6.830 Lab 3 Writeup

Radhika Malik
Lab Partner: Somani Patnaik

Design Decisions

- Locking granularity: The locking granularity we use is page-level locking; each `PageId` has an associated lock (locking mechanisms are further explained in the next section).

- Deadlock detection: We use timeouts for deadlock detection. In our code, if a transaction in a thread cannot get a lock it is requesting, the thread sleeps for 1ms and then tries again. We keep a counter that keeps track of the number of times the thread tries to get a lock. If the counter exceeds 200, a deadlock is detected.

- Deadlock resolution: For deadlock resolution, the transaction aborts itself after the timeout. Thus, if the maintained counter exceeds 200, the transaction thread throws a `TransactionAbortedException`.

- `BufferPool` eviction: We modify our LRU replacement policy from lab2 to not evict dirty pages (for a NO STEAL policy). Instead of evicting the least recently used page, we evict the least recently used non-dirty page. If all pages in the `BufferPool` are dirty, a `DbException` is thrown.

- Transaction Abort/Commit- When a transaction commits, all its dirty pages are flushed to disk and all its locks are released. When a transaction aborts, all its locks are released and all its dirty pages are removed from the `BufferPool`. Subsequently if the pages removed by an aborted transaction are needed, they will be re-read from disk, restoring them to their on-disk state.

Changes to the API

There were no changes to the API. However, we added two classes to do transaction locking; these classes are called `PageLock` and `LockTracker`, and are described as follows-

- `PageLock`: It is the locking structure used to lock pages. Each `PageId` has a unique `PageLock` associated with it.
  - A `PageLock` has a type (null/Shared/Exclusive). If type is `null` it signifies that the lock is not held by any transaction, a type `Exclusive` indicates that it is held by 1 transaction and a type `Shared` indicates that it is held by 1 or more transactions.
  - It also stores the set of transactions that currently hold the lock.

- `LockTracker`: This is the class that is responsible for all the locking mechanisms. It includes methods to check if a transaction is allowed to get a shared/exclusive lock, grant shared/exclusive lock to a transaction as well as release locks.
  - This class maintains two in-memory hash tables, one which stores a mapping from a `PageId` to its `PageLock` (we call this table `pageToLock`) and the other, which stores a mapping from `TransactionId` to the set of `PageLocks` the transaction holds (we call this table `transactionToLocks`).
  - Everytime a page is loaded into the `BufferPool` from disk, we check whether it already has an entry in the `pageToLock` mapping and if not, we create a lock for the page and add it to the mapping. When a page is flushed back to disk, its locks are still kept track of and are referred to when the page is read into the `BufferPool` again.
  - `BufferPool` includes an instance of `LockTracker` and `BufferPool.getPage()` calls methods of this instance to keep track of locks.
  - Other methods of `BufferPool` that need to modify locks (such as `releasePage()` and `transactionComplete()` also call methods of this `LockTracker` instance.

It is interesting to note that this was not the design we started out with. Initially, our had the same `PageLock` class as well the hash table mapping `TransactionId`'s to the set of locks held by the transaction. However, instead of keeping a table to map `PageId`'s to `PageLock`'s, we stored a `PageLock` inside each `PageId`. However, with this design, each time a page was evicted from the `BufferPool` and re-fetched from disk, since a new `PageId` was created for it, its locks would get lost.

Missing/Incomplete Parts of Lab

There are no missing/incomplete parts in this lab. The code passes all given unit tests as well as all given system tests.

Time Spent on the Lab

The two of us spent about 20 hours on this lab. Most of our time was spent debugging. At the end, we encountered some trouble debugging the tests with 5 or 10 threads; we had to change the way we had implemented `BufferPool.deleteTuple()` in lab2 to resolve this error.