Group 3: -

Name	PRN
Shailendra Mahadule	22620003
Omkar Penshanwar	21610018
Aditya Patil	21610005
Shamshad Choudhary	21610024
Rakesh Dharne	21610022
Shreyash Kamble	21610070

Experiment No 1

Aim

Study and design of Entity Relationship model. Construct ER Diagram for Sample Case Study (University/ Bank/Library etc.).

Introduction and Method

Entity-Relationship (ER) modeling is a fundamental technique used in database design to visually represent the entities, attributes, and relationships in a system. It aids in structuring and understanding the data and relationships that exist within a particular domain or system.

In this case, let us construct an ER diagram for a university. We'll consider entities like students, courses, teachers, and departments.

Entities:

Student

Course

Teacher

Department

Relationships:

Student - Enrolls in - Course (m:n): A student can enroll in multiple courses, and a course can have multiple students enrolled.

Teacher - Teaches - Course (1:1): A teacher can teach multiple courses, and a course can be taught by only one teacher.

Department - Offers - Course (1:n): A department offers multiple courses, and a course is offered by only one department.

Student - Belongs to - Department (n:1): A student belongs to one department, but a department can have multiple students.

Attributes:

Student: Student_ID (Primary Key), Name, Email, Address, Department_ID (Foreign Key)

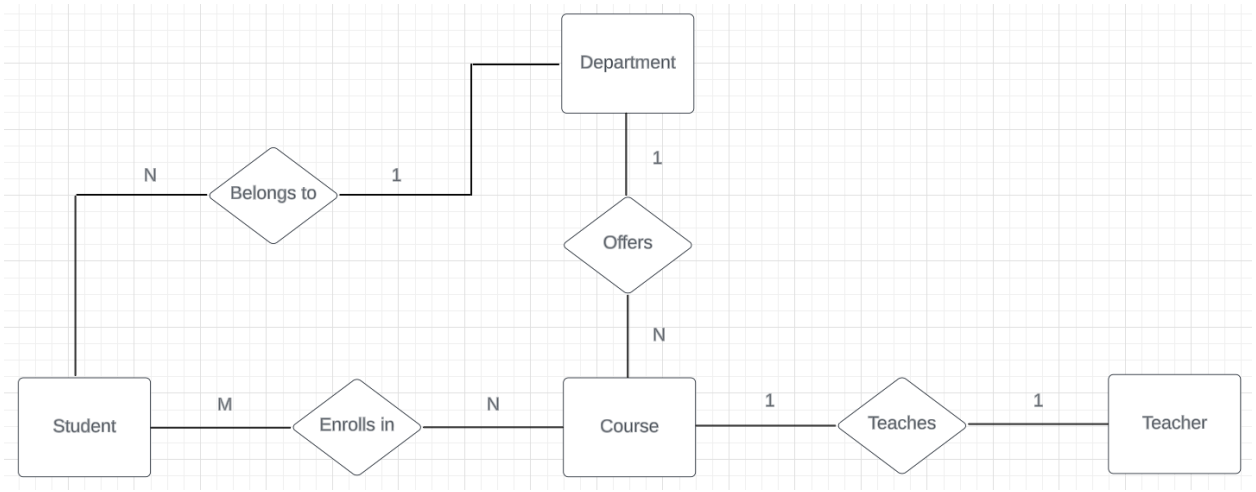
Course: Course_ID (Primary Key), Title, Credits, Department_ID (Foreign Key)

Professor: Professor_ID (Primary Key), Name, Email, Department_ID (Foreign Key)

Department: Department_ID (Primary Key), Name, Location

Results

ER diagram



Experiment No 2

Aim:

Create database schema from ER Model and apply database normalization.

Introduction and method:

The design and implementation of a database system play a pivotal role in managing information efficiently within an organization. In this context, we aim to create a well-structured database schema for a university system that encompasses various entities such as students, courses, teachers (instructors), and departments. This database system is intended to streamline the management of student enrollments, courses offered, teacher allocations, and departmental information.

```
CREATE TABLE Student (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(50),
    Address VARCHAR(100),
    Email VARCHAR(50),
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
);

CREATE TABLE Course (
    CourseID INT PRIMARY KEY,
    Title VARCHAR(100),
    Credits INT,
    DepartmentID INT,
    FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
);

CREATE TABLE Teacher (
    TeacherID INT PRIMARY KEY,
    Name VARCHAR(50),
    DepartmentID INT,
```

```
    FOREIGN KEY (DepartmentID) REFERENCES Department(DepartmentID)
);

CREATE TABLE Department (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(50)
);

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

Results:

First Normal Form (1NF):

Ensure atomicity and remove repeating groups.

All tables are in 1NF as each attribute contains only a single value, and there are no repeating groups.

Second Normal Form (2NF):

Eliminate partial dependencies.

The tables are in 2NF as each non-prime attribute is fully functionally dependent on the primary key.

Third Normal Form (3NF):

Remove transitive dependencies.

The tables are in 3NF as there are no transitive dependencies in the given schema.

BCNF (Boyce-Codd Normal Form):

Ensure that there are no non-trivial functional dependencies of attributes on anything other than a superkey.

The tables don’t have any non-trivial functional dependencies on anything other than superkeys. Hence, the schema is in BCNF.

This revised schema includes the entities Student, Course, Teacher, and Department, and is structured to maintain relationships and ensure data integrity through normalization.

Experiment No 3

Aim

The aim of this SQL demonstration is to illustrate the usage of Data Definition Language (DDL) and Data Manipulation Language (DML) commands in creating, altering, dropping, truncating, and renaming database tables, as well as performing operations like inserting, updating, and deleting data within these tables.

Introduction and Method

We'll employ SQL commands to create a database schema consisting of two tables: Department and Employee. The Department table stores department-related information, and the Employee table maintains details about

employees, including a foreign key referencing the Department table. Various DDL commands such as CREATE, ALTER, DROP, TRUNCATE, and RENAME will be utilized to manipulate the structure of these tables. Additionally, DML commands like INSERT, UPDATE, and DELETE will be used to insert, update, and delete data within the tables.

Results

DDL Commands:

(i). CREATE TABLE:

```
CREATE TABLE Department (
    DepartmentID INT PRIMARY KEY,
    DepartmentName VARCHAR(50) NOT NULL
);


CREATE TABLE Employee (
    EmployeeID INT PRIMARY KEY,
    EmployeeName VARCHAR(100) NOT NULL,
    DepartmentID INT,
    Salary DECIMAL(10, 2),
    CONSTRAINT FK_DepartmentID FOREIGN KEY (DepartmentID) REFERENCES
Department(DepartmentID)
);
```

(ii). ALTER TABLE:

```
ALTER TABLE Employee
```

ADD Salary DECIMAL(10, 2);

(iii). DROP TABLE:

DROP TABLE Employee;

(iv). TRUNCATE TABLE:

TRUNCATE TABLE Department;

(v). RENAME TABLE:

ALTER TABLE Employee

RENAME TO Faculty;

DML Commands:

(i). INSERT DATA:

INSERT INTO Employee (EmployeeID, EmployeeName, DepartmentID, Salary)
VALUES

(101, 'John Doe', 1, 50000.00),

(102, 'Jane Smith', 2, 60000.00),

(103, 'Bob Johnson', 1, 55000.00);

(ii). UPDATE DATA:

UPDATE Employee

SET Salary = 52000.00

WHERE EmployeeName = 'John Doe';

(iii) DELETE DATA:

```
DELETE FROM Employee

WHERE EmployeeName = 'Bob Johnson';
```

These SQL commands demonstrate the creation, alteration, dropping, truncating, renaming of tables, and performing operations to insert, update, and delete data in the database schema using DDL and DML commands.

Experiment No 4

Aim

Perform Insertion, Deletion, Modifying, Altering, Updating and Viewing records based on specific conditions.

Introduction and method

For this demonstration, we'll be utilizing a table named Students with the following structure:

```
CREATE TABLE Students (

    StudentID INT PRIMARY KEY,
```

```
    FirstName VARCHAR(50) NOT NULL,
    LastName VARCHAR(50) NOT NULL,
    Age INT,
    GPA DECIMAL(3, 2)
);
```

The goal is to execute various operations based on specific conditions using SQL commands. These operations include insertion of new student records, deletion of students based on GPA criteria, modification of student information, altering the table structure by adding a new column, updating particular fields for students, and viewing records based on specific GPA conditions.

Results:

(i). Insertion:

-> Insert new student records

```
INSERT INTO Students (StudentID, FirstName, LastName, Age, GPA) VALUES
(1, 'John', 'Doe', 20, 3.5),
(2, 'Jane', 'Smith', 22, 3.9),
(3, 'Bob', 'Johnson', 21, 3.2);
```

(ii). Deletion:

-> Delete students with GPA below 3.0

```
DELETE FROM Students
WHERE GPA < 3.0;
```

(iii). Modifying:

-> Modify age for student Jane Smith

```
UPDATE Students
```

```
SET Age = 23
WHERE FirstName = 'Jane' AND LastName = 'Smith';
```

(iv). Altering:

-> Add a new column to the Students table

```
ALTER TABLE Students
ADD Major VARCHAR(50);
```

(v). Updating:

-> Update major for student John Doe

```
UPDATE Students
SET Major = 'Computer Science'
WHERE FirstName = 'John' AND LastName = 'Doe';
```

(vi). Viewing Records Based on Specific Conditions:

-> View students with GPA greater than 3.5

```
SELECT *
FROM Students
WHERE GPA > 3.5;
```

Experiment No 5

Aim

Perform Aggregation and group by, having clause queries to retrieve summary information from database

Introduction and Method

Utilizing the Sales table in the database, we'll be employing SQL queries to perform aggregation functions combined with GROUP BY and HAVING clauses. These queries aim to summarize data and extract specific information based on defined criteria. The queries will calculate total revenue per category, identify categories surpassing revenue thresholds, and determine the count of products sold per category, applying additional filters for more nuanced results.

Results

1) Total Revenue per Category:

SELECT Category, SUM(Revenue) AS TotalRevenue

FROM Sales

GROUP BY Category;

2) Categories with Revenue Greater Than $10,000:

SELECT Category, SUM(Revenue) AS TotalRevenue

FROM Sales

GROUP BY Category

HAVING SUM(Revenue) > 10000;

3)Number of Products Sold per Category (with more than 50 products sold):

SELECT Category, COUNT(Product) AS NumberOfProducts

FROM Sales

GROUP BY Category

HAVING COUNT(Product) > 50;

4)Categories with Revenue Greater Than $10,000:

SELECT Category, SUM(Revenue) AS TotalRevenue

FROM Sales

GROUP BY Category

HAVING SUM(Revenue) > 10000;

Experiment No 6

Aim

Study of various types of integrity constraints (NOT NULL Constraint, DEFAULT Constraint, UNIQUE Constraint, PRIMARY Key, FOREIGN Key, CHECK Constraint).

Introduction and Method

In the realm of database management, integrity constraints serve as critical components to maintain the accuracy, reliability, and consistency of data stored within tables. This study delves into various types of integrity constraints commonly used in SQL databases, including NOT NULL, DEFAULT, UNIQUE, PRIMARY KEY, FOREIGN KEY, and CHECK constraints.

Results

- **NOT NULL Constraint:**

This constraint ensures that a column cannot have NULL values. It enforces the presence of a value in a particular column.

Example: CREATE TABLE Employees (ID INT NOT NULL, Name VARCHAR(50) NOT NULL);

- **DEFAULT Constraint:**

Specifies a default value for a column when an INSERT statement doesn't provide a specific value for that column.

Example: CREATE TABLE Orders (OrderID INT, OrderDate DATE DEFAULT GETDATE());

- **UNIQUE Constraint:**

Guarantees that all values in a column or a group of columns are unique within the table.

Example: CREATE TABLE Students (StudentID INT UNIQUE, Email VARCHAR(50) UNIQUE);

- **PRIMARY Key:**

A PRIMARY Key is a unique identifier for a record within a table. It uniquely identifies each record and does not allow NULL values.

Example: CREATE TABLE Books (BookID INT PRIMARY KEY, Title VARCHAR(100), Author VARCHAR(100));

- **FOREIGN Key:**

Establishes a link between data in two tables. It refers to the PRIMARY Key or UNIQUE constraint in another table, maintaining referential integrity between the two related tables.

Example:

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

- **CHECK Constraint:**

Imposes a condition on the values being inserted or updated in a column. It allows for enforcing domain integrity by restricting the values that can be placed in a column.

Example:

CREATE TABLE Employees (

ID INT,

Age INT CHECK (Age >= 18)

);

Experiment No 7

Aim

To perform set operations queries.

Introduction and Method

This study focuses on performing set operations, a fundamental aspect of SQL, to manipulate and extract data from tables. Specifically, the study involves executing set operation queries like UNION, INTERSECT, and EXCEPT (also known as MINUS) on two tables, table1 and table2, to showcase their functionalities and how they interact with data.

Two tables, table1 and table2, have been created, each containing 'id' and 'name' columns with primary keys on the 'id' column. The tables have been populated with sample data to enable the execution of set operation queries.

Results

-> Create table1

```
CREATE TABLE table1 (
   id INT,
   name VARCHAR(50),
   PRIMARY KEY (id)
);
```

-> Insert data into table1

```
INSERT INTO table1 (id, name)
VALUES
   (1, 'John'),
   (2, 'Alice'),
   (3, 'Bob'),
   (4, 'Eve');
```

-> Create table2

```
CREATE TABLE table2 (
   id INT,
   name VARCHAR(50),
   PRIMARY KEY (id)
);
```

-> Insert data into table2

```
INSERT INTO table2 (id, name)
VALUES
```

(2, 'Alice'),

(3, 'Bob'),

(5, 'Charlie'),

(6, 'David');

Set Operations:

1. Union

SELECT id, name FROM table1

UNION

SELECT id, name FROM table2;

This will give you a result set with combined unique rows from both tables.

2. Intersect

SELECT id, name FROM table1

INTERSECT

SELECT id, name FROM table2;

This will give you a result set with common rows between the two tables.

3. Except (Minus)

SELECT column1, column2 FROM table1

EXCEPT

SELECT column1, column2 FROM table2;

Experiment No 8

Aim

Create database views. Creation of views using views, Drop view.

Introduction and Method

The focus of this study is to explore the creation and utilization of database views in a SQL environment. Views in a database act as virtual tables derived from existing tables or other views, providing a dynamic perspective on the data without altering the underlying structure. This study involves creating views, nesting views within other views, querying views, and subsequently dropping both views and the sample table.

A sample table named employees has been created with fields such as employee_id, first_name, last_name, and department. Data pertaining to employees has been inserted into this table for demonstration purposes.

Results

-> Create a sample table

```
CREATE TABLE employees (
employee_id INT PRIMARY KEY,
first_name VARCHAR(50),
last_name VARCHAR(50),
department VARCHAR(50)
);
```

-> Insert some data into the table

```
INSERT INTO employees (employee_id, first_name, last_name, department)
        VALUES
           (1, 'John', 'Doe', 'HR'),
           (2, 'Alice', 'Smith', 'IT'),
           (3, 'Bob', 'Johnson', 'Finance'),
           (4, 'Eve', 'Williams', 'IT');
```

-> Create a view based on the employees table

```
CREATE VIEW hr_employees AS
SELECT employee_id, first_name, last_name
FROM employees
WHERE department = 'HR';
```

-> Create another view based on the hr_employees view

```
CREATE VIEW hr_manager AS
SELECT * FROM hr_employees
WHERE employee_id = 1;
```

-> Query the views

```
SELECT * FROM hr_employees;
SELECT * FROM hr_manager;
```

-> Drop the views

```
DROP VIEW hr_manager;
DROP VIEW hr_employees;
```

-> Drop the sample table

```
DROP TABLE employees;
```

Experiment No 9

Aim

Perform cross join, Natural Join, Inner join and Outer Join queries.

Introduction and Method

This study focuses on exploring various types of SQL joins - Cross Join, Natural Join, Inner Join, Left Join, Right Join, and Full Outer Join - and their functionalities within a database environment. Using a sample database, we'll execute queries involving these join operations to demonstrate how they retrieve and combine data from multiple tables.

A database named mydatabase has been created. Two tables, customers and orders, have been established with specific fields, primary keys, and sample data inserted into them.

Results

-> Creating a database

```
CREATE DATABASE mydatabase;
USE mydatabase;
```

-> Creating a customers table

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50),
    order_id INT
);
```

-> Inserting sample data into the customers table

```
INSERT INTO customers (customer_id, customer_name, order_id) VALUES
(1, 'John Doe', 101),
(2, 'Jane Smith', 102),
(3, 'Bob Johnson', 103);
```

-> Creating an orders table

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    total_amount DECIMAL(10, 2)
```

);

-> Inserting sample data into the orders table

INSERT INTO orders (order_id, order_date, total_amount) VALUES

(101, '2023-01-01', 50.00),

(102, '2023-02-01', 75.00),

(103, '2023-03-01', 100.00);

-> Perform cross join

SELECT * FROM customers CROSS JOIN orders;

-> Perform natural join

SELECT * FROM customers NATURAL JOIN orders;

-> Perform inner join

SELECT * FROM customers INNER JOIN orders ON customers.order_id = orders.order_id;

-> Perform left join

SELECT * FROM customers LEFT JOIN orders ON customers.order_id = orders.order_id;

-> Perform right outer join

SELECT * FROM customers RIGHT JOIN orders ON customers.order_id = orders.order_id;

-> Perform full outer join

SELECT * FROM customers FULL OUTER JOIN orders ON customers.order_id = orders.order_id;

Experiment No 10

Aim

Create a row level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table.

Introduction and Method

This study focuses on the creation and utilization of a row-level trigger within a MySQL database environment. Specifically, a trigger is set up for the customers table to capture and log operations such as INSERT, UPDATE, or DELETE performed on the table.

The study begins by creating a database named mydatabase and setting up two tables - customers and orders. Sample data is inserted into both tables to facilitate testing and demonstration of the trigger's functionality.

Results

-> Creating a database

```
CREATE DATABASE IF NOT EXISTS mydatabase;
USE mydatabase;
```

-> Creating a customers table

```
CREATE TABLE IF NOT EXISTS customers (
customer_id INT PRIMARY KEY,
customer_name VARCHAR(50),
order_id INT
);
```

-> Inserting sample data into the customers table

```
INSERT INTO customers (customer_id, customer_name, order_id) VALUES
(1, 'John Doe', 101),
(2, 'Jane Smith', 102),
(3, 'Bob Johnson', 103);
```

-> Creating an orders table

```
CREATE TABLE IF NOT EXISTS orders (
order_id INT PRIMARY KEY,
order_date DATE,
```

```
total_amount DECIMAL(10, 2)
);

-> Inserting sample data into the orders table
INSERT INTO orders (order_id, order_date, total_amount) VALUES
(101, '2023-01-01', 50.00),
(102, '2023-02-01', 75.00),
(103, '2023-03-01', 100.00);

-> Creating a trigger log table
CREATE TABLE IF NOT EXISTS customers_trigger_log (
log_id INT AUTO_INCREMENT PRIMARY KEY,
customer_id INT,
operation_type VARCHAR(10),
operation_timestamp TIMESTAMP
);

-> Creating a trigger
DELIMITER //

CREATE TRIGGER customers_trigger
AFTER INSERT OR UPDATE OR DELETE ON customers
```

```
FOR EACH ROW

BEGIN

DECLARE operation_type VARCHAR(10);


IF (NEW.customer_id IS NOT NULL AND OLD.customer_id IS NULL) THEN

SET operation_type = 'INSERT';

ELSEIF (NEW.customer_id IS NOT NULL AND OLD.customer_id IS NOT NULL) THEN

SET operation_type = 'UPDATE';

ELSEIF (NEW.customer_id IS NULL AND OLD.customer_id IS NOT NULL) THEN

SET operation_type = 'DELETE';

END IF;

INSERT INTO customers_trigger_log (customer_id, operation_type,
operation_timestamp)

VALUES (NEW.customer_id, operation_type, NOW());

END;

//

DELIMITER ;
```

Experiment No 11

Aim

Implement MYSQL database connectivity with python/Java. Implement Database queries (insert, delete, update) using ODBC/JDBC.

Introduction and Method

This demonstration showcases database connectivity and execution of basic CRUD (Create, Read, Update, Delete) operations using Python and MySQL. The Python code connects to a MySQL database and performs queries (insert, delete, select) using the mysql.connector library.

Python with MySQL Connectivity:

The Python script employs the mysql.connector module to establish a connection with the MySQL database. It includes functions to perform database operations such as displaying all records, inserting new records, and deleting existing records from the Student table.

Results

```
import mysql.connector

# Replace these with your database details

host = "localhost"

user = "root"

password = "anonymous"

database = "om"


# Create a connection

connection = mysql.connector.connect(

    host=host,

    user=user,

    password=password,

    database=database

)


cursor = connection.cursor()


def display () :
```

```
    query = "SELECT * FROM Student order by prn desc"
    cursor.execute(query)
    results = cursor.fetchall()

    for row in results:
        for it in row :
            print(it,end=" ")
        print()
def insert(d):
    query = "INSERT into Student (prn,name,age) values (%s, %s, %s)"
    data=d
    cursor.execute(query,data)
    connection.commit()

def delete(p):
    query="DELETE from Student where prn=%s"
    data=(p,)
    cursor.execute(query,data)
    connection.commit()

# # Example SELECT query
# query = "SELECT * FROM Student order by prn desc"
```

```
# cursor.execute(query)


# # Fetch the results
# results = cursor.fetchall()


# # Iterate through the results
# for row in results:
#     for it in row :
#         print(it,end=" ")
#     print()


# # Query 2 :
# query = "SELECT prn from Student"
# cursor.execute(query)
# results = cursor.fetchall()


# for row in results :
#     print(row[0])


# display()
# insert((21610088,"Amir Khan",24))
print("*****************")
```

```
display()

print("******************")

delete(21610090)

display()

cursor.close()

connection.close()
```

Experiment No 12

Aim

Study of Open Source NOSQL Database: MongoDB (Installation, Basic CRUD operations, Execution)

Introduction and Method

This study focuses on exploring the MongoDB NoSQL database, covering its installation process and basic operations for Create, Read, Update, and Delete (CRUD) operations, as well as the execution of queries using MongoDB's query language.

Results

1. Installation:

Windows:

- Download MongoDB Community Server from the official website: MongoDB Download Center.

- Follow the installation wizard to complete the installation.

- MongoDB will be installed as a Windows service, and the default data directory is C:\Program Files\MongoDB\Server\{version}\data\db.

2. Basic CRUD Operations:

Connect to MongoDB:

Open a terminal or command prompt and run:

Create (Insert) Operation:

```
// Switch to a specific database (create if not exists)
use mydatabase

// Insert a document into a collection
db.mycollection.insertOne({
    name: 'John Doe',
    age: 30,
```

```
    city: 'New York'
})
```

Read Operation:

```
// Find all documents in a collection
db.mycollection.find()

// Find documents with a specific condition
db.mycollection.find({ name: 'John Doe' })
```

Update Operation:

Copy code

```
// Update a document
db.mycollection.updateOne(
    { name: 'John Doe' },
    { $set: { age: 31 } }
)
```

Delete Operation:

```
// Delete a document
db.mycollection.deleteOne({ name: 'John Doe' })
```

3. Execution of Queries:

MongoDB provides a flexible and powerful query language. Here are some examples:

```
// Find documents with age greater than 25
```

```
db.mycollection.find({ age: { $gt: 25 } })

// Find documents with a specific field
db.mycollection.find({ city: { $exists: true } })

// Find documents with a specific value in an array
db.mycollection.find({ hobbies: 'reading' })

// Aggregation Pipeline Example (group by city and calculate average age)
db.mycollection.aggregate([
        { $group: { _id: '$city', avgAge: { $avg: '$age' } } }
])
```