6.830 Problem Set 3: SimpleDB Transactions

Assigned: Thursday, October 6
Due: Thursday, October 20

In this problem set, you will implement a simple locking-based transaction system in SimpleDB. You will need to add lock and unlock calls at the appropriate places in your code, as well as code to track the locks held by each transaction and grant locks to transactions as they are needed.

The remainder of this document describes what is involved in adding transaction support and provides a basic outline of how you might add this support to your database.

As with the previous problem set, we **strongly recommend** that you start as early as possible on this problem set. Locking and transactions can be quite tricky to debug!

0. Find bugs, be patient, earn candybars

This is the first time we have used SimpleDB in 6.830. Consequently, it is very possible you are going to find bugs, inconsistencies, and bad, outdated, or incorrect documentation, etc. We apologize profusely. We did our best, but, alas, we are fallible human beings.

We ask you, therefore, to do this problem set with an adventurous mindset. Don't get mad if something is not clear, or even wrong; rather, try to figure it out yourself or send us a friendly email. We promise to help out by sending bugfixes, new tarballs, etc.

...and if you find a bug in our code, we'll give you a candybar (see Section 3.3)!

1. Transactions, Locking, and Concurrency Control

You do not need to write a great deal of code for this problem set, but the code you do have to write is quite tricky. Before starting, you should make sure you understand what a transaction is and how strict two-phase locking (which you will use to ensure isolation and atomicity of your transactions) works.

In the remainder of this section, we briefly overview these concepts and discuss how they relate to SimpleDB.

1.1. Transactions

A transaction is a group of database actions (e.g., inserts, deletes, and reads) that are executed *atomically*; that is, either all of the actions complete or none of them do, and it is not apparent to an outside observer of the database that these actions were not completed as a part of a single, indivisible action.

In SimpleDB, a `TransactionID` object is created at the beginning of each query. This object is passed to each of the operators involved in the query. When the query is complete, the `BufferPool` method `transactionComplete` is called. Calling this method *commits* the transaction. At any point during its execution, an operator may throw a `TransactionAbortedException` exception, which indicates an internal error or deadlock has occurred. The test cases we have provided you with create the appropriate `TransactionID` objects, pass them to your operators in the appropriate way, and invoke `transactionComplete` when a query is finished. We have also implemented `TransactionID`.

We have provided you with a multithreaded test case, `test/TransactionTest` that creates a number of threads, with each thread running a single transaction that reads and updates the database.

1.2. Requesting Locks

You will need to implement strict two-phase locking. This means that transaction should acquire the appropriate type of lock on any object before accessing that object and shouldn't release any locks until after the transaction commits. We recommend locking at *page* granularity, though you should be able to implement locking at *tuple* granularity if you wish (please do not implement table-level locking).

You will need to add calls to SimpleDB (in `BufferPool`, for example), that allow a caller to request or release a (shared or exclusive) lock on a specific object on behalf of a specific transaction.

Rather than adding calls to these new routines in each of your operators, we recommend acquiring locks in the `BufferPool` (for example, in `getPage`). You will need to acquire a *shared* lock on any page (or tuple) before you read it, and you will need to acquire an *exclusive* lock on any page (or tuple) before you write it. You will notice that we are already passing around `Permissions` objects at various places in the BufferPool; these objects indicate the type of lock that the caller would like to have on the object being accessed (we have given you the code for the `Permissions` class.)

You will also want to think especially hard about acquiring locks in the following situations:

- Adding a new page to a `HeapFile`.
- Looking for an empty slot to insert tuples into.

1.3. Granting Locks

You will need to create data structures that keep track of which locks each transaction holds and that check to see if a lock should be granted to a transaction when it is requested.

You will need to implement shared and exclusive locks; recall that these work as follows:

- Before a transaction can read an object, it must have a shared lock on it.
- Before a transaction can write an object, it must have an exclusive lock on it.
- Multiple transactions can have a shared lock on an object.
- Only one transaction may have an exclusive lock on an object.
- If transaction *t* is the only transaction holding a shared lock on an object *o*, *t* may *upgrade* its lock on *o* to a exclusive lock.

If a transaction requests a lock that it should not be granted, your code should *block*, waiting for that lock to become available (i.e., be released by another transaction running in a different thread).

You need to be especially careful to avoid race conditions when writing the code that acquires locks -- think about how you will ensure that correct behavior results if two threads request the same lock at the same time (hint: you way wish to read about Synchronizing Threads in Java).

1.4. Recovery and Buffer Management

To simplify your job, we recommend that you implement a NO STEAL/FORCE buffer management policy. This means that:

- You shouldn't evict dirty (updated) pages from the buffer pool if they are locked by an uncommitted transaction.
- On transaction commit, you should force dirty pages to disk (e.g., write the pages out).

To further simplify your life, you may assume that SimpleDB will not crash while processing a `transactionComplete` command. Note that these three points mean that you do not need to implement log-based recovery, since you will never need to undo any work (since you never evict dirty pages) and you will never need to redo any work (since you force updates on commit and will not crash during commit processing).

1.5. Deadlocks and Aborts

It is possible for transactions in SimpleDB to deadlock (if you do not understand why, we recommend reading about deadlocks in Ramakrishnan). You will need to detect this situation throw a `TransactionAbortedException`; though there are many possible ways to detect deadlock, we recommend using something simple (for example, a timeout).

You should also make sure that your code behaves properly when the database system crashes or if the user explicitly aborts a transaction; you may wish to experiment with killing the system or a thread as the database runs.

1.6. The ACID Properties

To recap how transaction management works in SimpleDB, we briefly look at how it ensures that the ACID properties are satisfied:

Atomicity: Strict two-phase locking and careful buffer management ensure atomicity.
Consistency: The database is transaction consistent by virtue of atomicity. Other consistency issues (e.g., key constraints) are not addressed in SimpleDB.
Isolation: Strict two-phase locking provides isolation.
Durability: A FORCE buffer management policy ensures durability.

2. Practical Details

You will need to download the modified `grader.pl` from the course server. This new version runs the TransactionTest against your database. Note that you can immediately jump to running the relevant test for this assignment by invoking `grader.pl` with `-t 7`. However, in the end you need to pass all tests.

2.1. Implementation hints

Here's is a rough outline of one way you might proceed with enhancing your SimpleDB implementation with support for transactions:

- Add the new routines that allow callers to request and release locks to `BufferPool`.
- Add calls to request and release locks at the appropriate places within `BufferPool`. You may also need to acquire locks at certain places in `HeapFile`.
- Implement the classes and data structures that keep track of which locks a particular transaction holds.
- Add code that tests to see whether a given transaction should be allowed to acquire a lock and that blocks when locks are not available. This code should also detect deadlocks.

3. Logistics

You must submit your code (see below) as well as a short (2 pages, maximum) writeup describing your approach. This writeup should:

- Describe any design decisions you made, including your deadlock detection policy, locking granularity, etc.

- Discuss and justify any changes you made to the API.

3.1. Collaboration

This problem set should be manageable for a single person, but if you prefer to work with a partner, this is also OK. Larger groups are not allowed. Please indicate clearly who you worked with, if anyone, on your writeup.

3.2. Submitting your assignment

To submit your code, please create a `6.830-ps3.tar.gz` tarball (such that, untarred, it creates a `6.830-ps3/simpledb/src/` directory with your code) and email it. You may submit your code multiple times; we will use the latest version you that arrives before the submission deadline (before class on the due date). If applicable, please indicate your partner in your email. Please also attach your writeup as a PDF or text file.

3.3. Submitting a bug

Please submit (friendly!) bug reports. When you do, please try to include:

- A description of the bug.
- A `.java` file we can drop in the `src/simpledb/test` directory, compile, and run.
- A `.txt` file with the data that reproduces the bug. We should be able to convert it to a `.dat` file using `PageEncoder`.

If you are the first person to report a particular bug in the code, we will give you a candy bar!

3.4 Grading

65% of your grade will be based on whether or not your code passes the test program (`grader.pl`) we run over it. These programs will be a superset of the test programs we have provided for you. Before handing in your code, you should make sure you pass all tests in `grader.pl` (including the tests you already passed in PS2 -- it's very likely the addition of transactions will break tests that worked before!). Notice that `grader.pl` gives you some help on its usage:

```
$ ./grader.pl -h

grader.pl -- SimpleDB torture

Usage: grader.pl [-c] [-h] [-t TEST] [-v] [-x]
  -c            : don't clear .txt and .dat files on exit, default off
  -h            : print this help
  -t TEST       : only run test TEST, default all tests
  -v            : increase verbosity, default off
  -x            : exit on failed test, default off
```

```
Tests
  1 : simple table dump
  2 : filters
  3 : delete
  4 : insert
  5 : join
  6 : aggregates
  7 : transactions
```

You should look through the source code of `grader.pl` and make sure you understand roughly what is going on. It might help you debug your code.

Important: before testing, we will replace your `build.xml`, `PageEncoder.java`, `grader.pl`, and the entire contents of the `src/simpledb/test/` directory with our version of these files! This means you cannot change the format of `.dat` files! You should therefore be careful changing our APIs. This also means you need to test whether your code compiles with our test programs. In other words, we will untar your tarball, replace the files mentioned above, compile it, and then grade it. It will look roughly like this:

```
$ gunzip 6.830-ps2.tar.gz
$ tar xvf 6.830-ps2.tar
$ cd ./6.830-ps2
[replace grader.pl, PageEncoder.java, and test/]
$ ant
$ ./grader.pl
[additional tests]
```

If any of these commands fail, we'll be unhappy, and, therefore, so will your grade.

An additional 35% of your grade will be based on the quality of your writeup and our subjective evaluation of your code.