Assignment 8: Database Design Patterns, Part 2

Similar to last week, this week you will explore two more advanced topics in relational database usage. The first part focuses on using transactions to ensure proper concurrent access. The second part focuses on working with hierarchies in two different representations.

A template archive is provided with this assignment, so that you can simply fill in the answers between the required annotations. When you complete the assignment, repackage this archive with the name **`cs121hw8-`*`username`***, and submit it on the course website.

Part A: Banking Transactions *(36 points)*

This part uses the banking database introduced early in the course. It is suggested that you reload this data set before beginning this part of the assignment.

In a file named **`bank-procs.sql`**, create three stored procedures to perform common banking transactions:

1) **`sp_deposit(account_number, amount, OUT status)`**

   Deposits **`amount`** into the specified account. Specifically, the procedure adds the amount to the current account balance.

2) **`sp_withdraw(account_number, amount, OUT status)`**

   Withdraws **`amount`** from the specified account. Specifically, the procedure subtracts the amount from the current account balance.

3) **`sp_transfer(account_1_number, amount, account_2_number, OUT status)`**

   Transfers **`amount`** from **`account_1_number`** to **`account_2_number`**. Specifically, the procedure subtracts the amount from **`account_1_number`**'s balance, and adds the amount to **`account_2_number`**'s balance.

Make sure to write useful comments for all of your procedures.

Scoring: sp_deposit is 10pts, sp_withdraw is 13pts, and sp_transfer is 13pts.

Each procedure also has a **`status`** value, which indicates the success or failure of the operation. **Your implementations must set this status value properly.** Values are as follows:

0	The operation was completed successfully.
-1	The specified amount is negative. None of these procedures should ever receive a negative amount.
-2	An account number was invalid. (For **`sp_transfer`**, you do not need to indicate which account was invalid.)
-3	The account being deducted from had insufficient funds. No account balance may go below 0.

(This variable should be easy to set; simply use it in a **`SET`** statement as usual, and the caller will be able to retrieve the value you set it to. Also, don't worry about the situation where multiple errors occur at the same time; in those situations, just report whatever error you detect first.)

These procedures must also behave correctly in concurrent usage. In other words, they must use transactions to read and modify values in a way that will be correct in the context of concurrent access. Specifically, your operations must work correctly at the ***repeatable-read*** isolation level.

Note 1: Inside a stored procedure, you must start a transaction with **`START TRANSACTION`**; you cannot use **`BEGIN`** since this is already used to demarcate blocks of code.

Note 2: Do not set the transaction isolation level in your submission! The test harness will take care of this for you. (In general, this is a connection-level property that is set at connect time, never inside a stored procedure.)

Here are some hints for determining if an account number is invalid:

- What you will likely want to do is to presume that account numbers are valid, and then check whether operations actually read or wrote a row in the database. This avoids extra code that checks if an account exists separate from the operation being performed on the account.
- A **`SELECT INTO`** statement that retrieves no results will leave the target variable as **`NULL`**. You can therefore check if the account didn't exist with a predicate like: ***`variable`*** **`IS NULL`**
- There is a helpful MySQL function **`ROW_COUNT()`** that returns the number of rows that the most recent **`INSERT`**/**`UPDATE`**/**`DELETE`** affected. Thus, you can perform an **`UPDATE`** operation, then immediately check if **`ROW_COUNT()`** is 0 or 1, and respond accordingly.

Finally, the **`RETURN`** statement is invalid in stored procedures (it can only be used in functions), so you will either have a series of **`IF`**/**`ELSE`** constructs, or you can investigate labels[1] and the **`LEAVE`** statement.[2] (The solution uses **`IF`**/**`ELSE`**; it isn't that gross.)

The Bank Tester

To test the correctness of your procedures, a Java testing harness **`banktester.jar`** is provided to exercise your program from multiple concurrent threads. On a system with Java, you can invoke it like this, using the provided wrapper script:

```
./bank_tester -D your_db -u your_username -p your_password
```

(A **`bank_tester.bat`** batch file is also provided for Windows users with Java.)

The program will start multiple threads, each with its own connection, and then will perform concurrent accesses to your banking database for 30 seconds. After this time is completed, the program will reconcile all operations and verify that your database reflects the expected amounts.

IMPORTANT NOTE:

> The bank tester reports several kinds of failures; **some of these failures are expected!** For example, you should expect overdrafts to occur (this needs to be tested), and deadlocks may also occur. You should try to minimize the chance for deadlocks in your functions, but they will happen from time to time. The main issue you must fix is if actual account balances are inconsistent with expected balances, or if account balances go negative.
>
> You may also notice during testing that the bank tester will say that a balance is expected to be negative; this will occur if you don't properly detect overdrafts. The tester simply tries a

[1] http://dev.mysql.com/doc/refman/5.5/en/statement-labels.html
[2] http://dev.mysql.com/doc/refman/5.5/en/leave-statement.html

sequence of deposits, withdrawals and transfers, and keeps track of the balances if these operations are completed successfully. So, if your code says the withdrawal or transfer succeeds when it shouldn't, the tester will expect that the balance will go negative!

There are several other options to allow you to exercise your procedures in different ways:

- You can increase or decrease the number of accounts used with the –a *num_accts* option. As you decrease the number of accounts, the number of concurrent modifications (and also the number of deadlocks) will increase. Testing on large numbers of accounts is unlikely to expose concurrency bugs.
- You can increase or decrease the number of client threads used with the –t *num_threads* option. As you increase the number of threads, the number of concurrent modifications (and again the number of deadlocks) will increase. Testing with only one thread is useful to verify basic correctness of your code, but testing with a small number of threads will be unlikely to expose concurrency bugs.
- You can also specify different transaction isolation levels with the –I *level* option.
- You can also enable verbose output with the –v option.

Part B: Employee Hierarchies *(64 points)*

In class we discussed three common ways of representing hierarchies in a database. In this section you will explore two of them, the adjacency-list representation and the nested-sets representation.

The data you will use for this section is in the file **`make-emptree.sql`**, provided. Loading this file will create a table **`employee_hierarchy`**, which actually includes both representations of the hierarchy. From this table, two other tables are generated, **`employee_adjlist`** which includes only the adjacency-list representation, and **`employee_nestset`** which includes only the nested-sets representation. The hierarchies are identical. Every employee also has a salary, which is randomly generated, and unlike real life, it has nothing to do with where they appear in the hierarchy.

For these problems, you may not use **`employee_hierarchy`** directly; use **`employee_adjlist`** and **`employee_nestset`**. Some problems will specify one of these views that you are required to use.

Make sure you write clear comments describing your answers for each of these problems. A lack of commenting will result in point deductions.

Your implementations may benefit from the use of temporary tables, but you should be aware that MySQL's support for temp tables is pretty broken. Therefore, you should just create regular (i.e. non-temporary) tables outside of the UDF that uses them, and use them as if they were temp tables. Of course, this would never work in a multi-user environment, but the point is to learn the concepts.

You might also again benefit from the use of the **`ROW_COUNT()`** function, to tell whether an iterated query has actually made any changes.

Scoring: problem 1 is 14pts, 2 is 8pts, 3 is 6pts, 4 is 8pts, 5 is 14pts, 6 is 14pts.

Implement your answers to these questions in a file **`emptree.sql`**.

1. Write a user-defined function **`total_salaries_adjlist(emp_id)`** that uses **`employee_adjlist`** to compute the sum of all employee salaries in a particular subtree of the hierarchy. The specified employee's salary should also be included.

2. Write another user-defined function **`total_salaries_nestset(emp_id)`** that uses **`employee_nestset`** to compute the sum of all employee salaries in a particular subtree of the hierarchy. The specified employee's salary should also be included.

3. Using the **`employee_adjlist`** view, write the SQL query to find all employees that are leaves in the hierarchy; the result should include the employee ID, name, and salary, in that order.

   Note: In MySQL, the expression "***`attr IN (subquery)`***" will always evaluate to false if ***`subquery`*** produces any **`NULL`** values! This is also the case for "***`attr NOT IN (subquery)`***". The reason is the same as always: anything compared to **`NULL`** is always *unknown*.

4. Implement the same operation as in #3, but this time using the **`employee_nestset`** view.

5. Write a function **`tree_depth()`** that finds the maximum depth of the tree. You can use the tree representation of your own choosing to implement this function; however, explain why you chose the representation you chose in a comment before the function, and also make sure to explain how your implementation works. **A tree with a single node has a depth of 1!**

6. Given a particular employee ID *emp_id*, it should be obvious how one would retrieve that node's immediate children from the **`employee_adjlist`** representation. However, how would you do this for the **`employee_nestset`** representation? Implement a function **`emp_reports(emp_id)`** that uses the **`employee_nestset`** representation to compute how many "direct reports" (i.e. children) the specified employee has in the organization.

A Note about Hierarchical Schemas

The example schemas given for representing hierarchies are simply the minimal schemas necessary for representing the hierarchies. In general, a hierarchical schema could also include other values that would make specific operations against the hierarchy much easier. For example, each node could specify its depth in the tree, a relative index among its siblings, etc. A schema could even combine multiple hierarchy representations, if they are useful in different situations.

The main cost of including such additional information is simply maintaining it as the hierarchy changes structure. However, if structural changes occur infrequently and queries would benefit from the additional information, it would well be worth it.