Sistemas de Informação e Bases de Dados

Lab 07: Functions and Stored Procedures

If you have completed the previous guides, you should have the database of the Bank example already been created in your database account on the server `db.ist.utl.pt`. If you need to create the Bank database again, please refer to the instructions in Lab 01.

Tip: *Make sure you test all SQL statements, as well as the functions, and triggers appropriately. Use the psql command line console. If need, insert new values in tables, update values, or delete rows and then verify the results using SQL queries.*

Stored Procedures and Functions

1. Implement and test the following functionalities using PGPLSQL:

a) Write a function in **`net_worth`** that returns the absolute net worth of a client, that is, the difference between *(i)* all the money that customer has in accounts and *(2)* all the amounts client owes in loans to the bank. This function should have a parameter that identifies the customer name (whose net worth is to be obtained).

b) Write a function named **`branch_avg_diff`** that returns the difference between the average balance of all accounts of two given branches. The function must have two parameters that identify the branches to compare.

c) Using the **`branch_avg_diff`** function developed above, write an SQL query that determines the differences of account balances among all branches.

Tip: Consider that a query like:

```sql
SELECT b1.branch_name, b2.branch_name, b1.assets - b2.assets
FROM branch b1, branch b2
WHERE b1.branch_name <> b2.branch_name;
```

Returns the differences between the assets of all branches

d) Using the **`branch_avg_diff`**, write a query that determine is the branch that has the highest account balance among all branches (greater than every other).

Trigger-like behaviour

2. **Implement the following trigger and then implement the corresponding trigger-like behavior:**

   a) The depositor table associates customers with accounts. Create a trigger **tg_delete_account** with the following behaviour: whenever an account is deleted from the 'account' table, the records in the 'depositor' table that refer to that account are also deleted.

   b) Test the trigger created in above by inserting an account and a depositor and testing that after deleting the account, the corresponding depositor is also deleted:

```
INSERT INTO account VALUES ('B-100','Downtown', 100);
INSERT INTO depositor VALUES ('Cook','B-100');

SELECT * FROM depositor;

DELETE FROM account
WHERE account_number = 'B-100';

SELECT * FROM depositor;
```

   c) Remove (drop) the trigger created in (a) above

   d) Type the SQL commands necessary to alter the 'depositor' table, so that if an account is deleted from the 'account' table, the records in the 'depositor' table that refer to that account are also deleted:

   Drop the current foreign key from 'account' into on the 'depositor' table:

```
\d depositor

ALTER TABLE depositor DROP CONSTRAINT fk_depositor_account;
```

   Add the new foreign key:

```
ALTER TABLE depositor ADD CONSTRAINT fk_depositor_account
  FOREIGN KEY(account_number) REFERENCES account(account_number)
ON DELETE CASCADE;

\d depositor
```

   e) Test the behaviour of **ON DELETE CASCADE** using the SQL commands of (b)

f) Repeat the previous paragraph for the case of loans and the corresponding tables ‘borrower’ and ‘loan’.

Triggers

3. Implement and test the following *triggers*:

a) Write a *trigger* named **tg_update_loan** that eliminates a loan from a customer when the respective amount owed is less than, or equal to zero. Consider that the down payments of a loan are made by updating the amount field in the ‘loan’ table (executing the UPDATE instruction).

The trigger must also ensure that when the loan amount is negative, the (negative) value is added as an asset of the respective branch. Then, the respective loan is eliminated from the borrower and loan tables. For example, if the amount owed is 100 and a down payment of 150 is made, the loan must be eliminated, and the branch assets must increase by 50.

b) Test your **tg_update_loan** using the following instructions:

```sql
-- Clean up any existing rows if needed
DELETE FROM borrower WHERE loan_number ='1111';
DELETE FROM loan WHERE loan_number ='1111';

-- Insert the new values
INSERT INTO loan VALUES('1111', 'Central', 100);
INSERT INTO borrower VALUES('Adams', '1111');

-- Check the new versions if the tables ‘loan’, ‘borrower’, and ‘branch’
SELECT * FROM loan;
SELECT * FROM borrower;
SELECT * FROM branch;

-- Update ‘loan’
UPDATE loan
SET amount = amount - 101
WHERE loan_number = '1111';

-- Check ‘branch’ again...
SELECT * FROM branch;
```

c) Suppose that the bank has decided that it will stop providing loans to customers who are not associated with accounts. Write a *trigger* **tg_verify_account** that prevents this case, i.e., the customer from associating with the loan.