Databasur og SQL

Dogebank - A Banking System

Brandur Arnoarson - 2018.160
Jógvan í Garði Patursson - 2018.208
Bartal Jógvansson Djurhuus - 2020.280

Databasur og SQL 5038.20

NÁTTÚRUVÍSINDADEILDIN
Faculty of Science and Technology

FRÓÐSKAPARSETUR FØROYA
University of the Faroe Islands

Heiti / Title	***Dogebank - A Banking System*** Dogebank - Ein Bankaskipan
Høvundar / Authors	Brandur Arnoarson, Jógvan í Garði Patursson og Bartal Jógvansson Djurhuus
Vegleiðari / Supervisor	Hans Blaasvær , Fróðskaparsetur Føroya
Ábyrgdarvegleiðari / Responsible Supervisor	Hans Blaasvær, Fróðskaparsetur Føroya
Ritslag / Report Type	Uppgáva í Databasum og SQL
Latið inn / Submitted	3. juni 2021

NVD*Rit*	2013:08
©	Náttúruvísindadeildin og høvundarnir 2021
ISSN	1601-9741
Útgevari / Publisher	Náttúruvísindadeildin, Fróðskaparsetur Føroya
Bústaður / Address	Nóatún 3, FO 100 Tórshavn, Føroyar (Faroe Islands)
☎ • 🖷 • @	+298 352 550 • +298 352 551 • nvd@setur.fo

1 Abstract

In this paper we set out to build a banking system with the help of a Database Management System built upon *PostgreSQL*. We built the Graphical User Interface around the framework ASP.NET Core. To host the database, we installed the PostgreSQL system onto a Ubuntu 20.04 server hosted by the University of the Faroe Islands. Before the implementation of the system itself, we outlined the construction of the system using ER diagrams and Table Diagrams, and also sketched the GUI using computer software.

The construction of the backend and frontend are described and visualized in this project. Furthermore, we touch upon central aspects of the theory involved, and explain their meaning and how they are implemented in the system. The result, a functioning banking system, as well as the process of building it are then analyzed and discussed in the end. During this segment, we find that there are close to an infinite number of ways in which this project could have been constructed, some ways easier, some ways better, and some quicker. But we find that this was the best way for us to implement the system at this point in time.

Contents

2 Introduction

We were assigned this Banking System project during the skyrocketing fame of the cryptocurrency Dogecoin. And therefore we thought it would be very fitting to call it "Dogebank" (an almost-homophone of Deutche Bank). Who wouldn't want to keep track of your own Dogecoins while simultaneously being able to transfer Dogecoins for your kids' lunch money. Is your wife spending too much Doge on shoes again? You can keep track of her spendings with this system too. With this system, you could seamlessly deposit, withdraw and transfer Dogecoins without the assistance of any bank employee. Do you wish to implement this software to start up your own Doge affiliate bank? No problem! This system leaves your employees practically out of work, because everything is automized. Calculating interest rates, executing cash drafts and even setting up new bank accounts all happens automatically. Heck, you might not even need employees.

3 Problem Statement

With the theories of this course as baseline, this project seeks to construct and implement a Database Management System run using PostgreSQL. This system will serve as backend for a fully functional banking system with a Graphical User Interface. The criteria for the project include the construction of an ER-diagram and the implementation of said diagram into the system. Moreover it is required to implement central features of SQL i.e. triggers, stored procedures, constraints and aggregated views. Finally we want to explain the process in which the result was acquired.

4 Design

During the design phase, we started by roughly outlining the structure of the system by the use of simple Table diagrams. Thereafter we made a sketch of the GUI with the help of a blackboard. Later on we sharpened the scope of our Table diagram by using the principles of the Entity Relationship Diagram Model (ER diagram). This helped us figure out the relational aspect of the system, and in which direction the various elements of the diagram were related to each other. We will go into further depth of ER-diagrams in the following segment, as well as showing the diagram related to our project. With these blueprints at hand, we started building the database.

ER Diagrams
When designing the ER Diagram it is important to get a list of all the different things the system should consist of. In our case these would be e.g. a customer, an account, a transaction, and so on; these are represented as entities. And the relationship between them can be represented in an ER diagram. In the case of the sets customer, and account, a has-a relationship is represented. Because the customer can have many accounts, and an account can only have one customer, the relationship is a one-to-many relationship. All of our entities have a binary relationship. Figure 1 shows the ER diagram.

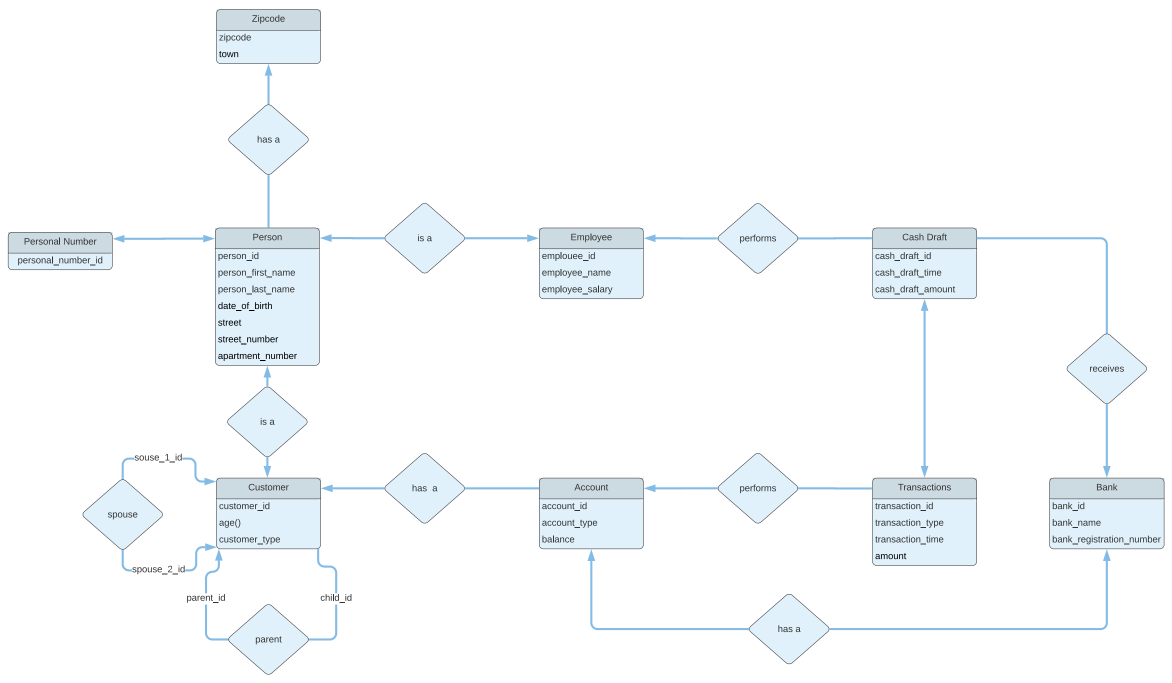


Figure 1: Entity Relationship Diagram

Normalization is used to reduce redundancy in database tables. The higher the normal form in a database, the less redundancy is present.[3]
First Normal Form is the lowest form of normalization and the bare minimum to create a decent database. The requirements to achieve this are single valued columns, values of same type in columns, unique name of columns, and indifference to row order.[3]
Second Normal Form fulfills 1FN and should not have partial dependency. If an attribute is only partly dependent on a primary key is has partial dependency.[3]
Third Normal Form fulfills 2FN and should not have transitive dependency Transitive dependency is when attributes are dependent on non primary key attributes instead of primary key attributes.[3]
We have made sure that all these requirements from 1FN to 3FN are met in our database.

In figure 2 the table diagram is shown. It shows the transformation in the ER diagram from the entities and relationships to tables. The has-a relationship between customer and Account has become a CustomerHasAccount table. The arrows indicate the one-to-one, one-to-many, many-to-one, or many-to-many relationship. For example, the customer can have multiple accounts, therefore Customer to CustomerHasAccount is a one-to-many relationship. And the CustomerHasAccount to Account relationship is a one-to-one because each account only appears once in the CustomerHasAccount table.

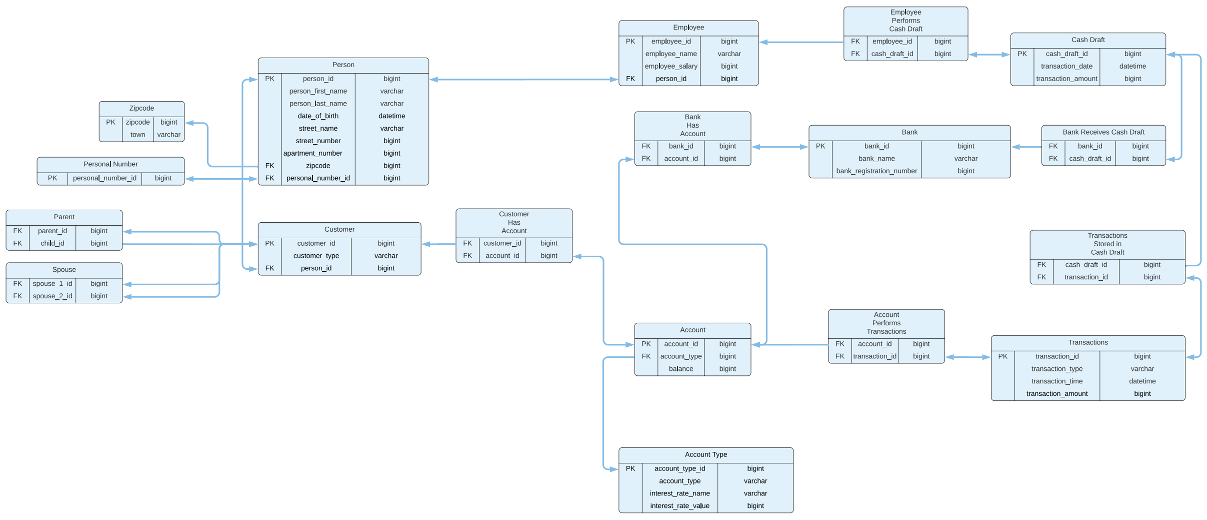


Figure 2: Table Diagram

In figure 3 the use case diagram for the customer is shown. The **Login** use case only interacts with the front end. **Choose Self Customer**, **Choose Spouse Customer**, and **Choose Child Customer** can be chosen after logging into the system. This will result in showing the available accounts to user. **Choose Account From Customer** selects an account to perform actions on. This shows the balance and past transactions of that account. If it is the users own account is it possible to choose **Deposit Into Account**, **Withdraw From Account**, and **Transfer To Account**. These function allows the user to input a balance and account numbers to execute a deposit, withdrawal or transfer.

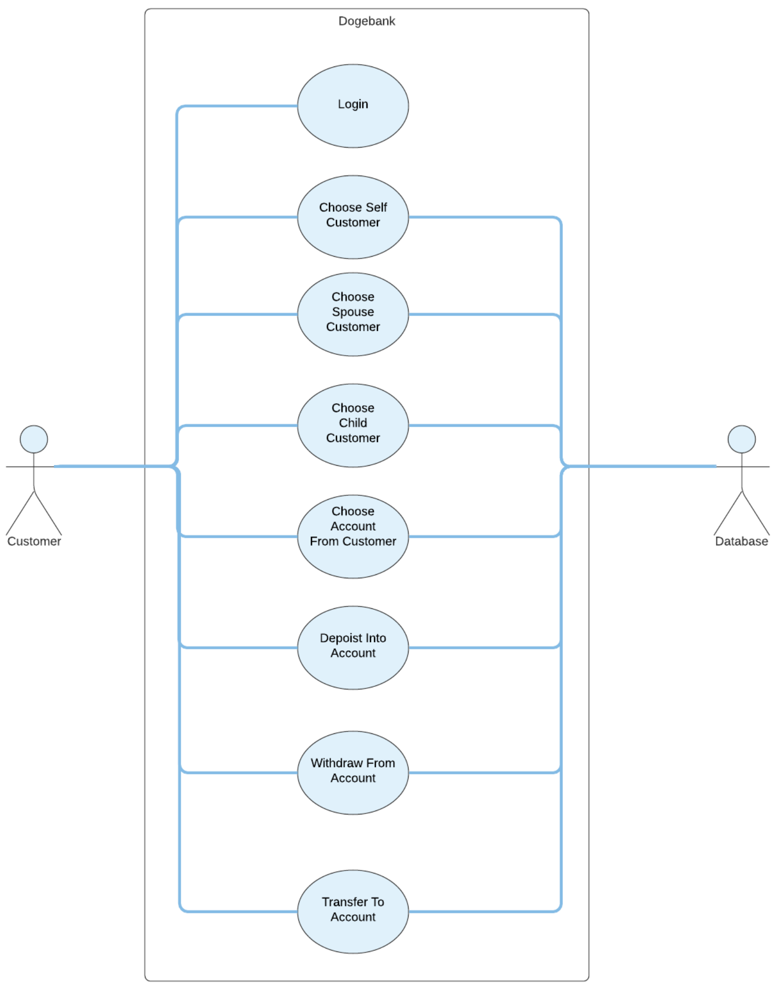


Figure 3: Use Case Diagram, Customer

In figure 4 the use case diagram for the employee is shown. These are the functions which the user does not see. In reality these are the functions which we executed from our IDE to simulate a employee. **Performs Cash Draft** takes the list of Cash Draft and withdraws and deposit money to the banks account. **Execute Interest Calculation** is a procedure which calculates interest on a bank account and updates the balance accordingly. It could be made with a timestamp trigger. But for the purpose of demonstration it is executed manually. **Create Person**, **Create Customer** and, **Create Account** are all stored procedures with insert functions that create an instance of Person, Customer, and Account, respectively.

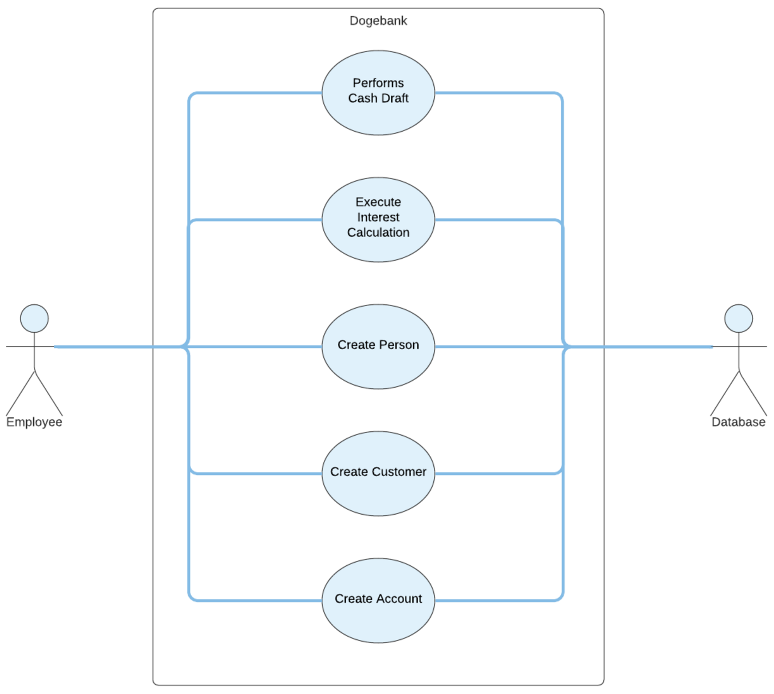


Figure 4: Use Case Diagram, Employee

4.1 GUI

During the preliminary days of planning, we discussed how the frontend should look. As most people, we had some preconceptions how an online bank should look, and therefore started to sketch out a rough idea of how the GUI should look like on a blackboard. After the initial sketch we moved onto using software to draw out the segments and elements. In this case, we used Fluid UI to produce the following outlines:

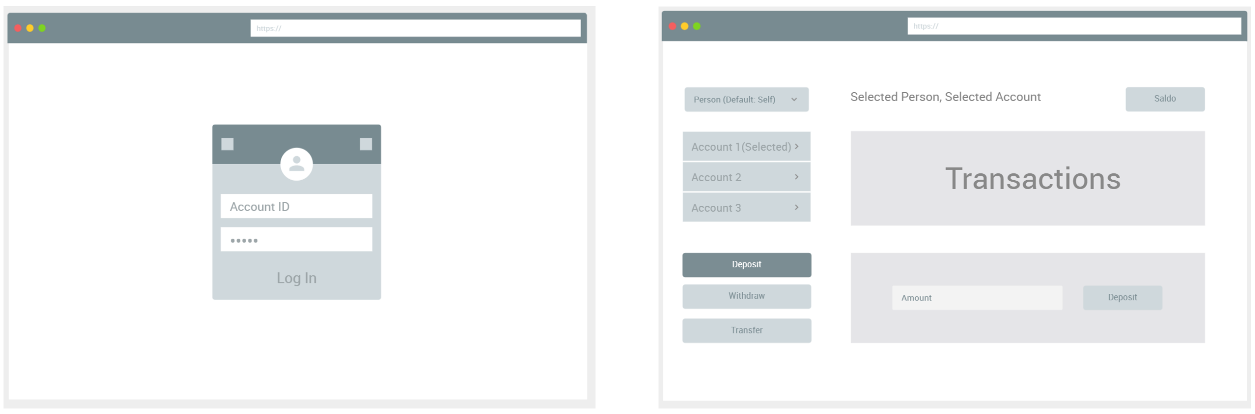


Figure 5: GUI login and interface pages

This served as a good starting point, but not neccessarily a direct goal point, given that the project did not require a perfectly designed interface. So we designed the GUI as we saw fit, given the timeline. In the segment 'Future Work' we have added some screenshots of how the GUI looks like at the time of writing.

5 Product

In this section we will be introducing the core concepts of what makes our system work.

5.1 Linux

As we needed to host our database somewhere, we looked into all the popular hosting services, such as Amazon and Google etc., but they all had some limitations. So we chose to host our database on a Linux machine similar to the one we have used in a previous course about Linux. We could have chosen between several Linux distros and even Windows to install the server onto, but we chose Ubuntu 20.04 purely because of familiarity from the above-mentioned course.

Doing research about getting Oracle to run on Ubuntu, we learned that getting it to run on Ubuntu would require a great deal of manual installation. Furthermore, Oracle was mainly recommended to be run on Windows. While it was possible to run on other OS'es, it was mainly recommended to be run on other distros than Ubuntu, as they could be installed more automatically. So this left us with the options of doing a manual installation, switching to Windows or switching SQL client.

We chose PostgreSQL, as it works well with Ubuntu and has a lot of documentation. PostgreSQL and SQL have some differences, but these differences are negligeble for our usecase. As long as it supports stored-procedures well, it is a good enough method for us.

This said, it is entirely possible to host the database locally, which we also have heavily relied on during the construction phase. This feature is very helpful for the programmer, as it saves him the trouble of losing connection to the server, the server losing power during a blackout, or other irregularities often encountered in an industry relying on various individual parts functioning in unison.

5.2 Database Management Systems "DBMS"

"A Database Management System is a collection of interrelated data and a set of programs to access those data." [3] When setting up a banking system as ours, it could entirely be possible to manage every entry manually by using something as simple as an Excel spreadsheet. But when you have, say a thousand or more customers, this process would be too cumbersome. Therefore it helps us immensely to implement a DBMS for our project.

DBMSs are built specifically for these types of things. It is much simpler to treat the different types of data as different entities and to store the information in such a way that it is easy to get entries that relate to eachother in some way. Our DBMS uses PostgreSQL, runs on a Linux server with access through an SSH tunnel, and data is presented to the user through our ASP.NET GUI, while some internal commands are run either through a shell like PSQL or an IDE like VSCode with the proper extensions.

With such a system in hand, we are able to properly differentiate when different data is needed, and who should have access to handle specific data. Though it is a possibility, for the purposes of this project, we will not be implementing user access control.

5.3 PostgreSQL

PostgreSQL is an open-source object relational database management system originally underwent development as POSTGRES at Berkeley University in 1986. The system has gone through many changes over these past years, but one of the big changes was the implementation and compatability with SQL, hence the name change to PostgreSQL. [1].

5.3.1 ACID

Another important feature has been the compliancy of the ACID property requirements. The ACID are a set of strict enforcement on the properties of a DBMS. Let's break up the accronym, and look at each requirement closer in accordance to the book of the course:

- Atomicity - "Either all operations of the transaction are reflected properly in the database, or none are."[3]
- Consistency - "Execution of a transaction in isolation (i.e., with no other transaction executing concurrently) preserves the consistency of the database."[3]
- Isolation - "Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j, it appears to T_i that either T_j finished execution before T_i started or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system."[3]
- Durability - "After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures."[3]

Atomicity means we have to ensure that transactions are properly executed on the database, and that actions that belong to a single transaction should either all happen or all be avoided. For instance, when making a transfer between two accounts, it should never be possible to insert into the receiving account without also subtracting from the sending account. A thing of note for this project is that psql has auto-commit turned on by default, so most single action transactions do not need an explicit commit to ensure the transactions completeness. Transactions with multiple operations are surrounded by BEGIN and either ROLLBACK or COMMIT to ensure atomicity.

Consistency means that we have to ensure the consistency of the database after every transaction. An inconsistent state is a state where one or more values in the database break the constraints set up in the relational schema. This could e.g. be if we are trying to delete person from our database, while we still have a customer that refers to this person, our FOREIGN KEY constraint on customer would point to an empty row (a NULL value), and thus leave the database in an inconsistent state.

Isolation means that the database system needs a service (*concurrency-control*) to interleave the commands of the different operations in such a way that read operations of the different transactions are performed at a point the reflects the database at it would be had the different transactions been performed serially.

Durability ensures that all data resulting from a transaction are properly recorded in persistent storage (disk). To ensure this, Postgres uses Write-Ahead Logging (*WAL*). Write-Ahead Logging means that transactions are not recorded to disk immediately upon execution, but rather a log that is sufficient to replicate the transaction are written to disk. In time, these logs will be propagated to the tables on the disk, and in the case of crash, the server can look through the WAL log to recover any transactions that have not yet been written to disk. The eventual propagation of transactions to the disk is performed by the checkpointer, which continuously works on reading the logs and writing the changes to disk. An added benefit of this is the fact that the checkpointer only moves from one persistent, atomic and consistent system to another, and can therefore write from the logs in an order that ensures fewer I/O calls to disk, it can prioritize writing operations that are stored close to eachother on disk.

5.3.2 Relevant Features of PostgreSQL / SQL

DDL

DDL stands for Data Definition Language, and is the language used for creating the database schema to be used in the solution. This includes creating the relations needed in the solution, specifying what types are used for each attribute in the tuples, as well as specifying constraints for the different attributes and relations.

CREATE is used to create tables, views, indexes, stored procedures, and triggers. When creating tables, the types of values to be used need to be specified, as well as the constraints to use on the different values.

ALTER is used to alter tables and columns. It can be used to add constraints, remove constraints, change types of values or adding new values to a table.

DROP is used to remove the different entities created with CREATE from the database. DROP has no regards for values in a table, but drops everything that has been added. However, DROP will not allow you to drop tables that have foreign key constraints pointing to it, so before dropping, we need to clear all foreign key constraints that reference this table

VIEW is a virtual table, which in essence is a shorthand for another SQL statement. This allows for quicker access to specific presentation of data. In our case, we are using a view with the aggregation method SUM(), which we use to sum up the balances of all accounts, then grouping them by type of account, which allows the bank to easier get statistics on the breakdown of money.

The different *values* that can be used when creating tables are in general:

- **INTEGER** - An integer number. Postgres defines this as a signed four-byte integer. For our purposes, we are using bigint, which gives access to a signed eight-byte integer.
- **REAL** - A single precision floating-point number. Numeric can be used to define a floating point number of any given precision.
- **CHAR(*(n)*)** - A string of *n* characters. We are mostly using **VARCHAR*(n)***, which allows for a string of up to *n* characters, but can be shorter as well.
- **TIMESTAMP** - A date and time using ISO format without timezone (TIMESTAMPTZ can be used if timezone is needed)

Constraints

Constraints are central components of the *table definition.* They work to limit the inputs that can be inserted into the rows of the tables. By implementing constraints into our system we can limit the causality of human errors. The simplest types of constraints are:

- **NOT NULL** - specifies that a value has to be given for the entry
- **UNIQUE** - specifies that no other tuple in the table can have the same value in this attribute
- **PRIMARY KEY** - is used to uniquely identify a tuple. It combines *NOT NULL* and *UNIQUE*
- **FOREIGN KEY** - links two tables. Ensures that no action can be taken on the other table that violates the constraints of this table, this could be deleting an entry that is referenced if the *FOREIGN KEY* is set to *NOT NULL*, deleting the column for the constraint or dropping the other table entirely.
- **CHECK*(condition)*** - adds a specific check as a constraint. This can as an example be to ensure that a numeric value is greater than 0, or that an age is greater than 18.

DML

DML stands for Data Manipulation Language, and is the query language used for working with the data on the database.

The **SELECT** statement is a statement to select data from a database. It is often used in conjunction with the **from** statement, in which you can select data from specific tables. The output of the select statement will either be a value (string, numerical, boolean etc.), or a new table of relations will be outputted. The result from the select statement wil be stored in the result-set.

The **INSERT** statement allows you to insert data into the rows of specific tables. In order to insert the values into the rows, it is required to specify the *values* in which to insert, and these have to conform to the constraints set out in the DDL.

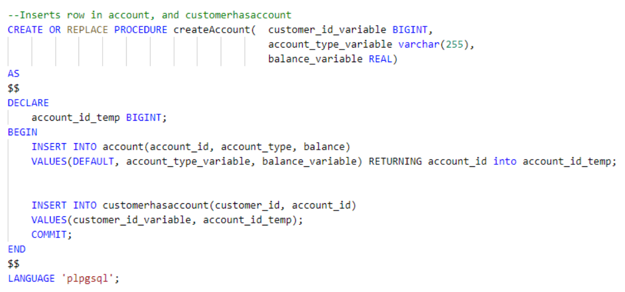
The **UPDATE** statement is used to change values of a record, for instance updating the balance of a bank account when transacions are performed.

The **DELETE** statement is used when deleting a record from a table. It is important when deleting to use the correct *WHERE* clause, as omitting a WHERE clause will remove every single entry from the selected table.

Using the *FROM* clause that selects which table the operation is to be performed on, as well as using the *WHERE* clause to specify which rows in the table to operate on, these four statements can be mixed and matched to generate almost every output from the database that the user wants.

Stored Procedures

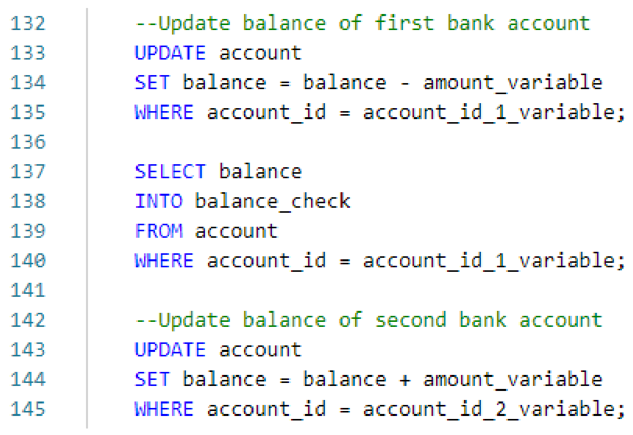
Stored Procedures are functions that are stored for repetitious use in the database. These procedures are useful e.g. if you have queries that are executed multiple times, if the parameter values are never changed, or to save time from retrieving data from the database to the front end, instead just processing the data in the database.



```sql
--Inserts row in account, and customerhasaccount
CREATE OR REPLACE PROCEDURE createAccount(  customer_id_variable BIGINT,
                                            account_type_variable varchar(255),
                                            balance_variable REAL)
AS
$$
DECLARE
    account_id_temp BIGINT;
BEGIN
    INSERT INTO account(account_id, account_type, balance)
    VALUES(DEFAULT, account_type_variable, balance_variable) RETURNING account_id into account_id_temp;


    INSERT INTO customerhasaccount(customer_id, account_id)
    VALUES(customer_id_variable, account_id_temp);
    COMMIT;
END
$$
LANGUAGE 'plpgsql';
```

Figure 6: createAccount, Stored Procedure

Figure 6 shows the createAccount stored procedure. First a new account is created, which uses the trigger that generates account number with the trigger function to evaluate weighted crosssum. Then the new account id is used when creating the connection in the customerHasAccount table. As we can see, we also start this process with the BEGIN keyword. This begins a new transaction that we commit after completion to ensure atomicity.



```sql
--Update balance of first bank account
UPDATE account
SET balance = balance - amount_variable
WHERE account_id = account_id_1_variable;

SELECT balance
INTO balance_check
FROM account
WHERE account_id = account_id_1_variable;

--Update balance of second bank account
UPDATE account
SET balance = balance + amount_variable
WHERE account_id = account_id_2_variable;
```

Figure 7: Transfers Update Balance, Stored Procedure

Figure 7 shows the update balance part of the transfers procedure. The first bank account has its balance subtracted by the given amount, and a local variable is assigned the new balance value. The second account has its balance added by the given amount.

Figure 8 shows the insert transactions part of the transfers procedure. A row is inserted into the transactions

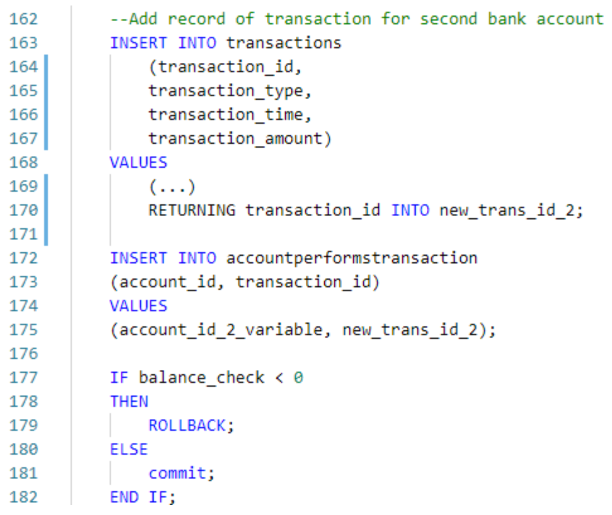

```
--Add record of transaction for second bank account
INSERT INTO transactions
    (transaction_id,
    transaction_type,
    transaction_time,
    transaction_amount)
VALUES
    (...)
    RETURNING transaction_id INTO new_trans_id_2;

INSERT INTO accountperformstransaction
(account_id, transaction_id)
VALUES
(account_id_2_variable, new_trans_id_2);

IF balance_check < 0
THEN
    ROLLBACK;
ELSE
    commit;
END IF;
```

Figure 8: Transfers Insert Transactions, Stored Procedure

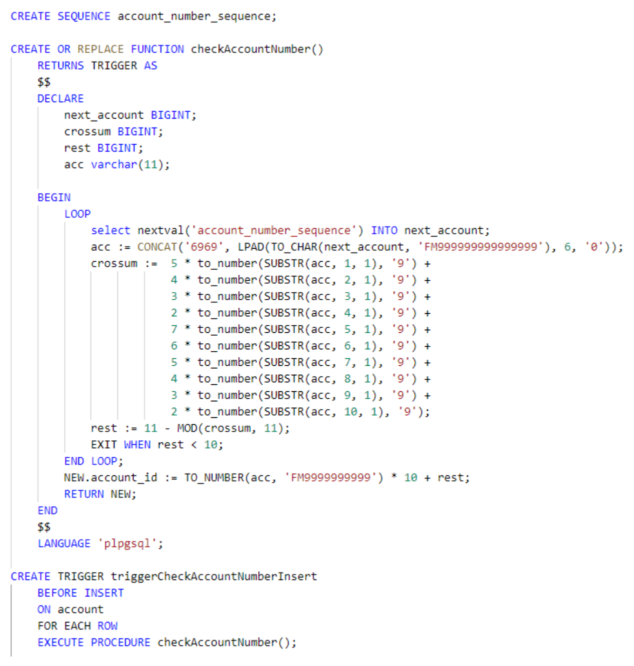
table for each account number. The accountperformstransaction table also has a row added, which makes the relationship between the tables account and transactions. This will allow the user to access a list of their transactions, through this link betweeen account and transactions.

Finally, we check if the new balance of senders account is positive. If it is not, we rollback the entire transaction, as we will not allow an account to go into overdraft. If the new balance is positive, we commit the transaction.

Triggers

Triggers are one of the fundamental features of PostgreSQL in order for automization to work on a database. Triggers are features that tell the database to execute functions whenever certain operations are performed. The main importance of using triggers is to make sure certain conditions are met before performing operations. An example we use it for is when creating new accounts. As account numbers need to have a weighted sum divisible by 11, it is difficult to implement this as a regular constraint on the columns. Using a trigger, we ensure that anytime a new account is created, we call the function to generate a new account number based on our specification.

The implementation of triggers can greatly improve the efficiency of a system. One way in which it is beneficial to use triggers, is because they reside in the database, and not in the frontend - this allows for any user with sufficient access privileges to use them. This allows any application to rely on said triggers instead of implementing them on each application on its own. We will only be using ASP.NET, and are therefore not going to use it in such a manner, but the functionality is required and also very useful for possible future implementations and modifications. Implementation of triggers in PostgreSQL differs slightly to other systems like Oracle and MySQL. Whereas other implementations allow for writing the functionality within the trigger itself, PostgreSQL needs a separate trigger function to be called by the trigger.



```
CREATE SEQUENCE account_number_sequence;

CREATE OR REPLACE FUNCTION checkAccountNumber()
    RETURNS TRIGGER AS
    $$
    DECLARE
        next_account BIGINT;
        crossum BIGINT;
        rest BIGINT;
        acc varchar(11);

    BEGIN
        LOOP
            select nextval('account_number_sequence') INTO next_account;
            acc := CONCAT('6969', LPAD(TO_CHAR(next_account, 'FM999999999999999'), 6, '0'));
            crossum :=  5 * to_number(SUBSTR(acc, 1, 1), '9') +
                        4 * to_number(SUBSTR(acc, 2, 1), '9') +
                        3 * to_number(SUBSTR(acc, 3, 1), '9') +
                        2 * to_number(SUBSTR(acc, 4, 1), '9') +
                        7 * to_number(SUBSTR(acc, 5, 1), '9') +
                        6 * to_number(SUBSTR(acc, 6, 1), '9') +
                        5 * to_number(SUBSTR(acc, 7, 1), '9') +
                        4 * to_number(SUBSTR(acc, 8, 1), '9') +
                        3 * to_number(SUBSTR(acc, 9, 1), '9') +
                        2 * to_number(SUBSTR(acc, 10, 1), '9');
            rest := 11 - MOD(crossum, 11);
            EXIT WHEN rest < 10;
        END LOOP;
        NEW.account_id := TO_NUMBER(acc, 'FM9999999999') * 10 + rest;
        RETURN NEW;
    END
    $$
    LANGUAGE 'plpgsql';

CREATE TRIGGER triggerCheckAccountNumberInsert
    BEFORE INSERT
    ON account
    FOR EACH ROW
    EXECUTE PROCEDURE checkAccountNumber();
```

Figure 9: Check account number

Figure 9 shows an example of a trigger. If we start at the bottom, we first create a trigger on account that will run before any insert into the account table. For each row that should be inserted, we execute the checkAccount procedure.

We also create a new sequence to be used to generate account numbers.

If we look in the checkAccount function, we see that it returns a trigger, which is a requirement for triggers in PostgreSQL. In the logic, we start by opening a loop. This loop will continue until we have a valid account number. We start by reading in the next sequential number from our sequence. We then leftpad the number with 0's and add the banks registry number to the front. We then take the weighted crosssum of the number and take the mod of this. When the crosssum is ok, we end the loop and add the new account number with the correct checksum to the new entry, and return this new entry. After this, the entry is allowed to be inserted into the database.

5.4 Frontend - ASP.NET Core

We used the ASP.NET Core framework as basis for our frontend development. ASP.NET Core is a free and open-source modular web framework developed by Microsoft[2]. The framework is useful for us because it features C# and HTML5 compatability, as well as being easy to handle the connection to our server. For ease of implementation, we decided to forego using the EntityFramework functionality in ASP.NET, which is built to handle database interactivity. Instead, we are setting up the connection directly using Npgsql which is a dataprovider for connecting .NET to PostgreSQL, both setting up the connection and executing queries. Another strong benefit of using ASP.NET to build the application is the availability of models. With this, we are able to define the different entities we want to use, and can therefore conditionally render the webpage based on user input. We are using 3 models in our code, corresponding to different entities in our database: Person, Transaction, and Account. Adding a list of relatives to Person also allows us to grant the user access to view accounts belonging to their spouse or children, but will not be able to perform any actions on these. We also made a model for Actions, which allows us to perform withdraws, deposits and transfers between accounts from this interface. With this system, we are able to "login" to a specific user, view relatives and accounts associated with the user and perform transactions on any of the accounts belonging to the user.

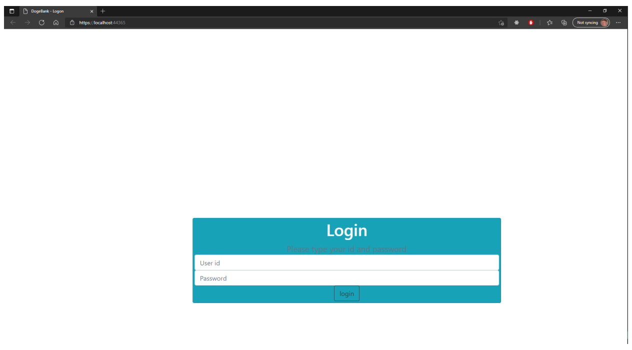


Figure 10: Login screen

Our current login screen. The intent is to make the functionality so that what is written into the login form will be the user that is presented on the following pages, for now it is hardcoded to a specific user.

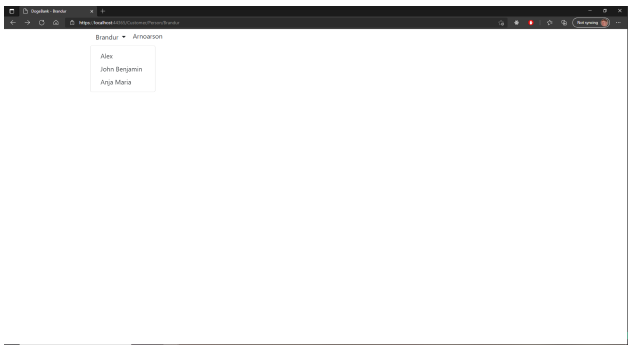
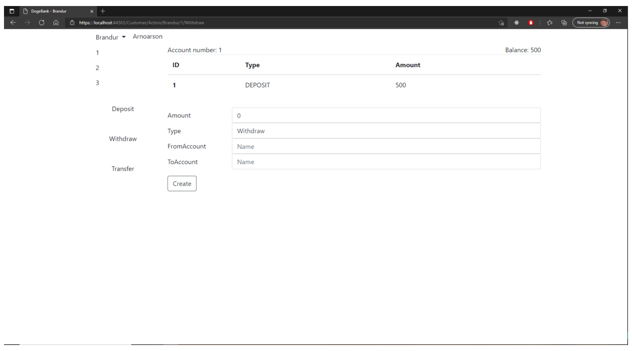


Figure 11: Viewing relatives

In figure 11 we see that logged in as Brandur, we see all relatives of this user, and can select them. Pressing one of the names on this list will bring up information about the different accounts belonging to that person.



Account number: 1 Balance: 500

ID	Type	Amount
1	DEPOSIT	500

Amount: 0
Type: Withdraw
FromAccount: Name
ToAccount: Name

Figure 12: Performing widthdraw action

In figure 12 we see the the full view when a withdraw action has been selected. To get to this page, we have reloaded the page for different segments: We start by selecting Brandur, this gives us the accounts on the left. Clicking on account 1 gives us the buttons for transfer, deposit, and withdraw, as well as the information about the account and the transactions that are shown on the right. Then pressing the Withdraw button gives us the form on the right, with the field for type of transaction already filled out.

5.5 Installation Process

Initially we installed Ubuntu 18.04 on a server, which worked as expected. But a problem occurred when trying to implement our system on the server, namely that the latest version of PostgreSQL that Ubuntu 18.04 features is v. 10. This turned out to be a problem, as *procedures*-compatability first was featured in PostgreSQL 11. This cost us a bit of time, and partly hindered us from having the system fully functioning before the presentation. But as we easily could run the system locally, we just moved onto the report and coding. This doesn't exclude the possibility of the project from being finalized, as we have already updated Ubuntu to 20.04. This update let us update from PostgreSQL v.10 to v.12. But it wasn't as easy as just updating PostgreSQL, as we didn't want to lose any data on our old database i.e. user permission etc.. So to do that, we had to make a backup of the old database and install the new version in parallel, which allowed us to move configuration files and such between the two different versions. Then we just had to change the listening port on v. 12 to the default listening port of v. 10 while removing v. 10 from the system.

But in the hopes of us getting the server up and running on Ubuntu before the presentation, the installation process, quickly summarized, is as follows: We installed Ubuntu 20.04 on a server hosted by *setur.cloud.fo*. After giving each group member sudo access, we installed PostgreSQL onto the server. To access the PostgreSQL CLI, we need to switch to our PostgreSQL user with the command:

sudo -i -u postgres

While logged in as the postgres user we are free to use the database management system by entering the psql command.

In order for each member to get view and read access in PostgreSQL, we needed to make us all superusers with the command:

ALTER USER <user> **WITH SUPERUSER;**

With help from the built-in SSH-client in VS Code, we can log into the server and install an extension called PostgreSQL to establish a connection to the database management system. We can now run Postgres commands in VS Code, even from home.

6 Future Work

In the previous subsection concerning the frontend implementation, we presented the GUI as it was at the time of writing. If compared to the mockups showcased in the design section, it is clear to see that they do not look similar to eachother. Although the rough features are present, we still aim to get the GUI polished and looking better for the actual presentation. We will not promise that it will look exactly as the initial mockup, as the design is still open for improvements along the way. Regarding functional improvements to the GUI, we will implement the ability of viewing the aggregation of the balance of the user-accounts.

With this said, there are also improvements needed for the backend. One of these improvements is the ability to get the interest rates from AccountType. At the time of writing, this functionality is hard-coded in the procedure InterestRate. Furthermore, the transaction formula is not functional at the moment, as it is not connected to the database at this point. This will all be fixed at the time of the presentation.

7 Assessment

Although the system is functioning as intended, there are several ways in which we could have executed it differently. As briefly mentioned in the design segment, we could have hosted a virtual Windows machine in which we could have installed Oracle. This would have enabled us the option of following the instructions from the book of the course more closely by using SQL instead of PostgreSQL. This could have saved us some time on i.e. the trigger implementation, as the syntax is not entirely identical in PostgreSQL and SQL. This caused some minor inconveniances, which we found not being all too troublesome to overcome in the end. While touching upon the subject of what could have been done differently, there could practically have been an infinite amount of ways of doing this project. As there are so many different ways of making the system work, there are most likely a vast amount of ways to make it more efficient, quicker, less time consuming and better functioning. But given our current skills and knowledge of the subject, we find that this has been the better solution for us. Our choice of using PostgreSQL instead of SQL, allowed us to aquire some extratextual skills that could come in handy for us in the future.

8 Conclusion

We find that the project went well overall. Although there were some minor hiccups in getting the frontend connected to the backend and some trouble getting used to the PostgreSQL syntax, we didn't find it all too difficult. But that said, we weren't completely done with the project when we handed this report in, so we can't say that it went exactly as planned. But this didn't hinder us in finalizing the system before the presentation, as we were rather close to finishing the system at that point. All in all, we found that this course was instructive and we feel that we have gained useful knowledge moving forward.

9 Bibliography

References

[1] 1. what is postgresql?, May 2021.

[2] Asp.net core, May 2021.

[3] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database system concepts.* McGraw-Hill, 7 edition, 2020.

10 Appendix ER Diagram

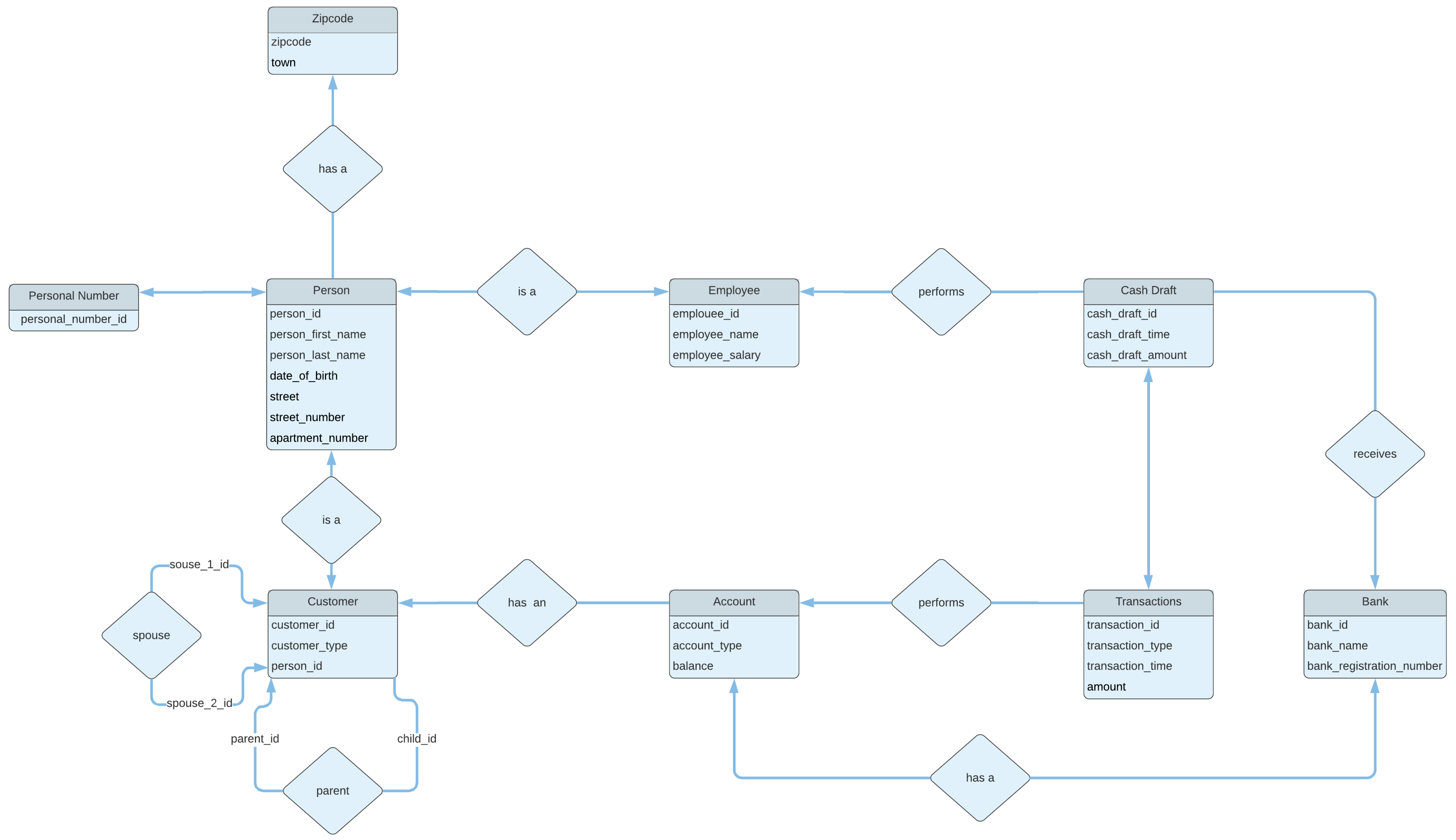


Figure 13: ER Diagram

11 Appendix Table Diagram

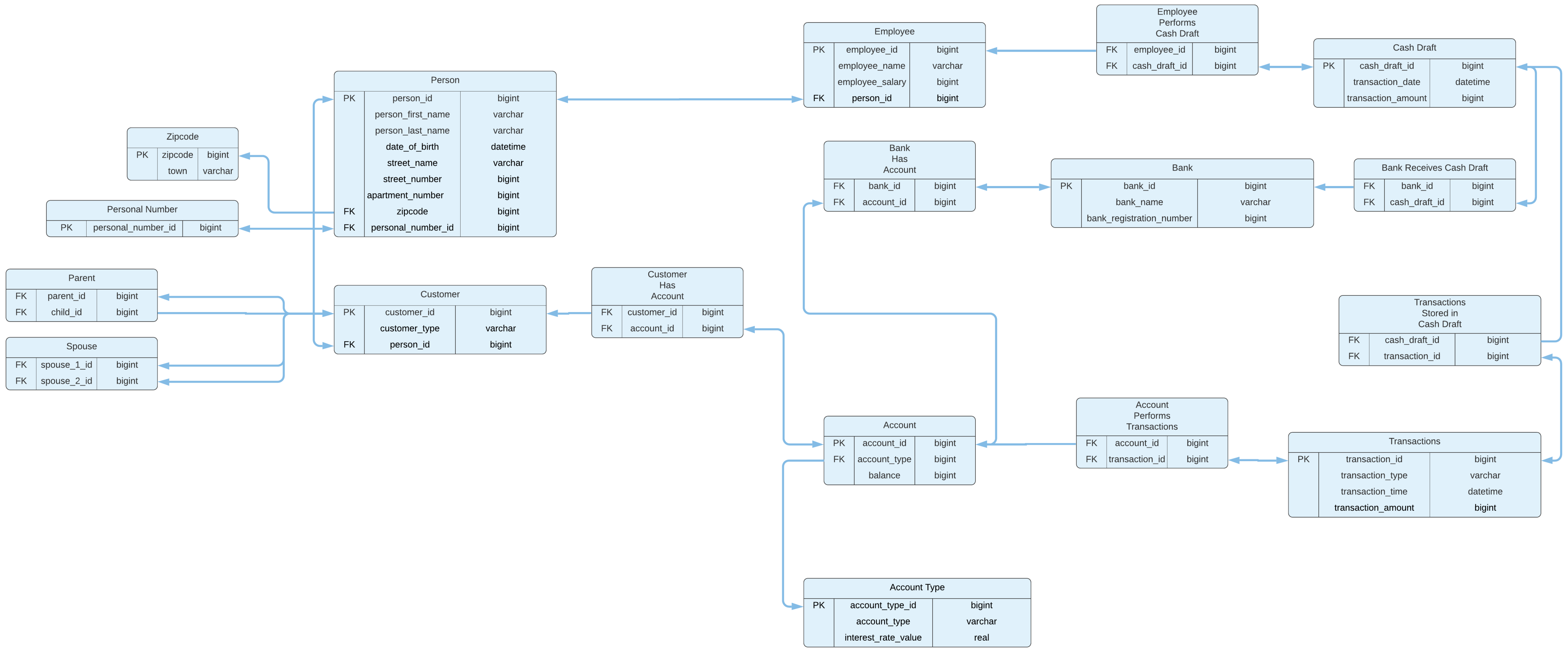


Figure 14: Table Diagram

12 Appendix DDL

```
-----------------------------------
---------SQL DATABASE--------------
-----------------------------------
-----------DOGEBANK----------------
-----------------------------------
------------GROUP 7----------------
-----------------------------------

----------Create Tables------------

--Create Zipcode table--
DROP TABLE IF EXISTS Zipcode;
CREATE TABLE Zipcode(
    zipcode BIGINT NOT NULL PRIMARY KEY,
    town varchar(255) NOT NULL
    );

--Create Personal Number table--
DROP TABLE IF EXISTS PersonalNumber;
CREATE TABLE PersonalNumber(
    personal_number_id BIGINT NOT NULL PRIMARY KEY
    );

--Create Person table--
DROP TABLE IF EXISTS Person;
CREATE TABLE Person(
    person_id BIGINT NOT NULL PRIMARY KEY,
    person_first_name varchar(255) NOT NULL,
    person_last_name varchar(255) NOT NULL,
    date_of_birth timestamp NOT NULL,
    street_name varchar(255) NOT NULL,
    street_number BIGINT NOT NULL,
    apartment_number BIGINT,
    zipcode BIGINT NOT NULL,
    personal_number_id BIGINT NOT NULL,
    CONSTRAINT fk_zipcode
        FOREIGN KEY(zipcode)
            REFERENCES zipcode(zipcode),
    CONSTRAINT fk_personal_number
        FOREIGN KEY(personal_number_id)
            REFERENCES PersonalNumber(personal_number_id)
    );

--Create Parent table--
DROP TABLE IF EXISTS Parent;
CREATE TABLE Parent(
    parent_id BIGINT NOT NULL,
    child_id BIGINT NOT NULL,

    CONSTRAINT fk_parent
        FOREIGN KEY(parent_id)
            REFERENCES Person(person_id),
    CONSTRAINT fk_child
        FOREIGN KEY(child_id)
            REFERENCES Person(person_id)
    );

--Create Spouse table--
```

```
DROP TABLE IF EXISTS Spouse;
CREATE TABLE Spouse(
    spouse_1_id BIGINT NOT NULL,
    spouse_2_id BIGINT NOT NULL,
    CONSTRAINT fk_spouse_1
        FOREIGN KEY(spouse_1_id)
            REFERENCES Person(person_id),
    CONSTRAINT fk_spouse_2
        FOREIGN KEY(spouse_2_id)
            REFERENCES Person(person_id)
);

--Create Customer table--
DROP TABLE IF EXISTS Customer;
CREATE TABLE Customer(
    customer_id BIGINT NOT NULL PRIMARY KEY,
    customer_type varchar(255) NOT NULL,
    person_id BIGINT NOT NULL,
    CONSTRAINT fk_person
        FOREIGN KEY(person_id)
            REFERENCES Person(person_id)
);

--Create Account table--
DROP TABLE IF EXISTS Account;
CREATE TABLE Account(
    account_id BIGINT NOT NULL PRIMARY KEY,
    account_type varchar(255) NOT NULL,
    balance real NOT NULL
);

--Create Customer Has Account table--
DROP TABLE IF EXISTS CustomerHasAccount;
CREATE TABLE CustomerHasAccount(
    customer_id BIGINT NOT NULL,
    account_id BIGINT NOT NULL,
    CONSTRAINT fk_customer
        FOREIGN KEY(customer_id)
            REFERENCES Person(person_id),
    CONSTRAINT fk_account
        FOREIGN KEY(account_id)
            REFERENCES Account(account_id)
);

--Create Employee table--
DROP TABLE IF EXISTS Employee;
CREATE TABLE Employee(
    employee_id BIGINT NOT NULL PRIMARY KEY,
    employee_name varchar(255) NOT NULL,
    employee_salary BIGINT NOT NULL,
    person_id BIGINT NOT NULL,
    CONSTRAINT fk_person
        FOREIGN KEY(person_id)
            REFERENCES Person(person_id)
);

--Create Cash Draft table--
DROP TABLE IF EXISTS CashDraft;
CREATE TABLE CashDraft(
    cash_draft_id BIGINT NOT NULL PRIMARY KEY,
    transaction_date timestamp NOT NULL,
    transaction_amount real NOT NULL
);

--Create Employee Performs Cash Draft table--
DROP TABLE IF EXISTS EmployeePerformsCashDraft;
CREATE TABLE EmployeePerformsCashDraft(
    employee_id BIGINT NOT NULL,
    cash_draft_id BIGINT NOT NULL,
    CONSTRAINT fk_employee
        FOREIGN KEY(employee_id)
            REFERENCES Employee(employee_id),
    CONSTRAINT fk_cashdraft
        FOREIGN KEY(cash_draft_id)
            REFERENCES CashDraft(cash_draft_id)
);

--Create Transactions table--
DROP TABLE IF EXISTS Transactions;
CREATE TABLE Transactions(
    transaction_id SERIAL NOT NULL PRIMARY KEY,
    transaction_type varchar(255) NOT NULL,
    transaction_time timestamp NOT NULL,
    transaction_amount real NOT NULL
);

--Create Transaction Stored In Cash Draft table--
DROP TABLE IF EXISTS TransactionStoredInCashDraft;
CREATE TABLE TransactionStoredInCashDraft(
    cash_draft_id BIGINT NOT NULL,
    transaction_id BIGINT NOT NULL,
    CONSTRAINT fk_cashdraft
        FOREIGN KEY(cash_draft_id)
            REFERENCES CashDraft(cash_draft_id),
    CONSTRAINT fk_transaction
        FOREIGN KEY(transaction_id)
            REFERENCES Transactions(transaction_id)
);

--Create Account Performs Transaction table--
DROP TABLE IF EXISTS AccountPerformsTransaction;
CREATE TABLE AccountPerformsTransaction(
    account_id BIGINT NOT NULL,
    transaction_id BIGINT NOT NULL,
    CONSTRAINT fk_account
        FOREIGN KEY(account_id)
            REFERENCES Account(account_id),
    CONSTRAINT fk_transaction
        FOREIGN KEY(transaction_id)
            REFERENCES Transactions(transaction_id)
);

--Create Account Type table--
DROP TABLE IF EXISTS AccountType;
CREATE TABLE AccountType(
    account_type_id BIGINT NOT NULL,
    interest_rate_name varchar(255) NOT NULL,
    interest_rate_value real NOT NULL,
    account_id BIGINT NOT NULL
```

```
--Create Bank table--
DROP TABLE IF EXISTS Bank;
CREATE TABLE Bank(
    bank_id BIGINT PRIMARY KEY,
    bank_name VARCHAR(255) UNIQUE NOT NULL,
    bank_registration_number BIGINT UNIQUE NOT NULL
);

--Create Bank Has Account table--
DROP TABLE IF EXISTS BankHasAccount;
CREATE TABLE BankHasAccount(
    bank_id BIGINT NOT NULL,
    account_id BIGINT NOT NULL,
    CONSTRAINT fk_bank
        FOREIGN KEY(bank_id)
            REFERENCES Bank(bank_id),
    CONSTRAINT fk_account
        FOREIGN KEY(account_id)
            REFERENCES account(account_id)
);

DROP TABLE IF EXISTS BankReceivesCashDraft;
CREATE TABLE BankReceivesCashDraft(
    bank_id BIGINT NOT NULL,
    cash_draft_id BIGINT NOT NULL,
    CONSTRAINT fk_bank
        FOREIGN KEY(bank_id)
            REFERENCES Bank(bank_id),
    CONSTRAINT fk_cashdraft
        FOREIGN KEY(cash_draft_id)
            REFERENCES CashDraft(cash_draft_id)
);

--Create a View for getting total balances
CREATE OR REPLACE VIEW TotalBalances AS
    SELECT account_type, SUM(balance) AS sum
    FROM account
    GROUP BY account_type;
```

13 Appendix DML

```
    -----------------------------------
-----------SQL DATABASE--------------
-----------------------------------
-----------DOGEBANK----------------
-----------------------------------
------------GROUP 7----------------
-----------------------------------
DROP IF EXISTS TRIGGER triggerCheckAccountNumberInsert ON account;
DROP IF EXISTS FUNCTION checkAccountNumber();
DROP IF EXISTS SEQUENCE account_number_sequence;

CREATE SEQUENCE account_number_sequence;

CREATE OR REPLACE FUNCTION checkAccountNumber()
    RETURNS TRIGGER AS
    $$
    DECLARE
        next_account BIGINT;
        crossum BIGINT;
        rest BIGINT;
        acc varchar(11);

    BEGIN
        LOOP
            select nextval('account_number_sequence') INTO next_account;
            acc := CONCAT('6969', LPAD(TO_CHAR(next_account, 'FM999999999999999'), 6, '0'));
            crossum :=  5 * to_number(SUBSTR(acc, 1, 1), '9') +
                        4 * to_number(SUBSTR(acc, 2, 1), '9') +
                        3 * to_number(SUBSTR(acc, 3, 1), '9') +
                        2 * to_number(SUBSTR(acc, 4, 1), '9') +
                        7 * to_number(SUBSTR(acc, 5, 1), '9') +
                        6 * to_number(SUBSTR(acc, 6, 1), '9') +
                        5 * to_number(SUBSTR(acc, 7, 1), '9') +
                        4 * to_number(SUBSTR(acc, 8, 1), '9') +
                        3 * to_number(SUBSTR(acc, 9, 1), '9') +
                        2 * to_number(SUBSTR(acc, 10, 1), '9');
            rest := 11 - MOD(crossum, 11);
            EXIT WHEN rest < 10;
        END LOOP;
        NEW.account_id := TO_NUMBER(acc, 'FM9999999999') * 10 + rest;
        RETURN NEW;
    END
    $$
    LANGUAGE 'plpgsql';

CREATE TRIGGER triggerCheckAccountNumberInsert
    BEFORE INSERT
    ON account
    FOR EACH ROW
    EXECUTE PROCEDURE checkAccountNumber();




-----------------------------------
-----Stored Procedure Transfer-----
-----------------------------------

--Drop Triggers
```

```
DROP IF EXISTS TRIGGER trigger_transactions_insert ON transactions;
DROP IF EXISTS TRIGGER trigger_account_balance_insert ON account;

--Trigger when logging transacitons
CREATE TRIGGER trigger_transactions_insert
    BEFORE INSERT
    ON transactions
    FOR EACH ROW
    BEGIN

    END;

--Trigger when updating bank account balance
CREATE TRIGGER trigger_account_balance_insert
    BEFORE UPDATE
    ON account
    FOR EACH ROW
    EXECUTE PROCEDURE

--Trigger Procedure to check if bank balance is enough
CREATE OR REPLACE FUNCTION checkbalance()
RETURNS TRIGGER
AS
$$
BEGIN
    --Check if amount is larger than bank balance
    IF NEW.amount < 0
        IF NEW.amount > OLD.balance
        --RETURN NULL/ABORT??
    END IF;
END;
$$
LANGUAGE 'plpgsql';

DROP SEQUENCE transaction_sequence;
CREATE SEQUENCE transaction_sequence;


CREATE OR REPLACE PROCEDURE transfers(account_id_1_variable BIGINT, account_id_2_variable BIGINT, amount
LANGUAGE 'plpgsql'
AS
$$
DECLARE
    balance_check BIGINT;
    new_trans_id_1 BIGINT;
    new_trans_id_2 BIGINT;
BEGIN
    --Update balance of first bank account
    UPDATE account
    SET balance = balance - amount_variable
    WHERE account_id = account_id_1_variable;

    SELECT balance
    INTO balance_check
    FROM account
    WHERE account_id = account_id_1_variable;

    --Update balance of second bank account
    UPDATE account
    SET balance = balance + amount_variable
    WHERE account_id = account_id_2_variable;


    --Add record of transaction for first bank account
    INSERT INTO transactions
    (transaction_id, transaction_type, transaction_time, transaction_amount)
    VALUES
    (DEFAULT, 'Outgoing', CURRENT_TIMESTAMP, -amount_variable) RETURNING transaction_id INTO new_trans_i


    INSERT INTO accountperformstransaction
    (account_id, transaction_id)
    VALUES
    (account_id_1_variable, new_trans_id_1);

    --Add record of transaction for second bank account
    INSERT INTO transactions
    (transaction_id, transaction_type, transaction_time, transaction_amount)
    VALUES
    (DEFAULT, 'Incoming', CURRENT_TIMESTAMP, amount_variable) RETURNING transaction_id INTO new_trans_ic


    INSERT INTO accountperformstransaction
    (account_id, transaction_id)
    VALUES
    (account_id_2_variable, new_trans_id_2);

    IF balance_check < 0
    THEN
        ROLLBACK;
    ELSE
        commit;
    END IF;
END;
$$



-----------------------------------
--------Select Functions-----------
-----------------------------------



--Get all transactions of customer--
CREATE OR REPLACE FUNCTION getAllTransactions(account_id_variable varchar(255))
    RETURNS TABLE(transaction_id_variable BIGINT, transaction_type_variable varchar(255), transaction_ti
    AS
    $$
    BEGIN
        SELECT *
        FROM Transactions
        WHERE transaction_id = (
            SELECT transaction_id
            FROM AccountPerformsTransaction
            WHERE account_id = (
                SELECT account_id
                FROM Account
                WHERE account_id = account_id_variable(
                )
            )
        );
    END;
    $$
    LANGUAGE 'plpgsql';

--Show all accounts of person--
CREATE OR REPLACE FUNCTION showAllAccounts(person_id_variable varchar(255))
    RETURNS TABLE(account_id_variable BIGINT)
    AS
    $$
    BEGIN
        SELECT account_id
        FROM Account
        WHERE account_id = (
            SELECT account_id
            FROM CustomerHasAccount
            WHERE customer_id = (
                SELECT customer_id
                FROM Customer
                WHERE person_id = person_id_variable
            )
        );
    END;
    $$
    LANGUAGE 'plpgsql';

--Show all spouses, and children of customer--
CREATE OR REPLACE FUNCTION showAllSpousesOrChildren(customer_id_variable varchar(255))
    RETURNS TABLE(child_or_spouse_id_variable BIGINT)
    AS
    $$
    BEGIN
        SELECT child_id
            FROM Parent
            WHERE parent_id = (
                SELECT customer_id
                FROM Customer
                WHERE customer_id = customer_id_variable
            )
            UNION
            SELECT spouse_2_id
            FROM Spouse
            WHERE spouse_1_id = (
                SELECT customer_id
                FROM Customer
                WHERE customer_id = customer_id_variable
            );
    END;
    $$
    LANGUAGE 'plpgsql';


--Show all accounts of child--
CREATE OR REPLACE FUNCTION showAllAccountsOfChild(child_id_variable varchar(255))
    RETURNS TABLE(account_id_variable BIGINT)
    AS
    $$
    BEGIN
        SELECT account_id
        FROM CustomerHasAccount
        WHERE customer_id = child_id_variable;
    END;
    $$
    LANGUAGE 'plpgsql';


--Show all accounts of spouse--
CREATE OR REPLACE FUNCTION showAllAccountsOfSpouse(spouse_2_id_variable varchar(255))
    RETURNS TABLE(account_id_variable BIGINT)
    AS
    $$
    BEGIN
        SELECT account_id
        FROM CustomerHasAccount
        WHERE customer_id = spouse_2_id_variable;
    END;
    $$
    LANGUAGE 'plpgsql';


----------------------------------
----------Insert Functions--------
----------------------------------

CREATE OR REPLACE PROCEDURE inertIntoBankReceivesCashDraft()

CREATE OR REPLACE PROCEDURE createAccount(customer_id_variable BIGINT, account_type_variable varchar(25
AS
$$
DECLARE
    account_id_temp BIGINT;
BEGIN
    INSERT INTO account(account_id, account_type, balance)
    VALUES(DEFAULT, account_type_variable, balance_variable)
    RETURNING account_id INTO account_id_temp;

    INSERT INTO customerhasaccount(customer_id, account_id)
    VALUES(customer_id_variable, account_id_temp);
    COMMIT;
END
$$
LANGUAGE 'plpgsql';

--Account--
CREATE OR REPLACE PROCEDURE insertIntoAccount(account_type_variable varchar(255), balance_variable real
    AS
    $$
    BEGIN
        INSERT INTO Account(
            account_id,
            account_type,
            balance)
        Values(DEFAULT, account_type_variable, balance_variable);
        COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';

--AccountPerformsTransaction--
CREATE OR REPLACE PROCEDURE insertIntoAccountPerformsTransaction(account_id_variable BIGINT, transactio
    AS
    $$
    BEGIN
    INSERT INTO AccountPerformsTransaction(
        account_id,
        transaction_id)
    Values(account_id_variable, transaction_id_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';

--AccountType--
CREATE OR REPLACE PROCEDURE insertIntoAccountType(account_type_id_variable BIGINT, interest_rate_name_va
    AS
    $$
    BEGIN
    INSERT INTO AccountType(
        account_type_id,
        interest_rate_name,
        interest_rate_value,
        account_id)
    Values(account_type_id_variable, interest_rate_name_variable, interest_rate_value_variable, account_
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--CashDraft--
CREATE OR REPLACE PROCEDURE insertIntoCashDraft(cash_draft_id_variable BIGINT, transaction_date_variable
    AS
    $$
    BEGIN
    INSERT INTO CashDraft(
        cash_draft_id,
        transaction_date,
        transaction_amount)
    Values(cash_draft_id_variable, transaction_date_variable, transaction_amount_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--Customer--
CREATE OR REPLACE PROCEDURE insertIntoCustomer(customer_id_variable BIGINT, customer_type_variable varch
    AS
    $$
    BEGIN
    INSERT INTO Customer(
        customer_id,
        customer_type,
        person_id)
    Values(customer_id_variable, customer_type_variable, person_id_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';
```

```
--CustomerHasAccount--
CREATE OR REPLACE PROCEDURE insertIntoCustomerHasAccount(customer_id_variable BIGINT, account_id_variabl
    AS
    $$
    BEGIN
    INSERT INTO CustomerHasAccount(
        customer_id,
        account_id)
    Values(customer_id_variable, account_id_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--Employee--
CREATE OR REPLACE PROCEDURE insertIntoEmployee(employee_id_variable BIGINT, employee_name_variable varch
    AS
    $$
    BEGIN
    INSERT INTO Employee(
        employee_id,
        employee_name,
        employee_salary,
        person_id)
    Values(employee_id_variable, employee_name_variable, employee_salary_variable, person_id_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--EmployeePerformsCashDraft--
CREATE OR REPLACE PROCEDURE insertIntoEmployeePerformsCashDraft(employee_id_variable BIGINT, cash_draft_
    AS
    $$
    BEGIN
    INSERT INTO EmployeePerformsCashDraft(
        employee_id,
        cash_draft_id)
    Values(employee_id_variable, cash_draft_id_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--Parent--
CREATE OR REPLACE PROCEDURE insertIntoParent(parent_id_variable BIGINT, child_id_variable BIGINT)
    AS
    $$
    BEGIN
    INSERT INTO Parent(
        parent_id,
        child_id)
    Values(parent_id_variable, child_id_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';
```

```
--Person--
CREATE OR REPLACE PROCEDURE insertIntoPerson(person_id_variable BIGINT, person_first_name_variable varch
    AS
    $$
    BEGIN
    INSERT INTO Person(
        person_id,
        person_first_name,
        person_last_name,
        date_of_birth,
        street_name,
        street_number,
        apartment_number,
        zipcode,
        person_number_id)
    VALUES (person_id_variable, person_first_name_variable, person_last_name_variable, date_of_birth_var
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--PersonalNumber--
CREATE OR REPLACE PROCEDURE insertIntoPersonalNumber(personal_number_id_variable BIGINT)
    AS
    $$
    BEGIN
    INSERT INTO PersonalNumber(
        personal_number_id)
    VALUES (personal_number_id_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--Spouse--
CREATE OR REPLACE PROCEDURE insertIntoSpouse(spouse_1_id_variable BIGINT, spouse_2_id_variable BIGINT)
    AS
    $$
    BEGIN
    INSERT INTO Spouse(
        spouse_1_id,
        spouse_2_id)
    Values(spouse_1_id_variable, spouse_2_id_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--Transactions--
CREATE OR REPLACE PROCEDURE insertIntoTransactions(transaction_id_variable BIGINT, transaction_type_vari
    AS
    $$
    BEGIN
    INSERT INTO Transactions(
        transaction_id,
        transaction_type,
        transaction_time,
        transaction_amount)
    Values(transaction_id_variable, transaction_type_variable, transaction_time_variable, transaction_am
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--TransactionStoredInCashDraft--
CREATE OR REPLACE PROCEDURE insertIntoTransactionStoredInCashDraft(cash_draft_id_variable BIGINT, transa
    AS
    $$
    BEGIN
    INSERT INTO TransactionStoredInCashDraft(
        cash_draft_id,
        transaction_id)
    Values(cash_draft_id_variable, transaction_id_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';


--Zipcode--
CREATE OR REPLACE PROCEDURE insertIntoZipcode(zipcode_variable BIGINT, town_variable varchar(255))
    AS
    $$
    BEGIN
    INSERT INTO Zipcode(
        zipcode,
        town)
    Values(zipcode_variable, town_variable);
    COMMIT;
    END;
    $$
    LANGUAGE 'plpgsql';
```

14 Appendix GIT link

Git will be updated as we proceed with changes

https://github.com/JogvanPatursson/Bankaskipan