CHRISTU JYOTHI
INSTITUTE OF TECHNOLOGY & SCIENCE
Affiliated to JNTUH | Jangaon, Telangana 506167

DEPARTMENT OF

COMPUTER SCIENCE ENGINEERING

REGULATION: R22

DATABASE MANAGEMENT SYSTEM LAB MANUAL (CS407PC)

PREPARED BY:

L SURESH

ASST.PROF.(CSE)

INDEX

: Introduction:

DBMS stands for **Database Management System**. It is a software system designed to manage databases and provide an interface for users to interact with, store, retrieve, and modify data. DBMS ensures that data is stored efficiently, is accessible, and remains consistent and secure.

Key Functions of a DBMS:

1. **Data Storage Management**: Manages how data is stored and accessed in the database.
2. **Data Retrieval**: Provides efficient ways to retrieve data through queries.
3. **Data Security**: Ensures data is protected and accessible only by authorized users.
4. **Data Integrity**: Maintains data accuracy and consistency, preventing data anomalies.
5. **Backup and Recovery**: Ensures that data can be recovered in case of failure.
6. **Concurrency Control**: Manages access to the database by multiple users to prevent conflicts and inconsistencies.
7. **Data Modeling**: Organizes data into structured formats, such as tables, to define relationships among various data elements.

Types of DBMS:

1. **Hierarchical DBMS**: Organizes data in a tree-like structure, with records having a parent-child relationship.
2. **Network DBMS**: Organizes data in a graph-like structure, where each record can have multiple relationships.
3. **Relational DBMS (RDBMS)**: Stores data in tables (relations) and allows complex queries using SQL. Examples: MySQL, PostgreSQL, Oracle.
4. **Object-Oriented DBMS (OODBMS)**: Stores data as objects, similar to object-oriented programming. Example: ObjectDB.
5. **NoSQL DBMS**: Used for handling unstructured data, typically for large-scale and real-time applications. Examples: MongoDB, Cassandra.

Examples of Popular DBMS:

- **MySQL**
- **Oracle DB**
- **PostgreSQL**
- **Microsoft SQL Server**
- **MongoDB**
- **SQLite**

:: Definitions:

- **SQL** stands for **Structured Query Language**. It is a standard programming language used to manage and manipulate relational databases. SQL is used for tasks such as querying, inserting, updating, and deleting data, as well as creating and modifying database structures like tables and indexes.
- **Data** refers to raw facts, figures, or information that can be processed or analysed to gain meaning, insight, or value. In computing and information technology, data can be anything from numbers and text to images, audio, and more. It is the foundational element that is stored, processed, and analysed in various systems, including databases, applications, and data analytics platforms.
- **Information** refers to processed or organized data that is meaningful and useful. While data are raw facts or figures, information is the result of processing, organizing, or analysing data to provide context, relevance, and purpose. Information helps in decision-making, problem-solving, and enhancing knowledge.
- **Database** is a collection of structured data that is stored and managed in a way that makes it easy to access, update, and manipulate. Databases are used to store, organize, and manage large amounts of data efficiently and securely, allowing users and applications to interact with that data using various methods, like querying or transaction processing.
- **Table** in a database is a collection of data organized in a structured format, typically in rows and columns. It is the fundamental unit of data storage in a relational database management system (RDBMS). Each table represents an entity or concept, and the columns within the table represent attributes or properties of that entity.
- **Record(Row/TUPLE)** in a database refers to a single, complete set of related data values in a table. It represents one instance or entry of an entity and is typically organized in a row. Each record contains data for every column defined in the table.
- **Column** in a database refers to a vertical set of data values in a table, where each column represents a specific attribute or field of the records stored in the table. Each column contains data of a specific data type and holds a particular kind of information about the entity described by the table.
- **Entity-Relationship (E-R) Model** is a conceptual framework used to represent the structure of a database. It helps in designing databases by visually depicting the relationships between different entities and their attributes. The model was introduced by **Peter Chen** in 1976 and is widely used in database design, especially in the early stages of creating a database.
- **Entity** is a distinct object or thing in the real world that can be identified and described. It is often represented as a rectangle in an ER diagram.
  <u>Example:</u> *Customer, Order, Product are all entities.*
- **Attributes** describe the properties or characteristics of an entity. Attributes are usually represented as ovals connected to their corresponding entities in the ER diagram.
  <u>Example:</u> *A Customer entity might have attributes like CustomerID, FirstName, LastName, Email.*
- **Relationship** represents how two or more entities are related to each other. It is represented by a diamond in the ER diagram.
  <u>Example:</u> *A Customer places an Order, so there is a relationship between the Customer and Order entities.*

WEEK-1
Concept Design With E-R Model

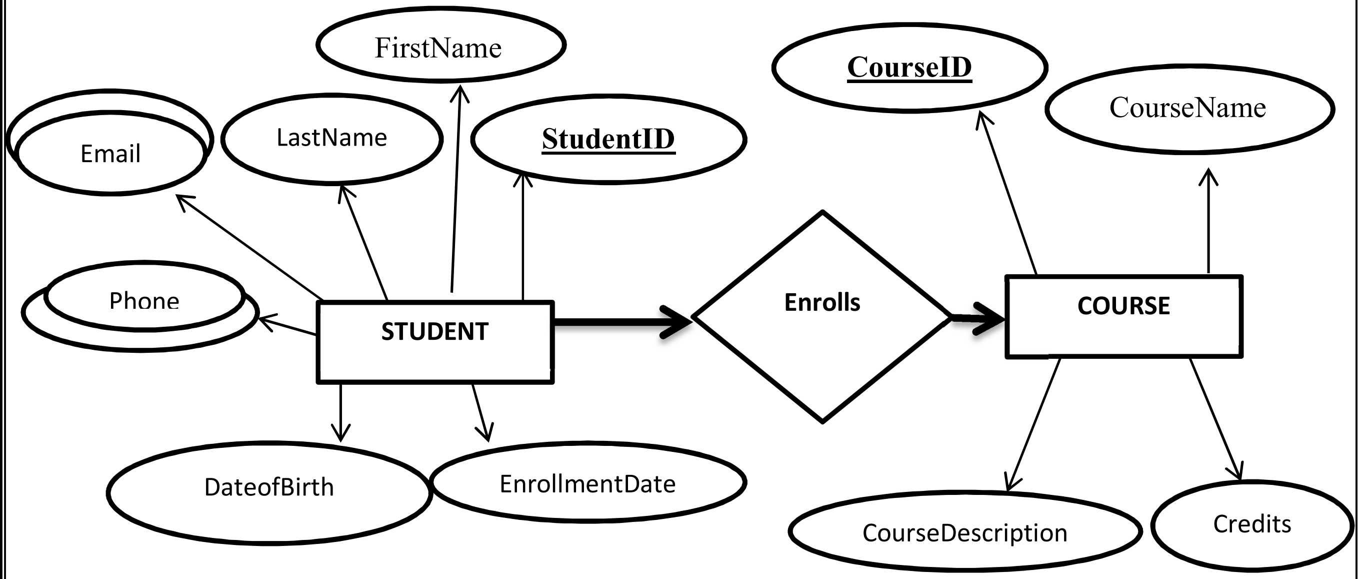
AIM: Analyze the problem and come with the entities in it. Identify what Data has to be persisted in the databases.

The **Student Management System** includes entities like **Student**, **Course**, **Class**, **Faculty**, **Exams**, and **Results**, with relationships defined between them. Below is the complete E-R model, including entities, attributes, relationships, and SQL queries.

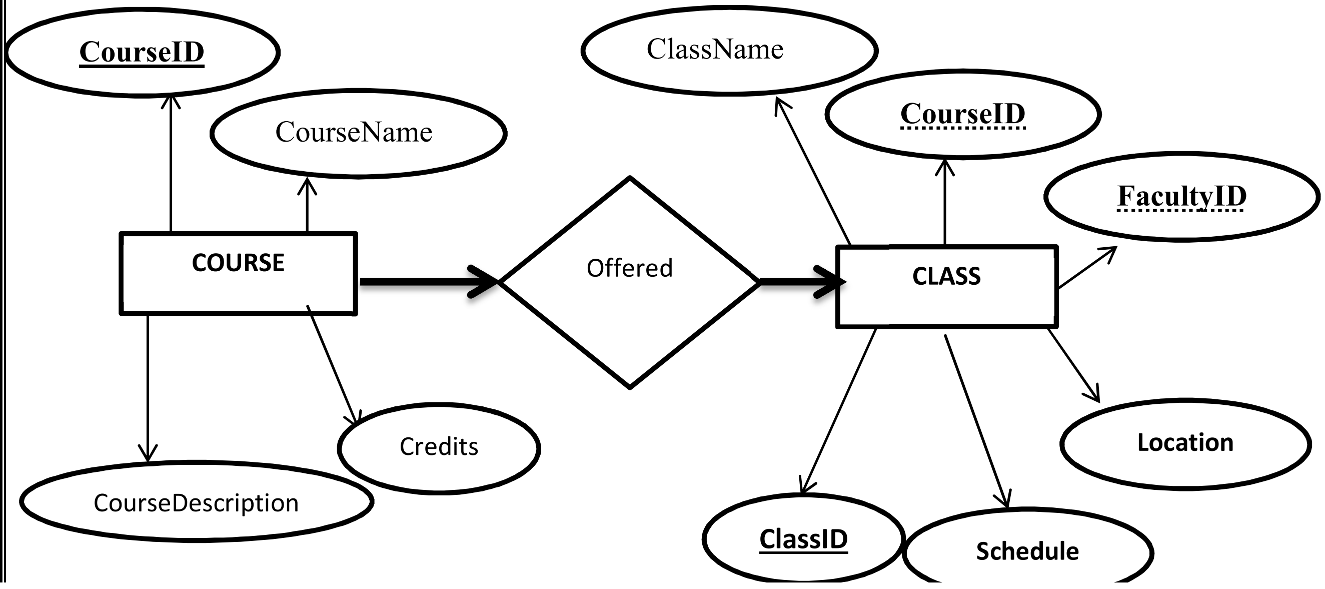
E-R Diagram:

Below is the conceptual view of the relationships between the entities:

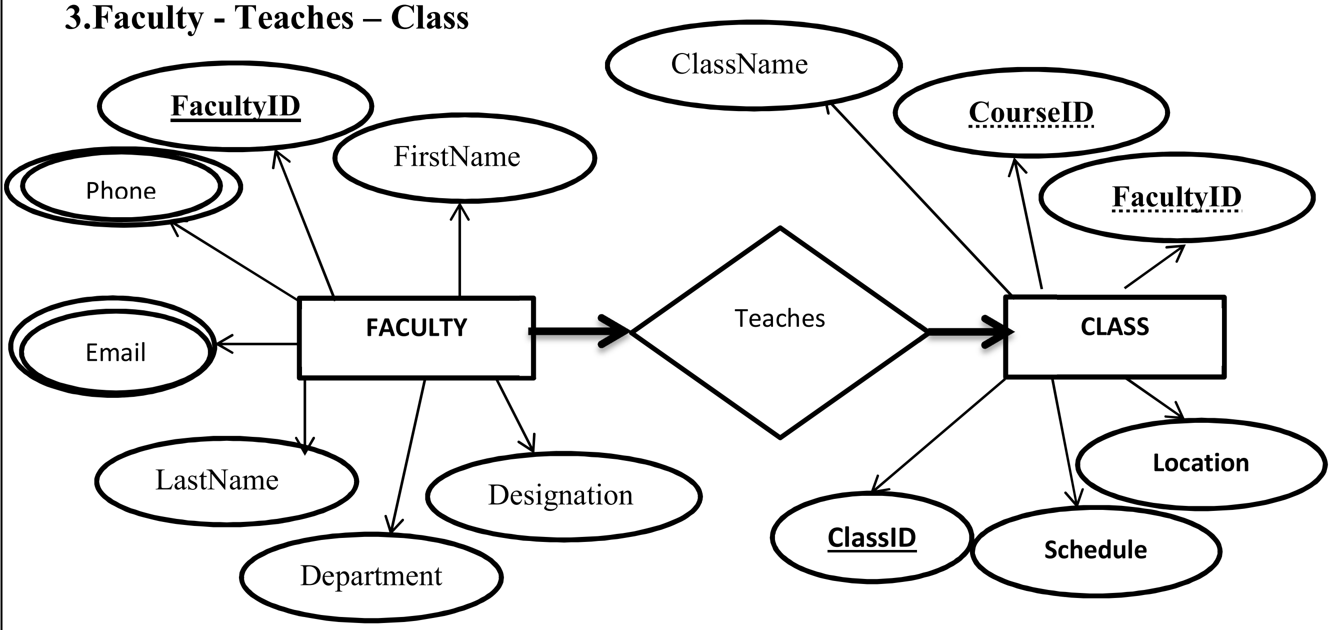
1. **Student - Enrolls in - Course**



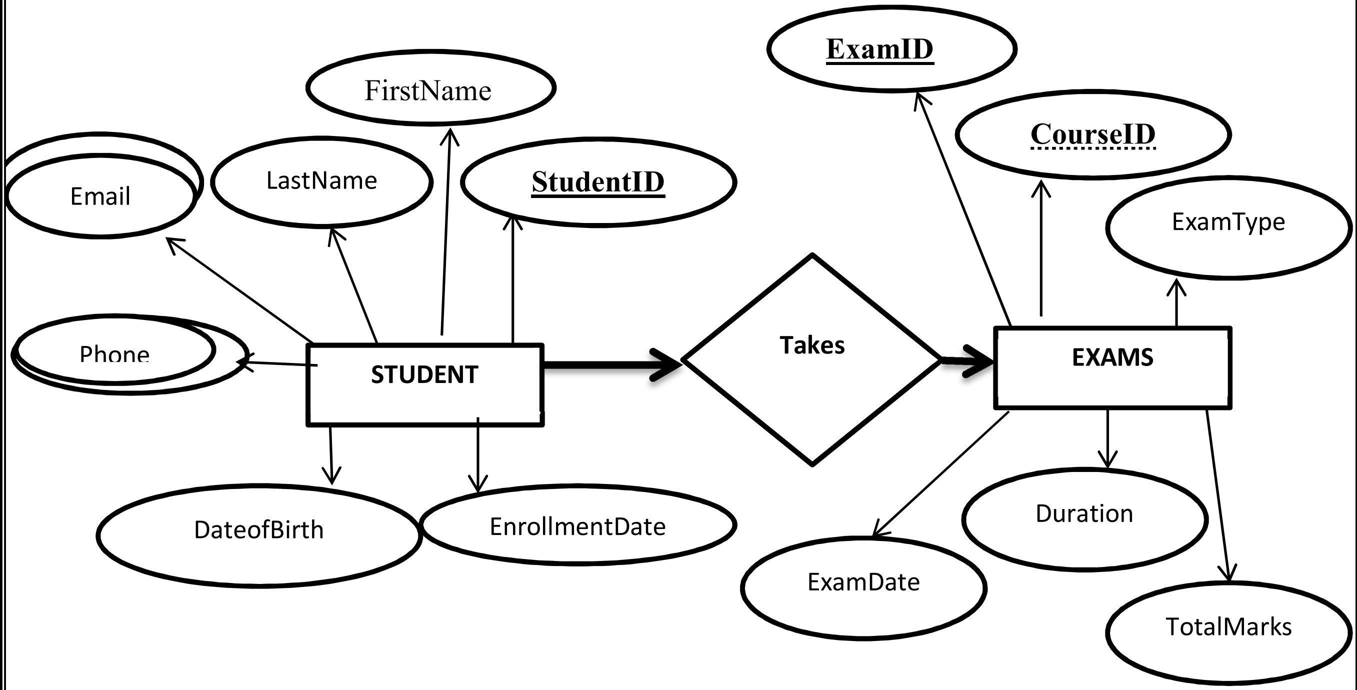
2. **Course - Offered in - Class**

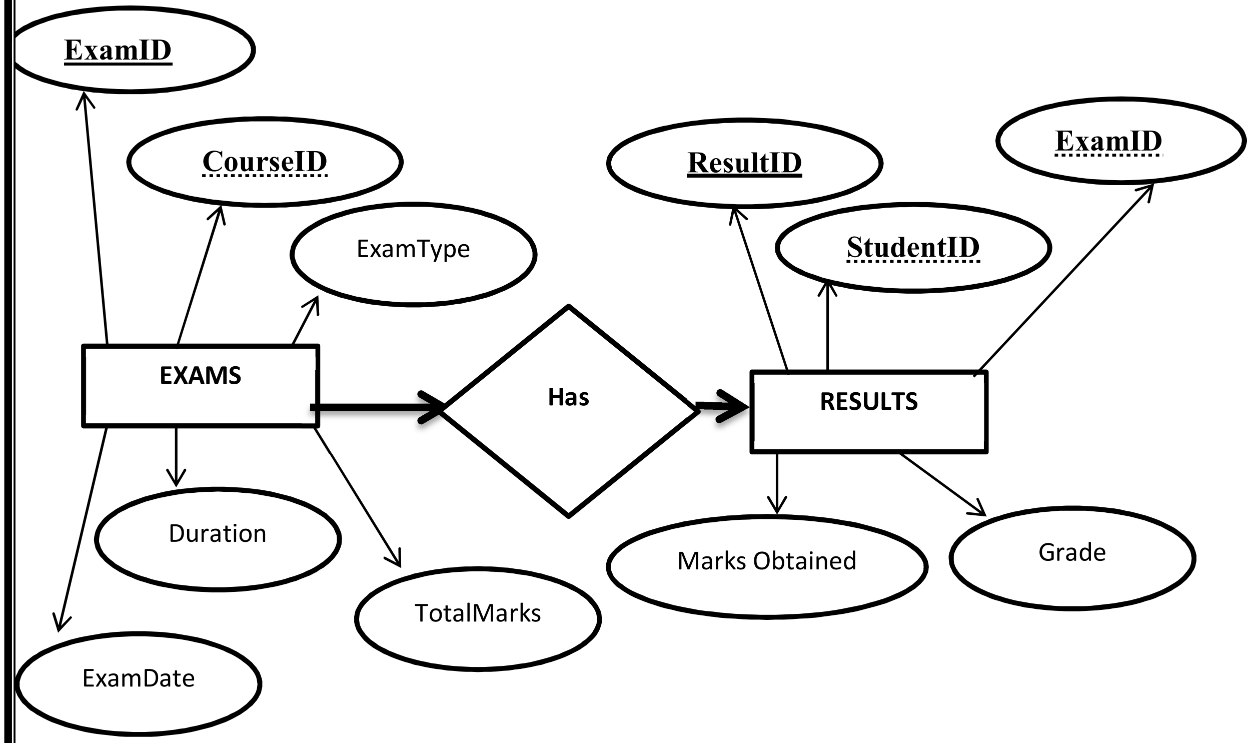


3.Faculty - Teaches – Class



4.Student - Takes – Exam




ExamID
CourseID
ExamType
EXAMS
Duration
TotalMarks
ExamDate
Has
ResultID
StudentID
ExamID
RESULTS
Marks Obtained
Grade

WEEK-2

RELATIONAL MODEL

AIM: Creating Entities and Relationships for Student Management ER-Diagram

The relational model represents how data is stored in **Relational Databases**. A relational database consists of a collection of tables each of which is assigned a unique name.

Entities and Attributes:

1. **Student**
   - StudentID (Primary Key)
   - FirstName
   - LastName
   - Email
   - Phone
   - DateOfBirth
   - EnrollmentDate
2. **Course**
   - CourseID (Primary Key)
   - CourseName
   - CourseDescription
   - Credits
3. **Class**
   - ClassID (Primary Key)
   - ClassName
   - Schedule
   - Location
   - FacultyID (Foreign Key to Faculty)
   - CourseID (Foreign Key to Course)
4. **Faculty**
   - FacultyID (Primary Key)
   - FirstName
   - LastName
   - Email
   - Phone
   - Department
   - Designation
5. **Exam**
   - ExamID (Primary Key)
   - CourseID (Foreign Key to Course)
   - ExamDate
   - ExamType (e.g., Midterm, Final)
   - Duration
   - TotalMarks
6. **Results**
   - ResultID (Primary Key)
   - StudentID (Foreign Key to Student)
   - ExamID (Foreign Key to Exam)
   - MarksObtained
   - Grade

Relationships:

1. **Student - Enrolls in - Course**
   - A **student** can enroll in many **courses**, and a **course** can have many **students** (many-to-many relationship).
   - We need a junction table to store this relationship.
2. **Course - Offered in - Class**
   - A **course** is offered in multiple **classes**, and each **class** is for a single **course** (one-to-many relationship).
3. **Faculty - Teaches - Class**
   - A **faculty** teaches many **classes**, and each **class** is taught by one **faculty** (one-to-many relationship).
4. **Student - Takes - Exam**
   - A **student** can take many **exams**, and each **exam** is taken by many **students** (many-to-many relationship).
   - A junction table is required here as well.
5. **Exam - Has - Results**
   - An **exam** has **results** for multiple **students** (one-to-many relationship).

WEEK – 3
NORMALIZATION

Normalization is a process in **DBMS** that organizes data efficiently by eliminating redundancy and ensuring data integrity. It involves dividing large tables into smaller, related tables to reduce duplication and improve consistency.

Let's apply **normalization** to a **Student Management System** step by step.

Normalization of the Student Table

We have a **Student Table** with the following attributes:

- **StudentID** (Primary Key)
- **FirstName**
- **LastName**
- **Email**
- **Phone**
- **DateOfBirth**
- **EnrollmentDate**

Let's apply **Normalization** step by step.

Step 1: Unnormalized Form (UNF)

The **Unnormalized Form (UNF)** contains redundant and unstructured data. If students have multiple phone numbers or emails stored in the same column, it violates atomicity.

Unnormalized Table (UNF)

StudentID	FirstName	LastName	Email	Phone	DateOfBirth	EnrollmentDate
101	John	Doe	john@gmail.com	9876543210, 9876504321	2000-05-10	2023-06-15
102	Emma	Smith	emma@yahoo.com	9876543211	2001-08-20	2023-06-16
103	Alex	Brown	alex@gmail.com, alex@outlook.com	9876543212	2002-02-25	2023-06-17

Problems in UNF:

❌**Non-atomic values** (multiple emails and phone numbers in a single column).
❌**Data redundancy** (Email and Phone numbers are stored together).

Step 2: First Normal Form (1NF)

Rules of 1NF:

✔ Ensure **atomicity** (each column should have a single value).
✔ Ensure **uniqueness** (each row should have a unique identifier).

1NF Table

StudentID	**FirstName**	**LastName**	**Email**	**Phone**	**DateOfBirth**	**EnrollmentDate**
101	John	Doe	john@gmail.com	9876543210	2000-05-10	2023-06-15
101	John	Doe	NULL	9876504321	2000-05-10	2023-06-15
102	Emma	Smith	emma@yahoo.com	9876543211	2001-08-20	2023-06-16
103	Alex	Brown	alex@gmail.com	9876543212	2002-02-25	2023-06-17
103	Alex	Brown	alex@outlook.com	NULL	2002-02-25	2023-06-17

✅**Each field contains atomic values** (single phone, single email per row).
❌**Still redundant (Student details are repeated for multiple emails/phones).**

Step 3: Second Normal Form (2NF)

Rules of 2NF:

✔ The table must be in **1NF**.
✔ **Remove partial dependencies** (every non-key attribute should depend on the whole primary key).

Decomposing into Two Tables:

Students Table

StudentID	FirstName	LastName	DateOfBirth	EnrollmentDate
101	John	Doe	2000-05-10	2023-06-15
102	Emma	Smith	2001-08-20	2023-06-16
103	Alex	Brown	2002-02-25	2023-06-17

Student_Contacts Table

ContactID	StudentID	Email	Phone
1	101	john@gmail.com	9876543210
2	101	NULL	9876504321
3	102	emma@yahoo.com	9876543211
4	103	alex@gmail.com	9876543212
5	103	alex@outlook.com	NULL

✅**Removed partial dependency (Contacts stored separately).**
❌**Still redundant (Email and Phone stored in the same column).**

Step 4: Third Normal Form (3NF)

Rules of 3NF:

✔ The table must be in **2NF**.
✔ **Remove transitive dependencies** (non-key attributes should not depend on other non-key attributes).

Final Decomposition into Three Tables:

Students Table

StudentID	**FirstName**	**LastName**	**DateOfBirth**	**EnrollmentDate**
101	John	Doe	2000-05-10	2023-06-15
102	Emma	Smith	2001-08-20	2023-06-16
103	Alex	Brown	2002-02-25	2023-06-17

Student_Emails Table

EmailID	**StudentID**	**Email**
1	101	john@gmail.com
2	103	alex@gmail.com
3	103	alex@outlook.com
4	102	emma@yahoo.com

Student_Phones Table

PhoneID	**StudentID**	**Phone**
1	101	9876543210
2	101	9876504321
3	102	9876543211
4	103	9876543212

✅**No transitive dependencies (Phones and Emails stored separately).**
✅**Fully normalized (Minimal redundancy, optimal storage).**

Final Optimized Database Schema (3NF)

1. **Students (StudentID, FirstName, LastName, DateOfBirth, EnrollmentDate)**
2. **Student_Emails (EmailID, StudentID, Email)**
3. **Student_Phones (PhoneID, StudentID, Phone)**

Week 4: Practicing DDL Commands

AIM: Practicing DDL Commands (CREATE, ALTER):
Solutions: Creating "**Student**, **Course**, **Class**, **Faculty**, **Exams**, and **Results**" Tables using CREATE command.

DDL Commands are:

- CREATE
- ALTER
- DROP
- TRUNCATE
- RENAME

SQL Schema Creation:

1. **Student Table**:

```
CREATE TABLE Student (
   StudentID INT PRIMARY KEY,
   FirstName VARCHAR(50),
   LastName VARCHAR(50),
   Email VARCHAR(100),
   Phone VARCHAR(15),
   DateOfBirth DATE,
   EnrollmentDate DATE
);
```

2. **Course Table**:

```
CREATE TABLE Course (
   CourseID INT PRIMARY KEY,
   CourseName VARCHAR(100),
   CourseDescription TEXT,
   Credits INT
);
```

3. **Class Table**:

```
CREATE TABLE Class (
   ClassID INT PRIMARY KEY,
   ClassName VARCHAR(50),
   Schedule DATETIME,
   Location VARCHAR(100),
   FacultyID INT,
   CourseID INT,
   FOREIGN KEY (FacultyID) REFERENCES Faculty(FacultyID),
   FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
```

);

4. **Faculty Table**:

```
CREATE TABLE Faculty (
   FacultyID INT PRIMARY KEY,
   FirstName VARCHAR(50),
   LastName VARCHAR(50),
   Email VARCHAR(100),
   Phone VARCHAR(15),
   Department VARCHAR(50),
   Designation VARCHAR(50)
);
```

5. **Exam Table**:

```
CREATE TABLE Exam (
   ExamID INT PRIMARY KEY,
   CourseID INT,
   ExamDate DATE,
   ExamType VARCHAR(50),
   Duration INT,
   TotalMarks INT,
   FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

6. **Results Table**:

```
CREATE TABLE Results (
   ResultID INT PRIMARY KEY,
   StudentID INT,
   ExamID INT,
   FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
   FOREIGN KEY (ExamID) REFERENCES Exam(ExamID)
);
```

7. **Staff Table**:

```
CREATE TABLE Staff (
   StaffID INT,
   ExamID INT,
);
```

8. **Enrollment Table** (Junction Table for many-to-many relationship between Student and Course):

```
CREATE TABLE Enrollment (
   StudentID INT,
   CourseID INT,
   EnrollmentDate DATE,
   PRIMARY KEY (StudentID, CourseID),
   FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
   FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
);
```

8. **ExamRegistration Table** (Junction Table for many-to-many relationship between Student and Exam):

```
CREATE TABLE ExamRegistration (
   StudentID INT,
   ExamID INT,
   PRIMARY KEY (StudentID, ExamID),
   FOREIGN KEY (StudentID) REFERENCES Student(StudentID),
   FOREIGN KEY (ExamID) REFERENCES Exam(ExamID)
);
```

ALTER Command: Adding Columns in Results Table:

```
   ALTER TABLE results ADD COLUMN Marksobtained INT,
                              ADD COLUMN Grade VARCHAR(2);
```

DROP Command: Deleting the Staff Table

```
         DROP TABLE Staff;
```

Week 5: Practicing DML Commands

AIM: **Practicing DML Commands:**

Solutions: DML COMMANDS on "**Student**, **Course**, **Class**, **Faculty**, **Exams**, and **Results**" Tables using INSERT command.

DML Commands are:

- *INSERT*
- *UPDATE*
- *DELETE*

Sample Data Insertion:

1. **Insert into Faculty**:

```
INSERT INTO Faculty (FacultyID, FirstName, LastName, Email, Phone,
Department, Designation)
VALUES (1, 'John', 'Doe', 'john.doe@university.com', '123-456-7890', 'Computer
Science', 'Professor');
```

2. **Insert into Course**:

```
INSERT INTO Course (CourseID, CourseName, CourseDescription, Credits)
VALUES (101, 'Database Management', 'An introduction to database design and
management', 3);
```

3. **Insert into Class**:

```
INSERT INTO Class (ClassID, ClassName, Schedule, Location, FacultyID,
CourseID)
VALUES (1, 'DBMS101', '2025-03-01 09:00:00', 'Room 101', 1, 101);
```

4. **Insert into Student**:

```
INSERT INTO Student (StudentID, FirstName, LastName, Email, Phone,
DateOfBirth, EnrollmentDate)
VALUES (1, 'Alice', 'Smith', 'alice.smith@email.com', '555-123-456', '2000-01-15',
'2025-02-20');
```

5. **Insert into Enrollment**:

```
INSERT INTO Enrollment (StudentID, CourseID, EnrollmentDate)
VALUES (1, 101, '2025-02-20');
```

6. **Insert into Exam**:

```
INSERT INTO Exam (ExamID, CourseID, ExamDate, ExamType, Duration,
TotalMarks)
VALUES (1, 101, '2025-06-01', 'Final', 120, 100);
INSERT INTO Exam (ExamID, CourseID, ExamDate, ExamType, Duration,
TotalMarks)
VALUES (2, 101, '2025-06-02', 'Final', 120, 100);
```

7. **Insert into Results**:

```
INSERT INTO Results (ResultID, StudentID, ExamID, MarksObtained, Grade)
VALUES (1, 1, 1, 85, 'A');
```

8. **Insert into ExamRegistration**:

```
INSERT INTO ExamRegistration (StudentID, ExamID)
VALUES (1, 1);
```

UPDATE COMMAND:

Updating the result table record:

```
update Results SET Grade='A' where StudentID=1;
```

DELETE COMMAND:
Deleting the record from exam table:

```
Delete *  exam where ExamID=2;
```

WEEK -6
Querying (using ANY, ALL, IN, EXISTS, NOTEXISTS, UNION,INTERSECT, Constraints Etc)

AIM: CREATING STUDENT TABLE, INSERT DATA I THE TABLE AND QUERYING USING ABOVE CONDITIONS.

SOLUTION:

- **CREATE A STUDENT TABLE:**

```
CREATE TABLE Students (
   StudentID INT PRIMARY KEY,
   FirstName VARCHAR(50),
   LastName VARCHAR(50),
   Email VARCHAR(100) UNIQUE,
   Phone VARCHAR(15) UNIQUE,
   DateOfBirth DATE,
   EnrollmentDate DATE
);
```

- **INSERTING THE 5 RECORDS TO STUDENT TABLE**

```
INSERT INTO Students (StudentID, FirstName, LastName, Email, Phone,
DateOfBirth, EnrollmentDate) VALUES
(101, 'John', 'Doe', 'john.doe@example.com', '9876543210', '2000-05-10', '2023-
06-15'),
(102, 'Emma', 'Smith', 'emma.smith@example.com', '9876543211', '2001-08-20',
'2023-06-16'),
(103, 'Alex', 'Brown', 'alex.brown@example.com', '9876543212', '2002-02-25',
'2023-06-17'),
(104, 'Sophia', 'Davis', 'sophia.davis@example.com', '9876543213', '2003-01-
10', '2023-06-18'),
(105, 'Michael', 'Wilson', 'michael.wilson@example.com', '9876543214', '2004-
09-14', '2023-06-19');
```

1. Using `ANY` (Compare with any value from a subquery)

```
SELECT * FROM Students
WHERE StudentID = ANY (SELECT StudentID FROM Students WHERE
EnrollmentDate >= '2023-06-17');
```

- Retrieves students whose **StudentID** matches any student enrolled on or after **2023-06-17**.

2. Using `ALL` (Compare with all values in a subquery)

```
SELECT * FROM Students
WHERE StudentID > ALL (SELECT StudentID FROM Students WHERE
DateOfBirth < '2001-01-01');
```

- Finds students whose **StudentID** is greater than all StudentIDs of students born before **2001-01-01**.

3. Using IN (Match any value in a list)

```
SELECT * FROM Students WHERE StudentID IN (101, 103, 105);
```

- **Fetches details of students with** StudentID **101, 103, or 105.**

4.Using EXISTS (Check if a subquery returns results)

```
SELECT * FROM Students S
WHERE EXISTS (SELECT 1 FROM Students WHERE Email LIKE
'%@example.com');
```

- Returns all students if at least one student has an email ending with @example.com.

5. Using NOT EXISTS (Check if a subquery returns no results)

```
SELECT * FROM Students S
WHERE NOT EXISTS (SELECT 1 FROM Students WHERE Email LIKE
'%@gmail.com');
```

- **Returns students only if no student has an email ending with @gmail.com.**

6. Using UNION (Combine results of two queries, removing duplicates)

```
SELECT FirstName, LastName FROM Students
UNION
SELECT FirstName, LastName FROM Students WHERE EnrollmentDate >
'2023-06-16';
```

7.Using INTERSECT (Find common records between two queries)

```
SELECT FirstName FROM Students
INTERSECT
SELECT FirstName FROM Students WHERE DateOfBirth <
'2002-01-01';
```

- **Returns students whose FirstName appears in both queries.**

8. Applying Constraints

Adding Constraints to `Students` Table

```
ALTER TABLE Students

ADD CONSTRAINT chk_dob CHECK (DateOfBirth < '2010-01-01');  -- Prevents future
birthdates

ALTER TABLE Students

ADD CONSTRAINT unique_email UNIQUE (Email);   -- Ensures unique emails

ALTER TABLE Students

ADD CONSTRAINT unique_phone UNIQUE (Phone);   -- Ensures unique phone
numbers
```

WEEK 7

Queries using Aggregate functions, GROUP BY, HAVING and Creation and dropping of Views.

AIM: Here are SQL queries demonstrating **aggregate functions, GROUP BY, HAVING, and Views** on the Students table.

1. Using Aggregate Functions

Aggregate functions perform calculations on multiple rows of a column.

SELECT COUNT(StudentID) AS TotalStudents FROM Students;

- ✓ **Counts the total number of students.**

SELECT MIN(DateOfBirth) AS OldestStudent, MAX(DateOfBirth) AS YoungestStudent FROM Students;

- ✓ **Finds the oldest and youngest students by date of birth.**

SELECT AVG(YEAR(CURDATE()) - YEAR(DateOfBirth)) AS AverageAge FROM Students;

- ✓ Calculates the average age of students.

2. Using GROUP BY

Groups data based on a specific column and applies aggregate functions.

SELECT EnrollmentDate, COUNT(StudentID) AS TotalEnrolled
FROM Students
GROUP BY EnrollmentDate;

- ✓ **Counts students enrolled on each unique EnrollmentDate.**

3. Using HAVING (Filter Grouped Data)

HAVING is used with GROUP BY to filter aggregated results.

SELECT EnrollmentDate, COUNT(StudentID) AS TotalEnrolled
FROM Students
GROUP BY EnrollmentDate
HAVING COUNT(StudentID) > 1;

- ✓ **Displays only enrollment dates where more than 1 student enrolled.**

4. Creating a View

A **view** is a virtual table based on a query.

```
CREATE VIEW StudentDetails AS
SELECT StudentID, FirstName, LastName, Email, EnrollmentDate
FROM Students;
```

- ✓ Creates a view named StudentDetails to display selected student details.

Querying the View:

```
SELECT * FROM StudentDetails;
```

5. Dropping a View

If a view is no longer needed, you can drop it.

```
DROP VIEW StudentDetails;
```

- ✓ Deletes the `StudentDetails` view.

Week 8

(Triggers (Creation of insert trigger, delete trigger, update trigger)

AIM: CREATE SQL Triggers on the `Students` Table

Triggers are special stored procedures that execute **automatically** when an event (INSERT, DELETE, UPDATE) occurs on a table.

1. Creating an `INSERT` Trigger

This trigger logs inserted records into an AuditLog table.

Create the AuditLog Table

```
CREATE TABLE AuditLog (

   LogID INT PRIMARY KEY AUTO_INCREMENT,

   ActionType VARCHAR(50),

   StudentID INT,

   Timestamp DATETIME DEFAULT CURRENT_TIMESTAMP

);
```

Create the AFTER INSERT Trigger

```
CREATE TRIGGER after_student_insert
AFTER INSERT ON Students
FOR EACH ROW
INSERT INTO AuditLog (ActionType, StudentID)
VALUES ('INSERT', NEW.StudentID);
```

- ✓ **Automatically logs new student records when inserted.**

2. Creating a DELETE Trigger

This trigger logs deleted student records.

- ✓ **Create the AFTER DELETE Trigger**

```
CREATE TRIGGER after_student_delete
AFTER DELETE ON Students
FOR EACH ROW
INSERT INTO AuditLog (ActionType, StudentID)
VALUES ('DELETE', OLD.StudentID);
```

3. Creating an UPDATE Trigger

This trigger logs updates to student email addresses.

- ✓ **Create the BEFORE UPDATE Trigger**

```
CREATE TRIGGER before_student_update
BEFORE UPDATE ON Students
FOR EACH ROW
INSERT INTO AuditLog (ActionType, StudentID)
VALUES ('UPDATE', OLD.StudentID);
```

- ✓ **Logs the StudentID before updating a record.**

4. Testing the Triggers

- ✓ **Insert a New Student**

```
INSERT INTO Students (StudentID, FirstName, LastName, Email, Phone,
DateOfBirth, EnrollmentDate)
VALUES (106, 'Liam', 'Johnson', 'liam.johnson@example.com', '9876543215',
'2003-07-22', '2023-06-20');
```

- ✓ **Delete a Student:**

```
DELETE FROM Students WHERE StudentID = 101;
```

- ✓ **Update a Student Email**

```
UPDATE Students SET Email = 'john.doe@newmail.com' WHERE StudentID
= 102;
```

✓ **View Audit Logs**

SELECT * FROM AuditLog;

5. Dropping a Trigger

If you need to remove a trigger:

DROP TRIGGER IF EXISTS after_student_insert;

WEEK 9
(PROCEDURES)

AIM : A **Stored Procedure** is a reusable SQL code block that can accept input parameters, process data, and return results.

1. Creating a Stored Procedure for Inserting a Student

This procedure inserts a student into the Students table.

```
DELIMITER $$

CREATE PROCEDURE InsertStudent(
   IN p_StudentID INT,
   IN p_FirstName VARCHAR(50),
   IN p_LastName VARCHAR(50),
   IN p_Email VARCHAR(100),
   IN p_Phone VARCHAR(15),
   IN p_DateOfBirth DATE,
   IN p_EnrollmentDate DATE
)
BEGIN
   INSERT INTO Students (StudentID, FirstName, LastName, Email,
Phone, DateOfBirth, EnrollmentDate)
   VALUES (p_StudentID, p_FirstName, p_LastName, p_Email,
p_Phone, p_DateOfBirth, p_EnrollmentDate);
END $$

DELIMITER ;
```

✓ **Calling the Procedure:**

```
CALL InsertStudent(107, 'David', 'Miller', 'david.miller@example.com',
'9876543216', '2002-12-10', '2023-06-21');
```

WEEK – 10
(USAGE OF CURSORS)

AIM: To demonstrate cursor usage.

A cursor in DBMS is a database object used to retrieve and process query results one row at a time. Normally, SQL works on a whole set of rows at once, but when we need to handle each record individually (for calculations, validations, or updates), a cursor is used. Cursors are mainly used inside stored procedures, functions, and triggers. They are very useful when row-by-row processing is required, such as generating reports or applying conditional logic to each record.

Example use cases:

- Calculating grade for each student
- Updating salary for each employee based on conditions
- Processing marks of students one by one

Types of Cursors

1. Implicit Cursor: Automatically created by DBMS for statements like INSERT, UPDATE, DELETE.
2. Explicit Cursor: Created and controlled by the programmer for SELECT queries returning multiple rows.

Program:

```
DECLARE done INT DEFAULT 0;
DECLARE v_sid INT;

DECLARE cur_student CURSOR FOR
SELECT SID FROM STUDENT;

DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = 1;

OPEN cur_student;

read_loop: LOOP
  FETCH cur_student INTO v_sid;
  IF done = 1 THEN
    LEAVE read_loop;
  END IF;
  -- process each SID here
END LOOP;

CLOSE cur_student;
```

Explanation:This cursor reads student IDs one by one from the STUDENT table and allows processing each record individually inside the loop.