Solent University

**ADVANCED DATABASE SYSTEMS (QHO541)**

**Author: Enea Rina**

**Student Number: 10407625**

**Course Title: BSc (Hons) Computer sience**

**Module Lider: Dr. Muhammad Akram**

1. **Introduction to NoteBank**

The **Bank management system** called ‘NoteBank’ is a database and program solution designed to support and optimize the operations of a modern banking institution and their clients. As financial institutions handle a vast amounts of data daily, the need for a secure and efficient system has never been greater. This database system is developed to ensure management of banking processes, secure handling of sensitive data, and real-time accessibility to crucial information.

In addition to managing data, the database ensures data integrity, security, and scalability. It incorporates features like foreign key constraints, automated triggers to maintain consistency and optimize performance. Furthermore, the system's design allows it to adapt to the bank's growing needs, making it future-proof for scaling operations and handling an increasing volume of transactions and customers.

The objective of this project is to design, implement, and test a database solution that not only meets the business requirements but also provides a foundation for advanced banking features. Through normalization, query optimization, and rigorous testing, the system ensures that it can support complex banking operations while delivering an efficient and user-friendly experience through a program developed in python that lets the clients of the bank manage their accounts and money.

1. **Part 1.**
2. **Business case and requirements:**

The business case selected is a **Bank Management System** designed to manage essential and basic banking operations. The system is built to store and process data for multiple interconnected entities such as **branches**, **staff**, **users**, **accounts**, **loans**, **credit cards**, **transactions**, and **customer support**. By implementing these entities, the system provides an overview of the common bank's operations and supports key functionalities like transaction and user data and accounts processing, loan management, and customer issues.

The database is structured to ensure functionality, data integrity, and scalability. It supports features like secure data management,strong relationships between entities, and automated triggers for operational efficiency.

To establish a strong foundation for the construction and modeling of the database, seven key business requirements were taken in consideration to guide its design and implementation:

* Managing branches and their total balances while tracking details as their unique identifiers, location, number of staff and clients. For example by calculating the total assets held by each branch by summing the balances of all associated accounts.
* Tracking staff information and their role by storing and managing their data such as names, roles, email, salary, and assigned branches to easily individuate members as managers and accountants and retrieve their information if necessary.
* Recording users personal data and linking them to an account related to a specific branch. In this way we can have track of the users and the related personal account and also verify their data if necessary. All of this while implementing a cryptography algorithm to ensure the privacy of personal information's.
* Managing loans and their statuses as the system will track loans issued to users including loan amounts, interest rates and current statuses (active, paid, defaulted). In this way the system can even generate a report of overdue loans or loans with high outstanding amounts.
* Handling credit card details, the database will store credit card information, including card numbers, expiration dates, credit limits, and user associations.
* Recording all financial transactions, such as deposits, withdrawals, and transfers, will be recorded with timestamps, amount, account associations, recipient, and transaction types. In this way we can provide a statement of recent transactions for an account.
* Providing Customer support interactions that will be logged with timestamps, issues reported, and resolutions provided to ensure that the staff members can resolve the issues.

1. **Entity Relationship Diagram**

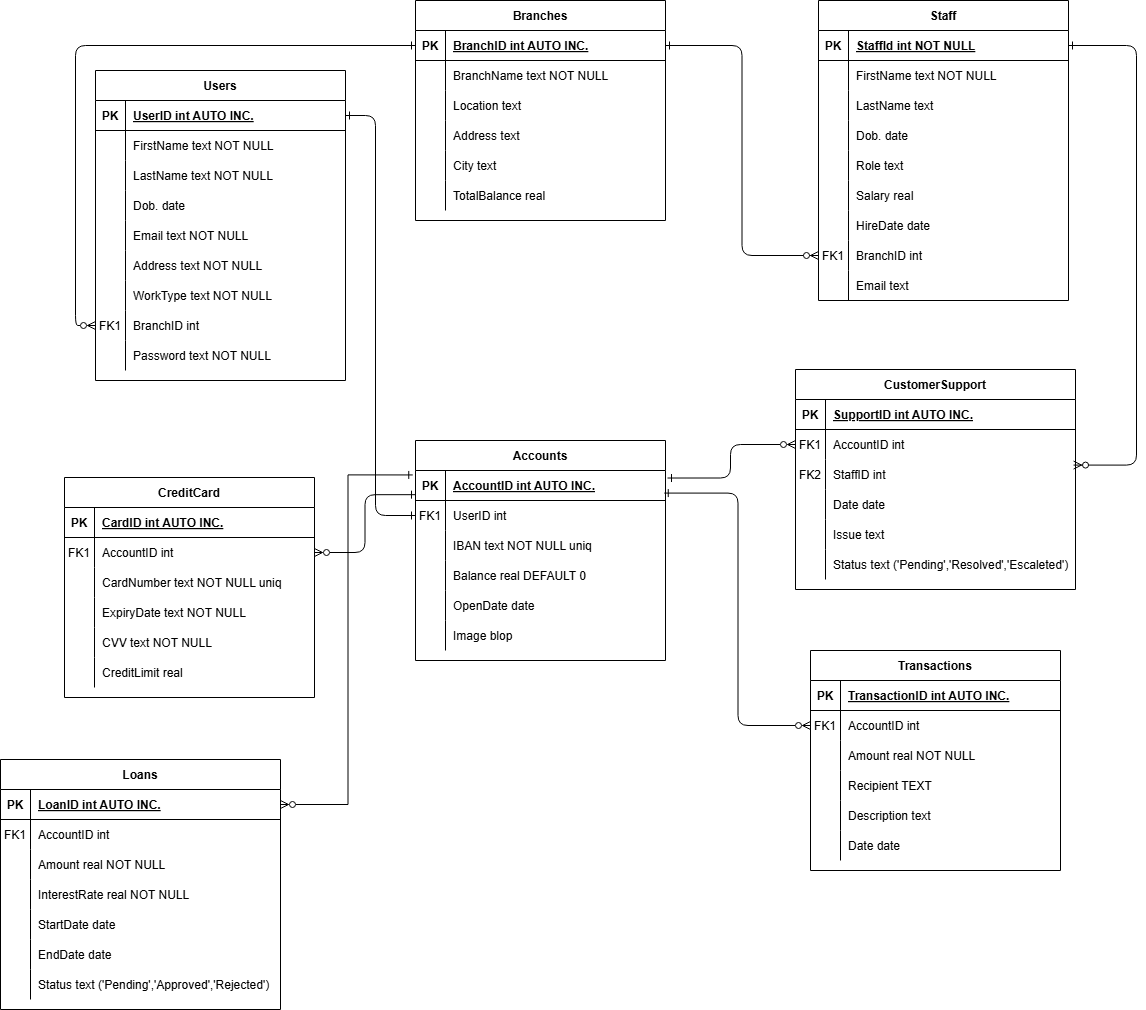
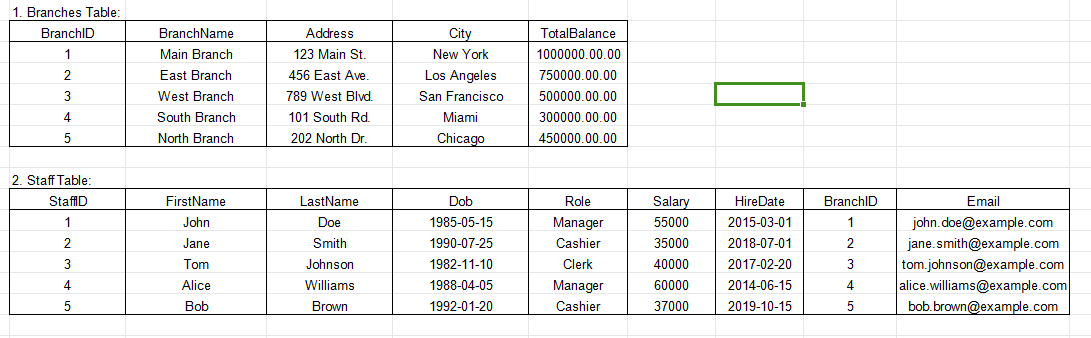
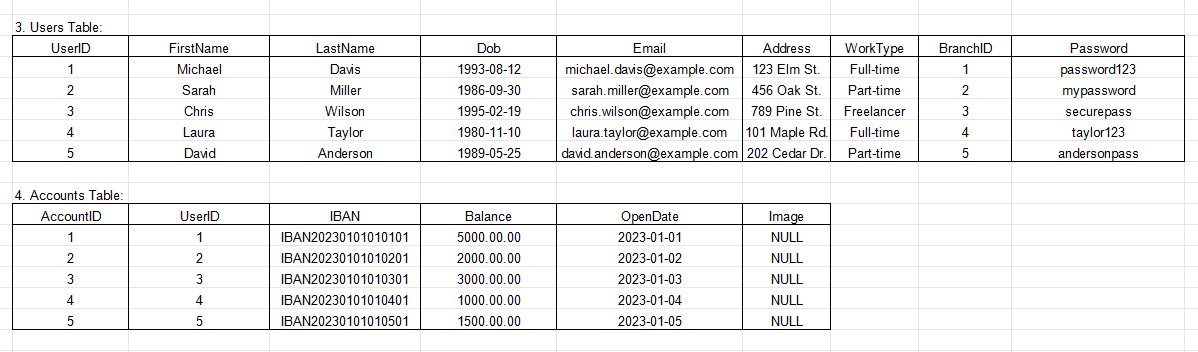
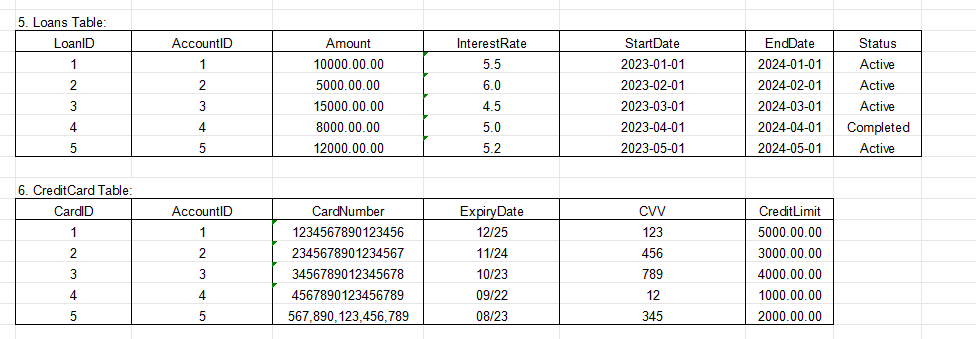
****

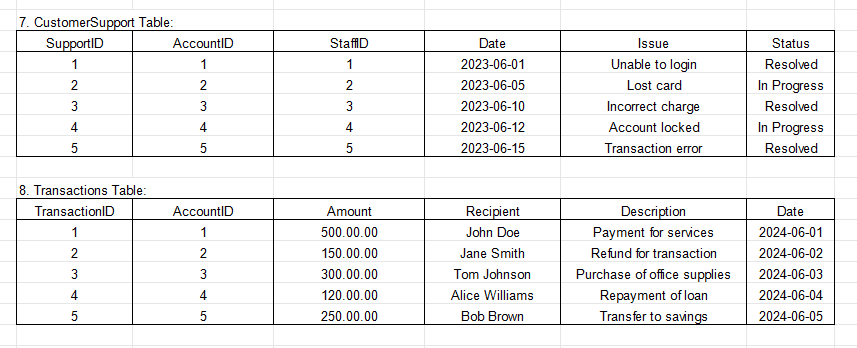
Photo 1 (Entity Relationship Database)

1. **File with collected data for the database**









1. **Sql query database development**

-- Table: Branches

CREATE TABLE Branches (

BranchID INTEGER PRIMARY KEY AUTOINCREMENT,

BranchName TEXT NOT NULL,

Address TEXT,

City TEXT,

TotalBalance REAL

);

-- Table: Staff

CREATE TABLE Staff (

StaffID INTEGER PRIMARY KEY AUTOINCREMENT,

FirstName TEXT NOT NULL,

LastName TEXT NOT NULL,

Dob DATE,

Role TEXT,

Salary REAL,

HireDate DATE,

BranchID INT,

Email TEXT,

FOREIGN KEY (BranchID) REFERENCES Branches(BranchID)

);

-- Table: Users

CREATE TABLE Users (

UserID INTEGER PRIMARY KEY AUTOINCREMENT,

FirstName TEXT NOT NULL,

LastName TEXT NOT NULL,

Dob DATE,

Email TEXT NOT NULL,

Address TEXT NOT NULL,

WorkType TEXT NOT NULL,

BranchID INT,

Password TEXT NOT NULL,

FOREIGN KEY (BranchID) REFERENCES Branches(BranchID)

);

-- Table: Accounts

CREATE TABLE Accounts (

AccountID INTEGER PRIMARY KEY AUTOINCREMENT,

UserID INT,

IBAN TEXT NOT NULL UNIQUE,

Balance REAL DEFAULT 0,

OpenDate DATE,

Image BLOB,

FOREIGN KEY (UserID) REFERENCES Users(UserID)

);

-- Table: Loans

CREATE TABLE Loans (

LoanID INTEGER PRIMARY KEY AUTOINCREMENT,

AccountID INT,

Amount REAL NOT NULL,

InterestRate REAL NOT NULL,

StartDate DATE,

EndDate DATE,

Status TEXT,

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

-- Table: CreditCard

CREATE TABLE CreditCard (

CardID INTEGER PRIMARY KEY AUTOINCREMENT,

AccountID INT,

CardNumber TEXT NOT NULL UNIQUE,

ExpiryDate TEXT NOT NULL,

CVV TEXT NOT NULL,

CreditLimit REAL,

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

-- Table: CustomerSupport

CREATE TABLE CustomerSupport (

SupportID INTEGER PRIMARY KEY AUTOINCREMENT,

AccountID INT,

StaffID INT,

Date DATE,

Issue TEXT,

Status TEXT,

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID),

FOREIGN KEY (StaffID) REFERENCES Staff(StaffID)

);

-- Table: Transactions

CREATE TABLE Transactions (

TransactionID INTEGER PRIMARY KEY AUTOINCREMENT,

AccountID INT,

Amount REAL NOT NULL,

Recipient TEXT NOT NULL,

Description TEXT,

Date DATE,

FOREIGN KEY (AccountID) REFERENCES Accounts(AccountID)

);

1. **Referential Integrity constraints**

* **Branches** is related to **Staff**, **Users**, and **Accounts** via a foreign key BranchID. This ensures that all staff, users, and accounts are correctly associated with an existing branch, allowing the system to track branch operations.
* **Staff** is connected to **CustomerSupport** via a foreign key StaffID. This ensures that every customer support ticket is handled by an existing staff member, maintaining accountability and data accuracy.
* **Users** is linked to **Accounts** via a foreign key UserID. This allows the system to track and relate all the accounts data to one user and let it log-in with his credentials.
* **Accounts** is connected to **Loans**, **CreditCard**, **Transactions**, and **CustomerSupport** via a foreign key AccountID. This ensures that all loans, credit cards, and transactions are tied to a valid account, maintaining the integrity of financial data.

1. **Part 2. Database Development**
2. **Implementation of two Triggers**

To maintain accurate branch financial data a trigger is designed to automatically updating the TotalBalance field in the Branches table whenever an account's balance is updated. It ensures that the branch's total balance reflects the sum of all associated account balances.

CREATE TRIGGER UpdateBranchBalance

AFTER UPDATE ON Accounts

FOR EACH ROW

BEGIN

UPDATE Branches

SET TotalBalance = (

SELECT COALESCE(SUM(Balance), 0)

FROM Accounts

WHERE UserID IN (

SELECT UserID

FROM Users

WHERE BranchID = (

SELECT BranchID

FROM Users

WHERE UserID = NEW.UserID

)

)

)

WHERE BranchID = (

SELECT BranchID

FROM Users

WHERE UserID = NEW.UserID

);

END;

The second trigger ensures that every new user automatically has an associated account created in the Accounts table, simplifying user onboarding and ensuring consistent user-account relationships so in this way each time a new user registers automatically an account will be created associated with that user.

CREATE TRIGGER create\_account\_after\_user\_insert

AFTER INSERT ON Users

FOR EACH ROW

BEGIN

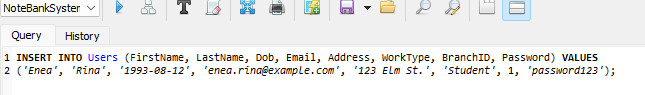
-- Insert a new account associated with the newly created user

INSERT INTO Accounts (UserID, IBAN, OpenDate)

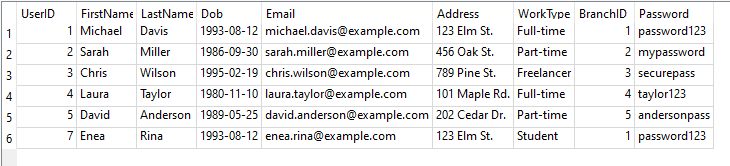
VALUES (NEW.UserID, 'IBAN' || strftime('%Y%m%d%H%M%S', 'now'), DATE('now'));

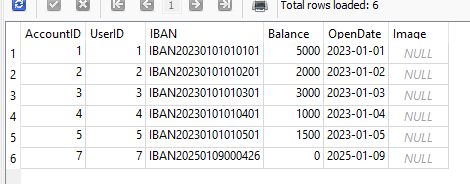
END;

Now to demonstrate the correct functionality for both the triggers a new User will be inserted to the database and automatically an account will be created in the Accounts table. After the Account is created we will set the balance of that account to £100 to demonstrate that the trigger UpdateBranchBalance will update the Main branch that is the corrispective of the User Account.

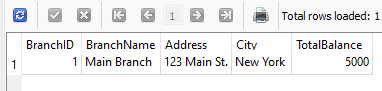


Now in the Users Table and Accounts Table in the last row we will se the new data stored



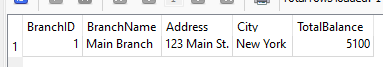


Now to test the trigger that updates automatically the Branch balance we will set the balance of the new account generated to £100 and we will see that amount added to the branch total.





Now the TotalBalance from Branches will automatically be:



1. **Implementation of two views**

A view called ActiveLoans that provides a comprehensive list of all ‘Active’ loans, including borrower details and their associated branch is implemented. This helps the business monitor active loans across branches and ensure borrowers are meeting repayment terms.

CREATE VIEW ActiveLoans AS

SELECT

L.LoanID,

L.Amount,

L.InterestRate,

L.StartDate,

L.EndDate,

L.Status,

A.IBAN,

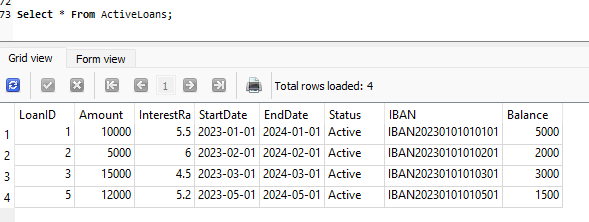
A.Balance

FROM Loans L

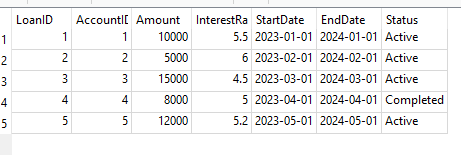
JOIN Accounts A ON L.AccountID = A.AccountID

WHERE L.Status = 'Active';

Result of the View with the data provided:



We can see that the one not Active has not been selected:



The second view called BranchPerformance summarizes each branch's financial status, current users and total stuff, providing key insights into total account balances and loan activity. This information is critical for evaluating branch performance.

CREATE VIEW BranchSummary AS

SELECT

B.BranchID,

B.BranchName,

COUNT(U.UserID) AS TotalUsers,

SUM(A.Balance) AS TotalBranchBalance,

COUNT(S.StaffID) AS TotalStaff

FROM Branches B

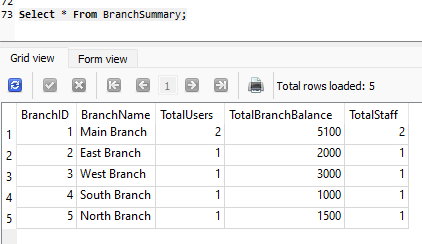
LEFT JOIN Users U ON B.BranchID = U.BranchID

LEFT JOIN Accounts A ON U.UserID = A.UserID

LEFT JOIN Staff S ON B.BranchID = S.BranchID

GROUP BY B.BranchID;

Result of the view provided:



1. **Querys that meet business requirements**
2. **Print a message:**

**To print a message in our database we create a case where we need to ensure that a transaction is not lower than 0 since it would be impossible, if the case happens a message will be printed. To meet the requirement a trigger will be created that ensures the check:**

CREATE TRIGGER check\_transaction\_amount

BEFORE INSERT ON Transactions

FOR EACH ROW

BEGIN

-- Ensure that the transaction amount is greater than zero

SELECT CASE

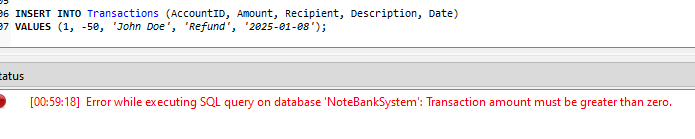
WHEN NEW.Amount <= 0 THEN

RAISE(ABORT, 'Transaction amount must be greater than zero.')

END;

END;

**Test of the query trying to insert a transaction of -50£:**



**We can also ensure to print a message with querys like:**

SELECT 'Transaction executed successfully!' AS Message, \*

FROM Transactions

WHERE TransactionID = 1;

**That provides this output:**



1. **Use of Join between two or more tables**

To demonstrate the use of JOIN between two or more tables we will recreate a first case where we need to output a list of all transactions, showing the transaction amount, date, the user who made the transaction, and the branch associated with the user.

SELECT

Transactions.TransactionID,

Transactions.Amount,

Transactions.Date,

Users.FirstName || ' ' || Users.LastName AS UserName,

Branches.BranchName AS BranchName

FROM

Transactions

JOIN

Accounts ON Transactions.AccountID = Accounts.AccountID

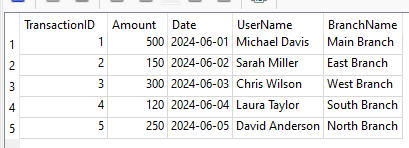
JOIN

Users ON Accounts.UserID = Users.UserID

JOIN

Branches ON Users.BranchID = Branches.BranchID;

**Output:**



**A second demonstrative query calculates the total number and amount of transactions performed at each branch.**

SELECT

Branches.BranchID,

Branches.BranchName,

COUNT(Transactions.TransactionID) AS TotalTransactions,

SUM(Transactions.Amount) AS TotalTransactionAmount

FROM

Branches

JOIN

Users ON Branches.BranchID = Users.BranchID

JOIN

Accounts ON Users.UserID = Accounts.UserID

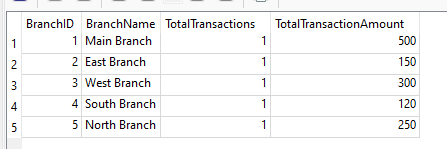
JOIN

Transactions ON Accounts.AccountID = Transactions.AccountID

GROUP BY

Branches.BranchID, Branches.BranchName

**Output:**



1. **Use of GROUP BY with HAVING**

To demonstrate a use of GROUP BY with HAVING we will be summarizing account balances by branch and filtering branches with total balances exceeding £2000.

SELECT

Branches.BranchID,

Branches.BranchName,

SUM(Accounts.Balance) AS TotalBalance

FROM

Branches

JOIN

Users ON Branches.BranchID = Users.BranchID

JOIN

Accounts ON Users.UserID = Accounts.UserID

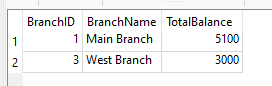
GROUP BY

Branches.BranchID, Branches.BranchName

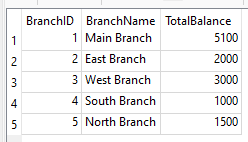
HAVING

SUM(Accounts.Balance) > 2000;

**Output:**



**If we display all the branches with their total outputs we can see that the result of our query is correct:**



**In a second case scenario we need to count the number of active loans in each branch and filters branches with more than 1 active loans.**

SELECT

Branches.BranchID,

Branches.BranchName,

COUNT(Loans.LoanID) AS ActiveLoanCount

FROM

Branches

JOIN

Users ON Branches.BranchID = Users.BranchID

JOIN

Accounts ON Users.UserID = Accounts.UserID

JOIN

Loans ON Accounts.AccountID = Loans.AccountID

WHERE

Loans.Status = 'Active'

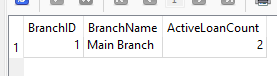
GROUP BY

Branches.BranchID, Branches.BranchName

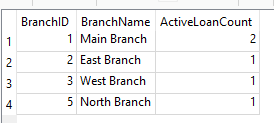
HAVING

COUNT(Loans.LoanID) > 1;

**Output:**



**If we display all of them we can see the correctness of the previous query:**

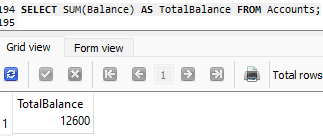


1. **Use of SQLite Functions**

To implement the SUM() function will be provided a query calculates the **total funds held in all accounts** across the bank. It helps the management monitor the bank's overall financial strength and provides insights into liquidity.

SELECT SUM(Balance) AS TotalBalance FROM Accounts;

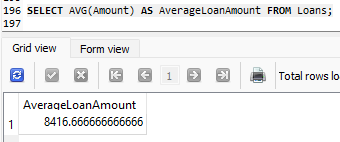
Output:



In a second case to implement the AVG() function will be provided a query that calculates the **average size of loans issued** by the bank. It provides insights into the typical loan amount and helps evaluate whether the bank is meeting its loan objectives.

SELECT AVG(Amount) AS AverageLoanAmount FROM Loans;

Output:



In a third scenario to implement a function a query categorizes accounts based on their balance (LOW, MEDIUM, HIGH) , allowing the bank to **segment customers** for targeted services or promotions.

SELECT

AccountID,

IBAN,

Balance,

CASE

WHEN Balance < 1000 THEN 'Low'

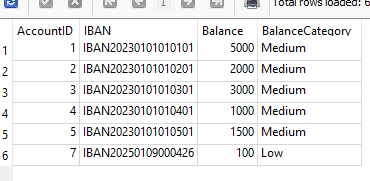
WHEN Balance BETWEEN 1000 AND 10000 THEN 'Medium'

ELSE 'High'

END AS BalanceCategory

FROM Accounts;

Output:



1. **Optimisation querys**
2. **dedeede**
3. **Part 3**