Data Management, Warehousing, And Analytics

Lab3: Transaction & Normalization
(Summer 2023)

Submitted by: Arihant Dugar (B00917961)

GitLab repo: https://git.cs.dal.ca/dugar/csci5408_s23_b00917961_arihant_dugar/-/tree/main/Lab3

Summary:

In the lab 3, we explored transactions, normalization, and save points in a banking application database. Transactions ensure data integrity by committing or rolling back changes as a single unit. Normalization organizes data into separate tables to eliminate redundancy. Save points allow us to set recovery points within a transaction and rollback if needed. We handled transaction status verification using business logic and adjusted the update balance query based on the status. Understanding these concepts is vital for maintaining data consistency and designing reliable database systems.

Steps Performed:

1. Created a database with given attributes for table.
2. Added test data.
3. Explored save points.
4. Added save points to the store procedure in case of transaction fails.
5. Verified that in case of declined it is rollback to the savepoint.

Lab Exercise:

1. **Design a banking application database with the customer's details (minimal attributes - name, mailing address, permanent address, primary email, primary phone number), account details (minimal attributes – account number, account balance) and account transfer details (minimal attributes - account number, date of transfer, recipient name, status(varchar value waiting/accepted/declined)).**

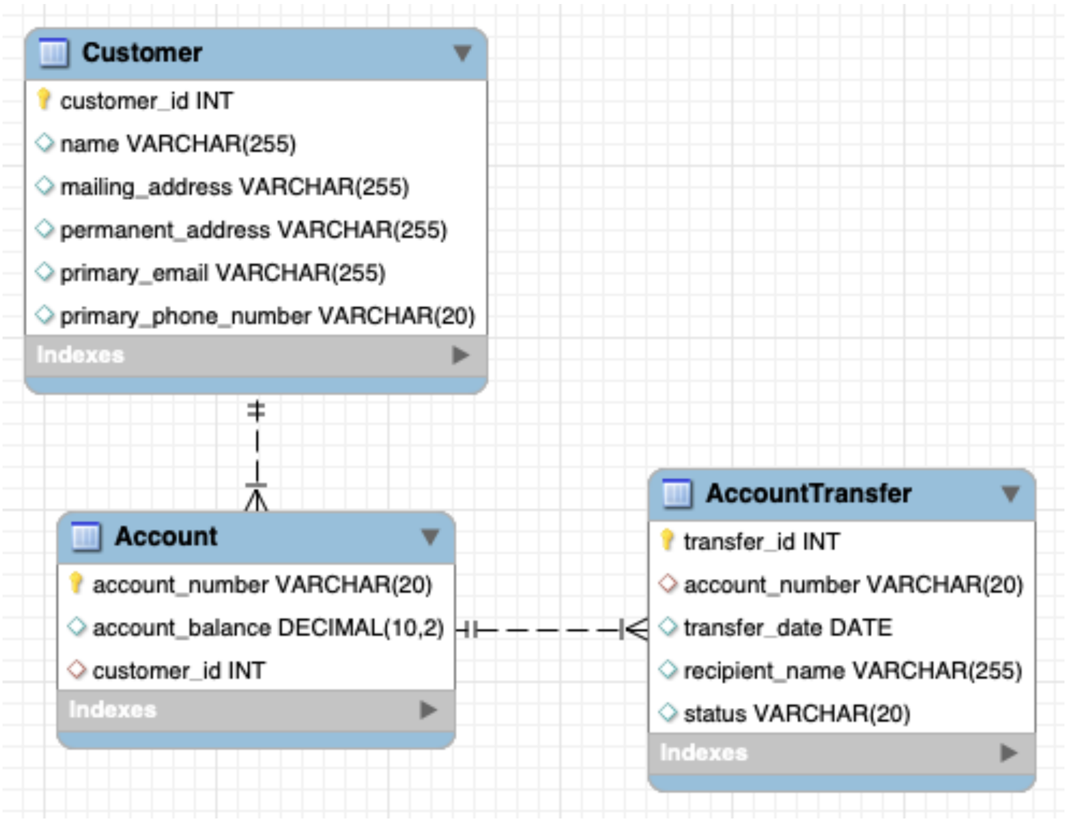
   Below are the queries for the banking tables created under the schema **bank**:

```
CREATE TABLE Customer (
  customer_id INT PRIMARY KEY,
  name VARCHAR(255),
  mailing_address VARCHAR(255),
  permanent_address VARCHAR(255),
  primary_email VARCHAR(255),
  primary_phone_number VARCHAR(20)
);

CREATE TABLE Account (
    account_number VARCHAR(20) PRIMARY KEY,
    account_balance DECIMAL(10, 2),
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES Customer(customer_id)
);

CREATE TABLE AccountTransfer (
    transfer_id INT PRIMARY KEY,
    account_number VARCHAR(20),
    transfer_date DATE,
    recipient_name VARCHAR(255),
    status VARCHAR(20),
    FOREIGN KEY (account_number) REFERENCES Account(account_number)
);
```

The schema representation is below:


Customer
customer_id INT
name VARCHAR(255)
mailing_address VARCHAR(255)
permanent_address VARCHAR(255)
primary_email VARCHAR(255)
primary_phone_number VARCHAR(20)
Indexes
Account
account_number VARCHAR(20)
account_balance DECIMAL(10,2)
customer_id INT
Indexes
AccountTransfer
transfer_id INT
account_number VARCHAR(20)
transfer_date DATE
recipient_name VARCHAR(255)
status VARCHAR(20)
Indexes

Sample queries for test data:

```
INSERT INTO Customer (customer_id, name, mailing_address, permanent_address,
primary_email, primary_phone_number) VALUES (1, 'John Doe', '123 Main St', '456
Elm St', 'john.doe@example.com', '555-1234');

INSERT INTO Account (account_number, account_balance, customer_id)
VALUES ('123456789', 1000.00, 1);

INSERT INTO AccountTransfer (transfer_id, account_number, transfer_date,
recipient_name, status) VALUES (1, '123456789', '2023-06-01', 'Jane Smith',
'waiting');
```

2. **Create a transaction environment where on every account debit, the record in the account details table is updated but not committed. Upon this update, insert a record in the account transfer details table with waiting state. (2nd + 3rd together below)**
3. **Assume that the transaction status gets verified by some X business logic (pure assumption here, no need to implement any logic), the transaction status is changed to either accepted or declined. Based on this status, handle the update balance query.**

```
DELIMITER //
CREATE PROCEDURE ProcessTransfer()
BEGIN
	DECLARE status ENUM('waiting', 'accepted', 'declined');
   START TRANSACTION;
   SAVEPOINT before_debit_savepoint;

   UPDATE Account
   SET account_balance = account_balance - 5
   WHERE account_number = '123456789';

   INSERT INTO AccountTransfer (transfer_id, account_number, transfer_date,
recipient_name, `status`)
   VALUES (2, '123456789', CURDATE(), 'Justin', 'waiting');

   UPDATE Account
   SET account_balance = account_balance + 5
   WHERE account_number = 1001;

   SET @randomNumber := FLOOR(RAND() * 2);
   SET status = IF(@randomNumber = 0, 'accepted', 'declined');

   IF status = 'accepted' THEN
      update AccountTransfer set `status` = 'accepted' where transfer_id =2;
      COMMIT;
   ELSE
      ROLLBACK TO before_debit_savepoint;
```

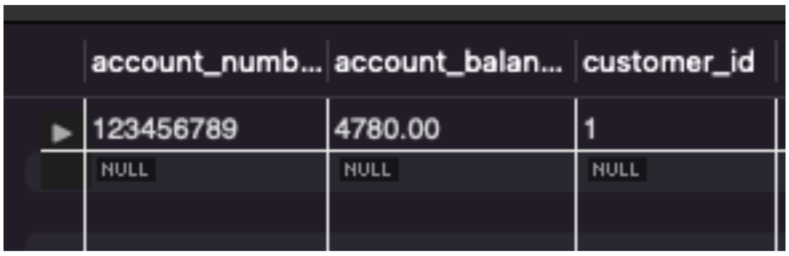
```
    INSERT INTO AccountTransfer (transfer_id, account_number, transfer_date, 
recipient_name, `status`)
    VALUES (2, '123456789', CURDATE(), 'Justin', 'declined');
  END IF;

END//
DELIMITER ;
CALL ProcessTransfer();
DROP PROCEDURE IF EXISTS ProcessTransfer;
```

Assumptions:
Creating a random number and setting the status in AccountTransfer if **accepted** or else reverting back to the save point **before_debit_savepoint**.

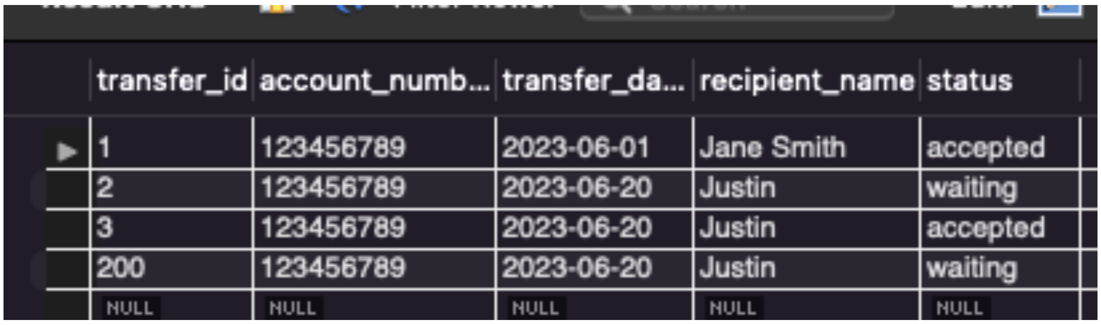
Output:

Account:



account_numb...	account_balan...	customer_id
123456789	4780.00	1
NULL	NULL	NULL

AccountTransfer:



transfer_id	account_numb...	transfer_da...	recipient_name	status
1	123456789	2023-06-01	Jane Smith	accepted
2	123456789	2023-06-20	Justin	waiting
3	123456789	2023-06-20	Justin	accepted
200	123456789	2023-06-20	Justin	waiting
NULL	NULL	NULL	NULL	NULL