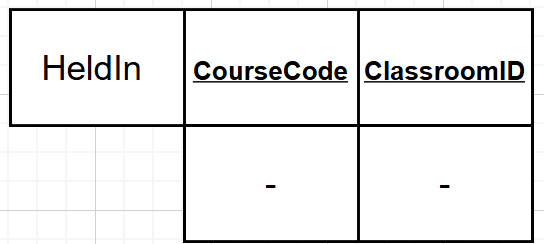
**3NF:** No dependencies between non-key attributes exist.

**7.Enrollment** (StudentID (PK, FK), CourseCode (PK, FK), EnrollmentDate)

**1NF:** All attributes have atomic values.

**2NF:** EnrollmentDate depends fully on both StudentID and CourseCode.

**3NF:** No transitive dependencies are found.

******8.Teaches** ,**HeldIN, WorksFor**

Teaches (TeacherID (PK, FK), CourseCode (PK, FK))

HeldIn (CourseCode (PK, FK), ClassroomID (PK, FK))

WorksFor(StaffID (PK, FK), SchoolID (PK, FK))

**1NF:** All values are atomic.

**2NF:** Composite primary keys correctly define the relationships, and all attributes fully depend on them.

**3NF:** There are no non-key attributes, hence no transitive dependencies.

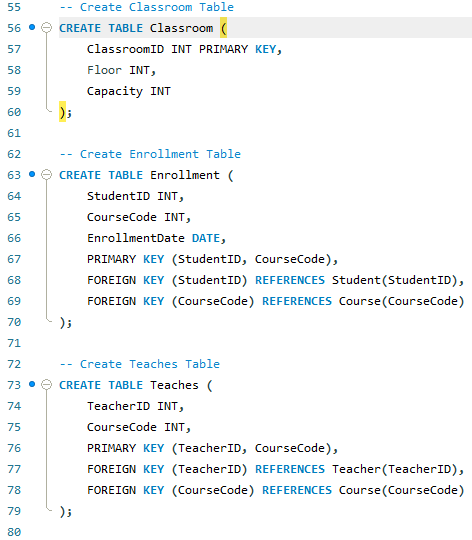
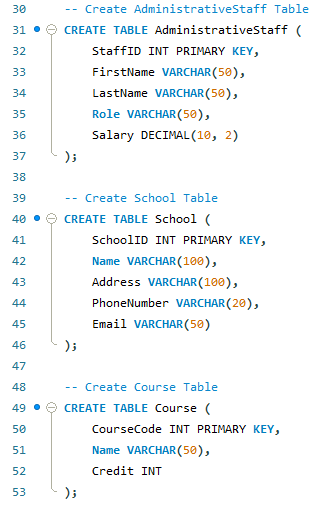
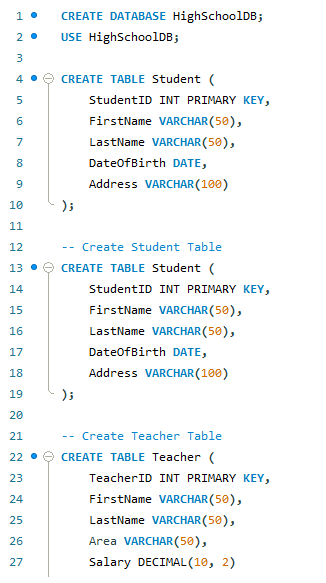
In conclusion all tables have been normalized to third normal form (3NF). No redundant data, partial dependencies, or transitive dependencies remain. This ensures data consistency, integrity, and minimizes redundancy in the database structure.

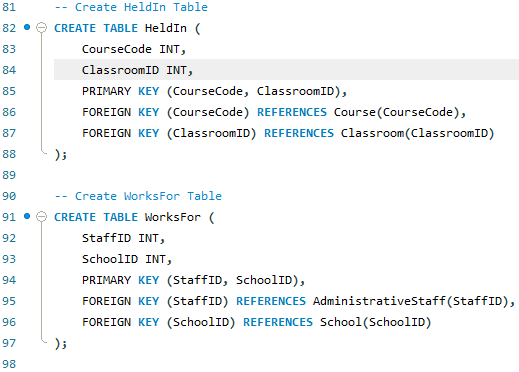
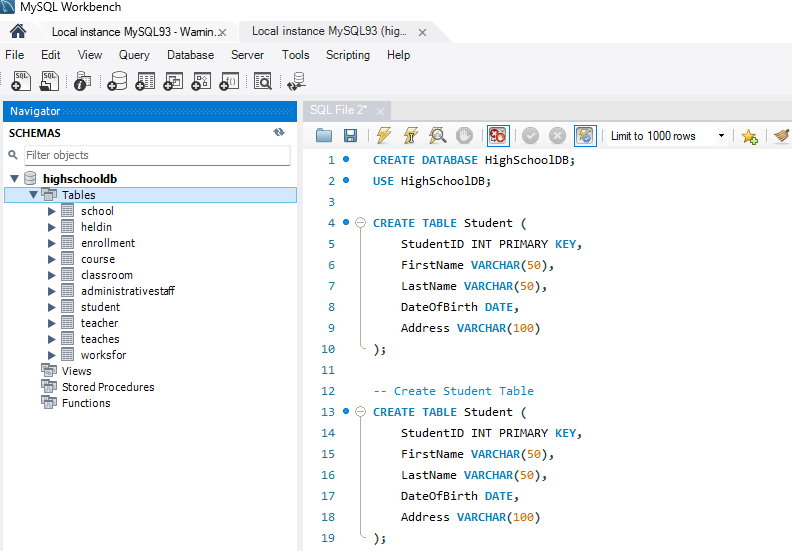
**8. DDL and DML**

**8.1 DDL Implementation**

In this phase, I used MySQL Workbench to create and manage the physical design of the database. This tool made it easier to write and execute SQL commands through its graphical interface.

The conceptual schema that I designed in the earlier phases was converted into SQL code using DDL (Data Definition Language) statements. With these commands, I created the necessary tables for each entity, defined their attributes and data types, and established primary and foreign keys to organize the relationships between tables

********

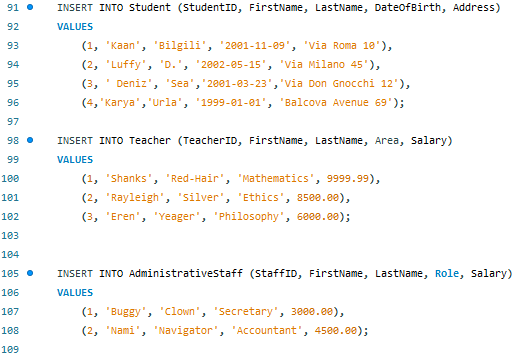
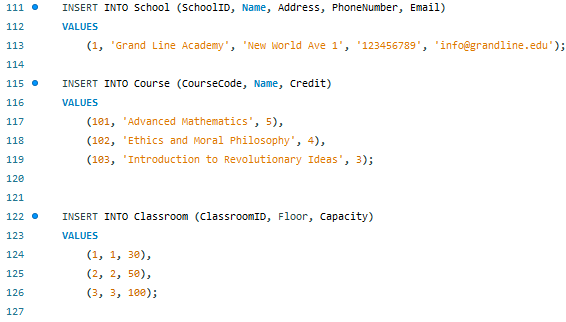


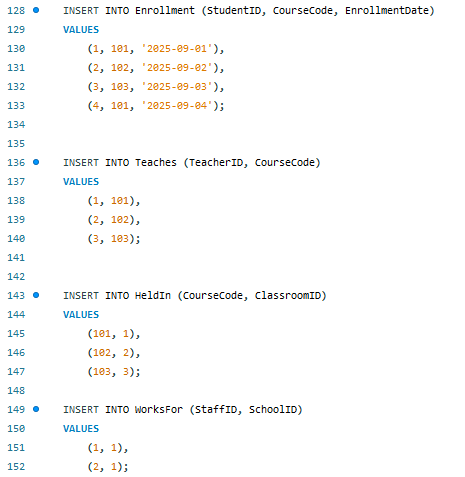
The figures above shows the final database schema created in MySQL Workbench, including all the necessary tables for the High School Management System.

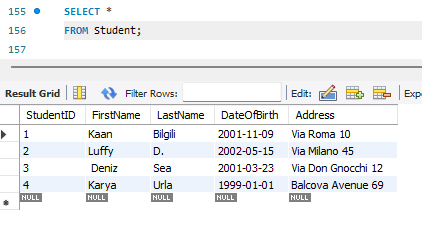
**8.2 DML Implementation**

In this phase, I used MySQL Workbench to insert sample data into the previously created tables using DML (Data Manipulation Language) statements. This allowed me to test the integrity of the database structure and verify that the relationships between tables were correctly established.

Using INSERT INTO commands, I populated the tables with sample records that reflect realistic scenarios based on the case study. These data entries were important to ensure that the defined primary and foreign keys function as expected and that the database supports accurate data retrieval through relational queries.

The following screenshots show the execution of DML statements and the successful insertion of data into the database.

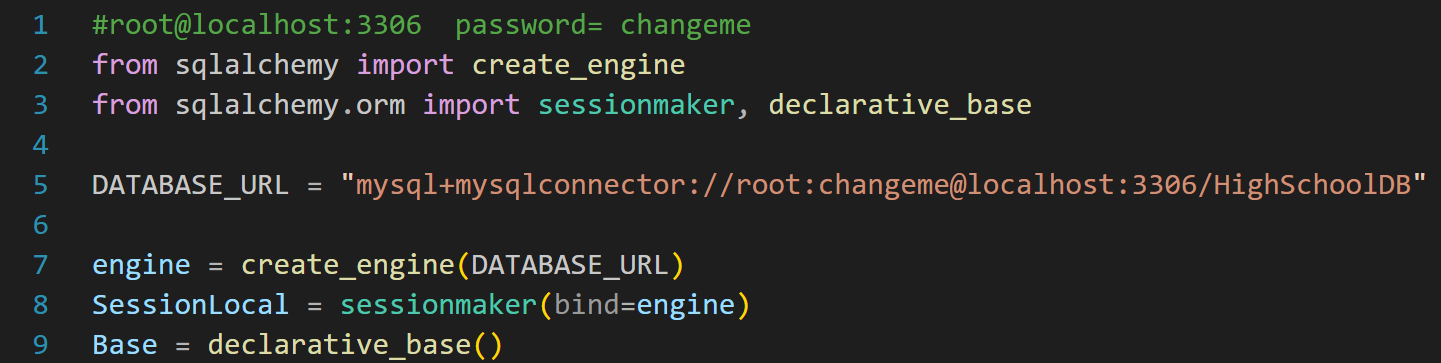
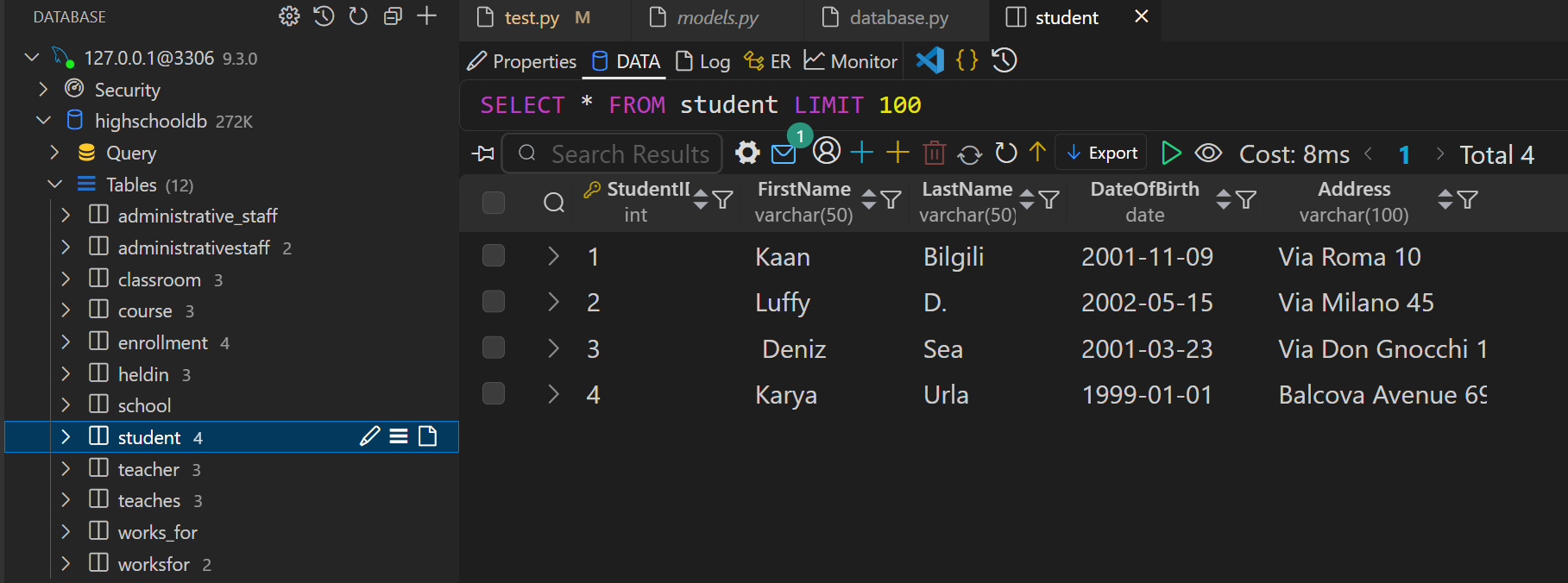




The figure confirms that the data was successfully inserted into the Student table and that the database is functioning as expected. The SELECT \* FROM Student; query correctly returns the inserted records, verifying that the DML operations were executed without errors.

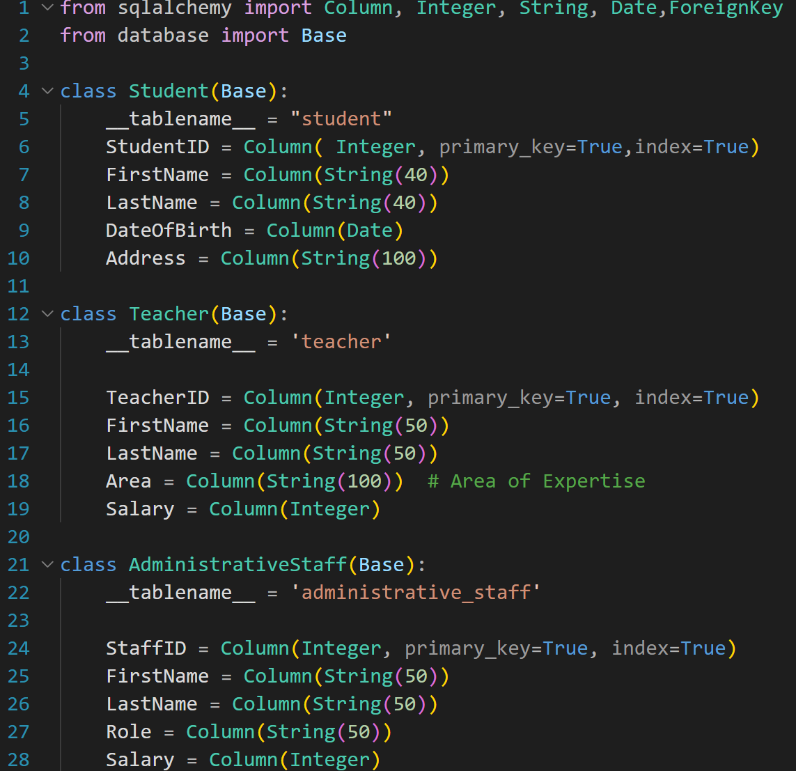
**9.SQLAlchemy-Based Application Development**

**9.1 Database.py**

**** ****

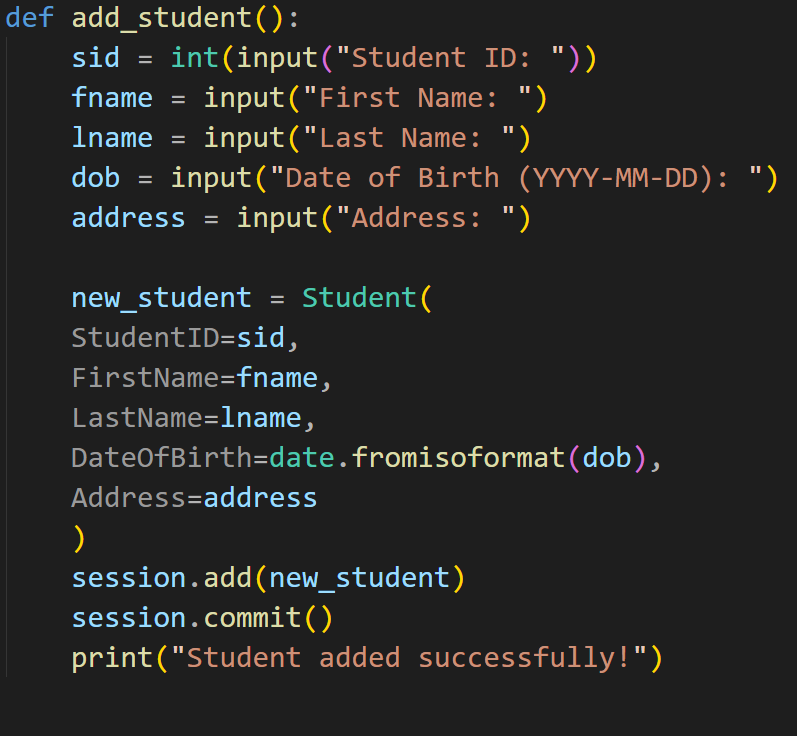
In this section, the database connection is established using SQLAlchemy's create\_engine() and sessionmaker(). The DATABASE\_URL specifies the connection details required to access the MySQL database. After successfully connecting, the student table is accessed and queried directly from the Visual Studio Code environment using SQLAlchemy ORM. The results show that the data has been inserted correctly and is retrievable from the database.

**9.2 Models.py**

The figure above demonstrates how the database tables are modeled as Python classes using the SQLAlchemy ORM. Each class represents a table in the database, and the attributes within the classes define the corresponding table columns. Primary keys and data types are explicitly declared to ensure referential integrity and correct data representation. This object-oriented approach simplifies database operations by abstracting SQL queries into class-based interactions.

**9.3 Test.py**

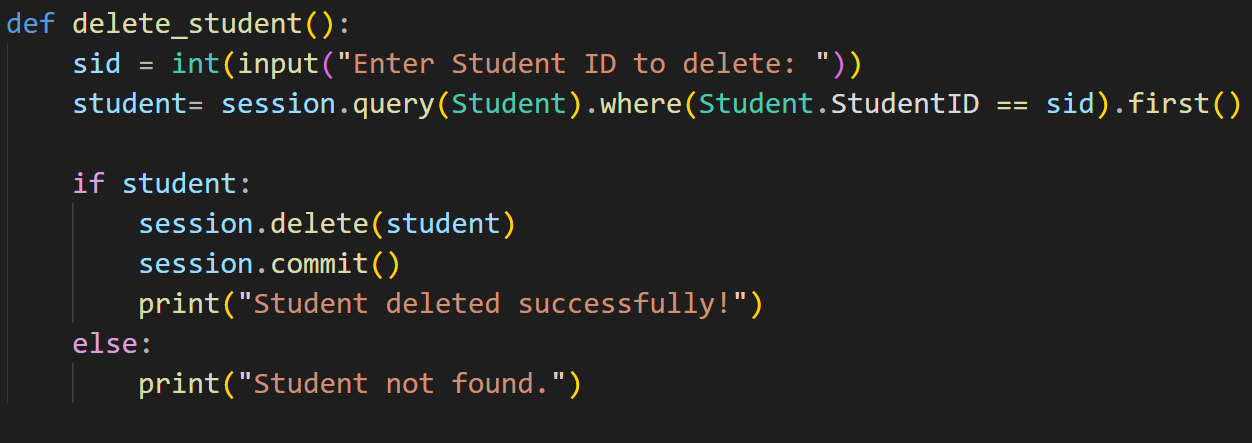
**9.3.1 add\_student() function**

****

INSERT INTO Student (StudentID, FirstName, LastName, DateOfBirth, Address)

VALUES (sid, 'fname', 'lname', 'dob', 'address');

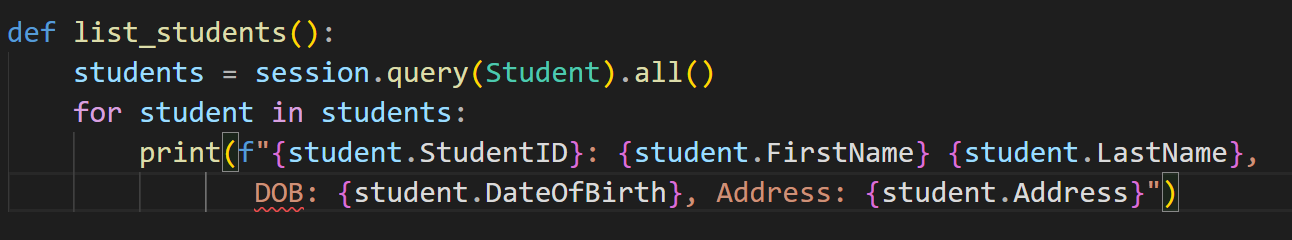
**9.3.2 delete\_student() function**



DELETE FROM Student

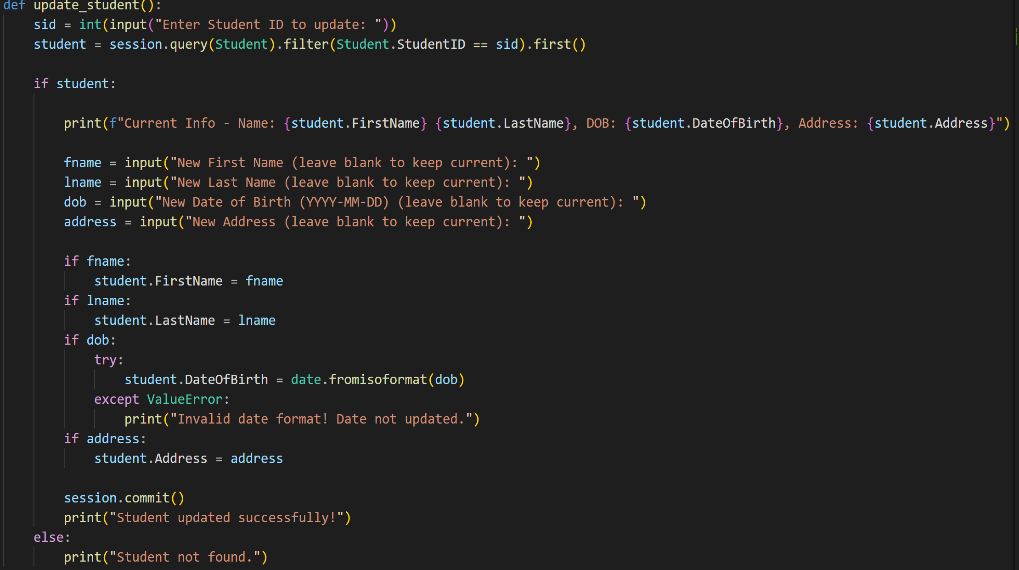
WHERE StudentID = sid;

**9.3.3 list\_students() function**

****

SELECT StudentID, FirstName, LastName, DateOfBirth, Address FROM Student;

**9.3.4 list\_students() function**

****

UPDATE Student

SET FirstName = 'fname',

LastName = 'lname',

DateOfBirth = 'YYYY-MM-DD',

Address = 'address'

WHERE StudentID = sid;