Description:
Lab 3 was about implementing the parts that handle SimpleDB transactions. Transaction queries should execute atomically and be serializable. In this lab I implemented a LockManager that handles the locking and unlocking of pages, following Strict two-phase locking. In this lab I had to make some adjustments to the BufferPool's eviction policy to follow a NO STEAL policy. SimpleDB uses a NO STEAL policy for recoverability, in the event a transaction aborts or does a ROLLBACk we need to revert back any changes a transaction may have made to pages. When implementing the locking, I chose to do page granularity locking, that is a transaction will hold a lock to the whole page when doing a read or write. My LockManager handled the logic for detecting deadlocks, aborting the current transaction trying to acquire a lock. Finally, I implemented the remaining methods in BufferPool that would handle transaction completion (either through a commit or abort).

Implementation:
LockManager:

- The LockManager has a subclass Lock. The Lock class holds information on the type of lock it is (Unused, Shared, Exclusive) along with information on the transactions that currently hold that lock.
- The LockManager holds several mappings:
    - a mapping from pageId to a Lock (each page will have a single lock that transactions will try to acquire)
    - a mapping from transactionId to a set of pages that transaction holds locks to
    - a mapping from transactionId to a set of other transactions that transaction is currently waiting on
- A call to BufferPool's getPage() will first try to acquire a lock for that page, waiting if it is unable to. Upon a call to LockManager's acquireLock(), the lock manager will check conditions for the permission the current transaction is trying to acquire a lock for.
    - If the transaction is trying to do a READ_ONLY on a shared lock, then it can gain access, updating necessary data structures.
    - If the transaction is trying to do a READ_WRITE on a shared lock, it must first wait until all other transactions have relinquished the lock to that page.
    - If no other transaction holds a lock to a page than the transaction can freely gain access to the lock or upgrade a lock to an exclusive lock.
- Any time a transaction is unable to acquire a lock it will first check for any deadlocks, aborting if it detects a deadlock or waiting otherwise.
    - I used a dependency graph when doing deadlock detection. It starts by first looking at all the transactions holding a lock to the page. For each of

those transactions it looks at all the other transactions it is currently waiting on. If any of those transactions is waiting for the current transaction, it means we have encountered a deadlock, aborting the current transaction.

- Following strict two-phase locking, a transaction does not relinquish any of its locks until it has committed or aborted. Therefore we only make a call to BufferPool's releasePages() (where we release page locks) in our call to transactionComplete().
- As mentioned, I had to make adjustments to BufferPool's eviction policy so that it works with a NO STEAL policy (we do not evict dirty pages).
- BufferPool's transactionComplete() will either flush all pages to disk if the commit flag is true or revert back to the pages' old data if false. I used HeapPage's getBeforeImage() to revert back to the old data.

Unit Test:
We can add a unit test that tests for efficient memory usage. The unit test will test to make sure we no longer have reference to a transaction once it has committed or aborted. Likewise, the test would check to make sure we no longer have reference to a page that no current transaction holds a lock to and/or also not present in the BufferPool. I feel this could be a useful thing to include in our implementation since in a real database it is likely that transactions will touch many different pages.