FPT UNIVERSITY
INFORMATION SYSTEMS MAJOR
FACULTY OF INFORMATION TECHNOLOGY

FPT UNIVERSITY

ASSIGNMENT REPORT

DATABASE DESIGN

SUPERVISOR	**:** Mrs. NGUYỄN THỊ THU THẢO
GROUP	**:** 7
SUBJECT	**:** DB1202
CLASS	**:** SE1913

HO CHI MINH CITY 11/2025

ASSIGNMENT

Project Title: Banking System

Course: DBI202 | Instructor: Ms. Nguyễn Thị Thu Thảo

ClassID: SE1913
Group ID: 7

Group Members:

Student ID	Full Name	Group	Group Mark	Contribution (%)	Mark	Note
SE203334	Hà Nguyễn Tiến Đạt	7		100		
SE190596	Trần Hữu Việt	7		100		
SE193659	Mai Thành Được	7		100		
SS190849	Lê Nguyên Ngọc	7		100		

Contents

I. INTRODUCTION

1. Introduction

The objective of this project is to develop a comprehensive database system for a banking organization. The system is designed to manage daily operations, including customer records, accounts, loans, transactions, and employees. The final system ensures security, reliability, and support for regulatory compliance, providing a foundation for effective customer service and financial management.

2. Project Objectives

- Manage customers, accounts, and transactions (deposit, withdrawal, transfer).
- Support branch and employee management.
- Manage loans and generate reports.
- Provide secure login, multi-factor authentication, and audit logging.

3. Overview of data models

We analyzed four primary data models:

- **Structured (RDBMS):** Ideal for core financial data requiring high consistency and ACID compliance.
- **Semi-Structured (NoSQL/JSON):** Best for flexible data like varied customer profiles.
- **Unstructured (Blob Storage):** Suited for storing large files like scanned KYC documents.
- **Graph (Graph DB):** Excellent for analyzing complex relationships, such as in fraud detection.

4. Proposed Data Model / Design

After evaluating the project's core requirements, we propose a **hybrid design** with the **Structured (RDBMS) model as its foundation**.

- **Core System (RDBMS):** The RDBMS model was chosen for managing all critical financial entities, which our team **decomposed** into six core components: **customers, accounts, branches, employees, loans, and transactions**. This approach guarantees the high consistency, reliability, and ACID compliance essential for financial operations.
- **Supplemental Models:** While the RDBMS is central, the system is designed to integrate with other models for specific functions:
    - **Unstructured (Blob Storage)** is recommended for storing static assets like scanned KYC (Know Your Customer) documents.
    - **Graph (Graph DB)** is proposed for future implementation of advanced features, such as real-time fraud detection, by analyzing complex transaction relationships.

This report will focus primarily on the design and implementation of the core **RDBMS** component.

II. HIGH-LEVEL DESIGN

1. Scenario

The banking system is based on the following business rules:

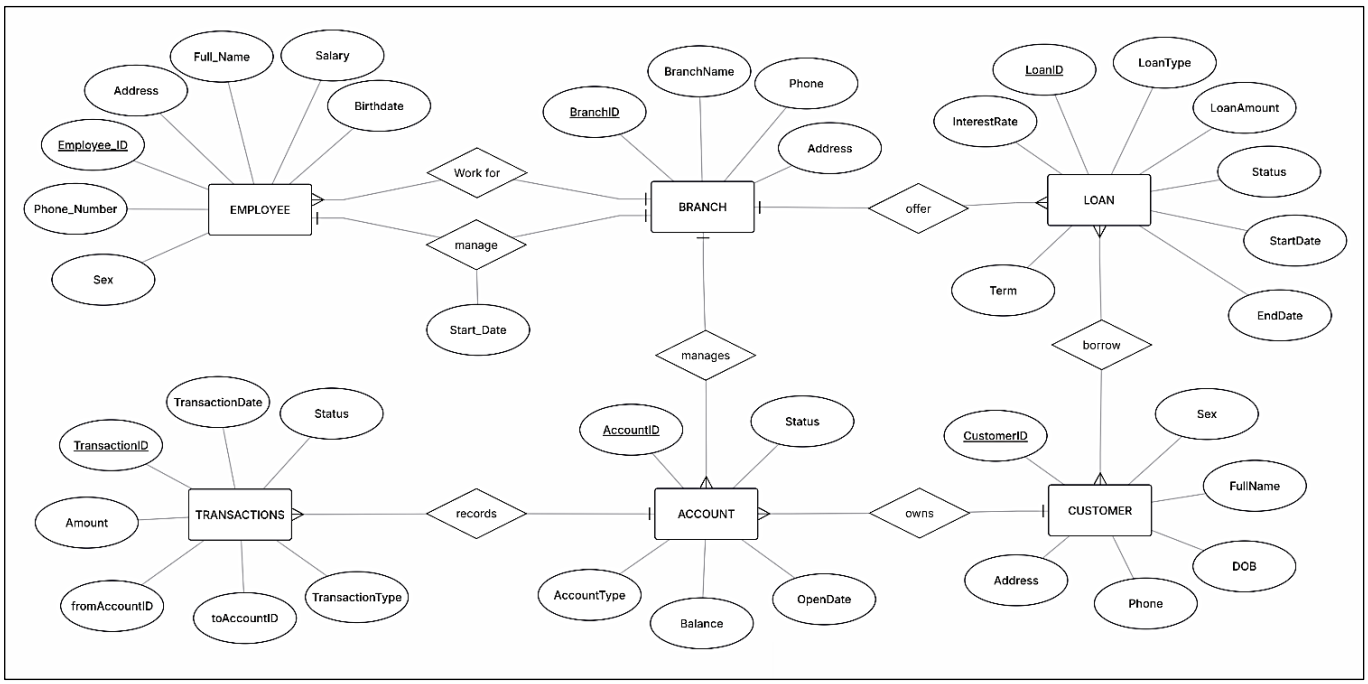
- A **branch** can serve multiple **customers**.
- **Branches** are identified by a **Branch ID**, **branch name**, **phone number**, and **address**.
- **Customers** are identified by a **Customer ID**, **full name**, **date of birth (DOB)**, **address**, **phone number**, and **sex**.
- A **customer** can have one or more **accounts**.
- **Accounts** are identified by an **Account ID**, **account type**, **balance**, **status**, **open date**, the associated **Branch ID**, and the associated **Customer ID**.
- A **customer** can avail of one or more **loans**. A loan can also be associated with multiple customers.
- **Loans** are identified by a **Loan ID**, **loan type**, **loan amount**, **start date**, **status**, the associated **Branch ID**.
- **Employees** are associated with **branches** in two ways:
    - An employee **works for** one specific branch.
    - An employee **manages** one specific branch.
- All account activities (**deposit**, **withdrawal**, and **transfer**) are recorded in a single **TRANSACTIONS** table.

2. Identify entities, relationships and their properties

The core entities identified for the RDBMS portion of the system are:

- EMPLOYEE
- BRANCH
- CUSTOMER
- ACCOUNT
- LOAN
- TRANSACTIONS
- BRANCH_MANAGEMENT *(decomposed from the 1:1 "manages" rule)*
- CUSTOMER_LOAN *(decomposed from the N:M "borrow" rule)*

3. Create the Conceptual Model (ERD)



The conceptual model was developed to identify the core components of the banking system. This ERD clearly identifies the system's entities, their attributes, and the relationships between them, including cardinalities.

ERD Description:

Entities: EMPLOYEE, BRANCH, CUSTOMER, ACCOUNT, LOAN, TRANSACTIONS.

Attributes:

- **EMPLOYEE**: Employee_ID, Full_Name, Address, Phone_Number, Sex, Birthdate, Salary
- **BRANCH**: BranchID, BranchName, Phone, Address
- **CUSTOMER**: CustomerID, FullName, DOB, Address, Phone, Sex
- **ACCOUNT**: AccountID, AccountType, Balance, Status, OpenDate
- **LOAN**: LoanID, LoanType, LoanAmount, StartDate, EndDate, Term, Status, InterestRate
- **TRANSACTIONS**: TransactionID, TransactionDate, Status, TransactionType, Amount, fromAccountID, toAccountID

Relationships & Cardinalities:

- An EMPLOYEE *work for* one BRANCH (N:1).
- An EMPLOYEE *manages* one BRANCH (1:1), resolved via the BRANCH_MANAGEMENT table.
- A BRANCH *manages* many ACCOUNTS (1:N).
- A BRANCH *offers* many LOANS (1:N).
- A CUSTOMER *owns* many ACCOUNTS (1:N).
- Many CUSTOMER *borrow* many LOANS (N:M), resolved via the CUSTOMER_LOAN linking table.
- An ACCOUNT *records* many TRANSACTIONS (1:N), implemented via fromAccountID and toAccountID to handle all transaction types.

4. Develop the Logical Model (RD schems)

The ERD was converted into a relational schema (Logical Diagram). This process involved defining primary and foreign keys and implementing the linking tables identified during decomposition.

Relational Schema:

BRANCH (BranchID, BranchName, Phone, Address)

CUSTOMER (CustomerID, FullName, DOB, Address, Phone, Sex)

EMPLOYEE (EmployeeID, Full_Name, Address, Phone_Number, Sex, Birthdate, Salary, BranchID)

- BranchID (FK) → BRANCH.BranchID

BRANCH_MANAGEMENT (BranchID, ManagerID)

- BranchID (FK) → BRANCH.BranchID
- ManagerID (FK, Unique) → EMPLOYEE.Employee_ID

ACCOUNT (AccountID, AccountType, Balance, Status, OpenDate, CustomerID, BranchID)

- CustomerID (FK) → CUSTOMER.CustomerID
- BranchID (FK) → BRANCH.BranchID

LOAN (LoanID, LoanType, LoanAmount, StartDate, EndDate, Term, Status, InterestRate, BranchID)

- BranchID (FK) → BRANCH.BranchID

CUSTOMER_LOAN (CustomerID, LoanID)

- CustomerID (FK) → CUSTOMER.CustomerID
- LoanID (FK) → LOAN.LoanID

TRANSACTIONS (TransactionID, TransactionDate, Status, TransactionType, Amount, fromAccountID, toAccountID)

- FromAccountID (FK) → ACCOUNT.AccountID
- ToAccountID (FK) → ACCOUNT.AccountID

III. LOGICAL DESIGN

1. Determine the functional dependencies

- **Branch**
  BranchID → BranchName, Phone, Address
  *(The **branch ID** uniquely determines the branch name, and address)*
- **Customer**
  CustomerID → FullName, DOB, Address, Phone, Sex
  *(**CustomerID** uniquely determines the customer's name, date of birth, address, phone, and gender)*
- **Employee**
  Employee_ID → Full_Name, Address, Phone_Number, Sex, Birthdate, Salary, BranchID
  *(**EmployeeID** uniquely determines all personal and employment details of the employee)*
- **Account**
  AccountID → AccountType, Balance, Status, OpenDate, CustomerID, BranchID
  *(**AccountID** uniquely determines the account type, balance, status, open date, customer, and branch)*
- **Loan**
  LoanID → LoanType, LoanAmount, StartDate, EndDate, Term, Status, InterestRate, BranchID
  *(**LoanID** uniquely determines the loan type, amount, start date, end date, term, status, interest rate and the associated branch)*
- **Transactions**
  TransactionID → TransactionDate, Status, TransactionType, Amount, fromAccountID, toAccountID
  *(**TransactionID** uniquely determines all details of the transaction, including its type, status, amount, and the associated source (from) and destination (to) accounts)*
- **Branch_Management**

  BranchID → ManagerID

  ManagerID → BranchID

  (Due to the UNIQUE constraint on ManagerID, ensuring a 1:1 relationship)
- **Customer_Loan:**

  (No non-key attributes. The entire relation is a composite key).

Pattern Recognition Process

To identify functional dependencies (FDs), our team analyzed the relationships among attributes within each entity in the relational schema. We identified attributes that are uniquely determined by others (for example, a unique identifier such as **Employee_ID**,

BranchID, or **CustomerID**). By comparing sample data and observing which attributes consistently appeared together, we recognized patterns such as:

- **BranchID** uniquely determined the branch name, phone number, and address.
- **CustomerID** uniquely determined the customer's full name, date of birth (DOB), address, phone number, and sex.
- **Employee_ID** consistently determined an employee's full name, address, phone number, sex, birthdate, salary, and BranchID.
- **AccountID** uniquely determined the account type, balance, status, open date, BranchID, and CustomerID.
- **LoanID** uniquely determined the loan type, loan amount, start date, end date, term, status, interest rate, and BranchID.
- **TransactionID** uniquely determined all details for a single event (date, type, amount), including both the source (fromAccountID) and destination (toAccountID). This unified pattern was recognized to cover deposits, withdrawals, and transfers.
- The **1:1 "manages" rule** was recognized as a pattern requiring its own relation (Branch_Management) to enforce the unique mapping from a manager to a branch.
- The **N:M "borrow" rule** was recognized as a pattern with no simple determinant, requiring a composite key linking table (Customer_Loan).

These patterns helped us define the FDs and ensure that the database model reflects real-world constraints accurately.

Algorithmic Reasoning Steps

To systematically derive the FDs and keys, our group followed a step-by-step reasoning process:

1. Identify Determinants: We started by identifying unique identifiers (determinants) for each entity, such as:

   Employee_ID for the **Employee** entity.

   BranchID for the **Branch** entity.

   CustomerID for the **Customer** entity.

   AccountID for the **Account** entity.

   LoanID for the **Loan** entity.

   TransactionID for the **Transaction** entity.

   BranchID and **ManagerID** for the Branch_Management entity.

   {**CustomerID**, **LoanID**} for the **Customer_Loan** entity.

2. Find Dependent Attributes: Next, we identified the attributes that depend on these determinants (e.g., **BranchID → BranchName, Phone, Address** in the **Branch** table)
3. Compute Attribute Closure: We performed closure checks to ensure there was no redundancy. This helped confirm that each determinant could uniquely determine all other attributes in the entity. For instance:

   Employee_ID → {Full_Name, Address, Phone_Number, Sex, Birthdate, Salary, BranchID}

   BranchID → {BranchName, Phone, Address}
4. Determine Candidate Keys: Attributes with full closures were selected as candidate keys.
5. Choose Primary Keys: From the candidate keys, we selected the most suitable primary key based on simplicity and clarity.

This logical process made it easier to derive correct dependencies and ensure normalization consistency.

2. Find the keys

Entity	Candidate Key(s)	Primary Key
BRANCH	{BranchID}	BranchID
CUSTOMER	{CustomerID}	CustomerID
EMPLOYEE	{EmployeeID}	EmployeeID
ACCOUNT	{AccountID}	AccountID
LOAN	{LoanID}	LoanID
TRANSACTIONS	{TransactionID}	TransactionID
BRANCH_MANAGEMENT	{BranchID}, {ManagerID}	BranchID
CUSTOMER_LOAN	{CustomerID, LoanID}	{CustomerID, LoanID}

IV. NORMALIZATION

1. Anomaly Detection and Normalization

The initial database design may encounter several anomalies such as:

Redundancy anomaly:

Employee information may be repeated for each branch they work in, leading to duplicated data.

Update anomaly:

If an employee's branch address changes, all related records must be updated manually, increasing the risk of inconsistency.

Insertion anomaly:

It is impossible to add a new branch without assigning at least one employee because of the foreign key dependency.

Deletion anomaly:

Deleting a branch that an employee belongs to may cause the loss of related employee information.

2. Identify Normal Forms

We analyze each relation based on the **Relational Schema** to determine its normal form.

BRANCH (BranchID, BranchName, Phone, Address)

FD: BranchID → BranchName, Phone, Address
→ All attributes depend on a single primary key.
→ **In BCNF**

CUSTOMER (CustomerID, FullName, DOB, Address, Phone, Sex)

FD: CustomerID → FullName, DOB, Address, Phone, Sex
→ All attributes depend on the primary key.
→ **In BCNF**

EMPLOYEE (EmployeeID, Full_Name, Address, Phone_Number, Sex, Birthdate, Salary, BranchID)

FDs: EmployeeID → Full_Name, Address, Phone_Number, Sex, Birthdate, Salary, BranchID
→ All non-key attributes depend fully on the primary key.
→ **In BCNF**

ACCOUNT (AccountID, AccountType, Balance, Status, OpenDate, CustomerID, BranchID)

FD: AccountID → AccountType, Balance, Status, OpenDate, CustomerID, BranchID
→ The primary key determines all other attributes.
→ **In BCNF**

LOAN (LoanID, LoanType, LoanAmount, StartDate, EndDate, Term, Status, InterestRate, BranchID)

FD: LoanID → LoanType, LoanAmount, StartDate, EndDate, Term, Status, InterestRate, BranchID
→ Fully functionally dependent on the primary key.
→ **In BCNF**

TRANSACTIONS (TransactionID, TransactionDate, Status, TransactionType, Amount, fromAccountID, toAccountID)

FD: TransactionID → TransactionDate, Status, TransactionType, Amount, fromAccountID, toAccountID
→ All non-key attributes depend only on the primary key.
→ **In BCNF**

BRANCH_MANAGEMENT (BranchID, ManagerID)

FDs: BranchID → ManagerID, ManagerID → BranchID.

Both determinants are candidate keys.

→ **In BCNF**

CUSTOMER_LOAN (CustomerID, LoanID)

This relation has no non-key attributes.

→ **In BCNF**

3. Perform Decomposition

The normalization of our database to BCNF (as identified in the previous step) was achieved precisely through the **Decomposition** process. The initial conceptual design (in Part II) would have suffered from anomalies if not for these crucial decompositions:

- **N:M Decomposition (Loan-Customer):** The conceptual "borrow" (N:M) relationship was decomposed into the Customer_Loan relation . This is the standard algorithmic step to resolve a many-to-many relationship, preventing data redundancy and update anomalies.
- **1:1 Decomposition (Employee-Branch):** The "manages" (1:1) relationship was decomposed into the Branch_Management relation. This resolves potential update anomalies, avoids NULL values (which would occur if ManagedBranchID were stored in Employee), and ensures the 1:1 constraint is strictly enforced.
- **Logic Decomposition (Transactions):** The business logic for transactions was unified. Instead of the old two-table model (Transactions and Transfer), we recognized the pattern could be decomposed into a single Transactions table using two foreign keys (fromAccountID, toAccountID).

This decomposition process ensures the final schema is in BCNF, maintains the Lossless Join property, and preserves dependencies.

4. Final Normalized Schema

All relations satisfy **BCNF**, ensuring:

- No redundancy or anomaly issues
- Data integrity and consistency
- Efficient updates and scalability

1. Branch

- `BranchID` (Primary Key)
- `BranchName`
- `Phone`
- `Address`

2. Customer

- `CustomerID` (Primary Key)
- `FullName`
- `DOB`
- `Address`
- `Phone`
- `Sex`

3. Employee

- `EmployeeID` (Primary Key)
- `Full_Name`
- `Address`
- `Phone_Number`
- `Sex`
- `Birthdate`
- `Salary`
- `BranchID` (Foreign Key → `BRANCH.BranchID`)

4. Branch_Management

- `BranchID` (Primary Key, Foreign Key → `BRANCH.BranchID`)
- `ManagerID` (Unique, Foreign Key → `EMPLOYEE.Employee_ID`)

5. Account

- `AccountID` (Primary Key)
- `AccountType`
- `Balance`
- `Status`
- `OpenDate`
- `CustomerID` (Foreign Key → `Customer.CustomerID`)
- `BranchID` (Foreign Key → `Branch.BranchID`)

6. Loan

- `LoanID` (Primary Key)
- `LoanType`
- `LoanAmount`
- `StartDate`
- `EndDate`
- `Term`
- `Status`
- `InterestRate`
- `BranchID` (Foreign Key → `Branch.BranchID`)

7. Customer_Loan

- `CustomerID` (Primary Key, Foreign Key → `CUSTOMER.CustomerID`)
- `LoanID` (Primary Key, Foreign Key → `LOAN.LoanID`)

8. Transactions

- `TransactionID` (Primary Key)
- `TransactionDate`
- `Status`
- `TransactionType`
- `Amount`
- `fromAccountID` (Foreign Key → `ACCOUNT.AccountID`)
- `toAccountID` (Foreign Key → `ACCOUNT.AccountID`)

V. PHYSICAL DESIGN

1. Design the Physical Model

The logical schema was mapped to a physical model, defining specific data types and properties for implementation in a SQL database.

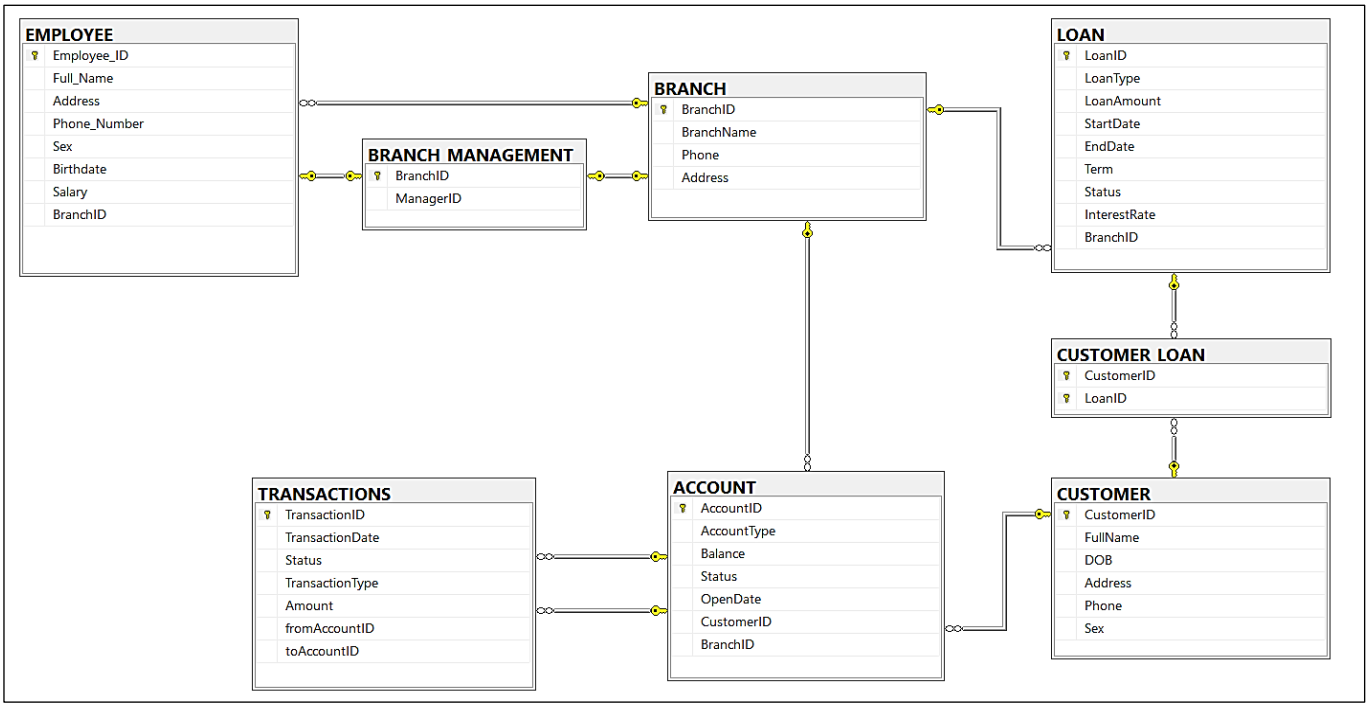


Table Definitions:

```
CREATE TABLE BRANCH (
    BranchID INT PRIMARY KEY,
    BranchName VARCHAR(255) NOT NULL UNIQUE,
    Phone VARCHAR(20),
    Address VARCHAR(255)
)
CREATE TABLE CUSTOMER (
    CustomerID INT PRIMARY KEY,
```

```
    FullName VARCHAR(255) NOT NULL,
    DOB DATE,
    Address VARCHAR(255),
    Phone VARCHAR(20) NOT NULL UNIQUE,
    Sex VARCHAR(10) CHECK (Sex IN ('Male', 'Female')) NOT NULL
)

CREATE TABLE EMPLOYEE (
    Employee_ID INT PRIMARY KEY,
    Full_Name VARCHAR(255) NOT NULL,
    Address VARCHAR(255),
    Phone_Number VARCHAR(20) NOT NULL UNIQUE,
    Sex VARCHAR(10) CHECK (Sex IN ('Male', 'Female')),
    Birthdate DATE,
    Salary DECIMAL(15,2) NOT NULL CHECK (Salary > 0),

    BranchID INT NOT NULL,
    FOREIGN KEY (BranchID) REFERENCES BRANCH(BranchID)
)

CREATE TABLE BRANCH_MANAGEMENT (
    BranchID INT PRIMARY KEY,
    ManagerID INT NOT NULL UNIQUE,

    FOREIGN KEY (BranchID) REFERENCES BRANCH(BranchID),
    FOREIGN KEY (ManagerID) REFERENCES EMPLOYEE(Employee_ID)
)

CREATE TABLE ACCOUNT (
    AccountID INT PRIMARY KEY,
    AccountType VARCHAR(100) NOT NULL,
    Balance DECIMAL(12,2) NOT NULL CHECK (Balance >= 0) DEFAULT 0,
    Status VARCHAR(50) NOT NULL CHECK (Status IN ('Active', 'Closed', 'Frozen')),
    OpenDate DATE NOT NULL,

    CustomerID INT NOT NULL,
    BranchID INT NOT NULL,

    FOREIGN KEY (CustomerID) REFERENCES CUSTOMER(CustomerID),
    FOREIGN KEY (BranchID) REFERENCES BRANCH(BranchID)
)

CREATE TABLE LOAN (
    LoanID INT PRIMARY KEY,
    LoanType VARCHAR(100) NOT NULL,
    LoanAmount DECIMAL(12,2) NOT NULL CHECK (LoanAmount > 0),
```

```
    StartDate DATE NOT NULL,
    EndDate DATE,
    Term VARCHAR(50),
    Status VARCHAR(50) NOT NULL CHECK (Status IN ('Pending', 'Approved', 'Paid Off',
'Default')),
    InterestRate DECIMAL(5,2) NOT NULL CHECK (InterestRate >= 0),
    BranchID INT NOT NULL,
    FOREIGN KEY (BranchID) REFERENCES BRANCH(BranchID)
 )

 CREATE TABLE CUSTOMER_LOAN (
    CustomerID INT NOT NULL,
    LoanID INT NOT NULL,

    PRIMARY KEY (CustomerID, LoanID),
    FOREIGN KEY (CustomerID) REFERENCES CUSTOMER(CustomerID),
    FOREIGN KEY (LoanID) REFERENCES LOAN(LoanID)
 )

 CREATE TABLE TRANSACTIONS (
    TransactionID INT PRIMARY KEY,
    TransactionDate DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP,

    Status VARCHAR(50) NOT NULL,
    TransactionType VARCHAR(50) NOT NULL,
    Amount DECIMAL(12,2) NOT NULL CHECK (Amount > 0),

    fromAccountID INT,
    toAccountID INT,

    FOREIGN KEY (fromAccountID) REFERENCES ACCOUNT(AccountID),
    FOREIGN KEY (toAccountID) REFERENCES ACCOUNT(AccountID),
    CHECK (fromAccountID IS NOT NULL OR toAccountID IS NOT NULL)
 )
```

2. Basic retrieval to advanced operations

SQL Queries

Basic:

- **Description:** Retrieve all customers in the system.

```
SELECT * FROM CUSTOMER
```

Results Messages

	CustomerID	FullName	DOB	Address	Phone	Sex
1	1	Nguyen Van An	1988-03-15	123 Le Loi, Ha Noi	0905123456	Male
2	2	Tran Thi Bich	1992-07-09	45 Nguyen Trai, Da Nang	0905789123	Female
3	3	Le Hoang Nam	1985-12-22	78 Hai Ba Trung, HCM	0912345678	Male
4	4	Pham Minh Chau	1998-04-01	23 Nguyen Van Cu, Hue	0978123987	Female
5	5	Do Quang Huy	1990-09-30	89 Phan Dinh Phung, Ha Noi	0904556677	Male

- **Description:** Find all employees who work in BranchID = 1 and have a salary over 17,000,000.

```
SELECT Employee_ID, Full_Name, Salary, BranchID FROM EMPLOYEE
WHERE BranchID = 1 AND Salary > 17000000
```

Results Messages

	Employee_ID	Full_Name	Salary	BranchID
1	1	Pham Quang Tuan	18000000.00	1

- **Description:** List transactions from October 12th to October 21st, 2024, sorted by date.

```
SELECT * FROM TRANSACTIONS
WHERE TransactionDate >= '2024-10-12' AND TransactionDate <= '2024-10-21'
ORDER BY TransactionDate
```

Results Messages

	TransactionID	TransactionDate	Status	TransactionType	Amount	fromAccountID	toAccountID
1	4	2024-10-12 16:00:00.000	Success	Withdrawal	700000.00	2	NULL
2	5	2024-10-15 11:20:00.000	Success	Deposit	3000000.00	NULL	5
3	6	2024-10-20 08:00:00.000	Success	Transfer	1000000.00	2	1

- **Description:** Count the total number of customers.

```
SELECT COUNT(CustomerID) AS TotalCustomers FROM CUSTOMER
```

Results Messages

	TotalCustomers
1	5

Intermediate:

- **Description**: Retrieve the customer's FullName, AccountID and the Balance of all accounts they own.

```
SELECT c.FullName, a.AccountID, a.Balance
FROM CUSTOMER c
JOIN ACCOUNT a ON c.CustomerID = a.CustomerID
```

Results Messages

	FullName	AccountID	Balance
1	Nguyen Van An	1	5000000.00
2	Tran Thi Bich	2	12000000.00
3	Le Hoang Nam	3	3500000.00
4	Pham Minh Chau	4	7000000.00
5	Do Quang Huy	5	25000000.00

- **Description**: Show all branches and the count of employees in each (even if a branch has zero employees).

```
SELECT b.BranchName, COUNT(e.Employee_ID) AS EmployeeCount
FROM BRANCH b
LEFT JOIN EMPLOYEE e ON b.BranchID = e.BranchID
GROUP BY b.BranchName
```

Results Messages

	BranchName	EmployeeCount
1	Ha Noi Branch	2
2	Da Nang Branch	1
3	HCM Branch	2

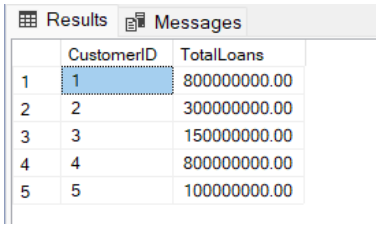
- **Description:** Count the number of loans per BranchID.

```
SELECT BranchID, COUNT(LoanID) AS LoanCount
FROM LOAN
GROUP BY BranchID
```

	BranchID	LoanCount
1	1	2
2	2	2
3	3	1

- **Description:** Show customers who have a total loan amount greater than 40,000,000.

```
SELECT cl.CustomerID, SUM(l.LoanAmount) AS TotalLoans
FROM LOAN l
JOIN CUSTOMER_LOAN cl ON l.LoanID = cl.LoanID
GROUP BY cl.CustomerID
HAVING SUM(l.LoanAmount) > 40000000;
```

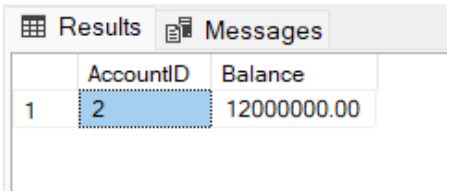


Results Messages

	CustomerID	TotalLoans
1	1	800000000.00
2	2	300000000.00
3	3	150000000.00
4	4	800000000.00
5	5	100000000.00

- **Description**: Find accounts that have sent or withdrawn transactions greater than 1,000,000.

```
SELECT AccountID, Balance FROM ACCOUNT
WHERE AccountID IN (
   SELECT fromAccountID FROM TRANSACTIONS
   WHERE Amount > 1000000 AND fromAccountID IS NOT NULL
);
```

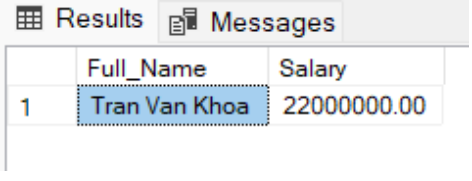


Results Messages

	AccountID	Balance
1	2	12000000.00

- **Description**: Find employees who earn more than the average salary of employees in 'HCM Branch'.

```
SELECT Full_Name, Salary FROM EMPLOYEE
WHERE Salary > (
   SELECT AVG(Salary) FROM EMPLOYEE
   WHERE BranchID = (
            SELECT BranchID FROM BRANCH
            WHERE BranchName = 'HCM Branch')
)
```



Results Messages

	Full_Name	Salary
1	Tran Van Khoa	22000000.00

Advanced:

- **Description**: Find the FullName of customers who have a balance higher than the average balance of all 'Savings' accounts.

```
SELECT FullName FROM CUSTOMER
WHERE CustomerID IN (
   SELECT CustomerID FROM ACCOUNT
   WHERE Balance > (
      SELECT AVG(Balance) FROM ACCOUNT
      WHERE AccountType = 'Savings'
   )
)
```

	FullName
1	Tran Thi Bich
2	Do Quang Huy

- **Description**: Find all customers who have at least one 'Active' account.

```
SELECT c.FullName
FROM CUSTOMER c
WHERE EXISTS (
   SELECT 1 FROM ACCOUNT a
   WHERE a.CustomerID = c.CustomerID AND a.Status = 'Active'
)
```

Results Messages

	FullName
1	Nguyen Van An
2	Tran Thi Bich
3	Do Quang Huy

- **Description**: Find the names of customers who have an 'Active' account that has *received a deposit* over 1,000,000.

```
SELECT FullName FROM CUSTOMER
WHERE CustomerID IN (
  SELECT CustomerID FROM ACCOUNT
  WHERE Status = 'Active' AND AccountID IN (
     SELECT toAccountID FROM TRANSACTIONS
     WHERE Amount > 1000000 AND TransactionType = 'Deposit'
  )
)
```

Results Messages

	FullName
1	Do Quang Huy

- **Description**: Find the employee(s) with the highest salary.

```
SELECT Full_Name, Salary FROM EMPLOYEE
WHERE Salary >= ALL (
   SELECT Salary FROM EMPLOYEE
)
```

Results Messages

	Full_Name	Salary
1	Tran Van Khoa	22000000

- **Description**: Get a list (CustomerID, FullName) of customers who have *both* an ACCOUNT and a LOAN with the bank.

```
SELECT CustomerID, FullName FROM CUSTOMER
WHERE CustomerID IN (SELECT CustomerID FROM ACCOUNT)
INTERSECT
SELECT CustomerID, FullName FROM CUSTOMER
WHERE CustomerID IN (SELECT CustomerID FROM CUSTOMER_LOAN)
```

Results Messages

	CustomerID	FullName
1	1	Nguyen Van An
2	2	Tran Thi Bich
3	3	Le Hoang Nam
4	4	Pham Minh Chau
5	5	Do Quang Huy

Functions

fn_CalculateCustomerAge

- **Description:** Calculates a customer's current age based on their Date of Birth (DOB).

```
CREATE FUNCTION fn_CalculateCustomerAge(@DOB DATE)
RETURNS INT AS
BEGIN
   RETURN DATEDIFF(YEAR, @DOB, GETDATE());
END
```

- Execution:

```
SELECT CustomerID, FullName, dbo.fn_CalculateCustomerAge(DOB) AS Age
FROM CUSTOMER
```

Results Messages

	CustomerID	FullName	Age
1	1	Nguyen Van An	37
2	2	Tran Thi Bich	33
3	3	Le Hoang Nam	40
4	4	Pham Minh Chau	27
5	5	Do Quang Huy	35

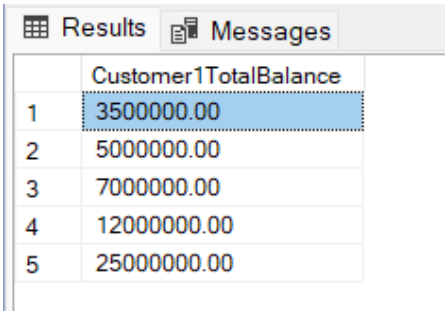
fn_GetCustomerTotalBalance

- **Description:** Calculates the total balance of a specific customer (CustomerID) by summing the balances of all their accounts.

```
CREATE FUNCTION fn_GetCustomerTotalBalance(@CustomerID INT)
RETURNS DECIMAL(18,2)
AS
BEGIN
   DECLARE @TotalBalance DECIMAL(18,2);
   SELECT @TotalBalance = SUM(Balance) FROM ACCOUNT
   WHERE CustomerID = @CustomerID;
   RETURN ISNULL(@TotalBalance, 0);
END
```

- Execution:

```
SELECT DISTINCT dbo.fn_GetCustomerTotalBalance(CustomerID)
AS Customer1TotalBalance
FROM ACCOUNT
```



	Customer1TotalBalance
1	3500000.00
2	5000000.00
3	7000000.00
4	12000000.00
5	25000000.00

Stored Procedures

sp_AddNewCustomer

- **Description:** Inserts a new customer record into the CUSTOMER table.

```
CREATE PROCEDURE sp_AddNewCustomer
   @CustomerID INT, @FullName NVARCHAR(100),
   @DOB DATE, @Address NVARCHAR(255),
   @Phone VARCHAR(20), @Sex VARCHAR(10)
AS
BEGIN
   INSERT INTO CUSTOMER (CustomerID, FullName, DOB, Address, Phone, Sex)
   VALUES (@CustomerID, @FullName, @DOB, @Address, @Phone, @Sex);
END
```

- Execution:

```
EXEC sp_AddNewCustomer 6, N'Ngo Huy Long', '1990-01-01', N'321 ABC Street, Da Lat', '0992645378', 'Male'
```

Results Messages

	CustomerID	FullName	DOB	Address	Phone	Sex
1	1	Nguyen Van An	1988-03-15	123 Le Loi, Ha Noi	0905123456	Male
2	2	Tran Thi Bich	1992-07-09	45 Nguyen Trai, Da Nang	0905789123	Female
3	3	Le Hoang Nam	1985-12-22	78 Hai Ba Trung, HCM	0912345678	Male
4	4	Pham Minh Chau	1998-04-01	23 Nguyen Van Cu, Hue	0978123987	Female
5	5	Do Quang Huy	1990-09-30	89 Phan Dinh Phung, Ha Noi	0904556677	Male
6	6	Ngo Huy Long	1990-01-01	321 ABC Street, Da Lat	0992645378	Male

sp_MakeDeposit

- **Description:** Performs a simple deposit transaction. It updates the account's balance and inserts a record into the TRANSACTIONS table to log the event.

```
CREATE PROCEDURE sp_MakeDeposit
   @AccountID INT,
   @Amount DECIMAL(12, 2)
AS
BEGIN
   IF @Amount <= 0
   BEGIN
      RAISERROR('Deposit amount must be positive.', 16, 1);
      RETURN;
   END

   DECLARE @NextTransactionID INT;
   SELECT @NextTransactionID = ISNULL(MAX(TransactionID), 0) + 1
   FROM TRANSACTIONS;

   UPDATE ACCOUNT SET Balance = Balance + @Amount
   WHERE AccountID = @AccountID;

   INSERT INTO TRANSACTIONS (TransactionID, TransactionType, Status, Amount,
fromAccountID, toAccountID)
   VALUES (@NextTransactionID, 'Deposit', 'Success', @Amount, NULL, @AccountID);

   PRINT 'Deposit successful.';
END;
```

- Execution (Deposit 500,000):

```
EXEC sp_MakeDeposit @AccountID = 1, @Amount = 500000;
```

Results Messages

	AccountID	AccountType	Balance	Status	OpenDate	CustomerID	BranchID
1	1	Savings	5500000.00	Active	2022-01-10	1	1

	TransactionID	TransactionDate	Status	TransactionType	Amount	fromAccountID	toAccountID
1	1	2024-10-01 10:00:00.000	Success	Withdrawal	200000.00	1	NULL
2	2	2024-10-05 14:30:00.000	Success	Withdrawal	1500000.00	2	NULL
3	3	2024-10-10 09:45:00.000	Success	Deposit	500000.00	NULL	3
4	4	2024-10-12 16:00:00.000	Success	Withdrawal	700000.00	2	NULL
5	5	2024-10-15 11:20:00.000	Success	Deposit	3000000.00	NULL	5
6	6	2024-10-20 08:00:00.000	Success	Transfer	1000000.00	2	1
7	7	2025-11-11 15:19:21.430	Success	Deposit	500000.00	NULL	1

Triggers

Prevent branch deletion

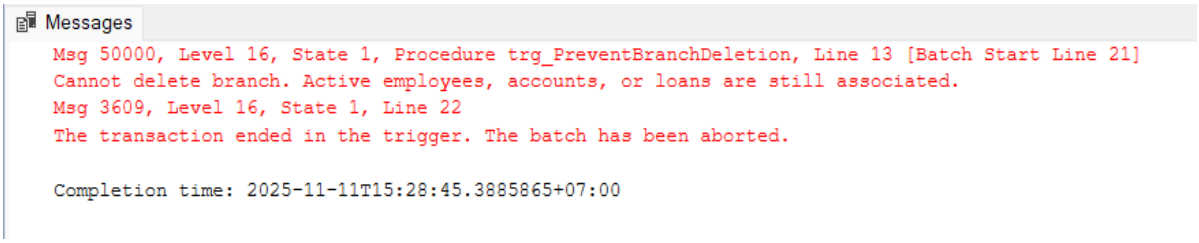
- **Description:** This trigger prevents the deletion of a BRANCH record (parent record) if there are still associated child records (employees, accounts, or loans) linked to it. This helps ensure referential integrity.

```
CREATE TRIGGER trg_PreventBranchDeletion
ON BRANCH
INSTEAD OF DELETE
AS
BEGIN
   DECLARE @BranchID INT;
   SELECT @BranchID = d.BranchID FROM deleted AS d;

   IF EXISTS (SELECT 1 FROM EMPLOYEE WHERE BranchID = @BranchID) OR
      EXISTS (SELECT 1 FROM ACCOUNT WHERE BranchID = @BranchID) OR
      EXISTS (SELECT 1 FROM LOAN WHERE BranchID = @BranchID)
   BEGIN
      RAISERROR('Cannot delete branch. Active employees, accounts, or loans are still
associated.', 16, 1);
      ROLLBACK TRANSACTION;
   END
   ELSE
   BEGIN
      DELETE FROM BRANCH WHERE BranchID = @BranchID;
   END
END;
```

- Execution: The command fails and returns the error: "Cannot delete branch. Active employees, accounts, or loans are still associated."

```
DELETE FROM BRANCH WHERE BranchID = 1
```



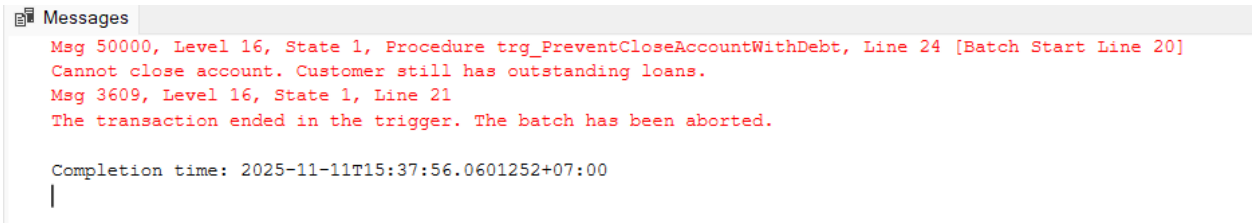
trg_PreventCloseAccountWithDebt

- **Description:** Enforces a critical business rule. This trigger prevents changing an account's **status** to 'Closed' if the **Customer** who owns that account has any outstanding loans (any loan status other than 'Paid Off').

```
CREATE TRIGGER trg_PreventCloseAccountWithDebt
ON ACCOUNT
AFTER UPDATE
AS
BEGIN
  IF EXISTS (SELECT 1 FROM inserted WHERE Status = 'Closed')
  BEGIN
    IF EXISTS (
      SELECT 1 FROM inserted i
      JOIN CUSTOMER_LOAN cl ON i.CustomerID = cl.CustomerID
      JOIN LOAN l ON cl.LoanID = l.LoanID
      WHERE i.Status = 'Closed' AND l.Status <> 'Paid Off'
    )
    BEGIN
      RAISERROR ('Cannot close account. Customer still has outstanding loans.', 16, 1);
      ROLLBACK TRANSACTION;
    END
  END
END;
```

- Execution: This will fail with error: Cannot close account. Customer still has outstanding loans.

```
UPDATE ACCOUNT SET Status = 'Closed' WHERE AccountID = 1;
```



- Execution: The command succeeds (test case for AccountID = 2, whose owner has a 'Paid Off' loan).

```
UPDATE ACCOUNT SET Status = 'Closed' WHERE AccountID = 2;
```

Results Messages

	AccountID	AccountType	Balance	Status	OpenDate	CustomerID	BranchID
1	2	Checking	12500000.00	Closed	2021-05-22	2	2

Create View and Index

View: vw_CustomerAccountDetails

- **Description:** Defines a view to simplify complex queries. This view provides a straightforward way to see customer account details without needing to write the JOINs manually.

```
CREATE VIEW vw_CustomerAccountDetails AS
SELECT
   C.CustomerID, C.FullName, C.Phone,
   A.AccountID, A.AccountType, A.Balance, A.Status AS AccountStatus,
   B.BranchName
FROM CUSTOMER AS C
JOIN ACCOUNT AS A ON C.CustomerID = A.CustomerID
JOIN BRANCH AS B ON A.BranchID = B.BranchID;
```

- Execution:

```
SELECT * FROM vw_CustomerAccountDetails WHERE FullName LIKE N'Nguyen Van A%'
```

Results Messages

	CustomerID	FullName	Phone	AccountID	AccountType	Balance	AccountStatus	BranchName
1	1	Nguyen Van An	0905123456	1	Savings	5005000.00	Active	Ha Noi Branch

Index: idx_Account_CustomerBranch

- **Description:** Creates a composite index on the **CustomerID** and **BranchID** columns in the **ACCOUNT** table. This index is designed to improve the performance of queries that frequently filter or join on both the customer and their branch.

```
CREATE INDEX idx_Account_CustomerBranch
ON ACCOUNT(CustomerID, BranchID);
```

- **Execution:** Test query that benefits from the index.

```
SELECT AccountID, Balance FROM ACCOUNT
WHERE CustomerID = 1 AND BranchID = 1;
```

Results Messages

	AccountID	Balance
1	1	6000000.00

VI. CONCLUSION

1. AI Utilization

AI tools (in study mode) were used as a supportive partner.

Design Phase: AI was prompted to validate our identified functional dependencies and to confirm that our proposed schema was consistent with 3NF normalization principles.

Implementation Phase: AI assisted in debugging and refining SQL code. For example, it helped validate the logic for the unified **Transactions** table, ensuring that the use of fromAccountID and toAccountID could correctly handle deposit, withdrawal, and transfer events, and helped refine the logic for the *sp_MakeDeposit* procedure.

2. Teamwork & Collaboration

1. **Hà Nguyễn Tiến Đạt: Database Design & Core Schema**

   Role: Design and implement the overall database schema.

   Details:

   - Define tables (**Customer**, **Account**, **Employee**, **Branch**, **Loan**, **Transactions**, **Branch management**, **Customer loan**).
   - Set up **primary keys, foreign keys, and constraints**.
   - Ensure **data normalization** and integrity.
   - Provide **ERD diagram** and schema documentation.

2. **Trần Hữu Việt: Customer & Account Management**

   Role: Build customer and account management modules.

   Details:

   - Implement APIs for **add/edit/delete customers**.
   - Implement APIs for **create/update accounts** (savings, current, fixed deposit).
   - Design a **basic user interface** for customer/account operations.
   - Ensure account status (active, closed, frozen) is properly managed.

3. **Mai Thành Được: Transaction Management**

   Role: Develop transaction handling functions.

   Details:

   - Build APIs to handle the **unified transaction logic** (processing different TransactionType values within the single **Transactions** table).
   - Ensure **transaction ACID compliance** and proper error handling.
   - Link transactions to **audit log** for traceability.
   - Generate **transaction receipts** for customer confirmation.

4. **Lê Nguyên Ngọc: Loan, Security & Reporting**

   Role: Handle loan operations, security, and reporting features.

   Details:

   - Implement APIs for **loan applications, repayments, and loan status tracking**.
   - Develop **secure login system** (username/password + optional MFA).
   - Maintain **audit logging** for employee/customer actions.
   - Create **reports** (e.g., transaction history, loan status, customer summaries).