****

**Advanced Database Management Systems Project**

**MariaDB**

****

**By**

**Group - 7**

|  |  |
| --- | --- |
| **Name** | **UNumber** |
| **Sandeep Karanam** | **U94742075** |
| **Sandeep Reddy Nimma** | **U53426553** |
| **Greeshma Namana** | **U24918180** |
| **Rohit Vadlala** | **U93298968** |

**Q1) A detailed description of the DBMS (MariaDB DBMS):**

**Definition:**

MariaDB is a free, open-source database system that's very similar to MySQL. It was developed by the same people who created MySQL, and it's designed in a way that you can switch from MySQL to MariaDB without causing big problems for your applications or databases. In simple terms, you can use MariaDB instead of MySQL with minimal fuss.

**Historical Context:**

Michael "Monty" Widenius, the original creator of MySQL, launched MariaDB in 2009 as a result of concerns about the future of MySQL under Oracle's management. The name of the MariaDB is represented after Monty's youngest daughter’s name, "Maria".

**Key Features:**

MariaDB offers several notable features, including:

* Improved Performance and Scalability: MariaDB is optimized for better efficiency and scalability compared to MySQL.
* Aria Storage Engine: This is MariaDB's default storage system, and it's renowned for its quick speed and crash recovery capabilities.
* Upgrades to InnoDB: MariaDB has made enhancements to InnoDB that lead to superior concurrency and overall performance.
* Thread Pool: This functionality makes sure that several connections and queries are managed effectively at the same time.
* Diverse Storage Options:
* TokuDB: Well-known for its proficiency in handling large datasets and data compression skills.
* ColumnStore: Its columnar format makes it appropriate for scenarios involving massive amounts of data, and it is designed for real-time analytics and OLAP operations.
* MyRocks: An LSM-based storage system praised for its operational efficiency.
* Robust Security Measures:
* Data Encryption: Enables encryption at several levels - be it entire tablespaces, individual tables, or specific columns.
* Access Control Based on Roles: Provides a convenient system to assign and withdraw privileges.
* Audit Feature: Allows for in-depth tracking of user interactions and SQL actions to meet compliance standards.
* Reliability & Duplication:
* Galera Cluster: Supports synchronous multi-master replication, addressing the requirements of high availability and expandability.
* GTID Replication: Streamlines the duplication process and enhances failover management.
* Enhanced SQL & Schema Features:
* Window Functions: Offers advanced SQL tasks related to processing within a group of rows.
* CTE (Common Table Expressions): Improves the query framework and supports recursive query execution.
* JSON Handling: It offers a comprehensive range of tools for working with JSON data, from the creation, interpreting to querying.
* Backup Tools: Features solutions such as MariaBackup for uninterrupted backups.
* Expandability: MariaDB offers the flexibility in installing plugins, custom storage engines and user-defined functions.

**Compatibility:**

MariaDB is a database system that can handle a wide range of SQL commands, functions, and other database-related tasks. It comes with various handy tools like mysqladmin and mysqldump, which are command-line programs. Additionally, there are useful add-ons like the audit plugin available. These features make managing databases easier when using MariaDB.

**Licensing and Community:**

* Open Source: MariaDB is licensed under the GPL, version 2, which ensures both its community commitment and continuous availability at no cost.
* MariaDB Foundation: This body is responsible for monitoring and guiding the development of MariaDB and guaranteeing that it remains transparent and actively engages the community.

**Adoption & Use Cases:**

* MariaDB is adopted by numerous businesses from small startup companies to big organizations across various industries. Owing to its open-source nature, performance enhancements, and robust community support which ultimately makes it a favored choice for web databases, data warehousing, and more.

**Core Functions of Maria DBMS**

**Overview**:

1. Data Retrieval:

Index Structures: MariaDB uses indexes to quickly find data. Think of indexes like a table of contents in a book that helps you locate information without reading the entire book. This speeds up most queries.

Query Optimizer: This is like a smart assistant for queries. It looks at the query and figures out the best way to execute it, considering factors like data size, available indexes, and system resources. MariaDB keeps improving this assistant to make queries faster and more efficient.

Buffer Pool: MariaDB's InnoDB storage engine has a special cache in memory called the buffer pool. It stores frequently accessed data, so the database doesn't have to keep reading from the hard disk. This makes data retrieval much faster.

Read-Ahead Techniques: Imagine it's predicting what pages of a book you'll need to read next and loads them in advance. This way, when you actually need that data (make a query), it's already in memory, reducing the need to read from the disk. This speeds up data retrieval by reducing the time it takes to get the data you want.

2. Data Manipulation:

In MariaDB, Data Manipulation Language (DML) is essential for working with data. Here's a closer look at these operations:

INSERT: This lets you add new records to a table. You can use variations like INSERT IGNORE to ignore errors and INSERT ON DUPLICATE KEY UPDATE to update a record if a duplicate primary key is found.

UPDATE: It's used to change existing records in a table based on specific conditions. You can also use JOINs to modify data based on related information in another table.

DELETE: This command removes records from a table. Be cautious, especially without a WHERE clause, as it can delete all records in the table. MariaDB also has DELETE... RETURNING, which returns the deleted rows.

MERGE: MariaDB doesn't have a direct MERGE command, but it offers similar functionality with INSERT... ON DUPLICATE KEY UPDATE to insert new records or update existing ones based on duplicate keys.

When dealing with large amounts of data, MariaDB offers some efficient options:

INSERT: You can insert multiple records using a single INSERT command with multiple sets of values, which is faster than individual inserts.

LOAD DATA INFILE: This is a powerful tool for quickly importing data from text files, especially for large datasets. You can specify formats, delimiters, and handle data transformations during the import.

DELAYED INSERTS: MariaDB has INSERT DELAYED, which queues insert requests for later, useful when immediate confirmation isn't needed, and it can boost application performance by deferring data insertion.

3. Data Integrity and Accuracy:

Constraints are like rules that help MariaDB keep its data in good shape. Let's dive into these rules:

PRIMARY KEY: Think of this as a special tag that makes sure every item in a table is unique. It's like giving every item a distinct barcode. This helps prevent duplicates and makes it easy to find things quickly.

FOREIGN KEY: This is like connecting two tables in a special way. It links one table to another through a code. This ensures that the relationship between these tables stays consistent and accurate.

UNIQUE: It ensures that everything in a list is different. For example, it makes sure no two people have the same email address. Unique things stay unique.

CHECK: It's like setting a rule for a particular column, such as making sure that the year you enter as a birth year is reasonable. It stops you from entering weird or incorrect data.

NOT NULL: This says that certain information can't be left blank. For example, it makes sure you always enter an address in a contact form.

Transactions are like safeguarding mechanisms for database operations. Here's what they do:

Atomicity: It ensures that when you do several things at once, either all of them happen, or none of them do. It prevents things from being half-done, which could mess up the data.

Consistency: After you do a bunch of things, the database should end up in a sensible state. Even if lots of people are making changes at the same time, everything should still make sense in the end.

Isolation: If two people are changing the same thing at the same time, one of them has to wait their turn. This prevents them from interfering with each other and causing a mess.

Durability: Once you say you've done something, it's written in stone. Even if the computer crashes, your changes are safe and sound. MariaDB uses special techniques to make sure nothing is lost.

4. Concurrency Control:

Multi-version Concurrency Control (MVCC) is a smart system that MariaDB uses to handle many things happening in the database at the same time.

Here's what it does:

Versioned Records: When someone makes changes to a piece of information, MVCC keeps a copy of the old version. This way, other people who are looking at that information can still see the original data. It's like having different books for each person, so they each see what they expect.

Read Consistency: MVCC makes sure that when someone is reading something, it doesn't block someone else who's trying to change it. Even if you're in the middle of updating something, others can still see what it looked like before you started, so everything keeps running smoothly.

Phantom Reads Prevention: Sometimes, when one person is reading, they might see things that weren't there before or miss things that were there. MVCC helps prevent this confusion by showing everyone a clear picture of the data, just like taking a snapshot of a moment.

Now, let's discuss about locks:

Row-level Locking: MariaDB is clever in how it locks things. Instead of locking the whole table, it can lock just one row. So, if you're changing one thing, someone else can change something else at the same time. This makes things much faster.

Shared and Exclusive Locks: Shared locks allow many people to read something at the same time, but only one person can change it (exclusive lock). It's like sharing a book but not letting others write in it.

Deadlock Detection: Sometimes, when everyone is waiting for each other, the database gets stuck. MariaDB has a way to notice this and fix it by canceling one of the actions so the others can continue.

In a nutshell, MVCC and these smart locking methods make sure that MariaDB can handle lots of things happening all at once, keeping data safe and transactions running smoothly, even when the database is super busy.

5. Redundancy Removal:

MariaDB has some clever techniques to get rid of redundant data, which is when you have the same information stored in multiple places. This can save space and make things work better.

Normalization: This is like a structured way of organizing data so that you don't store the same thing over and over. MariaDB helps by making sure that data that depends on something else is stored in a smart way, reducing repetition.

Decomposition: Instead of copying the same information in many places, you can split it up and keep it in one central spot. For instance, you can have a separate list of addresses and refer to it when needed, so you don't repeat the same address multiple times.

Referential Integrity: MariaDB keeps an eye on the connections between different pieces of data. It ensures that these connections stay consistent, which not only keeps the data reliable but also stops you from duplicating the same information all over.

Storage Engines: Different ways of storing data can also help reduce redundancy:

So, MariaDB uses these tricks to organize data efficiently, get rid of repeated stuff, and use storage space wisely. This not only saves space but also makes the database quicker and more efficient.

6. Data Independence:

Data independence is a crucial concept in modern databases, and MariaDB ensures it in two ways:

Logical Data Independence: This means that when you make changes to how data is organized logically, like adding new categories or changing data types, it won't disrupt how users or applications see and use the data. With MariaDB, you can make these changes without having to redesign existing applications. It's like rearranging your data without affecting how people view it. MariaDB also supports creating different “views" of the data for applications, so even if the underlying organization changes, the way data is presented to applications remains consistent.

Physical Data Independence: This ensures that changes in how data is physically stored and managed, like improving data storage efficiency, won't impact how users and applications access the data. In MariaDB, you can optimize how the data is stored without affecting how applications interact with it. MariaDB also supports different ways of storing data, so you can switch or optimize your data storage method without causing disruptions to how applications work with the data. It's like upgrading the storage system without affecting how people use the data.

7. Data Relationships:

In databases, maintaining connections between data is crucial. MariaDB, a database system, has some key features to help with this:

Foreign Keys:Foreign keys are like the glue that holds different parts of your database together. They make sure your data is linked correctly and stays reliable.

Referential Actions: In MariaDB, you can set rules for what happens when you update or delete a piece of data. For example, if you delete something in one table, the related data in another table can be automatically deleted too (called ON DELETE CASCADE). Or, you can set it to make the related data value become empty if the linked data is deleted (ON DELETE SET NULL).

Consistency Maintenance: MariaDB keeps things in order by making sure that when you add, change, or delete data, it doesn't mess up the connections between different pieces of information.

Indexed Foreign Keys: MariaDB makes sure that finding related data is quick by creating an index on the foreign keys. This helps when you're searching for connected data.

Join Operations: Joining data is like combining pieces of information from different tables in your database. In MariaDB, there are a few ways to do this:

* INNER JOIN: This gets you data that matches a specific condition from both tables. It only gives you results when there's a match in both tables.
* LEFT JOIN: This gets you all the data from the left table and the matching data from the right table. If there's no match, it includes NULL in the result.
* RIGHT JOIN: This works like LEFT JOIN but gets all the data from the right table and matching data from the left table. It also uses NULL when there's no match.
* Advanced Joins: Besides these basic joining methods, MariaDB lets you do more complex things like joining a table with itself (SELF JOIN) or creating the combination of all possible pairs from two tables (CROSS JOIN).

Optimization: MariaDB is smart about making all these connections work quickly. It uses advanced techniques to handle complex queries with many joins efficiently. This means you get your results fast, even when your database is large and complex.

**Advantages of MariaDB**

While there are some features not supported in MariaDB, it has many advantages:

* MariaDB supports more storage engines, giving users more options.
* It can handle a larger number of connections, which can be beneficial for high-traffic scenarios.
* Replication in MariaDB can be almost twice as fast as in MySQL.
* MariaDB is generally faster and easier to work with.
* It allows the creation of versioned tables, which can be useful for managing historical data.
* MariaDB offers compatibility features with Oracle Database, making it easier for users to switch from Oracle.
* MariaDB Server supports a columnar storage format, which is useful for certain types of data.
* MariaDB also provides a distributed SQL database solution called MariaDB Xpand.

Conclusion:

MariaDB, which can be demonstrated as a modern RDBMS, incorporates robust mechanisms in a way, handling data retrieval, manipulating the data, integrity, concurrency, and relationships effectively. It successfully handles fundamental database operations and obstacles, as it strictly follows the relational principles ensuring its adherence to cope-up with the continuous optimizations and enhancements.

**Q2) A detailed description of the KDD Nuggets referenced data**

**Bank Database Design**

**Objective:** The main objective of the bank database that is taken from the KDD Nuggets referenced data is ensuring seamless transactions of the bank's operations, by taking the structure of the bank as a prime focus element, managing the client interactions with the system effectively, and efficient manner of handling the data.

Within the selected dataset, multiple discrete datasets have been found, each pertaining to a particular aspect of financial activities and transactions. The combination of these separate datasets is what we call the "bank dataset." We have designed a comprehensive and interconnected database system, dubbed the "bank database," to handle and analyze the vast amount of information found in these datasets.

To implement our strategy, we first create unique entities in the database, each of which corresponds to a different dataset. These entities act as well-organized data containers, guaranteeing the appropriate preservation of the relationships and underlying structure among the various forms of banking information. By doing this, we can make the most of this large dataset and realize its full potential, making it more accessible, organized, and insightful.

This structured methodology helps us get valuable insights from the bank dataset and improves the data management process overall. It enables us to monitor financial movements, examine transactional trends, examine client behavior, and offer insightful information to assist strategic decision-making in the financial industry. In essence, our design and implementation of the bank database facilitate a more profound exploration of the complex financial landscape encapsulated by these multiple datasets.

Unique entities created in the Database corresponding to each unique dataset:

1. Account:

This entity is more focussed on representing the banking accounts of clients: Attributes which could be included are, Account\_ID, Balance, Date\_Created, Branch\_ID (foreign key that is linking to Branches), account\_type\_id (foreign key that is linking to account\_types), and client\_id (foreign key that is linking to Clients).

2. Account\_Types:

This entity describes various kinds of accounts that are available such as: savings, checking, fixed deposit, etc. Attributes which could be included are, Account\_Type\_ID, Type\_Name, Interest\_Rate, and other specific features that are related to the account type.

3. Branches:

This entity provides the information that is in particular towards the branches of the bank. Attributes which could be included are, Branch\_ID, Branch\_Name, Branch\_Location, Branch\_Manager, etc.

4. Clients:

This entity focuses on providing the information about the clients/customers of the bank clients. Attributes which could be included are, Client\_ID, Client\_Name, Address, Contact\_Details, etc.

5. Transactions:

This entity primarily aims in keeping the records of all the transactions that are operated between accounts. Attributes which could be included are, Transaction\_ID, Source\_Account\_ID (foreign key that is linking to Account), Destination\_Account\_ID (foreign key that is linking to Account), Amount, Transaction\_Date, and Transaction\_Type\_ID (foreign key that is linking to Transaction\_Types).

6. Transaction\_Type:

This entity provides the information about the type of transaction that would occur such as deposit, transfer, withdrawal, etc. Attributes which could be included are, Transaction\_Type\_ID, Type\_Description, Applicable\_Fees, etc.

**Key Relationships:**

Accounts to Branches: A branch can service more than one account, but each account is linked to a single branch.

Accounts to Account Types: Each kind of account is unique, although each type might include more than one account in its classification.

Clients to Accounts: Although a client may hold many accounts, each account is held exclusively by that client.

Transactions to Accounts: Multiple transactions may be made on a single account. There is a source account and maybe a destination account for every transaction.

Transactions to Transaction\_Type: Every transaction has a unique type, although there are several transactions that can be connected to a single type.

**Additional Features:**

Transaction Procedure: When a transaction takes place, this process records the information and modifies the source and destination accounts. This process also manages transaction costs, which vary according to the kind of transaction.

**Optimization Strategies:**

Indexing: To expedite data retrieval processes, the database employs indexing on certain columns, which functions as a directory.

Denormalization: When the same data is stored in multiple tables, there's a chance that the database design purposefully introduces redundancy. As a result, fewer joins are required, which lowers the complexity and processing time of some queries.

In essence, this dataset offers a thorough understanding of a banking system, safeguarding the data integrity, structuring the relational data, and managing the operations effectively.

**Q-3 A detailed description of the Product (Transactional or Analytical). You must describe why the design of the Product makes it Transactional or Analytical.**

Based on the banking database, we have provided a detailed description which focuses on the transactional nature of the product:

**Product Description:**

The product outlined in the banking database we are trying to build has a comprehensive transactional banking system that is engineered in such a manner to manage and record daily banking transactions. It is composed of several essential parts that are required to carry out financial operations within the bank:

Client: This entity is a representation of the bank's clients; it stores essential personal data that enables individualized banking services and account management.

Account: This records each customer's bank account's balance and status, along with additional financial information. Attributes such as balance\_account and status are indicative of the system's ability to reflect changes in account data in real time.

Branch: This illustrates how the system may be configured to handle activities at different locations by representing the bank's many branches, including its administrative and physical divisions.

Transaction: This is an essential component that keeps track of every financial transaction made in client accounts. It is essential for monitoring cash flows in real time and keeping an accounting of all financial operations.

Account Type: By classifying bank accounts into distinct groups, this helps the system in identifying different transaction policies and account attributes within the system.

Transaction Type: The range of transactional activities that are likely to be performed within the banking system is represented by this term. These operations may include transfers, withdrawals, and deposits, each of which may be subject to different processing protocols.

**Transactional Nature of the Product:**

The banking database, that we are trying to build demonstrates attributes that are indicative of a transactional system for several compelling reasons:

1. Real-Time Processing: The design of the system enables it to respond to actions as they happen swiftly. This is critical in a banking environment because precise current state reporting is required for account balances and transaction records, which must be updated instantly.

2. ACID Compliance: Based on the entities and relationships, it seems that the following design will likely support ACID properties:

Atomicity: A transaction cannot be left in an intermediate state; it is viewed as a single unit that either fully succeeds or fails.

Consistency: The database upholds business standards by staying in a consistent state both before and after each transaction.

Isolation: Transactions are prevented from other concurrent transactions' interference, that is to preserve the integrity of each individual transaction.

Durability: An operation is permanent because once a transaction is committed, it cannot be reversed.

3. Normalized Schema: We are attempting to create a banking database that has a normalized database design, with separate entities for transactions, accounts, customers, and so on. Integrity is crucial in transactional systems, as normalization reduces duplication and guarantees that data modifications do not result in inconsistencies.

4. Detailed Record-Keeping: As required for transaction processing, the transaction entity indicates that the system is designed to capture comprehensive data for each financial operation.

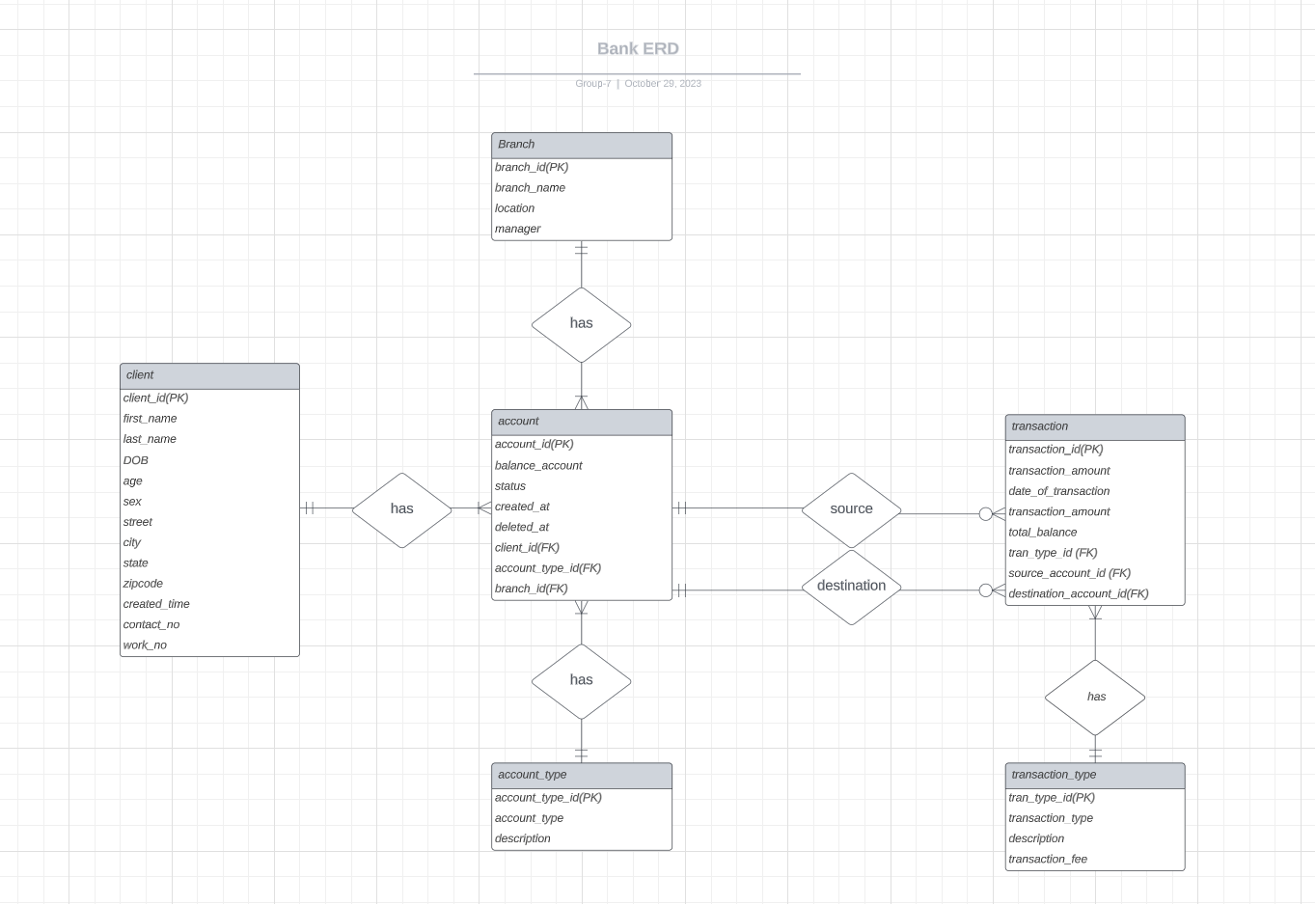
5. Data Integrity and Relationships: The system appears to maintain relational integrity by using primary keys (PK) and foreign keys (FK), which guarantee that all transactions are linked to the appropriate accounts and customers.

6. Concentrated on Current Data: The attributes that are found in the entities account and transaction, like date\_of\_transaction, status, balance\_account, and transaction\_amount, are a significant aspect of a system that would demand the need of accurate and current data that is easily accessible to assist in providing the customer feedback swiftly and operational decision-making effectively.

7. Operational Effectiveness: The banking database that we are trying to build appears to be better optimized for transactional operations, in regards to it, we have specifically tailored the system for creating, reading, updating, and deleting data—activities that are fundamental to transactional systems, rather than being equipped for complex queries and analysis.

These elements of design together point to a transactional banking system that is equipped for managing large volumes of transactions that is capable of functioning effectively and securely

**Q-4) A detailed description of the Product data structures (Transactional or Analytical)**

****

The product, as shown in the Entity-Relationship Diagram (ERD), is a banking system that manages customer interactions, accounts, and transactions.

Below is a detailed description of the data structures that comprise the system:

**Entity Descriptions:**

1. Client Entity:

* client\_id (PK): A primary key that uniquely determines each client.
* first\_name, last\_name: The client's full name.
* DOB, age: The client's date of birth and age.
* sex: The client's gender.
* street, city, state, zipcode: The client's address
* created\_time: Timestamp for when the client record was created.
* contact\_no, work\_no: Different contact numbers for reaching the client, separating personal and work.

2. Account Type Entity:

* account\_type\_id (PK): A primary key that uniquely determines the type of an account.
* account\_type: The name or type of the account (e.g., checking, savings, business).
* description: A description of the account type.

3. Branch Entity:

* branch\_id (PK): A primary key that uniquely determines a bank branch.
* branch\_name: The name of the branch.
* location: The location of the branch.
* manager: The manager in charge of the branch.

4. Account Entity:

* account\_id (PK): A primary key that uniquely recognises an account.
* balance\_account: The current balance of the account.
* status: The status of the account (e.g., active, suspended, closed).
* created\_at: Timestamps for the creation of the account.
* deleted\_at: Timestamps for the deletion of the account.
* client\_id (FK): Foreign key linking to the client entity.
* account\_type\_id (FK): Foreign key linking to the account type entity.
* branch\_id (FK): Foreign key linking to the branch entity where the account is managed.

5. Transaction Entity:

* transaction\_id (PK): A primary key that uniquely determines a transaction.
* transaction\_amount: The amount of money involved in the transaction.
* date\_of\_transaction: The date when the transaction took place.
* total\_balance: The total balance after the transaction was completed.
* source\_account\_id, destination\_account\_id (FKs): Foreign keys identifying the accounts involved in the transaction.

6. Transaction Type Entity:

* tran\_type\_id (PK): A primary key that uniquely determines the type of transaction.
* transaction\_type: The name or type of the transaction (e.g., deposit, withdrawal, transfer).
* description: A description of the transaction type.
* transaction\_fee: Any fee associated with this type of transaction.

**Data Structure Characteristics:**

The data structure is designed to indicate a relational database model, which is noticeable from the usage of primary and foreign keys to maintain relationships between different entities. The entities are normalized to reduce data redundancy and ensure data integrity. This normalization is crucial for transactional systems, which require high data quality and the ability to reflect the current state of data at any given moment.

The focus on individual transactions, account balances, and real-time processing indicates that the system is built for operational efficiency and is capable of handling a high volume of banking transactions. The system is likely optimized for quick access and manipulation of data, supporting fast CRUD (Create, Read, Update, Delete) operations essential for day-to-day banking activities.

In conclusion, the data structures are meticulously designed to serve the needs of a transactional banking system, facilitating efficient and secure financial operations.

**Relationship Descriptions:**

In Entity-Relationship Diagram (ERD), cardinality and modality are used to describe the relationships between entities.

Cardinality:

Cardinality defines the number of instances of one entity that can or must be associated with the number of instances of another entity.

Modality:

Modality indicates whether the presence of an entity in a relationship is mandatory or optional.

* Mandatory: The relationship must have an associated entity. For example, in the account-transaction relationship, every transaction must be linked to an account.
* Optional: The entity may or may not have an associated entity in the relationship. For example, a client might not have any accounts, which would make the client's side of the client-account relationship optional.

1. Client-Account Relationship:

* Cardinality: One-to-Many (1:N), This indicates that one client can have multiple accounts but each account is associated with only one client.
* Modality: The modality here is likely mandatory on the account side and optional on the client side, suggesting that an account must be associated with one client, but a client does not need to have an account.

2. Account-Transaction Relationship:

* Cardinality: One-to-Many (1:N). An account can be associated with multiple transactions either as a source or a destination, but each transaction is associated with only one source account and one destination account.
* Modality: The modality would be mandatory for transactions since every transaction must have an associated account. For accounts, it could be optional as there may be accounts without any transactions (e.g., a newly opened account).

3. Branch-Account Relationship:

* Cardinality: One-to-Many (1:N). A branch can manage multiple accounts.
* Modality: Usually mandatory for accounts since an account has to belong to a branch, but optional for branches as there might be branches with no accounts assigned yet (though this is unusual in a real-world scenario).

4. Account Type-Account Relationship:

* Cardinality: One-to-Many (1:N). Each account has one specific type, but each type can be assigned to many accounts.
* Modality: The modality is typically mandatory on both sides since every account needs to have a type, and account types are predefined categories in the system that should have associated accounts.

5. Transaction Type-Transaction Relationship:

* Cardinality: One-to-Many (1:N). Each transaction has one type, and each type can be applied to many transactions.
* Modality: The modality is usually mandatory for transactions, as each transaction must have a type. For transaction types, it is mandatory in the sense that the system should have existing transactions of that type.

To sum it up, the cardinality and modality of the relationships in the ERD guide the database constraints and business rules that need to be enforced in the banking system. They ensure that the system correctly reflects the logical structure of the bank's operations.

**Q-5) A detailed description of the CRUD operations if it is a transactional or detailed description of the ETL process if it is Analytical.**

**Create Operations:**

We've designed a set of CREATE TABLE statements to build the foundation for our bank database. Each statement creates a distinct table in the database, and together, these tables are responsible for organizing and storing crucial data that supports our bank's day-to-day activities. Now, let's delve into the specifics of each table and how they contribute to our database's functionality:

1. Branch table:

* branch\_id serves as a unique identifier for each branch, incrementing automatically with each new addition. It's the primary key of this table.
* branch\_name: This field holds the branch's name, allowing for variable character input up to 50 characters.
* location: We store the physical address of each branch here, permitting up to 50 characters.
* manager: This is where we put the manager's name, with a 50 character limit.

2. Client table:

* client\_id, we give each client a unique identifier, which is the primary key and auto-increments. We keep personal details such as first\_name, last\_name, DOB, age, sex, street, city, province, and zip code to store our clients' demographic information.
* created\_time: This timestamp defaults to the insertion time, noting when a client profile was created.
* contact\_no and work\_no: These fields are reserved for the client's personal and work contact numbers, respectively.

3. Account\_type Table:

* account\_type\_id: This primary key uniquely identifies each account type.
* account\_type: Here we describe the account type.
* description: We provide a text description for each account type.

4. Account Table:

* account\_id: As the primary key, it uniquely identifies each account and auto-increments.
* balance\_account: This decimal field with precision allows us to record account balances accurately.
* status, created\_at, and deleted\_at: These fields track the status and the timestamps for account creation and deletion.

We have also included foreign keys that link to the client, account\_type, and branch tables to maintain relationships.

5. Transaction\_type Table:

* tran\_type\_id: This is the unique identifier for each transaction type, serving as the primary key.
* transaction\_type and description: These fields name and describe the type of transaction.
* transaction\_fee: We specify transaction fees here.

6. Transaction Table:

* transaction\_id: Each transaction gets a unique identifier, which is our primary key.
* transaction\_amount, date\_of\_transaction, total\_balance: We use these to detail the transaction's financials and timing.

To maintain data accuracy and reliability in our database, we use foreign keys to connect to the transaction\_type and account tables, helping us identify the source and destination of funds.

Our database design prioritizes data integrity by using NOT NULL constraints, which ensure that important fields are always filled out. Additionally, we simplify the process of adding new data by automatically generating IDs, and we keep track of historical data by recording timestamps.

**Read Operations:**

We have two methods for importing data into our MariaDB database:

1. SQL INSERT
2. LOAD DATA LOCAL INFILE

In our database design, we've used the SQL INSERT method for three of our six tables, and the remaining three tables are populated using the LOAD DATA LOCAL INFILE method.

When we're importing data, especially when it involves date-related information, we ensure that the data format matches the requirements of our STR\_TO\_DATE functions. When using INSERT INTO transaction statements, we deliberately set the total\_balance field to NULL, as it aligns with our design intentions. It's crucial that the IDs we reference, such as tran\_type\_id, source\_account\_id, and destination\_account\_id, already exist in the database to avoid foreign key constraint issues.

We're meticulous about checking our database against various constraints, like NOT NULL, UNIQUE, and FOREIGN KEY, to make the data import process as smooth as possible. If we encounter any errors during the process, the error messages usually guide us in troubleshooting and resolving the issues effectively.

**Update Operations:**

We've crafted these SQL UPDATE statements to make changes to the records in both the transaction and account tables within our banking application database. These updates consider the transaction types and apply specific business rules based on the type of transaction we're processing. Each operation has a direct impact on the account balances and ensures that the total balance is updated accurately after the transaction is completed.

1. Update for transaction\_type = 1:

In this update, we handle transactions of type 1. We reduce the balance\_account in the account table by the transaction amount and the associated transaction fee. Our operation ensures that the new balance is a decimal with 15 digits of precision and 2 decimal places. We also update the total\_balance column in the transaction table to reflect the new balance in the account table for that specific transaction.

1. Update for transaction\_type = 2:

For transactions of type 2, our command decreases the account balance and sets the total\_balance of the transaction to the new balance after subtracting both the transaction amount and the transaction fee. Both the updated balance and total\_balance are cast to the specified decimal precision and scale.

1. Update for transaction\_type = 3:

This update handles transactions of type 3 differently. Instead of decreasing the account balance, it increases the balance by the transaction amount but still subtracts the transaction fee. This could represent a deposit or a refund. We update the total\_balance accordingly to show the account balance after the transaction is applied.

1. Update for transaction\_type = 4 (source account):

For transactions of type 4, our statement targets the source account, decreasing its balance by both the transaction amount and the transaction fee. We also update the total\_balance in the transaction table to reflect the new balance after the transaction. This scenario typically represents a transfer or payment where funds are deducted from the source account.

1. Update for transaction\_type = 4 (destination account):

This update specifically focuses on the destination account for transactions of type 4, indicating the receiving end of a transfer. Here, the balance of the destination account is increased by the transaction amount, and no transaction fee is subtracted. This is because fees are usually deducted from the sender's account during a transfer.

1. Update client age:

This maintenance operation involves updating the age of our clients in the client table. We calculate the difference in years between the client's date of birth (DOB) and the current date (CURDATE()), effectively updating the age to the client's current age. This ensures that the age field remains accurate over time without the need for manual adjustments each year.

For all these update operations, it's crucial to verify that the appropriate transaction types exist and that the transactions are associated with the correct account and transaction type records. These updates are designed to maintain the integrity of our database, preventing issues such as overdrafts or incorrect balance calculations. Additionally, we use proper locking mechanisms to avoid race conditions, which could lead to inconsistent data when multiple updates occur concurrently.

**Delete Operations:**

We are using this SQL statement to remove specific records from the transaction table within the bank database. We're focusing on what we can call "void transactions," and we can tell this from the conditions we've set in the WHERE clause of the statement.

Here's a breakdown of our DELETE operation:

1. We begin by saying USE bank; which instructs the database system to switch to the bank database. All the actions we take after this will be within the context of this specific database.
2. Next, we specify DELETE FROM transaction, indicating that we want to delete records from the transaction table. It's important to remember that the DELETE operation is quite powerful, and it permanently removes data that matches the conditions we set.
3. We then use WHERE transaction\_amount IS NULL to filter the records and identify those rows where the transaction\_amount column contains a NULL value. In the context of banking, we consider a transaction with a NULL amount as incomplete, invalid, or unprocessed, which is why we're referring to them as "void" transactions.
4. Additionally, we include AND tran\_type\_id <> 5. This means that not only should the transaction\_amount be NULL, but also the tran\_type\_id (transaction type identifier) should not be equal to 5. This suggests that we're excluding transactions of type 5 from this deletion operation, likely because a transaction type of 5 might have a valid reason to have a NULL transaction amount, such as indicating a certain kind of pending or non-monetary transaction.

With these conditions combined, our DELETE operation will examine the transaction table in the bank database, find all records where the transaction amount is missing (i.e., it's NULL) and where the transaction type is not 5, and then proceed to delete those records.

Before executing this DELETE command, it's generally a good practice to run a SELECT query using the same WHERE clause to review which records will be affected. This helps ensure we understand the impact of the deletion because once the DELETE operation is executed, the data removal is irreversible, unless there are backup systems or rollback mechanisms in place. Also, we need to be aware that if there are any referential integrity constraints, like foreign keys, dependent on the rows we're deleting, the database system may prevent the operation to maintain data integrity and prevent issues like orphaned records.

**Business Questions:**

Question- 1: What is the total revenue generated from transaction fees for a given time period ?

SELECT SUM(tt.transaction\_fee) AS total\_revenue

FROM transaction t

JOIN transaction\_type tt ON t.tran\_type\_id = tt.tran\_type\_id

WHERE t.date\_of\_transaction BETWEEN '2023-01-01' AND '2023-12-31';

**Total transfer fee generated from transactions is $37 in whole 2023 year**

Question- 2: Which accounts have a negative balance, and how many such accounts exist?

SELECT a.account\_id, a.balance\_account

FROM account a

WHERE a.balance\_account < 0;

SELECT COUNT(\*) AS negative\_balance\_accounts\_count

FROM account

WHERE balance\_account < 0;

**There are no accounts with negative balance in the account in all branches**

Question-3: What is the average transaction amount per account type for the last quarter?

SELECT aty.account\_type, AVG(t.transaction\_amount) AS average\_transaction\_amount

FROM transaction t

JOIN account a ON t.source\_account\_id = a.account\_id

JOIN account\_type aty ON a.account\_type\_id = aty.account\_type\_id

WHERE t.date\_of\_transaction >= DATE\_SUB(CURDATE(), INTERVAL 3 MONTH)

GROUP BY aty.account\_type;

**Average transaction amount of each type is displayed below, student account has the highest average transaction amount**

Question-4: Which clients have performed transactions over a certain high-value threshold in the last year?

SELECT DISTINCT c.client\_id, c.first\_name, c.last\_name

FROM client c

JOIN account a ON c.client\_id=a.account\_id

JOIN transaction t ON a.account\_id = t.source\_account\_id

WHERE t.transaction\_amount > 500 -- assuming the threshold value is 500

AND t.date\_of\_transaction BETWEEN '2023-01-01' AND '2023-12-31';

**Only two clients ELLIS KIRK and SCOTTY ARMOUR have done transactions over the threshold which is more than 500.**