Fall 2010 CS 3200 Class Project: Milestone 8 (Final)

The goal for this milestone is to use transactions and understand better the tradeoff between consistency and performance.

This milestone is to be completed individually (i.e., no teams). You can discuss problems with other students, but you have to create all deliverables yourself from scratch. In particular, it is not allowed to copy somebody else's code or text and modify it.

The report for this milestone is due on Monday, **December 6 at 5pm**. For late submissions you will lose one percentage point per hour after the deadline. This milestone is worth 15% of your overall homework score. Please email the deliverables to both me and Yue. You should receive a confirmation email from either of us. If you have not received a confirmation email within 12 hours after submitting your solution or by the time of the deadline, whichever comes first, you need to email us immediately to make sure we actually received your submission. (Of course, if you submit too close to the deadline, you might receive a confirmation sometime within the next 30-60 minutes after you submitted.) If you need to send multiple files, please create a single zip file. Many other attachments types, in particular rar files, are rejected by the CCIS mail server.

JDBC and Transactions

As we discussed a few weeks ago, JDBC supports transactions and various isolation levels. For this homework, a transaction essentially works as follows:

1. Connect to the database server as usual.
2. Make sure you turn auto-commit off for the Connection object.
3. For the connection, also set the transaction isolation level to the desired value.
4. Perform the actual work of the transaction, i.e., *actions* that read or update database objects.
5. Call commit on the Connection object to complete the transaction.
6. Close the connection.

Each transaction is implemented by its own Java program. (You will receive an example in an email.) For each scenario, we will concurrently execute two such transactions that work on some common database objects.

Running Two Transactions Concurrently

We illustrate the general approach with an example. Assume the person with pID = 1 in the Person table has name 'Joe'. Consider a transaction T1 performing the following actions:

1. Execute query "SELECT pID, name FROM Person WHERE pID = 1".
2. Wait 10 sec.
3. Execute query "SELECT pID, name FROM Person WHERE pID = 1" again.
4. Commit.

Let T2 be another transaction that does the following:

1. Execute query “SELECT pID, name FROM Person WHERE pID = 1”.
2. Execute update “UPDATE Person SET name = 'newName' WHERE pID = 1”.
3. Commit.

To be able to see in which order the different actions of T1 and T2 are executed, your transactions should output a status message immediately after each action. If the action is a query, output the current time, the transaction number, and the query result. For example, after the JDBC call that submitted the first query in T1, you can print a message like “2010-11-30 10:05:13, T1: 1, Joe”. Similarly, T2 can print the following message right after issuing the update action: “2010-11-30 10:05:17, T2: update processed.” You can get the current system time by using java.util.Calendar’s getTime() method. (The provided transaction example does not necessarily create the required output—you need to modify it accordingly!)

Notice the wait time we introduced in T1. We need this to enforce an attempted interleaving of the actions of T1 and T2. The 10 seconds are an arbitrary choice and you might choose a different wait time. The main requirement for the wait time is that it should be long enough to allow you to start T2 while T1 is waiting between the first and second execution of its query. Without locking, starting T2 immediately after T1 should result in the following execution order of their individual actions:

T1: first query
  T2: query
  T2: update
  T2: commit
T1: second query
T1: commit

If you will actually see this interleaved order of T1 and T2’s actions will depend on the selected isolation level. Here is one method you can use to attempt the desired interleaving:

1. Open two command shell windows.
2. In the first shell, type the Java command for running T1. In the second shell, type the Java command for running T2. Do **not** yet hit [Enter] for either.
3. Hit [Enter] in the first command shell to start T1.
4. As soon as you see T1’s first status message displaying the query result, you know that it just started waiting for about 10 secs. Hit [Enter] in the second command shell to start T2 during this wait time, as early as possible. Without any locking, T2 would now perform all its actions and hence we would see the desired interleaving of actions.
5. See what actually happens. In particular, in which order do the other status messages appear in the two command shells? Do we see the attempted interleaving of actions? Or does T2 get delayed by the DBMS until T1 has completed? And so on.

We will explore this for several scenarios.

Dirty Reads (WR Conflict)

Set up T1 and T2 so that they attempt to interleave their actions in the following order: (Use wait times to achieve this ordering, e.g., by inserting Thread.sleep(10000) at the right place in T1. Also, make sure you output status messages immediately after each action—as discussed for the example above.):

T1: insert a *new* dorm building into table DormBuilding.
  T2: find all buildingNum's for dorm buildings that have no rooms.
  T2: Commit.
T1: insert 10 rooms for the new dorm building into the RoomContain table.
T1: find all buildingNum's for dorm buildings that have no rooms.
T1: Commit.

Run this experiment first by setting T2's isolation level to Connection.TRANSACTION_READ_UNCOMMITTED and T1's isolation level to Connection.TRANSACTION_READ_COMMITTED. Then run the same experiment again, but this time setting the isolation level for *both* transactions to Connection.TRANSACTION_READ_COMMITTED. When running these transactions, make sure the **triggers** enforcing the constraints on the number of rooms per dorm building and on preventing buildings without rooms are **disabled**. (DISABLE TRIGGER name ON table;)

Report what you observe. In particular, in which order are the actions of T1 and T2 performed? Is the behavior different depending on the isolation level for T2? Which isolation level would result in better performance and which would prevent dirty reads?

Now repeat both experiments but with both room-number constraint triggers **enabled**. Does either of the two triggers fire and roll back either of the two transactions for either of the two isolation levels? What does this tell you about the moment triggers are fired? In particular, are they fired right after an action is performed or are they fired at the very end of a transaction?

Unrepeatable Reads (RW Conflict)

We now need different transaction implementations. Set up T1 and T2 so that they attempt to interleave their actions in the following order:

T1: read some tuple from table Person, e.g., the one with pID=1.
  T2: read the same tuple.
  T2: update the tuple by changing the person's name, e.g., to 'newName'.
  T2: Commit.
T1: read the same tuple again.
T1: Commit.

As before, make sure T1 waits long enough between its two read operations to allow you to start T2 during that wait time with several seconds to spare.

Run this experiment first for isolation level Connection.TRANSACTION_READ_COMMITTED by choosing this isolation level for *both* transactions. Then run the same experiment again, but this time setting isolation level for both transactions to Connection.TRANSACTION_REPEATABLE_READ.

Report what you observe, like for the previous experiment.

Phantoms (Part 1)

We again will need different transactions for this part. Set up T1 and T2 so that they attempt to interleave their actions in the following order:

T1: run COUNT(*) on table Person.
    T2: insert a new person into table Person.
    T2: Commit.
T1: run the same COUNT(*) query again.
T1: Commit.

Run this experiment first for isolation level Connection.TRANSACTION_REPEATABLE_READ by choosing this isolation level for *both* transactions. Then run the same experiment again, but this time setting isolation level for both transactions to Connection.TRANSACTION_SERIALIZABLE.

Report what you observe, like for the previous experiment.

Phantoms (Part 2)

Run the same experiments as for Phantoms, part 1, but with the following modifications applied to T1 and T2:

T1: instead of running COUNT(*) on the entire Person table, compute the count of the number of persons with pID < 100.

T2: make sure the newly inserted person has pID > 100.

Report if anything is different compared to part 1. Briefly explain the possible reasons for any differences you see.

Deliverables

Submit a report that discusses all the items requested above.

Submit the Java source files you used for all the experiments.