What is a Database?

A Database Management System (DBMS) is software that enables users to create, manage, and manipulate databases. It provides a way for organizations to store, organize, and retrieve large amounts of data quickly and efficiently. In essence, a DBMS is a software application that acts as an interface between the database and the users or applications that need to access the data.

Type: Hierarchical, Network, Relational, NoSQL

In DBMS, hierarchical, network, relational, and NoSQL databases represent different approaches to organizing and managing data.

Hierarchical databases use a tree structure with parent-child relationships.

Network databases are more flexible, allowing for many-to-many relationships.

Relational databases store data in tables with rows and columns, using SQL for querying. NoSQL databases offer flexible, schema-less structures for handling large volumes of unstructured data.

Hierarchical Database:

Structure: Organizes data in a tree-like structure with a single root node and branches.

Relationships: Primarily supports one-to-many relationships (one parent node can have multiple child nodes).

Use Cases: Suitable for scenarios with clear hierarchies, like organizational structures, file systems, or taxonomies.

Flexibility: Less flexible than other models, especially for complex relationships.

Network Database:

Structure: Uses a graph-like structure, allowing data to have multiple connections.

Relationships: Supports one-to-many and many-to-many relationships, offering more flexibility than hierarchical databases.

Use Cases: Well-suited for complex environments with intricate relationships, like social networks or airline reservation systems.

Complexity: Can be more complex to design and manage than hierarchical databases.

Relational Database (RDBMS):

· **Structure:** Organizes data in tables with rows and columns.

· **Relationships:** Uses SQL for querying and managing relationships between tables. · **Use Cases:** Commonly used in applications like CRM, financial systems, and e-commerce.

· **Advantages:** Offers data integrity through normalization, data consistency, and ACID properties.

NoSQL Database:

· **Structure:**

· Offers flexible, schema-less structures, meaning they don't require a predefined schema like RDBMS.

· **Data Types:** Supports various data types, including document-oriented, key-value, wide column, and graph databases.

· **Use Cases:** Ideal for big data, real-time applications, and scenarios where flexibility and scalability are crucial.

· **Scalability:** Often designed for horizontal scaling, making them well-suited for handling large volumes of data

DBMS Vs RDBMS

A DBMS (Database Management System) is a software application that manages and organizes data within a database, while an RDBMS (Relational Database Management System) is a specific type of DBMS that uses a relational model to structure data. RDBMS stores data in tables with rows and columns, allowing for relationships between tables through keys, while DBMS can store data in various formats like hierarchical, network, or object-oriented.

Here's a more detailed breakdown:

DBMS (Database Management System):

· **Definition:** A software system that helps manage and organize data within a database. · **Data Structure:** Can store data in different formats, including hierarchical, network, or object-oriented models.

· **Relationships:** May not enforce relationships between tables as tightly as RDBMS. · **Example:** Windows Registry, Microsoft Access, and XML can be considered examples of DBMS.

· **Use cases:** Smaller organizations with less data or applications that require less complex data organization might use a DBMS.

RDBMS (Relational Database Management System):

· **Definition:** A specific type of DBMS that uses the relational model for data organization. · **Data Structure:** Stores data in tables with rows and columns, enabling relationships between tables using keys (primary and foreign keys).

· **Relationships:** Enforces relationships between tables through the use of foreign keys, allowing for data consistency and integrity.

· **Query Language:** Typically uses SQL (Structured Query Language) for data manipulation.

· **Example:** MySQL, PostgreSQL, Microsoft SQL Server, and Oracle Database are examples of RDBMS.

· **Use cases:** Larger organizations with complex data, transactional systems, and applications requiring robust data management are well-suited for RDBMS. **Key Differences**:

Feature DBMS RDBMS
Data Structure Hierarchical, network, object-oriented Tabular (rows and columns) Relationships

May not enforce relationships Enforces relationships (keys)

Query Language Varies SQL (Structured Query Language) Scalability Can be limited Highly scalable

Complexity Simpler More complex Data Consistency May vary Enforced through constraints. Use

Cases: Smaller organizations, simpler data Larger organizations, complex data

Three Level Architecture

The **Three-Schema Architecture** is a well-structured framework for **database management systems (DBMS)** that separates the database into three distinct layers. This separation enhances flexibility, abstraction, and independence, making it easier to manage and modify the database. The architecture, proposed by the **ANSI/SPARC** committee, serves as a guide for designing and interacting with databases. Let's explore its components, advantages, and practical implications.

Overview of the Three-Schema Architecture

The three layers of this architecture are:

1. **External Schema** (View Level)
2. **Conceptual Schema** (Logical Level)
3. **Internal Schema** (Physical Level)

Each layer is independent of the others, ensuring data abstraction and separation of concerns. **1.**

External Schema (View Level)

· **What it does**: Provides tailored views of the database to different users or user groups. · **Purpose**: Offers customized access while hiding irrelevant details.

· **Example**:A customer sees their account balance, while a bank manager accesses detailed financial reports.

Characteristics:

· Multiple external schemas can exist.

· Security and privacy are ensured by restricting access to specific data.

2. Conceptual Schema (Logical Level)

· **What it does**: Represents the logical structure of the entire database.

· **Purpose**: Provides a unified view of the data for all external schemas, independent of how data is physically stored.

· **Example**: Defines entities like Customers and Accounts with attributes and relationships.

Characteristics:

· Hides physical storage details.

· Maintains data integrity and enforces constraints.

· Focuses on logical design and relationships.

3. Internal Schema (Physical Level)

· **What it does**: Deals with the physical storage of data on hardware.

· **Purpose**: Optimizes storage, retrieval, and performance.

· **Example**: Specifies file structures, indexes, and storage strategies.

Characteristics:

· Determines how data is physically stored in the database.

· Manages performance aspects like data compression and indexing.

Illustration of the Three Layers

Real-World Analogy

Think of the architecture as a **library system**:

· **External Schema**: A user interacts with a search catalog, seeing only books in their area of interest (e.g., fiction or non-fiction).

· **Conceptual Schema**: The librarian knows the entire catalog, including all books, authors, and classifications.

· **Internal Schema**: The library storage system determines how books are physically arranged (e.g., by ISBN or shelf number).

Advantages of the Three-Schema Architecture

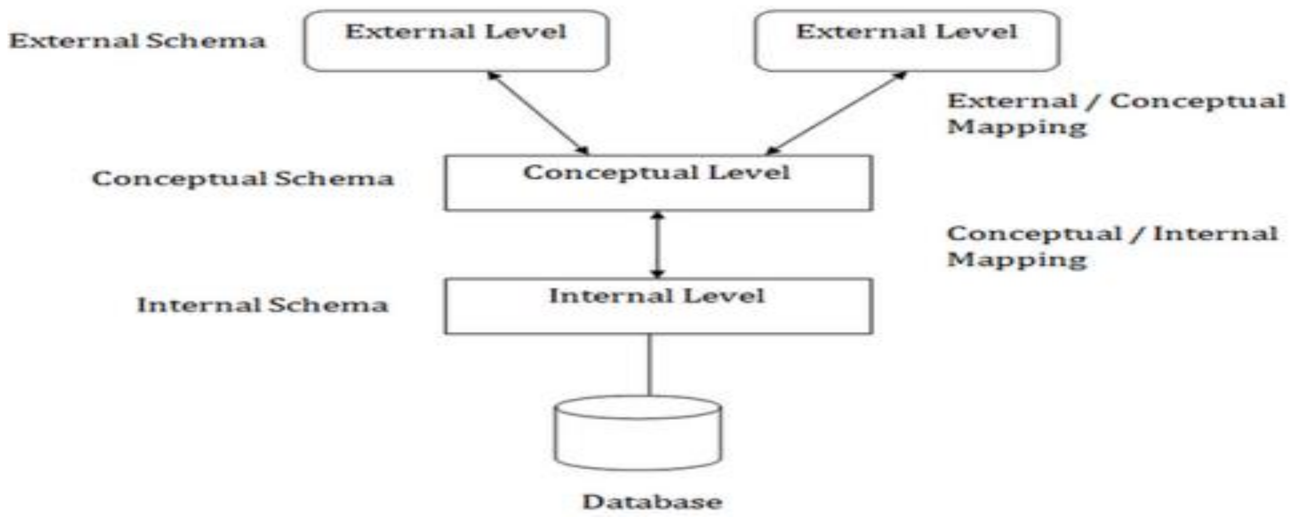
· **Data Independence**: Data independence is a key benefit of this architecture · **Logical Independence**: Changes to the conceptual schema do not affect external schemas.

Changes to the logical structure (e.g., adding a new entity) don't disrupt external views. Example: Adding a new column Email in the Customer table won't affect the customer's view of their account balance.

· **Physical Independence**: Changes to the internal schema do not affect the conceptual schema.

Changes to physical storage (e.g., reorganizing files) don't impact the logical schema. Example: Shifting from one storage format to another doesn't affect queries. · **Flexibility**: Multiple external schemas allow customized views for different users. · **Security**: Sensitive data is protected by limiting access at the external level. · **Maintainability**: Modular design simplifies updates and system upgrades.

Diagram of the Three-Schema Architecture



Use Cases of the Three-Schema Architecture

1. **Database Development**: Ensures a clear separation between user requirements and physical implementation.
2. **Data Security**: Restricts user access to only relevant information.
3. **Scalability**: Facilitates system upgrades without affecting users.

Limitations of the Three-Schema Architecture

1. **Complex Implementation**: Requires careful design to ensure independence between layers. 2. **Overhead**: Abstraction layers may introduce additional complexity in querying and system management.

Component of DBMS and Data independence

Data Independence Levels:

· **Physical Data Independence:** Enables changes to the physical storage of the data without impacting the logical structure or user views. For example, migrating to a

different storage medium like from tape to disk does not necessitate changes to the application.

· **Logical Data Independence:** Allows changes to the logical structure of the database (like adding or deleting columns) without affecting the application programs or user views. For example, renaming a table or adding a new attribute can be done without altering the application.

DBMS Components:

A typical DBMS includes:

· **Hardware:** Physical components like servers, storage devices, and networking equipment.

· **Software:** The DBMS software itself, which manages the database, handles queries, and enforces security.

· **Data:** The actual data stored in the database, including tables, indexes, and views. · **Procedures:** Stored procedures, functions, and triggers that automate database operations.

· **Database Access Language:** Languages like SQL used to interact with the database. · **Users:**

Individuals who interact with the DBMS, including database administrators and application users

How to Design a Database for Online Banking System

Designing a **relational database** for an **online banking system** is an important and interesting task that requires careful **planning** and consideration. The database serves as the backbone of the **banking system** that **stores** and **organizes** large amounts of financial data **securely**.

In this guide, we will explore **How to Design a Relational Database for an Online Banking System** by designing the **entity relationship** diagram and **Database Model** with the help of various required **entities**, **attributes,** and **relationships**.

Database Design for Online Banking System

The database for the **Online Banking System** must efficiently manage **Account Details**, **Customer Details,** and **Transactions**. **Users** should be able to log in to the system and perform banking operations like **balance check**, **deposit**, **withdraw** and **money transfer**. Let's understand the overview of the **Online Banking System** through the below terms.

1. **User Management:**

· We should manage user accounts which include registration, login and authentication. · Store user details such as name, email, address, and contact information securely.

2. **Account Management:**

· Maintain records of bank accounts associated with each user.

· Track account balances, transaction history, and account status (active, closed, etc.).

3. **Transaction Processing:**

· Handle various types of transactions, including deposits, withdrawals, transfers, and bill payments.

· Ensure transactions are secure, accurate, and recorded in real time.

4. **Security Features:**

· Implement robust security measures to protect sensitive data and prevent unauthorized access.

· Use encryption techniques to secure data transmission over the network. 5.

Reporting and Analytics:

· Generate reports on account activity, transaction history, and user behavior. · Use analytics to identify trends, detect fraud, and improve service offerings.

6. **Customer Support:**

· Provide customer support features, such as chat support, FAQs, and helpdesk services. · Maintain records of customer interactions and feedback for improving services.

Online Banking System Features

· **User Login**: The user login feature is a important component of online banking systems by providing a secure gateway for users to access their accounts. Users must authenticate themselves with a valid **user ID** and **password** and sometimes additional security measures like OTP or biometric authentication are used for added security.

· **Check Balance:** After logging in the users can check the available balance in their bank accounts. This feature provides users with **real-time** information about their finances and helping them to manage their funds more effectively.

· **Send Money:** Online banking systems allow users to transfer money from their account to another valid account. Users need to provide the **recipients account number** and other necessary details to complete the **transaction** securely.

· **Add Beneficiary:** To make fund transfers easier and faster users can add beneficiaries to their account. Adding a beneficiary requires providing the recipient's account details and verifying the **relationship**, ensuring that the transfer is **authorized**.

· **Receive Money:** When a user receives money into their account, the transaction is recorded and the account balance is updated accordingly. This feature allows users to receive funds from other users or **external** sources.

· **Transaction History**: Users can view their transaction history, which includes details such as the **amount**, **time** and type of **transaction** like **deposit**, **withdrawal**, fund transfer. This feature helps users keep track of their financial activities and monitor their spending patterns.

Entities and Attributes of Online Banking System

1. Customer

· **CustomerID (primary key):** Unique identifier for each customer
· **Name:** Name of the customer.
· **Address:** Address of the customer.
· **Contact**: Contact details of the customer (e.g.: Phone number).
· **Username (Unique) :** username to login to the banking system.
· **Password:** Encrypted password of the user.

2. Account

· **AccountID (primary key):** Unique identifier of account.
· **CustomerID (foreign key referencing Customer):** Identifier of the customer who owns the account.
· **Type:** Defines the account type whether savings, current etc.
· **Balance:** Amount of money available in the account.

3. Transaction

· **TransactionID (primary key):** Unique identifier of the transaction, automatically increments.
· **AccountID (foreign key referencing Account):** Identifies the account where the transaction took place.
· **Type**: Defines the transaction type i.e. deposit or withdrawal.
· **Amount**: Shows the balance amount used for the transaction.
· **Timestamp:** The date and time of the transaction.

4. Beneficiary

· **BeneficiaryID (primary key):** Unique identifier of the beneficiary.
· **CustomerID(foreign key referencing Customer):** Identifies the customer who added the beneficiary.
· **Name:** Name of the beneficiary.
· **AccountNumber:** Account Number of the beneficiary.
· **BankDetails:** Bank Name of the Beneficiary added.

Relationships Between These Entities

Customer - Account relationship

· **One to Many Relationship:** One customer is allowed to create many accounts. · **Foreign Key:** CustomerID in Account Table is referencing CustomerID in Customer Table.

Customer - Beneficiary Relationship

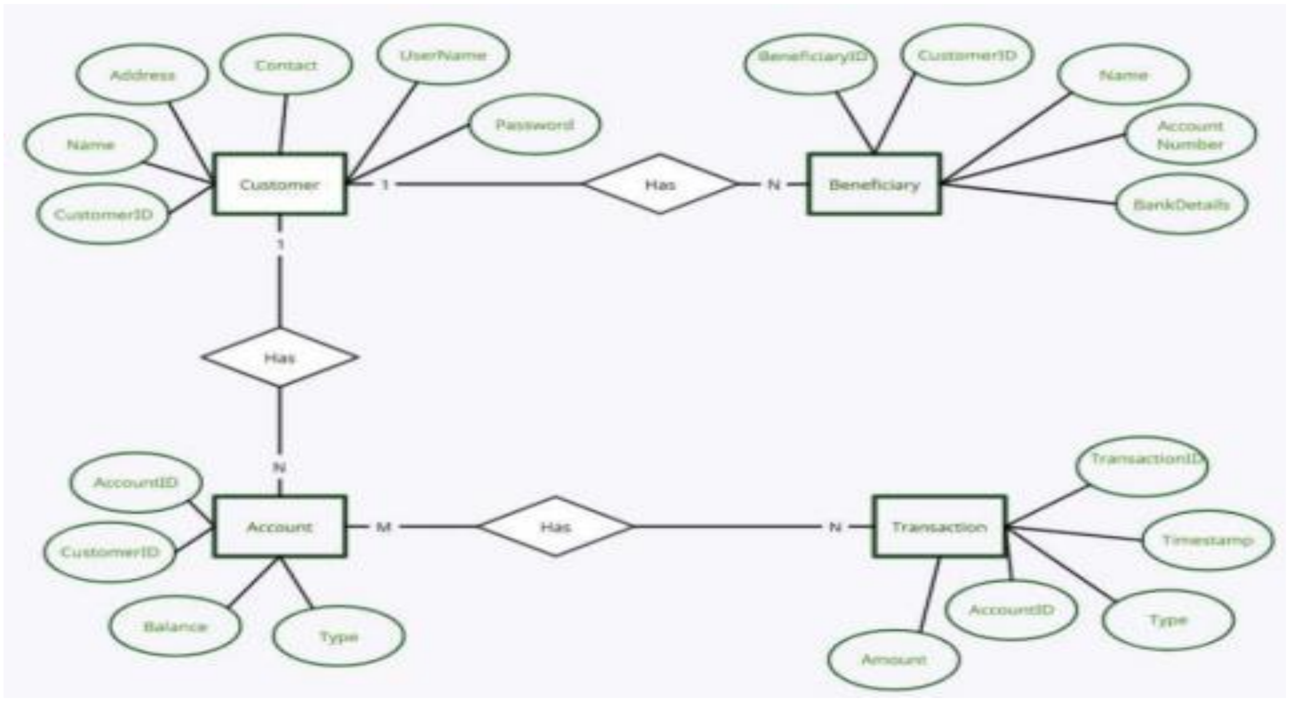
· **One to Many Relationship:** A customer can add multiple beneficiaries. · **Foreign Key:** CustomerID in Beneficiary is referencing to CustomerID in Customer Table.

Account - Transaction Relationship

· **Many to Many Relationship:** An account can have multiple transactions and a transaction can involve multiple account numbers.

· **Foreign Key:** AccountID in Transaction is referencing to AccountID in Account Table.

Representation of ER Diagram



ER Diagram for Bank Database

An **"entity-relationship diagram"** is a kind of flowchart of a database that helps us to analyze the requirements and design of the database. It conveys the relationship between several entities of a specified system and their attributes.

It is a basic diagrammatic structure to represent a database and is considered good to start with an ER diagram before implementing the database system.

In this ER diagram of the bank database, we have eight entities,

1. **Customer**, to represent the customers.

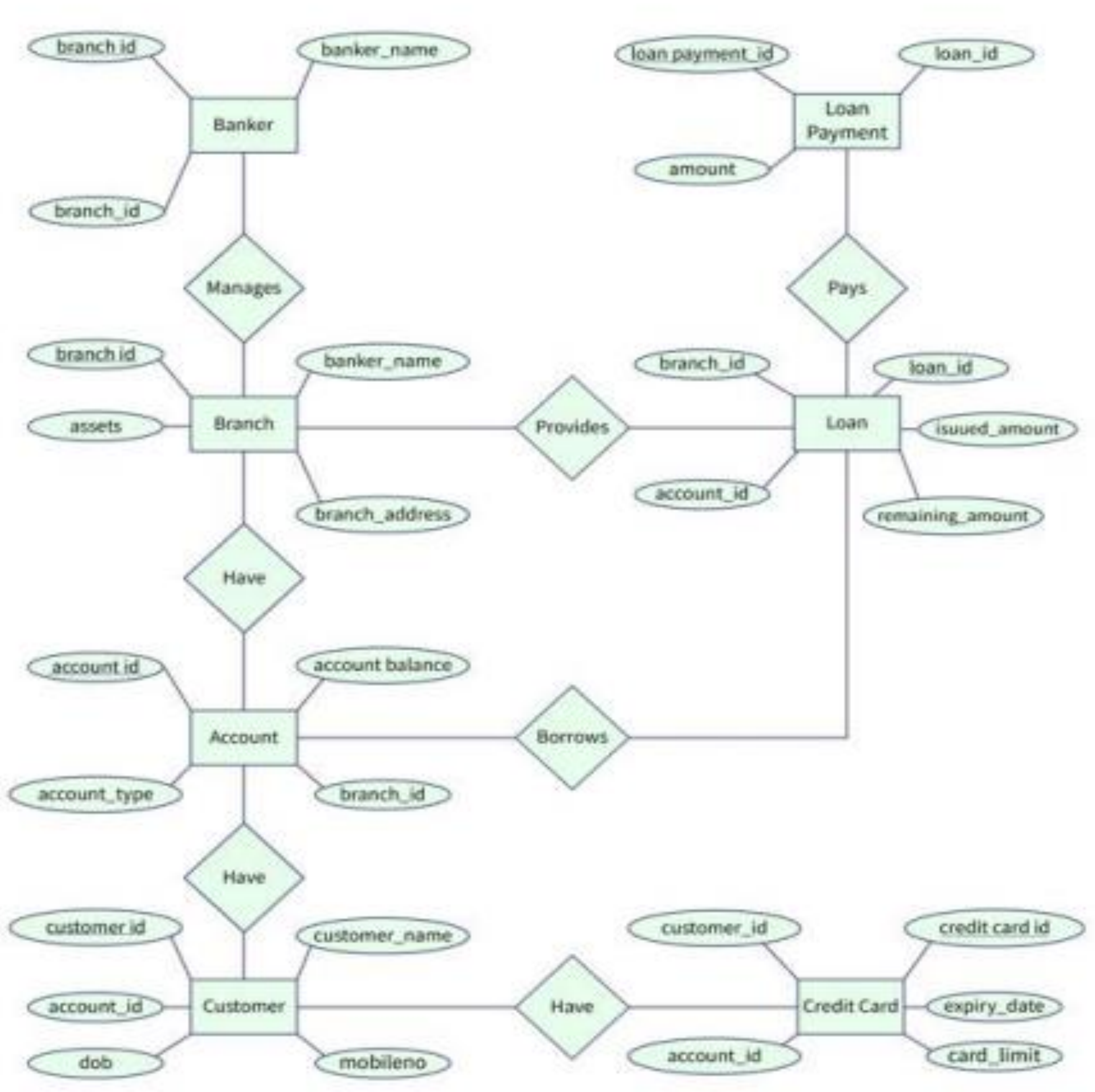
2. **Banker**, to represent the Banker, who manages the entire branch. 3. **Branch**, to represent a branch of a bank.

4. **Loan**, To represent the loan granted by the branch to the customer's account. 5. **Account**: To represent the bank account of any customer.

6. **Transaction**, to represent the transactions of customers for any account. 7. **Credit Card**, To represent the Credit card of any associated customer and account.

8. **Loan Payment**, To represent the Payment towards the loan.

ER Diagram consists of some shapes which have their significance i.e., Rectangles are used to represent the entities, Rhombus is used to represent the association between the entities, and Oval represents the attributes of an entity.



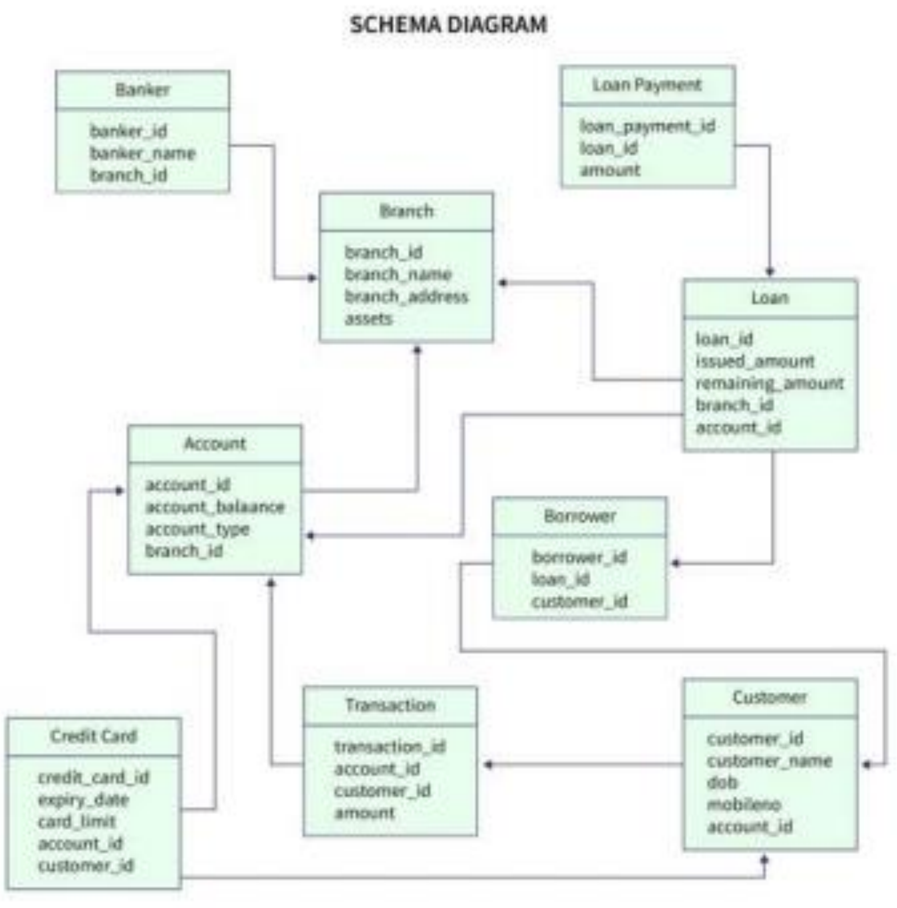
Schemas

The schema is an analytical layout or blueprint of the database that represents the logical view, like how the tables will look in the actual implementation of the database.

The schema diagram also represents the entity and its relationships but in the form of tables.

Below, a schema diagram is shown for the banking system. The arrow out of the rectangular box represents the relationship between them.

For example, there is an arrow from Banker to Branch which means there is a property branch_id which refers to the Branch.



The **Schema Diagram** is the more programmatic and technical structure that is used by the database developer, but the ER Diagram is a much more basic and non-technical perspective that is used by the business analyst, etc. The purpose of both is to logically represent a Database.

Overall the schema conveys the description of the database. The following schemas will be designed for the banking database, We will perform sql queries for the banking database after visualizing the Schemas; it will help us to understand the system in a better way.

Branch Schema

It will represent the branches of a bank.

Field	**Type**
branch_id	INT
assets	INT

branch_name	TEXT
branch_address	TEXT

· **branch_id :**
It is an id given to each branch to identify them uniquely.
· **branch_name :**
The name of the branch.
· **branch_address :**
The address of that particular branch.
· **assets :**
The total assets of that branch.

Banker Info Schema

This schema represents which banker is managing any particular branch.

Field	**Type**
banker_id	INT
banker_name	TEXT
branch_id	INT

· **banker_id :**
It is an id to uniquely identify each banker.
· **banker_name :**
The name of banker.
· **branch_id :**
The reference id of the branch which is being managed by that particular banker.

Account Schema

The account schema is for the customer's account in the bank.

Field	**Type**
account_id	INT
account_balance	INT
account_type	TEXT
branch_id	INT

· **account_id:**
The unique id to identify any account.
· **account_balance :**
The total balance in the account.
· **account_type :**
The type of the account.
· **branch_id :**
The branch with which the account is associated.

Customer Schema

This schema represents the customers of the bank.

Field	**Type**
customer_id	INT
customer_name	TEXT
mobile_no	INT
dob	DATE
account_id	INT

customer_id :
The unique id for each customer
· **customer_name :**

The name of the customer.
- **dob :**
  Date of birth of the customer.
- **mobileno :**
  The mobile number of the customer.
- **account_id :**
  The accounts associated with the particular customer.

Transaction Schema

This schema represents the transactions of accounts through any customers of the bank.

Field	**Type**
transaction_id	INT
transaction_type	TEXT
amount	INT
customer_id	INT
account_id	INT

transaction_id :
The unique id for each transaction.
- **transaction_type :**
  The type of transaction i.e debit/credit.
- **amount :**
  The amount of transaction.
- **customer_id :**
  The customer who initiated the transaction.
- **account_id :**
  The accounts associated with the particular customer.

Loan Schema

It will represent the loans taken by customers and provided by the branch.

Field	**Type**
loan_id	INT

remaining_amount	INT

Field	**Type**
issued_amount	INT
branch_id	INT
account_id	INT

· **loan_id :**
Unique id for each loan.
· **issued_amount :**
The original loan amount issued by the branch to the customer. ·
remaining_amount :
The debt amount which is remaining on the customer.

· **account_id :**

The account associated with the loan.

· **branch_id :**

The branch from which the loan was borrowed.

Loan Payment Schema

The payment for a loan will be represented by this schema, each loan payment corresponds to some amount along with the loan id.

Field	**Type**
loan_payment_id	INT
amount	INT
loan_id	INT

· **loan_payment_id :**
  The unique id of each payment towards the loan.
· **loan_id :**
  The loan which is associated with the payment.
· **amount :**
  The amount of payment.

Borrower Schema

The customers who have taken out any loan will be represented by this schema. Each borrower ID corresponds to a loan ID and the associated customer along with that loan.

Field	**Type**
borrower_id	INT
customer_id	INT
loan_id	INT
customer_name	TEXT

· **borrower_id**
  The unique id of each borrower.
· **loan_id**
  The loan is associated with the borrower.
· **customer_id**
  The ID of the customer who has taken the loan.
· **customer_name**
  The name of the customer who has taken the loan.

Credit Card Schema

This schema represents the credit card and related details for any customer.

Field	**Type**
credit_card_id	INT

card_limit	INT
expiry_date	DATE
customer_id	INT
account_id	INT

- **credit_card_id :**
  Unique ID to identify any credit card.
- **customer_id :**
  The customer is associated with the credit card.
- **account_id :**
  The account ID is associated with the credit card.
- **expiry_date :**
  The expiry date of the credit card.
- **card_limit :**
  Total amount of limit of the card.

Creating Bank Database Tables Using MySQL

The design phase and all conceptual discussions have now been completed. The next step is to implement these schemas and relations in the database in the form of tables to represent that skeleton structure inside databases. Tables are a collection of related data where each row represents a data entry, which is usually called a tuple, and reflects the information about any real-world object.

We will store each entity in the form of tables.

Create Database

Initially, we are going to create a database.

```
CREATE DATABASE banking_system;
```

Show Databases

We can view databases with the show databases statement.

```
SHOW databases;
+--------------------+
| Database |
+--------------------+
| banking_system |
| information_schema |
| mysql |
| performance_schema |
| sys |
+--------------------+
```

Switch Database

When the database is successfully created, we have to switch to it for work.

```
USE banking_system;
```

Create Tables

Here we will create tables corresponding to each entity and their relationship as described in the Schema Diagram.

Before starting to write the queries, here is a short description of what you will see in most of the queries.

· The **NOT NULL** written along with the field represents the value for this field and must be provided at the time of insertion of data into the table.

· The **AUTO_INCREMENT** written along with the field represents the value for that field and will be inserted internally and incremented at each insertion.

· The **VARCHAR**(30) represents that a particular field can store a maximum of 30 characters in that field value.

· **PRIMARY KEY**(field) represents that the field is going to be the primary key. · **FOREIGN KEY**(field) REFERENCES other_table(key_from_other_table), this syntax is used to refer to another table by using the field as a foreign key.

Branch

This query will create a table named branch having the branch ID as the primary key, branch name, assets, and branch address.

```
CREATE TABLE branch(
 branch_id INT NOT NULL AUTO_INCREMENT,
 branch_name VARCHAR(30) NOT NULL,
 assets INT NOT NULL,
 branch_address VARCHAR(255) NOT NULL,
 PRIMARY KEY(branch_id)
);
```

Banker Info

This query will create a table named banker_info, having banker ID as a primary key, banker name, and branch ID.

```
CREATE TABLE banker_info(
 banker_id INT NOT NULL AUTO_INCREMENT,
 banker_name VARCHAR(255) NOT NULL,
 branch_id INT NOT NULL,
 PRIMARY KEY (banker_id),
FOREIGN KEY (branch_id) REFERENCES branch(branch_id)
);
```

Account

This query will create a table named account, having account ID as a primary key, account type, and account balance. This table will refer to the branch table by the foreign key branch ID.

```
CREATE TABLE account(
 account_id INT NOT NULL AUTO_INCREMENT,
  account_type VARCHAR(30) NOT NULL,
 account_balance INT NOT NULL,
 branch_id INT NOT NULL,
 PRIMARY KEY (account_id),
 FOREIGN KEY (branch_id) REFERENCES branch(branch_id)
);
```

Customer

The query written below will create a table named customer, which will contain the customer ID as the primary key, customer name, mobile number, and date of birth. The account ID is a foreign key that will be used to refer to the account table and will be used to create an association between customers and their accounts.

```
CREATE TABLE customer(
 customer_id INT NOT NULL AUTO_INCREMENT,
 customer_name VARCHAR(30) NOT NULL,
 mobileno VARCHAR(10) NOT NULL,
 dob DATE,
 account_id INT NOT NULL,
 PRIMARY KEY (customer_id),
 FOREIGN KEY (account_id) REFERENCES account(account_id)
);
```

Transaction

The query will create a table named transaction, having transaction ID as a primary key, amount, customer ID, and account ID. This table will refer to the account and customer tables by the foreign key.

```
CREATE TABLE transaction(
 transaction_id INT NOT NULL AUTO_INCREMENT,
 amount INT NOT NULL,
 customer_id INT NOT NULL,
 account_id INT NOT NULL,
 PRIMARY KEY (transaction_id),
 FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
 FOREIGN KEY (account_id) REFERENCES account(account_id)
);
```

Credit Card

The query written below will create a table named customer credit card having credit card ID as a primary key, expiry date for that credit card, and card limit. The customer ID and account ID are foreign and will be used to refer to the associated customer and account for any particular credit card.

```
CREATE TABLE customer_credit_card(
credit_card_id INT NOT NULL AUTO_INCREMENT,
expiry_date DATE NOT NULL,
card_limit INT NOT NULL,
customer_id INT NOT NULL,
account_id INT NOT NULL,
PRIMARY KEY (credit_card_id),
FOREIGN KEY (customer_id) REFERENCES customer(customer_id),
FOREIGN KEY (account_id) REFERENCES account(account_id)
);
```

Loan

This query will create a table loan having loan ID as a primary key, issued amount, and the remaining amount. The branch ID will be used as a foreign key to refer to the branch that provided the loan, and the account ID will refer to the account on which the loan is being borrowed.

```
CREATE TABLE loan(
loan_id INT NOT NULL AUTO_INCREMENT,
issued_amount INT NOT NULL,
remaining_amount INT NOT NULL,
branch_id INT NOT NULL,
account_id INT NOT NULL,
PRIMARY KEY(loan_id),
FOREIGN KEY (branch_id) REFERENCES branch(branch_id),
FOREIGN KEY (account_id) REFERENCES account(account_id)
);
```

Loan Payment

This query will create a table named loan_payment, which will have the loan payment ID as the primary key and the amount of the payment. The loan ID will refer loan table and identify the loan for which payment is being done.

```
CREATE TABLE loan_payment(
loan_payment_id INT NOT NULL AUTO_INCREMENT,
amount INT NOT NULL,
loan_id INT NOT NULL,
PRIMARY KEY (loan_payment_id),
FOREIGN KEY (loan_id) REFERENCES loan(loan_id)
);
```

Borrower Table

This query will create a table named borrower, which will have the borrower ID as the primary key, customer ID, customer name, and loan ID.

```
CREATE TABLE borrower(
borrower_id INT NOT NULL AUTO_INCREMENT,
customer_id INT NOT NULL,
loan_id INT NOT NULL,
PRIMARY KEY (borrower_id),
FOREIGN KEY (loan_id) REFERENCES loan(loan_id),
FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
);
```

After running all these statements, we can see our tables with the show tables command. It will show you something like this,

```
+--------------------------+
| Tables_in_banking_system |
+--------------------------+
| account |
| banker_info |
| borrower |
| branch |
| customer |
| customer_credit_card |
| loan |
| loan_payment |
| transaction |
+--------------------------+
```

Also, we can view our schema by selecting any table, DESCRIBE banking_system.account;

```
+-----------------+-------------+------+-----+---------+----------------+
| Field | Type | Null | Key | Default | Extra |
+-----------------+-------------+------+-----+---------+----------------+
| account_id | int | NO | PRI | NULL | auto_increment |
| account_type | varchar(30) | NO | | NULL | |
| account_balance | int | NO | | NULL | |
| branch_id | int | NO | MUL | NULL | |
+-----------------+-------------+------+-----+---------+----------------+
```

Perform Querying

In this section, we will start performing the SQL queries for the banking database,

Create Branches.

To start operating with the banking system, the first mandatory thing is the branch, so we are inserting a few branches and their information into the database with the help of the Insert statement.

```
INSERT INTO branch (branch_name, branch_address, assets)
VALUES ('ABCD Branch', 'Sector No. 4, City XYZ, 100213, India', 100000000);
INSERT INTO branch (branch_name, branch_address, assets)
VALUES ('EFGH Branch', 'Street No. 3, Colony No. 9, City GHI, 239811 Uttar Pradesh',
230000000);
INSERT INTO branch (branch_name, branch_address, assets)
VALUES ('IJKL Branch', 'Apartment No. 87, Nearby XYZ Road, City MNO, 192211 Telangana',
650000000);
```

Output: If we select all the inserted data, it will look like this,

```
+-----------+-------------+-----------+-----------------------------------------------------------------+ | branch_id |
branch_name | assets | branch_address | +-----------+-------------+-----------+------------------------------------
----------------------------+ | 1 | ABCD Branch | 100000000 | Sector No. 4, City XYZ, 100213, India | | 2 |
EFGH Branch | 230000000 | Street No. 3, Colony No. 9, City GHI, 239811 Uttar Pradesh |
| 3 | IJKL Branch | 650000000 | Apartment No. 87, Nearby XYZ Road, City MNO, 192211 Telangana
|
+-----------+-------------+-----------+-----------------------------------------------------------------+
```

Create Bankers

A **banker** is a person who manages a branch. We can use a simple insert statement to store the banker's information in the database.

```
INSERT INTO banker_info (banker_name, branch_id)
VALUES ('ABC DEF', 1);
INSERT INTO banker_info (banker_name, branch_id)
VALUES ('PQR STU', 3);
INSERT INTO banker_info (banker_name, branch_id)
```

```
VALUES ('IJK MNO', 2);
```

Output :

```
+-----------+-------------+-----------+
| banker_id | banker_name | branch_id |
+-----------+-------------+-----------+
| 1 | ABC DEF | 1 |
| 2 | PQR STU | 3 |
| 3 | IJK MNO | 2 |
+-----------+-------------+-----------+
```

Create Accounts

To create an account, we can insert data in the account table along with providing the necessary information.

```
INSERT INTO account (branch_id, account_balance, account_type)
VALUES (1, 0, 'savings');
INSERT INTO account (branch_id, account_balance, account_type)
VALUES (2, 80000, 'current');
```

Output :

```
+------------+--------------+-----------------+-----------+
| account_id | account_type | account_balance | branch_id |
+------------+--------------+-----------------+-----------+
| 1 | savings | 0 | 1 |
| 2 | current | 80000 | 2 |
+------------+--------------+-----------------+-----------+
```

Associate the Customer with the Account

After adding the accounts' data, we can create associated customers with them.

```
INSERT INTO customer (customer_name, mobileno, dob, account_id)
VALUES ('XYZ ABC', 988324122, '1998-01-31', 2);
INSERT INTO customer (customer_name, mobileno, dob, account_id)
VALUES ('JKL PQRS', 898732112, '2000-05-14', 1);
```

Output :

```
+-------------+---------------+-----------+------------+------------+
| customer_id | customer_name | mobileno | dob | account_id |
+-------------+---------------+-----------+------------+------------+
| 1 | XYZ ABC | 988324122 | 1998-01-31 | 2 |
| 2 | JKL PQRS | 898732112 | 2000-05-14 | 1 |
+-------------+---------------+-----------+------------+------------+
```

Perform Transactions on the Account

To perform any transaction, we can insert the details of the transaction, the associated account, and the customer with that transaction.

After inserting data for the transaction, to keep the database consistent, we will need to update the balance in the account.

```
INSERT INTO transaction(amount, transaction_type, customer_id, account_id) VALUES (900, 'debit', 1, 2);
UPDATE account
SET account_balance = account_balance - 900
WHERE account_id=2;
INSERT INTO transaction(amount, transaction_type, customer_id, account_id) VALUES (10000, 'credit', 1, 1);
UPDATE account
SET account_balance = account_balance + 10000
WHERE account_id=1;
```

Output :

```
Transaction Table
+----------------+--------+-------------+------------+-----------------+
| transaction_id | amount | customer_id | account_id | transaction_type |
+----------------+--------+-------------+------------+-----------------+
| 1 | 900 | 1 | 2 | debit |
| 2 | 10000 | 1 | 1 | credit |
+----------------+--------+-------------+------------+-----------------+
Account Table
+------------+--------------+-----------------+-----------+
| account_id | account_type | account_balance | branch_id |
+------------+--------------+-----------------+-----------+
| 1 | savings | 10000 | 1 |
| 2 | current | 79100 | 1 |
+------------+--------------+-----------------+-----------+
```

Create a Loan for Accounts

We can allocate the loan to any account by inserting related information in the loan table.

```
INSERT INTO loan (branch_id, issued_amount, remaining_amount, account_id) VALUES (2, 10000, 10000, 1);
INSERT INTO loan (branch_id, issued_amount, remaining_amount, account_id)
VALUES (3, 500000, 500000, 2);
```

Output :

```
+---------+---------------+-----------------+-----------+------------+
| loan_id | issued_amount | remaining_amount | branch_id | account_id | +----
-----+---------------+-----------------+-----------+------------+
| 1 | 10000 | 10000 | 2 | 1 |
| 2 | 500000 | 500000 | 3 | 2 |
+---------+---------------+-----------------+-----------+------------+
```

Issue a Credit Card for the Customer's Account

We can issue the credit card by providing the customer ID, account ID, and other related information.

```
INSERT INTO customer_credit_card (expiry_date, card_limit, customer_id, account_id)
VALUES ('2023-05-21', 5000, 1, 2);
INSERT INTO customer_credit_card (expiry_date, card_limit, customer_id, account_id)
VALUES ('2025-10-15', 20000, 2, 1);
```

Output :

```
+----------------+-------------+------------+-------------+------------+
| credit_card_id | expiry_date | card_limit | customer_id | account_id | +----
------------+-------------+------------+-------------+------------+
| 1 | 2034-05-21 | 10000 | 1 | 2 |
| 2 | 2025-10-15 | 20000 | 2 | 1 |
+----------------+-------------+------------+-------------+------------+
```

Change the Expiry of the Credit Card

The updation of expiry_date or limit can be done with the **UPDATE** clause,

```
UPDATE customer_credit_card
SET expiry_date = '2034-05-21', card_limit=10000
WHERE customer_id=1;
SELECT * FROM customer_credit_card WHERE customer_id=1;
```

Output :

```
+----------------+-------------+------------+-------------+------------+
| credit_card_id | expiry_date | card_limit | customer_id | account_id |
+----------------+-------------+------------+-------------+------------+
| 1 | 2034-05-21 | 10000 | 1 | 2 |
+----------------+-------------+------------+-------------+------------+
```

Make a Payment Toward the Loan

To pay for a loan, we can insert the data in the loan payment table and subsequently decrease the remaining amount of the loan.

```
INSERT INTO loan_payment(loan_id, amount)
VALUES (1, 2500);
UPDATE loan
SET remaining_amount = remaining_amount-2500
WHERE loan_id=1;
SELECT * FROM loan WHERE loan_id = 1;
```

Output :

```
+---------+---------------+-----------------+-----------+------------+
| loan_id | issued_amount | remaining_amount | branch_id | account_id |
+---------+---------------+-----------------+-----------+------------+
| 1 | 10000 | 7500 | 2 | 1 |
+---------+---------------+-----------------+-----------+------------+
```

Create the Loan Borrower Table

In our database, the loan is related to the account, and also the customer is related to the account, so consider a situation where we want to fetch a list of all borrowers.

We can use the JOIN in SQL, which is used to join two tables according to any given common field of both tables, i.e., account ID.

So the query written below will insert the data in the borrower table, but this time we will not provide the values; instead, it will be extracted from the existing tables. The select statement in the following query will select some fields from the loan table and the customer table. Finally, the **INNER JOIN** will be done on the account_id.

```
INSERT INTO borrower(loan_id, customer_id, customer_name)
SELECT loan.loan_id, customer.customer_id, customer.customer_name FROM loan
INNER JOIN customer ON loan.account_id=customer.account_id;
```

Output :

```
+------------+------------+---------+--------------+
| borrower_id | customer_id | loan_id | customer_name |
+------------+------------+---------+--------------+
| 1 | 1 | 2 | XYZ ABC |
| 2 | 2 | 1 | JKL PQRS |
+------------+------------+---------+--------------+
```