



南開大學
Nankai University

计算机学院
数据库 SimpleDB 实验报告

Lab4

姓名：唐明昊

学号：2113927

专业：计算机科学与技术

2023 年 5 月 23 日

目录

1	Acquire Lock	2
1.1	PageLock	2
1.2	PageLockManager	2
1.3	getPage	3
2	releasePage	4
3	evictPage	4
4	TransactionComplete	5
5	Deadlocks	6
6	Extra Credit	6
6.1	Dependency Graphs	6
6.2	Abort others	7
7	注释/改动	9
8	Git Commit History	9

1 Acquire Lock

1.1 PageLock

在 page 的粒度下上锁，封装数据结构 PageLock，保存锁对应的事务以及锁的类型。

1.2 PageLockManager

PageLockManager 用于记录和管理事务所拥有的锁，以及确定是否能够将锁给与一个事务。

PageLockManager 首先需要有一个 Map 的结构来保存各个事务与锁的状态：

ConcurrentHashMap<PageId,ConcurrentHashMap<TransactionId,PageLock>>lockedPages

一个 page 可能会被多个事务上锁，所以使用 pageId 作为 key，索引出一个记录事务及其对应锁的 map。使用 map 是为了方便在知道 TransactionId 后定位到对应的 PageLock。

核心函数 acquireLock 由 bufferPool 的 getPage 调用，根据请求锁的类型以及当前页上锁的状况，确定能否进行加锁。

- 若当前页上没有锁，则当前事务可以直接对其加锁。
- 若当前页面上有锁，且当前事务对其未持有锁。需要进行判断，由于已经有锁，则不可能再加独占锁（写锁）；而如果想加共享锁（读锁），则需要判断当前页上是否有独占锁，没有即可加锁。
- 若当前事务已经对当前页持有锁。分情况讨论，已持有写锁，则一定可以再加锁；已持有的是读锁，并且请求读锁，可以加锁；若已持有读锁，并且想将其升级为写锁，则需要判断当前页上是否还有其他页面对其加锁。

acquireLock

```

1 public synchronized boolean acquireLock(PageId pid,TransactionId tid,int lockType) {
2     // 判断当前页是否有锁
3     if(!lockedPages.containsKey(pid)){
4         PageLock lock=new PageLock(tid,lockType);
5         // keep track of which locks each transaction holds
6         ConcurrentHashMap<TransactionId,PageLock> pageLocks=new ConcurrentHashMap<>();
7         pageLocks.put(tid, lock);
8         // 一页上面可以同时被多个事务加锁
9         lockedPages.put(pid, pageLocks);
10        return true;
11    }
12    // 当前页上有锁
13    ConcurrentHashMap<TransactionId,PageLock> pageLocks=lockedPages.get(pid);
14    if(!pageLocks.containsKey(tid)) { // 当前事务未对page上锁
15        if(lockType==PageLock.EXCLUSIVE) // 已经有锁，无法再上exclusive
16            return false;
17        // 看当前page上是否有exclusive
18        if(pageLocks.size()>1) { // 多于一把锁，肯定是shared
19            PageLock lock=new PageLock(tid,lockType);
20            pageLocks.put(tid, lock);
21            lockedPages.put(pid, pageLocks);
22            return true;

```

```

23     }
24     else { // 就一把锁
25         PageLock theOne = null;
26         for (PageLock temp : pageLocks.values())
27             theOne = temp;
28         if (theOne.getType() == PageLock.EXCLUSIVE) // 如果是 exclusive, 没法
29             return false;
30         else {
31             PageLock lock = new PageLock(tid, lockType);
32             pageLocks.put(tid, lock);
33             lockedPages.put(pid, pageLocks);
34             return true;
35         }
36     }
37 }
38 else { // 当前事务对page持有锁
39     PageLock lock = pageLocks.get(tid);
40     if (lock.getType() == PageLock.SHARED) { // 原来上面有把读锁
41         if (lockType == PageLock.SHARED) // 新请求还是读锁
42             return true;
43         else { // 新请求是写锁
44             // If transaction t is the only transaction holding
45             // a shared lock on an object o, t may upgrade its
46             // lock on o to an exclusive lock.
47             if (pageLocks.size() == 1) {
48                 lock.setType(PageLock.EXCLUSIVE);
49                 pageLocks.put(tid, lock);
50                 return true;
51             }
52             else { // 不止一个事务, 其他事务对它还有读锁
53                 return false; // 不能改成写锁
54             }
55         }
56     }
57     else // 原来就有一把写锁, 肯定行
58         return true;
59 }
60 }

```

1.3 getPage

修改 bufferPool 的 getPage 函数, 在从缓存中读取 page 前首先为 tid 获取锁。采取忙等待的方式, 不断循环调用 lockManager 的 acquireLock 函数。

getPage

```

1 // 先判断是什么锁
2 int lockType = perm == Permissions.READ_ONLY ? PageLock.SHARED : PageLock.EXCLUSIVE;
3 boolean isAcquired = false;

```

```

4 // 循环获取锁
5 while(!isAcquired) { // 忙等待
6     isAcquired=lockManager.acquireLock(pid, tid, lockType);
7 }

```

2 releasePage

在事务完成或者中断后，需要释放事务对页上的锁。bufferPool 调用 lockManager 的 releaseLock 函数即可。

特别的，在插入操作时，如果当前页上没有空的槽位，则需要立即释放对其的锁。

releaseLock

```

1 public synchronized boolean releaseLock(PageId pid, TransactionId tid) {
2     if(isHoldLock(pid, tid)) {
3         ConcurrentHashMap<TransactionId, PageLock> pageLocks=lockedPages.get(pid);
4         pageLocks.remove(tid); // 释放当前事务对page的锁
5         if(pageLocks.size()==0)
6             lockedPages.remove(pid); // 当前页上已没有事务对其有锁
7         return true;
8     }
9     return false;
10 }

```

3 evictPage

事务的修改只有在其被 commit 之后才写到磁盘上去，所以在丢弃 page 时，对于尚处在事务中的 page，不应该将其逐出缓存。更进一步，如果事务没有修改 page，即 page 非 dirty page，即使事务没有结束，仍然可以将其逐出，只是由于我前面的 operator 实现上可能有问题，会造成引用空对象，需要额外打补丁来维护缓存。

修改 evictPage 函数，在逐出前做判断。

evictPage

```

1 private synchronized void evictPage() throws DbException {
2     // cache-> LRU原则
3     // we must not evict dirty pages.
4     for(int i=0; i<numPages; i++) {
5         PageId pid=pageOrder.peek();
6         Page p=pages.get(pid);
7         // 报错? p为null? ->回滚造成的错
8         if(p==null) { // 打补丁
9             pages.remove(pid);
10            pageOrder.remove(pid);
11            continue;
12        }

```

```

13         if(p.isDirty()!=null) { // 是脏页
14             pageOrder.remove(pid); // 放到后面去
15             pageOrder.offer(pid);
16         }
17         else {
18             // 没必要刷新啊，反正只会evict不脏的
19             // try {
20             //     flushPage(pid);
21             // } catch (IOException e) {
22             //     // TODO Auto-generated catch block
23             //     e.printStackTrace();
24             // }
25             discardPage(pid);
26             return;
27         }
28     }
29     throw new DbException("all the pages in the bufferPool are dirty!");
30 }

```

4 TransactionComplete

事务结束，不管是提交还是中止，都需要调用 transactionComplete 函数。

当事务是正常提交结束时，调用 flushPages 函数，将当前事务锁定的页都进行刷新，并将其写入磁盘中。

若事务是非正常结束，则需要撤销之前对 page 所做的修改。遍历缓存，被当前事务污染的页都需要从磁盘中重新读取数据，恢复原始状态。

以上工作做完以后，需要调用 lockManager 的 releaseAllLock 函数释放当前事务所有的锁。

transactionComplete

```

1 public void transactionComplete(TransactionId tid, boolean commit)
2     throws IOException {
3     if(commit) {
4         // flush dirty pages associated to the transaction to disk
5         flushPages(tid);
6     }
7     else {
8         // restoring the page to its on-disk state
9         // page-level 需要锁吧
10        synchronized(this) {
11            for(Page page:pages.values()) {
12                // 遍历，找到当前事务弄脏的page
13                if(tid.equals(page.isDirty())) {
14                    PageId pid=page.getId();
15                    // 从磁盘读出原来的那页
16                    DbFile
                        table=Database.getCatalog().getDatabaseFile(pid.getTableId());

```

```

17         Page old=table.readPage(pid);
18         //pages.remove(pid);
19         // 放回缓存, 覆盖原来的page
20         pages.put(pid, old); // 定位出来就是这有问题, 写了个null?
21         // 调整链表
22         pageOrder.remove(pid);
23         pageOrder.offer(pid);
24     }
25 }
26 }
27 }
28 //release any state the BufferPool keeps regarding the transaction
29 lockManager.releaseAllLocks(tid);
30 }

```

5 Deadlocks

采用超时检测策略检测死锁, 当事务等待获取锁超过一段时间, 认为发现死锁。当死锁产生, 抛出异常, 中止当前事务。

getPage

```

1 // 先判断是什么锁
2 int lockType=perm==Permissions.READ_ONLY?PageLock.SHARED:PageLock.EXCLUSIVE;
3 boolean isAcquired=false;
4 // detect deadlock
5 long startTrying=System.currentTimeMillis();
6 // 循环获取锁
7 while(!isAcquired) { // 忙等待
8     isAcquired=lockManager.acquireLock(pid, tid, lockType);
9     long nowTrying =System.currentTimeMillis();
10
11     // resolve deadlock
12     if(nowTrying-startTrying>250) // timeout!
13         // 放弃当前事务t
14         throw new TransactionAbortedException();
15 }

```

6 Extra Credit

6.1 Dependency Graphs

采用依赖图检测是否存在死锁。采用 HashMap 记录某个事务所依赖的其他事务。

ConcurrentHashMap<TransactionId, Set<TransactionId>> dependencyGraph

在 acquireLock 时, 如果当前事务需要等待其他事务执行完才能获取锁, 则其需要依赖那些事务, 记录它所依赖的事务 (addDependency)。当一个事务获取到锁时, 它不再依赖其他事务, 消除依赖关

系 (removeDependency)。

在 getPage 循环获取锁时，每次循环迭代检测一遍，当前依赖图中是否构成圈，若存在则产生死锁。

采用拓扑排序的方法检测圈。

- 首先从当前事务往下广度优先搜索，将其能触及到的节点记录下来。
- 而后对记录的节点依次遍历，将其后继节点的入度加一。
- 采用拓扑排序，每次取出入度为 0 的节点，将其后继节点入度减一。当没有入度为 0 节点时中止循环，最终如果入度表还有剩余节点，则一定存在环。

isExistCycle

```

1 // 计算入度
2 for(TransactionId nowId:out) {
3     // 计算每个节点的入度
4     Set<TransactionId> outNodes=dependencyGraph.get(nowId);
5     if(outNodes==null)continue;
6     for(TransactionId outId:outNodes) {
7         Integer temp=inDegree.get(outId);
8         inDegree.put(outId, temp+1);
9     }
10 }
11 // 拓扑排序检测圈
12 while(true) {
13     int count=0;
14     for(TransactionId nowId:out) {
15         if(inDegree.get(nowId)==null)continue;
16         if(inDegree.get(nowId)==0) { // 入度为0
17             Set<TransactionId> outNodes=dependencyGraph.get(nowId);
18             // 移除该节点，后续节点入度减一
19             if(outNodes==null)continue;
20             for(TransactionId outId:outNodes) {
21                 Integer temp=inDegree.get(outId);
22                 inDegree.put(outId, temp-1);
23             }
24             inDegree.remove(nowId);
25             count++;
26         }
27     }
28     if(count==0) break; // 没有入度为0的节点了
29 }

```

6.2 Abort others

当死锁产生时，除了中止自身，还可以中止当前事务所依赖的其他事务。

由于前面依赖图检测时，在 acquireLock 阶段已经将当前事务依赖的事务记录下来，所以当死锁产生时，需要让其他事务中止，让其抛出异常。

用一个 HashMap 来记录所有事务的工作状态，在 getPage 阶段，只有事务为正常工作状态才会去 acquireLock，如果因为死锁被其他线程置为不能工作，则抛出异常。

abortOthers

```

1 // 当死锁产生，将自己以外的tid中止掉
2 public synchronized void abortOthers(TransactionId tid) {
3     Set<TransactionId> waiting=dependencyGraph.get(tid);
4     if(waiting==null)return;
5     for(TransactionId waitId:waiting) {
6         for(TransactionId waitId:waiting)
7             tidOnWorking.put(tid, false); // 别再工作了
8     }
9     removeDependency(tid);
10 }

```

getPage

```

1 // ——中止其他人
2 while(!isAcquired) {
3     // 初始化
4     Boolean temp=lockManager.tidOnWorking.get(tid);
5     if(temp==null)lockManager.tidOnWorking.put(tid, true);
6
7     if(lockManager.tidOnWorking.get(tid)) { // 当前线程在工作
8         isAcquired=lockManager.acquireLock(pid, tid, lockType);
9     }
10    else { // 被其他事务中止
11        throw new TransactionAbortedException();
12    }
13    long nowTrying =System.currentTimeMillis();
14    // resolve deadlock
15    if(nowTrying-startTrying>300) // timeout!
16        // 中止其他事务
17        lockManager.abortOthers(tid);
18 }

```

经实测，采取该方法速度并不比 abort 自身的方法慢多少，时间表现上效果相当。

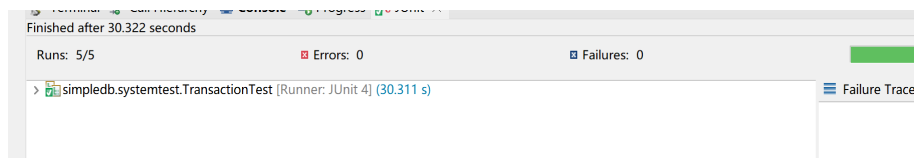


图 6.1: abort others

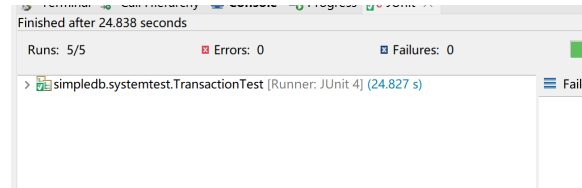


图 6.2: abort itself

7 注释/改动

在跑 `System.TransactionTest` 时, 正常跑耗时较长, 需要在 `getPage` 前增加 `synchronized` 关键字。在加了 `synchronized` 后跑该 Test 速度大为提升。但测试发现 `BTreeNextKeyLockingTest` 此时会卡住, 由于时间所限, 没能探究其底层逻辑。为了通过 `BTreeNextKeyLockingTest` 又需要将 `synchronized` 关键字删除。

8 Git Commit History

























22 May, 2023 - 1 commit		
 lab4 exercise 5	tang authored 17 hours ago	7c46d980  
21 May, 2023 - 1 commit		
 passed TransactionTest but slowly	tang authored 1 day ago	a9d0a352  
19 May, 2023 - 1 commit		
 still working	tang authored 3 days ago	7a41d25c  
18 May, 2023 - 1 commit		
 lab4 exercise5 --last two test	tang authored 4 days ago	2896b385  
17 May, 2023 - 3 commits		
 lab4 exercise4	tang authored 5 days ago	db99b509  
 lab4 exercise3	tang authored 5 days ago	1d8b618c  
 Lab4 exercise1&2	tang authored 5 days ago	01b06ac4  
16 May, 2023 - 1 commit		
 lab4 exercise1 --PageLock&LockManager	tang authored 6 days ago	22521363  

图 8.3: Git Commit History