Banking and Finance Management System Report

Contents

Problem Statement

Traditional banking operations often face major challenges such as operational inefficiencies, frequent errors, and limited real-time transparency due to manual or partially automated processes used for managing accounts, transactions, loans, and investments. These outdated methods result in slow customer service and restricted access to financial information. Additionally, banking staff spend excessive time handling repetitive tasks, which leads to delays, reduced productivity, and increased risks related to data security.

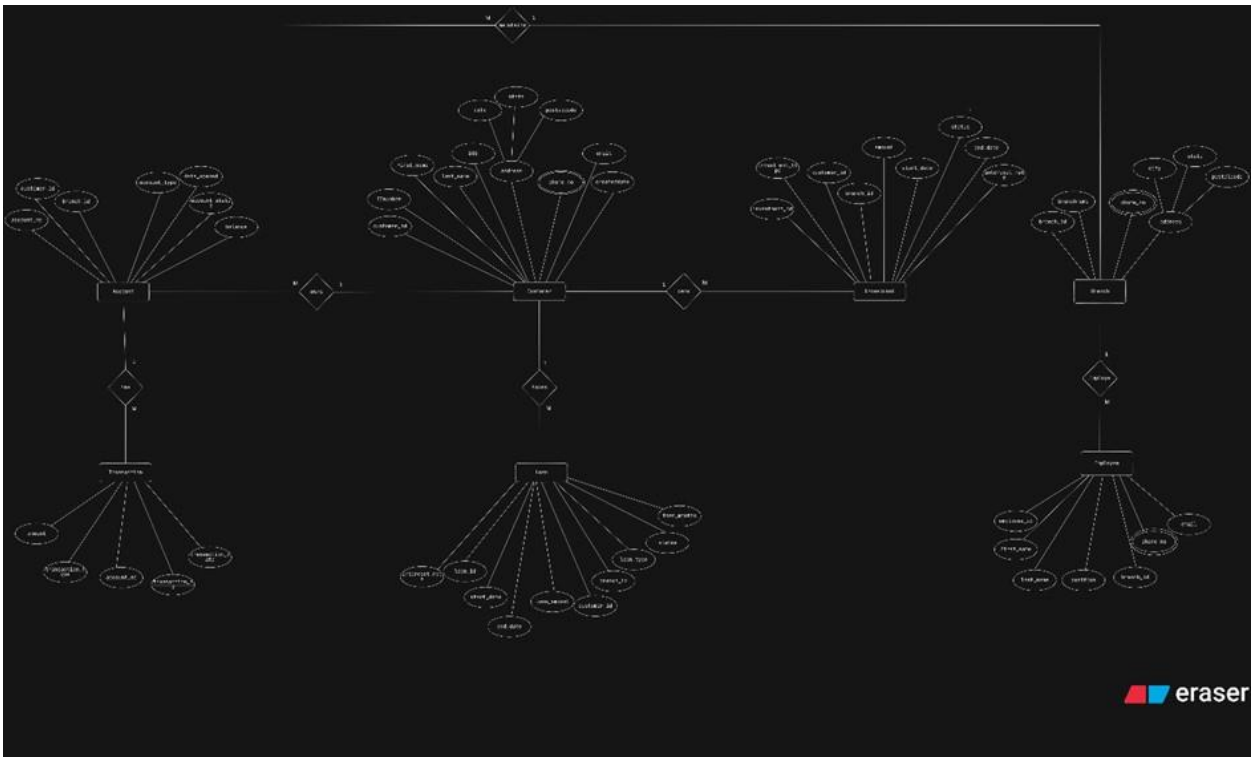
To address these issues, there is a growing need to modernize legacy banking systems by adopting a comprehensive, database-driven platform. Such a system would streamline daily banking activities, improve data accuracy, and ensure secure and reliable financial record management. By automating key processes and enforcing data integrity, the platform enhances operational efficiency and provides a faster, more transparent, and secure banking experience for both customers and employees.

Brief About The Project

The ***Banking and Financial Management System*** is designed to overcome limitations of traditional manual banking processes by providing a secure, centralized, and automated database solution. Conventional banking systems often involve paperwork, delayed updates, limited accessibility, and high chances of human errors. This leads to operational inefficiency, slow service delivery, and challenges in maintaining accurate customer and financial records.

This project aims to digitalize essential banking operations such as customer management, account creation, transactions, loan processing, branch management, employee allocation, and investment tracking. The system is built using a relational database model, ensuring structured storage, quick retrieval, and consistent handling of financial data. Core functionalities like triggers, stored procedures, functions, sequences, and constraints help enforce business rules automatically, improve data integrity, and reduce manual intervention

ER Diagram



ER To Tables

Customer **Account**

- **Relationship name:** owns
- **Cardinality:**
  - **One Customer** → can have **Many Accounts** 1 M
  - **One Account** → belongs to **one Customer Implementation:**
- Account.CustomerID **FK** referencing Customer.CustomerID
  -

□□ Customer □ Loan Relationship

- **name:** takes **Cardinality:**
- **One Customer** → can take **Many Loans** □1 □ M□
  - **One Loan** → belongs to **one Customer Implementation:**
  - Loan.CustomerID □ **FK** referencing Customer.CustomerID
-
  - **□□ Customer □ Investment**
- **Relationship name:** owns (or similar in your ER□ **Cardinality:**
- **One Customer** → can have **Many Investments** □1 □ M□
  - **One Investment** → belongs to **one Customer Implementation:**
  -
    - Investment.CustomerID □ **FK** referencing
- Customer.CustomerID

□□ Customer □ CustomerPhone (multi-valued attribute) Relationship

- **concept:** one customer, many phone numbers **Cardinality:**
- **One Customer** → **Many Phone Numbers** □1 □ M□
  - **One Phone record** → exactly **one Customer Implementation:**
  - CustomerPhone.CustomerID □ **FK** referencing Customer.CustomerID
- PK is composite: □CustomerID, PhoneNumber)
  -
  -

□□ Branch □ Account Relationship

- **name:** maintains **Cardinality:**
- **One Branch → Many Accounts** □1 □ M□
  - **One Account** → is maintained at **one Branch Implementation:**
  - Account.BranchID □ **FK** referencing Branch.BranchID
-
  - **□□ Branch □ Loan**
- **Relationship name:** (branch issues loan) **Cardinality:**
- **One Branch → Many Loans** □1 □ M□
  - **One Loan** → from **one Branch Implementation:**
  - Loan.BranchID □ **FK** referencing Branch.BranchID
- **□□ Branch □ Investment**
- **Relationship name:** (branch manages investment) **Cardinality:**
- **One Branch → Many Investments** □1 □ M□
  - **One Investment** → managed by **one Branch Implementation:**
  - Investment.BranchID □ **FK** referencing Branch.BranchID
-
  - **□□ Branch □ Employee**
- **Relationship name:** employs
- **Cardinality:**
  - **One Branch → Many Employees** □1 □ M□
  - **One Employee** → works at **one Branch Implementation:**
- Employee.BranchID □ **FK** referencing Branch.BranchID
  -

□□ Account □ BankTransaction □Transaction)

- **Relationship name:** has **Cardinality:**
- **One Account → Many Transactions** (1 : M)
  - **One Transaction** → belongs to **one Account Implementation:**
  - BankTransaction.AccountNumber → **FK** referencing Account.AccountNumber
- ## Normalization
  - This section explains how the database schema was normalized based on the ER model. The final structure ensures a clean and efficient banking database suitable for real-world operations.

Step 1: First Normal Form (1NF) All tables follow 1NF rules:

- **CustomerPhone table** handles multiple phone numbers separately, ensuring no repeated columns like Phone1, Phone2.
- **Customer,Account, Loan, Investment, Employee, and Branch tables** store atomic fields with no composite or multi-valued attributes.

✓ Result: The database contains clean, structured, and non-repetitive attribute data.

Step 2: Second Normal Form (2NF)

To satisfy 2NF, all non-key attributes must depend on the **entire primary key**.

- Most tables (Customer, Branch, Account, Loan, Investment, Employee) have a **single-column primary key**, so they are already in 2NF.
- **CustomerPhone** uses a composite key (**CustomerID, PhoneNumber)**, and both fields depend entirely on that key — there are no partial dependencies.

✓ Result: No attribute depends on part of a composite key, ensuring correctness and avoiding duplication.

Step 3: Third Normal Form (3NF) To satisfy 3NF, there must be:

- **No transitive dependency**
- Non-key fields should depend only on the primary key Changes applied:
- Derived values like **Account balance adjustments, loan EMI, maturity amount**, etc., are **not stored** — instead, they are computed using **functions and procedures** when needed.
- Branch, Account, Loan, and Investment relationships are maintained strictly through **foreign keys**, avoiding indirect dependencies.

✓ Result: The database eliminates redundancy, avoids update anomalies, and maintains strong referential integrity.

Summary of Normalization Benefits

By implementing 1NF, 2NF, and 3NF, the database achieves:

- Reduced redundancy
- Faster data retrieval and secure transaction handling
- No insertion, deletion, or update anomalies

Final Tables:

Before normalization: **7 tables**

After normalization: **8 optimized tables** (CustomerPhone added)

Table Name	Key Attributes (with PK, FK, Unique constraints)
Customer	CustomerID (**PK**), FirstName, LastName, DOB, Email (**UNIQUE**), IDNumber (**UNIQUE**), Street, City, State, PostalCode, CreatedDate
CustomerPhone	CustomerID (**FK → Customer.CustomerID**), PhoneNumber, **Composite PK (CustomerID, PhoneNumber)**
Branch	BranchID (**PK**), BranchName, Street, City, State, PostalCode, Phone
Employee	EmployeeID (**PK**), FirstName, LastName, Position, Salary, HireDate, Email, BranchID (**FK → Branch.BranchID**)
Account	AccountNumber (**PK**), CustomerID (**FK → Customer.CustomerID**), BranchID (**FK → Branch.BranchID**), AccountType, Balance, DateOpened, AccountStatus

BankTransaction	TransactionID **PK**, AccountNumber **FK Account.AccountNumber)**, TransactionType, Amount, TransactionDate, Description
Loan	LoanID **PK**, CustomerID **FK Customer.CustomerID**, BranchID **FK → Branch.BranchID**, LoanType, LoanAmount, InterestRate, TermMonths, StartDate, EndDate, Status
Investment	InvestmentID **PK**, CustomerID **FK Customer.CustomerID**, BranchID **FK Branch.BranchID**, InvestmentType, Amount, InterestRate, StartDate, EndDate, Status

Code Snippets Funds Transfer Procedure

```
CREATE OR REPLACE PROCEDURE Transfer_Amount(
    p_FromAccount    IN NUMBER,
    p_ToAccount      IN NUMBER,
    p_Amount         IN NUMBER,
    p_Description    IN VARCHAR2 DEFAULT 'Fund Transfer'
)
AS
    v_fromBalance NUMBER;
    v_count NUMBER;
BEGIN

    SELECT COUNT(*) INTO v_count
    FROM Account
    WHERE AccountNumber = p_FromAccount;

    IF v_count = 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Sender account does not exist.');
    END IF;

    SELECT COUNT(*) INTO v_count
    FROM Account
    WHERE AccountNumber = p_ToAccount;

    IF v_count = 0 THEN
        RAISE_APPLICATION_ERROR(-20002, 'Receiver account does not exist.');
    END IF;

    SELECT Balance INTO v_fromBalance
    FROM Account
    WHERE AccountNumber = p_FromAccount;

    IF v_fromBalance < p_Amount THEN
```

```
        RAISE_APPLICATION_ERROR(-20003, 'Insufficient balance for transfer.');
    END IF;

    UPDATE Account
    SET Balance = Balance - p_Amount
    WHERE AccountNumber = p_FromAccount;

    UPDATE Account
    SET Balance = Balance + p_Amount
    WHERE AccountNumber = p_ToAccount;

    INSERT INTO BankTransaction(AccountNumber, TransactionType, Amount,
TransactionDate, Description)
    VALUES (p_FromAccount, 'Transfer Out', p_Amount, SYSDATE, p_Descriptio
n);

    INSERT INTO BankTransaction(AccountNumber, TransactionType, Amount,
TransactionDate, Description)
    VALUES (p_ToAccount, 'Transfer In', p_Amount, SYSDATE, p_Description);
END;
/
begin
    TRANSFER_AMOUNT(1005,1003,5000,'Transfer');
end;
/
```

Before Transfer

	ACCOUNTNUMBER	BALANCE
1	1003	15000
2	1005	30000

After Transfer

	ACCOUNTNUMBER	BALANCE
1	1003	20000
2	1005	25000

Transaction Table

TRANSACTIONID	ACCOUNTNUMBER	TRANSACTIC	AMOUN	TRANSACTIONDATE	DESCRIPTION
14	1002	Transfer In	5000	11/21/2025, 6:08:28	Transfer
17	1005	Transfer Out	5000	11/21/2025, 6:11:30	Transfer
18	1003	Transfer In	5000	11/21/2025, 6:11:30	Transfer

Monthly Interest Posting Procedure

```
CREATE OR REPLACE PROCEDURE Post_Monthly_Interest(p_RatePercent IN N
UMBER
AS
BEGIN
   UPDATE Account
   SET Balance  Balance + Balance * (p_RatePercent / 100
   WHERE AccountType = 'Savings'
    AND AccountStatus = 'Active';
END;
/
begin
   Post_Monthly_Interest(2.5);
end;
/
```

Before balance of savings accounts

ACC	CUST	BRAI	ACCOUNTTYPE	BALANCE	DATEOPENEI	ACCOUNTSTATUS
1001	1	1	Savings	10000	11/19/2025,	Active
1002	2	1	Current	35000	11/19/2025,	Active
1003	3	2	Savings	20000	11/19/2025,	Active
1004	4	3	Savings	10000	11/19/2025,	Active

After updated balance

ACCOUNT	CUSTOMERID	BRANCHID	ACCOUNTTYPE	BALANCE	DATEOPENED	ACCOUNTSTATUS
1001	1	1	Savings	10250	11/19/2025,	Active
1002	2	1	Current	35000	11/19/2025,	Active
1003	3	2	Savings	20500	11/19/2025,	Active
1004	4	3	Savings	10250	11/19/2025,	Active

Loan EMI + Total Payable (Functions)

```
CREATE OR REPLACE FUNCTION Calculate_EMI
    p_Principal IN NUMBER,
    p_Rate      IN NUMBER,
    p_Months    IN NUMBER
 RETURN NUMBER
AS
    r NUMBER;
    n NUMBER;
    emi NUMBER;
BEGIN
    r := p_Rate / 100 / 12;
    n := p_Months;

    emi := p_Principal * r * POWER(1+r, n) / (POWER(1+r, n)  1);
```

```
    RETURN ROUND(emi, 2);
END;
/

CREATE OR REPLACE FUNCTION Loan_Total_Payable(
    p_Principal IN NUMBER,
    p_Rate      IN NUMBER,
    p_Months    IN NUMBER
) RETURN NUMBER
AS
    v_emi NUMBER;
BEGIN
    v_emi := Calculate_EMI(p_Principal, p_Rate, p_Months);
    RETURN ROUND(v_emi * p_Months, 2);
END;
/
```

LOANID	LOANAMOUNT	INTERESTRATE	TERMMONTHS	EMI	TOTALAMOUNT
1	500000	8.5	60	10258.27	615496.2
2	300000	9	48	7465.51	358344.48
3	150000	11.5	24	7026.05	168625.2
4	200000	10	36	6453.44	232323.84

Auto-Generated Primary Keys Using Sequences + Triggers & Add New Customer With Procedure

```
CREATE SEQUENCE CUSTOMER_SEQ START WITH 1 INCREMENT BY 1 ;
CREATE OR REPLACE TRIGGER BI_CUSTOMER
BEFORE INSERT ON CUSTOMER
FOR EACH ROW
BEGIN
  IF :NEW.CUSTOMERID IS NULL THEN
```

```
    SELECT CUSTOMER_SEQ.NEXTVAL INTO :NEW.CUSTOMERID FROM DUAL;
  END IF;
END;
/
begin
    Add_Customer(
    'Vijay','Bisht', DATE '1998-05-10', 'vijay@gmail.com', 'ID012',
    'Street 5','Delhi','Delhi','110005','9876543333');
end;
/
```

CUSTOMERID	FIRSTNAME	LASTNAME	DOB	EMAIL	IDNUMBER	STREET	CITY
1	Riya	Bisht	7/16/2005, 12:00:0(	riya@gmail.com	ID101	40 C	Chan
31	Vijay	Bisht	5/10/1998, 12:00:0(	vijay@gmail.com	ID012	Street 5	Delhi

Validation Trigger (Email Format Check)

```
CREATE OR REPLACE TRIGGER trg_customer_email_check
BEFORE INSERT OR UPDATE ON Customer
FOR EACH ROW
BEGIN
    IF :NEW.Email NOT LIKE '%_%.%' THEN
        RAISE_APPLICATION_ERROR(-20003, 'Invalid email format.');
    END IF;
END;
/
```

```
SQL> INSERT INTO Customer (FirstName, LastName, DOB, Email, IDNumber, Street, City, S
     VALUES ('Alisha','Rawat',DATE '2004-03-22','alisha@gmail','ID001172','Mandawali'

ORA-20003: Invalid email format.
ORA-06512: at "SQL_7D7VN3PZ6CS1F6T0SMEZLS7V0N.TRG_CUSTOMER_EMAIL_CHECK", line 3
ORA-04088: error during execution of trigger 'SQL_7D7VN3PZ6CS1F6T0SMEZLS7V0N.TRG_CUST
```

Views (Summary Views)

```
CREATE OR REPLACE VIEW Customer_Summary AS
SELECT
   c.CustomerID,
   c.FirstName || ' ' || c.LastName AS CustomerName,
   c.Email,c.City,c.State,
   (SELECT COUNT(*) FROM CustomerPhone cp WHERE cp.CustomerID = c.C ustomerID) AS
TotalPhones,
   (SELECT COUNT(*) FROM Account a WHERE a.CustomerID = c.CustomerI D) AS
TotalAccounts,
   (SELECT COUNT(*) FROM Loan l WHERE l.CustomerID = c.CustomerID) AS TotalLoans,
   (SELECT COUNT(*) FROM Investment i WHERE i.CustomerID = c.CustomerI
D) AS TotalInvestments
FROM Customer c
```

CUSTOMERID	FIRSTNAME	LASTNAME	DOB	EMAIL	IDNUMBER	STREET	CITY
1	Riya	Bisht	7/16/2005, 12:00:0(	riya@gmail.com	ID101	40 C	Chan
31	Vijay	Bisht	5/10/1998, 12:00:0(	vijay@gmail.com	ID012	Street 5	Delhi

Notification After Update

```
CREATE OR REPLACE TRIGGER trg_customer_after_update
AFTER UPDATE ON Customer
FOR EACH ROW
BEGIN
   DBMS_OUTPUT.PUT_LINE(
      'Customer updated: ID' || :OLD.CustomerID ||' | Old Name: ' || :OLD.First
Name || ' ' || :OLD.LastName ||
      ' | New Name: ' || :NEW.FirstName || ' ' || :NEW.LastName);
```

```
END;
/
```

```
SQL> UPDATE Customer
     SET firstName = 'sania'
     WHERE CustomerID = 5
```

```
Customer updated: 5 | Old Name: Sanya → New Name: sania
```

Business Logic Encapsulated in Stored Procedures

Add_Customer(FirstName,LastName,DOB,Email,IDNumber,Street,City,State,PostalCode,PhoneNumber)

Add_CustomerPhone(CustomerID, PhoneNumber)

Delete_Customer(CustomerID

Open_New_Account(CustomerID,BranchID,AccountType, OpeningBalance,Status(optional))

Close_Account(AccountNumber)

Do_Transaction(AccountNumber,TransactionType,Amount,Description (optional))

Post_Monthly_Interest(RatePercent)