Andrew Dawson
1220796

In this lab we provided support for transactions. In order to provide this support we had to implement locking. The high level idea is transactions have to acquire locks on pages before they can operate on them.

In this lab we implemented strict two phase locking that was both force and no steal. Strict two phase locking requires that a transaction acquires all locks before releasing any locks and that all locks are released at time of commit or abort. Force means that after a transaction commits it should force all dirtied page to disk. No steal means that if a page is dirtied by a transaction and the transaction is still running that page cannot be flushed to disk. No steal and force are the easiest to implement because they do not require us to do any clean up after an abort. In this lab we locked at the level of pages. It is possible to lock at many levels: database, table, page or tuple. It is also possible to have hierarchical locks.

In order for a transaction to read or write a page from the database it must hold the correct type of lock on that page. There are two types of locks shared locks and exclusive locks. Shared locks are required for reading a page and exclusive locks are required for writing a page. Many transaction can read a page at the same time, however, only one transaction can write at a time. Therefore the high level logic of the locking system is - check if a transaction has the needed lock, if not attempt to acquire a the correct lock type (shared or exclusive) if the lock is granted continue otherwise abort yourself.

Transactions can deadlock. Deadlock is simply when a cycle exists in the transaction dependency graph. Two approaches to resolve deadlock are cycle detection and timeouts. In cycle detection a graph is built - representing which transactions depend on which other transactions, a deadlock exists if this graph has a cycle in it. Timeouts abort transactions if they have not acquired the requested lock within some amount of time. Timeouts are imperfect for two reasons. Firstly, transactions that run for a long time can cause other transactions to abort when they did not have to. Secondly, timeouts are non deterministic; it is possible that after aborting a transaction it could just deadlock again. So in practice timeouts work but in theory transactions could deadlock indefinitely.

In this lab there were several design decisions relating to - evicting pages, dealing with deadlock and lock handling.

Using no steal required updating the eviction policy. No steal constitutes that dirty pages in the Buffer Pool do not get evicted. Previously the oldest page got evicted from the BufferPool when it was full. However according to no steal, if the oldest page is dirty it cannot be evicted. Therefore evict page now evicts the oldest page that is clean. This modification still incorporates the benefits of a LRU cache while still abiding by no steal requirements.

Timeouts are used to handle deadlock. The most difficult part of getting timeouts right was determining how long threads should wait before aborting themselves. There are two factors that dictate how threads will wait - timeout and wait time. Timeout is how long threads wait before aborting themselves while wait time is how long a thread goes to sleep before checking its total wait time against the timeout. The timeout basically determines how long a thread should try to acquire locks for before giving up. This number was set using trial and error. Additionally the timeout is randomized so that transactions waiting on locks do not all abort themselves at the same time. In the end the timeout was set to be a random value between 2 and 7 seconds. Having staggered timeouts resolves deadlock more effectively. The wait time is simply how long a thread waits before checking its total wait time again. When a thread waits inside a synchronized method is gives up the implicit lock and allows other threads to make progress. This is necessary because it will allow other threads to release locks - thereby freeing up the current thread. When using wait it is also important to use notifyAll. The notifyAll method will wake up all waiting threads. Therefore any time a lock is released notifyAll is called because that release could have freed some thread. The combination of wait time, timeouts and using notifyAll allows timeouts to resolve deadlock quickly.

A LockManager class was created because the locking logic was too complex to keep in the BufferPool. The LockManager class kept track of which transactions hold which locks. In order to do this I use three maps that keep track of lock types and the set of locks that each transaction holds. Since different threads access these maps concurrently race conditions become a problem. In order to defend against race conditions all methods in LockManager are made to be synchronized.

One unit test that could have been added was creating a large BufferPool filling it up with dirty pages and seeing that a DbException was thrown upon the BufferPool filling up. This would increase the testing coverage for page eviction.

I did not make any changes to the API. However, I did have to decide the API of LockManager. One of the key design decisions that I made in LockManager was that LockManager.acquireLock should return a boolean - true if the lock was acquired false otherwise. Returning a boolean makes checking if the lock was acquired very easy in BufferPool.