A Relational Database Approach

UNIVERSITY MANAGEMENT SYSTEM

[1]MOHAMED ASMATHULLA G

I. ABSTRACT

The study focuses on designing an optimized database system using an Entity-Relationship (ER) model and its relational mapping to ensure data integrity and minimize redundancy. Key entities identified include Student, Course, Professor, Department, and Enrollment, with Enrollment being a weak entity dependent on Student and Course. Attributes are categorized as primary keys, multi-valued, and derived attributes to enhance data organization. Relationship modeling defines one-to-one, one-to-many, and many-to-many relationships to establish clear connections between entities. The relational schema is structured using normalization techniques, including First Normal Form (1NF), to remove multi-valued attributes and improve efficiency. Various SQL join operations such as INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN, SELF JOIN, and CROSS JOIN are implemented to retrieve meaningful data relationships. The methodology ensures consistency, referential integrity, and streamlined database operations, demonstrating an effective approach to structured data management.

II. INTRODUCTION

Database management plays a crucial role in efficiently organizing and handling data in modern applications. This study focuses on designing and implementing a structured relational database system using the Entity-Relationship (ER) model and its transformation into a relational schema. The database includes key entities such as Student, Course, Professor, Department, and Enrollment, ensuring comprehensive data representation. Relationships between these entities, including one-to-one, one-to-many, and many-to-many, establish logical connections that facilitate accurate data retrieval. Normalization techniques, particularly First Normal Form (1NF), are applied to eliminate redundancy and maintain data consistency. Various SQL join operations, such as INNER JOIN, LEFT JOIN, and CROSS JOIN, are used to analyze inter-entity relationships and enhance data accessibility. This research highlights the significance of structured data modeling, referential integrity, and optimized querying in database design, ensuring efficient storage and retrieval of academic records.

III. METHODOLGY

In this study, we identify key entities such as Student, Course, Professor, Department, and Enrollment, where Enrollment is a weak entity dependent on Student and Course. The attributes for each entity include primary keys like Student_ID, Course_ID, and Department_ID, along with multi-valued attributes like Professor's Phone_Number and derived attributes such as Age (calculated from DOB).

Relationship modeling defines how entities interact: a one-to-one relationship exists between Course and Professor, a one-to-many relationship exists between Department and Professors, while a many-to-one relationship represents multiple Students enrolling in a single Course. A many-to-many relationship is present between Students and Courses, where each Student can enroll in multiple Courses, and each Course can have multiple Students.

Based on the ER model, the relational schema is structured with normalized tables, including foreign keys to maintain referential integrity. The Student table holds personal information, Course associates with a Professor, Enrollment links Students and Courses, and Department maintains academic divisions. Additionally, updates ensure data consistency, such as enforcing constraints for attributes like Grade and Credits, ensuring valid relationships between entities, and implementing ON DELETE CASCADE for referential integrity.

IV. ER MODEL

STEPS FOR CREATING AN ER MODEL

1. IDENTIFY ENTITIES

The first step in designing an ER model is to identify the key entities in the system. Entities represent real-world objects or concepts that have distinct identities.

2. DEFINE ATTRBUTES FOR EACH ENTITY

Relation Names

```
Student, Professor, Course, Department, Enrollment
```

Each entity has a set of attributes that define its characteristics. Attributes can be of different types:

- **Key Attribute:** A unique identifier for an entity.

  ```
  Student_ID,  Course_ID,  Department_ID, Professor_ID
  ```

- **Simple Attribute:** A single-valued characteristic (e.g., Name, DOB).

  ```
  Name, DOB, Age, Email, Department_Name, Credits,and more
  ```

- **Composite Attribute:** Attributes that can be divided into sub-parts.

  ```
  Address_Street, Address_City, Address_State
  ```

- **Derived Attribute:** An attribute that is derived from other attributes.

  ```
  Age
  ```

- **Multivalued Attribute:** An attribute that can have multiple values.

  ```
  Phone_no, Professor_Phone
  ```

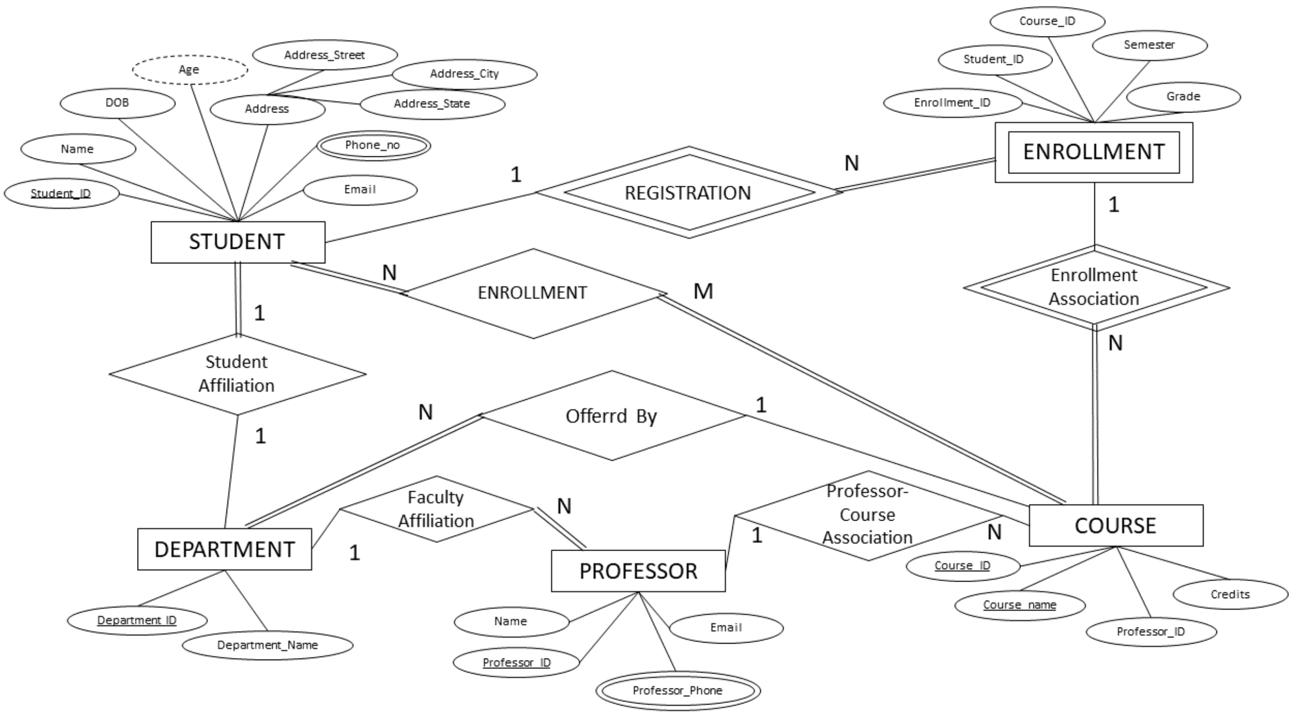


Figure 4.1

3. IDENTIFY RELATIFY BETWEEN ENTITIES

Determine how entities are related to each other. Relationships define associations between entities and specify their cardinality (e.g., One-to-One, One-to-Many, Many-to-Many).

ENTITIES	RELATIONSHIP TYPE	CARDINALITY	IDENTITY
Student ↔ Enrollment	1:N	One student has many enrollments	Weak Entity (Enrollment)
Course ↔ Enrollment	1:N	One course has many enrollments	Weak Entity (Enrollment)
Student ↔ Course	N:M	All students have many course	Strong Entity
Student ↔ Department	N:1	Many students belong to one department	Strong Entity
Professor ↔ Course	1:1	One professor teaches one course	Strong Entity
Department ↔ Professor	1:N	One department has many professors	Strong Entity

4. DRAW THE ER DIAGRAM

Using the identified entities, attributes, and relationships, construct an ER diagram. (Shown as figure 4.1)

- **Entities** are represented as rectangles.
- **Attributes** are represented as ovals connected to their entities.
- **Relationships** are represented as diamonds connecting entities.
- **Primary keys** are underlined, and foreign keys are marked accordingly

---

V. ER Model to Relational Mapping

Step 1: IDENTIFY STRONG ENTITIES

- Strong entities have a unique primary key and exist independently in the database. In our ER model, **Student, Department, Course, and Professor** are strong entities.
- Each of these entities is mapped to a relational table, where all simple attributes are included, and one attribute is chosen as the primary key.

Student:

Student_ID	Student _Name	DOB	Age	Address_Street	Address_City	Address_State	Email

Department:

Department_ID	Department_Name

Course:

Course_ID	Professor_ID	Course_Name	Credits

Professor:

Professor_ID	Professor_ Name	Professor _Email

Step 2: IDENTIFY WEAK ENTITIES

- Weak entities do not have a primary key of their own and depend on a strong entity for identification.
- In this model, **Enrollment** is a weak entity that represents the many-to-many relationship between Student and Course.
- A primary key (Enrollment_ID) is assigned, and foreign keys reference Student and Course.

Student:

Student_ID	Student _Name	DOB	Age	Address_Street	Address_City	Address_State	Email

Course:

Course_ID	Professor_ID	Course_Name	Credits

Enrollment:

Enrollment _ID	Student_ID	Course_ID	Semester	Grade

Step 3: MAPPING RELATIONSHIPS TO TABLE

a) One-To-One Relationship:

- A **one-to-one (1:1) relationship** occurs when a single record in one entity is associated with only one record in another entity, and vice versa.
- This type of relationship is used when an entity requires additional attributes that are rarely used or when data needs to be separated for security or efficiency purposes.
- In an ER diagram, this relationship is represented by a **straight line connecting two entities with "1" on both ends**.

Student:

Student_ID	Student _Name	DOB	Age	Address_Street	Address_City	Address_State	Email

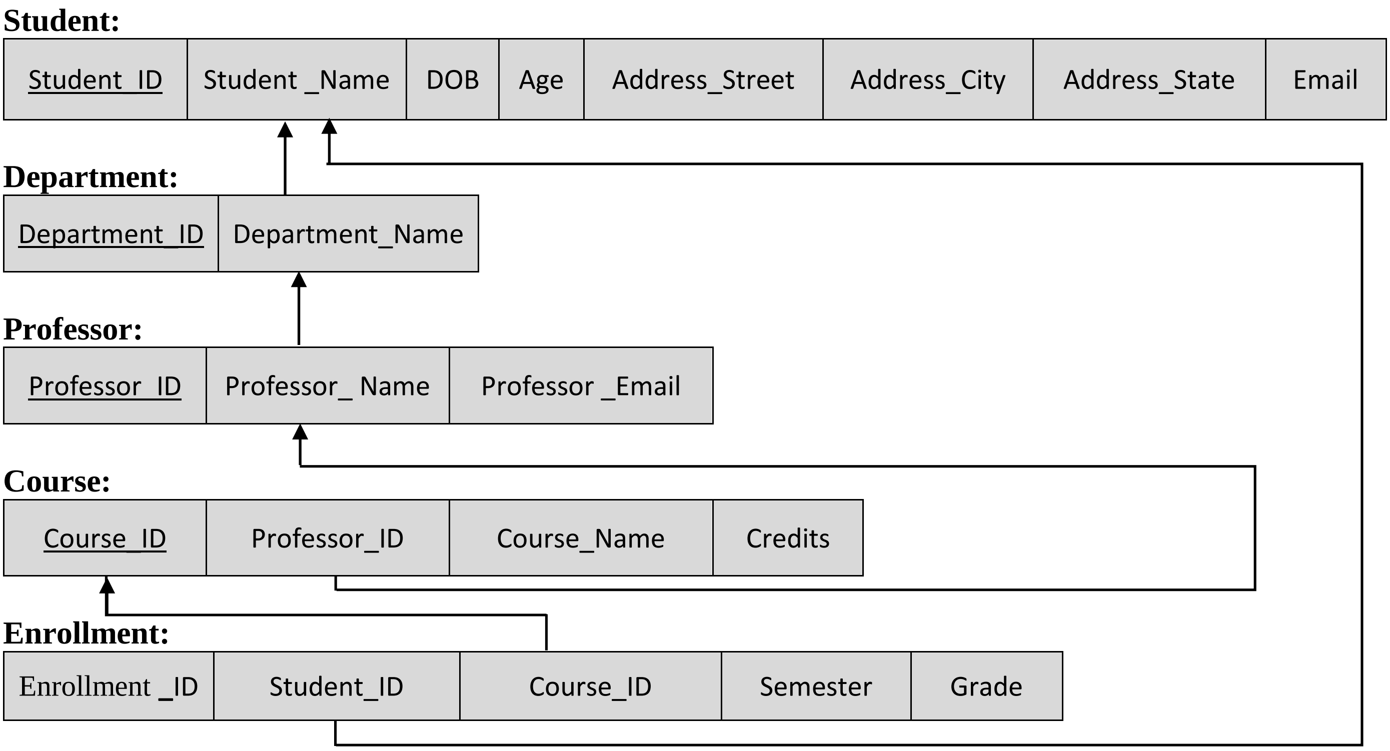
Department:

Department_ID	Department_Name

---

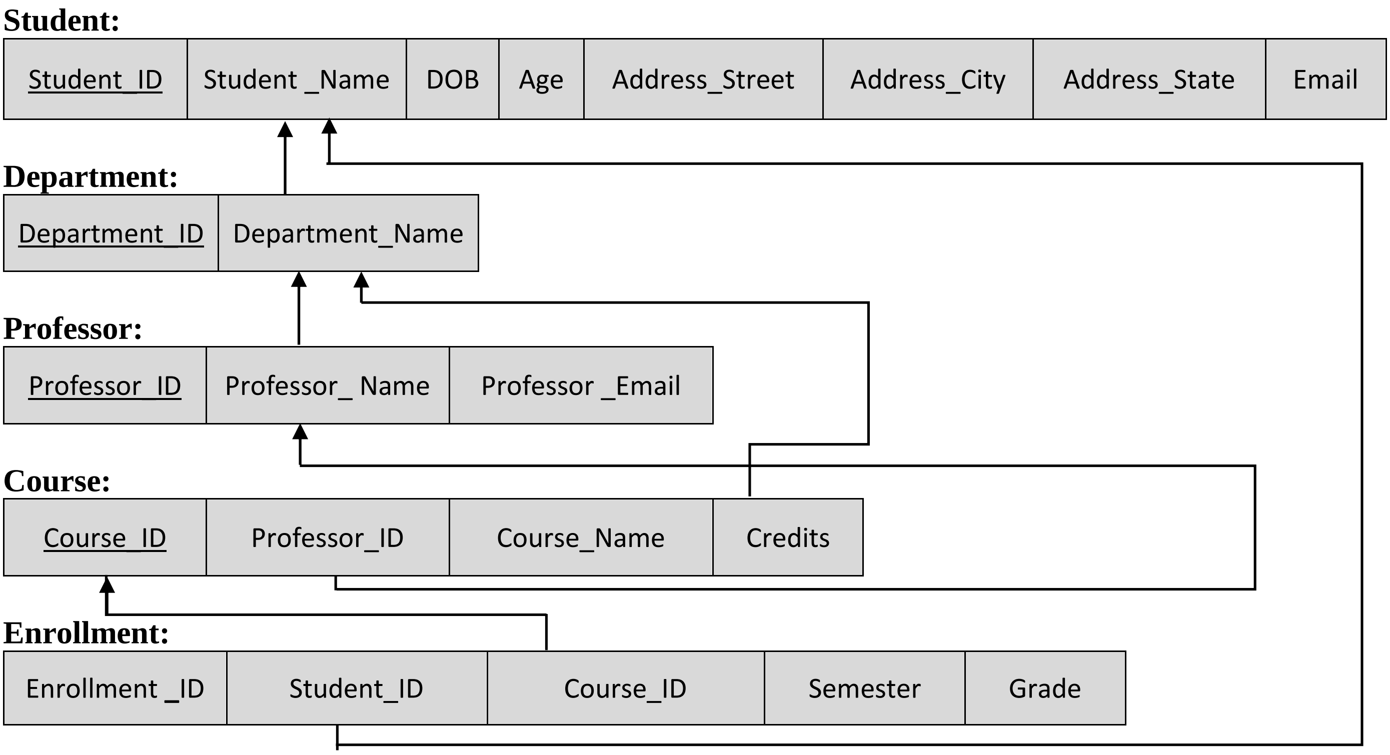
b) One-To-Many Relationships:

- A **one-to-many (1:N) relationship** occurs when a single record in one entity is associated with multiple records in another entity, but each record in the second entity is associated with only one record in the first entity.
- This type of relationship is used when an entity plays a **parent role** and another entity serves as its **child**, allowing multiple dependent records to exist under a single primary entity.
- In an **ER diagram**, this relationship is represented by a **straight line connecting two entities**, with **"1" on one end and "N" on the other** to indicate that one record in the first entity can relate to multiple records in the second entity.
- A **student** can have multiple **enrollments**, but each **enrollment** belongs to only one **student**.
- A **course** can have multiple **enrollments**, but each **enrollment** is linked to only one **course**.
- A **professor** can teach multiple **courses**, but each **course** is taught by only one **professor**.
- A **department** can have multiple **professors**, but each **professor** belongs to only one **department**.



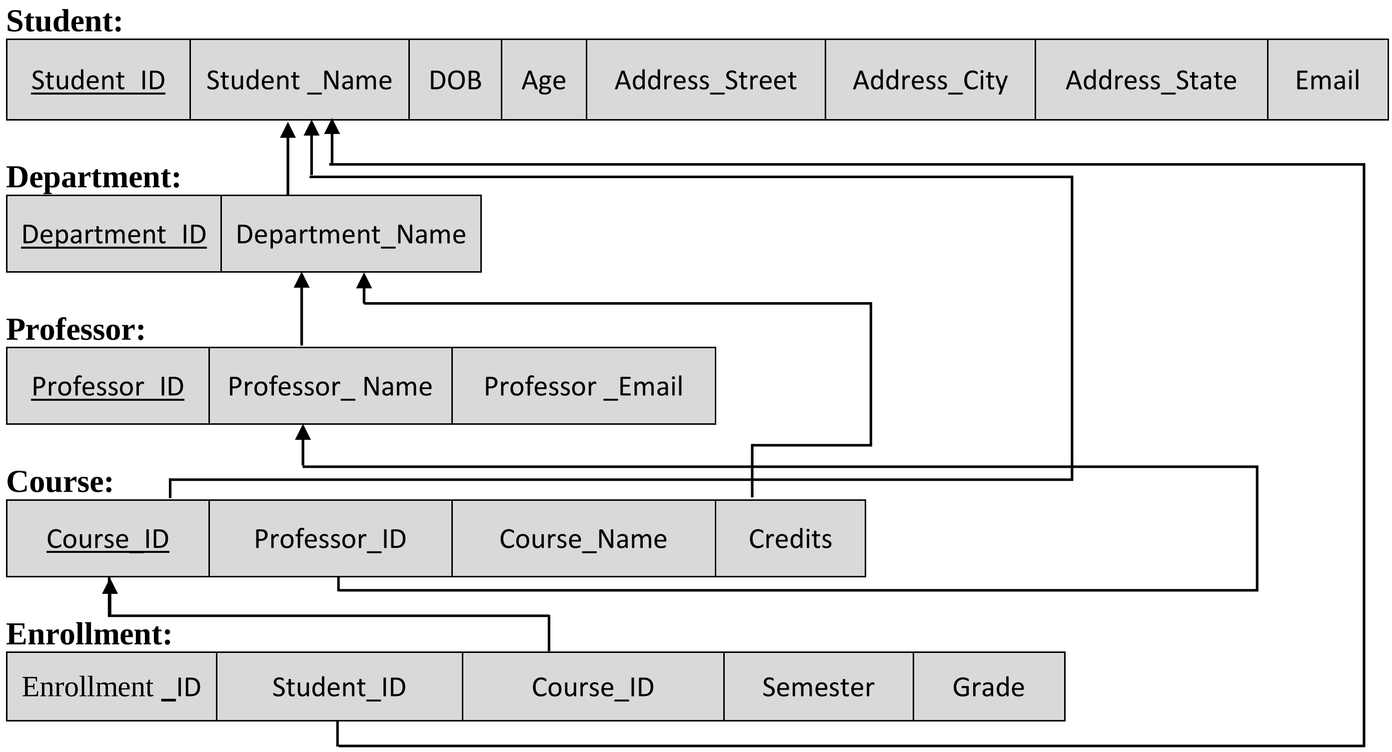
c) Many-To-One Relationship

- A **many-to-one (N:1) relationship** occurs when multiple records in one entity are associated with a single record in another entity, but each record in the second entity is associated with multiple records in the first entity.
- This type of relationship is used when an entity serves as a dependent or child, while another entity acts as a parent, allowing multiple records to be linked to a single record in the related entity.
- In an **ER diagram**, this relationship is represented by a **straight line** connecting two entities, with **"N" on one end and "1" on the other**, indicating that multiple records in the first entity map to a single record in the second entity.
- Multiple **courses** belong to one **department**, but each **course** is assigned to only **one department**.



d) Many-To-Many Relationship

- A **many-to-many (M:N) relationship** occurs when multiple records in one entity are associated with multiple records in another entity, meaning each record in the first entity can relate to multiple records in the second entity, and vice versa.
- This type of relationship is used when two entities have a **mutual association**, where multiple instances of one entity correspond to multiple instances of the other.
- In an **ER diagram**, this relationship is represented by **two entities connected through a bridge (junction) table**, which contains foreign keys referencing both entities.
- A **student** can enroll in **multiple courses**, and a **course** can have **multiple students enrolled** in it.



Step 4: HANDLE MULTIVALUED ATTRIBUTES

- A **multi-valued attribute** occurs when an entity can have **multiple values** for a single attribute, meaning it cannot be stored as a single atomic value in a relational database.
- To handle this, a **separate table** is created for the multi-valued attribute, which includes a foreign key referencing the main entity.
- A **student** can have **multiple phone numbers**, so we create a separate table to store them.
- A **professor** can have **multiple contact numbers**, requiring a separate table to store them.

Student:

Student_ID	Student _Name	DOB	Age	Address_Street	Address_City	Address_State	Email

Student_Phone_No:

Student_ID	Phone_No

Department:

Department_ID	Department_Name

Professor:

Professor_ID	Professor_ Name	Professor _Email

Course:

Course_ID	Professor_ID	Course_Name	Credits

Enrollment:

Enrollment _ID	Student_ID	Course_ID	Semester	Grade

Step 5: MANAGE DERIVED ATTRIBUTES

- Derived attributes are not physically stored in tables but computed dynamically using SQL queries. For example, instead of storing Age, it can be derived from the DOB attribute using a query like YEAR(CURRENT_DATE) - YEAR(DOB). This approach minimizes redundancy and improves data accuracy

---

VI. DATABASE IMPLEMENATION

SQL QUERIES FOR TABLE CREATION

a) The **Student** table is designed to store essential student information with a well-structured schema. The **`Student_ID`** serves as the **primary key**, ensuring each student has a unique identifier. Basic attributes such as **`Name`** and **`DOB`** are included, with **`Age`** being a **derived attribute** calculated from the date of birth. The **`Email`** field is enforced with a **unique constraint** to prevent duplicates. Additionally, the **address** is stored as a **composite attribute**, divided into **street, city, state, and zip code** for better normalization and data organization. This structure ensures data consistency, accuracy, and efficient retrieval.

```
Sql

CREATE TABLE Student (
    Student_ID INT PRIMARY KEY,          -- Primary Key (Unique
Identifier)
    Name VARCHAR(50) NOT NULL,            -- Simple Attribute
    DOB DATE NOT NULL,                    -- Simple Attribute
    Age INT GENERATED ALWAYS AS (YEAR(CURRENT_DATE) - YEAR(DOB)), --
Derived Attribute
    Email VARCHAR(100) UNIQUE NOT NULL,  -- Unique Constraint
    Address_Street VARCHAR(100),          -- Composite Attribute (Part 1)
    Address_City VARCHAR(50),             -- Composite Attribute (Part 2)
    Address_State VARCHAR(50),            -- Composite Attribute (Part 3)
    Address_Zip_Code VARCHAR(10)          -- Composite Attribute (Part 4)
);
```

---

b) The **Student_Phone** table is created to handle the **multi-valued attribute** of phone numbers, allowing each student to have multiple contact numbers while maintaining **database normalization**. The **`Student_ID`** serves as a **foreign key**, linking each phone number to a specific student, ensuring referential integrity. The **composite primary key (`Student_ID, Phone_Number`)** prevents duplicate phone entries for the same student. Additionally, the **`ON DELETE CASCADE`** constraint ensures that if a student record is deleted, all associated phone numbers are automatically removed, maintaining data consistency. This structure efficiently organizes multi-valued attributes while preserving database integrity.

```
sql

CREATE TABLE Student_Phone (
    Student_ID INT,                       -- Foreign Key to Student
    Phone_Number VARCHAR(15),             -- Multi-Valued Attribute
    PRIMARY KEY (Student_ID, Phone_Number),
    FOREIGN KEY (Student_ID) REFERENCES Student(Student_ID) ON DELETE
CASCADE
);
```

---

c) The `Course` table is structured to store details about academic courses while ensuring data integrity and consistency. The **`Course_ID`** serves as the `primary key`, uniquely identifying each course. The **`Course_Name`** is a `simple attribute` that holds the course title and is marked as `NOT NULL` to prevent missing values. The **`Credits`** attribute includes a `CHECK constraint` to ensure that only positive values are stored. The **`Professor_ID`** acts as a **foreign key**, establishing a relationship with the **Professor** table, linking each course to an assigned professor.

```
sql

CREATE TABLE Course (
    Course_ID INT PRIMARY KEY,            -- Primary Key
    Course_Name VARCHAR(50) NOT NULL,     -- Simple Attribute
    Credits INT CHECK (Credits > 0),      -- Constraint: Must be greater
than 0
    Professor_ID INT,                     -- Foreign Key to Professor
    FOREIGN KEY (Professor_ID) REFERENCES Professor(Professor_ID)
);
```

---

d) The **Professor** table is designed to store faculty details with a structured schema ensuring data integrity. The **`Professor_ID`** serves as the **primary key**, uniquely identifying each professor. The **`Name`** is a **simple attribute** marked as **NOT NULL** to ensure that every professor has a recorded name. The **`Email`** field is enforced with a **unique constraint**, preventing duplicate entries and ensuring each professor has a distinct email address.

```
sql

CREATE TABLE Professor (
    Professor_ID INT PRIMARY KEY,         -- Primary Key
    Name VARCHAR(50) NOT NULL,            -- Simple Attribute
    Email VARCHAR(100) UNIQUE NOT NULL    -- Unique Constraint
);
```

---

e) The **Professor_Phone** table is designed to handle the **multi-valued attribute** for professor contact numbers while maintaining **database normalization**. The **`Professor_ID`** acts as a **foreign key**, linking multiple phone numbers to a single professor while ensuring referential integrity. The **composite primary key (`Professor_ID, Phone_Number`)** prevents duplicate phone entries for the same professor. The **`ON DELETE CASCADE`** constraint ensures that if a professor record is deleted, all associated phone numbers are automatically removed.

```
sql

CREATE TABLE Professor_Phone (
    Professor_ID INT,                     -- Foreign Key to Professor
    Phone_Number VARCHAR(15),             -- Multi-Valued Attribute
    PRIMARY KEY (Professor_ID, Phone_Number),
    FOREIGN KEY (Professor_ID) REFERENCES Professor(Professor_ID) ON
DELETE CASCADE
);
```

---

f) The **Department** table is structured to store information about various academic departments while ensuring data integrity. The **`Department_ID`** serves as the **primary key**, uniquely identifying each department. The **`Department_Name`** is a **unique attribute**, preventing duplicate department names and ensuring each entry is distinct. The **NOT NULL** constraint ensures that every department has a valid name.

```sql
CREATE TABLE Department ( Department_ID INT PRIMARY KEY, -- Primary Key
Department_Name VARCHAR(50) UNIQUE NOT NULL -- Unique Attribute
);
```

---

g) The **Enrollment** table is designed to manage student course registrations while ensuring referential integrity. The **`Enrollment_ID`** serves as the **primary key**, uniquely identifying each enrollment record. The **`Student_ID`** and **`Course_ID`** are **foreign keys**, establishing relationships with the **Student** and **Course** tables, respectively. The **`Semester`** is a **simple attribute** marked as **NOT NULL**, ensuring valid enrollment records. The **`Grade`** attribute includes a **CHECK constraint**, restricting values to **A, B, C, D, or F** for data consistency. The **`ON DELETE CASCADE`** ensures that if a student or course is deleted, the related enrollment records are automatically removed, maintaining database integrity.

```sql
CREATE TABLE Enrollment (
    Enrollment_ID INT PRIMARY KEY,       -- Primary Key
    Student_ID INT,                      -- Foreign Key to Student
    Course_ID INT,                        -- Foreign Key to Course
    Semester VARCHAR(10) NOT NULL,       -- Simple Attribute
    Grade CHAR(1) CHECK (Grade IN ('A', 'B', 'C', 'D', 'F')), --
Constraint
    FOREIGN KEY (Student_ID) REFERENCES Student(Student_ID) ON DELETE
CASCADE,
    FOREIGN KEY (Course_ID) REFERENCES Course(Course_ID) ON DELETE
CASCADE
);
```

VII. NORMALIZATION FORM

First Normal Form (1NF)

DEFINITION:

First Normal Form (**1NF**) ensures that all attributes contain **atomic (indivisible) values** and that there are **no repeating groups or multi-valued attributes** in a table.

Steps to Achieve 1NF in Our Database:

1. **Identify multi-valued attributes** – In our case, the **`Student`** and **`Professor`** tables contain multiple phone numbers.
2. **Remove multi-valued attributes** – Create **separate tables** for phone numbers, linking them with the original entity using a **foreign key**.
3. **Ensure each column has atomic values** – Every attribute should store a single, indivisible value.

Before 1NF (Violating 1NF - Multi-Valued Attribute Present)

The **Phone_Numbers** column contains multiple values (comma-separated), which violates **1NF** since an attribute should hold only **atomic values**.

Student_id	Student_Name	DOB	Student_Phone	Email
1	Asmath	2001-03-15	9876543210, 8765432109	asmath@gmail.com
2	Dhanush	2000-07-22	9123456789, 9234567890	dhanush@gmail.com
3	Madhan	1999-12-05	9345678901, 9456789012	madhan@gmail.com
4	Nandhu	2002-05-19	9567890123, 9678901234	nandhu@gmail.com
5	Vikey	1998-09-10	9789012345, 9890123456	vikey@gmail.com

After Applying 1NF (Multi-Valued Attribute Removed)

To achieve **1NF**, we **remove the multi-valued attribute** and create a **separate table** for phone numbers.

```sql
CREATE TABLE Professor_Phone (
    Professor_ID INT,                         -- Foreign Key to Professor
    Phone_Number VARCHAR(15),                 -- Multi-Valued Attribute
    PRIMARY KEY (Professor_ID, Phone_Number),
    FOREIGN KEY (Professor_ID) REFERENCES Professor(Professor_ID) ON
DELETE CASCADE
);
```

```sql
CREATE TABLE Student_Phone (
    Student_ID INT,                         -- Foreign Key to Student
    Phone_Number VARCHAR(15),               -- Multi-Valued Attribute
    PRIMARY KEY (Student_ID, Phone_Number),
    FOREIGN KEY (Student_ID) REFERENCES Student(Student_ID) ON DELETE CASCADE
);
```

Student Table (Atomic Attributes)

Student_ID	Name	DOB	Email
1	Asmath	2001-03-15	asmath@email.com
2	Dhanush	2000-07-22	dhanush@email.com
3	Madhan	1999-12-05	madhan@email.com
4	Nandhu	2002-05-19	nandhu@email.com
5	Vikey	1998-09-10	vikey@email.com

Student_Phone Table (Multi-Valued Attributes as Separate Rows)

Student_ID	Phone_Number
1	9876543210
1	8765432109
2	9123456789
2	9234567890
3	9345678901
3	9456789012
4	9567890123
4	9678901234
5	9789012345
5	9890123456

VIII. JOIN OPERATION

1. INNER JOIN (Get students with enrolled courses)

- The **INNER JOIN** retrieves students who are enrolled in at least one course.
- Only records where there is a match in both **Student_After_1NF** and **Enrollment** tables will be displayed

```sql
SELECT s.Student_ID, s.Name, e.Course_ID, c.Course_Name, e.Grade
FROM Student s
INNER JOIN Enrollment e ON s.Student_ID = e.Student_ID
INNER JOIN Course c ON e.Course_ID = c.Course_ID;
```

Output:

Student_ID	Name	Course_ID	Course_Name	Grade
1	Asmath	101	Database Systems	A
1	Asmath	103	Computer Networks	B
2	Dhanush	102	Operating Systems	A
3	Madhan	101	Database Systems	C
4	Nandhu	105	Machine Learning	B
5	Vikey	104	Artificial Intelligence	A

---

2. LEFT JOIN (List all students with or without enrollment)

- The **LEFT JOIN** returns all students, even those **without enrollments**.
- If a student is not enrolled in any course, the **Course_ID and Course_Name will be NULL**.

```sql
SELECT s.Student_ID, s.Name, e.Course_ID, c.Course_Name
FROM Student s
LEFT JOIN Enrollment e ON s.Student_ID = e.Student_ID
LEFT JOIN Course c ON e.Course_ID = c.Course_ID;
```

Output:

Student_ID	Name	Course_ID	Course_Name
1	Asmath	101	Database Systems
1	Asmath	103	Computer Networks
2	Dhanush	102	Operating Systems
3	Madhan	101	Database Systems
4	Nandhu	105	Machine Learning
5	Vikey	104	Artificial Intelligence
6	Ramesh	NULL	NULL

---

3. RIGHT JOIN (List all courses with or without enrolled students)

- The **RIGHT JOIN** ensures that **all courses** are listed, even if there are **no enrolled students**.
- If no student is enrolled in a course, **Student_ID and Name will be NULL**.

```sql
SELECT s.Student_ID, s.Name, c.Course_ID, c.Course_Name
FROM Student s
RIGHT JOIN Enrollment e ON s.Student_ID = e.Student_ID
RIGHT JOIN Course c ON e.Course_ID = c.Course_ID;
```

Output:

Student_ID	Name	Course_ID	Course_Name
1	Asmath	101	Database Systems
1	Asmath	103	Computer Networks
2	Dhanush	102	Operating Systems
3	Madhan	101	Database Systems
4	Nandhu	105	Machine Learning
5	Vikey	104	Artificial Intelligence
NULL	NULL	106	Cloud Computing

---

4. FULL OUTER JOIN (List all students and courses, even if there is no enrollment)

- The **FULL OUTER JOIN** combines both **LEFT JOIN** and **RIGHT JOIN**.
- It includes all **students and courses**, even if **no enrollment exists**.

```sql
SELECT s.Student_ID, s.Name, c.Course_ID, c.Course_Name
FROM Student s
FULL OUTER JOIN Enrollment e ON s.Student_ID = e.Student_ID
FULL OUTER JOIN Course c ON e.Course_ID = c.Course_ID;
```

Output:

Student_ID	Name	Course_ID	Course_Name
1	Asmath	101	Database Systems
1	Asmath	103	Computer Networks
2	Dhanush	102	Operating Systems
3	Madhan	101	Database Systems
4	Nandhu	105	Machine Learning
5	Vikey	104	Artificial Intelligence
6	Ramesh	NULL	NULL
NULL	NULL	106	Cloud Computing

---

5. SELF JOIN (Finding professors who have the same department)

- The **SELF JOIN** is used to **compare values within the same table**.
- Here, we retrieve **pairs of professors** from the same department.

```sql
SELECT p1.Professor_ID AS Prof_1, p1.Name AS Professor_1,
       p2.Professor_ID AS Prof_2, p2.Name AS Professor_2
FROM Professor p1
JOIN Professor p2 ON p1.Professor_ID <> p2.Professor_ID;
```

Output:

Prof_1	Professor_1	Prof_2	Professor_2
201	Dr. Ravi Kumar	202	Dr. Priya Sharma
201	Dr. Ravi Kumar	203	Dr. Ayesha Khan
202	Dr. Priya Sharma	203	Dr. Ayesha Khan
203	Dr. Ayesha Khan	204	Dr. Arjun Patel

---

6. CROSS JOIN (All possible combinations of students and courses)

- The **CROSS JOIN** returns the **Cartesian product** of both tables.

- Every student is paired with **every course** (useful for creating enrollment lists).

```sql
SELECT s.Name, c.Course_Name
FROM Student_After_1NF s
CROSS JOIN Course c;
```

Output:

Name	Course_Name
Asmath	Database Systems
Asmath	Operating Systems
Asmath	Computer Networks
Asmath	Artificial Intelligence
Asmath	Machine Learning
Dhanush	Database Systems
Dhanush	Operating Systems
Dhanush	Computer Networks
Dhanush	Artificial Intelligence
Dhanush	Machine Learning

XI. CONCLUSION

The development of an optimized relational database for student-course enrollment ensures efficient data management, integrity, and retrieval. By implementing an ER model, we accurately define entities, attributes, and relationships while maintaining data consistency. Normalization, specifically First Normal Form (1NF), eliminates redundancy by separating multi-valued attributes into independent tables. Various SQL join operations facilitate seamless data retrieval, enabling meaningful insights into student enrollments, course allocations, and professor assignments. This structured approach improves database efficiency, minimizes redundancy, and enhances query performance, making the system scalable and reliable for academic institutions.

X. REFERENCES

[1] Kumarjai, "ER to Relational Mapping," *Medium*, Available at: https://medium.com/@kumarjai2466/er-to-relational-mapping-ac84b3c9f258.
[2] "Normal Forms in DBMS," *GeeksforGeeks*, Available at: https://www.geeksforgeeks.org/normal-forms-in-dbms.
[3] Hingorani, K., Gittens, D., & Edwards, N. (2017). "Reinforcing Database Concepts by Using Entity Relationship Diagrams (ERD) and Normalization Together for Designing Robust Databases." *Issues in Information Systems, 18*(1), 148-155.
[4] *Lecture 08: Mapping ER Models to Relational Models*, Northeastern University, Available at: https://course.ccs.neu.edu/cs3200sp18s3/ssl/lectures/lecture_08_mapping.pdf

---