Assignment-4: DBMS

Topics: Relational Algebra, ER Models, FDs, and Normalization

1. Basic Operations of Relational Algebra

Relational Algebra provides a set of operations to manipulate relations (tables) in a database.

Operation	Description	Example
SELECT (σ)	Filters rows based on a condition.	σDept = 'CSE'(STUDENT) — selects all CSE students.
PROJECT (π)	Selects specific columns.	πName,Dept(STUDENT) — shows only Name and Dept columns.
UNION (∪)	Combines tuples from two relations.	STUDENT1 ∪ STUDENT2
SET DIFFERENCE (−)	Shows tuples in one relation but not in the other.	STUDENT − ALUMNI
CARTESIAN PRODUCT (×)	Combines every tuple of one relation with another.	STUDENT × DEPARTMENT
RENAME (ρ)	Renames relation or attributes.	ρ(S)(STUDENT) — renames STUDENT as S
JOIN (⋈)	Combines related tuples using common attributes.	STUDENT ⋈STUDENT.DeptID = DEPARTMENT.DeptIDDEPARTMENT

2. Relational Algebra Expressions

Relations:

```
STUDENT(RollNo, Name, DeptID)
DEPARTMENT(DeptID, DeptName)
```

(a) Students in 'CSE' department:

πName(σDeptName='CSE'(STUDENT ⋈STUDENT.DeptID = DEPARTMENT.DeptID DEPARTMENT))

(b) Departments with no students:

πDeptName(DEPARTMENT) − πDeptName(STUDENT ⋈ DEPARTMENT)

3. ER Diagram for Online Shopping System

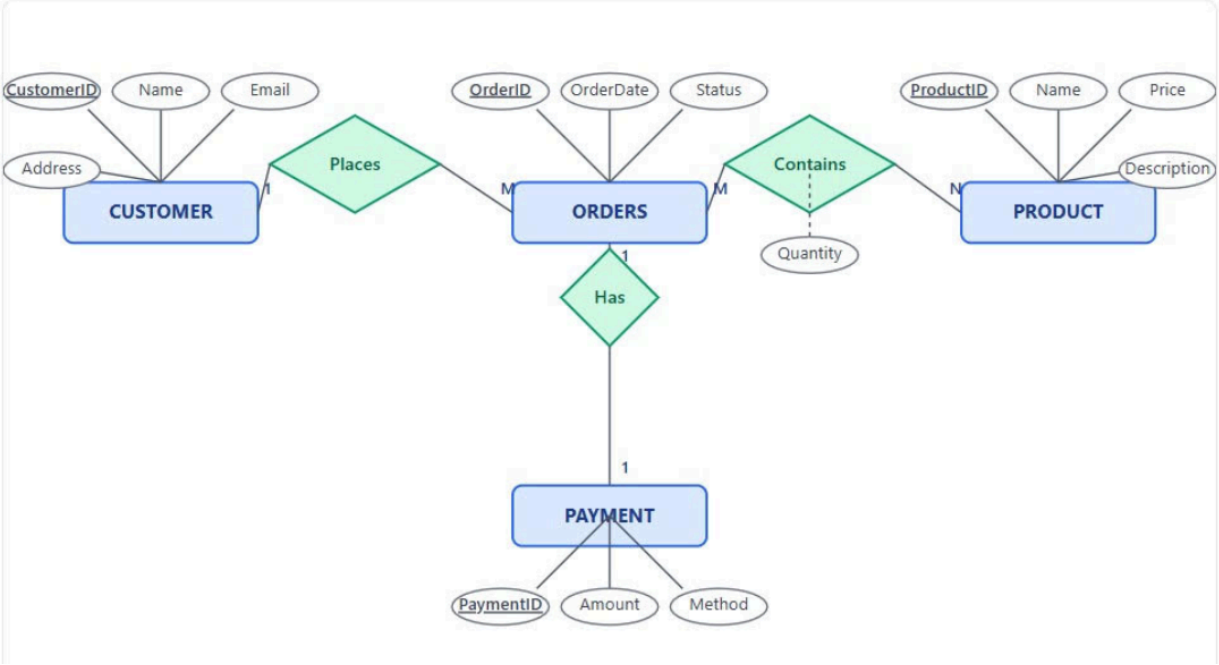
Entities:

- **Customer**(CustomerID, Name, Email, Phone)
- **Product**(ProductID, ProductName, Price, StockQty)
- **Order**(OrderID, OrderDate, CustomerID)
- **Payment**(PaymentID, OrderID, Amount, Date, Mode)

Relationships:

- Customer places Order → *One-to-Many*
- Order contains Product → *Many-to-Many* (Resolved using OrderDetails)
- Order makes Payment → *One-to-One*

ER Model:



Relational Schema:

```
CUSTOMER(CustomerID PK, Name, Email, Phone)
PRODUCT(ProductID PK, ProductName, Price, StockQty)
ORDER(OrderID PK, OrderDate, CustomerID FK)
ORDER_DETAILS(OrderID FK, ProductID FK, Quantity)
PAYMENT(PaymentID PK, OrderID FK, Amount, Date, Mode)
```

```
erDiagram
    CUSTOMER ||--o{ ORDER : places
    CUSTOMER {
        int CustomerID PK
        string Name
        string Email
        string Phone
    }
    ORDER ||--|{ ORDER_DETAILS : contains
```

```
    ORDER ||--|| PAYMENT : makes
    ORDER {
        int OrderID PK
        date OrderDate
        int CustomerID FK
    }
    PRODUCT ||--o{ ORDER_DETAILS : "included in"
    PRODUCT {
        int ProductID PK
        string ProductName
        decimal Price
        int StockQty
    }
    ORDER_DETAILS {
        int OrderID FK
        int ProductID FK
        int Quantity
    }
    PAYMENT {
        int PaymentID PK
        int OrderID FK
        decimal Amount
        date Date
        string Mode
    }
```

4. Strong vs Weak Entity Sets

Feature	Strong Entity	Weak Entity
Existence	Independent	Depends on strong entity
Key	Has primary key	Has partial key

Example	STUDENT(StudentID, Name)	DEPENDENT(StudentID FK, DependentName)
Relationship	Identifying not needed	Needs identifying relationship

```
erDiagram
    STUDENT ||--o{ DEPENDENT : has
    STUDENT {
        int StudentID PK
        string Name
    }
    DEPENDENT {
        int StudentID FK
        string DependentName
    }
```

5. Functional Dependency (FD)

A **Functional Dependency** (FD) is a relationship between attributes such that one attribute uniquely determines another.

Example:

If A → B, then for every value of A, there is exactly one value of B.

Candidate Key Identification Example:

```
STUDENT(StudentID, Name, Course, Instructor)
FDs: StudentID → Name, Course, Instructor
```

Here, StudentID uniquely identifies all attributes → Candidate Key = {StudentID}.

```
graph LR
    A["StudentID"] --> B["Name"]
    A --> C["Course"]
    A --> D["Instructor"]
```

6. Relation R(A, B, C, D, E)

FDs: {A → B, B → C, A → D, D → E}

```
graph LR
    A["A"] ⟶ B["B"]
    B ⟶ C["C"]
    A ⟶ D["D"]
    D ⟶ E["E"]
```

Step 1: Find Closure of A

A^+ = {A, B, C, D, E} → A is a **Candidate Key**.

Step 2: Normalize

- **1NF:** Already atomic.
- **2NF:** No partial dependency (A is single attribute key).
- **3NF:** Each non-prime attribute depends only on key.

Decomposition:

```
R1(A, B, C, D, E) →
R1(A, B, D)
R2(B, C)
R3(D, E)
```

7. 1NF, 2NF, 3NF Differences

Normal Form	Condition	Example
1NF	No multivalued attributes.	StudentID, Subjects {Math, CS}
2NF	1NF + No partial dependency on part of composite key.	(RollNo, Subject) → Marks
3NF	2NF + No transitive dependency.	RollNo → Dept → HOD

```
graph TD
    A["Unnormalized"] → B["1NF: Atomic Values"]
    B → C["2NF: No Partial Dependencies"]
    C → D["3NF: No Transitive Dependencies"]
    D → E["BCNF: Stricter Key Dependencies"]
```

8. Relational Algebra

(a) Employees with salary > 50,000:

σSalary > 50000(EMPLOYEE)

(b) Departments with >5 employees:

πDeptID(σCOUNT(EmpID) > 5(EMPLOYEE))

(Using aggregation equivalent).

9. Library Management System (ER + Schema)

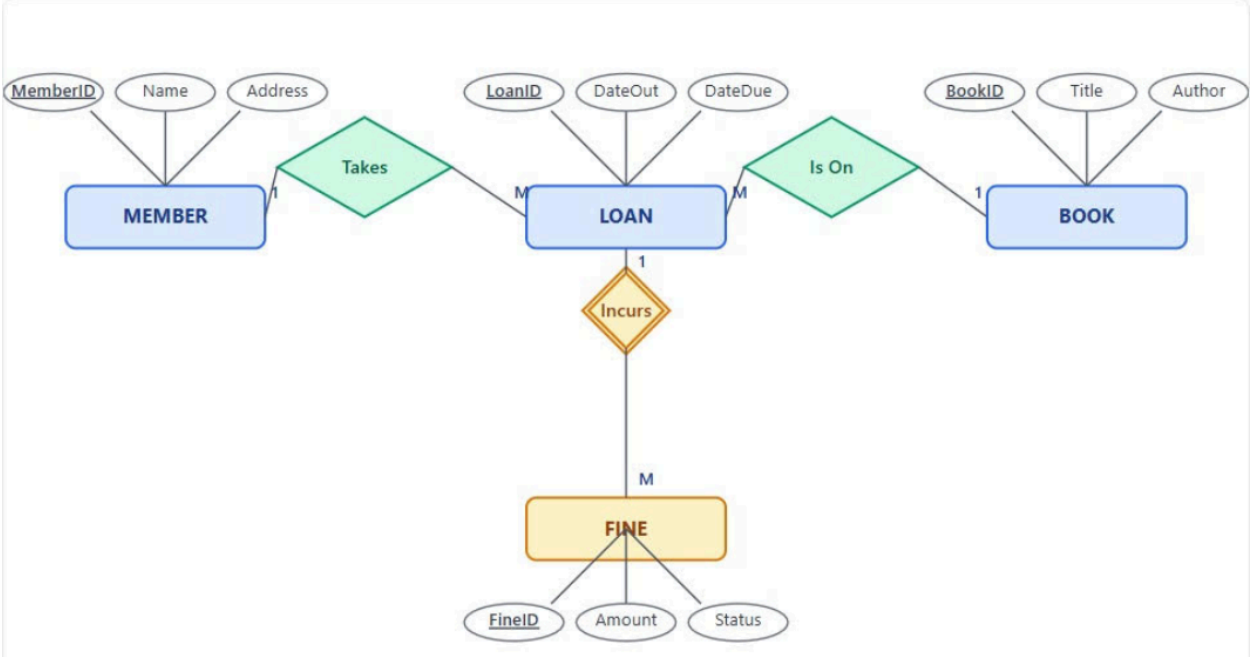
Entities:

- BOOK(BookID, Title, Author, Publisher)
- MEMBER(MemberID, Name, Address)
- LOAN(LoanID, BookID, MemberID, IssueDate, ReturnDate)
- FINE(FineID, LoanID, Amount, DatePaid)

Relationships:

- MEMBER borrows BOOK (via LOAN)
- LOAN may have FINE

ER Model:



Schema:

```
BOOK(BookID PK, Title, Author, Publisher)
MEMBER(MemberID PK, Name, Address)
LOAN(LoanID PK, BookID FK, MemberID FK, IssueDate, ReturnDate)
FINE(FineID PK, LoanID FK, Amount, DatePaid)
```

```
erDiagram
    MEMBER ||--o{ LOAN : borrows
    BOOK ||--o{ LOAN : "borrowed via"
    LOAN ||--o| FINE : "may have"

    BOOK {
        int BookID PK
        string Title
        string Author
        string Publisher
    }
```

```
MEMBER {
    int MemberID PK
    string Name
    string Address
}

LOAN {
    int LoanID PK
    int BookID FK
    int MemberID FK
    date IssueDate
    date ReturnDate
}

FINE {
    int FineID PK
    int LoanID FK
    decimal Amount
    date DatePaid
}
```

10. Relation R(StudentID, CourseID, InstructorID, InstructorName, CourseName)

FDs: {StudentID, CourseID → InstructorID; InstructorID → InstructorName; CourseID → CourseName}

```
graph LR
    A["StudentID, CourseID"] --> B["InstructorID"]
    B --> C["InstructorName"]
    D["CourseID"] --> E["CourseName"]
```

3NF Decomposition:

```
R1(StudentID, CourseID, InstructorID)
R2(InstructorID, InstructorName)
R3(CourseID, CourseName)
```

→ Each table now satisfies 3NF (no transitive dependencies).

11. Redundancy & Anomalies Reduction

Redundancy: Repetition of data (e.g., same InstructorName stored multiple times).

Anomalies:

- **Insertion:** Cannot insert course without student.
- **Updation:** Change in instructor name requires multiple updates.
- **Deletion:** Deleting last student may delete course info.

Normalization splits data into logical tables → removes these anomalies.

```
graph TD
   A["Unnormalized Table"] --> B["Redundancy Issues"]
   B --> C["Insertion Anomalies"]
   B --> D["Update Anomalies"]
   B --> E["Deletion Anomalies"]
   F["Normalization Process"] --> G["Decompose Tables"]
   G --> H["Remove Redundancy"]
   H --> I["Eliminate Anomalies"]
```

12. Relational Algebra Queries

Given:

```
EMPLOYEE(EmpID, Name, DeptID, Salary)
DEPARTMENT(DeptID, DeptName, Location)
```

(a) Average salary per department:

γDeptID, AVG(Salary)(EMPLOYEE)

(b) Names of employees in 'IT' dept:

πName(σDeptName='IT'(EMPLOYEE ⋈EMPLOYEE.DeptID=DEPARTMENT.DeptID DEPARTMENT))

13. STUDENT_INFO Decomposition

FDs: {StudentID → StudentName, CourseID → CourseName, InstructorID → InstructorName, CourseID → InstructorID}

```
graph LR
    A["StudentID"] ⟶ B["StudentName"]
    C["CourseID"] ⟶ D["CourseName"]
    C ⟶ E["InstructorID"]
    E ⟶ F["InstructorName"]
```

Anomalies:

- Redundant instructor and course names.
- Update anomalies.

Normalization:

1. **1NF:** Already atomic.
2. **2NF:** Key = (StudentID, CourseID). Split partial dependencies.

```
R1(StudentID, StudentName)
R2(CourseID, CourseName, InstructorID)
```

3. **3NF:** InstructorID → InstructorName → Separate table.

```
R3(InstructorID, InstructorName)
```

Final 3NF:

```
R1(StudentID, StudentName)
R2(CourseID, CourseName, InstructorID)
R3(InstructorID, InstructorName)
```

14. Relation R(A, B, C, D, E, F)

FDs: {A → B, B → C, CD → E, E → F, F → A}

```
graph LR
   A["A"] ⟶ B["B"]
   B ⟶ C["C"]
   CD["C, D"] ⟶ E["E"]
   E ⟶ F["F"]
   F ⟶ A
```

Step 1: Closure

From F → A → B → C, CD → E → F forms a loop.

Candidate Key = **{C, D}**.

Step 2: Highest Normal Form

There are transitive dependencies → Not in 3NF → Normalize.

3NF Decomposition:

```
R1(A, B, C)
R2(C, D, E)
R3(E, F)
```

15. EMP_PROJECT(EmpID, EmpName, ProjectID, ProjectName, DepartmentID, DepartmentName)

FDs: {EmpID → EmpName, ProjectID → ProjectName, DepartmentID → DepartmentName, EmpID → DepartmentID}

```
graph LR
    A["EmpID"] --> B["EmpName"]
    A --> C["DepartmentID"]
    C --> D["DepartmentName"]
    E["ProjectID"] --> F["ProjectName"]
```

Candidate Key: {EmpID, ProjectID}

Normalization:

1. **1NF:** Atomic.
2. **2NF:** Remove partial dependency on EmpID.

```
R1(EmpID, EmpName, DepartmentID)
R2(ProjectID, ProjectName)
```

3. **3NF:** DepartmentID → DepartmentName ⇒ separate table.

```
R3(DepartmentID, DepartmentName)
R4(EmpID, ProjectID)
```

Final 3NF:

```
R1(EmpID, EmpName, DepartmentID)
R2(ProjectID, ProjectName)
R3(DepartmentID, DepartmentName)
R4(EmpID, ProjectID)
```

```
erDiagram
    EMPLOYEE ||--o{ EMP_PROJECT : works_on
    EMPLOYEE ||--|| DEPARTMENT : belongs_to
    PROJECT ||--o{ EMP_PROJECT : has

    EMPLOYEE {
        int EmpID PK
```

```
    string EmpName
    int DepartmentID FK
}

PROJECT {
    int ProjectID PK
    string ProjectName
}

DEPARTMENT {
    int DepartmentID PK
    string DepartmentName
}

EMP_PROJECT {
    int EmpID FK
    int ProjectID FK
}
```

Normalization eliminates redundant department and project data.