- **What the lab is about**

In this lab I learned a lot about how to handle the concurrency in Strict 2PL with NO STEAL/FORCE rules. We used primitive locking method to reduce the problem of conflict like read write, write read, and write write conflict. In the first half of the lab, I implemented the lock manager class, which is a structure for organizing all the locks. It has 4 maps to keep track of the read and write locks on a page and read write locks a transaction holds. A transaction could hold as many read and write locks as it would like. However, for the lock on pages, we could either have any number of read locks on it or exactly one write lock on it but not both except the case that the only read lock on the page and the only write lock on the page is actually the same transaction. In this way, the locker manager could prevents all the conflict while offering shared locks for read. Based on this rule I have implement the lock method to grant locks to user's transaction on given page depends on the type of access specified by permission. It will return true and record the lock relation on the maps when lock is successfully granted. If fail to grant the lock, it will return false and put the waiting relation on the waitingInfo map to facilitate deadlock detection. I have implemented the release method as well, one is to release lock for releasing certain transaction's lock on certain page and the other is to release all the locks on a transaction. Another method isTransaction lock is used for checking if the pageId is After creating the lock manager, I have made the change to the getPage method in the BufferPool and so it could acquire lock before getting the page. If not, I will let the thread sleep to wait for other transaction to release lock. I have also implement the releaselock() and holdslock() to make sure the buffer pool's method is connect with locker manager's corresponding method.

For second half of the lab, I have made the Database to be NO STEAL by prevent evicting any dirty pages in the evictPage method before commit and if all pages are dirty in the buffer pool I will throw DbException. In order to prevent it to evict any dirty pages, I have implement to skip dirty page in the flushPage(PageId pid) method (the method to flush a single page) that is called by evictPage method. For transactionComplete method, if the transaction is commited, I will call flushPages(Transaction tid) to flush all dirty pages of a given transaction to satisfy FORCE . If the transaction is not commited (abort) then I will revert transaction affected page to their before image. For either way I will release the lock while transaction complete to ensure the strict 2PL. I have also implemented the isDeadLockOccurred, the dead lock detection method, it will use the helper method isWaitingResponse to recursively found directly and indirectly waiting relationship and check if the lock request of a transaction on a page will cause dead lock or not. The Buffer Pool's getPage method will call this method before aquiring lock to prevent the deadlock by throwing TransactionAbortException()

- **Unit Test to Add**

  For the Transaction test of multiple threads, the test run for all the thread is testing a workflow of reading tuple, deleting tuple and insert the tuple. It doesn't test for other work flow such as deleting, reading, inserting. And it only tests for a single commit for all transactions. Thus I would recommand to add test for different work flow and multiple commits and maybe involve more tuples rather than a single tuple.

- **Design Desicions**

  I adopted a coarser grain locking system and locked at the page level granularity because it has less overhead in managing lock. It cost less space and time to store the lock state. If we use fine granularity like have a lock status for each tuple then we have to keep track of all tuples that make the storaging of lock status expensive. Also, it is easier to implement. For deadlocks policies, I have adopted the dependency graph detection rather than timeout detection. Because the time out interval is not easily detected and it depends on different database. Also a long transaction might not caused by DeadLocks and we will thus false kill other transaction that takes a long time such as it has a large number of tuples to read. The dependency graph detection is more precise that it will only abort when it actually detects the cycle. However, I haven't written a new class such as DependencyGraph class because creating new object is costly in time complexity and actually a simple waitInfo map could just do as good to store the direct waiting information to detect deadlocks. I have choosed to abort the transaction that will cause deadlock rather than abort other transactions. Because we might have a large system of waiting dependency map, but just adding this single waiting relationship would cause a deadlock. It would be expensive and produce a lot of overhead to abort all the other transactions, thus I would rather choose to prevent the one that adding it would cause a deadlock by abort it. For LockManager, I used the four hashmaps including two hashmaps for read and write with pageId as key and transactions that holds lock on the page as value and two hashmaps for read and write with transaction as key and the locks it holds as value. Adopt this system could let us make use of hashmap to quickly get locking relationship.

- **Other declaration**

  I don't make any change to API and don't have any feedbacks.