Report: ACMDB LAB5

Jiasen Li

LIJIASEN0921@SJTU.EDU.CN

ACM HONOR CLASS

Design Decision

Lock

My lock is "Lock with TransactionId". There is a HashMap named *LockPool* from PageId to Lock. Each lock belongs to a PageId.

Long Live Lock

The Lock is not with BufferPool. I mean that there is a lock even if the Page is Evicted. This decision does not cost too much Memory Footprint because a Lock is much smaller than a Page. This decision can help me evict the Pages with Read Locks.

Data Structure

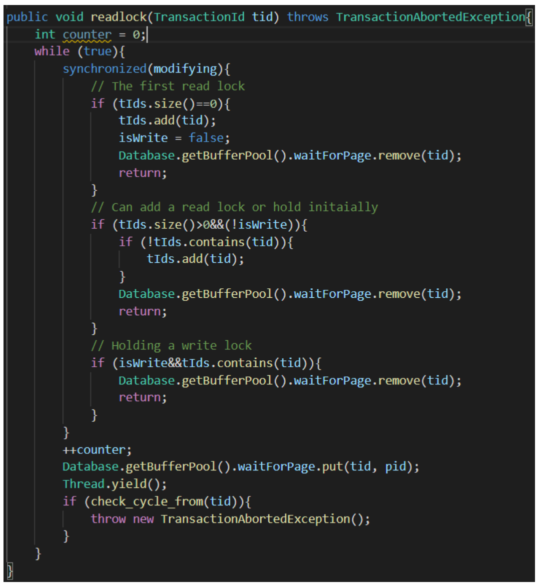
Each lock has a HashSet to store the TransactionIds. The Capacity of initial HashSet is 4, so it does not waste too much Memory Footprint. Each lock has a "iswrite" to show whether it is read mode or write mode.

Each Lock has a synch Object "Modifying" to ensure modifying is atomic.

How to Lock

I design a lock that does not depend on thread blocking. Instead, I let the thread sleep until it gets the lock.

The thread can wake up many times to check whether it can get the lock. In order not to cause race condition, I let the lock acquiring be atomic by using synch.



```java
public void readlock(TransactionId tid) throws TransactionAbortedException{
    int counter = 0;
    while (true){
        synchronized(modifying){
            // The first read lock
            if (tIds.size()==0){
                tIds.add(tid);
                isWrite = false;
                Database.getBufferPool().waitForPage.remove(tid);
                return;
            }
            // Can add a read lock or hold initaially
            if (tIds.size()>0&&(!isWrite)){
                if (!tIds.contains(tid)){
                    tIds.add(tid);
                }
                Database.getBufferPool().waitForPage.remove(tid);
                return;
            }
            // Holding a write lock
            if (isWrite&&tIds.contains(tid)){
                Database.getBufferPool().waitForPage.remove(tid);
                return;
            }
        }
        ++counter;
        Database.getBufferPool().waitForPage.put(tid, pid);
        Thread.yield();
        if (check_cycle_from(tid)){
            throw new TransactionAbortedException();
        }
    }
}
```

Here is the yield version. Yield can be easily replaced by sleep here. Sleeping time can be exponential of the counter, and the number of Environment Switching can be not too large. However, the yield version is okay, so I just use it.

Dead Lock Detection

I implement the dead lock detection using DFS search on the Graph of the Transactions. A lock, shared or exclusive, waits for the lock holders. And a transaction waits for a single lock.

Each time the thread wakes up, it checks whether it is in a cycle.

I use synch to make sure the lock is atomic.

BufferPool

My getPage function gets the lock at first and then get the page.

My releasePage function uses the lock I write.

My BufferPool has Normal Cache and Victim Cache. Normal Cache is for normal Pages. When a Page from Normal Cache cannot be evicted, it is pushed into the Victim Cache. When the lock is released, it goes to Normal Cache again.

My transactionComplete function checks that all the Dirty Pages are released with this tid after either flushing or discarding.

Read Locked Pages can be Evicted. They are released when the transaction completes.

Files are Equal

I treat the files equally. HeapFile and BTreeFile have their Pages in the BufferPool. I know that BTreeFile has getPage with dirtyPage Management. I do the same to HeapFile. HeapFile has the same dirtyPage Management.

All dirty pages are kept in the BufferPool. Because:

1) The Dirty Page must be in the memory

2) Java is a language with pointer, so keeping it in the BufferPool does not need more memory.

Transaction

NO STEAL/FORCE Strategy

Read Locks are released in advance. This is not necessary because such Pages can be evicted. They can wait for the release until the transaction ends. I implement this just for fun.

Ideas

Any Synch block should have no possibility to be blocked. I mean that it should always have the ability to pass through without another lock. Otherwise, A deadlock will occur at that point.

Any Global Variable should be taken good care of. It should either be atomically operated or be copied to local or be with lock.

Never lose any opportunity to increase its robustness, or you will feel bad when you debug.

Mark Dirty function can be useless, but it can be useful.

Standard Concurrent Data Structure is very useful, which can save a lot of locks.

Once it acquire write lock, it can be considered atomic, because no one else will enter.

Future Work (Not Required)

NO STEAL/FORCE is relatively slow. DB can have a buffer in the Memory for dirty pages.

Insertion operation of the Heap File can use multi thread automatically. I mean find the Pages at the same time after a single insertion. With such function, one can write higher level code and automatically use multi thread without any thread operation.

The locations of empty slots can be stored to speed up the Insertion.

A higher level code can be written. A log system can be improved.

I am using LRU with Victim Cache. The LRU can be replaced with other eviction strategy. The current LRU is using a Concurrent Tree Map to find the least recently used Page, with Log time. It can be reduced to constant time.

Some constant level speedup can be performed. For example, the BTreeFile could be Top-Down and have fewer dirty pages.

API Change

I added the lock and its operation functions. There is no big change. I tried different default values and see results.

There is only API Added. No API is changed.

My database is complete. However, It can be improved to gain higher efficiency.

How long did I spend

About 3 days. I heard that lab 5 is very difficult, so I treated my code ***very very carefully***. I think twice before I code, so I did not waste too much time in debugging.

With the help of TAs, I learnt a lot about the Transaction in the Database.

This is the last lab for Database Course Project. I feel very glad to have passed all the tests. I tested all the tests from lab 1 to 4 whenever I made a small change to my code. When I managed to pass 5.1 and lab 1 to 4, I found that the whole lab 5 is already implemented.

Think twice, code once. Debug can be avoided.