CSE 444: Lab 3 Write-up

1. **In your own words, describe what this lab was about: Describe the various components that you implemented and how they work. This part needs to demonstrate your understanding of the lab! If your description looks like a copy paste of the instructions or a copy-paste of the provided documentation you will lose points.**

   Handle Multiple Transactions
   In this lab, we ensure that Lab 3 is able to handle multiple transactions at once while being able to guarantee that the transactions are conflict serializable, and ensure that it adheres to the ACID properties (Atomicity, Consistency, Isolation, Durability) by following the Strict 2PL protocol locking. The locking protocol works such that we acquire a lock at the beginning of the transaction and no lock will be released until all necessary locks are acquired. The locks are only released when the transaction commits. There are however special scenarios like when looking for an empty slot to insert tuples, the Strict 2PL rule is violated. In the case that there are no free slots on a page, the locks will also be immediately released, despite not acquiring all the necessary locks yet (first-phase).

   NO-STEAL & FORCE
   We also implement a “NO-STEAL” and “FORCE” policy for the Lab. In this scenario. We implemented “NO-STEAL” by ensuring dirty pages are not evicted from the buffer pool if their locks are held by an uncommitted transaction. Pages could only be “dirty” when they are locked by a transaction because, in our implementation, we always flush dirty pages after a transaction commits and release locks on that page. We implemented “FORCE” by ensuring that we write pages to disk on a transaction commit.

   Lock Granularity
   We implemented lock granularity on shared and exclusive locks. In our implementation, multiple pages could have a shared lock on a specific page. These shared locks only allow for reading data. For exclusive locks, on the other hand, only one page is allowed to acquire an exclusive lock on a specific page at a single time because these locks allow writing. If a transaction requests an exclusive lock and the transaction is the only transaction already holding a shared lock on a page, it may upgrade its lock to an exclusive lock. Thus, relevant locks are acquired before a transaction wants to access content on a page.

   Lock Manager
   We implemented a LockManager.java to manage locking operations. It keeps track of the locks and checks to see if a lock should be granted permission to a transaction or not. BufferPool then calls the LockManager when a transaction tries to get a page using getPage(). Before a transaction gets a page, they would need to acquire a lock first, either a shared or exclusive lock depending on the transaction. If a transaction is denied a lock, the transaction will have to wait and check if it's a deadlock. If there is a deadlock, a TransactionAbortedException will be thrown. We detect and managed deadlocks in LockManager.java. When a transaction is complete, transactionComplete() in BufferPool is called and a passed boolean value will determine whether we commit or abort. Aborting will discard all pages associated with the transaction because those pages could be “dirty”. Committing a transaction will cause all associated pages to be flushed or written to disk. In both cases, the transaction will release the lock on the specific pages.

2. **Describe a unit test that could be added to improve the set of unit tests that we provided for this lab.**
   One unit test that could be added is one designed to test for evictPage’s NO STEAL policy because we found a minor bug which went undetected for a long time involving the random number generator that determines the page to be evicted. The test could focus on the boundary condition checks of the random

number generator, ensuring that the generator is producing numbers within a defined range to prevent any potential out-of-bound eviction. It remained undetected until the TransactionTest system test where testAllDirtyFails failed. If this unit test had been added, it would help decrease the amount of time spent debugging as we initially thought the bug was rooted from our LockManager class since all the other tests contained in TransactionTest involve locking and deadlock situations.

3. **Describe any design decisions you made, including your deadlock detection policy, locking granularity, etc.**

   Our LockManager class consists of three ConcurrentHashMaps. The first ConcurrentHashMap handles exclusive locks, mapping PageId to the TransactionId that holds an exclusive lock on that page. The second ConcurrentHashMap handles shared locks, mapping PageId to a set of TransactionIds that have shared locks on that page. Finally, the third ConcurrentHashMap is a lock tracker which will handle the dependency of each lock.

   To achieve page-level granularity, transactions can only acquire locks on pages under certain conditions. If there are no locks on a page, a transaction can acquire a lock on it. Additionally, a transaction can acquire a shared lock if the page is already locked by multiple transactions, or it can acquire an exclusive lock if no other transaction has locked the page. In the case where a transaction holds a shared lock on a page and wants to acquire an exclusive lock, it can do so only if it is the only transaction holding a shared lock on that page. Otherwise, the transaction must wait until the necessary locks are released.

   For deadlock detection, we implemented a lock tracker to handle the dependency of acquired locks using a ConcurrentHashMap. The graph maps TransactionId to a Set of TransactionIds, representing directed edges for dependencies. Deadlocks are detected by performing a DFS on the graph to find cycles. If a cycle exists in the dependency graph, it indicates that a transaction is blocked by another transaction that it depends on for a lock release. However, the blocking transaction is also blocked because it depends on the first transaction to release a lock, resulting in a deadlock where neither transaction can progress. In such cases, the deadlock is resolved by having the transaction abort itself.

   Whenever a transaction attempts to acquire a lock that is not immediately available, it is added to the dependency graph. The transaction being added is the key, while the transaction holding the required lock is the value. The added transaction checks if it forms a cycle in the graph. If a cycle exists, the transaction is considered deadlocked and it aborts itself to resolve the issue. Otherwise, the transaction continues to wait until the desired lock is released and then attempts to acquire it again.

   In evictPage(), we made a slight modification to only evict non-dirty pages. If all pages in the buffer pool were dirty, a DBException would be thrown. In transactionComplete(), the behavior depends on a boolean parameter. If the parameter is true, the transaction commits by flushing its pages to disk. If the parameter is false, the transaction aborts by discarding the dirty pages it used.

4. **Discuss and justify any changes you made to the API.**
   We did not make any changes to the API.

5. **If you have any feedback for us on the assignment, you can add it to the writeup, send us an email, or make a note and paste your note in the course evaluation form at the end of the quarter.**
   We did not make any changes to the API.