

Lab 3 - SimpleDB

Database Systems, CS-UH 2214

Assigned: April 25th, 2022

Due: May 9th, 2022



Figure 1: Yippee Yay! Let's get the bonus :)

This lab is based on MIT's 6.830 - Database Systems course. The labs were initially created by Prof. Sam Madden.

1 Overview

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!

You should begin with the code you submitted for Lab 2 (if you did not submit code for Lab 2, or your solution didn't work properly, contact us to discuss options.)

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 the video lectures, 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 logbased 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, though you should be able to implement locking at *tuple* granularity if you wish (please do not implement table-level locking). 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 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 an 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 (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 pagelevel locking, you will need to complete the following:

1. Modify `getPage()` to block and acquire the desired lock before returning a page.
2. Implement `releasePage()`. This method is primarily used for testing, and at the end of transactions.
3. 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 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.READWRITE` or `Permissions.READONLY`). You may also wish to double check that your implementation of `BufferPool.insertTuple()` and `BufferPool.deleteTuple()` 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:

1. 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()`.)

2. 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:

1. 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?
2. Looking for an empty slot into which you can insert tuples. Most implementations scan pages looking for an empty slot, and will need a READONLY 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*.

Unit Tests to Pass: 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.

Note that, in general, evicting a clean page that is locked by a running transaction is OK when using NO STEAL, as long as your lock manager keeps information about evicted pages around, and as long as none of your operator implementations keep references to Page objects which have been evicted.

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.

Unit Tests to Pass: 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 and Gehrke). You will need to detect this situation and throw a TransactionAbortedException.

There are many possible ways to detect deadlock. For example, you may implement a simple timeout policy that aborts a transaction if it has not completed after a given period of time. Alternately, you may implement cycle-detection in a dependency graph data structure. In this scheme, you would check for cycles in a dependency graph 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 .

Exercise 5: Implement deadlock detection and resolution in `src/simpledb/BufferPool.java`. Most likely, you will want to check for a deadlock whenever a transaction attempts to acquire a lock and finds another transaction is holding the lock (note that this by itself is not a deadlock, but may be symptomatic of one.) You have many design decisions for your deadlock resolution system, but it is not necessary to do something complicated. Please describe your choices in the lab writeup.

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.

Unit Tests to Pass: Your code should now should pass the TransactionTest system test (which may also run for quite a long time).

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.

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 choice of page eviction policy. If you used something other than a nested-loops join, describe the tradeoffs of the algorithm you chose.
- 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.

3.1 Submission

You will package your entire project into a folder with the title: 'Lab1-netID1-netID2-netID3'. **Zip this folder** and submit via Dropbox using this link.

<https://bit.ly/CSUH2214-S22-LAB3>

Only one student per group should submit and that student should always submit/resubmit to avoid multiple submissions per group.

In each folder, please include a readme file that includes a comma separated list of each member's full name.

We will not grade any submission that does not strictly follow the submission rules.

3.2 Collaboration

This lab should be manageable for a single person, but if you prefer to work in a group of up to three students, this is also OK. Larger groups are not allowed.

If you choose to work in a group, you need to determine how best to meet and work together online. In line with the university policy, group in-person gatherings are discouraged. If you cannot agree on a suitable and safe online meeting method before starting the lab, we strongly discourage you from working in a group.

3.3 Grading

Most 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.

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 therefore be careful changing our APIs. This also means you need to test whether your code compiles with our test programs.

```
[replace build.xml, HeapFileEncoder.java, and test]
$ #our tool to replace build.xml, HeapFileEncoder.java, and test
$ ant test
$ ant systemtest
[additional tests]
```

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

An **additional significant portion** of your grade will be based on the quality of your writeup and our subjective evaluation of your code.