ABSTRACT

This project is focused on designing and implementing a Database Management System for an ATM machine. The purpose of the system is to manage the operations and transactions carried out by customers through the ATM machine. The system will store all relevant information about the customers, their accounts, transactions, and ATM machine operations.

The database will be designed to handle various types of transactions such as withdrawals, deposits, and balance inquiries. It will also keep track of account balances and generate reports on transaction history. The system will utilize a user-friendly interface that will ensure that the transactions are fast, secure, and reliable. The system will be designed to ensure security by providing appropriate access controls and authentication protocols. The system will be designed to optimize performance to ensure that transactions are processed quickly and efficiently.

The ATM machine will be designed using a combination of hardware and software components. The hardware components will include a keypad, card reader, dispenser, and display screen. The software components will include the operating system, user interface, and transaction processing system. The project will also focus on implementing security measures such as authentication, authorization, and encryption to protect user data from unauthorized access.

The project will be implemented using a MySQL database. The user interface will be developed using Java Swing, and the database will be connected to the interface using JDBC. Overall, this project aims to provide a reliable and efficient DBMS for an ATM machine.

PROBLEM STATEMENT

The project entitled ATM system has a drastic change to that of the older version of banking system, customer feel inconvenient with the transaction method as it was in the hands of the bank employees. In our ATM system, the above problem is overcome here, the transactions are done in person by the customer thus makes the customers feel safe and secure. Thus, the application of our system helps the customer in checking the balance and transaction of the amount by validating the pin number therefore ATM system is more user friendly.

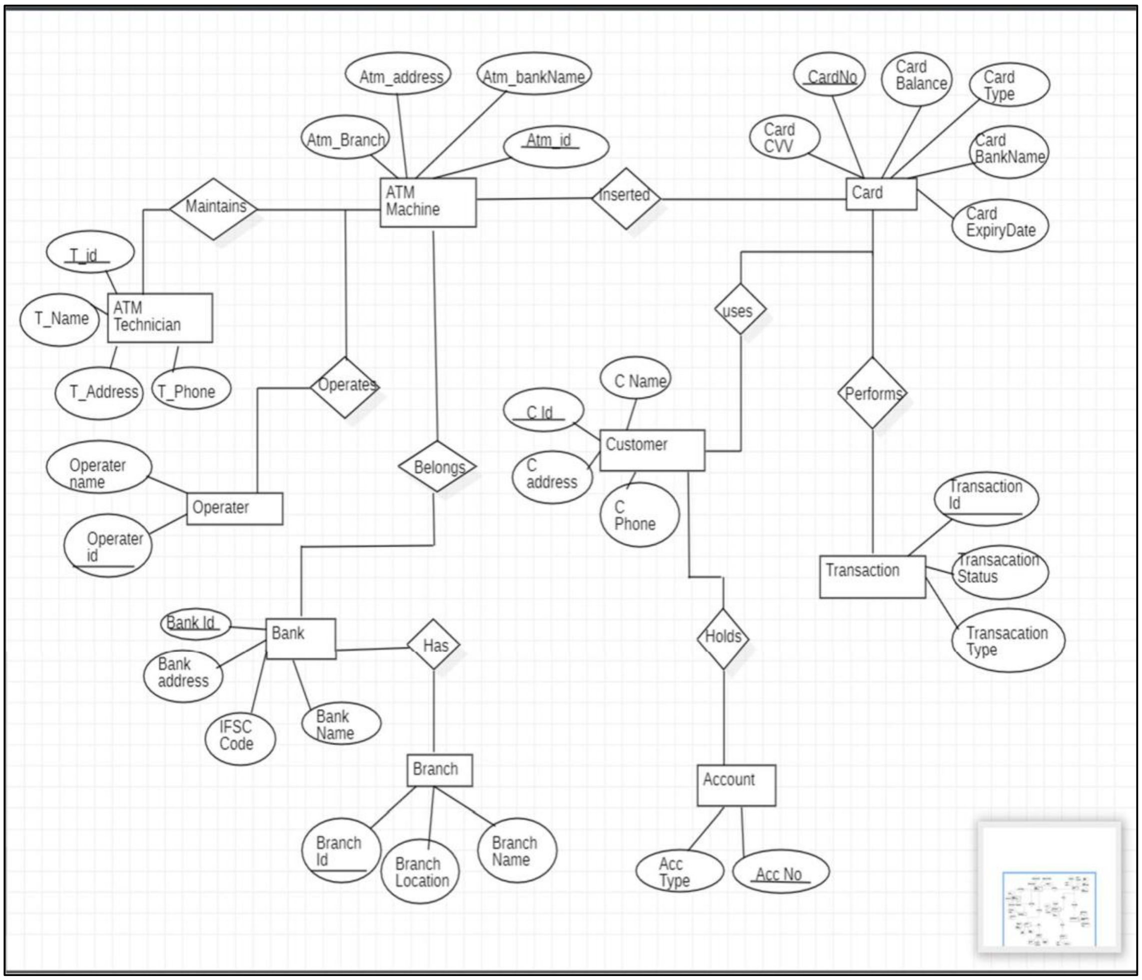
TABLE OF CONTENTS

1. PROBLEM UNDERSTANDING, IDENTIFICATION OF ENTITY AND RELATIONSHIPS, CONSTRUCTION OF DB USING ER MODEL FOR THE PROJECT

PROBLEM UNDERSTANDING:

Automated Teller Machine enables the clients of a bank to have access to their account without going to the bank. This is achieved only by developing the application using online concepts. When the product is implemented, the user who uses this product will be able to see all the information and services provided by the ATM, when he enters the necessary option and arguments. The product also provides services like request for cheques, deposit cash and other advanced requirements of the user. The data is stored in the database and is retrieved whenever necessary. The implementation needs ATM machine hardware to operate, or similar simulated conditions can also be used to successfully use the developed product.

ER DIAGRAM



Atm_address
Atm_bankName
Atm_Branch
Atm_id
ATM Machine
Maintains
Inserted
CardNo
Card Balance
Card Type
Card CVV
Card BankName
Card
Card ExpiryDate
T_id
T_Name
ATM Technician
T_Address
T_Phone
uses
Operates
Performs
C Name
C Id
Customer
C address
C Phone
Operater name
Operater
Operater id
Belongs
Transaction Id
Transacation Status
Transaction
Transacation Type
Bank Id
Bank
Has
Holds
Bank address
IFSC Code
Bank Name
Branch
Account
Branch Id
Branch Location
Branch Name
Acc Type
Acc No

2. DESIGN OF RELATIONAL SCHEMAS, CREATION OF DATABASE TABLES FOR THE PROJECT

RELATIONAL SCHEMA DIAGRAM

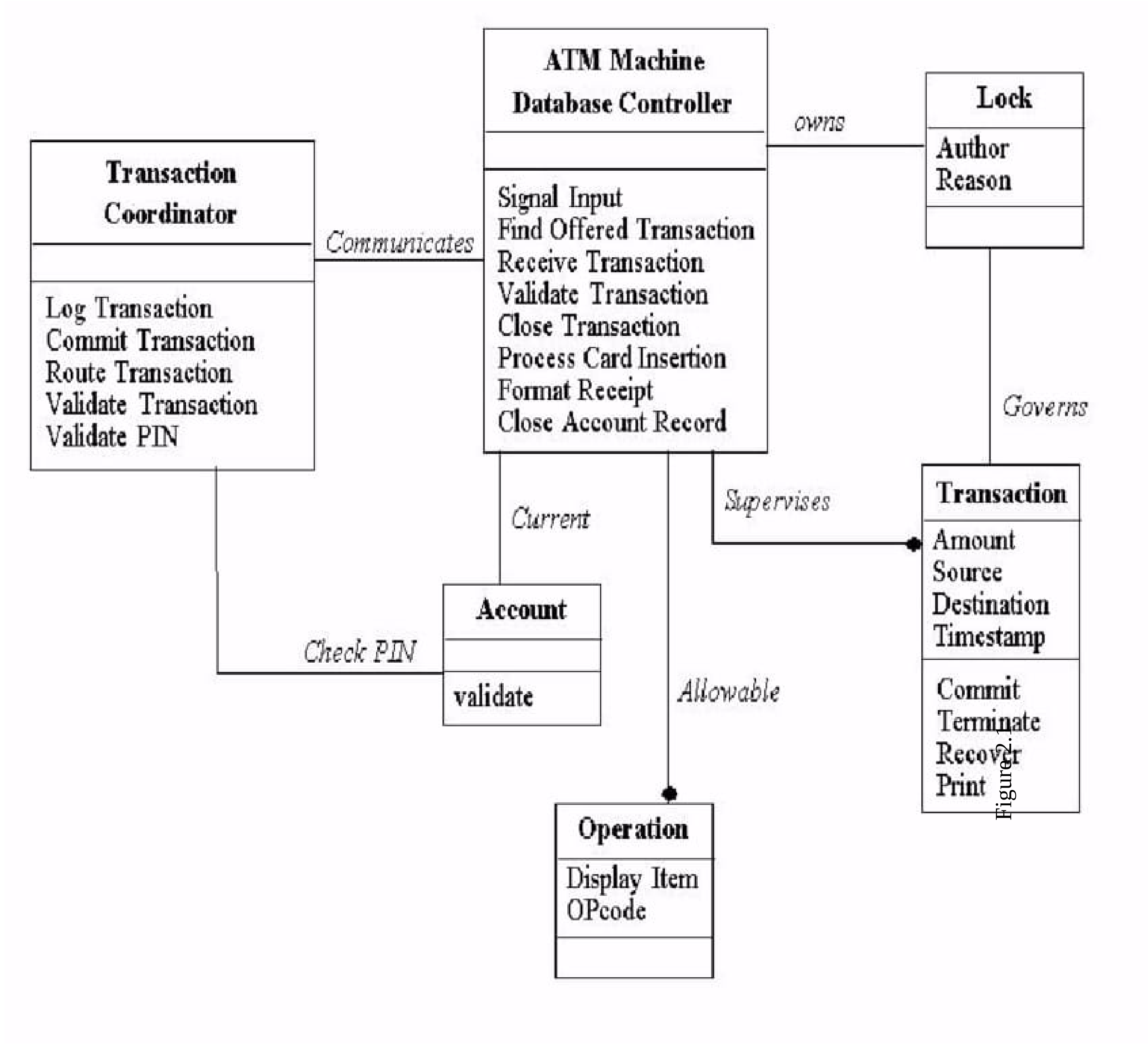


Figure 2.1

CREATION OF DATABASE TABLES

Creating table for Bank:

```
CREATE TABLE bank (
 bank_id INT NOT NULL PRIMARY KEY,
 bank_name VARCHAR (50),
 bank_location VARCHAR (50),
 bank_contact VARCHAR (20),
 bank_email VARCHAR (50)
);
--Insert values into Bank table
INSERT INTO bank (bank_id, bank_name, bank_location, bank_contact, bank_email)
VALUES
 (1, 'ABC Bank', 'New York', '+1-234-567-8901', 'info@abcbank.com'),
 (2, 'XYZ Bank', 'Los Angeles', '+1-987-654-3210', 'info@xyzbank.com'),
 (3, 'DEF Bank', 'Chicago', '+1-123-456-7890', 'info@defbank.com'),
 (4, 'GHI Bank', 'San Francisco', '+1-567-890-1234', 'info@ghibank.com'),
 (5, 'JKL Bank', 'Miami', '+1-890-123-5678', 'info@jklbank.com');
```

bank_id	bank_name	bank_location	bank_contact	bank_email
1	ABC Bank	New York	+1-234-567-8901	info@abcbank.com
2	XYZ Bank	Los Angeles	+1-987-654-3210	info@xyzbank.com
3	DEF Bank	Chicago	+1-123-456-7890	info@defbank.com
4	GHI Bank	San Francisco	+1-567-890-1234	info@ghibank.com
5	JKL Bank	Miami	+1-890-123-5678	info@jklbank.com

Creating table for ATM machine:

```
CREATE TABLE atm_machine (
 atm_id INT NOT NULL PRIMARY KEY,
 location VARCHAR(50),
 balance DECIMAL(10,2),
 bank_id INT,
 FOREIGN KEY (bank_id) REFERENCES bank(bank_id)
);
--Insert values into ATM Machine table
INSERT INTO atm_machine (atm_id, location, balance, bank_id)
VALUES
 (1, 'Times Square, New York', 50000.00, 1),
 (2, 'Beverly Hills, Los Angeles', 75000.00, 2),
 (3, 'Downtown, Chicago', 40000.00, 3),
 (4, 'Union Square, San Francisco', 30000.00, 4),
 (5, 'South Beach, Miami', 20000.00, 5);
```

atm_id	location	balance	bank_id
1	Times Square, New York	50000	1
2	Beverly Hills, Los Angeles	75000	2
3	Downtown, Chicago	40000	3
4	Union Square, San Francisco	30000	4
5	South Beach, Miami	20000	5

Creating table for Customer:

```
CREATE TABLE customer (
 customer_id INT NOT NULL PRIMARY KEY,
 first_name VARCHAR(50),
 last_name VARCHAR(50),
 email VARCHAR(50),
 phone_number VARCHAR(20),
 account_number VARCHAR(50),
 pin INT,
 bank_id INT,
 FOREIGN KEY (bank_id) REFERENCES bank(bank_id) );
--Insert values into Customer table
INSERT INTO customer (customer_id, first_name, last_name, email, phone_number,
account_number, pin, bank_id)
VALUES
 (1, 'John', 'Doe', 'john.doe@gmail.com', '+1-234-567-8901', '123456789', 1234, 1),
 (2, 'Jane', 'Smith', 'jane.smith@gmail.com', '+1-987-654-3210', '987654321', 4321, 2),
 (3, 'Alice', 'Johnson', 'alice.johnson@gmail.com', '+1-123-456-7890', '789456123', 5678, 3),
 (4, 'Bob', 'Williams', 'bob.williams@gmail.com', '+1-567-890-1234', '456789123', 8765, 4),
 (5, 'Charlie', 'Brown', 'charlie.brown@gmail.com', '+1-890-123-5678', '321987456', 2345,5);
```

customer_id	first_name	last_name	email	phone_number	account_num...	pin	bank_id
1	John	Doe	john.doe@gmail...	+1-234-567-8901	123456789	1234	1
2	Jane	Smith	jane.smith@gm...	+1-987-654-3210	987654321	4321	2
3	Alice	Johnson	alice.johnson@...	+1-123-456-7890	789456123	5678	3
4	Bob	Williams	bob.williams@g...	+1-567-890-1234	456789123	8765	4
5	Charlie	Brown	charlie.brown@...	+1-890-123-5678	321987456	2345	5

Create Card table:

```
CREATE TABLE card (
 card_number VARCHAR(50) NOT NULL PRIMARY KEY,
 customer_id INT,
 expiration_date DATE,
 cvv INT,
 card_type VARCHAR(50),
 bank_id INT,
 FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
 FOREIGN KEY (bank_id) REFERENCES bank(bank_id)
);
-- Insert values into Card table
INSERT INTO card (card_number, customer_id, expiration_date, cvv, card_type, bank_id)
VALUES
 ('1234567812345678', 1, '2024-10-31', 123, 'VISA', 1),
 ('2345678923456789', 2, '2023-06-30', 234, 'MasterCard', 2),
 ('3456789034567890', 3, '2022-09-30', 345, 'VISA', 3),
 ('4567890145678901', 4, '2023-12-31', 456, 'Discover', 4),
 ('5678901256789012', 5, '2024-05-31', 567, 'MasterCard', 5);
```

card_number	customer_id	expiration_date	cvv	card_type	bank_id
1234567812345678	1	2024-10-31	123	VISA	1
2345678923456789	2	2023-06-30	234	MasterCard	2
3456789034567890	3	2022-09-30	345	VISA	3
4567890145678901	4	2023-12-31	456	Discover	4
5678901256789012	5	2024-05-31	567	MasterCard	5

Create Transaction table:

```sql
CREATE TABLE transaction (
 transaction_id INT NOT NULL PRIMARY KEY,
 card_number VARCHAR(50),
 atm_id INT,
 transaction_date DATETIME,
 transaction_type VARCHAR(50),
 amount DECIMAL(10,2),
 balance DECIMAL(10,2),
 FOREIGN KEY (card_number) REFERENCES card(card_number),
 FOREIGN KEY (atm_id) REFERENCES atm_machine(atm_id)
);
-- Insert values into Transaction table
INSERT INTO transaction (transaction_id, card_number, atm_id, transaction_date,
transaction_type, amount, balance)
VALUES
 (1, '1234567812345678', 1, '2023-04-20 09:30:00', 'Withdrawal', 100.00, 4900.00),
 (2, '2345678923456789', 2, '2023-04-20 10:00:00', 'Deposit', 500.00, 75500.00),
 (3, '3456789034567890', 3, '2023-04-20 11:30:00', 'Withdrawal', 200.00, 3800.00),
 (4, '4567890145678901', 4, '2023-04-20 13:45:00', 'Balance Inquiry', 0.00, 30000.00),
 (5, '5678901256789012', 5, '2023-04-20 16:15:00', 'Withdrawal', 50.00, 19950.00);
```

transaction_id	card_number	atm_id	transaction_date	transaction_type	amount	balance
1	1234567812345678	1	2023-04-20 09:30:00	Withdrawal	100	4900
2	2345678923456789	2	2023-04-20 10:00:00	Deposit	500	75500
3	3456789034567890	3	2023-04-20 11:30:00	Withdrawal	200	3800
4	4567890145678901	4	2023-04-20 13:45:00	Balance Inquiry	0	30000
5	5678901256789012	5	2023-04-20 16:15:00	Withdrawal	50	19950

3.COMPLEX QUERIES BASED ON THE CONCEPTS OF CONSTRAINTS, SETS, JOINS, VIEWS, TRIGGERS AND CURSORS.

Constraints: Constraints are rules applied to table columns to ensure the integrity, accuracy, and reliability of the data stored in the table

```
ALTER TABLE accounts
ADD CONSTRAINT chk_balance CHECK (balance >= 0);  -- Ensures balance cannot be negati

ALTER TABLE accounts
ADD CONSTRAINT unq_email UNIQUE (email);  -- Ensures email addresses are unique.
```

Sets- :

The SETS operator in SQL combines the results of two or more SELECT statements into a single result set, excluding duplicate rows

```
-- Users who have deposited more than $500
CREATE VIEW Depositors AS
SELECT account_id
FROM transactions
WHERE amount > 500;

-- Users who have withdrawn more than $500
CREATE VIEW Withdrawers AS
SELECT account_id
FROM transactions
WHERE amount < -500;

-- Using UNION and a LEFT JOIN to find users who are only in Depositors and not in Wi
SELECT a.username
FROM accounts a
JOIN Depositors d ON a.id = d.account_id
LEFT JOIN Withdrawers w ON a.id = w.account_id
WHERE w.account_id IS NULL;
```

Joins:
 Joins in SQL combine data from two or more tables based on a related column between them, producing a single result set

```
SELECT a.username, t.transaction_id, t.amount
FROM accounts a
JOIN transactions t ON a.id = t.account_id
WHERE t.amount > 1.5 * (
    SELECT AVG(amount)
    FROM transactions
    WHERE account_id = a.id
);
```

Views:
A view in SQL is a virtual table generated by a stored query, allowing users to retrieve and manipulate data as if it were a regular table

```
CREATE VIEW AccountSummaries AS
SELECT a.id, a.username, a.balance, t.transaction_id, t.amount, t.transaction_date
FROM accounts a
JOIN transactions t ON a.id = t.account_id
ORDER BY t.transaction_date DESC;
```

Triggers:
Triggers in SQL are database objects that automatically execute in response to specified events on a table, enabling actions such as data validation, modification, or logging

```
DELIMITER $$

CREATE TRIGGER LogHighValueTransactions
AFTER INSERT ON transactions
FOR EACH ROW
BEGIN
    IF NEW.amount > 1000 THEN
        INSERT INTO transaction_log(account_id, log_message, log_time)
        VALUES (NEW.account_id, CONCAT('High value transaction of $', NEW.amount, '
    END IF;
END$$

DELIMITER ;
```

Cursors:

A cursor in SQL is a database object used to retrieve and iterate over a set of records returned by a SELECT statement, allowing sequential processing of individual rows

```
DELIMITER $$

CREATE PROCEDURE ListTransactions(IN acc_id INT)
BEGIN
    DECLARE done INT DEFAULT FALSE;
    DECLARE t_id INT;
    DECLARE t_amount DECIMAL(10,2);
    DECLARE cur CURSOR FOR SELECT transaction_id, amount FROM transactions WHERE acc
    DECLARE CONTINUE HANDLER FOR NOT FOUND SET done = TRUE;

    OPEN cur;

    FETCH cur INTO t_id, t_amount;
    WHILE NOT done DO
        SELECT CONCAT('Transaction ID: ', t_id, ' Amount: $', t_amount) AS Transacti
        FETCH cur INTO t_id, t_amount;
    END WHILE;

    CLOSE cur;
END$$

DELIMITER ;
```

4. ANALYZING THE PITFALLS, IDENTIFYING THE DEPENDENCIES, AND APPLYING NORMALIZATIONS

Normalization of Database:

Normalization is an important process in database design to ensure data integrity and reduce redundancy. In the case of an ATM (automated teller machine) system, normalization is essential to ensure that customer account information is safely stored and can be accessed efficiently.

There are several levels of normalization, with each level building upon the previous one. The most used levels are:

First Normal Form (1NF): This level requires that each table has a primary key and that all columns in the table are atomic (i.e., cannot be further divided).

In an ATM system, the account information can be stored in a single table with columns such as account number, account balance, customer name, and PIN. The account number can serve as the primary key for this table.

Second Normal Form (2NF): This level requires that all non-key attributes be dependent on the entire primary key.

For an ATM system, the account information table may need to be split into two tables to satisfy 2NF. One table could contain the account number and balance, while the other table could contain the customer’s name and PIN. This ensures that customer information is not duplicated and that changes to one record do not affect other records.

Third Normal Form (3NF): This level requires that all non-key attributes be not dependent on any other non-key attributes.

For an ATM system, the customer information table may need to be split further into two tables to satisfy 3NF. One table could contain the customer’s name and address, while the other table could contain the customer's PIN. This ensures that customer information is not duplicated and that changes to one record do not affect other records.

Normalization helps to minimize data redundancy and ensure that data is consistent and accurate. By following the normalization process, an ATM system can be designed to efficiently store data

Implementation using Dynamo DB:

DynamoDB is a NoSQL database service offered by AWS that is designed to provide fast and scalable performance. For an ATM system, DynamoDB can be used to store customer account information and transaction data.

1. **Define the data model**: Determine the data you need to store for the ATM system. For example, you may need to store information about Customer details like account number, name, account balance.

2. **Create a DynamoDB table**: Create a table in DynamoDB to store the data. Define the primary key for the table, which could be a single partition key or a combination of a partition key and a sort key.

3. **Define the table schema:** Define the attributes that you want to store in the table. You can also specify any secondary indexes that you want to create.

- Here is an implementation for an ATM system using DynamoDB:

Create a table for customer account information:

The table can have the following

attributes: Account Number

(partition key)

Account

Balance

Customer

Name PIN

Create a table for transaction history: The table can have the following attributes: Account Number (partition key) Timestamp (sort key)

Transaction Type (withdrawal, deposit, transfer) Amount Balance

❖ Create an IAM role with appropriate permissions for accessing DynamoDB

For account information: you can use the DynamoDB API to create, read, update, and delete records in the table. You can also use the Query operation to retrieve customer account information by account number.

For transaction history: you can use the Put Item operation to add a new transaction to the table, and the Query operation to retrieve transactions for a specific account number within a given time range.

Ensure data consistency and prevent race conditions, use conditional writes to update account balances when processing transactions. For example, when processing a withdrawal, you can use a conditional write to ensure that the account balance is sufficient before subtracting the amount from the balance. Use DynamoDB streams to capture changes to the transaction history table and trigger appropriate actions such as sending alerts or updating account balances in real-time.

Use DynamoDB's scalability features, such as auto-scaling and partitioning, to ensure that the system can handle increasing traffic and transaction volumes. In summary, DynamoDB can be a reliable and scalable choice for implementing an ATM system, with the flexibility to store and retrieve data quickly and efficiently and handle high levels of traffic and transaction volumes.

5. IMPLEMENTATION OF CONCURRENCY CONTROL AND RECOVERY MECHANISMS

Transactions:
We began a transaction before executing a sequence of SQL statements.
We committed the transaction to make changes permanent or rolled back to undo changes in case of errors.
Example:

```
--START TRANSACTION;

COMMIT;
--START TRANSACTION;

ROLLBACK;
```

Locking:
We used locks to control access to data and prevent concurrent transactions from interfering with each other.
We employed different types of locks, including shared locks (read locks) and exclusive locks (write locks).
Example:

```
-- Acquired a shared lock (read lock) on a table
LOCK TABLE Order READ;

-- Acquired an exclusive lock (write lock) on a table
LOCK TABLE Customer WRITE;

-- Released the locks
UNLOCK TABLES;
```

Logging:
We maintained a log of all database operations (inserts, updates, deletes) to facilitate recovery in case of system failures.
We used transaction logging to record before and after images of modified data.
Example:

```
-- Enabled logging for transactions
SET autocommit=0;
```

-- Inserted a row into the table
INSERT INTO adopter (user_id, name, address, DOB, income_range, phone_no,password)
VALUES ('johndoe', 'John Doe', '123 Main St, City, Country', '1990-01-01', 3,1234567890, 'password123');
-- Committed the transaction to log changesCOMMIT;

Recovery:
We used transaction logs to recover the database to a consistent state after asystem failure.
We rolled forward by replaying committed transactions from the log and rolledback uncommitted transactions.
Example:
-- Rolled forward to recover committed transactionsRECOVER;

-- Rolled back uncommitted transactions
ROLLBACK;

By using transactions, locks, logging, and recovery mechanisms effectively, we ensured data consistency, integrity, and durability in our database system, even in the presence of concurrent transactions and system failures.

Implementation of concurrency control and recovery mechanisms

Implementing effective concurrency control and recovery mechanisms is crucial for ensuring the integrity and reliability of your system. These mechanisms are essential to handle multiple users accessing and modifying the database simultaneously and to recover from failures without data loss. Here's a detailed guide on how you can implement these features:

Concurrency Control

Concurrency control in the ATM System is vital for maintaining data integrity and consistency when multiple users or transactions access and modify data concurrently. Transaction isolation levels, like Read Committed or Serializable, define data visibility and locking behavior, preventing anomalies such as dirty reads or non-repeatable reads. Locking mechanisms, such as acquiring exclusive locks on relevant rows during updates, ensure data integrity. Optimistic concurrency control, employing timestamps or version numbers, allows transactions to proceed independently, with conflicts resolved at commit time. These mechanisms ensure a smooth user experience while safeguarding data integrity in the system.

1. Transaction Isolation Levels:

Setting appropriate transaction isolation levels like Read Committed or Serializable can ensure the desired balance between data consistency and concurrency. For instance, using a higher isolation level like Serializable can prevent phenomena like dirty reads and non-repeatable reads by enforcing stricter locking mechanisms.

2. Row-Level Locking:

Implementing row-level locking allows the system to lock individual rows in the database, ensuring that only one transaction can modify a particular row at a time.

This mechanism can prevent conflicts and maintain data integrity, especially in scenarios where multiple users are accessing and updating the same data concurrently.

3. Optimistic Concurrency Control:

Optimistic concurrency control techniques, such as timestamp-based concurrency or versioning, can be employed to allow concurrent transactions to proceed independently without acquiring locks. Conflicts are detected at the time of commit, and appropriate resolution strategies are applied to maintain data consistency.

4. Deadlock Detection and Resolution:

Implementing deadlock detection mechanisms can help identify and resolve deadlock situations where transactions are waiting indefinitely for resources held by each other. Techniques such as timeout-based deadlock detection or deadlock prevention algorithms can be employed to mitigate deadlock occurrences.

5. Concurrency-aware Application Design:

Designing the application to be concurrency-aware can also help in managing concurrent access to data efficiently. This includes minimizing the duration of database transactions, reducing the scope of locks, and optimizing database queries to minimize contention.

Recovery Mechanisms

Recovery mechanisms in database management systems (DBMS) ensure data consistency and durability in the event of system failures or crashes. In the Farm Management System, where data integrity is paramount, implementing robust recovery mechanisms is essential. Several key recovery mechanisms include:

1. Transaction Logging:

Transaction logging involves recording all changes made by transactions to a log file before committing them to the database. In case of a system failure, the log file can be used to recover transactions by replaying or rolling back changes to restore the database to a consistent state.

2. Write-Ahead Logging (WAL):

Write-Ahead Logging is a technique where changes are first written to the log file before modifying the actual data in the database. This ensures that the log contains a record of all committed transactions before the corresponding data changes, providing a consistent recovery point in case of failure.

3. Checkpointing:

Checkpointing involves periodically writing database changes from memory to disk along with a record of the most recent checkpoint. In the event of a crash, the system can use the checkpoint to recover quickly by starting from a known consistent state rather than replaying all transactions from the beginning.

4. Transaction Undo/Redo:

Transaction undo and redo mechanisms are used to reverse or reapply changes made by transactions during recovery. Undo operations roll back incomplete transactions to maintain consistency, while redo operations reapply committed changes recorded in the log to ensure durability.

5. Database Backups:

Regular database backups are essential for disaster recovery. Backups provide a copy of the database at a specific point in time, allowing administrators to restore the database to a previous state in case of catastrophic failures or data corruption.

6. RAID (Redundant Array of Independent Disks):

RAID configurations provide fault tolerance by distributing data across multiple disks and using redundancy to recover from disk failures. RAID levels like RAID 1 (mirroring) and RAID 5 (striping with parity) ensure data availability and integrity.

By implementing these recovery mechanisms, the Farm Management System can minimize data loss, maintain data consistency, and ensure business continuity in the face of system failures or crashes. Regular testing and monitoring of these mechanisms are essential to verify their effectiveness and reliability in real-world scenarios.

Implementation in SQL

Here's how transactions and locking can be implemented in SQL within the context of the Farm Management System:

1. Transactions in SQL

```
-- Start a transaction BEGIN
TRANSACTION;

-- Perform database operations within the transaction UPDATE
AddAgroProducts
SET Price = Price * 1.1
WHERE FarmingType = 'Organic';

-- Commit the transaction COMMIT
TRANSACTION;

-- Rollback the transaction if an error occurs ROLLBACK
TRANSACTION;
```

6. CODE FOR THE PROJECT

DATABASE

```
import tkinter as tk
from tkinter import Frame, Label, Entry, Button, messagebox, simpledialog
import mysql.connector
from decimal import Decimal

class ATM:
    def _init_(self, master):
        self.master = master
        master.title("ATM Simulator")
        master.geometry('600x400')
        master.configure(bg='#37474F')

        # Database connection
        self.db = mysql.connector.connect(
            host='localhost',
            user='root',
            passwd='1234',
            database='atm_system'
        )
        self.cursor = self.db.cursor()

        # Define Frames
        self.frames = {
            "login": Frame(master, bg='#37474F'),
            "menu": Frame(master, bg='#37474F'),
            "balance": Frame(master, bg='#37474F'),
            "deposit": Frame(master, bg='#37474F'),
            "withdraw": Frame(master, bg='#37474F')
        }

        # Styling
```

```
        self.button_style = {'font': ('Helvetica', 12), 'bg': '#00796B', 'fg': 'white',
'padx': 10, 'pady': 10}
        self.label_style = {'bg': '#37474F', 'fg': 'white', 'font': ('Helvetica', 16)}
        self.entry_style = {'font': ('Helvetica', 14), 'bg': '#455A64', 'fg': 'white'}

        self.setup_frames()
        self.show_frame("login")

    def setup_frames(self):
        # Login components
        Label(self.frames['login'], text="Enter your username:",
**self.label_style).pack(pady=20)
        self.username_entry = Entry(self.frames['login'], **self.entry_style)
        self.username_entry.pack(pady=10)
        Label(self.frames['login'], text="Enter your password:",
**self.label_style).pack(pady=10)
        self.password_entry = Entry(self.frames['login'], show="*",
**self.entry_style)
        self.password_entry.pack(pady=10)
        Button(self.frames['login'], text="Login", command=self.login,
**self.button_style).pack(pady=20)

        # Main Menu components
        Button(self.frames['menu'], text="Check Balance & History",
command=self.show_balance, **self.button_style).pack(fill='x', pady=10)
        Button(self.frames['menu'], text="Deposit Money", command=lambda:
self.show_frame("deposit"), **self.button_style).pack(fill='x', pady=10)
        Button(self.frames['menu'], text="Withdraw Money", command=lambda:
self.show_frame("withdraw"), **self.button_style).pack(fill='x', pady=10)
        Button(self.frames['menu'], text="Logout", command=self.logout,
**self.button_style).pack(fill='x', pady=10)

        # Setup deposit and withdrawal frames
        self.setup_transaction_frame(self.frames['deposit'], 'deposit')
        self.setup_transaction_frame(self.frames['withdraw'], 'withdraw')

    def setup_transaction_frame(self, frame, action):
```

```
        Label(frame, text=f"Enter amount to {action}:",
**self.label_style).pack(pady=20)
        amount_entry = Entry(frame, **self.entry_style)
        amount_entry.pack(pady=10)
        Button(frame, text="Enter", command=lambda:
self.process_transaction(amount_entry.get(), action),
**self.button_style).pack(pady=10)
        Button(frame, text="Back", command=lambda: self.show_frame("menu"),
**self.button_style).pack()

    def process_transaction(self, amount, action):
        if amount.isdigit() and int(amount) > 0:
            amount = Decimal(amount)
            if action == "deposit":
                new_balance = self.current_balance + amount
                self.current_balance = new_balance
                messagebox.showinfo("Successful Deposit", f"Deposited ${amount}.
New balance: ${new_balance}")
            elif action == "withdraw":
                if amount <= self.current_balance:
                    new_balance = self.current_balance - amount
                    self.current_balance = new_balance
                    messagebox.showinfo("Successful Withdrawal", f"Withdrew
${amount}. New balance: ${new_balance}")
                else:
                    messagebox.showerror("Error", "Insufficient funds")
            self.update_account_balance(new_balance)
        else:
            messagebox.showerror("Error", "Invalid amount entered")

    def update_account_balance(self, new_balance):
        update_query = "UPDATE accounts SET balance = %s WHERE id = %s"
        self.cursor.execute(update_query, (new_balance, self.current_account_id))
        self.db.commit()

    def show_balance(self):
```

```
        self.show_frame("balance")
        balance_message = f"Your current balance is: ${self.current_balance:.2f}"
        Label(self.frames['balance'], text=balance_message, **self.label_style).pack()

    def show_frame(self, frame_name):
        for frame in self.frames.values():
            frame.pack_forget()
        self.frames[frame_name].pack(expand=True, fill='both')

    def login(self):
        username = self.username_entry.get()
        password = self.password_entry.get()
        query = "SELECT id, balance FROM accounts WHERE username = %s
AND password = %s"
        self.cursor.execute(query, (username, password))
        result = self.cursor.fetchone()
        if result:
            self.current_account_id, self.current_balance = result
            self.show_frame("menu")
        else:
            messagebox.showerror("Error", "Invalid username or password")

    def logout(self):
        self.show_frame("login")
        self.username_entry.delete(0, tk.END)
        self.password_entry.delete(0, tk.END)

def main():
    root = tk.Tk()
    app = ATM(root)
    root.mainloop()

if _name_ == "_main_":
    main()
```

Concurrency Code :

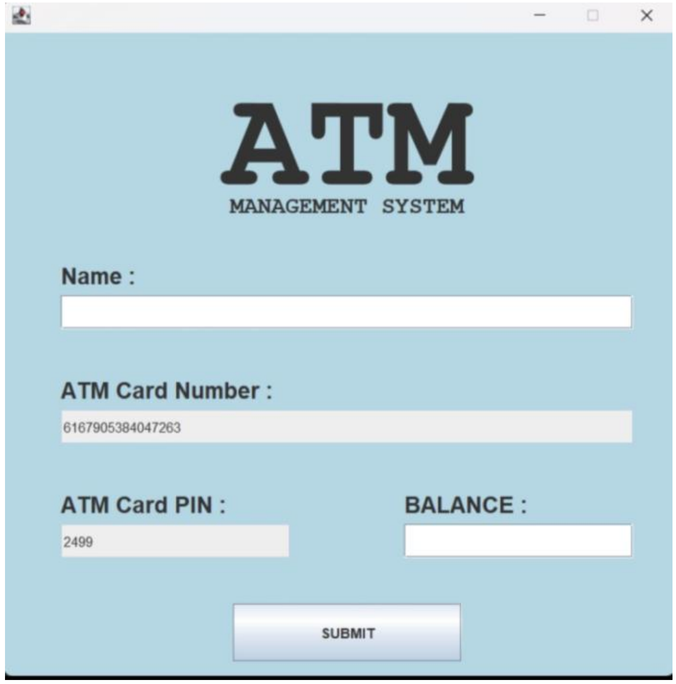
```
import tkinter as tk
from tkinter import Frame, Label, Entry, Button, messagebox, simpledialog
import mysql.connector
from decimal import Decimal
import threading

class ATM:
    def __init__(self, master):
        self.master = master
        master.title("ATM Simulator")
        master.geometry('600x400')
        master.configure(bg='#37474F')

        # Database connection
        self.db = mysql.connector.connect(
            host='localhost',
            user='root',
            passwd='1234',
            database='atm_system'
        )
        self.cursor = self.db.cursor()

        # Define Frames
        self.frames = {
            "login": Frame(master, bg='#37474F'),
            "menu": Frame(master, bg='#37474F'),
            "balance": Frame(master, bg='#37474F'),
            "deposit": Frame(master, bg='#37474F'),
            "withdraw": Frame(master, bg='#37474F')
        }

        # Styling
        self.button_style = {'font': ('Helvetica', 12), 'bg': '#00796B', 'fg': 'white', 'padx': 10,
'pady': 10}
        self.label_style = {'bg': '#37474F', 'fg': 'white', 'font': ('Helvetica', 16)}
        self.entry_style = {'font': ('Helvetica', 14), 'bg': '#455A64', 'fg': 'white'}

        self.setup_frames()
        self.show_frame("login")
```

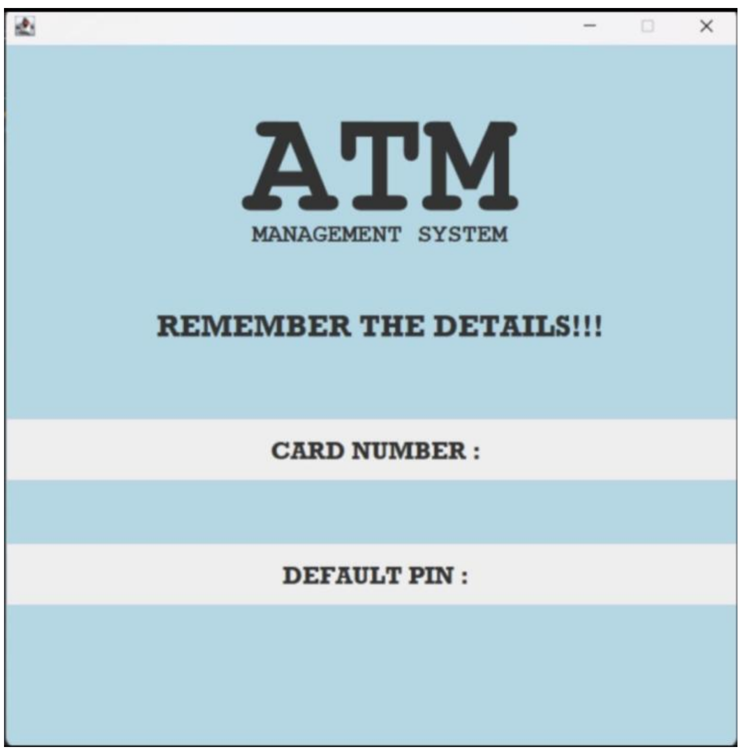
```
    def setup_frames(self):
        # Login components
        Label(self.frames['login'], text="Enter your username:",
**self.label_style).pack(pady=20)
        self.username_entry = Entry(self.frames['login'], **self.entry_style)
        self.username_entry.pack(pady=10)
        Label(self.frames['login'], text="Enter your password:",
**self.label_style).pack(pady=10)
        self.password_entry = Entry(self.frames['login'], show="*", **self.entry_style)
        self.password_entry.pack(pady=10)
        Button(self.frames['login'], text="Login", command=self.login,
**self.button_style).pack(pady=20)

        # Main Menu components
        Button(self.frames['menu'], text="Check Balance & History",
command=self.show_balance, **self.button_style).pack(fill='x', pady=10)
        Button(self.frames['menu'], text="Deposit Money", command=lambda:
self.show_frame("deposit"), **self.button_style).pack(fill='x', pady=10)
        Button(self.frames['menu'], text="Withdraw Money", command=lambda:
self.show_frame("withdraw"), **self.button_style).pack(fill='x', pady=10)
        Button(self.frames['menu'], text="Logout", command=self.logout,
**self.button_style).pack(fill='x', pady=10)

        # Setup deposit and withdrawal frames
        self.setup_transaction_frame(self.frames['deposit'], 'deposit')
        self.setup_transaction_frame(self.frames['withdraw'], 'withdraw')

    def setup_transaction_frame(self, frame, action):
        Label(frame, text=f"Enter amount to {action}:", **self.label_style).pack(pady=20)
        amount_entry = Entry(frame, **self.entry_style)
        amount_entry.pack(pady=10)
        Button(frame, text="Enter", command=lambda:
self.process_transaction(amount_entry.get(), action), **self.button_style).pack(pady=10)
        Button(frame, text="Back", command=lambda: self.show_frame("menu"),
**self.button_style).pack()

    def process_transaction(self, amount, action):
        if amount.isdigit() and int(amount) > 0:
            amount = Decimal(amount)
            if action == "deposit":
                threading.Thread(target=self.deposit, args=(amount,)).start()
            elif action == "withdraw":
                threading.Thread(target=self.withdraw, args=(amount,)).start()
```

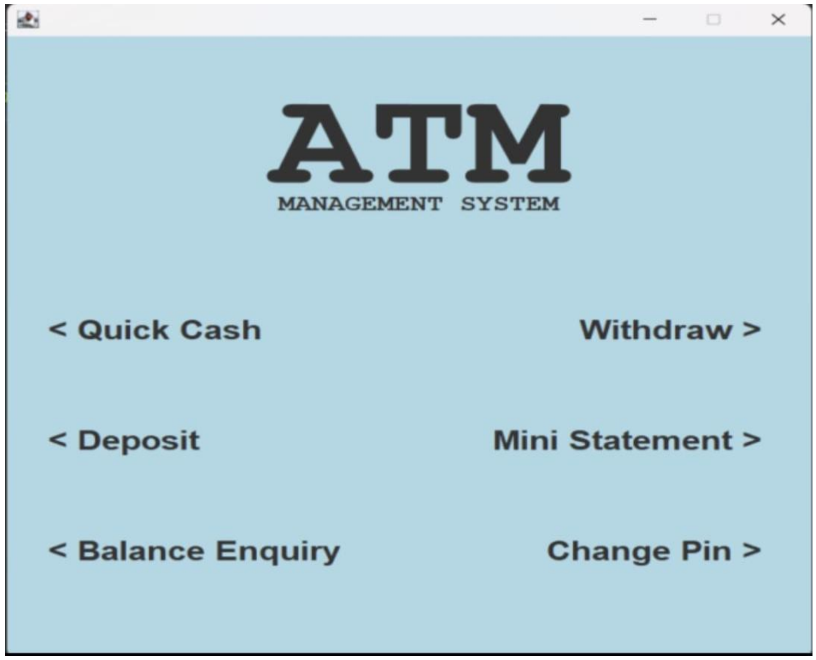
```
        else:
            messagebox.showerror("Error", "Invalid amount entered")

    def deposit(self, amount):
        new_balance = self.current_balance + amount
        self.current_balance = new_balance
        messagebox.showinfo("Successful Deposit", f"Deposited ${amount}. New balance:
${new_balance}")
        self.update_account_balance(new_balance)

    def withdraw(self, amount):
        if amount <= self.current_balance:
            new_balance = self.current_balance - amount
            self.current_balance = new_balance
            messagebox.showinfo("Successful Withdrawal", f"Withdrew ${amount}. New
balance: ${new_balance}")
            self.update_account_balance(new_balance)
        else:
            messagebox.showerror("Error", "Insufficient funds")

    def update_account_balance(self, new_balance):
        update_query = "UPDATE accounts SET balance = %s WHERE id = %s"
        self.cursor.execute(update_query, (new_balance, self.current_account_id))
        self.db.commit()

    def show_balance(self):
        self.show_frame("balance")
        balance_message = f"Your current balance is: ${self.current_balance:.2f}"
        Label(self.frames['balance'], text=balance_message, **self.label_style).pack()

    def show_frame(self, frame_name):
        for frame in self.frames.values():
            frame.pack_forget()
        self.frames[frame_name].pack(expand=True, fill='both')

    def login(self):
        username = self.username_entry.get()
        password = self.password_entry.get()
        query = "SELECT id, balance FROM accounts WHERE username = %s AND
password = %s"
        self.cursor.execute(query, (username, password))
        result = self.cursor.fetchone()
        if result:
            self.current_account_id, self.current_balance = result
```

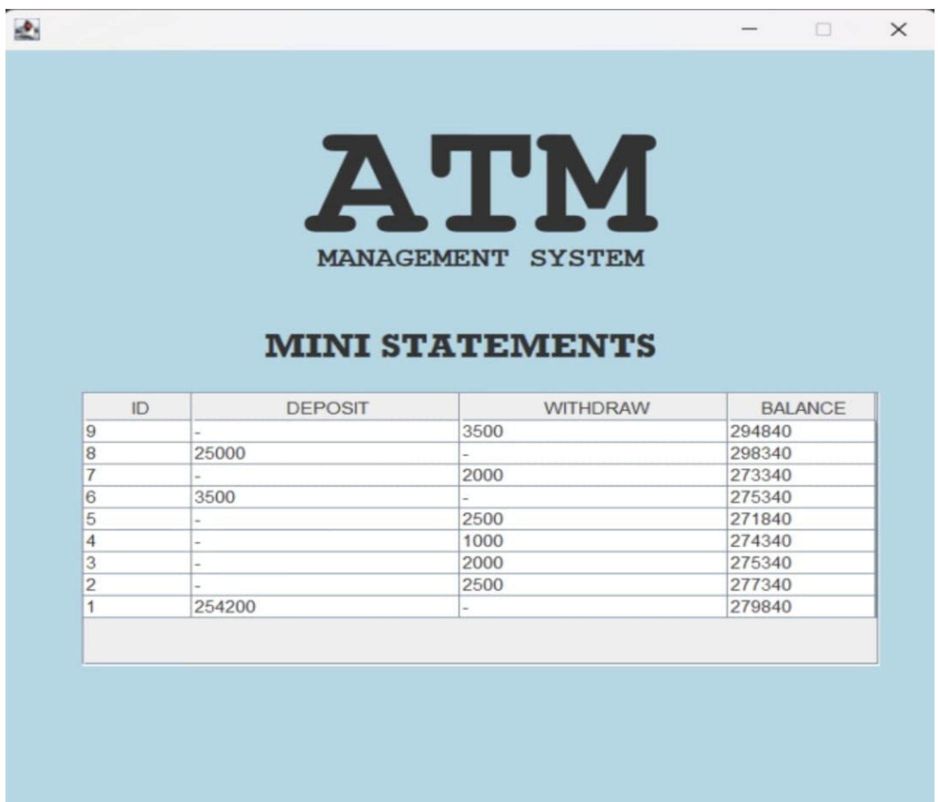
```
            self.show_frame("menu")
        else:
            messagebox.showerror("Error", "Invalid username or password")

    def logout(self):
        self.show_frame("login")
        self.username_entry.delete(0, tk.END)
        self.password_entry.delete(0, tk.END)

def main():
    root = tk.Tk()
    app = ATM(root)
    root.mainloop()

if __name__ == "__main__":
    main()
```

7. RESULT AND DISCUSSION

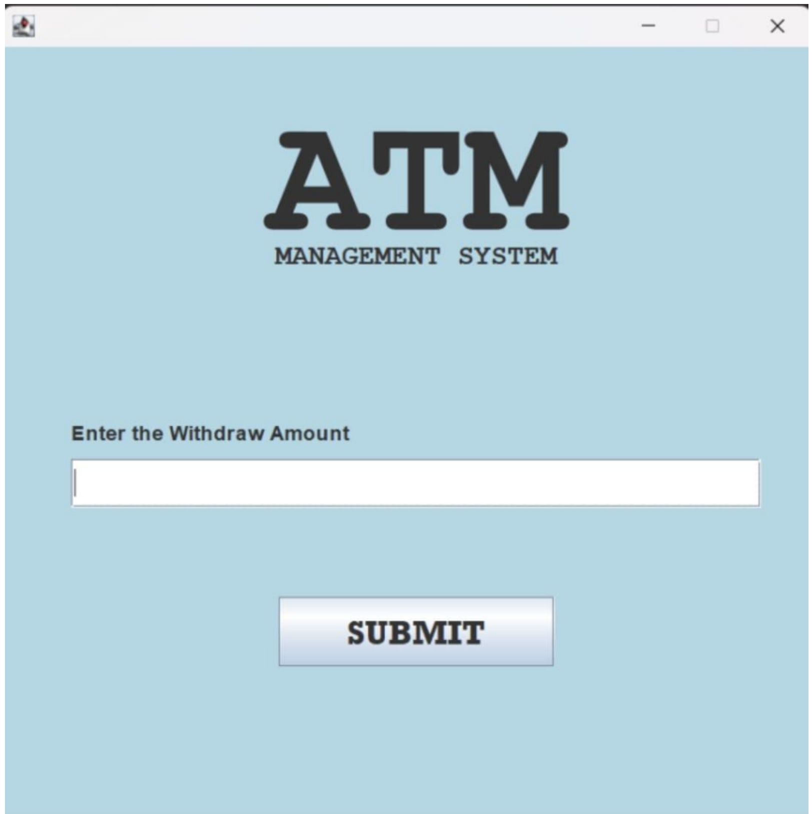


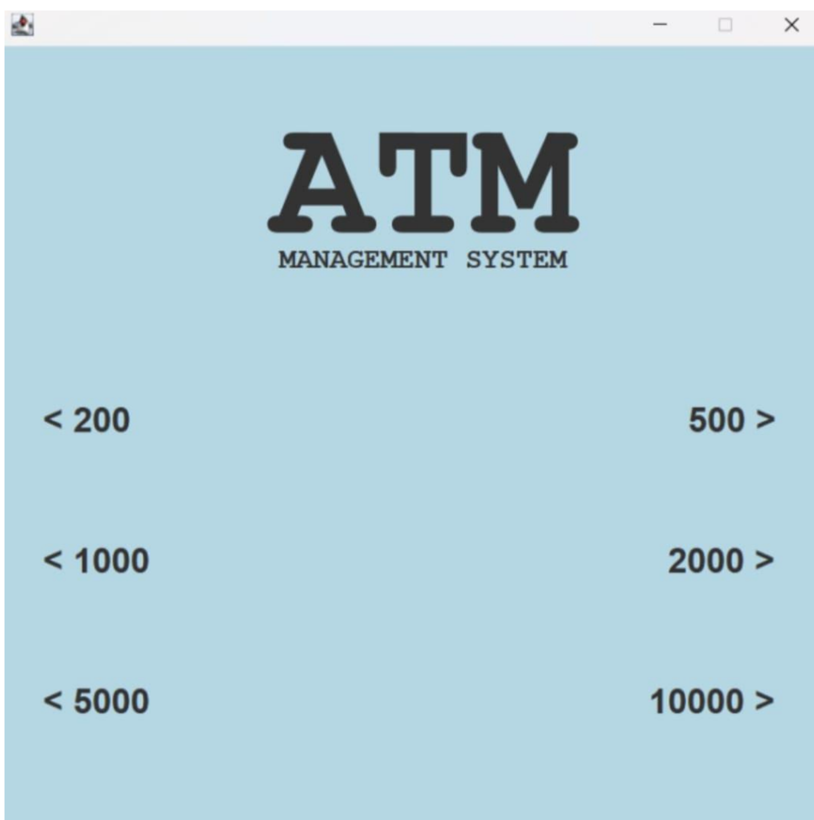


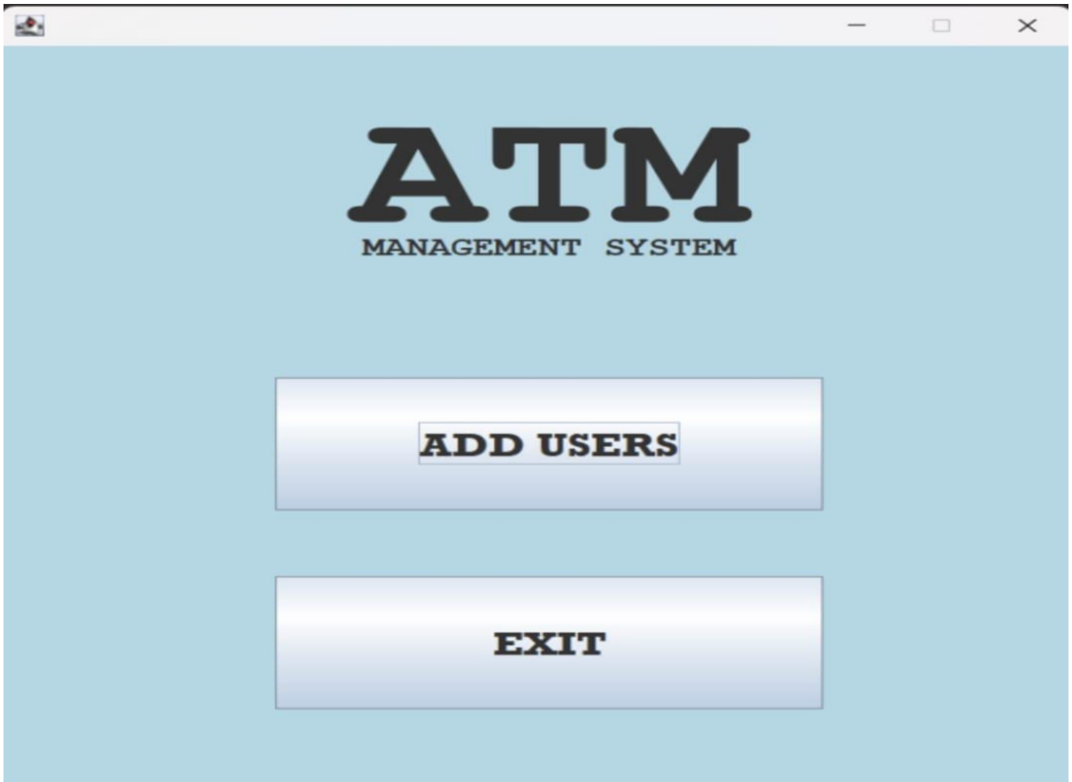


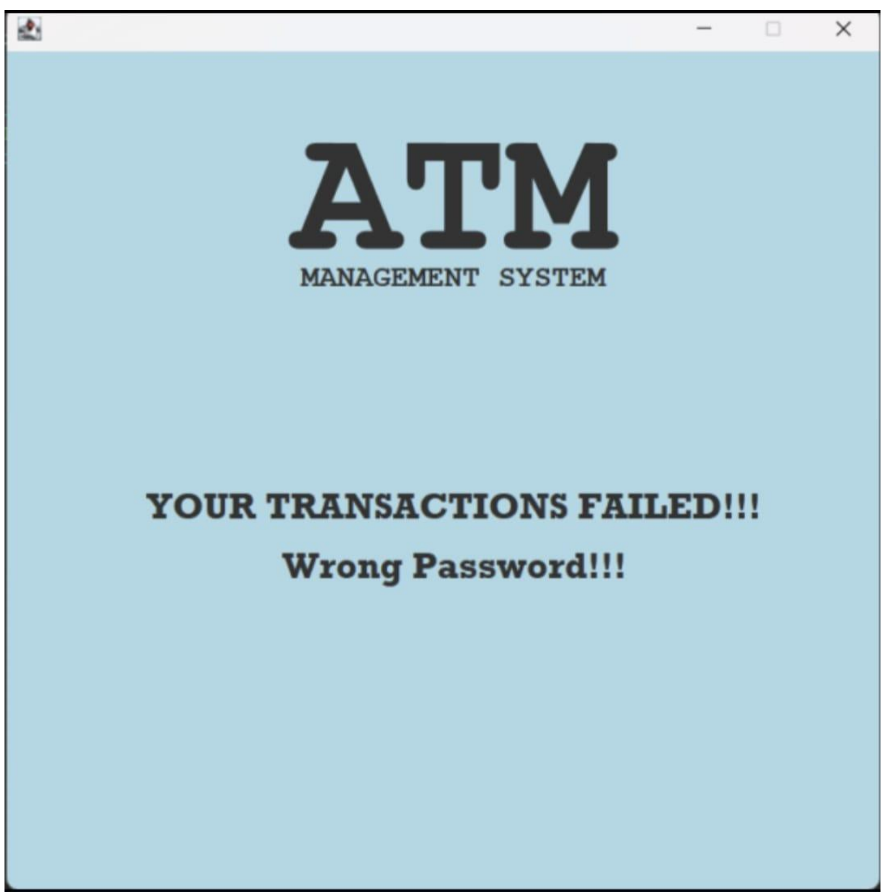


ID	DEPOSIT	WITHDRAW	BALANCE
9	-	3500	294840
8	25000	-	298340
7	-	2000	273340
6	3500	-	275340
5	-	2500	271840
4	-	1000	274340
3	-	2000	275340
2	-	2500	277340
1	254200	-	279840


ATM
MANAGEMENT SYSTEM
Enter the Withdraw Amount
SUBMIT


ATM
MANAGEMENT SYSTEM
< 200
500 >
< 1000
2000 >
< 5000
10000 >


ATM
MANAGEMENT SYSTEM
ADD USERS
EXIT


ATM
MANAGEMENT SYSTEM
YOUR TRANSACTIONS FAILED!!!
Wrong Password!!!

DISCUSSION

The ATM System project in DBMS is a comprehensive system that aims to provide secure and convenient banking services to customers. The system is designed with a robust database management system that ensures the security, integrity, and consistency of data. Overall, the ATM System project in DBMS is an excellent solution for banks that seek to provide efficient and reliable banking services to their customers.