CS1555 Recitation 9 Solution

Objective: To understand how triggers and cursors work

Before we start, download and run the script **bank_db.sql** from the course website to setup the database. The database instance is shown below:

Account

acc_no	**Ssn**	**Code**	**open_date**	**Balance**	**close_date**
123	123456789	1234	2008-09-10	500	null
124	111222333	1234	2009-10-10	1000	null

Loan

Ssn	**Code**	**open_date**	Amount	close_date
111222333	1234	2010-09-15	100	null

Bank

Code	Name	Addr
1234	Pitt Bank	111 University St

Customer

Ssn	Name	Phone	Addr	num_accounts
123456789	John	555-535-5263	100 University St	1
111222333	Mary	555-535-3333	20 University St	1

Alert

Alert_date	Balance	Loan

Notes:

- Triggers are defined on a single table in PostgreSQL.
- With the "*for each row*" option, the trigger is row-level. In this mode, there are 2 special variables **new** and **old** to refer to new and old tuples, respectively.

	Old	***New***
INSERT	No values (null)	Has the recently inserted tuple values
DELETE	Has the recently deleted tuple values	No values (null)
UPDATE	Has the values before UPDATE	Has the values after UPDATE

- If “for each row” is not specified, then the trigger is a statement trigger- i.e., the trigger is fired only once, when the triggering event is met, if the optional trigger constraint is met.
- The statements in the trigger function need to be properly ended with “**;**”
- Row level triggers return a record/row value or null:

	AFTER	**BEFORE**
INSERT	Ignored	new
DELETE	Ignored	old (non null to continue)
UPDATE	Ignored	new

- PL/SQL is SQL enhanced with control statement like any high-level programming languages. Examples include: If-Then-Else, Loops, etc. PLpgSQL is a PostgreSQL implementation of the PLSQL standard.

Part 1: Triggers

1. Create a trigger upon inserting a tuple into the table customer, it makes sure that the name is in upper cases.

```
create or replace function before_insert_on_customer()
returns trigger
as $$
begin
  new.name := upper(new.name);

  return new;
end;
$$ language plpgsql;

drop trigger if exists before_insert_on_customer on customer;
create trigger before_insert_on_customer
before insert on customer
for each row
execute procedure before_insert_on_customer();
```

2. To test how the trigger works, insert a new tuple in customer with their name in lower case, and then check whether their name was altered by the trigger to be in upper case. An example tuple may be with values ('987654321', 'foo bar', '555-535-3333', '0 walnut st', 0).

```
insert into customer values('987654321', 'foo bar', '555-535-333', '0
walnut st', 0);
```

3. Create a trigger that, when a customer opens new account (s), updates the corresponding num_accounts, to reflect the total number of accounts this customer has.

```
create or replace function func_1()
returns trigger as $$
begin
  update customer
  set num_accounts = num_accounts + 1
  where ssn = new.ssn;
  return new;
end;
$$ language plpgsql;

drop trigger if exists trig_1 on account;
create trigger trig_1
  after insert
  on account
  for each row
execute procedure func_1();
```

4. To test how the trigger works, insert a new account for customer '123456789', then display the num_accounts of that customer. An example tuple may be with values ('333', '123456789', '1234', '2010-10-10', 300, null).

```
insert into account values('333', '123456789', '1234', '2010-10-10',
300, null);
```

5. Similarly, create a trigger that, upon deleting an account, updates the corresponding num_accounts. To test the trigger, delete from the account entries for ssn='123456789'. Then check the value of num_accounts.

```
create or replace function func_2()
returns trigger as $$
begin
  update customer
  set num_accounts = num_accounts - 1
  where ssn = old.ssn;
  return old;
end;
$$ language plpgsql
```

```
drop trigger if exists trig_2 on account;
create trigger trig_2
  after delete
  on account
  for each row
execute procedure func_2();
```

6. To test the trigger, delete from the account entries for ssn='123456789'. Then check the value of num_accounts.

   ```
   delete from account where ssn='123456789';
   ```

7. Create a trigger that upon updating an account's balance, if the new balance is negative then sets the balance to 0 and create a new loan for the negative amount (for this database, assume that this can happen only once per day).

   ```
   create or replace function func_3()
   returns trigger as $$
   begin
     insert into loan
     values (new.ssn, new.code, current_date, abs(new.balance), null);
     new.balance := 0;

     return new;
   end;
   $$ language plpgsql;

   drop trigger if exists trig_3 on account;
   create trigger trig_3
     before
       update of balance
     on account
     for each row
     when (new.balance < 0)
   execute procedure func_3();
   ```

8. To test how the trigger works, update the balance of the account '124' to -50, then check the data in the Loan table.

   ```
   update account set balance = -50 where acc_no='124';
   ```

9. Create two triggers for Account and Loan tables that upon any changes in the two tables, if the sum of balance amount over all accounts is less than double the sum of loan amount over all loans, create a new alert with current date, total balance amount and total loan amount (for this database, assume that this can happen only once per day).

```
create or replace function func_4()
returns trigger as $$
declare
  totalBalance decimal(15, 2);
  totalLoan    decimal(15, 2);
begin
  select sum(balance) into totalBalance
  from account;
  select sum(amount) into totalLoan
  from loan;
  if totalBalance < totalLoan * 2 then
    insert into alert
    values (current_date, totalBalance, totalLoan);
  end if;
  return new;
end;
$$ language plpgsql;

drop trigger if exists trig_4_account on account;
create trigger trig_4_account
  after update or delete
  on account
execute procedure func_4();

drop trigger if exists trig_4_loan on loan;
create trigger trig_4_loan
  after insert or update
  on loan
execute procedure func_4();
```

10. To test the trigger, update the balance of the account '124' to 50, then check the data in the Alert table.

```
update account set balance = 50 where acc_no='124';
update account set balance = 50 where acc_no='123';
```

Part 2: Cursors

1. Create a function that returns a report with the phone number and the name of each customer that can pay their loan.
   We are having a lucky customer that is going to get double discount for their loan if they pay today. The rest of the customers are going to get a regular discount if they pay their loan today.
   The function should have as parameters the lucky customer and the discount and the output should be like the following:

   ```
   [555-535-3333] Mary you are getting the special double discount of
   2% if you pay today, [555-535-5263] John you are getting the
   discount of 1% if you pay today
   ```

```
CREATE OR REPLACE FUNCTION check_customers_can_pay(rand_number integer, discount integer)
RETURNS text AS $$
DECLARE
    report       TEXT DEFAULT '';
    rec_customer RECORD;
    count integer := 0;
    cur_customers CURSOR
        FOR SELECT name, ssn, phone
            FROM customer;
BEGIN
    -- Open the cursor
    OPEN cur_customers;

    LOOP
        -- fetch row into the film
        FETCH cur_customers INTO rec_customer;
        -- exit when no more row to fetch
        EXIT WHEN NOT FOUND;

        -- build the output
        IF count = rand_number THEN
            IF can_pay_loan(rec_customer.ssn) THEN
                report := report ||', ['||rec_customer.phone||'] '|| rec_customer.name || ' you
are getting the special double discount of ' || 2*discount ||'% if you pay today' ;
            END IF;
        ELSE
            IF can_pay_loan(rec_customer.ssn) THEN
                report := report ||', ['||rec_customer.phone||'] '|| rec_customer.name || ' you
are getting the discount of ' || discount ||'% if you pay today' ;
            END IF;
        END IF;
        count := count + 1;
    END LOOP;

    -- Close the cursor
    CLOSE cur_customers;

    RETURN report;
END;
$$
    LANGUAGE plpgsql;

select check_customers_can_pay(0,1);
```