6.830 Lab 4 writeup

Katie Siegel

Collaborated with Ankush Gupta

1 Design Decisions

To modularize the logic associated with obtaining shared and exclusive locks, I created a new lock manager class, called DbLock. One DbLock object is kept in the BufferPool, and tracks all shared and exclusive locking for the pages read via that BufferPool. Within my separate DbLock class, I chose to heavily rely on ConcurrentHashMaps to keep track of shared and exclusive locks. ConcurrentHashMaps are designed to avoid concurrency issues with accessing and deleting methods, which was important if multiple threads were to be accessing the DbLock object at the same time. I also chose to lock on the entire DbLock object when obtaining new locks, just to prevent concurrency errors when multiple threads try to create a lock on the same page at the same time.

Within the DbLock class, I created different methods and data structures to handle shared locks and exclusive locks–I tried to separate logic out into different methods as often as possible. I kept the locks in two different ConcurrentHashMaps. The exclusive locks data structure mapped page IDs to transactionIDs, reflecting that only one transaction could have an exclusive lock on a page at a time. The shared locks data structure mapped page IDs to arrays of transactionIDs, reflecting that multiple transactions could have shared locks on a page at a time. When obtaining a new lock, I check both of these hash maps and make sure that the proper permissions can be obtained.

To resolve deadlock, I implemented a timeout method. Whenever a thread attempts to acquire a lock, the thread is given 200 milliseconds to do so, otherwise a TransactionAbortedException is thrown. In the meantime, the thread continuously attempts to acquire that lock by checking the shared locks and exclusive locks arrays over and over. I chose to implement the timeout method because it did not require me to significantly change the logic in my DbLock class. Instead, I merely had to check the System time at the beginning of the lock acquisition method, and then throw the error if the method took more than 200 milliseconds to acquire the lock.

2 Bonus Design Exercise

I implemented a dependency graph as well as the timeout method for resolving deadlocks. I implemented a dependency graph node class, LockNode, as well as the required code for using the dependency graph, on a separate branch named dependency.

Overall, my design for the dependency graph method was to create a new LockNode in the overall dependency graph for each lock that was obtained. When a thread failed to obtain a lock, I then checked the dependency graph for loops and threw an exception if a

dependency loop was found. Adding this check, compared to the other method, lead to a faster runtime for the Transaction SystemTest, as then the program did not have to wait for a timeout.

3 Changes to API

I did not make any changes to the API, and only edited two files. First, I modified the BufferPool java file to account for the new DbLock object and to call the DbLock object's acquireLock() method before a page was fetched from the pool. I also had to update my BufferPool file to handle eviction, inserting tuples, and deleting tuples properly. In the HeapFile class, I also had to change all calls to BufferPool.PAGE_SIZE to BufferPool.getPageSize(), as I had previously forgot to generalize these calls.

4 Missing or Incomplete Code

There is no missing or incomplete code in my project. However, it does not pass the Big File test. Additionally, when running the entire test suite, I realized that after the Big File test fails, most of the tests following the Big File test also fail, but when I run those tests separately, they all pass.

5 General Comments

I felt like it was unexpected that this lab did not ask us to use Java locks.