

## 2 Problem understanding and ER Model

### 2.1 Identification of Entity and Relationships

An Entity-Relationship (ER) diagram is a visual representation of the entities (such as people, objects, or concepts) within a system and the relationships between them. It illustrates how different entities are connected or related to each other in a database. ER diagrams use various symbols to represent entities, attributes, and relationships, helping to visualize the structure of a database and aiding in the design and understanding of complex systems.

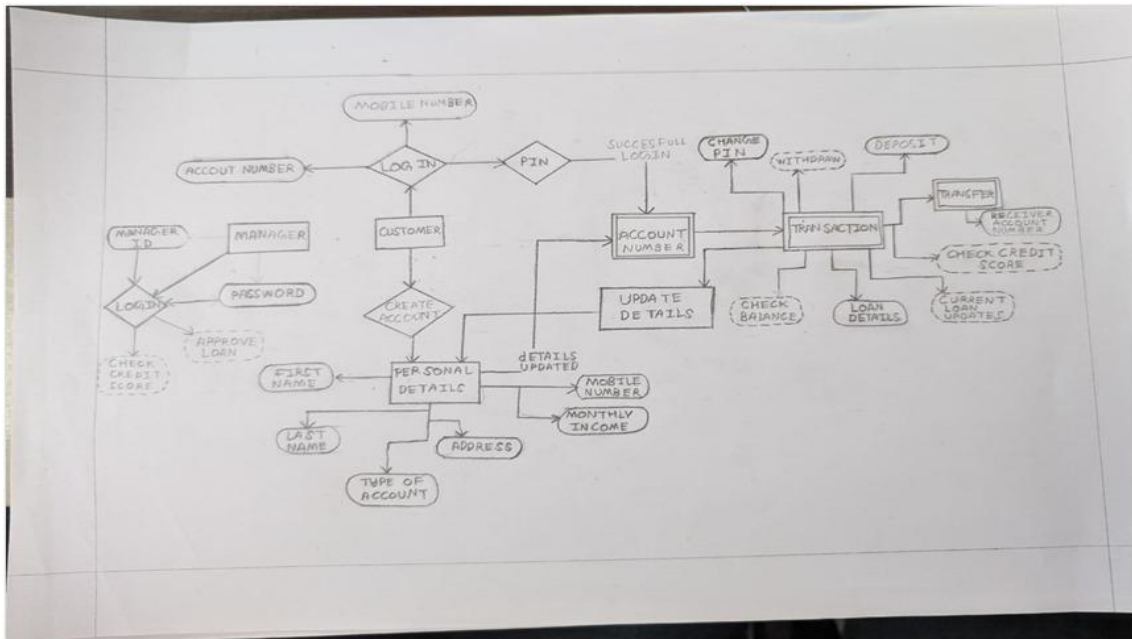


Figure 2.1

The provided ER diagram represents the Cardless Transaction Management System project.

### 2.2 Relational Schema Diagrams: -

Tables: - Represented by rectangles, each table in the diagram corresponds to a collection of related data. Tables typically represent entities such as customers, products, employees, etc.

Attributes (Columns): - Displayed within each table, attributes represent the specific data elements or fields. Attributes define the properties or characteristics of the entities represented by the table.

Relationships: - Lines connecting tables indicate relationships between them. The cardinality of relationships (e.g., one-to-one, one-to-many, many-to-many) is often represented using symbols or notation.

Manager		
Manager-Name	Manager-ID	Code
Sahil	Sa123	789
Sunaj	Su456	155

Personal Details							
First-Name	Last-Name	Type of Account	Account Number	Address	Monthly Income	Mobile Number	Mail
Nikhil	Sharma	Saving	42xxx97	P.F	2,00,000	98989999	98427@gmail.com
Luvish	Arora	Current	96xxx82	Estancia	2,00,000	41424242	La8742@gmail.com

Loan						
Loan-ID	Account Number	Monthly Income	Loan Amount	Date of Expiration	Collateral Description	Manager ID
142	42xxx87	2,00,000	15,00,000	2006-08-27	1045, Estancia	Sa123
165	96xxx82	2,00,000	10,00,000	2005-10-18	Cash Num:-4567	Su456

Credit Score				
Account Number	Current Loan ID	Loan Amount	Previous Loan Details	Current Credit Score
42xxx87	142	15,00,000	One month late	627
96xxx82	165	10,00,000	Two week late	702

Account					
First-Name	Last-Name	Account Number	Balance	Limit	Credit-Score
Yash	Srivastava	23xxx42	15,00,287	50,000	653
Nikhil	Sharma	42xxx87	1752,000	60,000	627

Fig. - 2.2

BRANCH			
BRANCH-ID	BRANCH-NAME	LOCATION	MANAGER-ID
101	Paschim DIST	LUCKNOW	Sa123
102	Akshaya DIST	VALSAD	Su456

TRANSACTION HISTORY					
SENDER-ID	RECEIVER-ID	AMOUNT	DATE	TRANSACTION-ID	MANAGER-ID
456BK624L	848LMA246Y	₹ 20	2004-07-24 LLMYY24L0L		Sa123
989BC489A	456BK624LA	₹ 80	2004-2-24 LLMYY24L0Y		Su456

INVESTMENT						
INVESTMENT-ID	ACCOUNT-NO	INVESTMENT-TYPE	INVESTMENT-AMT	START-DATE	END-DATE	MANAGER-ID
AJ82C375	44428912486	SAVINGS	₹ 130000	2006-12-24	2024-12-24	Sa123
X426PA12	77124891284	CURRENT	₹ 180000	2008-07-27	2026-07-27	Su456

EXPENSE CATEGORY	
EXPENSECATEGORY-ID	CATEGORY-NAME
10018	FOOD EXPENSES
10012	TAXES

EXPENSES				
EXPENSE-ID	EXPENSECATEGORY-ID	AMOUNT	DATE	MANAGER-ID
AB456PS1	10012	₹ 2000	2004-12-14	Su456
LMA04681	10018	₹ 17124	2008-11-21	Sa123

LUVISH ARORA (RA2211003011296)  
 SHARMA (RA2211003011298)  
 SRIVASTAVA (RA2211003011302)

Fig. - 2.3

### 3 Design of Relational Schemas

Based on the description provided, we can design a relational schema for the bank management system. Below is a simplified schema representation:

1. Manager (manager\_id, manager\_name, code): -
  - Manager\_ID (Primary Key)
  - Manager\_Name
  - Code
  
2. Branch (branch\_id, branch\_name, location, manager\_id): -
  - Branch\_ID (Primary Key)
  - Branch\_Name
  - Location
  - Manager\_ID (Foreign Key referencing Manager)
  
3. Personal\_Details (account\_number, first\_name, last\_name, account\_type, address, monthly\_income, email, mobile\_number, manager\_id): -
  - Account\_Number (Primary Key)
  - First\_Name
  - Last\_Name
  - Account\_Type
  - Address
  - Monthly\_Income
  - Email
  - Mobile\_Number
  - Manager\_ID (Foreign Key referencing Manager)
  
4. Loan (loan\_id, account\_number, monthly\_income, loan\_amount, collateral\_desc, manager\_id): -
  - Loan\_ID (Primary Key)
  - Account\_Number (Foreign Key referencing Personal\_Details)
  - Monthly\_Income
  - Loan\_Amount
  - Collateral\_Desc
  - Manager\_ID (Foreign Key referencing Manager)

5. Transfer\_History(transaction\_id, sender\_id, receiver\_id, amount, date, manager\_id):-
- Transaction\_ID (Primary Key)
  - Sender\_ID (Foreign Key referencing Personal\_Details)
  - Receiver\_ID (Foreign Key referencing Personal\_Details)
  - Amount
  - Date
  - Manager\_ID (Foreign Key referencing Manager)
6. Credit\_Score(account\_number,current\_loan\_id, loan\_amount, previous\_loan\_details, current\_credit\_score, manager\_id):-
- Account\_Number (Primary Key, Foreign Key referencing Personal\_Details)
  - Current\_Loan\_ID (Foreign Key referencing Loan)
  - Loan\_Amount
  - Previous\_Loan\_Details
  - Current\_Credit\_Score
  - Manager\_ID (Foreign Key referencing Manager)
7. Account(account\_number, balance, limit, credit\_score, manager\_id):-
- Account\_Number (Primary Key, Foreign Key referencing Personal\_Details)
  - Balance
  - Limit
  - Credit\_Score
  - Manager\_ID (Foreign Key referencing Manager)
8. Investment(investment\_id, account\_number, investment\_type, investment\_amount, start\_date, end\_date, manager\_id):-
- Investment\_ID (Primary Key)
  - Account\_Number (Foreign Key referencing Personal\_Details)
  - Investment\_Type
  - Investment\_Amount
  - Start\_Date

- End\_Date
- Manager\_ID (Foreign Key referencing Manager)

9. Expense\_Category(expense\_category\_id, category\_name):-

- Expense\_Category\_ID (Primary Key)
- Category\_Name

10. Expenses(expense\_id, expense\_category\_id, amount, date, manager\_id):-

- Expense\_ID (Primary Key)
- Expense\_Category\_ID (Foreign Key referencing Expense\_Category)
- Amount
- Date
- Manager\_ID (Foreign Key referencing Manager)

This schema provides a basic structure for the cardless transaction system management functionalities. The schema diagram consists of several tables like Account, Loan, Manager Investment, and Branch etc. An account can have many loans or investments (one-to-many relationships), while a branch might have one manager (one-to-many) or a manager could oversee multiple branches (potentially many-to-many).

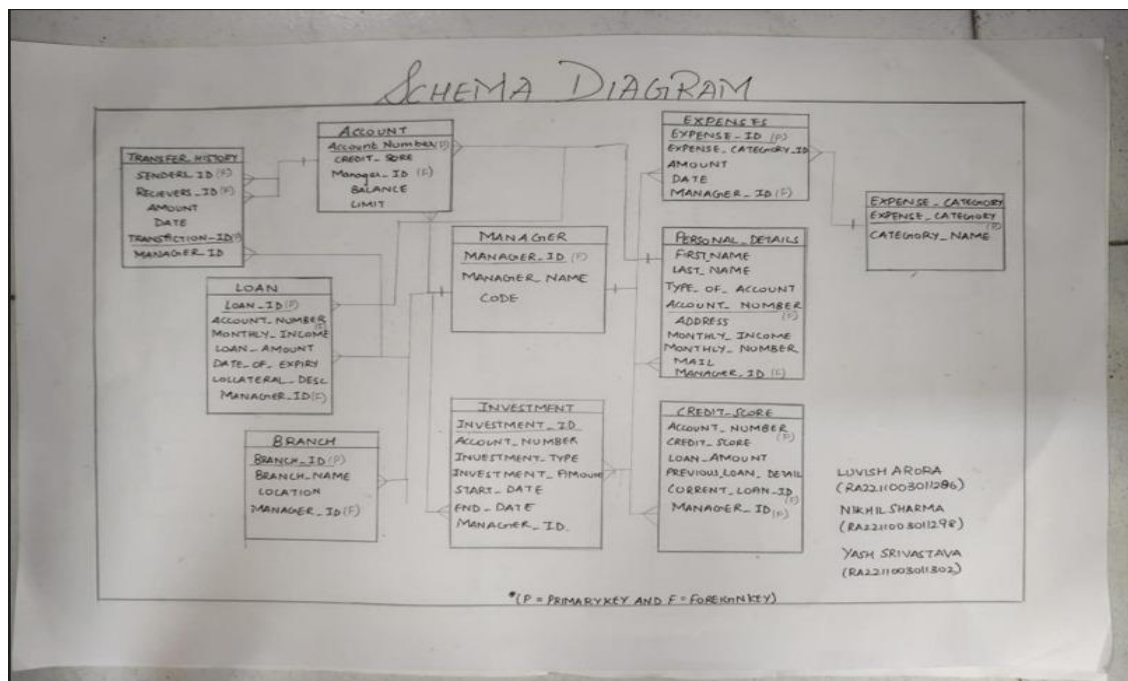


Figure 3.1

## 4 Creation of Cardless Transaction Management

### 4.1 DATA DEFINATION COMMANDS: -

#### CREATING TABLE

Basically, in the cardless management system, there are 9 entities i.e Account, Expenses, Loan, Manager, Branch, Investment\_Account, Customer\_Account, Personal\_Details, Saving\_Account.

#### CREATING DATABASE: -

```
CREATE DATABASE cardless_transaction;  
USE cardless_transaction
```

#### Manager: -

```
CREATE TABLE Manager (  
    manager_id INT PRIMARY KEY,  
    manager_name VARCHAR(50),  
    code VARCHAR(20)  
);
```

#### BRANCH:-

```
CREATE TABLE Branch (  
    branch_id INT PRIMARY KEY,  
    branch_name VARCHAR(50),  
    Location VARCHAR(100),  
    manager_id INT,  
    FOREIGN KEY (manager_id) REFERENCES Manager(manager_id)  
);
```

#### Personal Details:-

```
CREATE TABLE Personal_Details (  
    account_number INT PRIMARY KEY,  
    first_name VARCHAR(10),  
    last_name VARCHAR(50),  
    account_type VARCHAR(50),  
    address VARCHAR(100),  
    monthly_income DECIMAL(10, 2),  
    email VARCHAR(100),  
    mobile_number VARCHAR(20),  
    manager_id INT,  
    FOREIGN KEY (manager_id) REFERENCES Manager(manager_id)  
);
```

### **Loan:-**

```
CREATE TABLE Loan (  
  loan_id INT PRIMARY KEY,  
  account_number INT,  
  monthly_income DECIMAL(10, 2),  
  loan_amount DECIMAL(10, 2),  
  collateral_desc VARCHAR(100),  
  manager_id INT,  
  FOREIGN KEY (account_number) REFERENCES Personal_Details(account_number),  
  FOREIGN KEY (manager_id) REFERENCES Manager(manager_id)  
);
```

### **CREDIT SCORE:-**

```
CREATE TABLE Credit_Score (  
  account_number INT PRIMARY KEY,  
  current_loan_id INT,  
  loan_amount DECIMAL(10, 2),  
  previous_loan_details VARCHAR(100),  
  current_credit_score INT,  
  manager_id INT,  
  FOREIGN KEY (account_number) REFERENCES Personal_Details(account_number),  
  FOREIGN KEY (current_loan_id) REFERENCES Loan(loan_id),  
  FOREIGN KEY (manager_id) REFERENCES Manager(manager_id)  
);
```

### **ACCOUNT:-**

```
CREATE TABLE Account (  
  account_number INT PRIMARY KEY,  
  balance DECIMAL(10, 2),  
  'limit' DECIMAL(10, 2), -- Escaping the reserved keyword LIMIT  
  credit_score INT,  
  manager_id INT,  
  FOREIGN KEY (account_number) REFERENCES Personal_Details(account_number),  
  FOREIGN KEY (manager_id) REFERENCES Manager(manager_id)  
);
```

### **TRANSFER HISTORY:-**

```
CREATE TABLE Transfer_History (  
  transaction_id INT PRIMARY KEY,  
  sender_id INT,  
  receiver_id INT,  
  amount DECIMAL(10, 2),  
  date DATE,  
  manager_id INT,  
  FOREIGN KEY (sender_id) REFERENCES Personal_Details(account_number),  
  FOREIGN KEY (receiver_id) REFERENCES Personal_Details(account_number),  
  FOREIGN KEY (manager_id) REFERENCES Manager(manager_id)  
);
```

### **TRANSFER HISTORY:-**

```
CREATE TABLE Transfer_History (  
  transaction_id INT PRIMARY KEY,  
  sender_id INT,  
  receiver_id INT,  
  amount DECIMAL(10, 2),  
  date DATE,  
  manager_id INT,  
  FOREIGN KEY (sender_id) REFERENCES Personal_Details(account_number),  
  FOREIGN KEY (receiver_id) REFERENCES Personal_Details(account_number),  
  FOREIGN KEY (manager_id) REFERENCES Manager(manager_id)  
);
```

### **INVESTMENT:-**

```
CREATE TABLE Investment (  
  investment_id INT PRIMARY KEY,  
  account_number INT,  
  investment_type VARCHAR(50),  
  investment_amount DECIMAL(10, 2),  
  start_date DATE,  
  end_date DATE,  
  manager_id INT,  
  FOREIGN KEY (account_number) REFERENCES Personal_Details(account_number),  
  FOREIGN KEY (manager_id) REFERENCES Manager(manager_id)  
);
```

### **EXPENSE CATEGORY: -**

```
CREATE TABLE Expense_Category (  
  expense_category_id INT PRIMARY KEY,  
  category_name VARCHAR(50)  
);
```

### **EXPENSES: -**

```
CREATE TABLE Expenses (  
  expense_id INT PRIMARY KEY,  
  expense_category_id INT,  
  amount DECIMAL(10, 2),  
  date DATE,  
  manager_id INT,  
  FOREIGN KEY (expense_category_id) REFERENCES  
Expense_Category(expense_category_id),  
  FOREIGN KEY (manager_id) REFERENCES Manager(manager_id)  
);
```



## **ALTER TABLE**

### **ADD COLUMN: -**

ALTER TABLE account add(first\_name varchar(100));

### **DROP COLUMN: -**

ALTER TABLE account drop column first\_name;

### **MODIFY COLUMN: -**

ALTER TABLE Personal\_Details MODIFY first\_name VARCHAR(50);

## 4.2 DATA MANIPULATION COMMANDS: -

A DML statement is executed to Add new rows to a table, modify existing rows in a table, Remove existing rows from a table. A transaction consists of a collection of DML statements that form a logical unit of work.

### INSERT INTO: -

```
INSERT INTO Manager (manager_id, manager_name, code)
VALUES (1, 'Alice Smith', 'ABC123');
```

```
INSERT INTO Branch (branch_id, branch_name, location, manager_id)
VALUES (1, 'Main Branch', 'City Center', 1);
```

```
INSERT INTO Personal_Details (account_number, first_name, last_name, account_type,
address, monthly_income, email, mobile_number, manager_id)
VALUES (123456789, 'John', 'Doe', 'Savings', '123 Main St', 50000,
'john.doe@example.com', '555-1234', 1);
```

```
INSERT INTO Loan (loan_id, account_number, monthly_income, loan_amount,
collateral_desc, manager_id)
VALUES (1, 1, 5000.00, 10000.00, 'Car', 1);
```

```
INSERT INTO Transfer_History (transaction_id, sender_id, receiver_id, amount, date,
manager_id)
```

```
VALUES (1, 1, 2, 100.00, '2024-03-28', 1);
```

```
INSERT INTO Credit_Score (account_number, current_Loan_id, loan_amount,
previous_loan_details, current_credit_score, manager_id)
```

```
VALUES (1, 1, 10000.00, 'Previous loans paid on time', 700, 1);
```

```
INSERT INTO Account (account_number, balance, 'limit', credit_score, manager_id)
```

```
VALUES (1, 5000.00, 10000.00, 700, 1);
```

```
INSERT INTO Investment (investment_id, account_number, investment_type,
investment_amount, start_date, end_date, manager_id)
```

```
VALUES (1, 1, 'Stocks', 2000.00, '2024-01-01', '2024-12-31', 1);
```

```
INSERT INTO Expense_Category (expense_category_id, category_name)
```

```
VALUES (1, 'Food');
```

```
INSERT INTO Expenses (expense_id, expense_category_id, amount, date, manager_id)
```

```
VALUES (1, 1, 50.00, '2024-03-28', 1);
```

### **UPDATING VALUES: -**

#### **UPDATING PERSONAL DETAILS: -**

```
UPDATE Personal_Details  
SET first_name = 'New John'  
WHERE account_number = 1;
```

#### **UPDATING LOAN: -**

```
UPDATE Loan  
SET monthly_income = 6000.00  
WHERE account_number = 1;
```

#### **UPDATING PERSONAL DETAILS: -**

```
UPDATE Personal_Details  
SET monthly_income = 6500.00,  
email = 'jane.smith@example.org'  
WHERE account_number = 2;
```

## **4.1 DATA QUERY LANGUAGE COMMANDS (DQL):**

### **SELECT STATEMENT:-**

#### **SELECT COMMAND: -**

```
Select * from personal_details;
```

#### **SELECT COMMAND USING WHERE CLAUSE: -**

```
Select * from manager where manager_id>2;
```

#### **SELECT COMMAND USING COUNT BY GROUP BY: -**

```
Select account_type, COUNT(*) AS count_saving_account  
FROM Personal_Details  
GROUP BY account_type;
```

## 4.3 TRANSACTION CONTROL LANGUAGE COMMANDS: -

### TRANSACTION AND COMMIT: -

```
mysql> Start a transaction --
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql>
mysql> Update balance for account_number 1 --
mysql> UPDATE Account
-> SET balance balance 100
-> WHERE account_number = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
mysql>
mysql> - Update balance for account_number 2
mysql> UPDATE Account
-> SET balance balance + 100
-> WHERE account_number = 2;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
mysql>
mysql> -- Insert a new expense
mysql> INSERT INTO Expenses (expense_category_id, amount, date, manager_id)
-> VALUES (1, 100.00, '2024-03-28', 1);
ERROR 1364 (HY000): Field 'expense_id' doesn't have a default value
mysql>
mysql> -- If everything is successful, commit the transaction
mysql> COMMIT;
Query OK, 0 rows affected (0.00 sec)
mysql>
mysql> -- If there's an issue, rollback the transaction to undo the changes
mysql> ROLLBACK;
Query OK, 0 rows affected (0.00 sec)
```

### USING SAVEPOINT AND ROLLBACK: -

```
mysql> Start a transaction --
mysql> START TRANSACTION;
Query OK, 0 rows affected (0.00 sec)
mysql>
mysql> Update balance for account_number 1 --
mysql> UPDATE Account
-> SET balance balance 100
```

-> WHERE account\_number = 1;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql>

mysql> - Update balance for account\_number 2

mysql> UPDATE Account

-> SET balance balance + 100

-> WHERE account\_number = 2;

Query OK, 1 row affected (0.00 sec)

Rows matched: 1 Changed: 1 Warnings: 0

mysql>

mysql> -- Insert a new expense

mysql> INSERT INTO Expenses (expense\_category\_id, amount, date, manager\_id)

-> VALUES (1, 100.00, '2024-03-28', 1);

ERROR 1364 (HY000): Field 'expense\_id' doesn't have a default value

mysql>

mysql> -- If everything is successful, commit the transaction

mysql> COMMIT;

Query OK, 0 rows affected (0.00 sec)

mysql>

mysql> -- If there's an issue, rollback the transaction to undo the changes

mysql> ROLLBACK;

Query OK, 0 rows affected (0.00 sec)

## **SUBQUERIES: -**

### **SUBQUERY TO RETRIVE THE MANAGER DETAILS FOR EACH BRANCH:-**

```
mysql> SELECT branch_id, branch_name, location, (SELECT manager_name FROM
Manager
WHERE Manager.manager_id = Branch.manager_id) AS manager_name
FROM Branch;
```

### **SUBQUERY TO CALCULATE THE TOTAL LOAN AMOUNT FOR EACH ACCOUNT TILL ACCOUNT NUMBER 3:-**

```
mysql> SELECT account_number,
(SELECT SUM(Loan_amount)
FROM Loan
WHERE Loan.account_number = Account.account_number) AS total_loan_amount
FROM Account
WHERE account_number <= 3;
```

## 5. Complex queries based on the concepts of constraints, sets, joins, views, Triggers and Cursors.

Sure, let's delve into each of these concepts within the context of a bank management system:

### 1. Constraints: Constraints are rules enforced on data columns within a

table

#### Primary Key Constraint:

Ensures each record in a table is unique and not null. For example, ensuring each account has a unique account number.

In a cardless transaction management system, the primary key constraint ensures that each transaction record has a unique identifier.

#### Foreign Key Constraint:

Maintains referential integrity between two related tables. For instance, ensuring that transactions are linked to valid account numbers.

In a cardless transaction system, foreign key constraints ensure that transactions are associated with valid customer accounts.

#### Check Constraint:

Limits the values that can be inserted into a column. For example, restricting withdrawals to positive amounts.

Check constraints can be used to validate transaction amounts in a cardless system, ensuring they meet specified criteria.

#### Unique Constraint:

Ensures that all values in a column are unique. This might be used to ensure uniqueness of customer IDs.

Unique constraints ensure that each customer in a cardless system has a distinct identifier.

### 2. Sets, Joins,

#### Views: Sets:

Collections of distinct values manipulated in queries, allowing operations like union, intersection, and difference.

Sets can be utilized in a cardless system to identify common transaction patterns or detect anomalies in customer behavior.

#### Joins:

Combining rows from multiple tables based on related columns.

Joins are employed to merge customer data with transaction records in a cardless system, enabling comprehensive analysis.

#### Views:

Virtual tables representing the result of stored queries, facilitating simplified data access.

Views in a cardless system can present aggregated transaction data or real-time account balances for customer convenience.

### 3. Triggers:

#### Transaction Trigger:

SQL code executed automatically in response to specified events on a table.

Application: Transaction triggers in a cardless system update customer balance or send notifications for high-value transactions.

#### Security Trigger:

Trigger enforcing security measures such as authentication checks before executing transactions

#### 4. Cursors:

##### Transaction Cursor:

Database objects used to retrieve data one row at a time, suitable for complex logic. Cursors can be employed in a cardless system to iterate through transaction records for detailed analysis or auditing purposes.

##### Authorization Cursor:

Cursor verifying customer permissions or account status before processing transactions.

Authorization cursors ensure that customers meet specified criteria, such as account balance thresholds, before executing transactions in a cardless system.

### 5.1 Constraints:

- a. Query to retrieve foreign key constraints for the 'Loan' table:

```
SELECT column_name, constraint_name, referenced_column_name, referenced_table_name
```

e

```
-> FROM information_schema.key_column_usage
```

```
-> WHERE table_name = 'Loan';
```

This SQL query fetches key column details from the 'Loan' table, facilitating a deeper understanding of database constraints and relationships.

- b. Check Constraint - Transaction Management:

```
SELECT column_name, constraint_name, referenced_column_name, referenced_table_name
```

e

```
-> FROM information_schema.key_column_usage
```

```
-> WHERE table_name = 'Transfer_History';
```

This SQL query retrieves key column information for the 'Transfer\_History' table, aiding in understanding its constraints and relationships within the database schema.

### 5.2 Sets, Joins, Views:

#### Join Query - Customer and Account Management:

[Inner Join]

```
> SELECT *
```

```
-> FROM Branch
```

```
-> INNER JOIN Manager ON Branch.manager_id = Manager.manager_id;
```

This SQL query performs an inner join between the 'Branch' and 'Manager' tables based on the 'manager\_id' column, linking branches to their respective managers.



```
[Left Join]
SELECT *
FROM Loan
LEFT JOIN Personal_Details ON Loan.account_number=Personal_Details.account_number;
```

This SQL query executes a left join between the 'Loan' table and the 'Personal\_Details' table based on the 'account\_number' column, combining loan information with personal details where available.

```
[Right Join]
SELECT
FROM Personal_Details
RIGHT JOIN Expenses ON Personal_Details.manager_id = Expenses.manager_id;
```

This SQL query performs a right join between the 'Personal\_Details' table and the 'Expenses' table, connecting personal details with expense records based on the 'manager\_id' column.

```
[Full Outer Join]
SELECT *
FROM Account
LEFT JOIN Investment ON Account.account_number = Investment.account_number
UNION
SELECT *
FROM Account
RIGHT JOIN Investment ON Account.account_number = Investment.account_number
WHERE Account.account_number IS NULL;
```

This SQL query combines the results of a left join and a right join between the 'Account' and 'Investment' tables, ensuring all records from both tables are included. It accounts for unmatched records in 'Account' or 'Investment' tables.

Views:

```
CREATE VIEW Monthly_Expense_Report AS
SELECT MONTH(e.date) AS month, e.expense_category_id, ec.category_name,
SUM(e.amount) AS total_expense
FROM Expenses e
INNER JOIN Expense_Category ec ON e.expense_category_id = ec.expense_category_id
GROUP BY MONTH(e.date), e.expense_category_id, ec.category_name;
```

We create a view named Monthly\_Expense\_Report, selecting the month, expense category ID, category name, and total expense amount for each category monthly by joining the Expenses table with the Expense\_Category table on the expense\_category\_id column and aggregating the total expense using SUM(), followed by grouping the results by month, expense category ID, and category name..

```
CREATE VIEW Manager_Branch_Details AS
SELECT m.manager_id, m.manager_name, m.code, b.branch_id, b.branch_name, b.location
FROM Manager m
LEFT JOIN Branch b ON m. manager_id = b.manager_id;
```

View consolidates manager and branch information, ensuring inclusion of all managers, even those not associated with any branch, providing a comprehensive overview of managerial oversight and branch locations.

#### **4.1 Triggers:**

##### **Trigger for Loan Status Update - Transaction Management**

```
CREATE TRIGGER Update_Loan_Status -> AFTER INSERT ON Transfer_History
FOR EACH ROW
BEGIN
DECLARE total_loan DECIMAL(10, 2); DECLARE account_balance DECIMAL(10, 2);
-Calculate the total loan amount for the sender SELECT SUM(Loan_amount) INTO total_loan
FROM Loan
WHERE account_number = NEW.sender_id;
Retrieve the account balance for the sender SELECT balance INTO account_balance
FROM Account WHERE account_number = NEW.sender_id;
Check if the account balance is sufficient to cover the total loan amount
IF account_balance >= total_loan THEN
Update the loan status to 'Paid' if the balance is sufficient
UPDATE Loan SET Loan status = 'Paid'
WHERE account_number = NEW.sender_id;
ELSE
END IF;
Update the loan status to 'Overdue' if the balance is insufficient
UPDATE Loan
SET loan status = 'Overdue' WHERE account _number NEW.sender_id;
END;
```

This trigger updates the loan\_status column in the Loan table based on the comparison between the total loan amount for a sender and their account balance after each insertion into the Transfer\_History table..

##### **Account Credit Score Update Trigger on Personal Details Insertion**

```
CREATE TRIGGER Update_Account_Credit_Score AFTER INSERT ON Personal_Details
FOR EACH ROW
BEGIN
Update the credit_score column in the Account table
UPDATE Account
SET credit_score=NEW.monthly_income*0.1 WHERE account_number=NEW.account_number;
END;
```

This trigger updates the credit\_score column in the Account table with a calculated value based on the monthly\_income of a newly inserted row in the Personal\_Details table.

### 5.3 Cursors:

Branch Information Retrieval Using Cursor and Handler

```
DECLARE branch_id_val INT;
DECLARE branch_name_val VARCHAR(50); DECLARE location_val VARCHAR(100);
DECLARE done BOOLEAN DEFAULT FALSE;
Declare cursor for branches
DECLARE branch_cursor CURSOR FOR SELECT branch_id, branch_name, location
FROM Branch;
Declare handler for branches cursor
DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;
-Open cursor
OPEN branch_cursor;
Loop through branches
branch_loop: LOOP
FETCH branch_cursor INTO branch_id_val, branch_name_val, location_val;
IF done THEN
LEAVE branch_loop;
END IF;
Output branch information
SELECT CONCAT('Branch ID:', branch_id_val, ', Branch Name:', branch_name_val, ',
Location:', location_val);
```

This query calculates the average balance of all accounts by iterating over account balances using a cursor and then computing the average

## 6. Analyzing the pitfalls, identifying the dependencies, and applying normalizations

In database management, functionalities refer to the capabilities or operations that a database system provides to users for storing, retrieving, manipulating, and managing data. These functionalities ensure that the data is organized, accessible, and secure. Some common functionalities in a database management system (DBMS) include:

1 Transaction Storage: This feature involves securely storing transaction data in a structured manner within the database, typically employing tables, rows, and columns.

2 Transaction Retrieval: Users can efficiently retrieve transaction data from the database using various querying methods, filters, and search criteria to access relevant transactional details.

3 Transaction Processing: Users are empowered to perform critical operations such as initiating, authorizing, and processing transactions, as well as updating transaction records within the database seamlessly.

4 Data Security: Database systems implement robust mechanisms to safeguard transaction data, including stringent user authentication protocols, access control measures, and encryption techniques to protect sensitive transactional information.

5 Concurrency Control: This functionality ensures smooth handling of simultaneous transactional operations by multiple users, mitigating conflicts and inconsistencies to maintain data integrity and reliability.

6 Transaction Integrity: Database systems enforce stringent data integrity constraints to uphold the accuracy and consistency of transactional data, including enforcing unique identifiers, maintaining referential integrity, and validating transactional inputs.

7 Audit Trails: Transaction management systems incorporate comprehensive audit trail features to track and log all transaction-related activities, ensuring accountability and transparency in transaction processing.

In a database management system, functional dependencies are constraints that describe the relationships between attributes (columns) in a relation (table). There are different types of functional dependencies:

1. Single-valued Dependency (SV):

- Occurs when one attribute uniquely determines another attribute in transaction records, ensuring data consistency and integrity.

2. Multi-valued Dependency (MVD):

- Describes a relationship where a set of attributes functionally determines another set of attributes within transaction data, independent of primary keys..

3. Transitive Dependency (TD):

- Involves one attribute determining another attribute through a third attribute, which is crucial for maintaining data consistency in transaction processing..

4. Partial Dependency:

- Occurs when an attribute is functionally dependent on only a part of the transaction data, rather than the entire dataset, which can lead to data redundancy and anomalies.

5. Functional Dependency Closure:

- Represents the set of all functional dependencies logically implied by a given set of functional dependencies in transaction data, aiding in normalization and data modeling.

Understanding and identifying these functional dependencies is crucial in database design, normalization, and ensuring data integrity.

Normalization in cardless transaction management involves organizing transaction data efficiently to minimize redundancy and dependency, thereby ensuring data integrity and consistency. Normal forms such as First Normal Form (1NF), Second Normal Form (2NF), Third Normal Form (3NF), Boyce-Codd Normal Form (BCNF), Fourth Normal Form (4NF), and Fifth Normal Form (5NF) help in achieving optimal database designs for cardless transaction processing.

1 First Normal Form (1NF): Transaction data is organized into tables where each column contains atomic values, ensuring no repeating groups of columns, which enhances data organization and accessibility.

2 Second Normal Form (2NF): To attain 2NF, transactional tables must adhere to 1NF standards, with all non-key attributes being fully functionally dependent on the primary key. This ensures data integrity and reduces redundancy in transaction records.

3 Third Normal Form (3NF): Achieving 3NF entails meeting 2NF requirements and eliminating transitive dependencies within transactional tables. This ensures that non-key attributes are not dependent on other non-key attributes, enhancing data consistency and reliability.

4 Boyce-Codd Normal Form (BCNF): BCNF, a stricter version of 3NF, mandates that every determinant within transactional tables is a candidate key. This stringent normalization minimizes data redundancy and ensures robust database design.

5 Fourth Normal Form (4NF): 4NF aims to eliminate multi-valued dependencies by decomposing transactional tables into smaller, more granular tables. This enhances data organization and reduces dependency issues in transaction records.

6 Fifth Normal Form (5NF): Also known as Project-Join Normal Form (PJNF), 5NF addresses scenarios where transactional tables contain join dependencies. By resolving these dependencies, 5NF promotes more efficient and well-structured database designs, facilitating seamless transaction management

Each normal form addresses specific types of data redundancy and dependency issues, with higher normal forms generally resulting in more efficient and well-structured database designs.

### **Pitfalls in Relational Database Design**

#### 1] Redundancy:

Storing the same data in multiple places can lead to inconsistencies and inefficiencies.

#### 2] Inconsistency:

Data inconsistencies can occur when different parts of the database hold different versions of the same data.

#### 3] Inefficiency:

Poor database design can result in slower query performance and increased storage requirements.

#### 4] Complexity:

A complex database schema can be difficult to understand and maintain, leading to errors and inefficiencies.

Ø Functional dependency (FD) is a constraint that specifies the relationship between two attributes in a database table.

### **Functional Dependency in Transaction Table**

TransactionID  $\rightarrow$  {CustomerID, TransactionDate, TotalAmount}

CustomerID  $\rightarrow$  {TransactionID, TransactionDate,

TotalAmount} TransactionDate  $\rightarrow$

{TransactionID, CustomerID, TotalAmount}

TotalAmount  $\rightarrow$  {TransactionID, CustomerID, TransactionDate}

### **Functional Dependency in Personal\_details Table**

Account\_number  $\rightarrow$  first\_name  
Account\_number  $\rightarrow$  last\_name  
Account\_number  $\rightarrow$   
mobile\_number mobile\_number  $\rightarrow$   
email email  $\rightarrow$  first\_name  
Mobile\_number  $\rightarrow$  first\_name  
Mobile\_number  $\rightarrow$  last\_name  
Manager\_id of personal\_details  $\rightarrow$  manager\_id of manager\_details

### **Functional Dependency in Manager table**

Manager\_id  $\rightarrow$  manager\_name  
Manager\_id  $\rightarrow$  code

### **Functional Dependency in Branch**

Branch\_id  $\rightarrow$  branch\_name  
Branch\_name, location  $\rightarrow$  branch\_id  
Branch\_id, branch\_name  $\rightarrow$  manager\_id

### **Functional Dependency in Loan Table**

Loan\_id  $\rightarrow$  account\_number  
Loan\_amount, collateral\_desc  $\rightarrow$  manager\_id  
Account\_number  $\rightarrow$  monthly\_income

### **Functional Dependency in Investment Table**

Investment\_id  $\rightarrow$  investment\_type  
Investment\_id, investment\_type  $\rightarrow$  start\_date  
Start\_date  $\rightarrow$  time\_period  
time  $\rightarrow$  end\_date  
Account\_number  $\rightarrow$  manager\_id

## Normalization

It eliminates redundancy data. It ensures data dependency makes sense.

### FIRST NORMAL FORM (1NF)

A relation is said to be in First Normal Form If and only if all the attributes of relation are atomic (Single valued) in nature.

It must not contain any multi valued (or) composite attributes.

Personal\_details – Rules violated

Personal\_details Table- Address is not Atomic(violating 1NF)

**Input Table**

Field	Type	Null	Key	Default	Extra
account_number	int	NO	PRI	NULL	
first_name	varchar(50)	YES		NULL	
last_name	varchar(50)	YES		NULL	
account_type	varchar(50)	YES		NULL	
address	varchar(100)	YES		NULL	
monthly_income	decimal(10,2)	YES		NULL	
email	varchar(100)	YES		NULL	
mobile_number	varchar(20)	YES	MUL	NULL	
manager_id	int	YES		NULL	

**Decomposed Table**

Field	Type	Null	Key	Default	Extra
account_number	int	NO	PRI	NULL	
first_name	varchar(50)	YES		NULL	
last_name	varchar(50)	YES		NULL	
account_type	varchar(50)	YES		NULL	
monthly_income	decimal(10,2)	YES		NULL	
email	varchar(100)	YES		NULL	
mobile_number	varchar(20)	YES		NULL	
manager_id	int	YES	MUL	NULL	

### SECOND NORMAL FORM (2NF)

A relation is said to be in Second Normal Form, If and only if it is already in First Normal Form. There exist no Partial Dependency.

Transaction Table -- No rules violated

TransactionItems Table – ItemDescription is dependent on ItemID(violating 2NF)

Items Table – No rules violated

### Input Table

Field	Type	Null	Key	Default	Extra
account_number	int	NO	PRI	NULL	
first_name	varchar(50)	YES		NULL	
last_name	varchar(50)	YES		NULL	
account_type	varchar(50)	YES		NULL	
monthly_income	decimal(10,2)	YES		NULL	
email	varchar(100)	YES		NULL	
mobile_number	varchar(20)	YES		NULL	
manager_id	int	YES	MUL	NULL	

### Decomposed Table

Field	Type	Null	Key	Default	Extra
account_number	int	NO	PRI	NULL	
first_name	varchar(50)	YES		NULL	
last_name	varchar(50)	YES		NULL	
email	varchar(100)	YES		NULL	
mobile_number	varchar(20)	YES		NULL	
manager_id	int	YES	MUL	NULL	

Field	Type	Null	Key	Default	Extra
account_number	int	NO	PRI	NULL	
account_type	varchar(50)	YES		NULL	
monthly_income	decimal(10,2)	YES		NULL	

### THIRD NORMAL FORM (3NF)

It is in 3NF. It doesn't have any non-prime attribute that depends on another non-prime attribute.

Transaction Table -- No rules violated

TransactionItems Table – TransactionItems.ItemName and TransactionItems.ItemPriceCategory depend on Items.ItemID, which is not part of the primary key of TransactionItems. (violating 3NF)

Items Table – No rules violated



### Input Table

Field	Type	Null	Key	Default	Extra
investment_id	int	NO	PRI	NULL	
account_number	int	YES	MUL	NULL	
investment_type	varchar(50)	YES		NULL	
investment_amount	decimal(10,2)	YES		NULL	
start_date	date	YES		NULL	
end_date	date	YES		NULL	
time_period	varchar(15)	YES		NULL	

### Decomposed Table

Field	Type	Null	Key	Default	Extra
investment_id	int	NO	PRI	NULL	
account_number	int	YES	MUL	NULL	
investment_type	varchar(50)	YES		NULL	
investment_amount	decimal(10,2)	YES		NULL	

Field	Type	Null	Key	Default	Extra
investment_id	int	NO	PRI	NULL	
startdate	date	YES		NULL	
enddate	date	YES		NULL	
timeperiod	varchar(15)	YES		NULL	

#### Functional Dependency in customer table

Account\_number → first\_name Account\_number → last\_name

Account\_number → mobile\_number mobile\_number → email

email → first\_name Mobile\_number →

first\_name

Mobile\_number → last\_name

#### Functional Dependency in account table

Account\_number → first\_name

Account\_number → last\_name

Account\_number → mobile\_number

#### Functional Dependency in transaction table

Investment\_id → investment\_type

Investment\_id, investment\_type → start\_date

Start\_date → time\_period

time → end\_date

Account\_number → manager\_id

## **Importance of Each Normalization Step**

Normalization is crucial in database design to reduce redundancy and improve data integrity. Each step of normalization addresses specific types of anomalies:

1NF makes the database more adaptable to query by ensuring that every column is atomic.

2NF further increases the efficiency of the database by minimizing redundancy and dependency anomalies.

3NF ensures that all the data is logically stored, thus further increasing the coherence and efficiency of the database system.

## **Diagram and Explanation of Normalization Process**

The normalization process involves restructuring the database to minimize redundancy and dependency issues. Here is a conceptual diagram showing the normalization steps and their impact on functional dependencies:

1. Initial Table: Contains all data with visible redundancies and complex dependencies.
2. 1NF: Removes duplicate records, ensures atomicity.
3. 2NF: Removes partial dependencies, isolates data so each table contains data for a specific function.
4. 3NF: Removes transitive dependencies, ensuring all data directly depends on the primary key.
5. Final Structure: Shows how tables can be joined based on primary and foreign keys to recreate the original dataset without redundancy.

## **Implementation of concurrency control and recovery mechanisms**

Concurrency control and recovery mechanisms are crucial components of database management systems, especially in the context of a bank management system where data integrity, consistency, and reliability are paramount.

## 1. Concurrency Control

Concurrency control techniques ensure that multiple cardless transactions executed concurrently by users do not interfere with each other, preserving data consistency. Techniques include:

- **Locking:** Transactions acquire locks on relevant data items to prevent concurrent access. For instance, when processing a cardless withdrawal, the system locks the account balance to prevent simultaneous withdrawals that may lead to overdrawing.
- **Timestamping:** Transactions are assigned timestamps, and conflicts are resolved based on their timestamps. Older transactions may be prioritized over newer ones to maintain consistency.
- **Optimistic Concurrency Control:** Transactions proceed without acquiring locks initially. Conflicts are detected during transaction validation, and the transaction may be aborted and retried if conflicts arise.

## 2. Recovery Mechanisms:

Recovery mechanisms ensure that the database can recover to a consistent state after failures, such as system crashes or network disruptions. Techniques include:

**Write-Ahead Logging (WAL):** Transactional changes are logged before being applied to the database. In case of a failure, changes can be replayed from the log to restore the database to a consistent state.

**Checkpointing:** Periodic checkpoints capture the database state, including active transactions and modified data, providing a recovery point. This ensures that the system can recover to a consistent state post-failure.

**Shadow Paging:** Multiple versions of the database are maintained. When a transaction modifies data, a new version is created, and pointers are updated. If a failure occurs, the system can revert to the last consistent state by discarding changes made after the last checkpoint.

In a cardless transaction management system, these mechanisms ensure that data integrity, consistency, and reliability are upheld even during system failures or simultaneous transactions, safeguarding the integrity of financial data and transactions.

## 1. Concurrency Control:

- *Account Management:* in a cardless transaction system, concurrency control ensures that concurrent access and modification of account data, such as balance updates or account closures, do not conflict. Techniques like locking or timestamping are employed to serialize operations and maintain data consistency. For instance, if one transaction is updating a customer's contact information while another is processing a cardless payment from the same account, concurrency control mechanisms prevent data inconsistencies.

- *Transaction Management*: Multiple transactions may attempt to modify the same account concurrently in a cardless transaction system. Concurrency control mechanisms prevent scenarios like simultaneous fund transfers from the same account, ensuring correct balances. Techniques such as locking specific accounts during transactions or employing optimistic concurrency control help manage concurrency effectively, preserving transactional integrity.

- *Employee Management*: Concurrency control mechanisms ensure that conflicting changes to employee records, such as updating salary or job title, are managed appropriately. Locking mechanisms or timestamp-based concurrency control can be employed to handle concurrent access to employee data securely.

- *Customer Management*: Concurrent access to customer information, such as updating contact details or creating new accounts, requires concurrency control to prevent data inconsistencies. Techniques like locking customer records during updates or employing optimistic concurrency control help maintain accurate and consistent customer data.

## **2. Recovery Mechanisms:**

- *Account Management*: Recovery mechanisms, such as write-ahead logging (WAL), ensure that changes made to account balances during cardless transactions are durable. In the event of a system crash or failure, the system can recover and replay transactions from the log to restore account balances to a consistent state.

- *Transaction Management*: Recovery mechanisms play a crucial role in maintaining transactional integrity. Write-ahead logging ensures that committed transactions are durably stored, allowing the system to recover from failures and replay transactions to maintain data consistency. Techniques like check pointing provide recovery points to restore the system to a consistent state after a crash.

- *Employee Management*: Recovery mechanisms safeguard employee data against loss or corruption. For example, techniques like shadow paging maintain multiple versions of employee records, enabling the system to revert to a consistent state in case of failures.

- *Customer Management*: Recovery mechanisms ensure the durability of customer data modifications. By logging customer-related transactions and employing techniques like check pointing, the system can recover customer data to a consistent state after a failure.

- In summary, concurrency control and recovery mechanisms are essential components of a cardless transaction management system, ensuring data consistency, integrity, and durability across all aspects of the system.

## 7. Code for the project

```
from tkinter import *
from tkinter import messagebox
import mysql.connector

# Database connection details
db_config = {
    "host": "localhost",
    "user": "root",
    "password": "123456",
    "database": "cardless_transactions"
}

# Function to establish database connection
def connect_to_database():
    try:
        db_connection = mysql.connector.connect(**db_config)
        return db_connection
    except mysql.connector.Error as error:
        messagebox.showerror("Database Error", f"Error connecting to the database: {error}")
        return None

a=0

# Function to authenticate account holder login
def authenticate_login():
    account_number = account_number_entry.get()
    pin = pin_entry.get()
    global a
    a=account_number

    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()

        # Placeholder query to check account holder credentials
        query = "SELECT * FROM login WHERE account_number = %s AND pin = %s"
        cursor.execute(query, (account_number, pin))
        result = cursor.fetchone()
```

```

    if result:
        messagebox.showinfo("Login Successful", "Welcome, Account
Holder!")
        open_account_holder_window()
    else:
        messagebox.showerror("Login Failed", "Invalid account number or
PIN")

    cursor.close()
    db_connection.close()
    return account_number

# Function to open a new window for account holder actions
def open_account_holder_window():
    # Remove all widgets from the main window
    clear_main_screen()

    # Create account holder dashboard
    welcome_label = Label(root, text="Successful Login\nAccount Holder
Dashboard", font=("Arial", 20), fg="white",
        bg="#34495e")
    welcome_label.pack(pady=20)

    # Buttons for account holder actions
    personal_details_button = Button(root, text="Personal Details",
command=personal_details, font=("Arial", 14),
        bg="#3498db", fg="white", padx=10, pady=5, bd=2)
    account_details_button = Button(root, text="Account Details",
command=account_details, font=("Arial", 14),
        bg="#3498db", fg="white", padx=10, pady=5, bd=2)
    loan_status_button = Button(root, text="Loan Status", command=loan_status,
font=("Arial", 14), bg="#3498db",
        fg="white", padx=10, pady=5, bd=2)
    transfer_history_button = Button(root, text="Transfer History",
command=transfer_history, font=("Arial", 14),
        bg="#3498db", fg="white", padx=10, pady=5, bd=2)
    investment_button = Button(root, text="Investment", command=investment,
font=("Arial", 14), bg="#3498db",
        fg="white", padx=10, pady=5, bd=2)

```

```

    credit_score_button = Button(root, text="Credit Score",
command=credit_score, font=("Arial", 14), bg="#3498db",
                                fg="white", padx=10, pady=5, bd=2)
    logout_button = Button(root, text="Logout", command=show_main_screen,
font=("Arial", 14), bg="#c0392b", fg="white",
                                padx=10, pady=5, bd=2)

    personal_details_button.pack(pady=10)
    account_details_button.pack(pady=10)
    loan_status_button.pack(pady=10)
    transfer_history_button.pack(pady=10)
    investment_button.pack(pady=10)
    credit_score_button.pack(pady=10)
    logout_button.pack(pady=20)
# Placeholder functions for different actions
def personal_details():
    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()
        global a

        # Fetch personal details based on account_number
        query = "SELECT * FROM personal_details WHERE account_number =
%s"
        cursor.execute(query, (a,))
        result = cursor.fetchone()

        if result:
            messagebox.showinfo("Personal Details", f"Account Number:
{result[0]}\n"
                                f"First Name: {result[1]}\n"
                                f>Last Name: {result[2]}\n"
                                f"Account Type: {result[3]}\n"
                                f"Address: {result[4]}\n"
                                f"Monthly Income: {result[5]}\n"
                                f>Email: {result[6]}\n"
                                f"Mobile Number: {result[7]}")
        else:
            messagebox.showerror("Error", "Personal details not found.")

```

```

    cursor.close()
    db_connection.close()

def account_details():
    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()
        global a

        # Fetch account details based on account_number
        query = "SELECT * FROM account WHERE account_number = %s"
        cursor.execute(query, (a,))
        result = cursor.fetchone()

        if result:
            messagebox.showinfo("Account Details", f"Account Number:
{result[0]}\n"
                                     f"Balance: {result[1]}\n"
                                     f"Limit: {result[2]}\n"
                                     f"Credit Score: {result[3]}\n"
                                     f"Manager ID: {result[4]}")
        else:
            messagebox.showerror("Error", "Account details not found.")

    cursor.close()
    db_connection.close()

def loan_status():
    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()
        global a
        # Fetch loan details based on account_number
        query = "SELECT * FROM loan WHERE account_number = %s"
        cursor.execute(query, (a,))
        results = cursor.fetchall()

        if results:
            message = "Loan Details:\n"
            for result in results:

```



```

        message += f"Loan ID: {result[0]}, Monthly Income: {result[2]},
Loan Amount: {result[3]}, Collateral Description: {result[4]}, Manager ID:
{result[5]}\n"
        messagebox.showinfo("Loan Status", message)
    else:
        messagebox.showerror("Error", "No loan details found.")

    cursor.close()
    db_connection.close()

def transfer_history():
    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()
        global a
        # Fetch transfer history based on sender_id or receiver_id
        query = "SELECT * FROM transfer_history WHERE sender_id = %s OR
receiver_id = %s"
        cursor.execute(query, (a, a))
        results = cursor.fetchall()

        if results:
            message = "Transfer History:\n"
            for result in results:
                message += f"Transaction ID: {result[0]}, Sender ID: {result[1]},
Receiver ID: {result[2]}, Amount: {result[3]}, Date: {result[4]}, Manager ID:
{result[5]}\n"
            messagebox.showinfo("Transfer History", message)
        else:
            messagebox.showerror("Error", "No transfer history found.")

        cursor.close()
        db_connection.close()

def investment():
    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()
        global a
        # Fetch investment details based on account_number

```

```

query = "SELECT * FROM investment WHERE account_number = %s"
cursor.execute(query, (a,))
result = cursor.fetchone()

if result:
    messagebox.showinfo("Investment Details", f"Investment ID:
{result[0]}\n"
                                f"Investment Type: {result[2]}\n"
                                f"Investment Amount: {result[3]}\n"
                                f"Start Date: {result[4]}\n"
                                f"End Date: {result[5]}\n"
                                f"Manager ID: {result[6]}")
else:
    messagebox.showerror("Error", "Investment details not found.")

cursor.close()
db_connection.close()

def credit_score():
    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()
        global a
        # Fetch credit score details based on account_number
        query = "SELECT * FROM credit_score WHERE account_number = %s"
        cursor.execute(query, (a,))
        result = cursor.fetchone()

        if result:
            messagebox.showinfo("Credit Score", f"Current Loan ID: {result[1]}\n"
                                f"Loan Amount: {result[2]}\n"
                                f"Previous Loan Details: {result[3]}\n"
                                f"Current Credit Score: {result[4]}\n"
                                f"Manager ID: {result[5]}")
        else:
            messagebox.showerror("Error", "Credit score details not found.")

        cursor.close()
        db_connection.close()
def view_account_details():

```

```

# Get manager_id from the login page
manager_id = manager_id_entry.get()

db_connection = connect_to_database()
if db_connection:
    cursor = db_connection.cursor()

    # Fetch account details for the specific manager_id
    query = "SELECT * FROM account WHERE manager_id = %s"
    cursor.execute(query, (manager_id,))
    results = cursor.fetchall()

    if results:
        message = "Account Details:\n"
        for result in results:
            message += f"Account Number: {result[0]}, Balance: {result[1]},
Credit Limit: {result[2]}, Credit Score: {result[3]}\n"
        messagebox.showinfo("Account Details", message)
    else:
        messagebox.showerror("Error", "No account details found for this
manager.")

    cursor.close()
    db_connection.close()

def view_loan_details():
    # Get manager_id from the login page
    manager_id = manager_id_entry.get()

    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()

        # Fetch loan details for the specific manager_id
        query = "SELECT * FROM loan WHERE manager_id = %s"
        cursor.execute(query, (manager_id,))
        results = cursor.fetchall()

        if results:
            message = "Loan Details:\n"

```

```

        for result in results:
            message += f"Loan ID: {result[0]}, Monthly Income: {result[2]},
Loan Amount: {result[3]}, Collateral Description: {result[4]}\n"
            messagebox.showinfo("Loan Details", message)
        else:
            messagebox.showerror("Error", "No loan details found for this
manager.")

        cursor.close()
        db_connection.close()

def view_investment_details():
    # Get manager_id from the login page
    manager_id = manager_id_entry.get()

    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()

        # Fetch investment details for the specific manager_id
        query = "SELECT * FROM investment WHERE manager_id = %s"
        cursor.execute(query, (manager_id,))
        results = cursor.fetchall()

        if results:
            message = "Investment Details:\n"
            for result in results:
                message += f"Investment ID: {result[0]}, Investment Type:
{result[2]}, Investment Amount: {result[3]}, Start Date: {result[4]}, End Date:
{result[5]}\n"
            messagebox.showinfo("Investment Details", message)
        else:
            messagebox.showerror("Error", "No investment details found for this
manager.")

        cursor.close()
        db_connection.close()

def view_manager_branch_details():
    # Get manager_id from the login page

```

```

manager_id = manager_id_entry.get()

db_connection = connect_to_database()
if db_connection:
    cursor = db_connection.cursor()

    # Fetch manager branch details for the specific manager_id
    query = "SELECT * FROM manager_branch_details WHERE manager_id
= %s"
    cursor.execute(query, (manager_id,))
    results = cursor.fetchall()

    if results:
        message = "Manager Branch Details:\n"
        for result in results:
            message += f"Branch ID: {result[0]}, Branch Name: {result[1]},
Location: {result[2]}, Manager ID: {result[3]}\n"
        messagebox.showinfo("Manager Branch Details", message)
    else:
        messagebox.showerror("Error", "No manager branch details found for
this manager.")

    cursor.close()
    db_connection.close()

# Modify show_manager_login_screen function to include new buttons for
manager actions
def after_show_manager_login_screen():
    # Hide all other widgets
    hide_all_widgets()

    # Add new buttons for manager actions
    account_details_button = Button(root, text="Account Details",
command=view_account_details, font=("Arial", 14),
                                bg="#3498db", fg="white", padx=10, pady=5, bd=2)
    loan_details_button = Button(root, text="Loan Details",
command=view_loan_details, font=("Arial", 14),
                                bg="#3498db", fg="white", padx=10, pady=5, bd=2)
    investment_details_button = Button(root, text="Investment Details",
command=view_investment_details, font=("Arial", 14),

```

```

        bg="#3498db", fg="white", padx=10, pady=5, bd=2)
    manager_branch_details_button = Button(root, text="Manager Branch
Details", command=view_manager_branch_details, font=("Arial", 14),
        bg="#3498db", fg="white", padx=10, pady=5, bd=2)
    logout_button = Button(root, text="Logout", command=show_main_screen,
font=("Arial", 14), bg="#c0392b", fg="white",
        padx=10, pady=5, bd=2)

```

```

account_details_button.pack(pady=10)
loan_details_button.pack(pady=10)
investment_details_button.pack(pady=10)
manager_branch_details_button.pack(pady=10)
logout_button.pack(pady=20)

```

# Function to authenticate manager login

```
def authenticate_manager_login():
```

```

    manager_id = manager_id_entry.get()
    code = code_entry.get()

```

```
db_connection = connect_to_database()
```

```
if db_connection:
```

```
    cursor = db_connection.cursor()
```

# Placeholder query to check manager credentials

```

    query = "SELECT * FROM manager WHERE manager_id = %s AND
code = %s"

```

```
    cursor.execute(query, (manager_id, code))
```

```
    result = cursor.fetchone()
```

```
if result:
```

```
    messagebox.showinfo("Login Successful", "Welcome, Manager!")
```

```
    after_show_manager_login_screen()
```

# Code to navigate to manager dashboard or perform actions

```
else:
```

```
    messagebox.showerror("Login Failed", "Invalid manager ID or code")
```

```
cursor.close()
```

```
db_connection.close()
```

# Function to insert account details into the database

```
import random
```

```
import random
```

```
def insert_account_details():
    # Retrieve entered account details
    first_name = first_name_entry.get()
    last_name = last_name_entry.get()
    account_type = account_type_entry.get()
    address = address_entry.get()
    monthly_income = float(monthly_income_entry.get()) # Convert to float
    email = email_entry.get()
    mobile_number = mobile_number_entry.get()
    balance = float(balance_label_entry.get()) # Convert to float

    db_connection = connect_to_database()
    if db_connection:
        cursor = db_connection.cursor()

        # Generate a random account number and check uniqueness

        account_number = None
        while True:
            account_number = random.randint(1, 1000)
            # Check if the generated account number is not already in use
            cursor.execute("SELECT * FROM personal_details WHERE
account_number = %s", (account_number,))
            if not cursor.fetchone(): # If no existing record found with this account
number
                break # Exit the loop

        # Select a random manager_id from the manager table
        cursor.execute("SELECT manager_id FROM manager")
        manager_ids = [row[0] for row in cursor.fetchall()]
        manager_id = random.choice(manager_ids)

        # Insert data into personal_details table
        insert_personal_query = "INSERT INTO personal_details
(account_number, first_name, last_name, account_type, address,
```

```
monthly_income, email, mobile_number, manager_id) VALUES (%s, %s, %s, %s, %s, %s, %s, %s)"
```

```
    personal_data = (account_number, first_name, last_name, account_type, address, monthly_income, email, mobile_number, manager_id)
```

```
    try:
        cursor.execute(insert_personal_query, personal_data)
        db_connection.commit()
        messagebox.showinfo("Success", "Account details submitted successfully!")
    except mysql.connector.Error as error:
        db_connection.rollback()
        messagebox.showerror("Database Error", f"Failed to insert account details: {error}")
```

```
# Calculate credit limit (35% of balance)
```

```
credit_limit = balance * 0.35
```

```
credit_limit = round(credit_limit, 2)
```

```
# Insert data into account table
```

```
credit_score=0
```

```
insert_account_query = "INSERT INTO account (account_number, balance, `limit`, credit_score, manager_id) VALUES (%s, %s, %s, %s, %s)"
```

```
    account_data = (account_number, balance, credit_limit, credit_score, manager_id) # Assuming credit_score is set to 0 initially
```

```
    try:
        cursor.execute(insert_account_query, account_data)
        db_connection.commit()
        messagebox.showinfo("Success", "Account details submitted successfully!")
    except mysql.connector.Error as error:
        db_connection.rollback()
        messagebox.showerror("Database Error", f"Failed to insert account details: {error}")
```

```
    cursor.close()
```

```
    db_connection.close()
```



```

# Function to show the main screen with login options
def show_main_screen():
    # Hide all other widgets
    hide_all_widgets()

    # Show main login widgets
    account_number_label.pack()
    account_number_entry.pack()
    pin_label.pack()
    pin_entry.pack()
    login_button.pack(pady=20)
    manager_login_button.pack(pady=10)
    create_account_button.pack(pady=10)
    exit_button.pack(side=LEFT, padx=10, pady=20)

# Function to show the manager login screen
def show_manager_login_screen():
    # Hide all other widgets
    hide_all_widgets()

    # Show manager login widgets
    manager_id_label.pack(pady=10)
    manager_id_entry.pack(pady=10)
    code_label.pack(pady=10)
    code_entry.pack(pady=10)
    global login_manager_button # Define the button as global
    login_manager_button = Button(root, text="Login",
    command=authenticate_manager_login, font=("Arial", 14), bg="#3498db",
    fg="white", padx=10, pady=5, bd=2)
    login_manager_button.pack(pady=10)
    back_button.pack(pady=10)

# Function to hide all widgets
def hide_all_widgets():
    for widget in root.winfo_children():
        widget.pack_forget()
def clear_main_screen():
    for widget in root.winfo_children():
        widget.pack_forget()

```

```

# Function to handle "Create Account" button click
def create_account():
    # Hide all other widgets
    hide_all_widgets()
    welcome_label = Label(root, text="Enter the following details",
font=("Arial", 20), fg="white", bg="#34495e")
    welcome_label.pack(pady=0.5)

    # Show account creation widgets
    first_name_label.pack(pady=5)
    first_name_entry.pack(pady=5)
    last_name_label.pack(pady=5)
    last_name_entry.pack(pady=5)
    account_type_label.pack(pady=5)
    account_type_entry.pack(pady=5)
    address_label.pack(pady=5)
    address_entry.pack(pady=5)
    monthly_income_label.pack(pady=5)
    monthly_income_entry.pack(pady=5)
    email_label.pack(pady=5)
    email_entry.pack(pady=5)
    mobile_number_label.pack(pady=5)
    mobile_number_entry.pack(pady=5)
    balance_label.pack(pady=5)
    balance_label_entry.pack(pady=5)
    submit_button.pack(pady=20)
    back_button.pack(pady=10)

# Function to handle "Exit" button click
def exit_application():
    if messagebox.askokcancel("Exit Application", "Do you want to exit?"):
        root.destroy()

# Main GUI window
root = Tk()
root.title("Cardless Transactions Management System")
root.geometry("800x600")
root.configure(bg="#34495e")

# Heading Label

```

```
heading_label = Label(root, text="Cardless Transactions Management System",
font=("Arial", 24), fg="white", bg="#34495e")
heading_label.pack(pady=20)
```

```
# Account Number Entry
```

```
account_number_label = Label(root, text="Account Number:", font=("Arial",
14), fg="white", bg="#34495e")
account_number_entry = Entry(root, font=("Arial", 14), bd=2)
```

```
# PIN Entry
```

```
pin_label = Label(root, text="PIN:", font=("Arial", 14), fg="white",
bg="#34495e")
pin_entry = Entry(root, show="*", font=("Arial", 14), bd=2)
```

```
# Manager Login Entry
```

```
manager_id_label = Label(root, text="Manager ID:", font=("Arial", 14),
fg="white", bg="#34495e")
manager_id_entry = Entry(root, font=("Arial", 14), bd=2)
code_label = Label(root, text="Code:", font=("Arial", 14), fg="white",
bg="#34495e")
code_entry = Entry(root, show="*", font=("Arial", 14), bd=2)
```

```
# Create Account Entry
```

```
first_name_label = Label(root, text="First Name:", font=("Arial", 14),
fg="white", bg="#34495e")
first_name_entry = Entry(root, font=("Arial", 14), bd=2)
last_name_label = Label(root, text="Last Name:", font=("Arial", 14),
fg="white", bg="#34495e")
last_name_entry = Entry(root, font=("Arial", 14), bd=2)
account_type_label = Label(root, text="Account Type:", font=("Arial", 14),
fg="white", bg="#34495e")
account_type_entry = Entry(root, font=("Arial", 14), bd=2)
address_label = Label(root, text="Address:", font=("Arial", 14), fg="white",
bg="#34495e")
address_entry = Entry(root, font=("Arial", 14), bd=2)
monthly_income_label = Label(root, text="Monthly Income:", font=("Arial",
14), fg="white", bg="#34495e")
monthly_income_entry = Entry(root, font=("Arial", 14), bd=2)
email_label = Label(root, text="Email:", font=("Arial", 14), fg="white",
bg="#34495e")
```

```

email_entry = Entry(root, font=("Arial", 14), bd=2)
mobile_number_label = Label(root, text="Mobile Number:", font=("Arial", 14),
fg="white", bg="#34495e")
mobile_number_entry = Entry(root, font=("Arial", 14), bd=2)
balance_label = Label(root, text="Balance:", font=("Arial", 14), fg="white",
bg="#34495e")
balance_label_entry = Entry(root, font=("Arial", 14), bd=2)

# Login Button
login_button = Button(root, text="Login", command=authenticate_login,
font=("Arial", 14), bg="#3498db", fg="white", padx=10, pady=5, bd=2)

# Manager Login Button
manager_login_button = Button(root, text="Manager Login",
command=show_manager_login_screen, font=("Arial", 14), bg="#3498db",
fg="white", padx=10, pady=5, bd=2)

# Create Account Button
create_account_button = Button(root, text="Create Account",
command=create_account, font=("Arial", 14), bg="#2ecc71", fg="white",
padx=10, pady=5, bd=2)

# Submit Button
submit_button = Button(root, text="Submit", command=insert_account_details,
font=("Arial", 14), bg="#2ecc71", fg="white", padx=10, pady=5, bd=2)

# Back Button
back_button = Button(root, text="Back", command=show_main_screen,
font=("Arial", 14), bg="#c0392b", fg="white", padx=10, pady=5, bd=2)

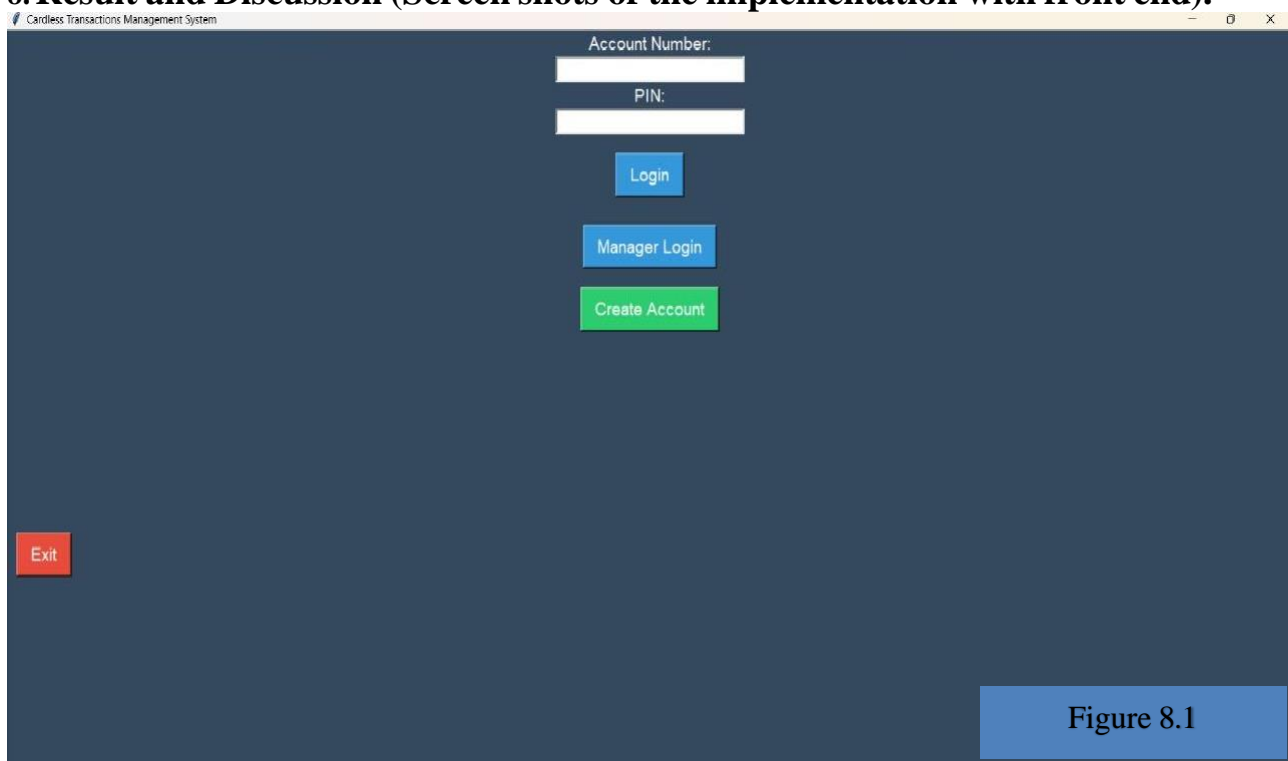
# Exit Button
exit_button = Button(root, text="Exit", command=exit_application,
font=("Arial", 14), bg="#e74c3c", fg="white", padx=10, pady=5, bd=2)

# Show main login widgets initially
show_main_screen()

# Run main window
root.mainloop()

```

## 8. Result and Discussion (Screen shots of the implementation with front end).



Cardless Transactions Management System

Account Number:

PIN:

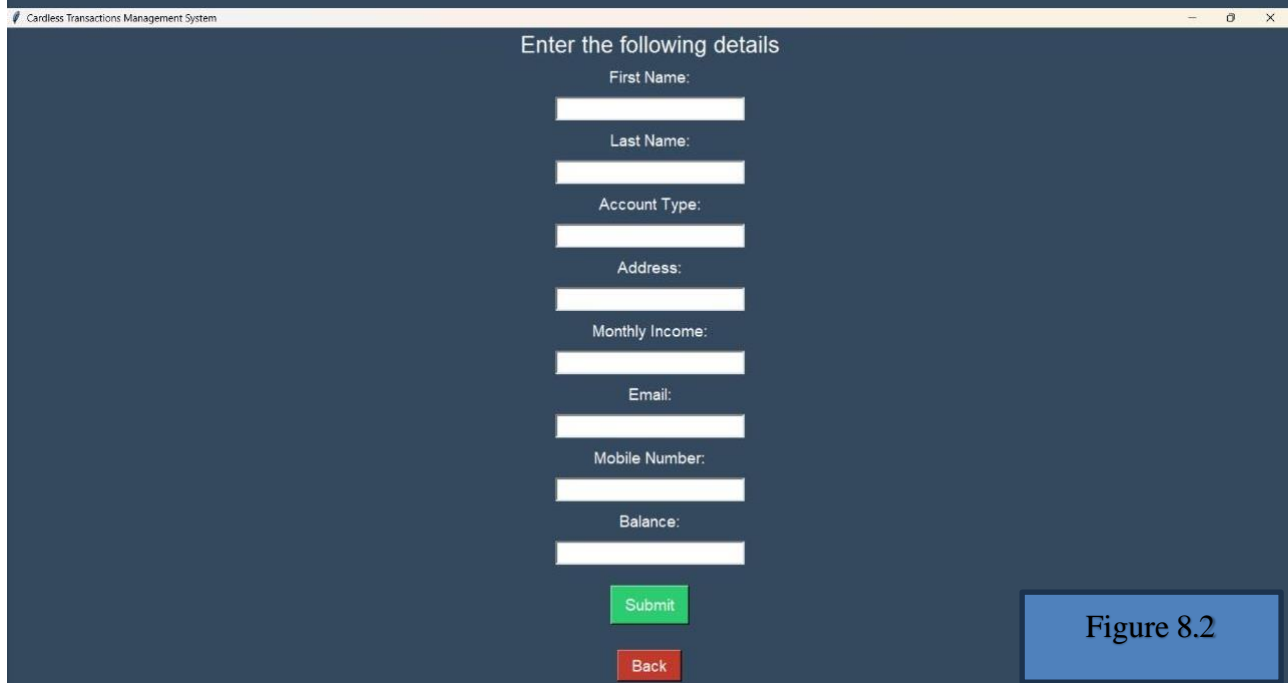
Login

Manager Login

Create Account

Exit

Figure 8.1



Cardless Transactions Management System

Enter the following details

First Name:

Last Name:

Account Type:

Address:

Monthly Income:

Email:

Mobile Number:

Balance:

Submit

Back

Figure 8.2

Cardless Transactions Management System

Enter the following details

First Name:  
Nikhil

Last Name:  
Sharma

Account Type:  
Current

Address:  
PF

Success  
Account details submitted successfully!  
OK

Mobile Number:  
9898989898

Balance:  
1000456

Submit

Back

Figure 8.3

Cardless Transactions Management System

Enter the following details

First Name:  
Nikhil

Last Name:  
Sharma

Account Type:  
Current

Address:  
PF

Monthly Income:  
500000

Email:  
nikhilsharma@gmail.com

Mobile Number:  
9898989898

Balance:  
1000456

Submit

Back

Figure 8.4

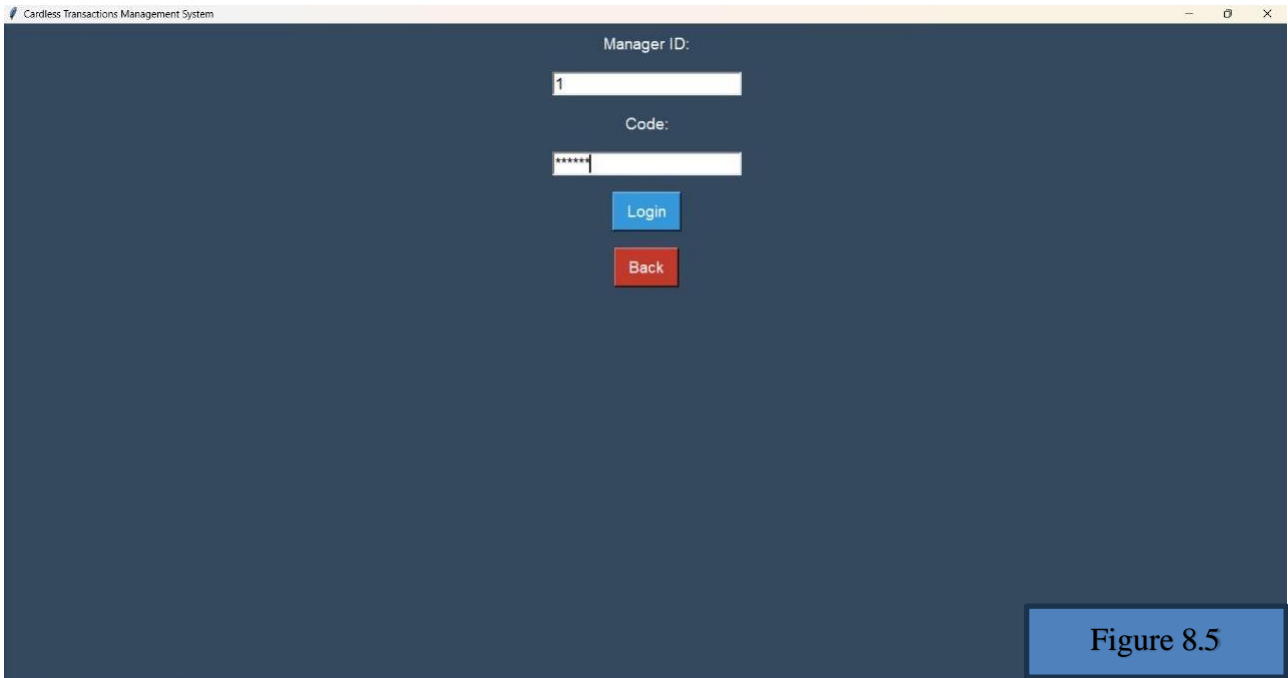


Figure 8.5

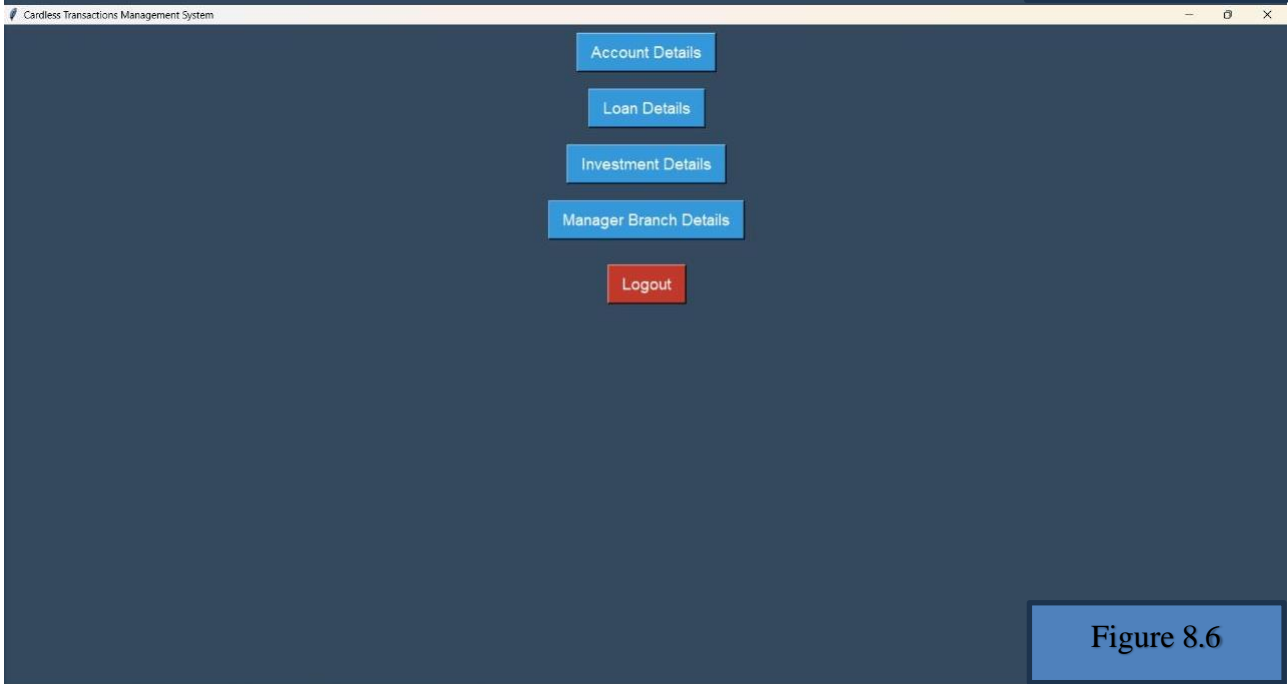


Figure 8.6

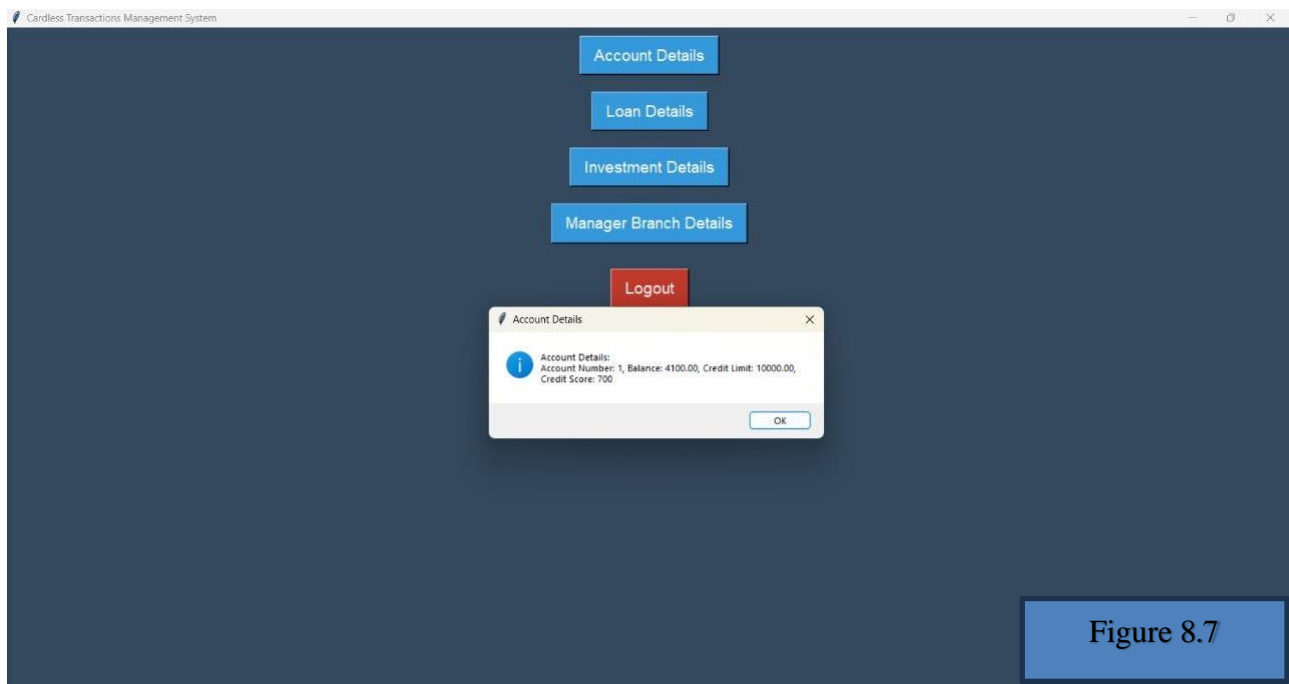


Figure 8.7

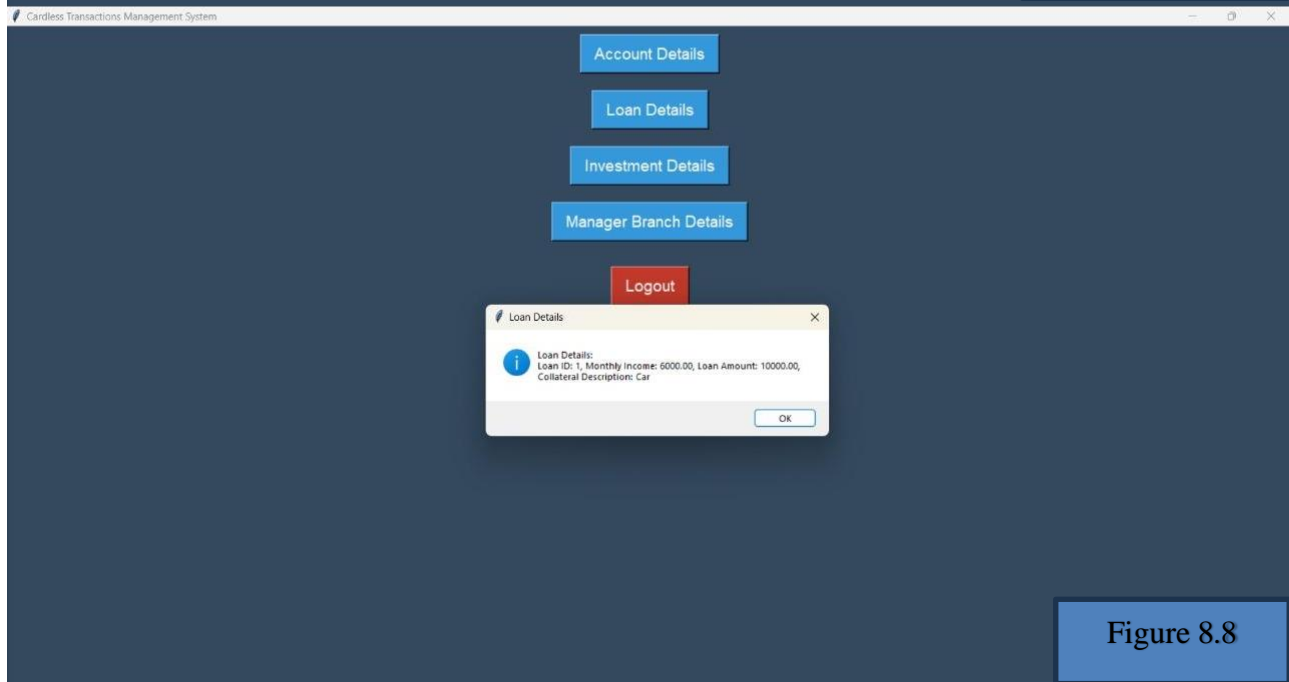


Figure 8.8



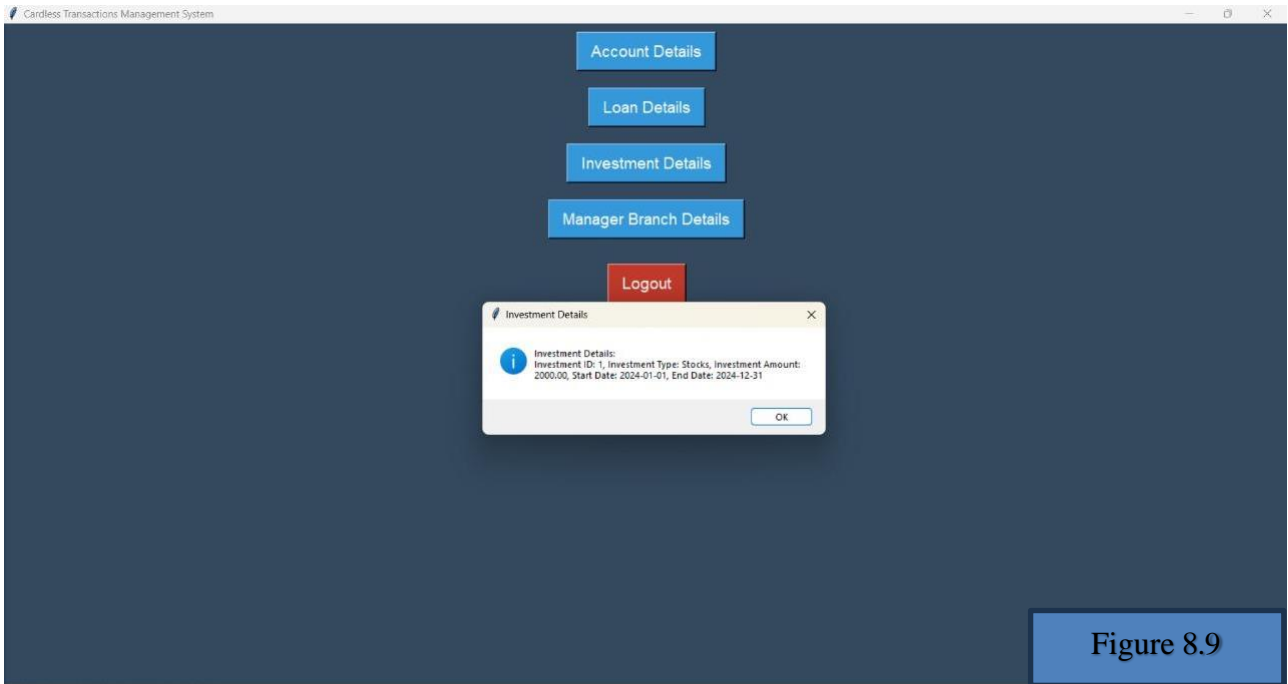


Figure 8.9

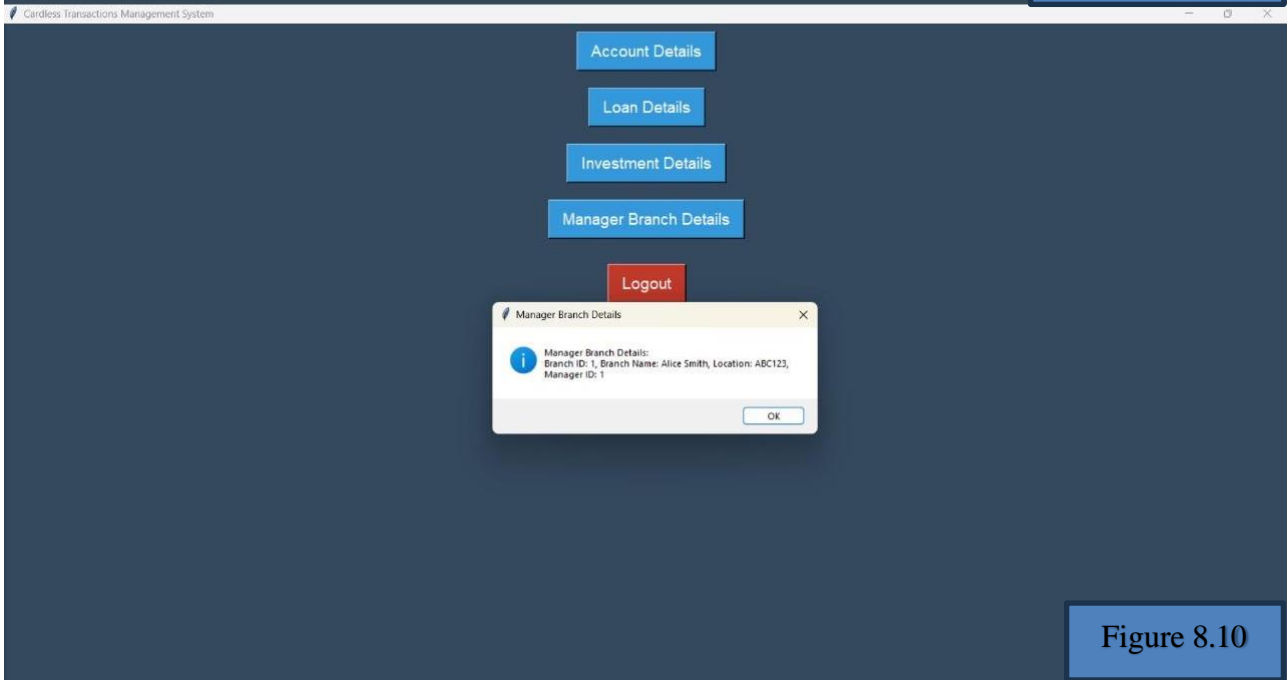
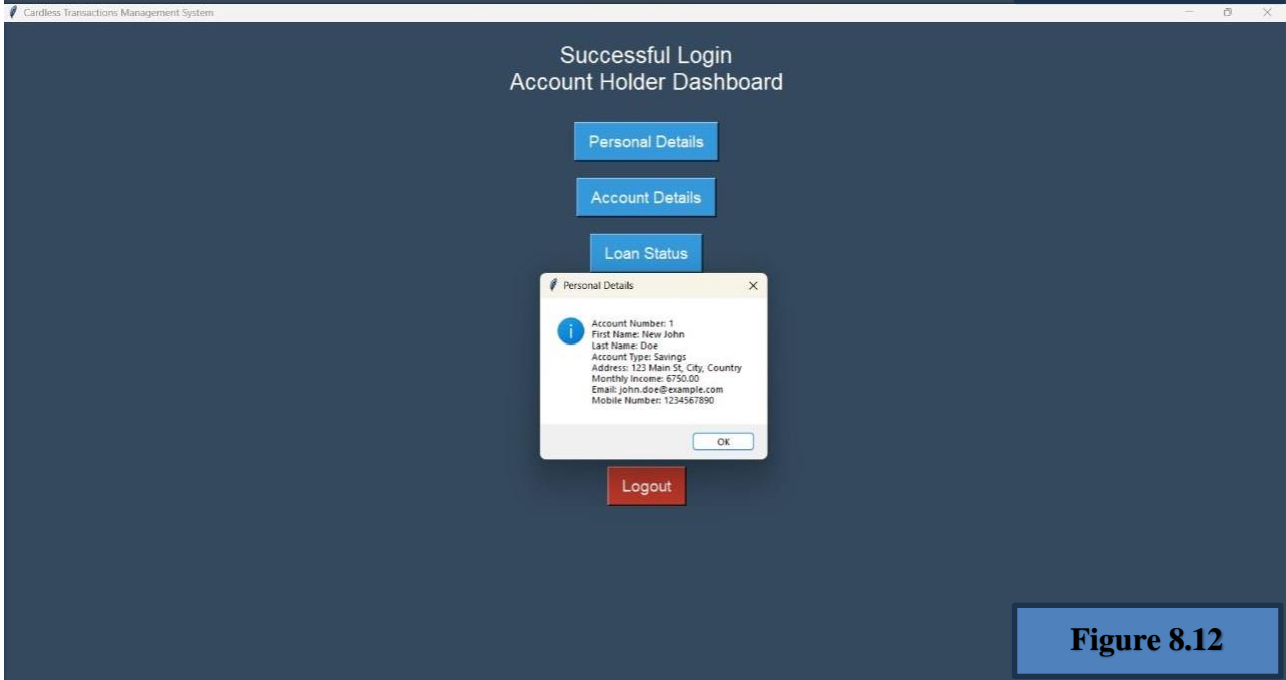
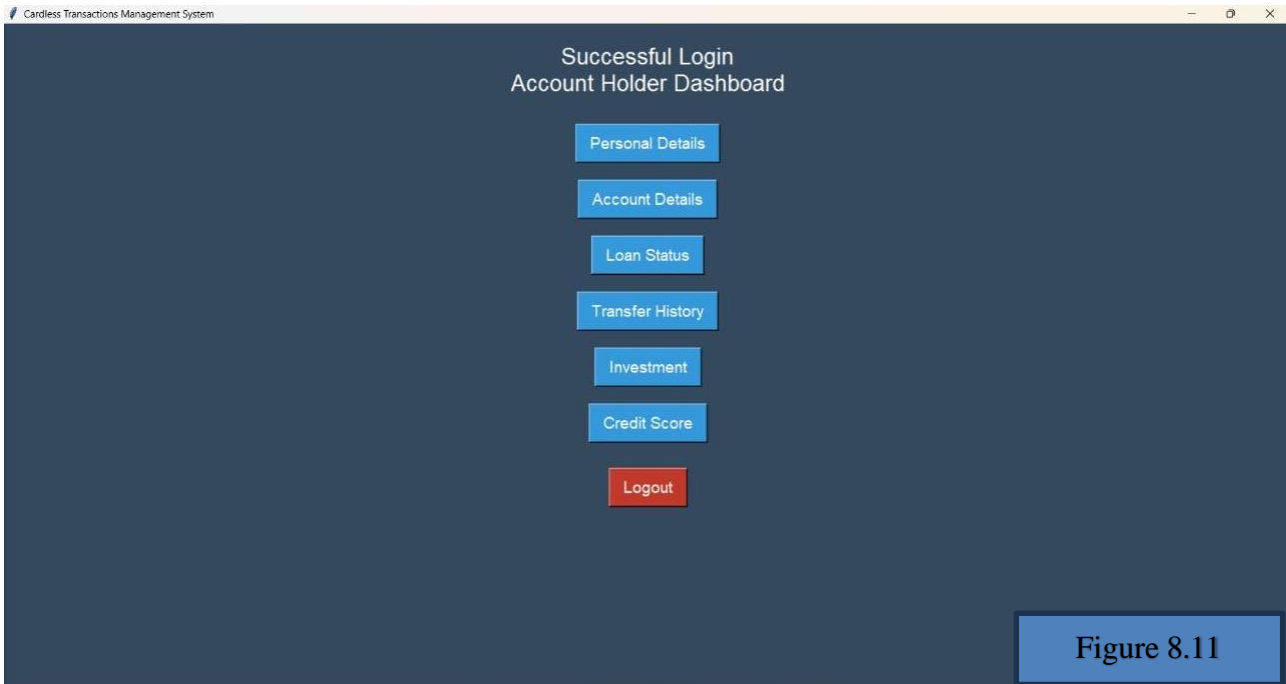
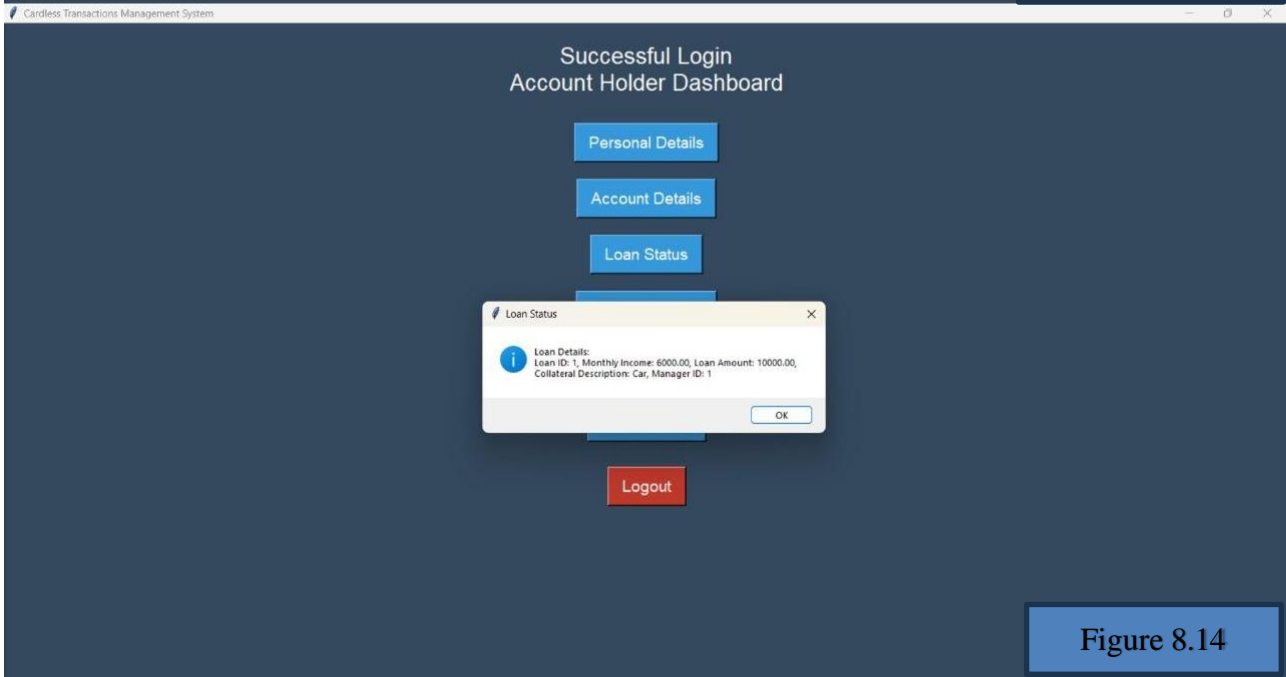
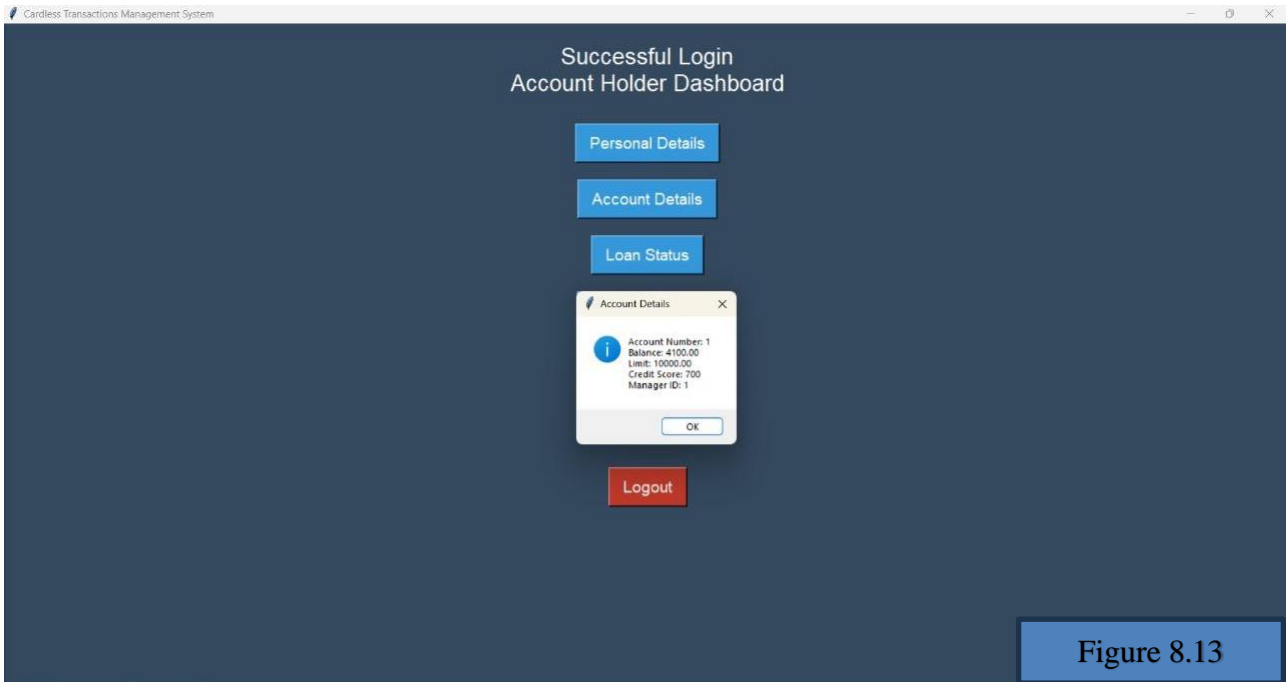


Figure 8.10





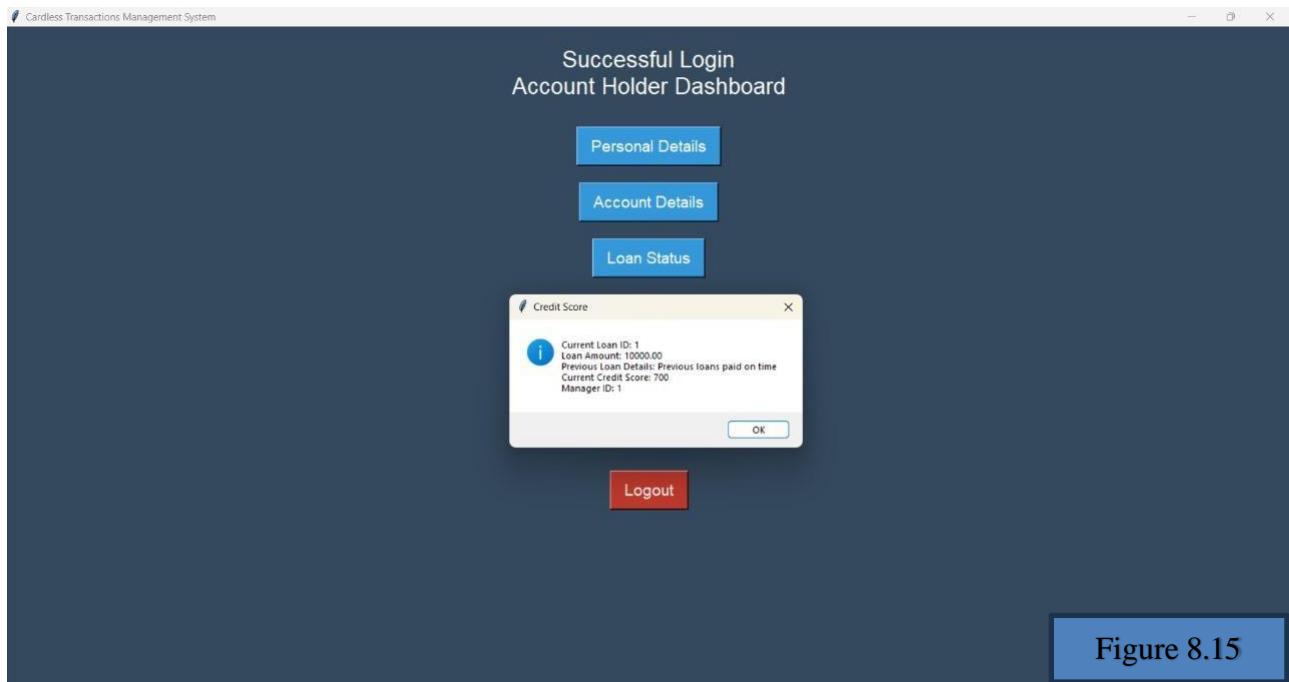


Figure 8.15

## Online course certificate



Elite

# NPTEL Online Certification

(Funded by the MoE, Govt. of India)



This certificate is awarded to

**NIKHIL SHARMA**

for successfully completing the course

## Data Base Management System

with a consolidated score of **61** %

Online Assignments	22.5/25	Proctored Exam	38.25/75
--------------------	---------	----------------	----------

Total number of candidates certified in this course: **6225**

**Jan-Mar 2024**

(8 week course)

**Prof. Haimanti Banerji**  
Coordinator, NPTEL  
IIT Kharagpur



Indian Institute of Technology Kharagpur



Roll No: NPTEL24CS21S544102389

To verify the certificate



No. of credits recommended: 2 or 3



# NPTEL Online Certification

(Funded by the MoE, Govt. of India)



This certificate is awarded to

**YASH SRIVASTAVA**

for successfully completing the course

## Data Base Management System

with a consolidated score of **49** %

Online Assignments	18.75/25	Proctored Exam	30/75
--------------------	----------	----------------	-------

Total number of candidates certified in this course: **6225**

**Jan-Mar 2024**

(8 week course)

**Prof. Haimanti Banerji**  
Coordinator, NPTEL  
IIT Kharagpur



Indian Institute of Technology Kharagpur



Roll No: NPTEL24CS21S644102689

To verify the certificate



No. of credits recommended: 2 or 3

# CERTIFICATE OF EXCELLENCE

THIS CERTIFICATE IS AWARDED TO

SCALER  
Topics

## Luvish Arora

In recognition of the completion of the tutorial: **DBMS Course - Master the Fundamentals and Advanced Concepts**

Following are the the learning items, which are covered in this tutorial

▶ 74 Video Tutorials   ▶ 16 Modules   ▶ 16 Challenges

08 March 2024



Anshuman Singh

Co-founder **SCALER** 



## **Conclusion**

In conclusion, the Cardless Transaction Management System plays a pivotal role in modern banking operations, revolutionizing processes, enhancing security, and elevating customer service standards. Through modules such as Employee Management, Account Management, and Transaction Processing, the system enables efficient management of employee access, account administration, and cardless transaction activities. By integrating robust authentication and authorization mechanisms, the system ensures secure access to sensitive data and functionalities, mitigating risks associated with unauthorized access and potential fraud. Moreover, the system's ability to accurately record and track cardless transactions provides valuable insights for decision-making and regulatory compliance. Overall, the Cardless Transaction Management System represents a crucial asset for banks and financial institutions, empowering them to deliver seamless services while upholding the highest standards of security and efficiency in today's dynamic banking landscape.

## References

### 1. Books:

- "SQL for Beginners: Learn the Structured Query Language for the Most Popular Databases including Microsoft SQL Server, MySQL, MariaDB, PostgreSQL, and Oracle" by Preston Prescott.
- "Database Management Systems" by Raghu Ramakrishnan and Johannes Gehrke.
- "Fundamentals of Database Systems" by Ramez Elmasri and Shamkant B. Navathe.

### 2. Online Courses and Tutorials:

- Coursera courses like "SQL for Data Science" by University of California, Davis, and "Managing Big Data with MySQL" by Duke University.
- Udemy courses such as "The Complete SQL Bootcamp 2021: Go from Zero to Hero" by Jose Portilla.
- W3Schools (<https://www.w3schools.com/sql/>) and SQLZoo (<https://sqlzoo.net/>) offer interactive tutorials and exercises on SQL.

### 3. Academic Papers and Journals:

- Academic databases like IEEE Xplore, ACM Digital Library, and Google Scholar for research papers related to SQL in the context of bank management systems.
- Journals such as the "Journal of Database Management" and "ACM Transactions on Database Systems" for articles on database management topics.

### 4. Online Resources:

- GitHub repositories
- SQL forums like Stack Overflow provide insights into common SQL queries and challenges faced in database management projects.

### 5. Official Documentation:

- Official documentation for SQL databases like MySQL (<https://dev.mysql.com/doc/>) or PostgreSQL (<https://www.postgresql.org/docs/>) for detailed information on SQL syntax, data types, and database management.