Use case:

Loan Database Management System

Aim: To prepare Loan Database management system.

<u>STEP 1:Implementation of ER Model and Schema</u>

<u>Steps involved in er model:</u>

1. Identify Entities Identify the main objects or entities that will be represented in the diagram. Entities typically represent real-world objects, people, or concepts (e.g., Employee, Customer, Product).

2. Determine Attributes For each entity, determine the attributes or properties that describe the entity (e.g., Employee attributes could be EmployeeID, Name, and Department).

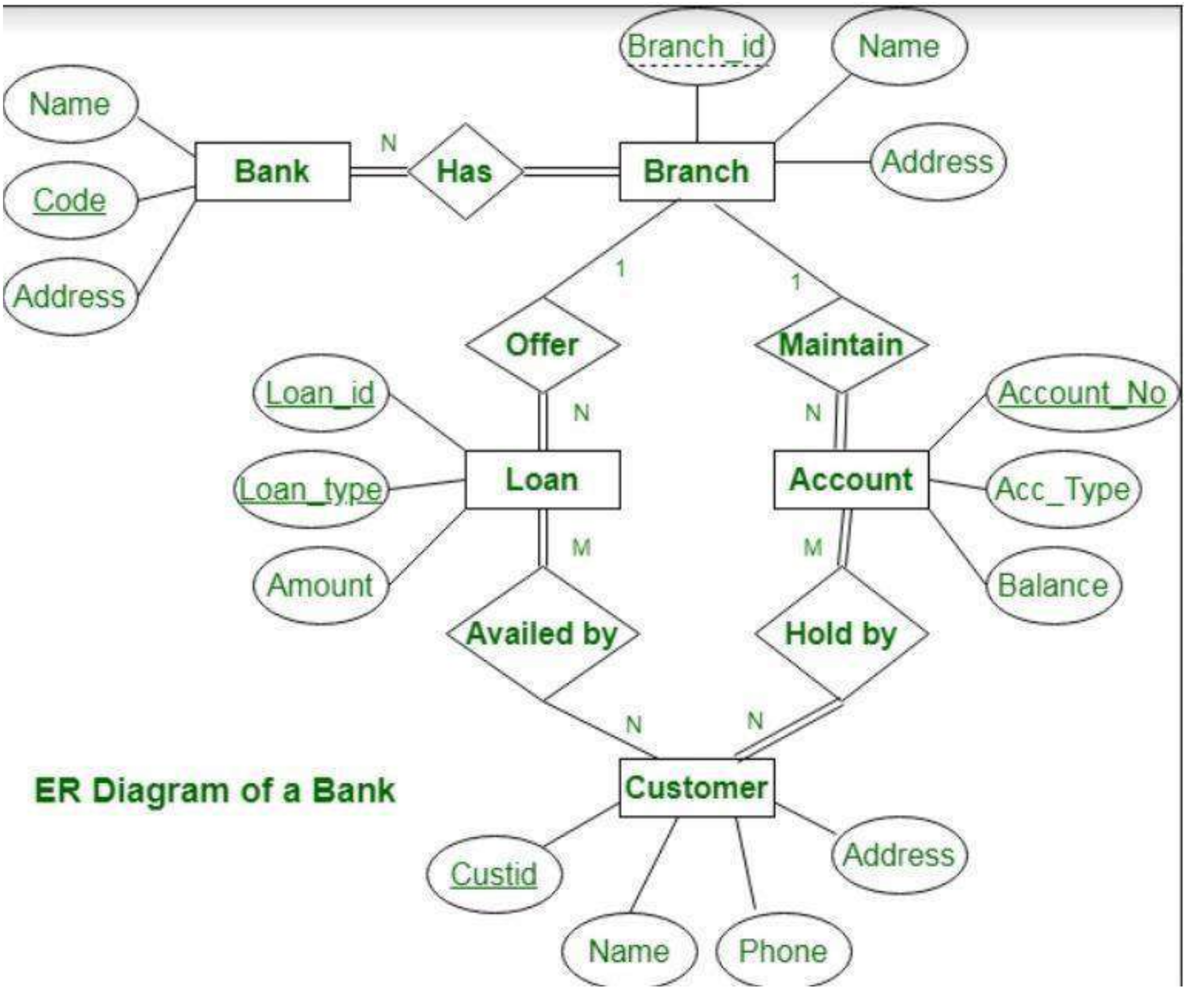
3. Identify Relationships Identify how the entities are related to one another (e.g., Employee works for Department). Relationships describe the associations between entities.

4. Determine Cardinality Define the cardinality of each relationship (i.e., one-to-one, one-to-many, or many-to-many). This shows how many instances of one entity relate to instances of another entity.

5. Draw the Diagram Draw Entities: Use rectangles to represent each entity. Add Attributes: Use ovals to represent attributes and connect them to the corresponding entity. Draw Relationships: Use diamonds to represent relationships and connect the entities involved. Indicate Cardinality: Use numbers or symbols (1, N) next to the relationship lines to show cardinality.

OUTPUT:

ER MODEL FOR LOAN DATABASE MANAGEMENT SYSTEM:



STEP 2 : Conversion of ER Model To Relation Model

Converting an Entity-Relationship (ER) Model to a Relational Model Is a Crucial Step in Designing a Relational Database. This Process Involves Translating the Entities, Attributes, and Relationships Defined in the ER Diagram Into Tables, Columns, and Keys in a Relational Schema. Below Are the Comprehensive Steps To Perform This Conversion Effectively.

Steps To Convert an ER Model to a Relational Model:

1: Mapping of Regular Entity

Types2: Mapping of Weak Entity

Types

3: Mapping of Binary 1:1 Relation Types

4: Mapping of Binary 1:N Relationship

Types. 5: Mapping of Binary M:N

Relationship Types. 6: Mapping of

Multivalued Attributes.

7: Mapping of N-Ary Relationship …

ER Model Components

1. **Entities: Identify the main entities involved. For a loan database, you might have:**
   - **Customer**
   - **Loan**
   - **Payment**
   - **Loan Officer**
2. **Attributes: Define attributes for each entity. For example:**
   - **Customer: CustomerID, Name, Address, PhoneNumber**
   - **Loan: LoanID, Amount, InterestRate, StartDate, CustomerID (foreign key)**
   - **Payment: PaymentID, Amount, PaymentDate, LoanID (foreign key)**
   - **Loan Officer: OfficerID, Name, Department**
3. **Relationships: Determine how entities relate to each other.**
   - **A Customer can have multiple Loans (One-to-Many).**
   - **A Loan can have multiple Payments (One-to-Many).**
   - **A Loan is managed by a Loan Officer (Many-to-One).**

Steps to Convert to Relational Model:

Create Tables for Entities: Each entity becomes a table. The primary key (PK) is identifiedfor each table.

Syntax For Er model:

```
CREATE TABLE Customer (
    CustomerID INT PRIMARY
    KEY,Name
    VARCHAR(255), Address
    VARCHAR(255),
    PhoneNumber
    VARCHAR(15)
);

CREATE TABLE Loan (
    LoanID INT PRIMARY
    KEY,Amount
    DECIMAL(10, 2),
    InterestRate DECIMAL(5,
    2),StartDate DATE,
    EndDate DATE,
    CustomerID INT,
    FOREIGN KEY (CustomerID) REFERENCES Customer(CustomerID)
);
CREATE TABLE Payment (
    PaymentID INT PRIMARY
    KEY,
    PaymentDate DATE,
    Amount DECIMAL(10,
    2),LoanID INT,
    FOREIGN KEY (LoanID) REFERENCES Loan(LoanID)
);
```

2. **Define Relationships: Relationships are established through foreign keys (FK).**

  - **In the Loan table, CustomerID is a foreign key referencing Customer(CustomerID), indicating that each loan belongs to one customer.**
  - **In the Payment table, LoanID is a foreign key referencing Loan(LoanID),indicating that each payment is associated with one loan.**

Summary of Relational Model:

- **Customer Table:**
  - **Holds customer details.**
- **Loan Table:**
  - **Holds loan details and includes a reference to the customer who took theloan.**
- **Payment Table:**
  - **Holds payment details and includes a reference to the loan for which the paymentwas made.**

STEP 3 : Implementation of DDL,DML,DCL,TCL, COMMANDS .

DDL COMMANDS:

CREATE :To Add a New Object to the

Database. ALTER : To Change the Structure of

the Database.

DROP : To Remove an Existing Object From the Database. ...

TRUNCATE : To Remove all Records From a Table, Including the Space Allocated To Store This Data.

DML_COMMANDS:

CREATE : To Add a New Object to the

Database. ALTER : To Change the Structure of

the Database.

DROP : To Remove an Existing Object From the Database. ...

TRUNCATE : To Remove all Records From a Table, Including the Space Allocated To Store This Data.

DCL COMMANDS :

DCL Includes Two Commands, GRANT and REVOKE

TCL COMMANDS :

Commit, Rollback, and Savepoint. TCL Commands Are Important for Maintaining ACID

Properties. These Commands Allow You To Commit or Discard Changes, Manage Savepoints, and Control theOverall Flow of Data Modifications.

DDL SYNTAX AND OUTPUT:

CREATE:

Create table loandatabase(sno number(20),

branchname varchar(20),customername varchar(20),

cust_id int,loan_id int);

OUTPUT:

```
Name                                      Null?    Type
----------------------------------------- -------- ---------------
SNO                                                NUMBER(20)
BRANCHNAME                                         VARCHAR2(10)
CUSTOMERNAME                                       VARCHAR2(10)
CUSTID                                             NUMBER(38)
LOANID                                             NUMBER(38)
```

ALTER:

alter table loandatabase modify customername varch

OUTPUT:

```
Name                                      Null?    Type
----------------------------------------- -------- ------------------
SNO                                                NUMBER(20)
BRANCHNAME                                         VARCHAR2(10)
CUSTOMERNAME                                       VARCHAR2(15)
CUSTID                                             NUMBER(38)
LOANID                                             NUMBER(38)
```

DML SYNTAX AND OUTPUT:

INSERT:

SQL> insert into loandatabase values(1,'kadapa','meena',123,234);1

row created.

SQL> insert into loandatabase values(2,'nellore','shalini',345,564);1

row created.

SQL> insert into loandatabase values(3,'guntur','bhoomi',435,657);1

row created.

SQL> insert into loandatabase values(4,'avadi','harshitha',457,567);

1 row created.

SQL> insert into loandatabase values(5,'produttur','teja',987,908); 1

row created.

SQL> insert into loandatabase values (6,'annanur','siri',409,108);
1 row created.

OUTPUT:

SNO	BRANCHNAME	CUSTOMERNAME	CUSTID	LOANID
1	kadapa	meena	597	234
2	nellore	shalini	345	564
3	guntur	bhoomi	435	657
4	kavali	harshitha	457	567
5	produttur	teja	987	908
6	annanur	siri	409	108

UPDATE:

SQL> update loandatabase set custid=123 where loanid=234;

SQL> update loandatabase set custid=597 where loanid=234;

SQL> update loandatabase set branchname='kavali' where custid=457;

OUTPUT:

SNO	BRANCHNAME	CUSTOMERNAME	CUSTID	LOANID
1	kadapa	meena	597	234
2	nellore	shalini	345	564
3	guntur	bhoomi	435	657
4	kavali	harshitha	457	567
5	produttur	teja	987	908

DELETE:

SQL> delete from loandatabase where customername='siri';

OUTPUT:

SNO	BRANCHNAME	CUSTOMERNAME	CUSTID	LOANID
1	kadapa	meena	597	234
2	nellore	shalini	345	564
3	guntur	bhoomi	435	657
4	kavali	harshitha	457	567
5	produttur	teja	987	908

SELECT:

SQL> select branchname from loandatabase;

OUTPUT:

```
BRANCHNAME
----------
kadapa
nellore
guntur
kavali
produttur
```

SQL> select distinct custid from loandatabase;

OUTPUT:

```
  CUSTID
--------
     597
     345
     435
     457
     987
```

SQL> select*from loandatabase where loanid between 657 and 908;

OUTPUT:

```
 SNO BRANCHNAME CUSTOMERNAME         CUSTID     LOANID
---- ---------- --------------- ---------- ----------
   3 guntur     bhoomi                  435        657
   5 produttur  teja                    987        908
```

TCL SYNTAX AND OUTPUT:

SQL> commit;

Commit complete.

SQL> savepoint k1;

Savepoint created.

SQL> rollback to

k1;Rollback

complete.

Step 4:Aggregate Functions

1. SUM: Calculate total loan amount

SELECT SUM(LoanAmount) AS

TotalLoanAmountFROM Loan;

2. AVG: Calculate average borrower age

SELECT AVG(DATEDIFF(CURRENT_DATE, DateOfBirth) / 365.25) AS AvgAge

FROM Borrower;

3. MAX: Find the largest loan amount

SELECT MAX(LoanAmount) AS

LargestLoanAmountFROM Loan;

4. MIN: Find the smallest loan amount

SELECT MIN(LoanAmount) AS

SmallestLoanAmountFROM Loan;

5. COUNT: Count number of borrowers

SELECT COUNT(BorrowerID) AS

NumBorrowersFROM Borrower;

Join Operations

1. INNER JOIN: Join borrowers with loans

SELECT B.FirstName, B.LastName,

L.LoanAmountFROM Borrower B

INNER JOIN Loan L ON B.BorrowerID = L.BorrowerID;

2. LEFT JOIN: Join borrowers with loans (include borrowers without

loans)SELECT B.FirstName, B.LastName, L.LoanAmount

FROM Borrower B

LEFT JOIN Loan L ON B.BorrowerID = L.BorrowerID;

3. RIGHT JOIN: Join borrowers with loans (include loans without

borrowers)SELECT B.FirstName, B.LastName, L.LoanAmount

FROM Borrower B

RIGHT JOIN Loan L ON B.BorrowerID = L.BorrowerID;

4. FULL OUTER JOIN: Join borrowers with loans (include borrowers and loans without matches)SELECT B.FirstName, B.LastName, L.LoanAmount

FROM Borrower B

FULL OUTER JOIN Loan L ON B.BorrowerID = L.BorrowerID;

Step 5:Nested Queries (Subqueries):

Nested queries, or subqueries, are useful when you need to use the result of one query as input toanother query.

Let's assume the following tables:

1. **Customers**

customer_id	name	email
1	John Doe	john@example.com
2	Jane Smith	jane@example.com

2. **Loans**

loan_id	customer_id	loan_amount	loan_date
101	1	5000	2023-08-01
102	2	10000	2023-09-15

3. **Payments**

payment_id	loan_id	payment_date	amount_paid
1	101	2023-09-01	500
2	101	2023-09-10	1000
3	102	2023-09-20	2000

```
SELECT name, email
FROM Customers
WHERE customer_id IN (
    SELECT customer_id
    FROM Loans
    WHERE loan_id IN (
```

```
        SELECT loan_id
        FROM Payments
        GROUP BY loan_id
        HAVING SUM(amount_paid) > 1000
    )
);
```

Output

Based on the sample data, the result might look like this:

name	email
John Doe	john@example.com
Jane Smith	jane@example.com

This nested query structure can be used to fetch any complex relationship between loans, payments, and customers based on specific criteria.

JOIN Query

```
SELECT
    c.name,
    c.email,
    l.loan_amount,
    l.loan_date,
    SUM(p.amount_paid) AS total_payments
FROM Customers c
JOIN Loans l ON c.customer_id = l.customer_id
LEFT JOIN Payments p ON l.loan_id = p.loan_id
GROUP BY c.customer_id, l.loan_id;
```

Output

name	email	loan_amount	loan_date	total_payments
John Doe	john@example.com	5000	2023-08-01	1500
Jane Smith	jane@example.com	10000	2023-09-15	2000

View:

CustomerLoanDetails

This view shows customer details along with loan information.

```
CREATE VIEW CustomerLoanDetails AS
SELECT
    c.customer_id,
    c.name,
    c.email,
    l.loan_id,
    l.loan_amount,
    l.loan_date
FROM Customers c
JOIN Loans l ON c.customer_id = l.customer_id;
```

Output of `CustomerLoanDetails`

customer_id	name	email	loan_id	loan_amount	loan_date
1	John Doe	john@example.com	101	5000	2023-08-01
2	Jane Smith	jane@example.com	102	10000	2023-09-15

Index

on payment_date in the Payments table

If you frequently query based on the payment_date, such as when finding overdue loans or recent payments, you can create an index on this column.

```
CREATE INDEX idx_payments_payment_date ON Payments (payment_date);
```

This index will speed up queries like:

SELECT *

FROM Payments

WHERE payment_date > '2023-09-01';

Output

Index Name	Table	Column
`idx_customers_customer_id`	Customers	customer_id
`idx_loans_loan_id`	Loans	loan_id
`idx_payments_loan_id`	Payments	loan_id
`idx_payments_payment_date`	Payments	payment_date

Limit

Get the First 2 Customers

you want to retrieve only the first 2 customers from the Customers table:

SELECT * FROM Customers LIMIT 2;

Output

customer_id	name	email
1	John Doe	john@example.com
2	Jane Smith	jane@example.com

Retrieve the Latest 2 Loan Records

To retrieve the most recent 2 loans based on the loan_date, you can order the loans by date in descending order and apply the LIMIT clause.

SELECT *

FROM Loans

ORDER BY loan_date DESC

LIMIT 2;

Output

loan_id	customer_id	loan_amount	loan_date
103	3	15000	2023-09-20
102	2	10000	2023-09-15

This query gives the two most recent loans issued.

STEP 6: NORMALIZATION(Grefith Tool)

Assume we have a table with the following attributes related to loans:

- **LoanID**
- **BorrowerName**
- **BorrowerPhoneNumbers**
- **LoanAmount**
- **LoanDate**
- **RepaymentSchedule**

Issues with the Initial Table

Initially, our table might look like this:

LoanID	BorrowerName	BorrowerPhoneNumbers	LoanAmount	LoanDate	RepaymentSchedule
1	John Doe	123-456-7890, 987-654-3210	$10,000	2023-01-15	Monthly
2	Jane Smith	555-123-4567	$5,000	2023-02-20	Bi-Weekly

Problems

1. **Repeating Groups**: The BorrowerPhoneNumbers attribute contains multiple values, violating the atomicity requirement of 1NF.
2. **Non-Atomic Values**: Phone numbers are not stored as atomic values, making it difficult to query or manipulate them individually.

Converting to 1NF

To convert the table to 1NF, we need to ensure that all attributes contain atomic values. We can achieve this by creating a separate entry for each phone number.

Normalized Table in 1NF

LoanID	BorrowerName	BorrowerPhoneNumber	LoanAmount	LoanDate	RepaymentSchedule
1	John Doe	123-456-7890	$10,000	2023-01-15	Monthly
1	John Doe	987-654-3210	$10,000	2023-01-15	Monthly
2	Jane Smith	555-123-4567	$5,000	2023-02-20	Bi-Weekly

Summary of Changes

- The BorrowerPhoneNumbers column was split into multiple rows for each unique phone number associated with a borrower.
- This results in redundancy for the LoanID, BorrowerName, LoanAmount, LoanDate, and RepaymentSchedule, but now all fields contain atomic values, which satisfies 1NF.

Next Steps

After achieving 1NF, you can proceed to further normalize the database to 2NF and 3NF, where you'll focus on removing partial and transitive dependencies to ensure a more efficient and structured database design. If you need guidance on those steps, feel free to ask!

<u>Step 1: Unnormalized Table</u>

Let's start with a simple, unnormalized table representing loans:

LoanID	BorrowerName	BorrowerPhoneNumbers	LoanAmount	LoanDate	RepaymentSchedule
1	John Doe	123-456-7890, 987-654-3210	$10,000	2023-01-15	Monthly
2	Jane Smith	555-123-4567	$5,000	2023-02-20	Bi-Weekly

Issues:

- **Repeating Groups**: Multiple phone numbers in a single field.
- **Non-Atomic Values**: The phone numbers are not atomic (individual).

<u>Step 2: First Normal Form (1NF)</u>

To convert to 1NF, we ensure that all fields contain atomic values:

LoanID	BorrowerName	BorrowerPhoneNumber	LoanAmount	LoanDate	RepaymentSchedule
1	John Doe	123-456-7890	$10,000	2023-01-15	Monthly
1	John Doe	987-654-3210	$10,000	2023-01-15	Monthly
2	Jane Smith	555-123-4567	$5,000	2023-02-20	Bi-Weekly

Step 3: Second Normal Form (2NF)

To achieve 2NF, we need to remove partial dependencies. In this table, BorrowerName, LoanAmount, LoanDate, and RepaymentSchedule depend only on LoanID, while BorrowerPhoneNumber does not depend solely on LoanID.

We can split the data into two tables:

1. **Loans Table**

LoanID	BorrowerName	LoanAmount	LoanDate	RepaymentSchedule
1	John Doe	$10,000	2023-01-15	Monthly
2	Jane Smith	$5,000	2023-02-20	Bi-Weekly

2. **BorrowerPhoneNumbers Table**

LoanID	BorrowerPhoneNumber
1	123-456-7890
1	987-654-3210
2	555-123-4567

Step 4: Third Normal Form (3NF)

To achieve 3NF, we need to remove transitive dependencies. In the **Loans Table**, if we had additional information like the borrower's address, it could create transitive dependencies if that data was also linked to BorrowerName.

For simplicity, assume we want to further normalize by adding a **Borrowers** table:

1. **Loans Table** (remains unchanged):

LoanID	BorrowerID	LoanAmount	LoanDate	RepaymentSchedule
1	1	$10,000	2023-01-15	Monthly

LoanID	BorrowerID	LoanAmount	LoanDate	RepaymentSchedule
2	2	$5,000	2023-02-20	Bi-Weekly

2. **Borrowers Table**

BorrowerID	BorrowerName	Address
1	John Doe	123 Elm St
2	Jane Smith	456 Oak Ave

3. **BorrowerPhoneNumbers Table** (remains unchanged):

LoanID	BorrowerPhoneNumber
1	123-456-7890
1	987-654-3210
2	555-123-4567

Summary

By following these steps, we've successfully normalized our loan database through:

- **1NF**: Ensured atomicity of fields.
- **2NF**: Eliminated partial dependencies.
- **3NF**: Removed transitive dependencies.

STEP 7: Applying the advanced data base for the traditional model

Database structure

1. Borrowers Collection

```
{
  "_id":
  ObjectId("..."),
  "name": "John Doe",
  "email": "johndoe@example.com",
  "phone": "123-456-7890",
  "address": {
    "street": "123 Elm St",
```

```
      "city":
      "Springfield",
      "state": "IL",
      "zip": "62701"
    },
    "date_joined": ISODate("2024-01-15"),
    "credit_score": 720
}
```

2. Loans Collection

```
{
    "_id": ObjectId("..."),
    "borrower_id": ObjectId("..."), // Reference to Borrower
    "amount": 15000,
    "interest_rate": 5.0,
    "term_months": 36,
    "status": "active",
    "start_date": ISODate("2024-01-15"),
    "collateral": [
      {
        "type": "car",
        "value": 20000
      }
    ]
}
```

3. Payments Collection

```
{
    "_id": ObjectId("..."),
    "loan_id": ObjectId("..."),  // Reference to Loan
    "amount": 500,
    "payment_date": ISODate("2024-02-01"),
```

```
  "status": "completed"
}
```

4. Lenders Collection

```
{
  "_id": ObjectId("..."),
  "name": "XYZ Finance
  Co.","contact_info": {
    "email": "contact@xyzfinance.com",
    "phone": "987-654-3210"
  }
}
```

2. CRUD Operations

Here’s how you can perform CRUD operations on this loan management system.

Create

OperationsInsert

a Borrower:

```
db.Borrowers.insertOne({ nam
  e: "John Doe",
  email: "johndoe@example.com",
  phone: "123-456-7890",
  address: {
    street: "123 Elm St",
    city: "Springfield",
    state: "IL",
    zip: "62701"
  },
  date_joined: new Date("2024-01-15"),
  credit_score: 720
})
;
```

Insert a Loan:

```
const borrowerId = ObjectId("..."); // Replace with the actual ObjectId of John Doe
db.Loans.insertOne({
  borrower_id: borrowerId,
  amount: 15000,
  interest_rate: 5.0,
  term_months: 36,
  status: "active",
  start_date: new Date("2024-01-15"),
  collateral: [
    {
      type: "car",
      value: 20000
    }
  ]
});
```

Insert a Payment:

```
const loanId = ObjectId("..."); // Replace with the actual ObjectId of the loan
db.Payments.insertOne({
  loan_id: loanId,
  amount: 500,
  payment_date: new Date("2024-02-01"),
  status: "completed"
});
```

Read Operations

Find a Borrower by Email:

```
const borrower = db.Borrowers.findOne({ email: "johndoe@example.com" });
```

Get All Loans for a Borrower:

```
const loans = db.Loans.find({ borrower_id: borrowerId }).toArray();
```

Get Payment History for a Loan:

```
const payments = db.Payments.find({ loan_id: loanId }).toArray();
```

Update Operations

Update Borrower Information:

```
db.Borrowers.updateOne(
  { _id: borrowerId },
  { $set: { phone: "987-654-3210", credit_score: 740 } }
);
```

Update Loan Status:

```
db.Loans.updateOne(
  { _id: loanId },
  { $set: { status: "closed" } }
);
```

Delete

OperationsDelete

a Payment:

```
db.Payments.deleteOne({ _id: ObjectId("...") }); // Replace with actual ObjectId
```

Delete a Loan:

```
db.Loans.deleteOne({ _id: loanId });
```

Delete a Borrower:

```
db.Borrowers.deleteOne({ _id: borrowerId });
```

3. Sample Outputs

Here are some example outputs you might expect after running the CRUD operations.

1. Output of Find Borrower by Email:

```
{
  "_id":
  ObjectId("..."),
  "name": "John Doe",
  "email": "johndoe@example.com",
  "phone": "123-456-7890",
  "address": {
    "street": "123 Elm St",
```

"city":
"Springfield",
"state": "IL",
"zip": "62701"
},
"date_joined": ISODate("2024-01-15"),
"credit_score": 720
}

2. Output of All Loans for a Borrower:

```
[
  {
    "_id": ObjectId("..."),
    "borrower_id": ObjectId("..."),
    "amount": 15000,
    "interest_rate": 5.0,
    "term_months": 36,
    "status": "active",
    "start_date": ISODate("2024-01-15"),
    "collateral": [
      {
        "type": "car",
        "value": 20000
      }
    ]
  }
]
```

3. Output of Payment History for a Loan:

```
[
  {
    "_id": ObjectId("..."),
```

```
        "loan_id":
        ObjectId("..."),"amount":
        500,
        "payment_date": ISODate("2024-02-01"),
        "status": "completed"
    }
]
```

Result :

Thus the given use case has been implemented using ER model, sql commands ,nested queries,aggregate function ,and finally normalization is done .convertion of traditional database to Advance Database done using mongoldb database.Thus implementation of loan database management system is implemented successfully with given constraints.