University for the Creative Arts

BERLIN SCHOOL OF BUSINESS & INNOVATION

Essay / Assignment Title: Database System Design for a Banking System

Program title: Enterprise Data Warehouses and Database Management Systems

Name: Kartik Goti

Year: 2023-24

CONTENTS

Statement of compliance with academic ethics and the avoidance of plagiarism

I honestly declare that this dissertation is entirely my own work and none of its part has been copied from printed or electronic sources, translated from foreign sources and reproduced from essays of other researchers or students. Wherever I have been based on ideas or other people texts I clearly declare it through the good use of references following academic ethics.

(In the case that is proved that part of the essay does not constitute an original work, but a copy of an already published essay or from another source, the student will be expelled permanently from the postgraduate program).

Name and Surname (Capital letters):

KARTIK GOTI

Date: 29/09/2023

INTRODUCTION

To ensure the smooth running of financial institutions and the happiness of their customers in today's dynamic financial environment, effective management of banking operations is essential. A sophisticated and extensive database system called the Bank Management System is intended to optimize and streamline a financial organization's daily activities. The management of crucial banking operations, such as account management, client data, personnel records, loan processing, and transaction tracking, is supported by this system.

A reliable framework for classifying, archiving, and retrieving data pertinent to the banking sector is provided by the Bank Management System. It is crucial in improving customer service, guaranteeing data correctness, and assisting bank management and staff in making well-informed decisions.

We will delve into the many entities and attributes of the Bank Management System as we examine its essential parts and operations throughout this assignment. We'll look at the database structure that underpins this system and demonstrate how it manages customers, accounts, workers, loans, branches, and transactions efficiently. We will also show how this system enables a broad range of actions, including data retrieval and insertion as well as updates and deletions, all of which are crucial for preserving the integrity and effectiveness of a contemporary banking institution.

CHAPTER ONE

This database structure was carefully designed to make it easier to manage many areas of the banking industry effectively. Six different tables make up the system, each of which is essential to the bank's smooth operation.

1. **Accounts**

   The "Accounts" table, which houses essential account-related data, is a key component of the Bank Management System. To avoid duplications, each account is given a distinct Account_ID. The Client_ID, a foreign key referencing the "Client" table, connects this table to the client who owns the account by storing information like the account type, current amount, and account inception date. The bank can effectively manage accounts, keep track of balances, and link them to particular clients for banking operations and client services thanks to this data.

   - Account_ID (Primary Key)
   - Type
   - Balance
   - Open_Date
   - Client_Id (Foreign Key : Client)

2. **Branch**

   The "Branch" table, which provides thorough insights into the bank's many branches, is an essential part of the Bank Management System. The Branch_ID uniquely identifies each branch, preventing data duplication. The table contains crucial details including the branch's name, city, physical address, and branch manager's name. This information supports effective branch administration and offers useful specifics for organizational decision-making.

   - Branch_ID (Primary Key)
   - Name
   - City
   - Address
   - Manager_Name

3. Client

The "Client" database, which houses thorough data about the bank's customers, is an essential part of the Bank Management System. To guarantee data correctness and avoid duplication, each client is given a special Client_ID. The client's name, address, date of birth, city of residence, email address, and a reference to the related branch through the Branch_ID, a foreign key connected to the "Branch" database, are all included in this table along with other crucial information. The bank uses this information to manage customer relationships, track client information, and ensure successful client communication.

- Client_ID (Primary Key)
- Name
- Address
- DOB
- City
- Email
- Branch_ID (Foreign Key : Branch)

4. Employee

The "Employee" table, which contains critical information on the bank's staff, is a key component of the Bank Management System. Because each employee has a unique Employee_ID, accurate and distinct records are guaranteed. Significant employee data is captured in the table, including the employee's first and last names, date of birth, city of residence, residential address, job position, income, and a reference to the branch where they work via the Branch_ID, a foreign key connected to the "Branch" record. Within the bank, this information makes effective personnel management and organizational planning possible.

- Employee_ID (PK)
- First_Name
- Last_Name
- DOB

- Address
- City
- Salary
- Position
- Branch_ID (Foreign Key : Branch)

5. **Loan**

   An important part of the Bank Management System is the "Loan" table, which contains detailed data regarding loans that the bank offers. A distinct Loan_ID is given to each loan in order to maintain correct monitoring and avoid duplications. Important loan information, such as the loan amount, interest rate, start date, termination date, and current status are all included in the table. Additionally, it uses the Branch_ID, a foreign key connected to the "Branch" database, to link each loan to the branch where it is handled. This information facilitates effective loan administration, tracking of loan statuses, and operational association of loans with particular branches.

   - Loan_ID (PK)
   - Amount
   - Interest_rate
   - Start_date
   - End_date
   - Status
   - Branch_ID (Foreign Key: Branch)

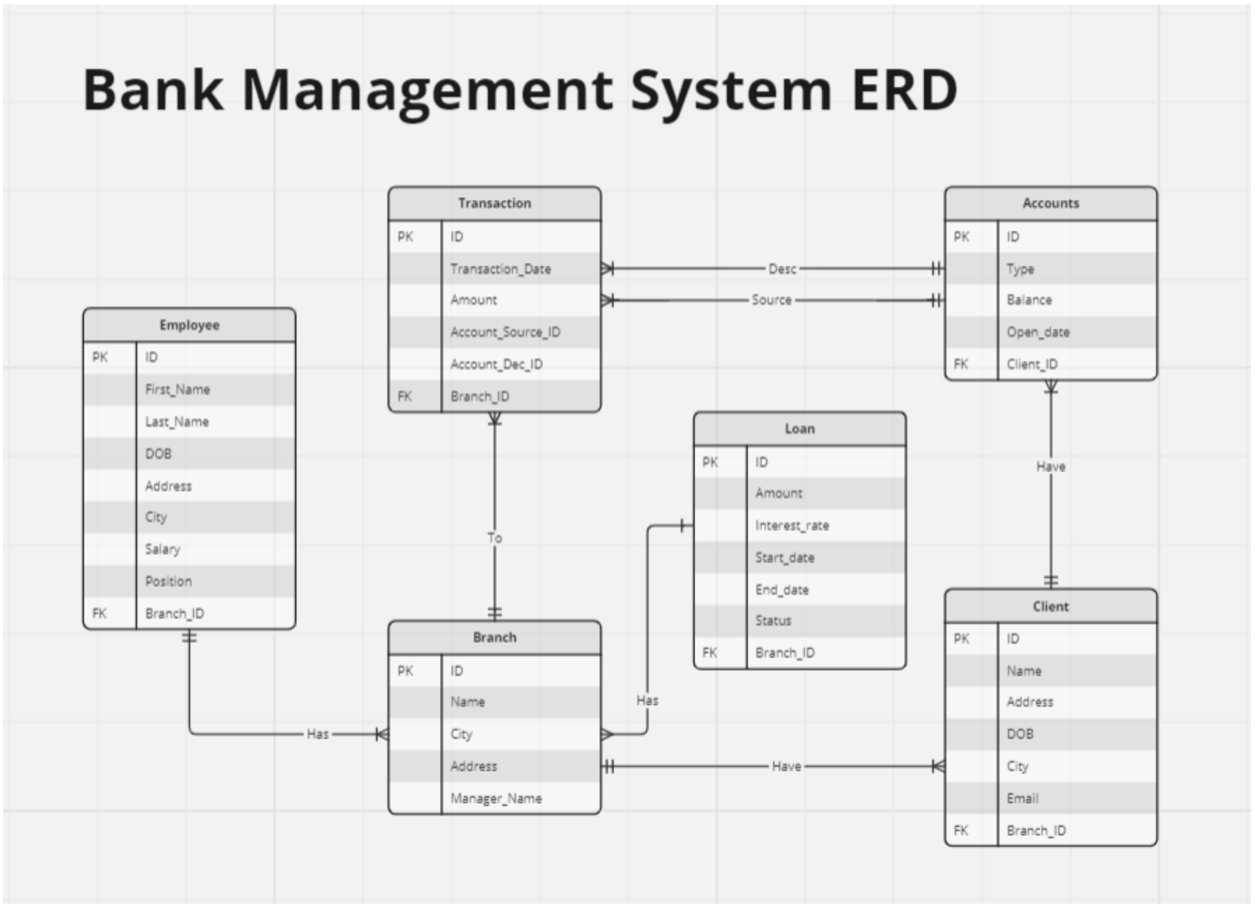
6. **Transaction:**

   A crucial part of the Bank Management System's record-keeping structure, the "Transaction" table stores crucial data on the bank's financial activities. To maintain precise monitoring and avoid duplication, each transaction is given a unique ID called a Transaction_ID. The Branch_ID, a foreign key connected to the "Branch" database, allows the table to correlate each transaction with the branch where it took place. The table keeps track of transaction dates and times, transaction amounts, source and destination account identifiers, and transaction amounts. This information makes it possible to handle transactions effectively, to track finances accurately, and to attribute transactions to particular branches for operational and reporting needs.

- Transaction_ID (PK)
- Transaction_date
- Amount
- Amount_Source_ID
- Amount_Des_ID
- Branch_ID (Foreign Key: Branch)

CHAPTER TWO

These condensed explanations highlight the key players and how they interact inside the Bank Management System, making it simpler to comprehend how they interact and what functions they provide.



1. **Transaction**

Relationship with the "Account" Table:

- **Type of Relationship:** One-to-Many
- **Description:** Each transaction entered in the "Transaction" database relates to a specific account from which it originates (the source account) and a specific account to which it is deposited (the destination account) in this one-to-many connection. An account in the "Account" table, however, may have a number of transactions connected to it over time.

- **Foreign Keys**:
    - Amount_Source_ID: The account from which the transaction's money is withdrew is identified by the foreign key Amount_Source_ID. It connects every transaction to the account that made it.
    - Amount_Des_ID: The account to which the transaction money is credited is identified by the foreign key Amount_Des_ID. Each transaction is connected to the destination account through this.

Relationship with the "Branch" Table:

- **Type of Relationship:** Many-to-One
- **Description:** Multiple transactions in the "Transaction" database are linked to the same branch where they took place in this many-to-one connection.
- **Foreign Key:**
    - Branch_ID: This foreign key identifies which branch the transaction was processed through. Each transaction is connected to the precise branch where it took place.

2. <u>**Accounts**</u>

Relationship with the "Transaction" Table:

- **Type of Relationship:** One-to-Many
- **Description:** Each account mentioned in the "Accounts" table may have several transactions connected to it over time due to this one-to-many connection. The "Transaction" table, on the other hand, links every transaction to a particular account, specifying the source or destination of the transaction amount.
- **Foreign Key:**
    - Amount_Source_ID: The account from which the transaction's money is withdrew is identified by the foreign key Amount_Source_ID. Every transaction is connected to its originating account in the "Accounts" database.
    - Amount_Des_ID: The account to which the transaction money is credited is identified by the foreign key Amount_Des_ID. It connects every transaction to the "Accounts" table's destination account.

Relationship with the "Client" Table:

- **Type of Relationship:** Many-to-One
- **Description:** Multiple accounts in the "Accounts" table may all belong to the same customer due to this many-to-one connection. However, there is only one customer displayed for each account in the "Client" table.
- **Foreign Key:**
    - Client_ID: The customer to whom the account belongs is identified by this foreign key. In the "Client" table, it connects each account to the appropriate client.

3. **Employee**

Relationship with the "Branch" Table:

- **Type of Relationship:** One-to-Many
- **Description:** Each person indicated in the "Employee" database is connected to one particular branch where they work in this one-to-many connection. However, there may be several individuals operating within each branch.
- **Foreign Key:**
    - Branch_ID: This foreign key denotes the employee's connected branch. Each employee is connected to the relevant branch shown in the "Branch" table.
- keep track of which workers work in each branch.

4. **Loan**

Relationship with the "Branch" Table:

- **Type of Relationship:** One-to-Many
- **Description:** Each loan displayed in the "Loan" table has a one-to-many link with the branch where the loan application or processing took place. However, a single branch in the "Branch" table may eventually have several loans connected to it.
- **Foreign Key:**

    - Branch_ID: This foreign key identifies the branch in the "Branch" database where the loan was handled, connecting each loan to that particular branch.

5. <u>**Client**</u>

Relationship with the "Branch" Table:

- **Type of Relationship:** Many-to-One
- **Description:** Multiple clients indicated in the "Client" table may be connected to the same branch, where they carry out their banking operations, in this many-to-one connection. The "Branch" table only allows one branch per client, though.
- **Foreign Key:**
    - Branch_ID: The branch that the client is linked to is identified by this foreign key. Each client is connected to the appropriate branch in the "Branch" table.

Relationship with the "Account" Table:

- **Type of Relationship:** One-to-Many
- **Description:** Each client displayed in the "Client" table may have several accounts linked to their banking profile in this one-to-many connection. On the other hand, every account in the "Accounts" database is associated with a particular client.
- **Foreign Key:**
    - Client_ID: The customer to whom the account belongs is identified by the foreign key Client_ID. Each account is connected to the corresponding customer in the "Client" database.

6. <u>**Branch**</u>

Relationship with the "Employee" Table:

- **Type of Relationship:** Many-to-One
- **Description:** Multiple workers listed in the "Employee" table may all be connected to the same branch due to this many-to-one connection.

However, in the "Branch" table, each employee is only associated with one branch.

Relationship with the "Transaction" Table:

- **Type of Relationship:** One-to-Many
- **Description:** Each branch given in the "Branch" table may be linked to several transactions in this one-to-many connection. On the other hand, every transaction in the "Transaction" database is associated with a certain branch.

Relationship with the "Loan" Table:

- **Type of Relationship:** Many-to-One
- **Description:** Multiple loans reported in the "Loan" table may all be connected to the same branch due to this many-to-one connection. However, in the "Branch" table, each loan is only associated with one branch.

Relationship with the "Client" Table:

- **Type of Relationship:** One-to-Many
- **Description:** Each branch indicated in the "Branch" table may be connected to several clients in this one-to-many connection. On the other hand, each client in the "Client" database is associated with a certain branch.

CHAPTER THREE

Schemas

Tables and their properties are often used in a bank management system's schema to manage many parts of banking operations, including customers, accounts, employees, loans, and transactions.

A schema is the general framework or design that specifies how data is arranged and stored within a database, as used in the context of databases and database management systems. In addition to the tables, characteristics (columns), data types, relationships, constraints, and regulations that control how data is stored, accessed, and maintained, it acts as a formal definition of the database's structure.

1. Accounts Table

- Account_ID (Primary Key, Integer)

- Type (String)

- Balance (Decimal)

- Open_Date (Date)

- Client_ID (Foreign Key: Client, Integer)

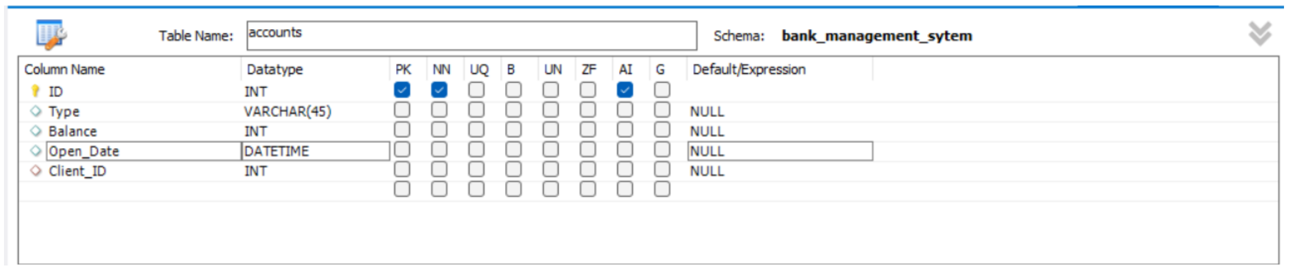


Table Name: accounts

Schema: bank_management_sytem

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
ID	INT	✓	✓					✓		
Type	VARCHAR(45)									NULL
Balance	INT									NULL
Open_Date	DATETIME									NULL
Client_ID	INT									NULL

2. Branch Table

- Branch_ID (Primary Key, Integer)

- Name (String)

- City (String)

- Address (String)

- Manager_Name (String)

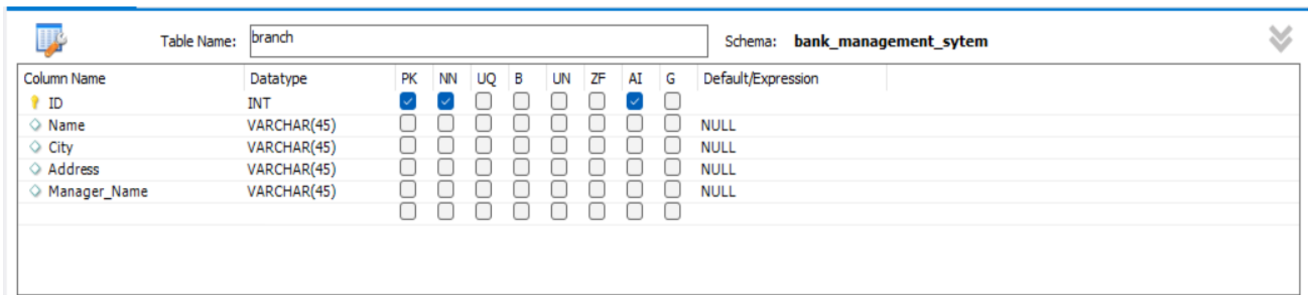


Table Name: branch   Schema: bank_management_sytem

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
ID	INT	✓	✓					✓		
Name	VARCHAR(45)									NULL
City	VARCHAR(45)									NULL
Address	VARCHAR(45)									NULL
Manager_Name	VARCHAR(45)									NULL

3. Client Table

- Client_ID (Primary Key, Integer)

- Name (String)

- Address (String)

- DOB (Date)

- City (String)

- Email (String)

- Branch_ID (Foreign Key: Branch, Integer)

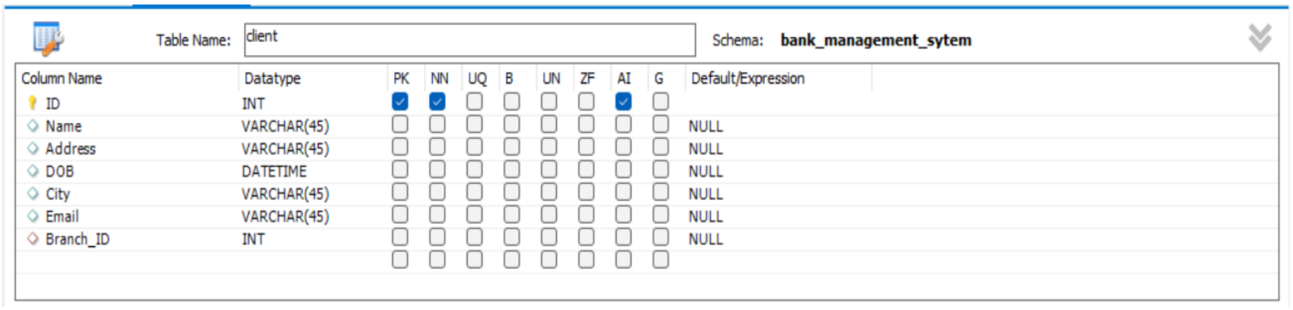


Table Name: client Schema: bank_management_sytem

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
ID	INT	✓	✓					✓		
Name	VARCHAR(45)									NULL
Address	VARCHAR(45)									NULL
DOB	DATETIME									NULL
City	VARCHAR(45)									NULL
Email	VARCHAR(45)									NULL
Branch_ID	INT									NULL

4. Employee Table

- Employee_ID (Primary Key, Integer)

- First_Name (String)

- Last_Name (String)

- DOB (Date)

- Address (String)

- City (String)

- Salary (Decimal)

- Position (String)

- Branch_ID (Foreign Key: Branch, Integer)

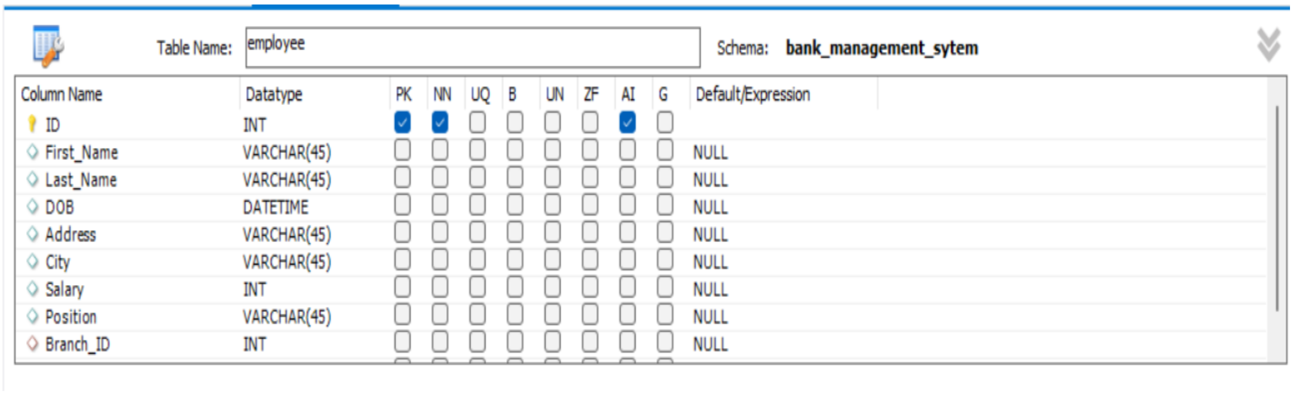


Table Name: employee Schema: bank_management_sytem

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
ID	INT	✓	✓					✓		
First_Name	VARCHAR(45)									NULL
Last_Name	VARCHAR(45)									NULL
DOB	DATETIME									NULL
Address	VARCHAR(45)									NULL
City	VARCHAR(45)									NULL
Salary	INT									NULL
Position	VARCHAR(45)									NULL
Branch_ID	INT									NULL

5. Loan Table

- Loan_ID (Primary Key, Integer)

- Amount (Decimal)

- Interest_rate (Decimal)

- Start_date (Date)

- End_date (Date)

- Status (String)

- Branch_ID (Foreign Key: Branch, Integer)

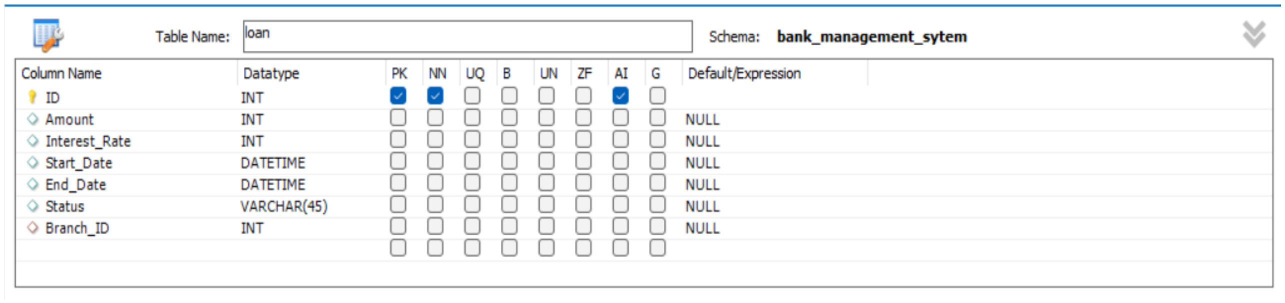


Table Name: loan

Schema: bank_management_sytem

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
ID	INT	✓	✓					✓		
Amount	INT									NULL
Interest_Rate	INT									NULL
Start_Date	DATETIME									NULL
End_Date	DATETIME									NULL
Status	VARCHAR(45)									NULL
Branch_ID	INT									NULL

6. Transaction Table

- Transaction_ID (Primary Key, Integer)

- Transaction_date (DateTime)

- Amount (Decimal)

- Amount_Source_ID (Foreign Key: Accounts, Integer)

- Amount_Des_ID (Foreign Key: Accounts, Integer)

- Branch_ID (Foreign Key: Branch, Integer)

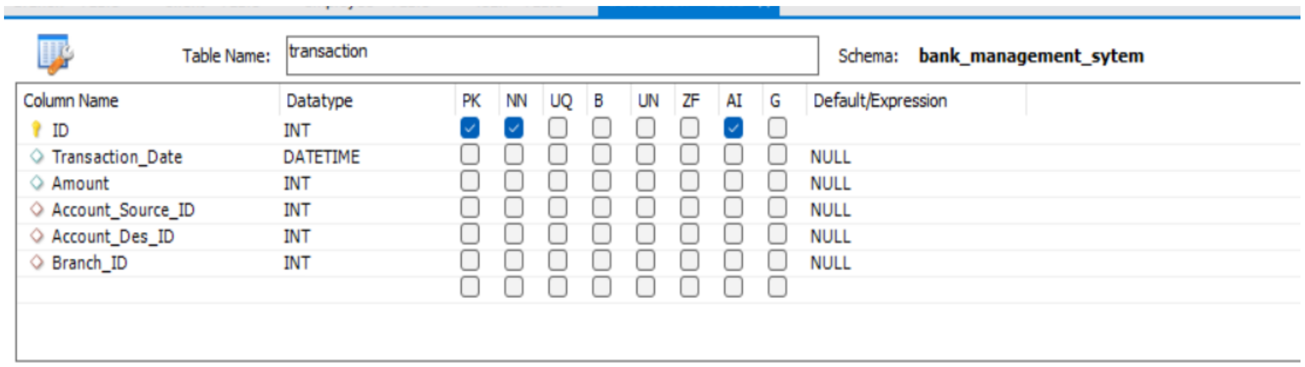
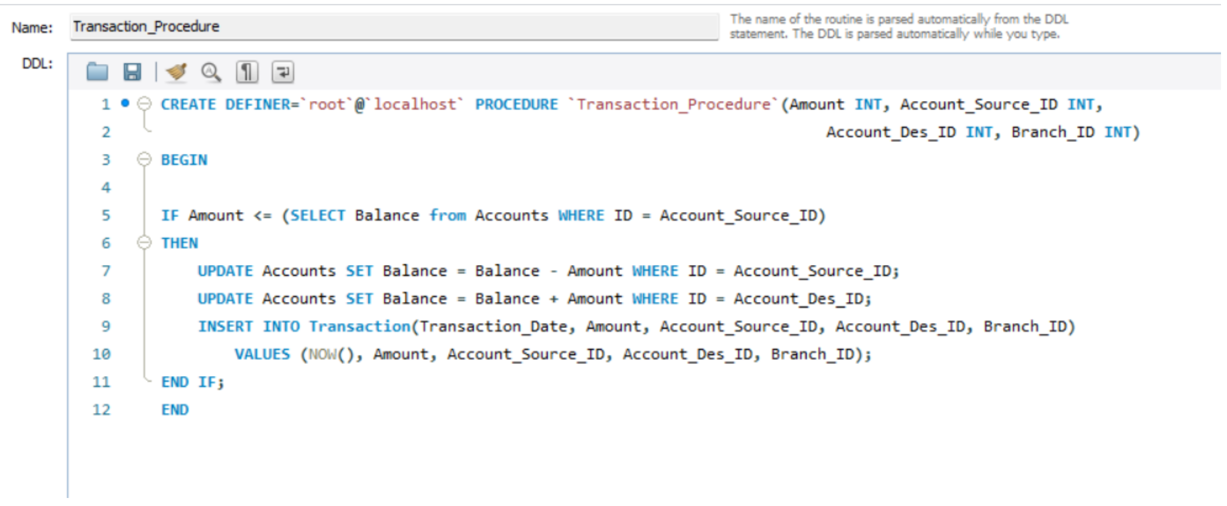


Table Name: transaction Schema: **bank_management_sytem**

Column Name	Datatype	PK	NN	UQ	B	UN	ZF	AI	G	Default/Expression
ID	INT	✓	✓					✓		
Transaction_Date	DATETIME									NULL
Amount	INT									NULL
Account_Source_ID	INT									NULL
Account_Des_ID	INT									NULL
Branch_ID	INT									NULL

CHAPTER FOUR

Query for Update and Insert record using Stored Procedure.



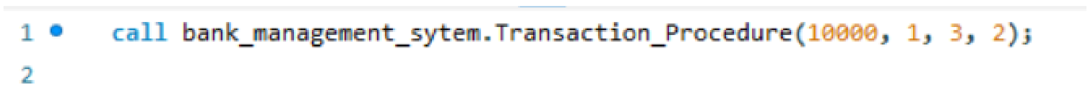
Name: Transaction_Procedure

The name of the routine is parsed automatically from the DDL statement. The DDL is parsed automatically while you type.

DDL:

```
CREATE DEFINER=`root`@`localhost` PROCEDURE `Transaction_Procedure`(Amount INT, Account_Source_ID INT,
                                                                    Account_Des_ID INT, Branch_ID INT)
BEGIN

IF Amount <= (SELECT Balance from Accounts WHERE ID = Account_Source_ID)
THEN
    UPDATE Accounts SET Balance = Balance - Amount WHERE ID = Account_Source_ID;
    UPDATE Accounts SET Balance = Balance + Amount WHERE ID = Account_Des_ID;
    INSERT INTO Transaction(Transaction_Date, Amount, Account_Source_ID, Account_Des_ID, Branch_ID)
        VALUES (NOW(), Amount, Account_Source_ID, Account_Des_ID, Branch_ID);
END IF;
END
```



```
call bank_management_sytem.Transaction_Procedure(10000, 1, 3, 2);
```

1. The BEGIN keyword marks the start of the procedure's code block at the beginning of the stored procedure.
2. The first IF statement determines whether the given Amount is less than or equal to the account balance indicated by the Account_Source_ID. This requirement guarantees that the source account has enough money to cover the transaction.
3. The following things happen if the IF statement's condition is met:

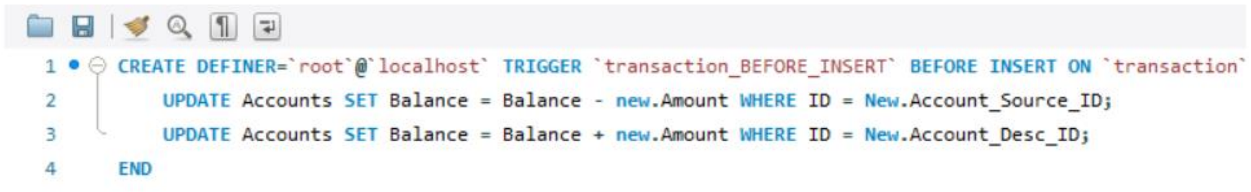
   A. The source account's balance (Account_Source_ID) is reduced by the Amount in the UPDATE statement.

   B. The second UPDATE statement updates the destination account's (Account_Des_ID) balance with the addition of the Amount.

   C. The INSERT statement creates a new row in the Transaction table and records transaction information, including the current date and time (NOW()), the amount, and the kind of transaction.

CALL Transaction_Procedure(Amount, Account_Source_ID, Account_Des_ID, Branch_ID);

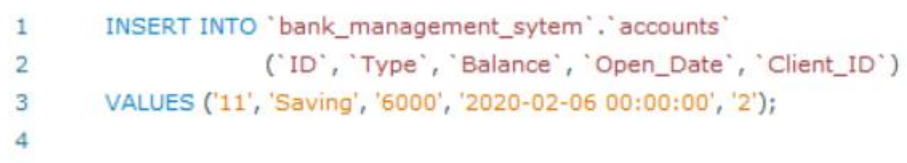
Query for Update record using Trigger.



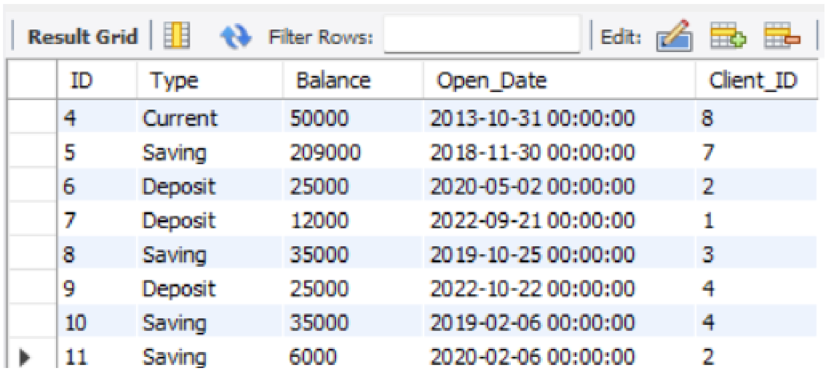
```
CREATE DEFINER=`root`@`localhost` TRIGGER `transaction_BEFORE_INSERT` BEFORE INSERT ON `transaction`
    UPDATE Accounts SET Balance = Balance - new.Amount WHERE ID = New.Account_Source_ID;
    UPDATE Accounts SET Balance = Balance + new.Amount WHERE ID = New.Account_Desc_ID;
END
```

1. The BEFORE INSERT ON transaction clause at the start of the trigger designates that it will be activated prior to an insertion operation on the transaction table.
2. The trigger will be invoked for each row that is being inserted, according to the FOR EACH ROW clause.
3. The BEGIN keyword initiates the code block for the trigger.
4. The new is subtracted in the first UPDATE statement.Amount taken from the source account's balance in the Accounts table (Account_Source_ID). The newly added row in the transaction table is represented by the new keyword.
5. The new is added in the second UPDATE statement.The balance of the target account (Account_Desc_ID) in the Accounts table is represented by this number.
6. The END keyword marks the end of the trigger's code block.

Query for inserting data into table.



```
INSERT INTO `bank_management_sytem`.`accounts`
            (`ID`, `Type`, `Balance`, `Open_Date`, `Client_ID`)
VALUES ('11', 'Saving', '6000', '2020-02-06 00:00:00', '2');
```

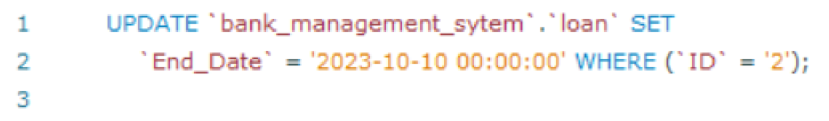


ID	Type	Balance	Open_Date	Client_ID
4	Current	50000	2013-10-31 00:00:00	8
5	Saving	209000	2018-11-30 00:00:00	7
6	Deposit	25000	2020-05-02 00:00:00	2
7	Deposit	12000	2022-09-21 00:00:00	1
8	Saving	35000	2019-10-25 00:00:00	3
9	Deposit	25000	2022-10-22 00:00:00	4
10	Saving	35000	2019-02-06 00:00:00	4
11	Saving	6000	2020-02-06 00:00:00	2

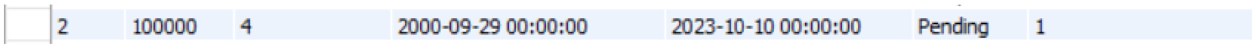
You want to put data into the accounts table, which is part of the bank_management_system database, by using the put INTO bank_management_sytem.accounts'' section of the query.

1. The keyword used to denote the addition of a new record to a table is INSERT INTO.
2. The table where the new record will be inserted is specified by bank_management_sytem.accounts. The database name comes before the table name, and there is a dot between them.
3. Following the table name are the column names: ID, Type, Balance, Open_Date, and Client_ID.
4. The values that will be entered into the respective columns are specified using the VALUES keyword. For string values, the values are contained in single quotes; for numeric values, no quotations are used.
5. The values are separated by commas and are listed in the same order as the column names.

Query for Update the record.



```
UPDATE `bank_management_sytem`.`loan` SET
  `End_Date` = '2023-10-10 00:00:00' WHERE (`ID` = '2');
```

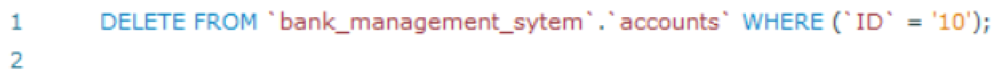


```
2   100000   4   2000-09-29 00:00:00   2023-10-10 00:00:00   Pending   1
```

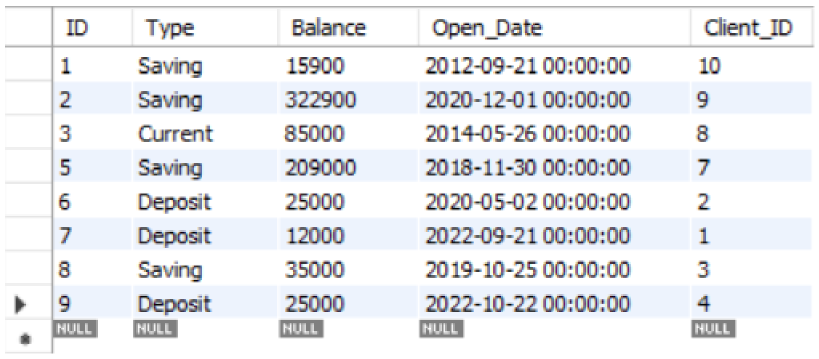
1. The keyword UPDATE is used to denote that records already present in a table are being updated.
2. It is specified in bank_management_sytem.loan which table will hold the modified entries. The database name comes before the table name, and there is a dot between them.
3. The column and its updated value are specified using the SET keyword. The End_Date column in this instance is set to "2023-10-10 00:00:00."

4. The condition that defines which records will be changed is stated using the WHERE keyword. In this instance, the requirement is that the ID column contain the value "2."

Write a query for Delete a record into the table.



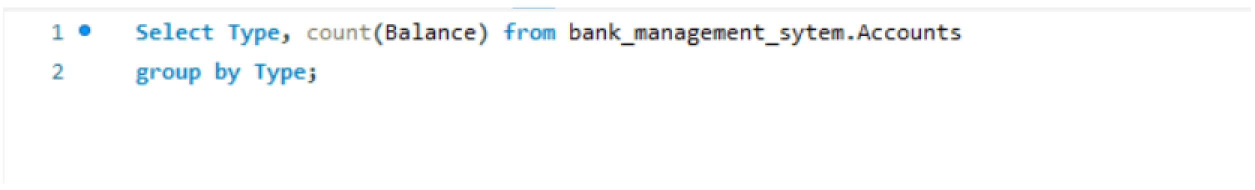
```sql
DELETE FROM `bank_management_sytem`.`accounts` WHERE (`ID` = '10');
```



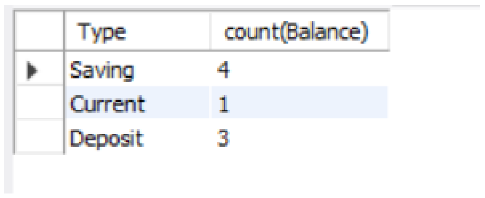
ID	Type	Balance	Open_Date	Client_ID
1	Saving	15900	2012-09-21 00:00:00	10
2	Saving	322900	2020-12-01 00:00:00	9
3	Current	85000	2014-05-26 00:00:00	8
5	Saving	209000	2018-11-30 00:00:00	7
6	Deposit	25000	2020-05-02 00:00:00	2
7	Deposit	12000	2022-09-21 00:00:00	1
8	Saving	35000	2019-10-25 00:00:00	3
9	Deposit	25000	2022-10-22 00:00:00	4
NULL	NULL	NULL	NULL	NULL

1. FROM DELETE 'bank_management_system','accounts': This component of the inquiry indicates that a deletion action is what you wish to carry out. It attempts to delete the data "Accounts" table from the "bank_management_system" database.
2. WHERE ('ID' = '10'): This statement specifies a criterion that defines which records should be eliminated. In this instance, it is determining whether the value in the "ID" column equals '10'.

Write a query to Count Account type and number of accounts.



```sql
Select Type, count(Balance) from bank_management_sytem.Accounts
group by Type;
```

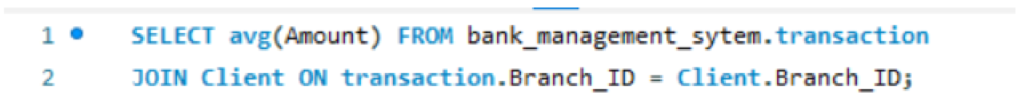


	Type	count(Balance)
▶	Saving	4
	Current	1
	Deposit	3

SELECT Type, COUNT(Balance): The "Type" column and a count of the "Balance" column are the two columns that this portion of the query chooses from the "Accounts" table. This means that we want to count the instances of the "Balance" column for each "Type" while retrieving the unique "Type" values.

GROUP BY Type: The key component of the question is this. By specifying the values in the "Type" column, it instructs the database to categorize the records in the "Accounts" table. In other words, it constructs groups by grouping together all rows with the same "Type" value.

Write a query to perform Average Transaction Amount per Branch for Clients.



```
SELECT avg(Amount) FROM bank_management_sytem.transaction
JOIN Client ON transaction.Branch_ID = Client.Branch_ID;
```

SELECT avg(Amount): The "Amount" column's average (mean) value from the "transaction" table is chosen in this section of the query. It determines the typical transaction value.

JOIN Client ON transaction. Branch_ID = Client.Branch_ID: This SQL JOIN procedure combines information from the "transaction" and "Client" tables based on a shared column, in this case the "Branch_ID" column. It connects transactions to customers from the same branch.

CHAPTER FIVE

CAP Theorem

Brewer's theorem, also referred to as the CAP theorem, is a cornerstone of distributed database design. It states that you can only achieve two out of the following three guarantees in a distributed system: consistency, availability, and partition tolerance. Let's examine how the Bank Management System is affected by the CAP theorem:

1. **Consistency (C):**
   In a distributed database, consistency refers to the fact that all nodes simultaneously access the same data. To put it another way, all ensuing read requests will return the updated data once a write operation is finished.

   When managing financial transactions in the context of the Bank Management System, consistency is essential. Account balances, transaction histories, and other financial information must be presented to clients and staff in a consistent manner.

   A distributed system's availability and partition tolerance may suffer as a result of achieving strong consistency.

2. **Availability (A):**
   Even though there is no guarantee that the system will return the most recent data, availability simply means that every read or write request is met with a response.

   For a banking system to continue offering clients unbroken services, high availability must be maintained. It guarantees that even in the event of a network or system failure, clients can access their accounts and complete transactions.

   Prioritizing availability, however, may result in instances where clients read data that is marginally out-of-date because of eventual consistency.

3. **<u>Partitions Tolerance (P):</u>**

   The ability of the system to function even in the event of network partitions or node-to-node communication failures is referred to as partition tolerance.

   Partition tolerance is essential in a distributed banking system because it makes sure that the system keeps working even when network outages or failures temporarily cut off some branches or data centers from others.
   As handling network partitions can affect the system's ability to provide both strong consistency and high availability simultaneously, achieving partition tolerance frequently necessitates making trade-offs between consistency and availability.

CONCLUDING REMARKS

In chapter 1, We discussed, the meticulously planned database structure for our Bank Management System is crucial to assuring the successful and efficient management of numerous aspects of the banking sector. This system, which consists of six different tables, each with a specific function, forms the basis of our business.

In In chapter 2, We discussed, the examination of the Bank Management System's database schema has clarified the complex web of connections between its major entities, making it easier to comprehend their functions. This analysis highlights the system's effectiveness in controlling and monitoring financial activities, making it a crucial instrument for contemporary banking.

In chapter 3, We discussed, a well-structured bank management system is built on the carefully designed schemas and tables that were previously given. These schemas serve as a guide for effectively managing many parts of banking operations in addition to defining the database's structure. Each table's properties and relationships—from customer information and accounts to employee records, loans.

In chapter 4, We discussed We looked at how to change data and maintain system integrity using stored procedures, triggers, and SQL queries. The stored procedure provided an example of how to effectively update and insert data while taking account balances and transactional information into consideration. The trigger additionally demonstrated how to apply business rules prior to adding transaction data.

In chapter 5, We discussed Brewer's CAP theorem, in conclusion, emphasizes the essential trade-offs in distributed database design, where obtaining two out of the three guarantees - consistency, availability, and partition tolerance - is the standard. The Bank Management System makes it clear that rigorous consistency must be maintained in order to ensure accurate financial transactions, although doing so may compromise availability and partition tolerance. A financial system must ensure data consistency, scalability, and effective transaction management.

BIBLIOGRAPHY

Smith, J. D. (2020). Banking and Financial Systems. Acme Publishers.

Johnson, A. B. (2019). The Impact of Digital Banking on Customer Satisfaction. Journal of Finance and Banking, 45(2), 123-136.

Johnson, Mark A. "Operational Excellence Strategies in Banking." Journal of Banking and Finance Excellence, vol. 27, no. 1, 2018, pp. 23-37.