

CS 339 Lab 4: SimpleDB Transactions

Assigned: Thursday, March 3, 2022

Due: Thursday, March 17, 2022 11:59 PM Central

In this lab, 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 lab, we recommend that you start as early as possible. Locking and transactions can be quite tricky to debug!

1. Getting started

You should begin by downloading the starter code on Canvas: [SimpleDB Lab 4](#). Then set up your API - e.g., Eclipse or IntelliJ - as in the previous SimpleDB labs.

2. Transactions, Locking, and Concurrency Control

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.

2.1. 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.

2.2. The ACID Properties

To help you understand how transaction management works in SimpleDB, we briefly review 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 (see Section 2.3 below).

2.3. Recovery and Buffer Management

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

As we discussed in class, this means that:

- You shouldn't evict dirty (updated) pages from the buffer pool if they are locked by an uncommitted transaction (this is NO STEAL).
- On transaction commit, you should force dirty pages to disk (e.g., write the pages out) (this is FORCE).

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 in this lab, since you will never need to undo any work (you never evict dirty pages) and you will never need to redo any work (you force updates on commit and will not crash during commit processing).

2.4. Granting Locks

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.

We recommend locking at *page* granularity; please do not implement table-level locking (even though it is possible) for simplicity of testing. The rest of this document and our unit tests assume page-level locking.

You will need to create data structures that keep track of which locks each transaction holds and 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 an exclusive lock.

If a transaction requests a lock that cannot be immediately granted, your code should *block*, waiting for that lock to become available (i.e., be released by another transaction running in a different thread). Be careful about race conditions in your lock implementation --- think about how concurrent invocations to your lock may affect the behavior. (you may wish to read about [Synchronization](#) in Java).

Exercise 1.

Write the methods that acquire and release locks in `BufferPool`. Assuming you are using page-level locking, you will need to complete the following:

- Modify `getPage()` to block and acquire the desired lock before returning a page.
- Implement `unsafeReleasePage()`. This method is primarily used for testing, and at the end of transactions.
- Implement `holdsLock()` so that logic in Exercise 2 can determine whether a page is already locked by a transaction.

You may find it helpful to define a `LockManager` class that is responsible for maintaining state about transactions and locks, but the design decision is up to you.

You may need to implement the next exercise before your code passes the unit tests in `LockingTest`.

2.5. Lock Lifetime

You will need to implement strict two-phase locking. This means that transactions 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.

Fortunately, the SimpleDB design is such that it is possible to obtain locks on pages in `BufferPool.getPage()` before you read or modify them. So, rather than adding calls to locking routines in each of your operators, we recommend acquiring locks in `getPage()`. Depending on your implementation, it is possible that you may not have to acquire a lock anywhere else. It is up to you to verify this!

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 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.)

Note that your implementation of `HeapFile.insertTuple()` and `HeapFile.deleteTuple()`, as well as the implementation of the iterator returned by `HeapFile.iterator()` should access pages using `BufferPool.getPage()`. Double check that these different uses of `getPage()` pass the correct permissions object (e.g., `Permissions.READ_WRITE` or `Permissions.READ_ONLY`). You may also wish to double check that your implementation of `BufferPool.insertTuple()` and `BufferPool.deleteTupe()` call `markDirty()` on any of the pages they access (you should have done this when you implemented this code in lab 2, but we did not test for this case.)

After you have acquired locks, you will need to think about when to release them as well. It is clear that you should release all locks associated with a transaction after it has committed or aborted to ensure strict 2PL. However, it is possible for there to be other scenarios in which releasing a lock before a transaction ends might be useful. For instance, you may release a shared lock on a page after scanning it to find empty slots (as described below).

Exercise 2.

Ensure that you acquire and release locks throughout SimpleDB. Some (but not necessarily all) actions that you should verify work properly:

- Reading tuples off of pages during a SeqScan (if you implemented locking in `BufferPool.getPage()`, this should work correctly as long as your `HeapFile.iterator()` uses `BufferPool.getPage()`.)
- Inserting and deleting tuples through `BufferPool` and `HeapFile` methods (if you implemented locking in `BufferPool.getPage()`, this should work correctly as long as `HeapFile.insertTuple()` and `HeapFile.deleteTuple()` use `BufferPool.getPage()`.)

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

- Adding a new page to a `HeapFile`. When do you physically write the page to disk? Are there race conditions with other transactions (on other threads) that might need special attention at the `HeapFile` level, regardless of page-level locking?
- Looking for an empty slot into which you can insert tuples. Most implementations scan pages looking for an empty slot, and will need a `READ_ONLY` lock to do this. Surprisingly, however, if a transaction t finds no free slot on a page p , t may immediately release the lock on p . Although this apparently contradicts the rules of two-phase locking, it is ok because t did not use any data from the page, such that a concurrent transaction t' which updated p cannot possibly effect the answer or outcome of t .

At this point, your code should pass the unit tests in `LockingTest`.

2.6. Implementing NO STEAL

Modifications from a transaction are written to disk only after it commits. This means we can abort a transaction by discarding the dirty pages and rereading them from disk. Thus, we must not evict dirty pages. This policy is called NO STEAL.

You will need to modify the `evictPage` method in `BufferPool`. In particular, it must never evict a dirty page. If your eviction policy prefers a dirty page for eviction, you will have to find a way to evict an alternative page. In the case where all pages in the buffer pool are dirty, you should throw a `DbException`. If your eviction policy evicts a clean page, be mindful of any locks transactions may already hold to the evicted page and handle them appropriately in your implementation.

Exercise 3.

Implement the necessary logic for page eviction without evicting dirty pages in the `evictPage` method in `BufferPool`.

2.7. Transactions

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 either *commits* or *aborts* the transaction, specified by the parameter flag `commit`. 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`.

Exercise 4.

Implement the `transactionComplete()` method in `BufferPool`. Note that there are two versions of `transactionComplete`, one which accepts an additional boolean **commit** argument, and one which does not. The version without the additional argument should always commit and so can simply be implemented by calling `transactionComplete(tid, true)`.

When you commit, you should flush dirty pages associated to the transaction to disk. When you abort, you should revert any changes made by the transaction by restoring the page to its on-disk state.

Whether the transaction commits or aborts, you should also release any state the `BufferPool` keeps regarding the transaction, including releasing any locks that the transaction held.

At this point, your code should pass the `TransactionTest` unit test and the `AbortEvictionTest` system test. You may find the `TransactionTest` system test illustrative, but it will likely fail until you complete the next exercise.

2.8. 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 & Gehrke). You will need to detect this situation and throw a `TransactionAbortedException`.

There are many possible ways to detect deadlock. A strawman example would be to implement a simple timeout policy that aborts a transaction if it has not completed after a given period of time. For a real solution, you may implement cycle-detection in a dependency graph data structure as shown in lecture. In this scheme, you would check for cycles in a dependency graph periodically or whenever you attempt to grant a new lock, and abort something if a cycle exists. After you have detected that a deadlock exists, you must decide how to improve the situation. Assume you have detected a deadlock while transaction t is waiting for a lock. If you're feeling homicidal, you might abort **all** transactions that t is waiting for; this may result in a large amount of work being undone, but you can guarantee that t will make progress. Alternately, you may decide to abort t to give other transactions a chance to make progress. This means that the end-user will have to retry transaction t .

Another approach is to use global orderings of transactions to avoid building the wait-for graph. This is sometimes preferred for performance reasons, but transactions that could have succeeded can be aborted by mistake under this scheme. Examples include the WAIT-DIE and WOUND-WAIT schemes.

Exercise 5.

Implement deadlock detection or prevention in `src/simpledb/BufferPool.java`. You have many design decisions for your deadlock handling system, but it is not necessary to do something highly sophisticated. We expect you to do better than a simple timeout on each transaction. A good starting point will be to implement cycle-detection in a wait-for graph before every lock request, and you will receive full credit for such an implementation. Please describe your choices in the lab writeup and list the pros and cons of your choice compared to the alternatives.

You should ensure that your code aborts transactions properly when a deadlock occurs, by throwing a `TransactionAbortedException` exception. This exception will be caught by the code executing the transaction (e.g., `TransactionTest.java`), which should call `transactionComplete()` to cleanup after the transaction. You are not expected to automatically restart a transaction which fails due to a deadlock -- you can assume that higher level code will take care of this.

We have provided some (not-so-unit) tests in `test/simpledb/DeadlockTest.java`. They are actually a bit involved, so they may take more than a few seconds to run (depending on your policy). If they seem to hang indefinitely, then you probably have an unresolved deadlock. These tests construct simple deadlock situations that your code should be able to escape.

Note that there are two timing parameters near the top of `DeadLockTest.java`; these determine the frequency at which the test checks if locks have been acquired and the waiting time before an aborted transaction is restarted. You may observe different performance characteristics by tweaking these parameters if you use a timeout-based detection method. The tests will output `TransactionAbortedExceptions` corresponding to resolved deadlocks to the console.

Your code should now should pass the `TransactionTest` system test (which may also run for quite a long time depending on your implementation).

At this point, you should have a recoverable database, in the sense that if the database system crashes (at a point other than `transactionComplete()`) or if the user explicitly aborts a transaction, the effects of any running transaction will not be visible after the system restarts (or the transaction aborts.) You may wish to verify this by running some transactions and explicitly killing the database server.

2.9. Design alternatives

During the course of this lab, we have identified some substantial design choices that you have to make:

- Locking granularity: page-level versus tuple-level
- Deadlock handling: detection vs. prevention, aborting yourself vs. others.

Bonus Exercise 6. (20% extra credit)

For one or more of these choices, implement both alternatives and experimentally compare their performance characteristics. Include your benchmarking code and a brief evaluation (possibly with graphs) in your writeup.

You have now completed this lab. Good work!

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 in deadlock handling, and list the pros and cons of your approach.
- Discuss and justify any changes you made to the API.
- Describe any missing or incomplete elements of your code.
- Describe how long you spent on the lab, and whether there was anything you found particularly difficult or confusing.

- Describe any extra credit implementation you have done.

3.1. Collaboration

This lab 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

Zip up your code and upload it to Canvas. Place your write-up in a file called `lab4-writeup.txt` with your submission in your main source file directory (the one that contains your `build.xml`). On Linux/MacOS, you can create your hand-in file by running the following command:

```
$ zip -r mylabgroupnum_lab4.zip cs339-lab4
```

3.3. Submitting a bug

SimpleDB is a relatively complex piece of code. It is very possible you are going to find bugs, inconsistencies, and bad, outdated, or incorrect documentation, etc.

We ask you, therefore, to do this lab 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.

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

- A description of the bug.
- A `.java` file we can drop in the `test/simplydb` 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 `HeapFileEncoder`.

You can also post on the class page on Piazza if you feel you have run into a bug.

3.4 Grading

75% of your grade will be based on whether or not your code passes the system test suite we will run over it. These tests will be a superset of the tests we have provided. Before handing in your code, you should make sure it produces no errors (passes all of the tests) from both `ant test` and `ant systemtest`.

- Given that this lab deals with concurrency, we will rerun the autograder after the due date to discourage trying buggy code until lucky. It is your responsibility to ensure that your code **reliably** passes the tests.
- This lab has a higher percentage of manual grading compared to previous labs. Specifically, we will be very unhappy if your concurrency handling is bogus (e.g., inserting `Thread.sleep(1000)` until a race disappears).

Important: before testing, we will replace your `build.xml`, `HeapFileEncoder.java` and the entire contents of the `test` directory with our version of these files. This means you cannot change the format of `.dat` files! You should also be careful changing our APIs. You should test that your code compiles the unmodified tests.

The first 75% of your grade will be based on how your hand-in performs with our unit tests. An additional 25% of your grade will be based on the quality of your writeup and our subjective evaluation of your code.

We had a lot of fun designing this assignment, and we hope you enjoy hacking on it!