Lab3

This lab is asking us to implement 2 phase commit with FORSE and NO-STEAL. In order to do that, we need to implement locking logic(as recommend in README, I created a helper class: LockManager) to process all the transaction lock acquirement. We need to make sure the database is recoverable.

The important part for my lab3:

1. LockManager data structure design. For me design, I used 4 HashMap in my lock manager:

   waitlist<tid, pid>: Record all the waiting request for a specific tid → pid

   tidToPages<tid, Set<pid>>: For a specific tid, record all the pages that been locked by this tid
   pageToTids<pid, Set<tid>>: For a specific pid, record all the transactions that been grant a lock with this page
   pageLockMap<pid, perm>: Record all the page with its lock type

   Not sure if there's a optimization of the data structure design, those 4 maps are all essential as I implementing my lockManager.

2. We need to be very familiar with the 2PL and the logic of locking granularity. Especially, we need to be very clear that when we are given (**pid, tid, perm**) whether should we grant the lock or not. Specifically, we divide the situation into 5 scenario:

   1) page has not been locked yet. Directly grant the lock

   2) page has been locked with SHARED lock:

      a) The acquire lock is also a SHARED lock, directly grant the lock

      b) The acquire lock is a EXCLUSIVE lock, grant only when this ***tid*** is the only holder for this SHARED lock & upgrade the lock type; otherwise refuse the acquirement

   2) page has been locked with EXCLUSIVE lock:

      a) The acquire lock is a SHARED lock, grant only when this tid holds this EXCLUSIVE lock

      b) The acquire lock is a EXCLUSIVE lock, grant only when this tid holds this EXCLUSIVE lock

3. We need to be familiar with basic Java multi-thread control. Specifically, we need to have all lockManager method synchronized since the lock processing should ensure atomicity. We need to have the thread wait() while the lockManager is not able to grant the lock; Need to notifyAll() thread when (1) a transaction complete or release a lock (2) a lock is successfully granted.

4. We need to carefully deal with FORCE and NO-STEAL policy.

   When a transaction complete with COMMIT. We should flush the page onto disk

   When a transaction complete with ABORT. We simply read the old data of this page. Overwrite the current page with the old data page in buffer pool.

Locking granularity: For this lab, I choose to use page level locking. Specifically, when a transaction acquire a lock on page pid. We update the information in lock manager about this page and transaction.

Deadlock design: For this lab, I’m using graph dependencies to detect dead lock. The general approach is that using BFS to search through the graph. When a ***tid*** tries to acquire a ***perm*** lock on ***pid***, we start deadlock detect:

> If this page has already been locked by tid with perm, return the deadlock detect(No need to detect)
>
> Then put the pid into a queue, while queue is not empty, do the following operation:
> Poll the page on top of the queue as curPid. Get all transactions that acquired a lock on this curPid. For each transaction, we compare it with tid: If equal, then we find a dead lock; otherwise, go to waitlist map and get the page that is acquiring by this transaction and put into queue.
>
> One thing to notice during this process, we’ll need to skip the deadlock if we found it in the first BFS search. Because pid could been locked by tid already with a SHARED lock with other transaction.

The deadlock resolving method I’m using is by abort the current transaction(The transaction that used as input parameter of detectDeadlock()). When a deadlock is founded in detectDeadlock() method, throw a new exception to abort the current transaction.

The unit test we could add. I personally think we could add a test to see if our **HeapFile.write()** function is actually working. ex. write the new INSERT/DELETE onto disk successfully. The HeapFileWrite test is only testing the functionality of whether we can append empty page on disk or not. But not test whether we've successfully write the modification to disk or not. I find my code with a bug(Not updating **header** in HeapPage, so always getting an empty page data with **HeapPage.getPageData()** method) but is able to pass HeapFileWrite test.