Final Project: E-Banking Application

Amina Al-Helali (101230205)

Emma Souannhaphanh (101160597)

Carleton University

BIT2008A: Multimedia Data Management

Mohammad Mahdi Heydari Dastjerdi

December 9, 2022

Table of Contents

Functionalities Interpretation

Clients

We will make a client table to store data regarding their user_id, first name, last name, and password. We will create a separate table to store a client's phone number and another table to store their address since these are multi-valued attributes.

Accounts

The account entity will have relationships with client, statements and transactions. With client, it will form a client-account relationship where a client can have multiple accounts and an account can have multiple clients. In that table there are also attributes for the client roles that a client can have within the account. With statements, it will form a statement relationship, where the associated account will be the source account in the statement. With transactions, it will form a transaction_to relationship, where the transaction will contain the account that the money is being deposited/withdrawn from. Account will also have a required signatures attribute that will be used to determine if the associated account has enough signatures to be confirmed.

Statements

The statement entity will have relationships with accounts, transactions, and clients. With transactions, it will form a relationship where the source account of the statement is the where the transaction is coming from. With accounts, it will form a relationship, where the source account is the account it is associated with and later when the statements need to be confirmed it will check if the required amount of signatures from the account are met. With a client, it will form a relationship of pay, initiate or sign. The statement will be divided 3 tables: statements which contain all of the basic information about the statements including the initiator, statement_signer which will contain the clients that have a sign role and are associated with the statement source account, and statement_confirmation which contains the client that has a pay role and is associated with the statement source account and also it will contain information on whether the statement has been confirmed or not.

Other Considerations

We made a few assumptions about the functionalities because we were a bit confused. Assumptions we made:

- Banking functionality is just a description of the project
- Cosigners are just clients with a sign role
- Account owners are just clients who have an account
- View roles can be false even if the other roles are true. Clients can still associate with a statement even if only the view role is true.
- The number of required signatures needed for a statement will be an attribute in the account table and will have to be less than the number of cosigners for that account.
- Can have 0 required signatures and 0 cosigners
- If a transaction type is a 'deposit' then money will be subtracted from the statement total. If 'withdrawal' then it will be added.
- Pay and confirmed mean the same thing

Design

Design 1 - Entities

STRONG **Client**
STRONG **Account**
WEAK **Transactions** CORRESPONDS WITH **Statements**
STRONG **Statements**

Design 2 - Entity and table attributes

client

client_id	**INT** Primary Key
first_name	**VARCHAR(30)** not NULL
last_name	**VARCHAR(30)** not NULL
password	**VARCHAR(50)** not NULL
lastModified	**TIMESTAMP** default now()

client_phone

client_id	**INT** Primary Key, Foreign Key(references client, client_id)
phone_numbr	**NUMERIC (10, 0)** not NULL
lastModified	**TIMESTAMP** default now()

client_address

client_id	**INT** Primary Key, Foreign Key(references client, client_id)
address	**VARCHAR (100)** not NULL
lastModified	**TIMESTAMP** default now()

account

account_id	**INT** Primary Key
total_balance	**NUMERIC(10,2)** default 0
account_type	**VARCHAR(20)** default ‘savings’ check (account_type = “savings” OR account_type = “checkings”)
num_cosigner	**INT** default 1
required_signatures	**INT** default 1 Check (required_signatures =< num_cosigners)
lastModified	**TIMESTAMP** default now()

client_account

client_id	**INT** Primary Key, Foreign Key(references client, client_id)
account_id	**INT** Primary Key, Foreign Key(references account, account_id)
sign_role	**BOOLEAN** NOT NULL
view_role	**BOOLEAN** NOT NULL
pay_role	**BOOLEAN** NOT NULL
lastModified	**TIMESTAMP** default now()

statements

statement_id	**INT** Primary Key
note	**VARCHAR(100)** default ‘ ’
source_account	**INT** Foreign Key(references account, account_id)
initiator_client	**INT** Foreign Key(references client, client_id)
total_amount	**NUMERIC(10,2)** default 0
lastModified	**TIMESTAMP** default now()

transactions

statement_id	**INT** Primary Key, Foreign Key(references statements, statement_id)
amount	**NUMERIC(10,2)** Primary Key default 0
type	**VARCHAR(30)** Primary Key default “withdrawal”, check (type = “withdrawal” OR type = “deposit”)
time	**TIMESTAMP** Primary Key default now()
note	**VARCHAR(100)** Primary Key default ‘ ’
trasaction_to	**INT** Primary Key, Foreign Key(references account, account_id)
lastModified	**TIMESTAMP** default now()

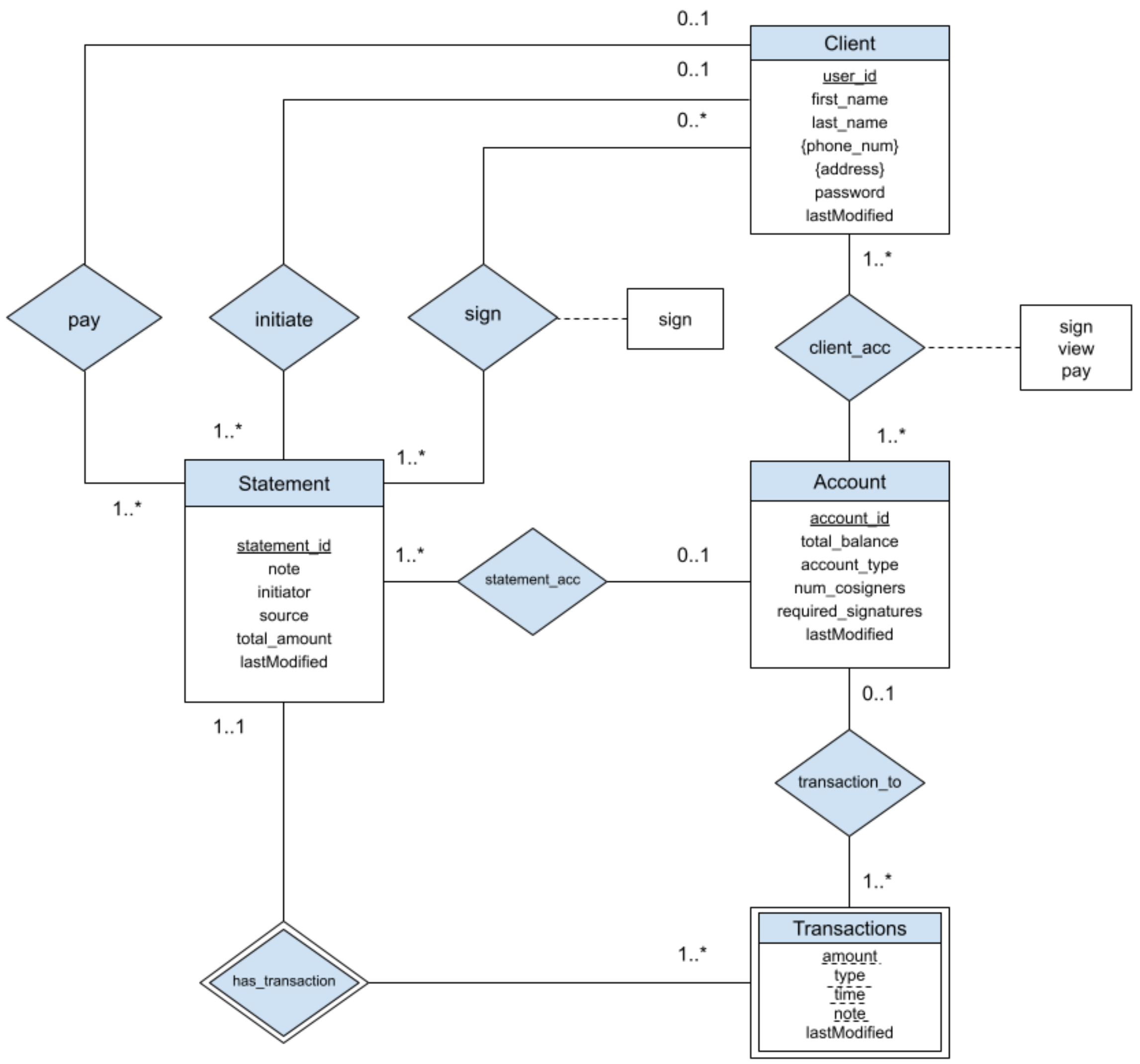
statement_signer

statement_id	**INT** Primary Key, Foreign Key(references statements, statement_id)
signer_id	**INT** Primary Key, Foreign Key(references client, client_id)
sign	**BOOLEAN** DEFAULT FALSE
lastModified	**TIMESTAMP** default now()

statement_confirmation

statement_id	**INT** Primary Key, Foreign Key (references statements, statement_id)
payer_id	**INT** Foreign Key (references client, client_id)
confirmed	**BOOLEAN** DEFAULT FALSE
lastModified	**TIMESTAMP** default now()

Design 3 - Entity-Relationship Diagram



Design 4 - Normal Form

Our relational database design has redundancy in certain places to help improve the performance in simplifying how triggers and functions are used.

One way we preserved the normal form and avoided redundancy is by separating the multivalued attributes, address and phone number in the clients into their own tables, with the client_id and phone number/address as the primary keys. This avoids redundancy because it avoids repeating rows in the clients table if a client were to have several phone numbers or addresses, which would decrease performance.

Redundancy was reduced by creating a separate table to store the signers of a statement. Since the statement has a many-to many signer relationship with the client we created a new table containing the signer and statements id along with whether the statement has been signed. This avoided redundancy because if we were to store the signers in the statement table, there would be several repeating rows for each statement if there is more than one signer and would be redundant.

Another way we avoided redundancy was by creating a table to represent the relationship that the client has with an account and the roles they have within it. This reduces redundancy because if we did not include this table, we would have to store the account that the client is associated with and their roles in the client table. That would be bad since **client** and **account** have a many-to-many relationship and there would be repeating rows in the client table if a client is associated with more than one account.

One way we could not avoid redundancy is by having a separate table for **statement_payer**, which contains the *statement_id*, *payer_id* (who is a client) and *confirmed*. This is redundant because there is a many-to-one relationship for the payer relationship, with **statements** having total participation and being on the many side. We could have just added the payer (*client_id*) to the statements table but then we found that it would cause some problems when creating the triggers for the statements table.

Another way we were not able to avoid redundancy was by having a weak entity set, **transactions**. We could remove the transaction entity but that would result in a loss of information. Therefore, redundancy is necessary in this case.

Design Rationale

Client and Account Relationship: Many-to-many

We interpreted the relationship between **client** and **account** to be many-to-many with total participation on both sides. This is because there must always be at least one account associated with a client, and there must always be a client associated with an account. In addition, there can be multiple clients associated with a single account (e.g. a joint account with three clients participating), and there can be multiple accounts associated with a single client (e.g. one person can have two savings and a checking account).

To represent this many-to-many relationship, we needed to create a new table called **client_account**. The table will have the foreign keys *client_id* and *account_id* to associate them both with **client** and **account**. In order to implement the functionality of each client having access to different accounts under certain roles (sign, view, pay), we created boolean attributes for each role in **client_account**.

Options for Attribute Values

We decided that if a value cannot be predicted, then the NOT NULL constraint will be added. Most values are designated NOT NULL; for example, we cannot predict a client's name, therefore the attribute *first_name* will be given the NOT NULL constraint. If a value can be predicted, then a DEFAULT value will be added. For example, we can make the total balance of an account 0 by default.

Account Type Restriction

Accounts will only have two types: “savings” or “checkings”. The account type will be a VARCHAR attribute that is of the savings type by default. We made sure the account type could only be savings or checkings by adding a CHECK constraint.

Weak and Strong Entities

When identifying the weak and strong entities we only focused on the four main entities outlined in the functionalities: **client, account, transaction, and statement**. **Client**, **account**, and **statements** were made strong entities because they can exist independently and each has a key that can uniquely identify each row. However, transactions is a weak entity because it does not have a unique identifier and its existence depends on whether a statement exists. Since the transactions' existence depends on the statements, we made **statements** the transactions entity's identifying entity and used *statement_id* and the attributes of the transactions to uniquely identify it.

Implementation Features

NOTE: We did not include any views in our code implementation because it was not specifically specified where we should use them, and we could not find a necessity for them in our implementation.

Logging/Queries Assumptions

- Only created functions for queries where requested input was ambiguous (for ex: Show the list of all transactions to a **certain account** that is not paid.)
- Declined signatures means where the client with a sign role has *sign* set to FALSE in the **statement_signer** table. (5) Show the list of all declined signatures of a certain client.)
- ‘Transaction that is initiated by a certain user’ Initiator of the statement that the transaction belongs to (8) Show the list of every transaction that is initiated by a certain user but the person does not have the ”sign” permission on it.)
- Only show deposits for transactions whose associated statements have been confirmed (9) Show the list of every deposit into a certain account that is above a certain amount.)

Operations

- When implementing the operations, we decided to use functions to implement them where the user would have to input the fields they would like to insert or edit/update.
- If the operation was associated with a client’s role, an if statement was added in the function to verify that the client has the role to perform an operation.

Triggers

Triggers for Transactions and Statement Total Balance Update

Assumptions: transactions cannot be updated

Triggers, can_transaction_insert_trigger BEFORE INSERT, can_transaction_update_trigger BEFORE UPDATE, and can_transaction_delete_trigger BEFORE DELETE, on the transactions table will call check_if_confirmed_transactions function. Function will check if the trigger operation was ‘UPDATE’, if yes, then an error will be displayed, informing that a transaction cannot be edited. The function will also check if the statement that the transaction is associated with is confirmed, if yes, then the transaction will not be inserted/deleted.

Trigger, set_statement_total_trigger AFTER INSERT on the transactions table will call set_statement_total function. The function will insert the transaction amount into the statement total balance if the transaction is added. Checks if the transaction type was ‘deposit’, if yes then the transaction amount will be subtracted from the statement total balance. If the transaction type is ‘withdrawal’ then the transaction amount will be added to the statement total balance.

Trigger, set_statement_total_delete_trigger AFTER DELETE on the transactions table will call the set_statement_total_delete function. The function will redo the transaction amount from the satsemnt_total balance if the transaction is deleted. Checks if the transaction type

was ‘deposit’, if yes then the transaction amount will be added to the statement total balance. If the transaction type is ‘withdrawal’ then the transaction amount will be subtracted to the statement total balance.
NOTE: since a statement cannot be edited if it has at least one signature, we added a pg_trigger_depth() function in the edit_statemnt_trigger, that checks if this trigger was called from another trigger, if it was then it will not call the function. This is done so we can still edit the statement total balance when transactions are added/removed even when the statement has at least one signature.

Triggers for Updating Accounts on Each Money Transfer

Triggers, update_account_balance_trigger AFTER UPDATE, and update_account_balance_insert_trigger AFTER INSERT, on statement_confrimation will call update_account_balance. The function will update the statement source account and all of the to-accounts in the transactions associated with the statement. First, whether the statement is confirmed is checked. The source account will be updated. Then there is a for loop for all of the transactions associated with the statement, and the to-account balances are updated.

Triggers for Role Checking/Verification

There are several triggers on the tables associated with statements that role check the clients.

For **statements**, the initiator will be checked BEFORE INSERT to make sure that they are associated with the source account.

For **statement_signer**, the signer will be checked BEFORE INSERT to make sure that they are associated with the source account and have the signer role.

For **statement_confirmation**, the payer will be checked BEFORE INSERT to make sure that they are associated with the source account and have the pay role.

We implemented these triggers to make sure that the clients in the tables actually have the appropriate roles and are associated with the right source account.

We did not check the roles on update because we are assuming the client cannot be changed, only inserted or deleted.

Triggers for Editing/Deleting statements

There is a trigger for BEFORE UPDATE on statements and BEFORE DELETE on statements. Both call the same function, statement_edit_delete. In that function, we check if the statement has been confirmed. If it has, then the statement will not be allowed to be deleted or edited. There will also be a check to see if there is at least one signature on the statement and the trigger operation was ‘UPDATE’, if yes the statement will not be allowed to be edited.

Verification of Account Operations

Assumption: Account operations is referring to statements

Sign/Unsign

Triggers, sign_unsign_trigger BEFORE UPDATE and BEFORE INSERT on statement_signer table. Checks if the statement is already confirmed, if it is, then the client will not be able to sign/unsign.

Pay

Assumption: pay and confirm are the same thing.
Triggers, confirm_statement_trigger BEFORE UPDATE and confirm_statement_insert_trigger BEFORE INSERT on statement_confirmation. Calls function confirm_statement. The function checks if the statement is already confirmed, if yes then the statement will not be allowed to be confirmed since it is already confirmed. It also checks if the number of signatures for the statement in the statement_signer table is less than the number of required signatures listed in the statement's source account, if yes then the statement will not be able to be confirmed.

Create

Assumption: Assuming it means when a statement is created.
Trigger, verify_intitator_trigger BEFORE INSERT on statements. Calls verify_initator function. Function checks that the initiator client is actually associated with the source account. If they are not, the statement will not be created.

Edit

Assumption: Assuming it means when a statement is edited.
Trigger, edit_statement_trigger BEFORE UPDATE on statements table. Calls function statement_edit_delete. Function checks if the statement has already been confirmed, if it has then editing will not be allowed .It also checks if there is at least one signature on the statement, if yes then editing will not be allowed.

Redundancy in the Code

One major redundancy in the trigger code was checking for whether the statement was confirmed, in multiple functions. While it is redundant, it is necessary to make the code more readable.
Another major redundancy was having very similar code for updating the statement total balance when a transaction was added or removed. There were two separate functions while the code was very similar. However, it was necessary because different operations would have to occur depending whether the transaction was added or deleted.

Storing Account Operation History

We were tasked with storing the history of all operations on an account including sign, unsign, pay, and initiation. Even though the task says to store the history of operations on an account, we interpret the task as storing operations on the statement that is associated with an account instead.

We decided that storing operation history means having audit tables that keep track of table updates in statement and statement-related tables.

Since we created multiple tables for statements – **statements**, **statement_signer**, and **statement_confirmation** – we decided to separate the operation logging history in order to simplify the function and trigger calls.

We divided the operations into three operations:
1. Initiation
2. Sign or unsign
3. Pay

We did this because each of these operations are not associated to the same, single statements table.

The operation "initiate" is associated with the statements table. The operation "sign/unsign" is associated with the statement_signer table. The "pay" operation is associated with the statement_confirmation table.

We created an audit table for each of the three operations:
1. **statement_initiate_audit**
2. **statement_sign_audit**
3. **statement_pay_audit**

Each new audit table has the new attributes *audit_id* and a *tstamp* that stores the timestamp of each statement operation, in addition to some duplicates of the **statements** attributes. The audit tables will be weak entities that use primary keys (*statement_id*, *client_id*, etc.) of the associated statement table as their own primary key. We decided to only include attributes that we considered useful or necessary for an audit of statement/account operations that were from the original tables.

For **statement_initiate_audit**, we deemed only keeping *statement_id, source_account,* and *initiator_client* were necessary to identify the statement, from which account the statement was initiated on, and by which client.

For **statement_sign_audit**, we deemed only keeping *statement_id*, *signer_id*, and *sign* was sufficient to identify the statement, the client who signed the statement, and whether the operation was signing or un-signing.

For **statement_pay_audit**, we deemed only keeping *statement_id*, *payer_id*, and *confirmed* were necessary to identify which statement was paid by which client.

Storing Most Recent Table Edits

The task was to store the last time the tables were edited. In order to avoid creating many more tables to store edit history for every single table, we decided to add a new column to each table called *lastModified* that is updated whenever a table is updated. The task only asks for storing the last time the tables were edited and not a complete edit history so simply adding a new column for timestamps is sufficient.

By default, the *lastModified* attribute will have a timestamp of value CURRENT_TIMESTAMP.

We created the function “update_timestamp()” that will update the *lastModified* attribute to CURRENT_TIMESTAMP. Then, we created triggers for every single table that will call “update_timestamp()” BEFORE UPDATE.

Automating Table Logging

We used triggers for each table in order to automate logging of the last modification of table edits. By using triggers, we make table logging automatic and avoid the need to call a function every single time to update the *lastModified* column.