Khai Tran

SID: 1728872

Writeup Lab3

The lab3 that we will work in advance in Transaction and Lock, Thread, and Deadlock. We learn how to manage transaction that satisfies the ACID properties. A transaction, we care about Strict two-phase locking isolation, consistent by virtue of atomicity, A FORCE buffer management policy ensures durability.

In recovery and buffer management, we use the technic which is implement a NO STEAL/FORCE buffer management policy. We know that a transaction T1 reads a data from page, but Bufferpool is full. Therefore, we need to clear some memory for T1 by sending some other pages in working memory to stable storage which could be dangerous because we are not sure that the pages which we want to push to stable storage has been committed or not which is stealing.

Forcing is every time a transaction commits, all the modify pages will be pushed to stable storage which is not good because each page may be modify by many transactions and will lead to slow the system down. Therefore, this lab has many issues that we must handle.

To implement Abort Eviction, we use No Steal which we do not evict the dirty pages in the evictPage method in BufferPool to make SimpleDB will not crash while processing a transactionComplete command by force updates on commit. We do not flush page to disk and do not require us to do any clean up after an abort. In this lab we focus to the level of pages, instead of lock on levels of database, table, page or tuple which is possible to have hierarchical locks.

We implement a new LockManager class that will hold these data structures and will manage locking and unlocking operations which get the data (TransactionId, PageId, Permissions) from getPage of BufferPool. In LockManager, we use ConcurrentHashMap<PageId, Permissions>, Map<TransactionId, Set<PageId>>, Map<PageId, Set<TransactionId>>, and ConcurrentHashMap<TransactionId, PageId> to keep tracking the pages with transactions in them. We want to use HashSet because we want to eliminate duplicate data which cause DeadLock. Because different threads access these maps can have conflicts, and to defend against race conditions, we implement all methods in LockManager with synchronized.

A transaction to read or write a page from the database, and it have to hold the correct type of lock on page which are shared locks (for reading) and exclusive locks (for writing). There are many transactions can read a page at a time, but only one transaction can write on a page at a time. The necessary to acquire the correct locks type and granted continue or abort itself.

To handle deadlock with transaction, Deadlock is simply when a cycle exists in the transaction dependency graph. There are two ways to resolve deadlock which is cycle detection and timeouts. In cycle detection a graph can be built by representing which transactions depend on other transactions, a deadlock exists if this graph has a cycle in it where duplicate may appear. Timeouts abort transactions when do not have the requested lock within an amount of time. There are two disadvantages about Timeouts. First, transactions can cause long time running and other transactions can abort unnecessary. Secondly, timeouts are not predictable and possible aborted transaction which leads to deadlock again.

Therefore, implement Timeouts could work for a few threads, but transactions could deadlock indefinitely when the system become complexity.

This lab3, I used dependency graphs because it run faster and help manage multi threads better than timeouts and it will be easy to fix deadlock than timeouts.

I think all unit tests are work well.

To detect Deadlock by dependency graphs. The most important fact of cycle detection is how we handle the thread in two maps ConcurrentHashMap<TransactionId, PageId> and HashMap<PageId, Set<TransactionId>> because when we create a lock in Lockmanger.java, we need to make sure all transaction from both two maps are consistency and no duplicate by using Set< TransactionId >. There are recursive methods to detect duplicate TransactionId in the same page which means that it will throw TransactionAbortedException if we want to add the same TransactionId in the HashMap<PageId, Set<TransactionId>> which lead to deadlock. To handle multi threads, this lab3 used synchronized () method from java, and it work very well for multi threads.

I did not change the API, but I have added a new LockManager.java class to handle Lock from BufferPool. BufferPool call lock.putkey(tid, pid, perm) in while() loop to find at least a lock from LockManager.java return true if there is a lock and otherwise. LockManager.java also give a set of PageId to BufferPool transactionComplete() to commit or abort the pages with locks and release the locks from pages.

Design alternatives I do both timeouts and dependency graphs. Timeouts is very slow compare to dependency graphs because when I had developed timeouts, I just passed one thread and I could not pass more than two threads when I adjust the time out in the loop, and it goes to infinity running. I thought if I fix it to pass 10 threads, but it may conflict and infinity running if more than 10 threads in the future. I feel more comfortable with dependency graphs because it runs the test very fast because I used control thread by synchronized () in java. Therefore, I chose to implement dependency graphs.