Project 5
SimpleDB

Out: November 19, 2017
***Due:** December 8, 2017, 11:59 PM*

1 Introduction

During this semester, we have talked a lot about the different components that comprise a DBMS. Now, you will have an opportunity to implement some of the algorithms you have seen inside a toy DBMS called SimpleDB.

SimpleDB is a multi-user transactional database server written in Java, which interacts with Java client programs via JDBC. The system is intended for pedagogical use only. The code is clean and compact. The APIs are straightforward. The learning curve is relatively small. Consequently, the system is intentionally bare-bones. It implements only a small fraction of SQL and JDBC, and does little or no error checking.

This project focuses on transactions and has three parts:

- Setup the SimpleDB database system
- Implement the *wait-die* deadlock avoidance algorithm
- Implement *nonquiescent* checkpointing

2 Part I: Setting Up SimpleDB

In order to set up SimpleDB, either run "`/course/cs1270/pub/simpledb/setup.sh`" **or** do the following manually:

- Download (`http://www.cs.bc.edu/~sciore/simpledb/SimpleDB_2.10.zip`) and extract the SimpleDB files.
- Add SimpleDB to the classpath:

```
echo export CLASSPATH=.:$HOME/SimpleDB_2.10:$CLASSPATH >> $HOME/.bashrc; source $HOME/.bashrc
```

- Run the server (creates student DB in `$HOME/student`):

```
javac simpledb/server/Startup.java; java simpledb.server.Startup student
```

- Run the `CreateStudentDB` and `StudentMajor` demo client programs from the `studentclient/simpledb` folder.

To make sure everything is working properly, try running the `SQLInterpreter` client demo. Enter some SQL statements and see what happens, but remember that SimpleDB only supports a subset of SQL. What happens when you try to execute an SQL statement that SimpleDB does not support?
Note:You will only need to modify some files in `SimpleDB_2.10/simpledb/tx`

3 Part II: Wait-Die Deadlock Avoidance

SimpleDB currently uses timeout to detect deadlock. (This means that if a transaction waits for too long, it is rolled back). Change it to use the wait-die deadlock avoidance strategy.
Wait-Die Scheme: This is a non-preemptive technique for deadlock prevention. A transaction T_1 requesting a resource (data item) currently held by another transaction T_2 is allowed to wait only if it has a timestamp smaller than that of T_2 i.e. T_1 is older than T_2, otherwise it gets rolled back (dies). In the case where a transaction requests to lock a resource already held with a conflicting lock by another transaction, one of two possibilities may occur:

- If the transaction holding the conflicting lock is older, then it's allowed to wait until the resource is available.
- If the transaction requesting the lock is younger then it dies; but, it is restarted later with a random delay with its original timestamp.

This scheme basically kills younger transactions but allows older ones to wait. Your code should modify the class `LockTable` as follows:

- The methods `sLock`, `xLock`, and `unLock` will need to take the transaction's `id`.
- The variable `locks` must be changed so that a block maps to a list of the transaction `ids` holding a lock on the block (instead of just an integer). Use a negative transaction `id` to denote an exclusive lock.
- Each time through the `while` loop in `sLock` and `xLock`, check to see if the thread needs to be aborted (i.e., if there is a transaction on the list that is older than the current transaction). If so, then it should throw a `LockAbortException`.
- You will also need to modify other classes that use `LockTable`. Figure out what those changes need to be.

The solution should require replacing a lot of code, but the resulting amount of code should not change much. To test your code, you can use the `WaitDieTest` program. The program creates three transactions (A,B,C), each in their own thread. Transactions A and C both need a lock held by transaction B. Under the wait-die algorithm, transaction A should wait, whereas transaction C should throw a `LockAbortException` exception. When you run it, you should get the following output:

```
new transaction: 1
transaction 1 committed
new transaction: 2
new transaction: 3
Tx B: write 1 start
Tx B: write 1 end
new transaction: 4
Tx C: write 2 start
Tx C: write 2 end
Tx C: read 1 start
Transaction C aborts
Undoing record <--This line comes from Part III
transaction 4 rolled back
Tx A: read 2 start
Tx A: read 2 end
Tx A: write 1 start
transaction 3 committed
Tx A: write 1 end
transaction 2 committed
```

4 Part III: Nonquiescent Checkpointing

SimpleDB currently quiesces the system when writing a checkpoint to the log, waiting for all active transactions to finish before allowing new transactions to start. Instead, implement a nonquiescent checkpointing strategy that logs a list of all active transactions at the time of the checkpoint to avoid blocking new transactions. You will need to do the following:

- Create a new class `NQCheckpoint` to handle a nonquiescent checkpoint record.
- Modify classes `LogRecord` and `LogRecordIterator` to understand `NQCheckpoint`.
- Modify the `Transaction` constructor so that it keeps track of the currently active transactions and calls the `RecoveryMgr` every 5 transactions to write an `NQCheckpoint`.
- Add a method `checkpoint` to `RecoveryMgr` that writes an `NQCheckpoint` to the log.
- Modify the `recover` method in `RecoveryMgr` to recognize and handle `NQCheckpoint`. Also modify it to print out the log records as it encounters them (so you can see what is going on). The recovery algorithm should read the log backwards, keeping track of all completed transactions. Upon encountering a nonquiescent checkpoint record `<NQCKPT T1,...,Tn>`, determine which of these transactions are still running. Then, continue reading the log backwards until finding the start record for the earliest of the enumerated transactions. All log records prior to this start record can be ignored.

- Add a `println` statement in the `undo` method of `SetIntRecord` so that while debugging, you can see when the `recover` and `rollback` methods actually undo a value.

To test your code, you can use the `NQCheckpointTest` program. When you run it, you should get the following output:

```
new transaction: 1
new transaction: 2
transaction 2 setint old=0 new=10002
transaction 2 committed
new transaction: 3
transaction 3 setint old=0 new=10003
new transaction: 4
transaction 4 setint old=0 new=10004
NQ CHECKPOINT: Transactions 1 3 4 are still active
new transaction: 5
transaction 5 setint old=0 new=10005
transaction 3 committed
transaction 5 committed
new transaction: 6
transaction 6 setint old=0 new=10006
new transaction: 7
transaction 7 setint old=0 new=10007
new transaction: 8
transaction 8 setint old=0 new=10008
transaction 4 committed
transaction 7 committed
new transaction: 9
transaction 9 setint old=0 new=10009
NQ CHECKPOINT: Transactions 1 6 8 9 are still active
new transaction: 10
transaction 10 setint old=0 new=100010
new transaction: 11
transaction 11 setint old=0 new=100011
transaction 6 committed
transaction 9 committed
transaction 11 committed
new transaction: 12
transaction 12 setint old=0 new=100012
new transaction: 13
transaction 13 setint old=0 new=100013
new transaction: 14
transaction 14 setint old=0 new=100014
```

```
transaction 8 committed
transaction 12 committed
transaction 14 committed
NQ CHECKPOINT: Transactions 1 10 13 are still active
new transaction: 15
transaction 15 setint old=0 new=100015
new transaction: 16
transaction 16 setint old=0 new=100016
new transaction: 17
transaction 17 setint old=0 new=100017
transaction 10 committed
transaction 15 committed
transaction 17 committed
transaction 1 committed
---------------------
new transaction: 18
Initiating Recovery
Here are the visited log records
<START 18>
<COMMIT 1>
<COMMIT 17>
<COMMIT 15>
<COMMIT 10>
<SETINT 17 [file testfile, block 63] 0 0>
<START 17>
<SETINT 16 [file testfile, block 62] 0 0>
Undoing record
<START 16>
<SETINT 15 [file testfile, block 61] 0 0>
<START 15>
<NQCHECKPOINT 1 10 13 >
<COMMIT 14>
<COMMIT 12>
<COMMIT 8>
<SETINT 14 [file testfile, block 60] 0 0>
<START 14>
<SETINT 13 [file testfile, block 59] 0 0>
Undoing record
<START 13>
```

Note that the last two values in the SETINT records are both 0. The first 0 denotes the offset in the block, and the second one denotes that the previous value at that offset is 0. Also note that only the updates for transactions 13 and 16 need to be undone. The

recovery method uses the NQCHECKPOINT record to know that it can stop after seeing the START record for transaction 13.

This test file gives you the flavor of how you can test your code. You should probably begin with a stripped down version of it and increase complexity as you work out the bugs.

5 Handin

Please hand in the following (note that the first item is due earlier than the others):

- **Due November 30:** The `$HOME/student/student.tbl` file.
- The entire `simpledb` subdirectory that contains all of the source code.
- A README file that clearly indicates which files you changed and how you changed them so we can easily figure out what you did.

To hand in the project, run the following command:

```
/course/cs127/bin/cs127_handin simpledb
```

Good luck!