Lab 5 **Hanjing Wang(516030910015)**
Database System ACM Class, Zhiyuan College, SJTU
Prof. **Feifei Li** Due Date: June 16, 2019

Design decision

simpledb.LockManager

At the very beginning, the idea of designing a LockManager class to deal with locks was inspired the "KISS" (Keep it simple and stupid) principle. However, I found the disentanglement of locks and cache could help me better organize the code and make the code thread-safe. The logical hierarchy is explained as follows.

First, the database should implement page-level locking. Therefore, there is a *ConcurrentHashMap* mapping *PageId* to relevent page locks(namely *locks*). LockStatus is the class indicating the locking status of a page. It has locking and unlocking methods and lock upgrading method. The LockStatus class is design to be thread-safe and hence the atomicity and correctness of page status modifying are promised. When a transaction is requesting for a(an) shared/exclusive lock on a certain page, it first get the corresponding LockStatus instance and call its locking method. If the transaction successfully get the lock it desired, then the transaction will be permitted to continue running. Otherwise, it will be blocked until other transations holding locks on this page complete.

Second, a deadlock-detect algorithm is neccessary here because multiple threads may form a dependency cycle preventing any of them from execution. In *LockManager* I implement a deadlock detect alogorithm based on depth-first search, which aims to find a cycle on the dependency chain start from the latest blocked transaction. If there is no deadlock detected, then the transaction will wait on the LockStatus instance.

Third, to support transaction commiting and aborting operations, the LockManager class has a mapping from *TransactionId* to *Map*{*PageId, Permission*} indicating all the page locks the transaction holds. Here is a trade-off between clarity of definition and the concurrency. We could use *HashSet* here to replace the *ConcurrentHashMap*, which is more consistent with the definition. However, *HashSet* is not a thread-safe class. Hence, we use *ConcurrentHashMap* at the cost of clarity of definition.

Difficulties

Lab5 roughly cost me 4 days. The major difficulty lies in the dealing with high-concurrency situation. And the most important point I got from the exercise is that disentanglement really helps.

At the very beginning, I tried to use *ReadWriteLock* class, which is part of Java features. However, those default implementations bind locks to threads when the probability that multiple threads are involved in a single transaction. Therefore, I have to design a lock structure bounded to transactions.

Then I code the *LockManager* class. Inspired by the "FIFO" principle in single-thread programming, the LockManager holds a waiting list of every page and wake the first one in the list when a lock on the page is released. If it fails, then it will reenter at the rear of the list and again, the first one in the list is waken. The design is too complicate for me to keep it thread-safe since there is a lot shared variants and unexcepted deadlocks happen when threads are visiting those resources.

Finally, I rewrote the *LockManager* class with the inspiration of "Guard Lock". I realized that the order of the execution of different threads actually does not matter as long as it is at the "Repeatable-Read" level. The wait-and-notify structure will fit the entire design. And it is done in a simpler way because of the disentanglement of locking and other transaction operations.