Lab 4 Write Up

In this lab we implemented a STEAL and NO-FORCE policy for our database system. This is in contrast to the NO STEAL and FORCE policy we implemented in Lab 3. To actually go about making the database support this policy as well as make the database recoverable in the case of a crash - we implemented logging in this lab. The logging we implemented is very similar to the ARIES protocol. The first change we implemented was to create a log entry for each change we make to the database - update, commit, aborts,begins or checkpoints. Since we're implementing a NO STEAL policy, this means the pages in memory can be flushed at any time, hence the only change we make before we commit is to force the log to disk after making the entries as needed. Hence, whenever an update needs to be made - insert / delete we make a log entry of it and store it with that transaction and occasionally keep forcing some of the log records to disk (when a transaction commits/ aborts/ at a checkpoint).

If a transaction aborts, we need to roll back all of it's changes to the way the pages were before the transaction had started. We do this by undoing all the updates by the transactions to the versions of the pages before the update executed. As a design decision, I added additional CLR records to the end of the log (before the ABORT log) that stated the change I had undone and what the previous and after versions of the page were then. Following this, when all the CLR records for the updates by the transaction have been written, I write an UNDO record and the log is flushed to disk.

In a similar format, during recovery, we go about finding all the transactions that hadn't been committed or aborted since the last transaction and realize that for these transactions all their updates needed to be aborted. During this phase, CLR records are treated in the same manner that Update Log records are treated and everything is redone since the last checkpoint. Following this, updates for the "loser" transactions are undone and CLR records for those are written as well. Once this is done, the database system can go about dealing with the next updates, etc as it would have if the system hadn't crashed (except that now some transactions have already been aborted so they need to start again).

I think the toughest part of the lab was understanding the code of LogFile and interpreting how to read the file and checking for the different possible format types. In addition to that, debugging in this lab was much more complex than lab 3 for me. This was primarily because I had multiple things working together for the log file and it was impossible almost to figure out why things were failing. I was mostly trying to debug using println statements so debugging and figuring out what the log file holds at each step was a lot to take in.

I had to change some of my implementation in other places in order to get the STEAL and NO FORCE policy to work correctly. One change I made was to my BufferPool.transactionComplete(commit = true). Initially I was just surfing through the dirty pages in my BufferPool and checking which of the pages were dirty and then checking the transaction that had set them as dirty to finally add the statements that set the before image of the page. However since in this STEAL policy, pages are flushed without any indication and while flushing we set their dirty value as false, I could no longer validate if a transaction had changed that page using the implementation I had before. Hence, I used the lockManager to figure out which pages had been modified by the transaction I'm looking at and used that instead to figure out which pages needed to have their before images re written. In addition, I also had to add another method in the LockManager class that releases all locks held by a certain transaction. Prior to this, to remove a transaction from the lock manager (once the transaction had committed or aborted), I would need the pageId of the page I wanted the transaction to be removed from. However, to support allowing eviction of any pages and removal of locks without having the pageID, I needed to implement this additional method that just takes in a transactionID and removes lock on any pages it might have a lock on. In addition, I also changed my eviction policy in evict page to a random eviction policy in which it randomly picks a page from the buffer pool and evicts it (whether it was dirty or not).

Unit Test that can be added - A Unit Test that can be added is to check what happens if the system crashes while rolling back or during recovery. This means there would be two crashes - one that simulated recovery and one that happens while the system is still recovering. This would help ensure if the system is indeed full proof and always recoverable.

Feedback - This was indeed a very interesting lab although really heavy on debugging but really interesting and fun to implement.