Practical 1: Database Design and Table Creation

Aim

To design a relational database schema for a college management system and implement it by creating the required tables using SQL CREATE TABLE command.

Theory

A relational database is a collection of data items with pre-defined relationships between them. These items are organized as a set of tables with columns and rows. The design process involves identifying entities (e.g., Student, Department, Course), their attributes, and the relationships between them.

The SQL CREATE TABLE command is a Data Definition Language (DDL) command used to define and create a new table in the database.

Syntax:

```
CREATE TABLE table_name (    column1 datatype
[constraints],    column2 datatype [constraints],
column3 datatype [constraints],
  ...
  [table_constraints]
);
```

Program:

```
-- Create the college database
CREATE DATABASE CollegeDB;

-- Select the database to use
USE CollegeDB;

-- Create the Departments table first (since Students and Courses depend on it)
CREATE TABLE Departments (
   DeptID INT PRIMARY KEY AUTO_INCREMENT,
   DeptName VARCHAR(50) NOT NULL,
   HOD VARCHAR(50)
);

-- Create the Students table
CREATE TABLE Students (
   StudentID INT PRIMARY KEY AUTO_INCREMENT,
   FirstName VARCHAR(50) NOT NULL,
   LastName VARCHAR(50),
   DateOfBirth DATE,
   Email VARCHAR(100) UNIQUE,
   DeptID INT,
   FOREIGN KEY (DeptID) REFERENCES Departments(DeptID)
      ON DELETE SET NULL
      ON UPDATE CASCADE
);

-- Create the Courses table
CREATE TABLE Courses (
   CourseID INT PRIMARY KEY AUTO_INCREMENT,
   CourseName VARCHAR(100) NOT NULL,
```

```
    Credits INT CHECK (Credits > 0),
    DeptID INT,
    FOREIGN KEY (DeptID) REFERENCES Departments(DeptID)
        ON DELETE CASCADE
        ON UPDATE CASCADE
);
```

☑ **Sample Output (Execution result)**

When you run these commands in MySQL Workbench or any SQL terminal, you'll see:

Query OK, 1 row affected (0.01 sec)
Database changed
Query OK, 0 rows affected (0.02 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.01 sec)

Conclusion : We successfully designed a relational database schema for a college and implemented it by creating three related tables (Departments, Students, Courses) using the CREATE TABLE command.

Practical 2: Applying Constraints

Aim

To apply various integrity constraints to the tables, such as PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE, CHECK, and DEFAULT.

Theory

Integrity constraints are rules that are applied to columns to ensure the accuracy and reliability of the data. They are a fundamental part of good database design.

- **PRIMARY KEY**: Uniquely identifies each row in a table.
- **FOREIGN KEY**: Establishes a link between data in two tables.
- **NOT NULL**: Ensures that a column cannot have a NULL value.
- **UNIQUE**: Ensures that all values in a column are different.
- **CHECK**: Ensures that a value in a column meets a specific condition.
- **DEFAULT**: Provides a default value for a column when none is specified.

Program

```
-- Assuming tables from Practical 1 are already created
USE CollegeDB;

-- Add a UNIQUE constraint to the Email column in Students table (already done in Pr1)
-- This line is safe; if already unique, it will throw a warning if re-added
ALTER TABLE Students
ADD CONSTRAINT UQ_Email UNIQUE (Email);

-- Add a CHECK constraint to the Students table to ensure age is above 18
-- MySQL supports CHECK constraints from version 8.0.16+
ALTER TABLE Students
ADD CONSTRAINT CHK_Age CHECK (TIMESTAMPDIFF(YEAR, DateOfBirth, CURDATE()) >= 18);

-- Add a DEFAULT value for the HOD column in Departments table
ALTER TABLE Departments
ALTER COLUMN HOD SET DEFAULT 'To be assigned';

-- Add a new table 'Faculty' with constraints
CREATE TABLE Faculty (
   FacultyID INT PRIMARY KEY AUTO_INCREMENT,
   FullName VARCHAR(100) NOT NULL,
   DeptID INT,
   Salary DECIMAL(10, 2) CHECK (Salary >= 30000),
   CONSTRAINT FK_FacultyDept FOREIGN KEY (DeptID)
      REFERENCES Departments(DeptID)
      ON DELETE SET NULL
      ON UPDATE CASCADE
);
```

Output:

```
Database changed
Query OK, 0 rows affected (0.02 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.01 sec)
Query OK, 0 rows affected (0.02 sec)
Query OK, 0 rows affected (0.03 sec)
```

Conclusion

We successfully applied various integrity constraints to the tables in our college database.
This ensures data consistency and prevents invalid or incomplete data from being entered.

Practical 3: ALTER, UPDATE, and DELETE SQL statements

Aim

To use the ALTER, UPDATE, and DELETE commands to modify the structure and data of existing tables.

Theory

These are essential Data Manipulation Language (DML) and Data Definition Language (DDL) commands for maintaining a database.

- **ALTER TABLE**: Used to add, delete, or modify columns in an existing table. It can also be used to add or drop constraints.
- **UPDATE**: Used to modify existing records in a table. The WHERE clause is crucial to specify which records to update.
- **DELETE**: Used to delete existing records from a table. The WHERE clause is crucial to specify which records to delete.

Program

```
-- Assuming tables from previous practicals already exist
USE CollegeDB;

-- Add a new column 'Semester' to the Students table
ALTER TABLE Students
ADD COLUMN Semester INT DEFAULT 1;

-- Update the Semester for all existing students (not necessary if default is set, but included)
UPDATE Students
SET Semester = 1;

-- Add some sample data to work with
-- (Avoid manually setting IDs if AUTO_INCREMENT is enabled)
INSERT INTO Departments (DeptName, HOD) VALUES ('Computer Science', 'Dr. Rao');
INSERT INTO Departments (DeptName, HOD) VALUES ('Mechanical Engineering', 'Dr. Verma');

-- Check the DeptIDs assigned automatically
SELECT * FROM Departments;

-- Insert sample students (using auto-generated StudentID)
INSERT INTO Students (FirstName, LastName, DateOfBirth, Email, DeptID)
VALUES
('John', 'Doe', '2003-05-12', 'john.doe@gmail.com', 1),
('Jane', 'Smith', '2002-08-20', 'jane.smith@gmail.com', 2);

-- Update a specific student's department
UPDATE Students
SET DeptID = 1
WHERE FirstName = 'Jane' AND LastName = 'Smith';

-- Delete a specific department
-- ⚠ This will fail if foreign key constraint prevents deletion.
-- To allow it, ensure ON DELETE CASCADE or SET NULL is set in Students table.
DELETE FROM Departments
WHERE DeptID = 2;

-- Verify final data
```

SELECT * FROM Departments;
SELECT * FROM Students;

Output:

☑ **Expected Outputs**

After all operations, your tables will look like:

Departments

DeptID	DeptName	HOD
1	Computer Science	Dr. Rao

Students

StudentID	FirstName	LastName	DateOfBirth	Email	DeptID	Semester
1	John	Doe	2003-05-12	john.doe@gmail.com	1	1
2	Jane	Smith	2002-08-20	jane.smith@gmail.com	1	1

Conclusion :

We successfully used ALTER TABLE to modify a table's structure, UPDATE to change data within the tables, and DELETE to remove records. These commands are essential for database maintenance.

Practical 4: Joins

Aim

To write queries to retrieve data from multiple tables using different types of joins.

Theory

A JOIN is a clause that combines rows from two or more tables based on a related column between them.

- **INNER JOIN**: Returns rows when there is at least one match in both tables.
- **LEFT JOIN**: Returns all rows from the left table, and the matched rows from the right table. The result is NULL from the right side if there is no match.
- **RIGHT JOIN**: Returns all rows from the right table, and the matched rows from the left table. The result is NULL from the left side when there is no match.
- **FULL OUTER JOIN**: Returns all rows when there is a match in one of the tables.

Program

```
-- Inner Join: Get students with their department names
SELECT S.FirstName, S.LastName, D.DeptName
FROM Students AS S
INNER JOIN Departments AS D ON S.DeptID = D.DeptID;

-- Left Join: Get all students and their department names (if they have one)
SELECT S.FirstName, S.LastName, D.DeptName
FROM Students AS S
LEFT JOIN Departments AS D ON S.DeptID = D.DeptID;

-- Right Join: Get all departments and their student names (if they have any)
SELECT S.FirstName, S.LastName, D.DeptName
FROM Students AS S
RIGHT JOIN Departments AS D ON S.DeptID = D.DeptID;
```

Output:

Inner join:

FirstName	LastName	DeptName
John	Doe	Computer Science
Jane	Smith	Computer Science

Left join:

FirstName	LastName	DeptName
John	**Doe**	**Computer Science**
Jane	**Smith**	**Computer Science**
Ramesh	**Patil**	**NULL**

Right join:

FirstName	LastName	DeptName
John	**Doe**	**Computer Science**
Jane	**Smith**	**Computer Science**
NULL	**NULL**	**Mechanical Engineering**

Conclusion

We successfully used INNER, LEFT, and RIGHT joins to retrieve combined data from the Students and Departments tables. Joins are fundamental for working with relational databases.

Practical 5: Aggregate Functions

Aim

To write queries that implement aggregate functions like MAX(), MIN(), AVG(), and COUNT().

Theory

Aggregate functions perform a calculation on a set of values and return a single value. They are often used with the GROUP BY clause.

- **COUNT()**: Returns the number of rows that match a specified criterion.
- **SUM()**: Returns the total sum of a numeric column.
- **AVG()**: Returns the average value of a numeric column.
- **MIN()**: Returns the smallest value of the selected column.
- **MAX()**: Returns the largest value of the selected column.

Program

-

```
-- Count the number of students
SELECT COUNT(*) FROM Students;

-- Find the maximum and minimum number of credits for courses
SELECT MAX(Credits), MIN(Credits) FROM Courses;

-- Calculate the average salary of faculty
SELECT AVG(Salary) FROM Faculty;

-- Count the number of students in each department
SELECT DeptID, COUNT(StudentID)
FROM Students
GROUP BY DeptID;
```

Conclusion :

We successfully used a variety of aggregate functions to perform calculations on our data, such as counting students, finding max/min values, and calculating averages. These functions are essential for data analysis.

Practical 6: Subqueries and Nested Queries

Aim

To implement the concept of nested queries and subqueries to retrieve data that depends on the result of another query.

Theory

A subquery, or nested query, is a query that is embedded within another query. The inner query executes first, and its result is used by the outer query.

- **Subquery in WHERE clause**: Used to filter data based on the result of the inner query. ● **Subquery in FROM clause**: The result of the subquery is treated as a temporary table.
- **Subquery in SELECT clause**: Used to return a single value for each row of the outer query.

Program

```
-- Find students in a specific department using a subquery
SELECT FirstName, LastName
FROM Students
WHERE DeptID = (SELECT DeptID FROM Departments WHERE DeptName = 'Computer
Science');

-- Find the courses that have more credits than the average credits
SELECT CourseName, Credits
FROM Courses
WHERE Credits > (SELECT AVG(Credits) FROM Courses);

-- Find the departments that have at least 2 students
SELECT DeptName
FROM Departments
WHERE DeptID IN (SELECT DeptID FROM Students GROUP BY DeptID HAVING COUNT(*) >=
2);
```

Conclusion

We successfully used subqueries to perform more complex data retrieval. This demonstrates how to combine the power of multiple queries to achieve a desired result.

Practical 7: Views, Sequences, and Indices

Aim

To create and manage views, sequences, and indices to enhance database performance and security.

Theory

- **Views**: A virtual table based on the result-set of an SQL statement. It does not store data but rather represents a predefined query. Views are used for security and to simplify complex queries.
- **Sequences**: A database object that generates a unique number for each request. They are often used for automatically generating primary key values.
- **Indices**: A data structure that improves the speed of data retrieval operations on a database table. An index is a pointer to the data in the table.

Program

-

```
-- Create a view to show student names and department names
CREATE VIEW StudentDeptView AS
SELECT S.FirstName, S.LastName, D.DeptName
FROM Students AS S
INNER JOIN Departments AS D ON S.DeptID = D.DeptID;

-- Query the new view
SELECT * FROM StudentDeptView;

-- Create an index on the LastName column of the Students table
CREATE INDEX idx_student_lastname ON Students(LastName);

-- Create a sequence for student IDs
CREATE SEQUENCE StudentID_Seq
START WITH 100
INCREMENT BY 1;

-- Use the sequence to insert a new student
INSERT INTO Students (StudentID, FirstName, LastName, DeptID)
VALUES (NEXT VALUE FOR StudentID_Seq, 'Alice', 'Smith', 101);
```

Conclusion :

We successfully created a view for simplified data access, an index to optimize query performance, and a sequence for automatic ID generation. These objects are crucial for building efficient and maintainable database applications.

Practical 8: Triggers

Aim

To perform queries for triggers.

Theory

A trigger is a set of SQL statements that is executed automatically in response to certain events on a table, such as an INSERT, UPDATE, or DELETE operation. Triggers are used to enforce business rules, audit data changes, or automate tasks.

- **Syntax:**

```
CREATE TRIGGER trigger_name
[BEFORE | AFTER] [INSERT | UPDATE | DELETE] ON table_name
FOR EACH ROW
BEGIN
  -- SQL statements to be executed
END;
```

- **BEFORE/AFTER**: Specifies when the trigger fires (before or after the event).
- **FOR EACH ROW**: The trigger will execute for every row affected by the DML statement.

Program

```
-- Assuming a table 'StudentLog' exists to log changes
USE CollegeDB;
CREATE TABLE StudentLog (
   LogID INT PRIMARY KEY AUTO_INCREMENT,     StudentID INT,
   Operation VARCHAR(10),
   OperationTime DATETIME
);

-- Create a trigger that logs every student insertion
CREATE TRIGGER AfterInsertStudent
AFTER INSERT ON Students
FOR EACH ROW
INSERT INTO StudentLog (StudentID, Operation, OperationTime)
VALUES (NEW.StudentID, 'INSERT', NOW());

-- Insert a new student to test the trigger
INSERT INTO Students (StudentID, FirstName, LastName, DeptID) VALUES (10, 'Mark', 'Wilson',
101);

-- Verify the log entry
SELECT * FROM StudentLog;
```

Conclusion:

We successfully created a trigger that automatically logs every student insertion into a separate log table. This demonstrates how triggers can be used for automation and auditing.

Practical 9: Insertion, Updation, and Deletion with Referrential Integrity

Aim

To perform operations for demonstrating the insertion, updation, and deletion using the referential integrity constraints.

Theory

Referential integrity is a relational database concept that ensures that relationships between tables are consistent. It is enforced using the FOREIGN KEY constraint. When a foreign key exists, it must reference a valid row in the parent table. The ON DELETE and ON UPDATE clauses of a foreign key constraint define the behavior when a row in the parent table is modified.

- **ON DELETE CASCADE**: When a row in the parent table is deleted, the corresponding rows in the child table are also automatically deleted.
- **ON DELETE SET NULL**: When a row in the parent table is deleted, the foreign key column in the child table is set to NULL.
- **ON DELETE RESTRICT**: Prevents the deletion of a row in the parent table if it has corresponding rows in the child table. This is the default behavior.

Program

```
-- First, drop the existing foreign key constraint
ALTER TABLE Students
DROP FOREIGN KEY Students_ibfk_1;

-- Now, recreate the foreign key with CASCADE behavior
ALTER TABLE Students
ADD CONSTRAINT FK_StudentDept CASCADE ON DELETE;

-- Insert some sample data to work with
INSERT INTO Departments (DeptID, DeptName) VALUES (103, 'Civil Engineering');
INSERT INTO Students (StudentID, FirstName, LastName, DeptID) VALUES (20, 'David', 'Lee', 103);
INSERT INTO Courses (CourseID, CourseName, DeptID) VALUES (301, 'Structural Analysis', 103);

-- Delete a department and observe the cascade effect
DELETE FROM Departments WHERE DeptID = 103;

-- Attempt to insert a student with a non-existent department ID (this should fail)
INSERT INTO Students (StudentID, FirstName, LastName, DeptID) VALUES (21, 'Emily', 'Clark', 999);
```

Conclusion :

We demonstrated how referential integrity, enforced by foreign keys with CASCADE behavior, automatically handles related data when a record in the parent table is deleted. We also showed how the database prevents data inconsistency by rejecting an insertion that violates referential integrity.

Practical 10: User and Role Management

Aim

To create and manage users and their roles to control access to the database.

Theory

User and role management is a critical aspect of database security.

- **CREATE USER**: Command to create a new user account.
- **GRANT**: Command to give privileges (permissions) to a user or role.
- **REVOKE**: Command to remove privileges from a user or role.
- **CREATE ROLE**: Command to create a role, which is a collection of privileges.

Program

```
-- Assuming a root user
-- Create a new user
CREATE USER 'student_user'@'localhost' IDENTIFIED BY 'password123';

-- Create a role for faculty
CREATE ROLE 'faculty_role';

-- Grant privileges to the role
GRANT SELECT, INSERT, UPDATE ON CollegeDB.Faculty TO 'faculty_role';

-- Grant the role to the user
GRANT 'faculty_role' TO 'student_user'@'localhost';

-- Revoke the insert privilege from the user
REVOKE INSERT ON CollegeDB.Faculty FROM 'student_user'@'localhost';

-- Drop the user and role
DROP USER 'student_user'@'localhost';
DROP ROLE 'faculty_role';
```

Conclusion

We successfully created a new user, a role, granted privileges to the role, and then assigned the role to the user. We also demonstrated how to revoke privileges. This shows how to implement basic access control for a database.