Shanghai Jiao Tong University
**Lab5 Report: Transaction**
Name: Yunxuan Xiao
Sid: 516030901268

Database 2019
June 14th

# 1 Implementations

In this writeup, I will introduce my implementation of *LockManager*, *BufferPool* and the bugs I encountered when testing these parts.

## 1.1 LockManager

### 1.1.1 Locks

There are two types of Lock in SimpleDB setting: Shared Lock and Exclusive Lock. Firstly I create an Object as lock for each page. Secondly, I initialize two ConcurrentHashMap to keep track of which transaction(s) maintain the Shared/Exclusive lock of the page. Since for a single page, multiple transactions can maintain the shared Lock of a page, while only one transaction can maintain the exclusive lock.

### 1.1.2 Acquire Lock

When a transaction start to require a lock, it tries to enter the critical section. If the lock is held by another transaction, current thread will be blocked outside the critical section and keep waiting until that transaction release the lock.

### 1.1.3 Exclusive Lock == Dirtied Pages

To ensure high throughput and high concurrency for our BufferPool, we cannot set the bufferpool as synchronized object. But this will make it unsafe to check a page is dirtied by which specific transaction through buffer pool, since the buffer pool might be concurrently modified by hundreds of transactions.

Here, a much more reliable way is to keep a ExclusiveLockMap to track which pages' $READ\_WRITE$ locks are held by a transaction. Since through the life time of a transaction, once it acquired a Exclusive Lock, it will holds the pages' lock exclusively all along. Moreover, the eviction policy ensures that a dirty page will not be write back to disc, which means the Exclusive Locks a transaction holds are equivalent to the pages dirtied by that transaction. In my code the method *getTransactionDirtiedPages()* returns this dirty page record and will be really useful when tackling TransactionComplete.

Here I have to acknowledge Xiaoyuan Liu to share this idea for me. Before that I was struggling with page missing problem in buffer pool caused by high concurrency.

### 1.1.4 Dependency Graph

To avoid and detect deadlock, we need a data structure to record the dependency between each transaction. Here I create a DAG as dependency graph, where each directed edge

starts from a transaction $t_1$ and points to transaction $t_2$ that holds the lock that $t_1$ intends to get.
Now it becomes really easy to detect deadlock. We simply have to run DFS when a transaction is newly added to the and see if there is a cycle. Throw a TransactionAbortedException if a deadlock is detected and the upper level code will handle this exception.

## 1.2 Buffer Pool

### 1.2.1 Revised Eviction Policy

Pitifully, I have to say I nearly totally rewrite my *BufferPool.java* in this lab. Previously I implemented a LRUCache to support the Least Recently Used eviction policy. It works quite well through lab1 to lab4, of course under single thread setting. When there are multiple threads, LRUCache becomes meaningless since you do not know which page is used first or later. Also the randomly eviction and insertion diminished the efficiency of LRU policy. So I changed the eviction policy as Random Eviction, which will evict a clean page once it was found.

### 1.2.2 Transaction Complete

If a transaction is commited, we just have to FORCE write the dirtied pages back to disc. If a transaction is aborted, we need to recover the page dirtied by this transaction. Here we read the original page again from disc since it must be a clean, unchanged old version. Finally we release page and give back all locks it holds.

# 2 Debugging

It's never an easy stuff to debug multi-thread program. It tooks me two days to pass BTreeTest. Here is some problems I encountered in debugging.

## 2.1 Safety and Efficiency Trade-off

At the very beginning, I tries to control data consistency using buffer pool only. I tried to add a lot of synchronized keywords to restrict access to pid2page map. It does make the insertion/deletion become more safe and greatly reduced the page missing error. However the program becomes quite slow and cannot pass previous BigJoinTest. So finally I give up trying and start using LockManager to ensure data consistency.

## 2.2 Abortion Recovery

When transaction is aborted, how to recover the dirtied page becomes a crucial problem to ensure Consistency. At beginning I used the pre-defined API: *GetBeforeImage()* and *SetBeforeImage()*. However, since its implementation in BTreeFile does not properly organize the image record, many tuples will be lost after transaction abortion if you use *setBeforeImage()*. Since it involves a lot of modifications and uncertainty to change its original code, I give up this idea and use the pagedata on disc to recover. As I said in section 1.1.3, the dirtied page will not be evicted to disc and the transaction holds a page exclusively through its lifetime, thus ensures the cleanness of page on disc. Every

time a transaction is aborted, I will replace the dirtied page in buffer pool with the old but clean version on disc.

### 2.3 Pass All Test

When changing implementations upon previous version, I should be very careful in order not to disobey previous tests. Therefore I firstly set the *run()* method in BTreeDeleter and BTreeInserter as synchronized to make sure the BTreeFile read/write implementation is correct. Secondly, I change all read-only permission into read-write to exclude the probability of deadlock and check the correctness of all other parts. Then finally check concurrent correctness. My implementation can pass BTreeTest in 20 seconds.

## 3 Conclusion

Lab5 takes me 3 days to implement and debug, which is the longest time ever before. I learned a lot about lock managing and multi-thread debugging. Also it helps a lot to adjust your design as the demand changing, although it might take more time.

Thank God, I have passed!