# II SEMESTER MCA

# DBMS Lab Programs

## 1.Concept Design with ER model

Creating a conceptual data model using an entity-relationship (ER) model to represent the data needed for a specific application. This model visualizes entities, their attributes, and the relationships between them. The goal is to translate a problem domain into a structured data representation that can be used to design a database.

Let's say a lab program involves designing a database for a library management system.

1. **1. Entities:**

- Books: Represents the physical books in the library.
- Members: Represents the library users.
- Authors: Represents the authors of the books.
- Loans: Represents the borrowing of books by members.

2. **2. Attributes:**

- Books: Title, ISBN, Publication Date, Number of Pages, Author ID.
- Members: Member ID, Name, Address, Email, Date of Birth.
- Authors: Author ID, Name, Nationality.
- Loans: Loan ID, Book ID, Member ID, Loan Date, Due Date, Return Date.

3. **3. Relationships:**

- Books and Authors: A Books entity is authored by an Author entity (many-to-one).
- Members and Loans: A Member entity can have multiple Loans (one-to-many).
- Books and Loans: A Book entity can be involved in multiple Loans (one-to-many).

4. **4. ER Diagram:**

- The entities would be represented as rectangles.
- Attributes would be represented as ovals connected to their respective entities.
- Relationships would be represented as diamonds connecting the relevant entities.

## 2. Design database and create tables. For Eg: Bank, College

**Create Database Bank**

CREATE DATABASE BankDB;

**2. Customers Table**

CREATE TABLE Customers (CustomerID INT PRIMARY KEY ,CustName VARCHAR(50), Phone VARCHAR(20),Address TEXT);

**3. Accounts Table**

CREATE TABLE Accounts ( AccountID INT PRIMARY KEY ,CustomerID INT,balance number(6,2), AccountType VARCHAR(20) FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID));

**4. Transactions Table**

CREATE TABLE Transactions (TransactionID INT PRIMARY KEY, AccountID INT, TransactionType VARCHAR(20), Amount DECIMAL(15, 2), FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID));

**5. Branch Table**

CREATE TABLE Branches (BranchID INT PRIMARY KEY, BranchName VARCHAR(100), Location  VARCHAR(100));

**6. Employees Table**

CREATE TABLE Employees (EmployeeID INT PRIMARY KEY,  Name VARCHAR(50), Position VARCHAR(50), BranchID INT, FOREIGN KEY (BranchID) REFERENCES Branches(BranchID));

---

**Insert a customer:**

INSERT INTO Customers VALUES ('John', 'Doe', '1234567890, '123 Main St');
INSERT INTO Accounts VALUES (1, 'Savings', 5000.00);

## College Database

- **Department Table**

CREATE TABLE Department (Department_ID INT PRIMARY KEY, Dept_Name VARCHAR(100) NOT NULL);

- **Student Table**

CREATE TABLE Student (Student_ID INT PRIMARY KEY,Name VARCHAR(100), DOB DATE, Department_ID INT,FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID));

- **Faculty Table**

CREATE TABLE faculty ( FACULTY_ID INT PRIMARY KEY, Name VARCHAR(100), Department_ID INT, FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID));

- **Course Table**

CREATE TABLE Course (Course_ID INT PRIMARY KEY, Course_Name VARCHAR(100),Credits INT,

Department_ID INT,FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID));

- **Enrollment Table**

```
CREATE TABLE Enrollment (Enrollment_ID INT PRIMARY KEY,Student_ID INT NOT
NULL,Course_ID INT NOT NULL,GRADE VARCHAR(5),FOREIGN KEY (Student_ID)
REFERENCES Student(Student_ID),FOREIGN KEY (Course_ID) REFERENCES
Course(Course_ID);
```

## 3. Applying Constraints to the College Database

- **PRIMARY KEY** – uniquely identifies a row in a table
- **FOREIGN KEY** – creates a link between two tables
- **NOT NULL** – ensures a column cannot have a NULL valu

### Department Table

```
CREATE TABLE Department(Department_ID INT PRIMARY KEY,Dept_Name
VARCHAR(100) NOT NULL);
```

### Student Table

```
CREATE TABLE Student(Student_ID INT PRIMARY KEY, Name VARCHAR(30) NOT NULL,
DOB DATE NOT NULL,department_ID INT,FOREIGN KEY (Department_ID) REFERENCES
Department(Department_ID));
```

---

### Instructor Table

```
CREATE TABLE Instructor(Instructor_ID INT PRIMARY KEY,Name VARCHAR(100) NOT
NULL,Email VARCHAR(100) UNIQUE NOT NULL,Department_ID INT NOT NULL,
FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID));
```

---

### Course Table

```
CREATE TABLE Course(Course_ID INT PRIMARY KEY,Course_Name VARCHAR(100) NOT
NULL,Credits INT NOT NULL,Department_ID INT NOT NULL,FOREIGN KEY
(Department_ID) REFERENCES Department(Department_ID) Foreign Key);
```

---

**Enrollment Table**

```
CREATE TABLE Enrollment (Enrollment_ID INT PRIMARY KEY,Student_ID INT NOT
NULL,Course_ID INT NOT NULL,GRADE VARCHAR(5),FOREIGN KEY (Student_ID)
REFERENCES Student(Student_ID),FOREIGN KEY (Course_ID) REFERENCES
Course(Course_ID));
```

---

## 4. Practicing DDL & DML Commands

### A. DDL Commands

**Add a column to `Student` table**

```
ALTER TABLE Student ADD Phone VARCHAR (15);
```

**Remove the Enrollment table**

```
DROP TABLE Enrollment;
```

**`TRUNCATE` – Delete all data but keep the structure**

```
TRUNCATE TABLE Student;
```

**`RENAME` – Rename a table**

```
RENAME TABLE Student TO Students;
```

---

### ◆ B. DML Commands

---

**✓ `INSERT` – Add new records**

```
INSERT INTO Department VALUES (1, 'Computer Science');
INSERT INTO Department VALUES (2, 'Mathematics');

-- Insert into Student
INSERT INTO Student VALUES (101, 'Alice', '02-06-2015', 1);
```

**✓ `UPDATE` – Modify existing records**

```
UPDATE Student SET Email = 'alice.cs@example.com' WHERE Student_ID = 101;
```

**✓ `DELETE` – Remove records**

```
DELETE FROM Student WHERE Student_ID = 101;
```

---

### SELECT Queries (DQL – part of DML)

```
SELECT * FROM Student;

Select students in Computer Science department
SELECT s.Name, d.Dept_Name FROM Student s JOIN Department d ON
s.Department_ID = d.Department_ID WHERE d.Dept_Name = 'Computer Science';
```

## 5. SQL Queries Using JOINs

**INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL OUTER JOIN**, SELF **JOIN**

### INNER JOIN

```
SELECT s.Student_ID, s.Name, d.Dept_Name FROM Student s INNER JOIN
Department d ON s.Department_ID = d.Department_ID;
```

### LEFT JOIN

Returns all rows from the left table, and matched rows from the right table (NULLs if no match).

**List all students and their departments (even if not assigned)**

```
SELECT s.Student_ID, s.Name, d.Dept_Name FROM Student s LEFT JOIN
Department d ON s.Department_ID = d.Department_ID;
```

---

### RIGHT JOIN

Returns all rows from the right table, and matched rows from the left (NULLs if no match).

**Example: List all departments and the students in them (even if no students)**

```
SELECT s.Name, d.Dept_Name FROM Student s RIGHT JOIN Department d ON
s.Department_ID = d.Department_ID;
```

---

### FULL OUTER JOIN

Returns all rows when there is a match in one of the tables (LEFT or RIGHT).

```
SELECT s.Name, d.Dept_Name FROM Student s LEFT JOIN Department d ON
s.Department_ID = d.Department_ID

UNION

SELECT s.Name, d.Dept_Name FROM Student s RIGHT JOIN Department d ON
s.Department_ID = d.Department_ID;
```

---

### Join with 3 Tables (JOIN )

**List student names, enrolled course names, and instructor names**

```
SELECT s.Name AS StudentName, c.Course_Name, i.Name AS InstructorName
FROM Enrollment e JOIN Student s ON e.Student_ID = s.Student_ID
JOIN Course c ON e.Course_ID = c.Course_ID JOIN Instructor i ON
c.Department_ID = i.Department_ID;
```

---

### SELF JOIN

**Show instructor pairs from the same department**

```
SELECT a.Name AS Instructor1, b.Name AS Instructor2, a.Department_ID
FROM Instructor a JOIN Instructor b ON a.Department_ID = b.Department_ID
AND a.Instructor_ID <> b.Instructor_ID;
```

## 6. SQL Queries with Aggregate Functions

---

### 📌 MAX() – Get the highest value

**Find the course with the maximum number of credits**

```
SELECT MAX(Credits) AS MaxCredits FROM Course;
```

### 📌 MIN() – Get the lowest value

**Find the earliest student DOB (youngest student)**

```
SELECT MIN(DOB) AS YoungestStudentDOB FROM Student;
```

---

### 📌 AVG() – Calculate average

**Find the average number of credits per course**

```
SELECT AVG(Credits) AS AverageCredits FROM Course;
```

---

### 📌 COUNT() – Count rows

**Count total number of students**

```
SELECT COUNT(*) AS TotalStudents FROM Student;
```

---

## Combining COUNT with GROUP BY

**Example: Count how many students are in each department**

```
SELECT d.Dept_Name, COUNT(s.Student_ID) AS StudentCount FROM Student s
JOIN Department d ON s.Department_ID = d.Department_ID GROUP BY
d.Dept_Name;
```

---

## ✅ AVG with GROUP BY

**Average credits per department**

```
SELECT d.Dept_Name, AVG(c.Credits) AS AvgCredits FROM Course c JOIN
Department d ON c.Department_ID = d.Department_ID GROUP BY d.Dept_Name;
```

## 7. Types of Integrity Constraints:

1. **PRIMARY KEY**
2. **FOREIGN KEY**
3. **UNIQUE**
4. **NOT NULL**
5. **CHECK**

---

## 📌 PRIMARY KEY

**Ensure `Student_ID` is unique and not NULL**

```
CREATE TABLE Student(Student_ID INT PRIMARY KEY,Name VARCHAR(100) NOT NULL,
DOB DATE,Department_ID INT,FOREIGN KEY (Department_ID) REFERENCES
Department(Department_ID));
```

## 📌 FOREIGN KEY

**Ensure `Department_ID` in `Student` table must exist in `Department` table**

```
CREATE TABLE Student(Student_ID INT PRIMARY KEY,Name VARCHAR(100) NOT NULL,
Email VARCHAR(100) NOT NULL,DOB DATE,Department_ID INT,FOREIGN KEY
(Department_ID) REFERENCES Department(Department_ID);
```

## 📌 UNIQUE

Ensures that all values in a column are different (no duplicates).

**Ensure that the `Email` of each student is unique**

```
CREATE TABLE Student(Student_ID INT PRIMARY KEY,Name VARCHAR(100) NOT NULL,
Email VARCHAR(100) UNIQUE NOT NULL,DOB DATE,Department_ID INT,
FOREIGN KEY (Department_ID) REFERENCES Department(Department_ID));
```

### 📌 NOT NULL

**Ensure `Name` and `Email` are always filled in the `Student` table**

```
CREATE TABLE Student(Student_ID INT PRIMARY KEY,Name VARCHAR(100) NOT NULL,
Email VARCHAR(100) NOT NULL,DOB DATE,Department_ID INT,FOREIGN KEY
(Department_ID) REFERENCES Department(Department_ID));
```

### 📌 CHECK

**Ensure that the `Credits` in the `Course` table is always between 1 and 6**

```
CREATE TABLE Course(Course_ID INT PRIMARY KEY,Course_Name VARCHAR(100) NOT
NULL,Credits INT NOT NULL,Department_ID INT,FOREIGN KEY (Department_ID)
REFERENCES Department(Department_ID),CHECK (Credits >=1 AND Credits <= 6));
```

This will prevent the insertion of a course with less than 1 or more than 6 credits.

---

## 8. Perform the following operation for demonstrating the insertion, updation and deletion using the referential integrity constraints

You can combine multiple constraints in one `CREATE` statement:

```
CREATE TABLE Instructor2(Instructor_ID INT PRIMARY KEY, Name VARCHAR(100)
NOT NULL,Department_ID INT NOT NULL,FOREIGN KEY (Department_ID) REFERENCES
Department(Department_ID),CHECK(Department_ID > 0));
```

**Referential Integrity Constraints in Action**

### 📌Insertion

**Insert Department:**

```
INSERT INTO Department (Department_ID, Dept_Name) VALUES (1, 'Computer
Science');
```

**Insert Student with Referential Integrity:**

```
INSERT INTO Student VALUES (101, 'John Doe', '20-09-2015',1);
```

### 📌 Update the department of a student to a new valid department ID.

```
UPDATE Student SET Department_ID = 2 WHERE Student_ID = 101;
```

📌 **Delete a Department with Referential Integrity**

```
DELETE FROM Department WHERE Department_ID = 1;
```

**Handling Deletion with Referential Integrity (CASCADE, SET NULL)**

To delete a department and automatically handle the deletion of dependent students (or set their `Department_ID` to NULL), we can use **ON DELETE CASCADE** or **ON DELETE SET NULL** in the foreign key constraint definition.

1. **Modify Foreign Key to Cascade Deletion:**

```
ALTER TABLE Student DROP CONSTRAINT fk_department;

ALTER TABLE Student ADD CONSTRAINT fk_department FOREIGN KEY
(Department_ID)REFERENCES Department(Department_ID)ON DELETE CASCADE;
```

With `ON DELETE CASCADE`, if we delete a `Department`, the related students will automatically be deleted from the `Student` table.

2. **Use ON DELETE SET NULL:**

If you want to **remove the department assignment** but not delete the student, you can modify the foreign key constraint like this:

```
ALTER TABLE Student DROP CONSTRAINT fk_department;

ALTER TABLE Student ADD CONSTRAINT fk_department FOREIGN KEY
(Department_ID) REFERENCES Department(Department_ID)ON DELETE SET NULL;
```

**Delete a Department with CASCADE**

If we had the `ON DELETE CASCADE` in place, we could now delete the department and have all associated students deleted automatically:

```
-- Delete department, and automatically delete all students in it
DELETE FROM Department
WHERE Department_ID = 1;
```

---

## 9. Write the query for creating the users and their roles
## Creating a User

```
CREATE USER john_doe IDENTIFIED BY securepassword DEFAULT TABLESPACE users
TEMPORARY TABLESPACE temp;
```

- `DEFAULT TABLESPACE`: Specifies the tablespace for storing user data.
- `TEMPORARY TABLESPACE`: Specifies the temporary tablespace for storing intermediate results during query processing.

### Granting Privileges to the User

After creating the user, you must grant necessary privileges so that the user can perform actions like creating tables, querying data, etc. These are system privileges (e.g., `CREATE SESSION`, `SELECT`, `INSERT`), and object privileges (e.g., `SELECT` on a specific table).

### Grant Basic Privileges

```
GRANT CREATE SESSION, CREATE TABLE TO john_doe;
```

---

## Creating Roles

A **role** in Oracle is a named collection of privileges. You can create roles and then assign them to users. Roles allow you to group related privileges together and manage them efficiently.

### Create a Role

```
CREATE ROLE admin_role;
```

---

### Grant Privileges to the Role

```
GRANT SELECT, INSERT, UPDATE ON employees TO admin_role;
GRANT CREATE SESSION, CREATE TABLE TO admin_role;
```

## Assigning Roles to Users

```
GRANT admin_role TO john_doe;
```

## Granting the `CONNECT` Privilege to the User

```
GRANT CONNECT TO john_doe;
```

---

## Revoking Roles and Privileges

```
REVOKE admin_role FROM john_doe;
```

### Revoke a Privilege

```
REVOKE CREATE SESSION FROM john_doe;
```

This removes the `CREATE SESSION` privilege from `john_doe`.

---

## Full Example: Creating Users, Roles, and Assigning Privileges

```
sql
CopyEdit
-- 1. Create a User
```

```
CREATE USER john_doe IDENTIFIED BY securepassword
DEFAULT TABLESPACE users
TEMPORARY TABLESPACE temp;

-- 2. Grant basic privileges to the user
GRANT CREATE SESSION, CREATE TABLE TO john_doe;

-- 3. Create a Role
CREATE ROLE admin_role;

-- 4. Grant privileges to the role
GRANT SELECT, INSERT, UPDATE ON employees TO admin_role;
GRANT CREATE SESSION, CREATE TABLE TO admin_role;

-- 5. Assign the Role to the User
GRANT admin_role TO john_doe;

-- 6. Grant CONNECT role to user
GRANT CONNECT TO john_doe;
```

---

## 10. Querying (using ANY, ALL, IN, EXISTS, NOT EXISTS, UNION, INTERSECT, Constraints)

- **ANY**: Find students who have a score higher than **any** student in department 1.

  ```
  SELECT Name, dob FROM Student WHERE dob > ANY (SELECT dob FROM Student
  WHERE Department_ID = 1);
  ```

- **ALL**: Find students who have a score higher than **all** students in department 1.

```
SELECT Name, dob FROM Student WHERE dob > ALL (SELECT Score FROM Student
WHERE Department_ID = 1);
```

---

- ### IN Operator

The `IN` operator is used to compare a value to a list of values. It is a shorthand for multiple `OR` conditions.

Find all students who belong to departments 1 or 2:

```
SELECT Name FROM Student WHERE Department_ID IN (1, 2);
```

You can also use a subquery with `IN` to check if a value is present in the result set of another query.

```
SELECT Name FROM Student WHERE Department_ID IN (SELECT Department_ID FROM
Department WHERE Dept_Name = 'Computer Science');
```

---

- ### EXISTS and NOT EXISTS Operators

Find students who have enrolled in at least one course:

```
SELECT Name FROM Student s WHERE EXISTS (SELECT 1 FROM Enrollment e WHERE
e.Student_ID = s.Student_ID);
```

Find students who have **not** enrolled in any course:

```
SELECT Name FROM Student s WHERE NOT EXISTS (SELECT 1 FROM Enrollment e
WHERE e.Student_ID = s.Student_ID);
```

---

- **UNION and UNION ALL Operators**

1. Get all students from departments 1 and 2 (removing duplicates):

```
SELECT Name FROM Student WHERE Department_ID = 1
UNION
SELECT Name FROM Student WHERE Department_ID = 2;
```

2. Get all students from departments 1 and 2 (including duplicates):

```
SELECT Name FROM Student WHERE Department_ID = 1
UNION ALL
SELECT Name FROM Student WHERE Department_ID = 2;
```

---

- **INTERSECT Operator**

The `INTERSECT` operator returns the common rows from the result sets of two queries.

Find students who are enrolled in both Course 101 and Course 102:

```
SELECT Student_ID FROM Enrollment WHERE Course_ID = 101
INTERSECT
SELECT Student_ID FROM Enrollment WHERE Course_ID = 102;
```

---

## Constraints (PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, CHECK)

- **PRIMARY KEY**: Uniquely identifies each record in a table. It cannot contain `NULL` values.
- **FOREIGN KEY**: Ensures referential integrity by linking two tables.
- **UNIQUE**: Ensures all values in a column are unique.
- **NOT NULL**: Ensures that a column cannot have a `NULL` value.
- **CHECK**: Ensures that all values in a column meet a specific condition.

### Example: Creating a Table with Constraints

```
CREATE TABLE Student (
    Student_ID INT PRIMARY KEY,               -- Primary Key
    Name VARCHAR(100) NOT NULL,               -- Not NULL
    Email VARCHAR(100) UNIQUE,                -- Unique constraint
    Department_ID INT,
    CONSTRAINT fk_department FOREIGN KEY (Department_ID) REFERENCES
Department(Department_ID),  -- Foreign Key
    CHECK (Score >= 0 AND Score <= 100)      -- Check constraint
);
```

## 7. Combining Everything in a Query

You can combine many of these operators to create complex queries. Here's an example:

### Complex Example:

Find all students who are either enrolled in a course offered by department 1 **or** have a score above 80, but exclude students who have failed any course (score < 50). Use `IN`, `EXISTS`, `NOT EXISTS`, and `UNION`.

```
SELECT Name FROM Student WHERE Department_ID = 1
UNION
SELECT Name
FROM Student
WHERE Score > 80
AND NOT EXISTS (
    SELECT 1 FROM Enrollment e
    JOIN Course c ON e.Course_ID = c.Course_ID
    WHERE e.Student_ID = Student.Student_ID AND c.Score < 50
);
```

## 10. Queries using Aggregate Functions, group By, Having and Creaion and Dropping of Views

Aggregate functions in SQL allow you to perform calculations on a set of values and return a single result. The most common aggregate functions are:

- **COUNT()**: Counts the number of rows.
- **SUM()**: Returns the sum of values.
- **AVG()**: Returns the average value.
- **MIN()**: Returns the minimum value.
- **MAX()**: Returns the maximum value.

1. **COUNT**: Get the total number of students in the `Student` table.

```
SELECT COUNT(*) AS TotalStudents FROM Student;
```

2. **SUM**: Find the total score of all students.

```
SELECT SUM(Score) AS TotalScore FROM Student;
```

3. **AVG**: Get the average score of students.

```
SELECT AVG(Score) AS AverageScore FROM Student;
```

4. **MIN and MAX**: Get the minimum and maximum score.

```
SELECT MIN(Score) AS MinScore, MAX(Score) AS MaxScore FROM Student;
```

---

### GROUP BY Clause

Find the average score for each department:

```
SELECT Department_ID, AVG(Score) AS AverageScore FROM Student
GROUP BY Department_ID;
```

---

### HAVING Clause

Find the departments that have an average score greater than 70:

```
SELECT Department_ID, AVG(Score) AS AverageScore FROM Student
GROUP BY Department_ID HAVING AVG(Score) > 70;
```

---

## Creating Views

Create a view to get the list of students who have a score greater than 80:

```
CREATE VIEW HighScoringStudents AS SELECT Name, Score FROM Student
WHERE Score > 80;
```

Once the view is created, you can query it just like a regular table:

```
SELECT * FROM HighScoringStudents;
```

---

### Dropping Views

```
DROP VIEW HighScoringStudents;
```

---