Banking System

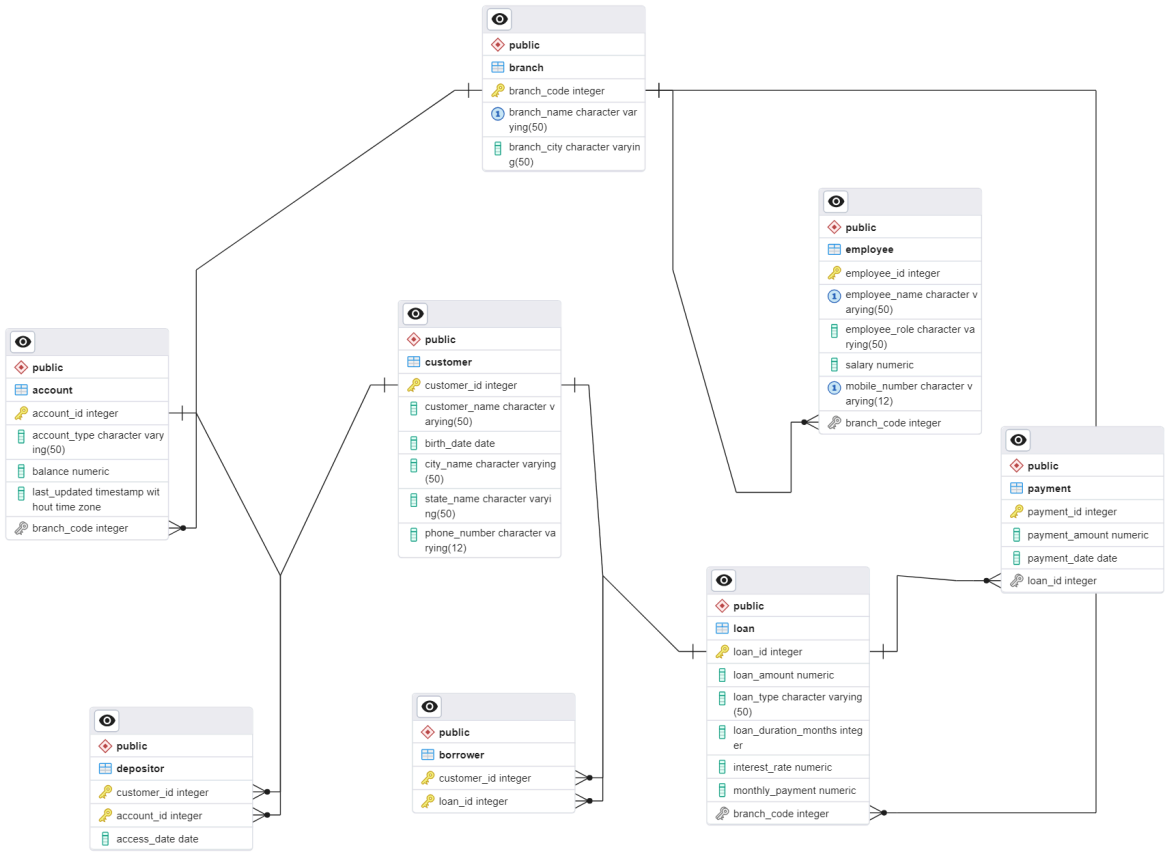
Group No. - 20

Members - Mohammad Adnan (112101028), M. Karthikeya (112101027), Sandala Sameer (112101042)

Project Description:

This Banking Management System is designed to streamline and manage various banking operations. This system aims to provide a secure, efficient, and user-friendly platform for both customers and bank staff to perform essential banking activities.

ER - Diagram:



Schema & Constraints:

1. Branch:
    - Branch_code - PRIMARY KEY
    - Branch_name - UNIQUE
    - Branch_city - NOT NULL

2. Loan:
    - Loan_id - PRIMARY KEY
    - Loan_amount - CHECK(loan_amount > 0)
    - Loan_type - NOT NULL
    - Loan_duration_months - NOT NULL
    - Interest_rate - NOT NULL
    - Monthly_payment - CHECK(monthly_payment > 0)
    - Start_date - NOT NULL
    - Branch_code - FOREIGN KEY
3. Customer;
    - Customer_id - PRIMARY KEY
    - Customer_name - NOT NULL
    - Birth_date - NOT NULL
    - City_name - NOT NULL
    - State_name - NOT NULL
    - Phone_number - NOT NULL
4. Employee:
    - Employee_id - PRIMARY KEY
    - Employee_name - UNIQUE
    - Employee_role - NOT NULL
    - Salary - CHECK(salary > 0)
    - Telephone_number - UNIQUE
    - Branch_code - FOREIGN KEY
5. Payment:
    - Payment_id - PRIMARY KEY
    - Payment_amount - NOT NULL
    - Payment_date - NOT NULL
    - Loan_id - FOREIGN KEY
6. Account:
    - Account_id - PRIMARY KEY
    - Account_type - NOT NULL
    - Balance - NOT NULL
    - Last_updated - DEFAULT current_timestamp NOT NULL
    - Branch_code - FOREIGN KEY
7. Depositor:
    - Customer_id, Account_id - PRIMARY KEY
    - Access_date - NOT NULL
8. Borrower:
    - Customer_id, Loan_id - PRIMARY KEY

Here Payment is a weak entity whose owner entity is Loan. A payment exists only if there is a loan. And the relation holding them is a dependent relation.
Employee has 2 roles: Officer (can edit data like INSERT, DELETE and UPDATE on all tables except Employee table itself) and Manager (can have access to all the tables including Employee table).

- Depositor (M:N)
- Borrower (M:N)

Here there are 2 more tables, payment_log and task table which are used to give more information and notify the employees about the pending work (like deposit, transfer, etc.) that need to be performed.

Functionality:

1. Bank structure: Organized into branches with unique names, branch codes, and city locations.
2. Customer details: Uniquely identified by customer ID; information includes name and city of residence.
3. Customer relationships: Customers may have multiple accounts and loans, associated with a specific bank employee.
4. Employee details: Identified by employee ID, information includes name, telephone number, role, and associated customers.
5. Account features: Each account has a unique number, can be held by multiple customers, and customers can have multiple accounts.
6. Account information: Bank maintains account details, including balance and recent customer access date.
7. Loan details: Loans originate at specific branches, held by one or more customers, each with a unique loan number, and the start date of loan.
8. Loan records: Bank records loan amount, payments, and payment details such as date and amount.
9. Payment identification: Although a payment doesn't uniquely identify across all loans, a payment number is specific to a particular payment for a given loan.
10. The bank uses a system where customers are kept safe from complicated technical details, while employees have access to customer information but sensitive data is hidden. Managers have complete access to manage everything in the bank, including employee data and branch operations.
11. A GUI is implemented for a better user experience and functionality performance.

TABLES:

Branch

Data Output Messages Notifications

	branch_code [PK] integer	branch_name character varying (50)	branch_city character varying (50)
1	0	Gandhi Maidan	Patna
2	1	L.B Nagar	Hyderabad
3	2	Chandranagar	Palakkad
4	3	Koramangala	Bengaluru
5	4	Bandra	Mumbai
6	5	Beemli	Vizag

Loan

	loan_id [PK] integer	loan_amount numeric	loan_type character varying (50)	loan_duration_months integer	interest_rate numeric	monthly_payment numeric	branch_code integer	start_date date
1	0	1000000	Education Loan	48	4.5	25000	0	2024-04-29
2	1	2000000	Business Loan	30	7.5	50000	1	2024-04-29
3	2	3000000	House Loan	60	5	35000	5	2024-04-29
4	3	100000	Personal Loan	10	8	20000	2	2024-04-12
5	4	3000000	Car Loan	36	6	30000	4	2024-03-26
6	5	2000000	Land Loan	30	5.5	40000	3	2024-04-29
7	6	500000	Personal Loan	24	6.5	22500	5	2024-04-29
8	7	10000000	House Loan	120	7	150000	3	2024-04-29

Account

Data Output Messages Notifications

	account_id [PK] integer	account_type character varying (50)	balance numeric	last_updated timestamp without time zone	branch_code integer
1	0	Saving Account	60000	2024-03-08 01:31:49.069735	0
2	1	Zero Balance Account	7000	2024-03-04 11:42:35	1
3	2	Checking Account	20000	2024-03-01 06:30:15	2
4	3	Zero Balance Account	3000	2024-02-18 10:02:00	3
5	4	Saving Account	150000	2024-03-08 01:31:49.069735	4
6	5	Checking Account	15000	2024-02-17 08:21:30	5
7	6	Checking Account	30000	2024-03-08 01:31:49.069735	0
8	7	Zero Balance Account	10000	2024-01-12 00:00:00	4
9	8	Saving Account	80000	2024-03-08 01:31:49.069735	1
10	9	Checking Account	65000	2024-02-14 07:12:09	3
11	10	Saving Account	200000	2024-03-08 01:31:49.069735	2

Payment

Data Output | Messages | Notifications

	payment_id [PK] integer	payment_amount numeric	payment_date date	loan_id integer
1	0	25000	2024-03-07	0
2	1	50000	2024-03-04	1
3	2	35000	2024-03-01	2
4	3	20000	2024-02-18	3
5	4	30000	2024-02-12	4
6	5	40000	2024-02-17	5
7	6	25000	2024-02-07	0
8	7	30000	2024-01-12	4
9	8	22500	2024-01-03	6
10	9	150000	2024-02-14	7
11	10	150000	2024-01-14	7

Customer

Data Output | Messages | Notifications

	customer_id [PK] integer	customer_name character varying (50)	birth_date date	city_name character varying (50)	state_name character varying (50)	phone_number character varying (12)
1	0	Swarna Sai Sumanth	2002-10-31	Vizag	Andhra Pradesh	6969696969
2	1	Rohan	2001-02-27	Hyderabad	Telangana	8732521233
3	2	Manish	2003-05-27	Bengaluru	Karnataka	8389772123
4	3	Adnan	2004-04-15	Patna	Bihar	6191619919
5	4	Nakul	2004-11-24	Palakkad	Kerala	8888888888
6	5	Kajol	1993-07-12	Mumbai	Maharashtra	7281653921
7	6	Shreya	2001-08-07	Bengaluru	Karnataka	8632123128
8	7	Sameer	1987-01-31	Patna	Bihar	6719182313

Total rows: 8 of 8 | Query complete 00:00:00.079

Employee

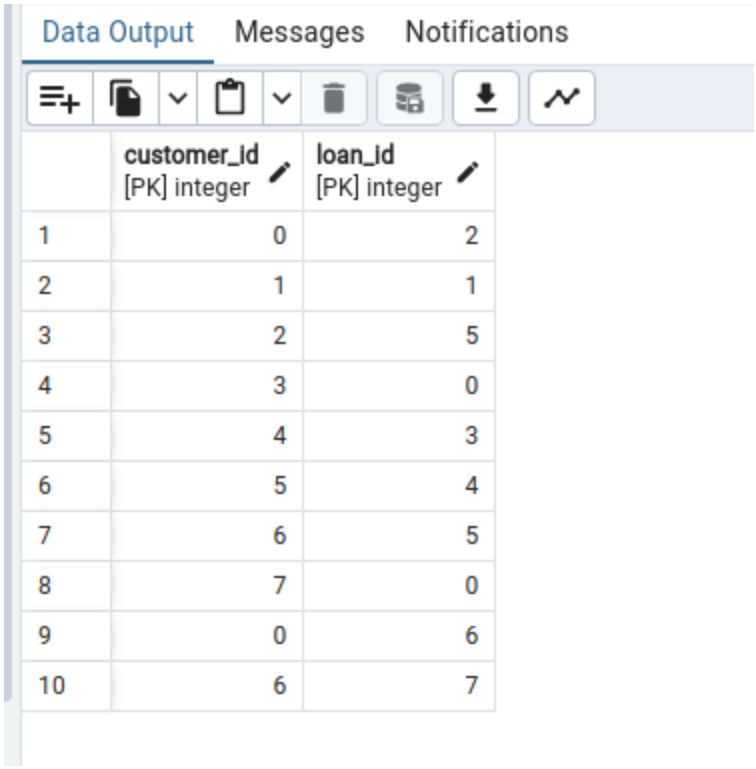
	employee_id [PK] integer	employee_name character varying (50)	employee_role character varying (50)	salary numeric	mobile_number character varying (12)	branch_code integer
1	0	Kiran	Manager	80000	8125637990	1
2	1	Mike	Manager	75000	9846736215	0
3	2	Vijay	Officer	50000	9483725646	3
4	3	Vaheed	Manager	80000	7686657654	4
5	4	Paul	Officer	60000	77698945348	5
6	5	Pranathi	Manager	85000	9988765643	5
7	6	Shruthi	Officer	45000	9273645519	2
8	7	Ravi	Manager	85000	9499664490	3
9	8	Ram	Officer	55000	8156763990	1
10	9	John	Officer	40000	9846123215	0
11	10	Venkat	Manager	70000	8663455199	2
12	11	Sharmila	Officer	50000	7647557654	4

Depositor

Data Output Messages Notifications

	customer_id [PK] integer	account_id [PK] integer	access_date date
1	0	5	2024-03-08
2	1	1	2024-03-08
3	2	3	2024-03-08
4	3	0	2024-03-08
5	4	2	2024-03-08
6	5	4	2024-03-08
7	6	9	2024-03-08
8	7	6	2024-03-08
9	4	10	2024-03-08
10	5	7	2024-03-08
11	1	8	2024-03-08

Borrower



Data Output | Messages | Notifications

	customer_id [PK] integer	loan_id [PK] integer
1	0	2
2	1	1
3	2	5
4	3	0
5	4	3
6	5	4
7	6	5
8	7	0
9	0	6
10	6	7

FUNCTIONS, PROCEDURES and TRIGGERS:

Function to show balance in account by taking customer_id:

The Show_balance function retrieves all account information for a given customer ID. Let's break down its functionality:

- Parameters & Return Type:
  - Input Parameter: cid (integer) - Customer ID for which the account balance is requested.
  - Return Type: Table with columns account_id (integer), account_type (varchar), and balance (numeric).
- SQL Query Explanation:
  - The query is used along with joining essential tables to retrieve required information. By using joins between the Customer, Depositor, and Account tables, the function ensures that only valid and related data is retrieved. This helps maintain data integrity by preventing inconsistencies or errors in the retrieved information.
- Conditions for Joining:
  - Customer.customer_id = cid, Customer.customer_id = Depositor.customer_id, Depositor.account_id = Account.account_id, These relationships enforce referential integrity, ensuring that data associations between entities (customers, depositors, and accounts) are accurate and consistent.

Overall, the Show_balance function helps preserve database consistency by ensuring accurate data retrieval, maintaining referential integrity, and providing a centralized approach to accessing account balance information.

```
189 create or replace function Show_balance (cid int)
190 returns table(account_id int, account_type varchar(50), balance numeric)
191 as $$
192 begin
193     return query
194     select Account.account_id, Account.account_type, Account.balance from Customer, Depositor, Account
195     where Customer.customer_id = cid
196     and Customer.customer_id = Depositor.customer_id
197     and Depositor.account_id = Account.account_id;
198 end; $$ language plpgsql;
199
200 select * from Show_balance (0);
```

Data Output | Messages | Notifications

	account_id integer	account_type character varying	balance numeric
1	5	Checking Account	15000

Procedure to deposit amount:

This procedure Deposit represents the amount to be deposited and the account ID from which the deposit is requested, respectively.

- Parameters: The procedure takes amoun_depositedt and aid as input parameters.
- Deposit Process:
    - It updates the account table by adding the deposit amount (amount_deposited) to the account's balance (balance+amount).
    - It then updates the depositor table, setting the access_date to the current date for the account (account_id=aid). This update signifies that the account was accessed or modified on the current date via deposit.

In summary, this procedure automates the deposit process by updating balances after successful deposits, and changing access dates for tracking purposes.

```
178 create or replace procedure Deposit (amount_deposited numeric, aid int)
179 as $$
180 begin
181     update account
182     set balance = balance + amount_deposited
183     where account_id = aid;
184     update depositor
185     set access_date = current_date
186     where account_id = aid;
187 end; $$ language plpgsql;
188
189 call Deposit (10000, 2);
190 select * from Account;
```

Data Output | Messages | Notifications

	account_id [PK] integer	account_type character varying (50)	balance numeric	last_updated timestamp without time zone	branch_code integer
1	0	Saving Account	60000	2024-03-08 01:31:49.069735	0
2	1	Zero Balance Account	7000	2024-03-04 11:42:35	1
3	3	Zero Balance Account	3000	2024-02-18 10:02:00	3
4	4	Saving Account	150000	2024-03-08 01:31:49.069735	4
5	5	Checking Account	15000	2024-02-17 08:21:30	5
6	6	Checking Account	30000	2024-03-08 01:31:49.069735	0
7	7	Zero Balance Account	10000	2024-01-12 00:00:00	4
8	8	Saving Account	80000	2024-03-08 01:31:49.069735	1
9	9	Checking Account	65000	2024-02-14 07:12:09	3
10	10	Saving Account	200000	2024-03-08 01:31:49.069735	2
11	2	Checking Account	30000	2024-03-01 06:30:15	2

Procedure to withdraw amount:

This procedure Withdraw represents the amount to be withdrawn and the account ID from which the withdrawal is requested, respectively.

- Parameters: The procedure takes amount and aid as input parameters.
- Balance Check: It checks if the balance in the specified account (aid) is sufficient to cover the withdrawal amount (amount). If the balance is less than the withdrawal amount, it raises a notice indicating "insufficient balance in account."
- Withdrawal Process:
    - If the balance is sufficient, it proceeds to update the account table by subtracting the withdrawal amount (amount) from the account's balance (balance-amount).
    - It then updates the depositor table, setting the access_date to the current date for the account (account_id=aid). This update signifies that the account was accessed or modified on the current date via withdrawal.

In summary, this procedure automates the withdrawal process by checking account balances, updating balances after successful withdrawals, and changing access dates for tracking purposes.

```
--procedure withdraw
create or replace procedure Withdraw(amount numeric,aid int)
as $$
begin
    if((select balance from account where account_id=aid) < amount) then
        raise notice 'insufficient balance in account';
    else
        update account
        set balance=balance-amount where account_id=aid;
        update depositor
        set access_date=current_date where account_id=aid;
    end if;
end;$$ language plpgsql;
call Withdraw(1000,5);
select * from account
```

Data Output | Messages | Notifications

	account_id [PK] integer	account_type character varying (50)	balance numeric	last_updated timestamp without time zone	branch_code integer
1	0	Saving Account	60000	2024-04-29 23:16:38.159726	0
2	1	Zero Balance Account	7000	2024-03-04 11:42:35	1
3	2	Checking Account	20000	2024-03-01 06:30:15	2
4	3	Zero Balance Account	3000	2024-02-18 10:02:00	3
5	4	Saving Account	150000	2024-04-29 23:16:38.159726	4
6	6	Checking Account	30000	2024-04-29 23:16:38.159726	0
7	7	Zero Balance Account	10000	2024-01-12 00:00:00	4
8	8	Saving Account	80000	2024-04-29 23:16:38.159726	1
9	9	Checking Account	65000	2024-02-14 07:12:09	3
10	10	Saving Account	200000	2024-04-29 23:16:38.159726	2
11	5	Checking Account	14000	2024-02-17 08:21:30	5

Procedure to transfer amount from one account to another:

The Transfer procedure is designed to facilitate fund transfers between two accounts (`sid` for sender ID and `rid` for receiver ID) while ensuring data consistency and integrity. Here are the points explaining its functionality:

- Parameters: The procedure takes three input parameters: sid (sender ID), rid (receiver ID), and amount (numeric)

- Transfer Process: If the sender's account doesn't have enough balance it results in exception raised as "Insufficient balance in sender account", otherwise it updates the account table twice:
    - Decreases the balance of the sender's account (`sid`) by the transfer amount (`balance-amount`).
    - Increases the balance of the receiver's account (`rid`) by the transfer amount (`balance+amount`).
    - Additionally, it updates the `depositor` table by setting the `access_date` to the current date for the sender's account (account_id=sid). This update signifies that the sender's account was accessed or modified via transfer.
- Data Integrity and Consistency: By performing the transfer process within a transaction (implicit within the procedure), the procedure helps maintain data integrity and consistency. This ensures that the sender's balance is deducted before the receiver's balance is updated, preventing inconsistencies due to concurrent transactions.

Overall, this procedure encapsulates the transaction logic for fund transfers, providing a centralized and controlled method for processing transfers.

```
--procedure transfer
create or replace procedure Transfer(sid int, rid int, amount numeric)
as $$
begin
    if((select balance from account where account_id=sid ) < amount) then
        raise notice 'insufficient balance in sender account';
    else
        update account
        set balance=balance-amount where account_id=sid;
        update account
        set balance=balance+amount where account_id=rid;
        update depositor
        set access_date=current_date where account_id=sid;
    end if;
end;$$ language plpgsql;
call Transfer(1,2,50);
select * from account
```

Data Output | Messages | Notifications

	account_id [PK] integer	account_type character varying (50)	balance numeric	last_updated timestamp without time zone	branch_code integer
1	0	Saving Account	60000	2024-04-29 23:16:38.159726	0
2	3	Zero Balance Account	3000	2024-02-18 10:02:00	3
3	4	Saving Account	150000	2024-04-29 23:16:38.159726	4
4	6	Checking Account	30000	2024-04-29 23:16:38.159726	0
5	7	Zero Balance Account	10000	2024-01-12 00:00:00	4
6	8	Saving Account	80000	2024-04-29 23:16:38.159726	1
7	9	Checking Account	65000	2024-02-14 07:12:09	3
8	10	Saving Account	200000	2024-04-29 23:16:38.159726	2
9	5	Checking Account	14000	2024-02-17 08:21:30	5
10	1	Zero Balance Account	6950	2024-03-04 11:42:35	1
11	2	Checking Account	20050	2024-03-01 06:30:15	2

Procedure to create account for a new customer who is not there before:

The create_new_account procedure inserts a new customer record into the customer table while ensuring data consistency and integrity. Here are the points explaining its functionality:

- Parameters: It takes parameters such as fullname, dob (date of birth), c_name (city name), s_name (state name), p_no (phone number), and acc_type (account type) to create a new customer account.
- Creation Process:
    - The customer_id for the new record is automatically generated by incrementing the maximum existing customer_id in the table.
    - It inserts a new account record into the account table with details such as the account type and branch code.
    - The account_id for the new record is automatically generated by incrementing the maximum existing account_id in the table.
    - The branch code is fetched from the branch table based on the city name (c_name).
    - It inserts a record into the depositor table to link the newly created customer and account.
    - The customer_id and account_id are fetched from the newly inserted customer and account records, respectively.
    - The access_date is set to the current date, indicating the date when the account was accessed or created.
- Data Integrity and Consistency:
    - Reliably generates new customer and account IDs by incrementing the maximum existing IDs, ensuring uniqueness and avoiding conflicts.

- Executes multiple insert operations within a single transaction, ensuring that either all operations complete successfully or none of them take effect. This maintains data consistency during the creation of new accounts.
- Establishes referential integrity by linking the new customer (customer_id) with the new account (account_id) through the depositor table, ensuring that only valid customer-account relationships are created.

Overall, this procedure enhances database functionality by automating the creation of new customer accounts, maintaining data consistency through transaction management, and ensuring data integrity and reliability through ID assignment and referential integrity enforcement.

```
--procedure new account
create or replace procedure create_new_account
(fullname varchar(50),dob date,c_name varchar(50),s_name varchar(50),p_no varchar(12),acc_type varchar(50))
as $$
begin
    insert into customer(customer_id,customer_name,birth_date,city_name,state_name,phone_number)
    values((select max(customer_id) from customer)+1,fullname,dob,c_name,s_name,p_no);
    insert into account(account_id,account_type,branch_code)
    values (((select max(account_id) from account)+1),acc_type,(select branch_code from branch where branch_city=c_name));
    insert into depositor(customer_id,account_id,access_date)
    values ((select max(customer_id) from customer),(select max(account_id) from account),current_date);
end;$$ language plpgsql;
```

Procedure to create a new account for an existing customer:

The create_another_account procedure inserts a new account record into the account table while ensuring data consistency and integrity. Here are the points explaining its functionality:

- Parameters: It takes parameters as cid (customer_id) and acc_type (account type)..
- Creation Process:
    - It inserts a new account record into the account table with unique account ID, account type, and branch code based on the customer's city.
    - Checks if the account type already exists for the customer and raises a notice if so.
    - The account_id for the new record is automatically generated by incrementing the maximum existing account_id in the table.
    - The branch code is fetched from the branch table based on the city name (c_name).
    - It inserts a record into the depositor table to link the newly created account with the customer.
    - The access_date is set to the current date, indicating the date when the account was accessed or created.
- Data Integrity and Consistency:
    - Ensures data integrity by preventing the creation of duplicate account types for the same customer.
    - Verifies customer-account relationships through joins between the customer, depositor, and account tables to check for existing account types.

Overall, this procedure enhances database functionality by automating the creation of new accounts and ensuring data integrity and reliability through ID assignment and referential integrity enforcement.

```
--procedure another account
create or replace procedure create_another_account(cid int,acc_type varchar(50))
as $$
begin
    if(acc_type = any(select account.account_type from customer,depositor,account where customer.customer_id=cid and customer.customer_id=depositor.customer_id and depositor.account_id=account.account_id)) then
        raise notice 'account type already exisits';
    else
        insert into account(account_id,account_type,branch_code)
        values (((select max(account_id) from account)+1),acc_type,(select branch_code from branch where branch_city=(select city_name from customer where customer_id=c_id)));
        insert into depositor(customer_id,account_id,access_date)
        values((select customer_id from customer where customer_id=cid),(select max(account_id) from account),current_date);
    end if;
end;$$ language plpgsql;
```

Trigger to trigger to give balance according to account type when we add a new account:
Need for the trigger : When we create a new account either for the new customer or the existing one we are not mentioning balance in the account. Based on the account type the function give_bal() sets the balance.
This trigger give_cal gets invoked after every insert on the account table.

```
--trigger for give balance according to account type
create or replace function give_bal()
returns trigger
as $$
begin
    if((select account_type from account where account_id=new.account_id)= 'Zero Balance Account') then
        update account
        set balance=0
        where account_id=new.account_id;
        return new;
    else
        update account
        set balance=5000
        where account_id=new.account_id;
        return new;
    end if;
end;$$ language plpgsql;

create trigger give_cal
after insert
on account
for each row
execute procedure give_bal();
```

Procedure to create a loan:
Procedure to create a loan. This takes customer_id, loan_amount, loan type, loan duration, interest rate.
We are inserting in the loan table, payment table and borrower table.
While inserting in the loan table we are not adding any monthly_payment which needs some calculation of compound interest for which we have created a separate trigger function. And in the payment table payment is set to 0 since the customer has not done any payment yet, this

refers to the customer just created the loan. In the borrower table we add the customer_id and the new loan_id.
We are getting branch code from the customer_id given as a parameter.

```
--procedure create loan
create or replace procedure create_loan(cid int,l_amount numeric,l_type varchar(50),l_duration int,int_rate numeric)
as $$
begin
    insert into loan(loan_id,loan_amount,loan_type,loan_duration_months,interest_rate,branch_code)
    values((select max(loan_id) from loan)+1,l_amount,l_type,l_duration,int_rate,
           (select branch_code from branch where branch_city=(select city_name from customer where customer_id=cid)));
    insert into borrower(customer_id,loan_id)
    values (cid,(select max(loan_id) from loan));
    insert into payment(payment_id,payment_amount,payment_date,loan_id)
    values((select max(payment_id) from payment)+1,0,current_date,(select max(loan_id) from loan));
end;$$ language plpgsql;
```

Trigger function to update monthly_payment in loan table:
Need for the trigger: when we create a new loan we are not mentioning anything about monthly_payment which will be calculated in this trigger function get_payment() by using compound interest.
After calculating monthly payment we update the loan with this monthly payment.
Trigger get_monthly_payment invokes after every insert on the loan table.

```
--trigger to get monthly payment
create or replace function get_payment()
returns trigger
as $$
declare pay_amount int;
begin
    pay_amount = ((select loan_amount from loan where
                loan_id=(select max(loan_id) from loan))*((select interest_rate from loan where loan_id=(select max(loan_id) from loan))/(12*100))*power(1+((sele
    update loan
    set monthly_payment=pay_amount where loan_id=(select max(loan_id) from loan);
    return new;
end;$$ language plpgsql;

create or replace trigger get_monthly_payment
after insert
on loan
for each row
execute procedure get_payment();
call create_loan(1, 100000, 'Education Loan', 24, 4.5);
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 93 msec.
```

Procedure to pay loans:
This is a procedure to pay loans. It takes customer_id, account_id and loan_id as parameters. We get monthly payment that the customer has to pay from loan table and if customer has enough balance to pay the loan amount gets deducted from his balance and also loan amount that he has to still pay will be reduced by monthly payment (paid amount). Since the loan amount is reduced, we have to recalculate monthly payment. And also since we have done the payment, a new record should be added in the payment table with customer_id, loan_id, paid amount and date. This will also be done by the trigger.
Here we have a new table payment_log which stores customer_id, account_id, loan_id which will be used in the trigger for calculating new monthly payment. So, we insert customer_id, account_id, loan_id into the payment_log table.

```
--procedure to pay loans
create or replace procedure pay_bills(cid int, aid int, lid int)
as $$
declare amount int;
declare bal int;
declare monthly_amt numeric;
begin
select loan.monthly_payment from loan join borrower on borrower.loan_id = loan.loan_id
join customer on customer.customer_id = borrower.customer_id where customer.customer_id = cid and loan.loan_id=lid into amount;
select balance from account where account_id = aid into bal;
if amount>bal then raise exception 'insufficient balance';
else
update account
set balance = balance - amount, last_updated = current_date where account_id = aid;
update loan
set loan_amount = loan_amount - amount where loan_id = lid;
insert into payment_log values(cid,aid,lid);
end if;
end;$$ language plpgsql;
```

Data Output | Messages | Notifications

```
CREATE PROCEDURE

Query returned successfully in 57 msec.
```

Trigger to calculate new monthly payment to update loan table and add details to payment table:

Need for trigger: This trigger is used to update payment table after paying loan and also to update the loan table with new monthly payment.

Function update_payment() returns a trigger which calculates new monthly amount by compound interest calculation. We get the loan information from the payment_log that we have created above.

After getting a new monthly payment we update the loan table with this new monthly payment and we insert a new record in the payment table with the payment_amount of the old monthly amount in the loan table.

Trigger pay_update gets invoked for every insertion on the payment_log table.

```
--trigger for payment_update
create or replace function update_payment()
returns trigger as $$
declare amount numeric;
declare lid int;
begin
    lid = new.lid;
    select monthly_payment from loan where loan_id = lid into amount;
    insert into payment (payment_id,payment_amount, payment_date, loan_id)
    values ((select max(payment_id) from payment)+1,amount, current_date, lid);
    update loan
    set monthly_payment = ((select loan_amount from loan where loan_id=lid)*((select interest_rate from loan where loan_id=lid)/(12*100))*power(1+((select interes
    where loan_id = lid;
    return new;
end;$$ language plpgsql;

create or replace trigger pay_update
after insert on payment_log
for each row
execute function update_payment();
```

Data Output | Messages | Notifications

```
CREATE TRIGGER

Query returned successfully in 85 msec.
```

Function to show number of accounts a customer has:

This function takes customer_id as a parameter and outputs the number of accounts that customer has.

```
--function no of accounts
create or replace function no_of_accounts(cid int,out ans int)
as $$
begin
    select count(*) into ans from depositor
    group by(customer_id) having customer_id=cid;
end;$$ language plpgsql;
```

VIEWS:

View to show customer details:

This view customer_details gives the details of customer name, date of birth, address and all the accounts the customer is holding.

For this we join the customer table, depositor table and account table to retrieve data about customers and their accounts.

```
--view for customer details
create or replace view customer_details
as
select customer.customer_name, customer.birth_date, customer.city_name||', '||customer.state_name as address, string_agg(account.account_type,', ') as all_account_types from customer
join depositor on customer.customer_id=depositor.customer_id
join account on depositor.account_id=account.account_id
group by customer.customer_id;

select * from customer_details;
```

Data Output | Messages | Notifications

	customer_name character varying (50)	birth_date date	address text	all_account_types text
1	Adnan	2004-04-15	Patna, Bihar	Saving Account
2	Kajol	1993-07-12	Mumbai, Maharashtra	Saving Account, Zero Balance Account
3	Nakul	2004-11-24	Palakkad, Kerala	Checking Account, Saving Account
4	Swarna Sai Sumanth	2002-10-31	Vizag, Andhra Pradesh	Checking Account
5	Shreya	2001-08-07	Bengaluru, Karnataka	Checking Account
6	Manish	2003-05-27	Bengaluru, Karnataka	Zero Balance Account
7	Sameer	1987-01-31	Patna, Bihar	Checking Account
8	Rohan	2001-02-27	Hyderabad, Telangana	Saving Account, Zero Balance Account

View to show loan details for a customer:

This view shows details of customers and their loans like loan_id, loan_amount

For this we join the customer table, borrower table and loan table to retrieve data about customers and loans they have.

```
--view for customers and their loans
create or replace view customer_loan
as
select customer.customer_id,customer.customer_name, loan.loan_id, loan.loan_amount from customer
join borrower on customer.customer_id=borrower.customer_id
join loan on borrower.loan_id=loan.loan_id;
```

View to show the payment history done by a customer to their loans:

This loan shows all the payments done on that loan by the customer from the payment table.

For this we will have to join customer table with borrower table and then with the payment table.

```
--view for payment done by a customer for a loan
create or replace view payment_for_loan
as
select customer.customer_id,customer.customer_name,payment.payment_amount,payment.loan_id from customer
join borrower on customer.customer_id=borrower.customer_id
join payment on payment.loan_id = borrower.loan_id;
```

ROLES:

There are 3 roles (Officer, Manager, Customer). Manager gets the access over all the tables including the employee table. Manager can make deletions and insertions on all tables. Whereas Officers do not get to make any update, delete or insert on the employee table.
All tables except the employee table will be granted to the Officer. Officer also gets to execute all the functions and procedures just like a manager. Only select privilege is given to the Officer on the employee table.
Customers only have select privilege on Customer table, Account table, Loan table, Payment Table of their customer id.
Whenever a customer wants to transfer an amount or create an account or loan (which needs update or insert on tables which customer cannot). For this we will have a task table which stores all the tasks and parameters for that particular task which will be inserted by the customer (customer will have insert access to this table). Then the employee also accesses this table and clears the tasks.

```
--create role officer
create role officer;
grant select on all tables in schema public to officer;
grant execute on all functions in schema public to officer;
grant execute on all procedures in schema public to officer;
grant insert, update, delete on branch, loan, customer, payment, account, depositor, borrower to officer;

--create manager role
create role manager;

grant all privileges on all tables in schema public to manager;
grant execute on all functions in schema public to manager;
grant execute on all procedures in schema public to manager;
grant select on all tables in schema public to manager;

--create role customer of bank
create role customer_of_bank;
grant select on customer,account,loan,payment to customer_of_bank;
```

List of all index:

```
--index for customer_id
create index ind_cid on customer using hash(customer_id);
--index for account id
create index ind_aid on account using hash(account_id);
--index for balance
create index ind_bal on account using Btree(account_id);
--index for employee_id
create index ind_emp_id on employee using hash(employee_id);
--index for salary
create index ind_salary on employee using Btree(salary);
--index for loan amount
create index ind_loan_amount on loan using Btree(loan_amount);
--index for loan_id
create index ind_lid on loan using hash(loan_id);
--index for loan_amount including loan_type
create index ind_l on loan(loan_amount) include(loan_type);
```

ind_cid, ind_aid are used while retrieving customer details where we need account_id and customer_id.
Index for loan amount is used when calculating monthly loan amount and also when creating a new loan where it gives maximum laon_id by scanning loan_ids.