Part 1:

EX 1 and 5: Ivan

Exercise 1:

- Prior to adapting getPage() to work with locking and implementing releasePage() and holdsLock(), we first created a nested LockManager class within the BufferPool class. The reason for the LockManager class is to help keep track of multiple transactions, what type of locks they need, for what pages do they need these locks, and whether or not the transaction in question is able to receive the lock. For the LockManger class, we created three HashMap data structures to keep track of three things: what transactions have exclusive locks and for what pages, what transactions have shared locks and for what pages, and what transactions are waiting for an exclusive lock from another transaction. The LockManager class allows transactions to acquire and release shared and exclusive locks at the page level, waiting for an exclusive lock from another transaction if need be.
- After implementing the LockManager class, we adapted the getPage() method in BufferPool so that if a transaction is requesting a page, based off the passed in permission, the transaction must first acquire a shared or exclusive lock on the page before acquiring the page, blocking if the transactions has to wait for the lock to be released by another transaction. Additionally, we implemented releasePage() and holdsLock(). The releasePage() method removes all shared and exclusive locks associated between the passed in page and transaction. The holdLock() method simply returns a boolean value indicating whether or not a transaction holds an exclusive or shared lock on the page.

Exercise 2:

- Implementing two-phase locking in which transactions only acquire the appropriate type of lock on any Page before accessing that Page and should not release any locks until after the transaction commits. By implementing locking at page level, operator and HeapFile only interact with Page through BufferPool by acquiring shared lock or exclusive lock. Locks will be released when a transaction commit or abort in exercise 4.

Exercise 3:

- To implement NO STEAL, we avoid writing any pages modified by uncommitted transactions to disk. This means that dirty pages remain in the BufferPool until the associated transaction either commits or aborts. To manage the eviction of pages from the BufferPool, we have implemented a Deque data structure for LRU policy and checks whether a page is dirty before evicting it. Only clean pages are considered for eviction.

Exercise 4:

- After a query is executed, the BufferPool's transactionComplete() method is invoked. This method either flushes all the dirty pages associated with the completed transaction to disk when the transaction is committed or restores the disk state of all dirty pages associated with that transaction in case of an abort. To restore the disk state of dirty pages, the Catalog is used to access the readPage() method on disk, which retrieves the page and stores it in the BufferPool.

Exercise 5:

- For this part of the lab, we implemented a method that would allow us to detect deadlocks when a transaction requests an exclusive lock on a given page that is already held by another transaction. For detecting deadlocks, rather than

implementing a timeout policy, we created a method that would check for cycles within the waitlist/dependency graph. A cycle in the dependency graph would be present if a transaction that currently has an exclusive lock on a page, which we can call T1, is waiting for an exclusive lock from a transaction, which we can call T2, that is waiting for an exclusive lock from T1. Therefore, T1 would be waiting for T2 to release an exclusive lock and T2 would be waiting for T1 to release an exclusive lock, creating a deadlock and disallowing both transactions from making progress. If a deadlock is not detected from the lock request, then the transaction will be placed in the waitlist to receive the exclusive lock from the transaction that currently holds it.

- When a potential deadlock is detected from a lock request, we resolve the deadlock by throwing a TransactionAbortedException() and aborting the transaction that requested the lock that would cause the potential deadlock, allowing the other transactions to continue and forcing the aborted transaction to restart after cleanup from transactionComplete().

Part 2: **Unit Test**

- In this lab, we implemented NO STEAL, I think a unit test for a case where BufferPool is full of dirty pages would be great. In this case, we just throw DbException().

Part 3:

Ivan

For Lab 3, the primary design decisions we had to make was how we will detect deadlocks, how we will resolve them, and at what level we will be locking at. Firstly, we decided to detect potential deadlocks from lock requests through cycle detection in a dependency graph, rather than a timeout policy. We decided against using a timeout policy due to the possibility of transactions being wrongfully aborted even though there was no deadlock due to the

transaction having to wait for a lock from another transaction that is performing a lot of operations on the page and taking up a lot of time, which may take up enough time for the timeout policy to just abort the waiting transaction. Therefore, we decided to use cycle detection to avoid the possibility of transactions being wrongfully aborted due to abnormally long wait times for a lock on a page. Secondly, we decided to resolve potential deadlocks by aborting the transaction that requested a lock for a page that would cause a deadlock. We decided to just abort the single transaction that would cause the deadlock, forcing it to restart after cleanup, rather than aborting all other transactions. Our design choice behind this was simply that it would be better to abort and restart 1 transaction and let 20 other commit, rather than having 1 transaction commit and 20 others abort and restart, causing a lot more cleanup to be needed, which takes up more time and is less efficient, and causing a lot of work to be undone. Lastly, we decided to implement locking granularity at the page level because the tests for lab 3 assume this and locking at the tuple level may be slower due to many more locks having to be tracked at once and locking at the table level wouldn't allow for very much concurrency. Therefore, we decided that locking granularity at the page level would allow for the most optimal concurrency and memory usage, optimal memory usage since fewer locks will have to be tracked in memory for locking at the page level compared to the tuple level.