Lab 3 Write Up

In this lab, we made our database capable of handling multiple transactions at once. To ensure Conflict serializability, we implemented Strict 2PL locking for transactions. Strict 2PL is the framework where all locks for a transaction are acquired before all the unlocking and the unlocking only happens after a transaction has committed. No unlocks (either read / write) happen before a transaction commits - hence ensuring atomicity. In Strict 2PL locking, I implemented locking on the granularity of pages. Multiple transactions can have read locks on the same page. However, only a single transaction can have an exclusive lock on a page. Exclusive locks are used when a page's content has to be changed - i.e. insertion / deletion of tuples in the page.

To implement Locking in this lab, I made a lock Manager class that keeps track of read locks and exclusive locks on pages. In addition to keeping track, the lock manager is also responsible for granting or denying locks to transactions if they can't have a lock on the page based on Strict 2PL conditions. For this lab, I made it possible for a transaction to have both a read lock and an exclusive lock on a page - which means a transaction can easily upgrade locks if needed. Hence, with this addition of requiring a lock every time a page needs to be read or written to, we can ensure conflict serializability of multiple transactions running at the same time. If a transaction doesn't get the lock the first time, I make sure that thread sleeps for a small duration of time before coming back and trying to get a lock again.

In this lab we also made sure to implement the NO STEAL and FORCE structure of the database system. This means no uncommitted pages are written to disk and as soon as a transaction commits, pages modified by that transaction are immediately written to disk. To accommodate for this change, I changed my page eviction policy such that no pages that were "dirty" would be sent to disk. Only pages that weren't written to by any transaction could be sent to disk since they were not dirty. Once a transaction commits, all pages that the transaction had touched are then instantly flushed to disk one by one. This ensures NO STEAL and FORCE structure of the database system. However, even with this structure, sometimes the pages we touch are okay to be flushed to database - for example, when we are looking for the correct page that has enough space to write the tuple on, we iterate over some pages to get the right page. In that condition, if we realize the page we're looking at doesn't have space to accommodate the tuple, we can release the lock on that page since we're not using it. This gives other transactions a chance to read the page if they're waiting for it.

Since we're dealing with multiple transactions and different page locks now, we might have a condition of deadlock. Hence, in this lab, I implemented cycle detection to avoid deadlock scenarios. Every time a transaction asks for a lock, I check if giving the transaction that lock would result in a deadlock. If so, I abort the transaction that is asking for the lock. To implement cycle detection I basically keep track of which transactions are waiting for other transactions. If there is a cycle at any point in this dependency graph, this means there is going to be a deadlock, hence the abort is needed to let other transactions go ahead. With commits or aborts, as previously described, all dirty pages are either forgotten and replaced with disk versions of the page(in case of abort) or flushed to disk (in case of commit). In either case, all pages held by that transaction are released.

Another important design decision to this lab is that every buffer pool has an individual lock manager instance. Different instances of buffer pool will have different lock manager instances. A single lock manager instance cannot be shared between multiple lock manager instances. This also makes sense since each database system has one buffer pool that can keep track of pages in memory. In the case of multiple buffer pools for a database system, a separate class can be created that holds multiple instances of lock managers and their ids. Hence, multiple buffer pools in a single database system can still reference the same lock manager using the id of the lock manager. In addition, making the lock manager buffer pool instance specific helps in resetting the database system as well since resetting the buffer pool also resets the lock manager and the database system starts anew.

The overall flow of the database system is as follows : A transaction asks the BufferPool for a specific page using the getPage() method in the Buffer Pool. The Buffer Pool in turn then asks the lock manager to give the transaction the lock the transaction needs on the page (based on the permissions passed). Before giving a lock to the transaction, the lock manager checks if the transaction can indeed get the lock and if so gives the lock to the transaction. If not, the lock manager

checks if giving the lock to the transaction could lead to a deadlock situation. If yes, the transaction is asked to be aborted, otherwise the transaction is asked to wait till the other lock is released. When a transaction finally commits, all pages modified by the transaction in the buffer pool are immediately pushed to disk. If it aborts, all pages modified by the transaction in the buffer pool are re loaded from the disk to get the original versions in the disk. In both cases, all page locks are released.

Hence, this is the entire structure of the database system (Simple DB) after Lab 3. It allows multiple transactions to operate at the same time and ensures conflict serializability.

Unit Tests that could be added:
Another unit test could be added would be to check if when a transaction commits, all pages in the buffer pool are indeed written to the disk and the buffer pool now contains the original page that was stored in the disk. This could also be checked for aborts, if after aborting a page in the buffer pool returns to the state of the page on the disk.

Design Decisions - I implemented a graph of transaction -> transactions waiting on this transaction. Essentially if T1 was waiting for T3, my graph would have T3 -> T1. This helped me identify deadlocks by just checking if the transaction that i was going to add would create a cyclic dependency (deadlock situation). Another design decision I took was to lock on the granularity of pages. In addition, I also changed my eviction policy such that I go over all pages all evict the first page that is not dirty. If none of the pages in the buffer pool are "clean", I throw a DbException stating nothing can be evicted right now.

API Changes - None.

Feedback - "Really intense lab, especially in terms of debugging. I spent an enormous amount of time just trying to figure out why the transaction test wasn't working always. But overall, I had a lot of fun with this lab. Shout out to the TAs for all their help. They're great!"