Indian Institute of Technology,
Gandhinagar

SQL Based Bank Account Management Tool

Chandra Shekhar

Keshav Yadav

Contents

1 Introduction

The **Bank Management System** is a software project that simulates how a modern bank works using a database. It helps manage customer accounts, allows deposits, withdrawals, and fund transfers, and keeps everything organized and accurate. All these operations are handled using **stored procedures**, which are special pre-written instructions in the database that ensure each action follows the bank's rules and that encapsulate all critical business logic, such as handling deposits, withdrawals, and inter-account transfers. It is implemented as a relational database in MySQL. The system is designed to follow **ACID properties (Atomicity, Consistency, Isolation, Durability)**, which guarantees that transactions are completed fully or not at all, account balances are always correct, multiple transactions do not interfere with each other, and data is saved safely. It also logs every action, including successful and failed transactions, so the bank can track all activities and handle errors efficiently.

Atomicity is achieved through transaction blocks (`START TRANSACTION...COMMIT/ROLLBACK`), guaranteeing that operations are either fully completed or not at all. Consistency is maintained by enforcing constraints, data types, and business rules within the procedures. The system's design provides a secure, scalable, and efficient foundation for managing customer accounts, tracking financial activities, and generating critical reports, thereby serving as a reliable single source of truth for the bank's operations.

2 Objectives

The primary objectives of this project are refined into the following technical and functional goals: The main goals of this project are:

- **Automate core banking operations:** Use stored procedures (`sp_deposit`, `sp_withdraw`, `sp_transfer_funds`) to handle deposits, withdrawals, and fund transfers securely and consistently. This approach minimizes application-layer errors, prevents SQL injection, and ensures that business rules are consistently enforced.
- **Maintain accurate data:** Ensure that account balances are updated atomically and reflect the true state of the account at all times using database transactions and locking mechanisms to avoid conflicts. Utilize **database transactions** with appropriate locking (`FOR UPDATE`) to prevent race conditions.
- **Secure user access:** Implement a secure user login system by storing **salted hashes** (`PasswordSalt`, `PasswordHash`) for passwords and security question answers, preventing unauthorized access even in the event of a data breach.
- **Keep detailed logs:** Log every successful transaction in the `TransactionLog` table, capturing the state of the account before and after the transaction. This creates a detailed history essential for auditing, customer disputes, and regulatory compliance.
- **Provide administrative control:** Allow administrators to define rules like overdraft limits or daily transaction limits, and enforce them automatically through the database.

3 System Description

The system is architected around a highly normalized relational database schema in MySQL. This design separates data into logical entities to reduce redundancy and improve data integrity. The core components of the schema are:

- **Entity Tables:** These store primary business objects.
  - `Customer`: Contains personal information about each client.
  - `Account`: Holds financial details like `CurrentBalance`, `AccountTypeID`, and status.
  - `Employee`: Stores records of bank employees who may manage accounts.
- **Junction and Linking Tables:**
  - `CustomerAccount`: A many-to-many link between customers and accounts, critically enabling **joint accounts** through an `OwnershipPercentage` field.
- **Lookup Tables:** These provide descriptive, static data to enforce consistency.
  - `AccountType`: Defines types like 'Checking' or 'Savings'.
  - `AccountStatusType`: Defines states like 'Active', 'Frozen', or 'Closed'.
  - `TransactionType`: Categorizes transactions, e.g., 'Deposit', 'Withdrawal'.
- **Security and Authentication Tables:**
  - `UserLogin`, `UserSecurityQuestion`, and `UserSecurityAnswer`: These tables collectively manage user credentials, using salted hashes to securely store sensitive information.
- **Operational and Logging Tables:**
  - `TransactionLog`: The primary audit log, recording every successful financial event.
  - `FailedTransactionLog`: A crucial diagnostic table that captures detailed context for any failed operation.
  - `OverDraftLog`: Specifically tracks instances where an account's balance has gone into the negative.
- **Policy and Rule Tables:**
  - `OverdraftPolicy` and `DailyTransactionLimit`: These tables allow administrators to define and enforce business rules at a granular level.

4 Inputs and Outputs

The system processes a variety of inputs and generates several types of outputs, both for end-users and internal bank administration.

4.1 Inputs

Inputs are primarily the parameters passed to the stored procedures.

- **Account Management Inputs:**
  - New Customer/Account: `sp_add_customer_with_account` takes `p_FirstName`, `p_LastName`, `p_Email`, `p_InitialDeposit`, and `p_AccountTypeID` to create a new customer and their primary account in a single atomic transaction.
- **Transaction Request Inputs:**
  - Deposit: `sp_deposit` requires `p_account_id`, `p_amount`, and the `p_userlogin_id`.
  - Withdrawal: `sp_withdraw` takes `p_account_id`, `p_amount`, and `p_userlogin_id`.
  - Transfer: `sp_transfer_funds` requires `p_from_account`, `p_to_account`, `p_amount`, and `p_userlogin_id`.
- **Administrative Inputs:**
  - These are direct `INSERT` or `UPDATE` statements executed by administrators on policy tables, such as setting a new interest rate in `SavingsInterestRate`.

4.2 Outputs

Outputs range from direct data returned to the application to changes in the database state and pre-compiled reports.

- **Direct Procedural Outputs:**
  - New Account IDs: The `sp_add_customer_with_account` procedure returns the newly created `p_CustomerID` and `p_AccountID` as `OUT` parameters.
  - Monthly Statements: `sp_generate_monthly_statement` returns multiple result sets summarizing account activity.
- **State Changes (Indirect Outputs):**
  - Updated Balances: The modification of the `CurrentBalance` in the `Account` table.
  - New Log Entries: The creation of new rows in `TransactionLog`, `OverDraftLog`, or `FailedTransactionLog`.
- **Reporting Outputs (for Bank Use):**
  - SQL Views: The system provides several pre-defined views that act as live reports, such as `V_Customers_Overdraft` and `V_Customers_TotalBalance_GT5000`.
- **Error Message Outputs:**
  - When a procedure fails, it returns a specific `SQLSTATE '45000'` with a descriptive error message like 'Daily withdrawal limit exceeded'.

5 Functional Modules

The system's functionality is logically partitioned into modules, which are implemented by specific tables and procedures.

- **Account Management Module:** This module is responsible for the lifecycle of customer and account data. It is implemented by the `Customer`, `Account`, and `CustomerAccount` tables, along with the `sp_add_customer_with_account` procedure.
- **Transaction Module:** This is the core engine of the bank. It is implemented entirely by the atomic stored procedures: `sp_deposit`, `sp_withdraw`, and `sp_transfer_funds`.
- **Error Handling Module:** This is a cross-cutting concern implemented within every major stored procedure via the `DECLARE EXIT HANDLER`, the `insert_failed_log` helper procedure, and the `FailedTransactionLog` table.
- **Reporting Module:** This module provides insights into the bank's data. It is implemented through stored procedures like `sp_generate_monthly_statement` and a suite of SQL `VIEW`s.

6 Stored Procedures Summary Table

The following table summarizes all major stored procedures in the Bank Management System, including their purpose, inputs, outputs, and logging behavior.

Procedure	Purpose	Inputs	Outputs / Logs
sp_add_customer_with_account	Creates a new customer and associated bank account	p_FirstName, p_LastName, p_Email, p_InitialDeposit, p_AccountTypeID	New CustomerID and AccountID; entries in Customer and Account tables; links in CustomerAccount table; logs successful creation
sp_deposit	Deposits money into an account	p_account_id, p_amount, p_userlogin_id	Updates CurrentBalance; entry in TransactionLog; errors in FailedTransactionLog
sp_withdraw	Withdraws money from an account	p_account_id, p_amount, p_userlogin_id	Updates CurrentBalance; entry in TransactionLog; logs insufficient funds or limit errors in FailedTransactionLog

Procedure	Purpose	Inputs	Outputs / Logs
sp_transfer_funds	Transfers money between two accounts	p_from_account, p_to_account, p_amount, p_userlogin_id	Updates balances of both accounts; entries in TransactionLog; errors logged in FailedTransactionLog; ensures atomicity
sp_generate_monthly_statement	Generates monthly transaction summary	p_account_id, p_month, p_year	Result set of all transactions for the month; opening and closing balances; suitable for reports
insert_failed_log	Logs failed transactions for auditing	Procedure name, input parameters as JSON, error message	Entry in FailedTransactionLog with detailed context

7 Error Handling and Logging

Robust error handling is a cornerstone of this system, designed to ensure no transaction is lost or improperly processed.

- **Proactive Validation:** Before attempting any financial operation, the procedures perform checks for insufficient funds, invalid account IDs, and daily transaction limits. For example, it checks if `v_balance_before + v_policy_amt` is less than the requested amount.
- **Centralized Exception Handling:** The `DECLARE EXIT HANDLER FOR SQLEXCEPTION` block in each stored procedure acts as a universal catch-all for any unexpected SQL error during a transaction.
- **Detailed Failure Logging:** Upon catching an error, the handler calls `insert_failed_log`, which records a structured log entry in `FailedTransactionLog` containing the procedure name, a `JSON` object with the input parameters, and the specific database error message.

8 Future Enhancements

Future enhancements could include:

- Developing a **REST API** on top of the database to allow integration with web and mobile applications.
- Implementing **Role-Based Access Control (RBAC)** by expanding on the `Role` field in the `LoginAccount` table.
- Adding automated auditing using database **triggers** to track changes to sensitive data fields.

9 Conclusion

The Bank Management System provides a secure and efficient platform for managing bank accounts and transactions. It ensures that all operations, such as deposits, withdrawals, and transfers, are handled accurately and safely using stored procedures and database transactions. The system also keeps detailed logs of every action, helping with auditing, error handling, and customer support. By automating these processes and maintaining data integrity, the project demonstrates a reliable way for banks to operate digitally.