End-Sem Project Report

Banking system database

Sughandhan, 111901049

Varun Ganta, 111901051

Narvik Nandan, 111901035

CS3120 - Database Management Systems Lab

Introduction

A bank is a financial institution which acts as an intermediate in financial transactions, a bank also provides financial support to its customers and other finance related services. The support/services provided include sanctioning loans, accepting deposits(and withdrawals), currency exchange, wealth management, etc.

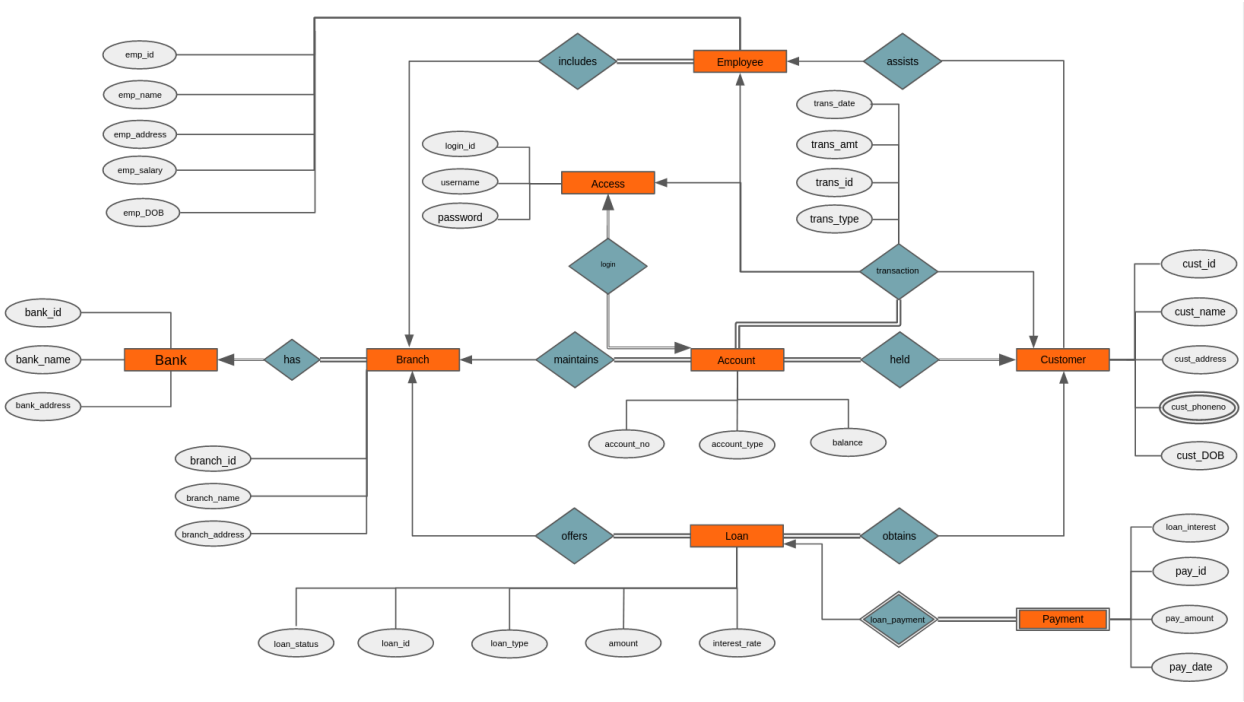
A database is an organized, well structured collection of data. This data is modeled into rows and columns in a series of tables, allowing for efficient querying. Banks are a very important part of the economy, this is because they provide essential services for both consumers and businesses and so it is of utmost importance to have a well defined and functional database management system to keep track of all financial transactions/services taking place.

The database management system we've constructed so far keeps tabs on the customers, employees, accounts registered, loans offered, the bank itself and its branches. Relations are used to account for basic banking tasks between these entities. These basic tasks would include offering loans, account transactions, repaying of loans, etc..

Entity Relationship Model

To conceptualize the design for the database and define the data elements and relationships for the banking system, we made an entity relationship model.

ER Diagram



Entities

There are 8 entities in our model to capture a simple banking system. They are :

I. **Bank**

This entity set represents the bank which encompasses branches under the bank, its customers, employees, etc. Attributes of this entity are bank ID, bank name and address.

II. **Branch**

This entity set tries to depict the branches of a bank. Each member of this entity set represents a specific branch of the corresponding bank and has branch ID, branch name and address as its attributes.

III. **Customer**

Each member of this entity set gives information about the customer of the bank. Attributes of this entity include customer ID, customer name, address, phone numbers and email ID.

IV. **Employee**

Since banks have employees working in each branch, this entity set tries to portray it. Attributes of this entity are employee ID, employee name, address and salary.

V. **Account**

The accounts maintained by the branches are represented by this entity. Each member of this entity corresponds to a bank account with attributes account number, account type (either “savings” or “checking”) and balance amount.

VI. **Loan**

The loan entity set has various loans offered by the branch to the customer with attributes such as loan ID, loan type, amount offered and the interest rate. Types of loans offered in this model are personal, business, home, student or automobile loans.

VII. **Payment**

This is a **weak entity** to represent repayment of loans. Weak entity as it only exists when the loan entity is present. Each member of this set has attributes payment ID, payment amount and payment date.

VIII. **Access**

The access entity is used whenever a person is accessing his / her account or making a transaction. Each member of this set has attributes login_id, username and password. Note that the access entity stores the attributes corresponding to both employees and customers.

Relations

Our ER model has 10 relationships between different entities that are necessary in a banking system. They are as follows :

I. **Has**

This is a **relationship between bank and branch** entity sets. Since a bank can have multiple branches, and a branch can only have utmost one bank associated with it,

this relation is a *one-to-many* relationship. Assuming that all banks have branches, both bank and branch entities are in *total participation* in this relationship.

II. Includes

This is a **relationship between branch and employee** entities. This captures the fact that every branch includes employees. Since a branch can have multiple employees and one employee can only work in a single branch, this is a *one-to-many* relation with *total participation* of the employee entity.

III. Maintains

To represent the fact that each branch maintains certain accounts, this **relationship between branch and account** is present. Once again, a branch maintains multiple accounts, whereas an account can only belong to one branch. Hence, this is a *one-to-many* relationship with *total participation* of the account entity set.

IV. Offers

This relationship **connects the branch and loan entities** and is used to represent the loan option given by branches. Branches offer multiple loans such as home loan, gold loan, etc., each of which have different characteristics. Since a particular loan corresponds to a particular branch of a bank, it is a *one-to-many* relation, with *total participation* of the loan entity.

V. Assists

The employees in the bank can also be personal bankers to assist customers in getting loans, lower interest rates, etc. This is captured by the “assists” relation which **links the employee and customer entity**. Since an employee can assist multiple customers, the relationship is *one-to-many*.

VI. Held

This is a **relationship between account and customer** entity sets and is a *many-to-many* relationship as multiple customers can have a single account (group account) and a customer can have multiple accounts. This is different from the transaction entity in the sense that the transaction entity is used to keep track of all transactions done on the account. Both entity sets are in *total participation.*

VII. Transaction

This relation is used to keep track of the transactions done by the **customer on his/her account**. The relation also has relational attributes such as transaction ID, transaction amount, date and type of transaction (withdrawing money or depositing money). Since an account must be present for every transaction, the account entity

is in *total participation*. Note that multiple customers can perform transactions from a single account (when the account is a group account) and multiple transactions can be done by a single customer. So, this is a *many-to-many* relationship.

VIII. Obtains

This is a **relationship between loan entity and customer entity**. This is a *many-to-many* relationship as multiple customers can apply for a single loan (as a group) and a customer can apply for multiple loans. Again, the loan entity is in *total participation* in this relationship.

IX. Login

This is a **relation between account and access entities**. This is a *one-to-one* relationship as an account can be associated with only one access and only one access is allowed per account. Again both the entities are in *total participation*.

X. Loan Payment

This is an **identifying relationship for the weak entity payment, joining loan and payment**. Since it is an identifying relation, the weak entity is in *total participation* and it is a *one-to-many* relationship as a loan can be repaid in multiple payments.

Relational Model

The representation of the Entity-Relationship diagram as a Relational Model is as follows :

(For simplicity, only the schema of the relations (tables) are shown.

Tables in our relational database

Primary keys are represented by 'underline' and foreign keys are represented by *'italics'.*

- **Bank** (bank_id, bank_name, bank_address)
- **Branch** (branch_id, branch_name, branch_address, *bank_id*)

  bank_id is a foreign key referencing bank table
- **Access** (login_id, username, password)
- **Employee** (emp_id,emp_name, emp_address, emp_salary, emp_DOB, *branch_id, login_id*)

- **Account** (account_no, account_type, balance, *branch_id*)
- **Loan** (loan_id, loan_type, loan_status, amount, interest_rate, *branch_id, emp_id*)
- **Customer** (cust_id, cust_name, cust_address, cust_DOB, *emp_id, login_id*)
- **Customer_Phoneno** (*cust_id*, cust_phoneno)
- **Customer_Account** (*cust_id, account_no*)
- **Customer_Loan** (*cust_id, loan_id*)
- **Branch_Loan** (*branch_id, loan_id*)
- **Payment** (pay_id, pay_amount, pay_date, loan_interest, *loan_id*)
- **Transactions** (trans_id, trans_type, trans_amt, trans_date, *account_no*, r_account_no, *cust_id, emp_id, login_id*)

  Incorporating foreign keys in many-to-many relation b/w account, customer, employee and access tables
- **Access_Account** (*login_id, account_no*)

  Incorporating foreign keys in many-to-many relation b/w access and account tables

Constraints

Constraints are a set of rules that ensure that when an authorized user makes changes to the database they do not disrupt the consistency of the data. Basically, constraints are features that specify the possible values an attribute can hold. They help users enter valid values to a table/database, according to each data type. In the case of an invalid insertion, an error message is displayed.

Constraints for tables

Bank (bank_id, bank_name, bank_address)

```
CREATE TABLE bank(
      bank_id SERIAL,
      bank_name VARCHAR(100) NOT NULL,
      bank_address TEXT NOT NULL,
      CONSTRAINT bank_pkey PRIMARY KEY (bank_id)
);
```

Bank ID is unique to all the different banks. Hence, the primary key constraint. Both bank name and bank address have NOT NULL constraints.

Branch (branch_id, branch_name, branch_address, *bank_id*)

```
CREATE TABLE branch(
        branch_id SERIAL,
        branch_name VARCHAR(100) NOT NULL,
        branch_address TEXT NOT NULL,
        bank_id INT NOT NULL,
        CONSTRAINT branch_pkey PRIMARY KEY (branch_id),
        CONSTRAINT bank_fk FOREIGN KEY (bank_id) REFERENCES bank
        (bank_id) ON UPDATE CASCADE ON DELETE CASCADE
);
```

Branch ID is again unique to all branches. Hence the primary key constraint. All other attributes have NOT NULL constraints. Bank_id is a foreign key constraint which references the bank table. On updating or deleting a branch, the effect should cascade.

Access (login_id, username, password)

```
CREATE TABLE access(
        login_id SERIAL PRIMARY KEY,
        username VARCHAR(100) UNIQUE NOT NULL,
        password VARCHAR(50) NOT NULL
);
```

Login_id is a primary key and username and password are not null attributes.

Employee (emp_id, emp_name, emp_address, emp_salary, emp_DOB, *branch_id*, *login_id*)

```
CREATE TABLE employee(
        emp_id SERIAL,
        emp_name VARCHAR(100) NOT NULL,
        emp_address TEXT NOT NULL,
        emp_salary INT NOT NULL check(emp_salary >= 10000 AND
        emp_salary <= 1000000),
        emp_DOB DATE NOT NULL check(date_part('year', AGE(emp_DOB))
        >= 18),
        branch_id INT NOT NULL,
        login_id INT NOT NULL,
```

```
        CONSTRAINT employee_pkey PRIMARY KEY (emp_id),

        CONSTRAINT branch_fk FOREIGN KEY (branch_id) REFERENCES
        branch (branch_id) ON UPDATE CASCADE ON DELETE CASCADE,

        CONSTRAINT access_fk FOREIGN KEY (login_id) REFERENCES
        access (login_id) ON UPDATE CASCADE ON DELETE RESTRICT

);
```

Emp_id is the primary key with most of the other attributes as NOT NULL. Emp_salary and emp_DOB have a check on them and branch_id and login_id are both foreign key constraints referencing branch and access tables.

Account (account_no, account_type, balance, *branch_id*)

```
CREATE TABLE account(
      account_no SERIAL PRIMARY KEY,
      account_type VARCHAR(25) NOT NULL check(account_type =
      'savings' OR account_type = 'checkings' OR account_type =
      'loan'),
      balance NUMERIC(12, 2) NOT NULL,
      branch_id INT NOT NULL,
      CONSTRAINT branch_fk FOREIGN KEY (branch_id) REFERENCES
      branch (branch_id) ON UPDATE CASCADE ON DELETE CASCADE
);
```

Account no is a primary key. Account type can be savings / checking / loan and balance is not null. It can be negative in case of loan accounts. Branch id is a foreign key constraint referencing the branch table.

Loan (loan_id, loan_type, loan_status, amount, interest_rate, *branch_id, emp_id*)

```
CREATE TABLE loan(

      loan_id SERIAL PRIMARY KEY,

      loan_type VARCHAR(25) NOT NULL check(loan_type = 'personal'
      OR loan_type = 'business' OR loan_type = 'home' OR loan_type
      = 'student' OR loan_type = 'automobile'),

      loan_status INT, -- If active 1, else 0

      amount NUMERIC(12, 2) NOT NULL check(amount >= 1000.0 AND
      amount <=10000000.0),

      interest_rate NUMERIC(4, 2) NOT NULL check(interest_rate >=
      3.0 AND interest_rate <= 12.5),
```

```
        branch_id INT NOT NULL,
        emp_id INT NOT NULL,
        CONSTRAINT branch_fk FOREIGN KEY (branch_id) REFERENCES
        branch (branch_id) ON UPDATE CASCADE ON DELETE CASCADE,
        CONSTRAINT emp_fk FOREIGN KEY (emp_id) REFERENCES employee
        (emp_id) ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Loan ID is a primary key. Loan type can be personal / business / home / student / automobile and loan status can be 0 or 1 for inactive and active. Amount and interest rate attributes have their respective checks. Branch ID and emp_id are foreign keys referencing branch and employee tables.

Customer (cust_id, cust_name, cust_address, cust_DOB, *emp_id, login_id*)

```
CREATE TABLE customer(
      cust_id SERIAL unique,
      cust_name VARCHAR(100) NOT NULL,
      cust_address TEXT NOT NULL,
      cust_DOB DATE NOT NULL check(date_part('year',
      AGE(cust_DOB)) >= 18), -- change this to DOB and take
      difference
      emp_id INT NOT NULL,
      login_id INT NOT NULL,
      CONSTRAINT customer_pkey PRIMARY KEY (cust_id),
      CONSTRAINT emp_fk FOREIGN KEY (emp_id) REFERENCES employee
      (emp_id) ON UPDATE CASCADE ON DELETE RESTRICT,
      CONSTRAINT access_fk FOREIGN KEY (login_id) REFERENCES
      access (login_id) ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Customer ID is the primary key and NOT NULL constraint is present on all other attributes. Check on customer age as customers should have a minimum age of 18. Emp_id and login_id are foreign key constraints referencing employee and access tables respectively.

Customer_Phoneno (*cust_id*, *cust_phoneno*) **[cust_ID is FK and PK]:**

```
CREATE TABLE customer_phoneno(
      cust_id INT,
      cust_phoneno VARCHAR(10) UNIQUE check (cust_phoneno ~*
      '\d\d\d\d\d\d\d\d\d\d'),
      PRIMARY KEY (cust_id, cust_phoneno),
```

```
        CONSTRAINT cont_fk FOREIGN KEY (cust_id) REFERENCES customer
        (cust_id) ON UPDATE CASCADE ON DELETE CASCADE
);
```

Here, customer_id is a primary key. There is a check on customer_phoneno to check for a valid phone number. Note that cust_id is also the foreign key in this table.

Customer_Account (*cust_id*, *account_no*) [HELD RELATION]

```
CREATE TABLE customer_account(
        cust_id INT,
        account_no INT,
        PRIMARY KEY (cust_id, account_no),
        CONSTRAINT cust_fk FOREIGN KEY (cust_id) REFERENCES customer
        (cust_id) ON UPDATE CASCADE ON DELETE CASCADE,
        CONSTRAINT acc_fc FOREIGN KEY (account_no) REFERENCES
        account (account_no) ON UPDATE CASCADE ON DELETE CASCADE
);
```

In this table, both cust_id and account_no are primary and foriegn keys.

Customer_Loan (*cust_id*, *loan_id*) [OBTAINS RELATION]

```
CREATE TABLE customer_loan(
        cust_id INT,
        loan_id INT,
        PRIMARY KEY (cust_id, loan_id),
        CONSTRAINT cust_fk FOREIGN KEY (cust_id) REFERENCES customer
        (cust_id) ON UPDATE CASCADE ON DELETE CASCADE,
        CONSTRAINT loan_fk FOREIGN KEY (loan_id) REFERENCES loan
        (loan_id) ON UPDATE CASCADE ON DELETE RESTRICT
);
```

In this table, both cust_id and loan_id are primary and foriegn keys.

Branch_Loan (*branch_id*, *loan_id*) [OFFERS RELATION]

```
CREATE TABLE branch_loan(
        branch_id INT,
        loan_id INT,
        PRIMARY KEY (branch_id, loan_id),
```

```
        CONSTRAINT branch_fk FOREIGN KEY (branch_id) REFERENCES
        branch (branch_id) ON UPDATE CASCADE ON DELETE CASCADE,

        CONSTRAINT loan_fk FOREIGN KEY (loan_id) REFERENCES loan
        (loan_id) ON UPDATE CASCADE ON DELETE RESTRICT

);
```

In this table, both branch_id and loan_id are primary and foriegn keys.

Payment (pay_id, pay_amount, pay_date, loan_interest, *loan_id*) **[loan_id is PK and FK]:**

```
CREATE TABLE payment(
        pay_id SERIAL PRIMARY KEY,
        pay_amount NUMERIC(12, 2) NOT NULL check(pay_amount >=
        100.0),
        pay_date TIMESTAMP WITHOUT TIME ZONE NOT NULL,
        loan_interest NUMERIC(4, 2) NOT NULL,
        loan_id INT NOT NULL,
        CONSTRAINT loan_fk FOREIGN KEY (loan_id) REFERENCES loan
        (loan_id) ON UPDATE CASCADE ON DELETE RESTRICT
);
```

Payment ID is the primary key and payment amount has a check for the minimum amount. Loan interest and pay_date have NOT NULL constraints on them and loan_id is the foreign key constraint.

Transactions (trans_id, trans_type, trans_amt, trans_date, *account_no, r_account_no, cust_id, emp_id, login_id*) [TRANSACTION RELATION]

```
CREATE TABLE transactions(

        trans_id SERIAL PRIMARY KEY,

        trans_type VARCHAR(50) NOT NULL check(trans_type = 'deposit'
        OR trans_type = 'withdraw' OR trans_type = 'transfer' OR
        trans_type = 'loan payment'),

        trans_amt NUMERIC(12, 2) NOT NULL check(trans_amt >= 100.0),

        trans_date TIMESTAMP WITHOUT TIME ZONE NOT NULL,

        account_no INT NOT NULL,

        r_account_no INT DEFAULT NULL, -- for receiver account when
        the transaction is of the type "transfer"
```

```
        cust_id INT NOT NULL,

        emp_id INT NOT NULL,

        login_id INT NOT NULL,

        CONSTRAINT acc_fk FOREIGN KEY (account_no) REFERENCES
        account (account_no) ON UPDATE CASCADE ON DELETE CASCADE,

        CONSTRAINT cust_fk FOREIGN KEY (cust_id) REFERENCES customer
        (cust_id) ON UPDATE CASCADE ON DELETE CASCADE,

        CONSTRAINT emp_fk FOREIGN KEY (emp_id) REFERENCES employee
        (emp_id) ON UPDATE CASCADE ON DELETE RESTRICT,

        CONSTRAINT login_fk FOREIGN KEY (login_id) REFERENCES access
        (login_id) ON UPDATE CASCADE ON DELETE RESTRICT

    );
```

Trans_id is the primary key and trans_type can be deposit / withdraw / transfer / loan payment. Trans_amt has the necessary checks and other attributes have NOT NULL constraints. Account_no, r_account_no, cust_id, emp_id and login_id are all foreign key constraints.

Access_account (*login_id*, *account_no*) [LOGIN RELATION]

```
    CREATE TABLE access_account(

        login_id INT NOT NULL,

        account_no INT NOT NULL,

        PRIMARY KEY (login_id, account_no),

        CONSTRAINT login_fk FOREIGN KEY (login_id) REFERENCES access
        (login_id) ON UPDATE CASCADE ON DELETE RESTRICT,

        CONSTRAINT acc_fk FOREIGN KEY (account_no) REFERENCES
        account (account_no) ON UPDATE CASCADE ON DELETE CASCADE

    );
```

In this table, both login_id and account_no are primary and foriegn keys.

Functions and Triggers

Functions and Procedures

We have many functions which range from helping the customer access their accounts to changing their credentials when required. We also have many functions and procedures that can be performed by employees and the admins.

The list of functions in our database:

- **create_account**

```
CREATE OR REPLACE PROCEDURE create_account(
        c_name VARCHAR(100),
        c_add TEXT,
        c_DOB DATE,
        c_phone VARCHAR(10)[],
        a_type VARCHAR(25),
        e_id INT,
        b_id INT
)
```

This procedure is used to create an account for the customer after they provide their details like name, address, DOB, phone numbers(max 2), account type (savings, checkings or loan), employee(responsible for assisting the customer through the transactions) and branch id(account is associated with loans)

On running the procedure, we use *md5* to generate username and passwords for the customer. We next have an if condition which checks if the number of phone numbers is 1 or 2. If it fails then we raise notice stating what the error is, else we proceed to inserting

- their credentials into the access table,
- their details into the customer table
- their customer id and phone numbers into the cust_phoneno table
- their account type, balance and branch_id into account table
- the login_id(retrieved from access table) and account_no into relation access_account
- the customer_id(retrieved from access table) and account_no into relation customer_account

Once all of the insertions have been executed without any problem, we then simultaneously create a user with the same name as our customer and grant the customer role to him. Finally we print a one time access account number, customer ID, username and password for the customer to note down and remember.

- **create_loan_account**

```
CREATE OR REPLACE PROCEDURE create_loan_account(
        c_name VARCHAR(100),
        c_add TEXT,
        c_DOB DATE,
        c_phone VARCHAR(10)[],
        a_type VARCHAR(25),
        e_id INT,
        b_id INT,
        l_amt INT,
        l_type VARCHAR(25),
        l_int NUMERIC(4,2)
)
```

l_amt is loan amount ; *l_type* is loan type and *l_int* is loan interest rate.

Here we will do exactly what all we did in the *create_account* procedure along with inserting into the loan (which keeps track of everything essential to identify the loan along with its other terms), customer_loan (the relation connecting customer table and the loan table), and branch_loan table (the relation connecting branch table and the loan table). The account table will have a negative balance symbolizing the amount of money they owe to the bank (along with the interest).

On failing to meet the constraints of the relations, we will raise the required errors. Finally we return the important credentials as a notice for the customer to remember and use.

- **add_employee**

```
CREATE OR REPLACE PROCEDURE add_employee(
        emp_name VARCHAR(100),
        emp_add TEXT,
        emp_salary INT,
```

```
        emp_DOB DATE,
        b_id INT
    )
```

This is a procedure to add employees into the bank database.. We will generate username and passwords for them using md5 and then insert the required details into the access and employee tables. Simultaneously, we create a user with the name of the employee and assign the employee role to him / her. Finally, the essential credentials are printed as a notice for the employees to use.

- **view_balance**

```
CREATE OR REPLACE FUNCTION view_balance(
        id INT,
        uname VARCHAR(50),
        pword VARCHAR(50)
    )
```

This function takes in the user credentials and verifies them. Once they are verified, we return the balance of the account with the account number = id if it exists. If it does not exist then we raise a notice telling the user to check their account number.

- **withdraw_amount**

```
CREATE OR REPLACE PROCEDURE withdraw_amount(
        id INT,
        amt NUMERIC(12, 2),
        uname VARCHAR(50),
        pword VARCHAR(50)
    )
```

This procedure is only applicable to accounts of type ‘savings’ and ‘checkings’. After checking the user credentials, If the account id (id) does not exist, or the amt(amount) constraints are not met, we raise what precisely was the issue. Since our bank only allows these types of accounts to have a minimum balance of 500 and maximum of 1,00,00,000 rupees we only allow withdrawals which meet this constraint. Then the balance is updated and the transaction is logged in the transactions table as “withdraw”. Notices are raised accordingly.

- **deposit_amount**

```
CREATE OR REPLACE PROCEDURE deposit_amount(
```

```
        id INT,
        amt NUMERIC(12, 2),
        uname VARCHAR(50),
        pword VARCHAR(50)
    )
```

This procedure is only applicable to accounts of type “savings” and “checkings”. If the account id (id) does not exist or the amt(amount) constraints are not met, we raise what precisely was the issue. As the transaction constraint is greater than equal to 500 and less than 1,00,00,000. Then the balance is updated and the transaction is logged in the transactions table as “deposit”. Notices are raised accordingly.

- **pay_loan**

```
CREATE OR REPLACE PROCEDURE pay_loan(
        acc_no INT,
        lo_id INT,
        amt NUMERIC(12, 2)
    )
```

This procedure is only applicable for “loan” accounts. As long as the loan is active a payment between 1,000 and 1,00,00,000 can be made. When the account balance becomes 0 we make the loan inactive implying the loan has been paid off. However we do not allow payments which result in the account balance becoming >0. If any constraints are not met, we will precisely raise what the issue is. If the constraints are met then we update the balance(remaining amount to be paid) and the transaction is logged as a “loan payment”.

- **update_creds**

```
CREATE OR REPLACE PROCEDURE update_creds(
        id INT,
        prev_uname VARCHAR(100),
        prev_pass VARCHAR(50),
        new_uname VARCHAR(100),
        new_pass VARCHAR(50)
    )
```

This is a procedure to update login and password for a customer with customer id = id with the new login credentials. If the customer id doesn’t exist we raise the error. Note

that only customers can update their credentials. Employees are stuck with their initial credentials.

- **show_transaction_log**

```
CREATE OR REPLACE FUNCTION show_transaction_log(
      id INT,
      startDate DATE,
      endDate DATE
)
```

This is a function to show the transaction logs for a given account. The function checks the user credentials and returns a table containing the necessary information of transactions between a start date and an end date.

- **transfer_amount**

```
CREATE OR REPLACE PROCEDURE transfer_amount(
      sender_id INT,
      receiver_id INT,
      amt NUMERIC(12, 2),
      uname VARCHAR(50),
      pword VARCHAR(50)
)
```

This is a procedure to transfer an amount from one account to another account. If constraints like both the sender's and receiver's account exist or the amount for transfer meets the minimum balance requirement for accounts and also the max limit for transaction, etc. aren't met, then we will precisely display what the issue was. On meeting the constraints, the balance of both the accounts are updated and the transaction is logged as a "transfer".

- **employee_assists**

```
CREATE OR REPLACE FUNCTION employee_assists(id INT)
```

This table takes in the employee id and returns a table which has the list of customers(id, name, account number) who are assisted by the current employee.

- **update_customer**

```
CREATE OR REPLACE PROCEDURE update_customer(
      id INT,
```

```
        address TEXT = NULL,
        phone_no VARCHAR(10)[] = NULL
    )
```

This is a procedure to update customer information like address and phone numbers corresponding to the given customer id (id).

- **update_employee**

```
CREATE OR REPLACE PROCEDURE update_employee(
        id INT,
        address TEXT = NULL,
        salary INT = NULL
    )
```

This is a procedure to update employee information like their address and phone numbers corresponding to the given employee id(id).

- **show_payment_log**

```
CREATE OR REPLACE FUNCTION show_payment_log(
        uname VARCHAR(50),
        pword VARCHAR(50),
        startDate DATE,
        endDate DATE
    )
```

This is a function / procedure to display the payments made by the user with the given username and password.

Note that some of the functions and procedures end with the line **'SECURITY DEFINER'.** This statement essentially helps any person who runs the function or procedure have the function owner privileges (which is basically the superuser privileges). This might seem like a security issue, but after careful evaluation and thought, we have used this statement on certain functions and procedures that can be used by the customers or employees. This is done due the fact that the users do not have the required privileges on all tables used in the functions or procedures. A slight improvement on these functions is to use triggers to perform some of the important operations like updating the bank balance, depositing and withdrawing money into accounts etc.

Triggers

```
CREATE or REPLACE FUNCTION check_interest()
RETURNS trigger
as $$
begin
    DECLARE interest_percent NUMERIC(4,2);
    SELECT interest_rate into interest_percent FROM loan WHERE
loan_id = new.loan_id;
    if(interest_percent > new.loan_int or interest_percent <
new.loan_int) THEN
        RAISE NOTICE 'Incorrect interest rate!';
    END IF;
    return new;
end;
$$ LANGUAGE plpgsql;

CREATE trigger loan_update
before INSERT
on payment
for each row
execute PROCEDURE check_interest();
```

This trigger will execute when a customer with a loan account tries to make their loan payment. If the interest rate doesn't match or there seems to be an error in meeting the constraints for loan payment, the trigger will throw an error.

Roles and Authentication

A role is a collection of privileges that one or more users/roles can be granted; they make the hierarchy in a system clear. Roles allow for privilege management for multiple users, which is a much better alternative to managing each user's privileges separately. So, it is important to classify the users into appropriate roles/categories and grant privileges accordingly. The roles in our database are as follows:

I. Branch Administrator

The branch administrator is the mini bank administrator who essentially has all the privileges to access and modify(superuser) all the relations local to a branch.

```
GRANT ALL ON branch, employee, customer, account,
access_account, customer_account, loan, customer_loan,
branch_loan TO branch_admin;

GRANT EXECUTE ON FUNCTION show_transaction_log, view_balance,
employee_assists TO branch_admin;

GRANT EXECUTE ON PROCEDURE update_customer, update_employee,
deposit_amount, withdraw_amount, transfer_amount, update_creds,
pay_loan, create_loan_account TO branch_admin;
```

II. Employee

Bank employees are responsible for the everyday operations that take place in the institution. They serve as the bridge between the bank and the customers and enhance their experience. The employees' role is to attend to the needs of the customers like assisting them in setting up accounts, sanctioning loans, recording transactions, etc. An employee is a super user who has select, update, insert and delete privileges on all tables and relations apart from the branch. However for the Employee and Branch table, they have select privileges.

```
GRANT ALL ON customer, customer_account, customer_phoneno,
account, transactions, access_account, loan, customer_loan,
branch_loan TO employee;

GRANT EXECUTE ON FUNCTION show_transaction_log, view_balance,
employee_assists TO employee;

GRANT EXECUTE ON PROCEDURE update_customer, deposit_amount,
withdraw_amount, transfer_amount, update_creds, pay_loan,
create_loan_account TO employee;
```

III. Customer

The bank customers expect a safe and reliable place to keep their money secure. The banks also serve as a promising means to borrow money and help the customers to meet their financial/business needs and potentially enjoy a good return. In our bank the customers can deposit/withdraw money from their accounts and transfer to another account as well. The customers can request for loans through the employees which gets sanctioned from the customer's main branch. The customer can also request for changing their credentials or view their account

details. Every customer by default needs to have an account associated with the bank. Our bank allows one customer to have multiple accounts.

The customers are not granted any sort of access directly to any of the tables. However for the following functions and procedures: **View_balance, withdraw_amount, deposit_amount, transfer_amount , show_transaction_log, show_payment_log, update credentials**, The customer is granted access to the respective tables needed for achieving the desired results due to the presence of **SECURITY DEFINER;**.

Views

Although we don't have explicit views created in the database, we do have few functions that can be run by the user to create a view containing only their information and select privileges.

- **customer_view**

```
CREATE OR REPLACE FUNCTION create_view_cust_details(

uname VARCHAR(50),

pword VARCHAR(50))

RETURNS void

AS $create_view_cust_details$

DECLARE

    log_id INT;

    c_name VARCHAR(100);

BEGIN

IF EXISTS (SELECT login_id FROM access WHERE username = uname
AND password = pword) THEN

     SELECT login_id INTO log_id FROM access WHERE username =
     uname AND password = pword;

     RAISE NOTICE '%', log_id;

     CREATE OR REPLACE VIEW customer_view AS (SELECT cust_name,
     cust_address, cust_DOB, cust_phoneno FROM (customer JOIN
     customer_phoneno ON customer.cust_id =
     customer_phoneno.cust_id) WHERE login_id = log_id);
```

```
        SELECT cust_name INTO c_name FROM customer WHERE login_id
        = log_id;

        EXECUTE 'GRANT SELECT ON customer_view TO "'||c_name||'"';

        RAISE NOTICE 'Temporary view called "customer_view" for
        customer has been created!';

    ELSE

        RAISE NOTICE 'Customer does not exist in database. Check
        username and password!';

    END IF;

    END;

    $create_view_cust_details$ LANGUAGE plpgsql

    SECURITY DEFINER;
```

The above function is used for creating the **customer_view**. What essentially happens here is that the customer first has to login before creating the view **customer_view**. Now they create the view and grant themself **select** privilege on **customer_view**. They were able to create this view even without any privileges to the **customer and customer_phoneno** tables because of the presence of **SECURITY DEFINER**. **SECURITY DEFINER** allows the function to be executed with the privilege of the user that owns it.

- **employee_view**

```
CREATE OR REPLACE FUNCTION create_view_emp_details(

uname VARCHAR(50),

pword VARCHAR(50))

RETURNS void

AS $create_view_emp_details$

DECLARE

    log_id INT;

    e_name VARCHAR(100);

BEGIN

IF EXISTS (SELECT login_id FROM access WHERE username = uname
AND password = pword) THEN
```

```
        SELECT login_id INTO log_id FROM access WHERE username =
        uname AND password = pword;

        CREATE OR REPLACE VIEW employee_view AS (SELECT emp_name,
        emp_address, emp_DOB, emp_salary FROM employee WHERE
        login_id = log_id);

        SELECT emp_name INTO e_name FROM employee WHERE login_id =
        log_id;

        EXECUTE 'GRANT SELECT ON employee_view TO "'||e_name||'"';

        RAISE NOTICE 'Temporary view called "employee_view" for
        employee has been created!';

    ELSE

        RAISE NOTICE 'Employee does not exist in database. Check
        username and password!';

    END IF;

    END;

    $create_view_emp_details$ LANGUAGE plpgsql

    SECURITY DEFINER;
```

The above function is used for creating the **employee_view**. What essentially happens here is that the employee first has to login before creating the view **employee_view**. Now they create the view and grant themself **select** privilege on **employee_view**. They were able to create this view even without any privileges to the **employee** table because of the presence of **SECURITY DEFINER**. **SECURITY DEFINER** allows the function to be executed with the privilege of user that own the it.

- **loanpayments_view**

```
CREATE OR REPLACE FUNCTION create_view_loan_payments(

uname VARCHAR(50),

pword VARCHAR(50))

RETURNS void

AS $$

DECLARE

    log_id INT;

    c_id INT;
```

```
    c_name VARCHAR(100);
BEGIN
    IF EXISTS (SELECT login_id FROM access WHERE username =
    uname AND password = pword) THEN
    SELECT login_id INTO log_id FROM access WHERE username =
    uname and password = pword;
    SELECT cust_id into c_id from customer where login_id =
    log_id;
    CREATE OR REPLACE VIEW loanpayments_view AS (SELECT * from
    (loan natural JOIN payment) where loan_id = (SELECT
    loan_id FROM customer_loan WHERE cust_id = c_id));
    SELECT cust_name INTO c_name FROM customer WHERE login_id
    = log_id;
    EXECUTE 'GRANT SELECT ON loanpayments_view TO
    "'||c_name||'"';
    RAISE NOTICE 'Temporary view called "loanpayment_view" for
    customer has been created!';
ELSE
    RAISE NOTICE 'Customer does not exist in database. Check
    username and password!';
END IF;
END;
$$ LANGUAGE plpgsql
SECURITY DEFINER;
```

The above function is used for creating the **loanpayments_view**. What essentially happens here is that the customer first has to login before creating the view **loanpayments_view**. Now they create the view and grant themself **select** privilege on **loanpayments_view**. They were able to create this view even without any privileges to the **loan, payment and customer_loan** tables because of the presence of **SECURITY DEFINER**. **SECURITY DEFINER** allows the function to be executed with the privilege of the user that owns it.

List of Indices

- `CREATE INDEX ON access USING HASH(username, password);`

The following is popularly used as a if statement which sort of mimics the customer or employee logging in to carry on their transactions or retrieve information. `Example Query : SELECT login_id FROM access WHERE username = uname AND password = pword;`

- `CREATE INDEX ON transactions USING BTREE(trans_date);`

We use B Tree Index on this because we have a function which finds the transaction log between a start date and an end date.

- `CREATE INDEX ON payment USING BTREE(loan_id, pay_date);`

We use Btree Index on this because some customers would like to know how many payments they have made for their loan from a specific date. `Example Query : SELECT * FROM payment WHERE loan_id = l_id and pay_date>= "2022-01-03";`