



南開大學
Nankai University

计算机学院
SimpleDB 实验报告

Lab4 Transactions

姓名：朱霞洋

学号：2113301

专业：计算机科学与技术

2023 年 5 月 23 日

摘要

lab4 实现与事务 Transaction 有关的方法，并实现页粒度的两端锁，实现 SimpleDB 多事务处理的功能。


代码链接：[朱霞洋的 Gitlab 仓库](#)


目录

1	提交历史	2
2	设计思路	2
2.1	Exercise 1 实现页粒度的锁并完善相应方法	2
2.2	Exercise 2+Exercise 5 getPage 的完善和死锁解决	7
2.3	Exercise 3 实现 NO STEAL 策略	8
2.4	Exercise4 实现 transactionComplete	8
2.5	重难点	9
3	改动部分	10
4	测试结果	10

1 提交历史

16 May, 2023 - 2 commits

 **lab4 fixed ConcurrentModificationException**
X_yang2113301 authored 6 days ago

 **lab4 completed fixing BTreeTest**
X_yang2113301 authored 6 days ago

15 May, 2023 - 1 commit


 **lab4 exercise1 and exercise2**
X_yang2113301 authored 1 week ago

图 1.1: 提交历史

2 设计思路

Transaction（事务）是一组以原子方式执行的数据库操作（插入、删除或读取等），要么所有的动作都完成了，要么一个动作都没有完成。

为了防止事务之间相互干涉导致读写异常，Lab4 中将为 SimpleDB 实现两段锁，两段锁的阶段分别为：

1. 第一阶段：扩展阶段，事务可以申请获得任意数据上的任意锁，但是不能释放任何锁；
2. 第二阶段：收缩阶段，事务可以释放任何数据上的任何类型的锁，但是不能再次申请任何锁

两段锁协议并不要求事务必须一次将所有要使用的数据全部加锁，因此遵守两段锁协议的事务可能发生死锁。于是 Lab4 中也要实现死锁的检测和解决。

此外，Lab4 还要实现 NO STEAL 策略，即事务对 page 的修改只有在 commit 之后才会写入到磁盘（flush），但是在之前的 Lab 中实现页面置换策略时，当置换掉的页面是 dirty page 时，也会将更改写回到磁盘。这是不允许的，所以需要完善 evictPage() 方法，当需要置换的 page 是 dirty page 时，需要跳过此 page，去置换下一个非 dirty 的 page。

2.1 Exercise 1 实现页粒度的锁并完善相应方法

SimpleDB 中的两端锁的要求如下：

1. 在事务可以读取一个对象之前，它必须拥有一个共享锁 (SHARED LOCK)；
2. 在一个事务可以写一个对象之前，它必须有一个互斥锁 (EXCLUSIVE LOCK)；
3. 多个事务可以在一个对象上拥有一个共享锁；

4. 只有一个事务能对一个对象具有互斥锁;
5. 如果对象 o 上只有事务 t 持有共享锁, 则 t 可以将其对 o 的锁升级为互斥锁;

为此, 首先在 BufferPool 中实现一个 Lock 类:

```
1 private class Lock{
2     public static final int SHARE = 0; //0 stands for shared lock,
3     // Multiple transactions can have a shared lock on an object before reading
4     public static final int EXCLUSIVE = 1; //1 stands for exclusive lock,
5     // Only one transaction may have an exclusive lock on an object before writing
6     private TransactionId tid;
7     private int type;
8
9     public Lock(TransactionId tid, int type){
10         this.tid = tid;
11         this.type = type;
12     }
13     public TransactionId getTid() {
14         return tid;
15     }
16     public int getType() {
17         return type;
18     }
19
20     public void setType(int type) {
21         this.type = type;
22     }
23 }
```

而后, 需要再创建一个 LockManager 类实现对每个事务在每个页上的锁的管理, 包括获取和释放, 而获取锁的逻辑如下:



图 2.2: 锁管理器

整个 BufferPool 中只需要一个 LockManager 成员变量即可。在 LockManager 中, 创建了一个 `ConcurrentHashMap<PageId,ConcurrentHashMap<TransactionId,PageLock>>` lockMap 对象来记录每个 Page 上事务 t 和其锁的映射, 该类型的变量在多线程访问时更加安全。

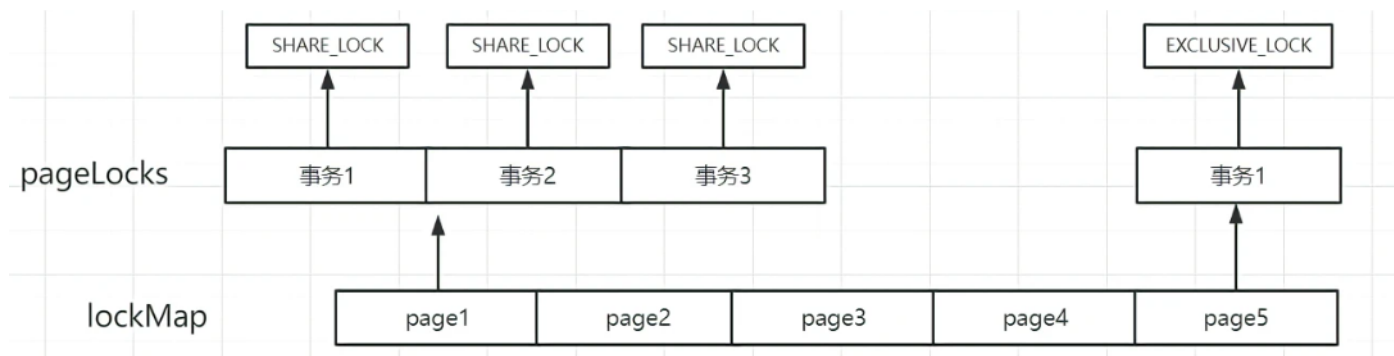


图 2.3: 页面上事务和锁的映射

具体的代码和注释如下:

```

1 private class LockManager{
2     private ConcurrentHashMap<PageId,ConcurrentHashMap<TransactionId,Lock>> pageLocks;
3     //use ConcurrentHashMap instead of HashMap cuz it's safer than the latter
4     //when facing multiple threads
5
6     public LockManager(){
7         pageLocks = new ConcurrentHashMap<PageId,ConcurrentHashMap<TransactionId,Lock>>();
8     }
9
10    public synchronized boolean acquireLock(PageId pid,TransactionId tid,int lockType)
11        throws TransactionAbortedException {

```

```

12     if(pageLocks.get(pid)==null){ // no existing locks, create newPageLock and return true
13         Lock newLock = new Lock(tid,lockType);
14         ConcurrentHashMap<TransactionId,Lock> newPageLock = new ConcurrentHashMap<>();
15         newPageLock.put(tid,newLock);
16         pageLocks.put(pid,newPageLock);
17         return true;
18     }
19     //if there's locks already
20     ConcurrentHashMap<TransactionId,Lock> nowPageLock = pageLocks.get(pid);
21
22     if(nowPageLock.get(tid)==null){ //no locks from tid
23         if(nowPageLock.size()>1){ //exists locks from other transactions
24             if(lockType == Lock.SHARE){
25                 Lock newLock = new Lock(tid,lockType); //if requiring a read lock
26                 nowPageLock.put(tid,newLock);
27                 pageLocks.put(pid,newPageLock);
28                 return true;
29             }
30             else{ //requiring a write lock
31                 return false;
32             }
33         }
34         else if(nowPageLock.size()==1){ //only one other lock,could be a read or write lock
35             Lock existLock = null;
36             for(Lock lock:nowPageLock.values())
37                 existLock = lock;
38             if(existLock.getType() == Lock.SHARE){ // if it is a read lock
39                 if(lockType == Lock.SHARE){ // require a read lock too, return true
40                     Lock newLock = new Lock(tid,lockType);
41                     nowPageLock.put(tid,newLock);
42                     pageLocks.put(pid,newPageLock);
43                     return true;
44                 }
45                 else if(lockType == Lock.EXCLUSIVE){
46                     return false;
47                 }
48             }
49             else if(existLock.getType() == Lock.EXCLUSIVE){ //some transaction is writing this page
50                 return false;
51             }
52         }
53     }
54     else{
55         Lock preLock = nowPageLock.get(tid);
56         if(preLock.getType() == Lock.SHARE){ //if previous lock is a read lock
57             if(lockType == Lock.SHARE){ //and require a read lock too
58                 return true;
59             }
60             else{ // if want a write lock

```

```

61         if(nowPageLock.size() == 1){ // if only this tid hold a lock,we could update it
62             preLock.setType(Lock.EXCLUSIVE);
63             nowPageLock.put(tid,preLock);
64             return true;
65         }
66         else{
67             throw new TransactionAbortedException();
68         }
69     }
70 }
71
72     else if(preLock.getType() == Lock.EXCLUSIVE){ //the previous one is a write lock
73         //means no other locks on this page
74         return true;
75     }
76 }
77 return false;
78 }
79
80 public synchronized boolean holdsLock(TransactionId tid,PageId pid){
81     if(pageLocks.get(pid) == null){
82         return false;
83     }
84     ConcurrentHashMap<TransactionId,Lock> nowPageLock = pageLocks.get(pid);
85     if(nowPageLock.get(tid) == null){
86         return false;
87     }
88     return true;
89 }
90
91 public synchronized boolean releaseLock(TransactionId tid,PageId pid){
92     if(holdsLock(tid,pid)){
93         ConcurrentHashMap<TransactionId,Lock> nowPageLock = pageLocks.get(pid);
94         nowPageLock.remove(tid);
95         if(nowPageLock.size() == 0){
96             pageLocks.remove(pid);
97         }
98         this.notifyAll();
99         return true;
100     }
101     return false;
102 }
103
104
105 public synchronized boolean TransactonCommitted(TransactionId tid){ // when a transaction
    completes,release all the locks
106     for(PageId pid : pageLocks.keySet()){
107         releaseLock(tid,pid);
108     }

```

```

109         return true;
110     }
111
112 }

```

在 lockManager 里的每一个方法前都加上了 synchronized 关键字，也是出于对多线程并发控制的考虑，其中 acquireLock 即是对以上获取锁的逻辑的代码实现，holdsLock 查看页面 p 上是否有事务 t 的锁；releaseLock 则是释放事务 t 在页面 p 上的锁；TransactionCommitted 则是释放一个事务 t 获取过的所有锁，表示该事务完成。

2.2 Exercise 2+Exercise 5 getPage 的完善和死锁解决

在此前实现的文件的读取页面的方法中，都是通过调用 BufferPool 的 getPage() 方法来完成的，因此只用在 BufferPool 中实现锁的获取即可。在此处我直接与 Exercise5 中的死锁检测和解决结合起来完善了 getPage() 方法。

在如 HeapFile 等文件的 getPage 函数中，调用 BufferPool 的 getPage 方法时会传输一个 Permission 对象，表示对该页面的操作权限，可根据此来对应判断该获取哪一种锁。除了 READ_ONLY 是 SHARED LOCK 外，其余的都是 EXCLUSIVE LOCK，因此 getPage 中需要加入这一段代码：

```

1  int lockType = (perm == Permissions.READ_ONLY)? Lock.SHARE : Lock.EXCLUSIVE;
2  //only READ_ONLY stands for a shared lock
3  long begin = System.currentTimeMillis();
4  boolean ifAcquired = false;
5  while (!ifAcquired){
6      ifAcquired = manager.acquireLock(pid,tid,lockType);
7      long rightNow = System.currentTimeMillis();
8      if(rightNow - begin > 500) { //timeout policy
9          //System.out.println("time out"+begin+" "+rightNow);
10         throw new TransactionAbortedException();
11     }
12 }

```

其中 lockType 会根据 Permission 的类型判断是哪一种锁，此后在 while 循环中不停尝试获取锁，如果在 500 毫秒内无法得到锁，那么说明有死锁存在，这便是 Exercise 5 中的超时检测死锁 (timeouts) 的机制。

此外，Exercise 2 中还提到，在事务向 page 中的空 slot 插入 tuple 时，首先需要遍历该 page 上有无空 slot，如果该 page 上没有空的 slot，则需要创建一个新的 page 将 tuple 插入其中。此时事务不会对已满的 page 进行操作，但它依旧持有这些 page 上的锁，其它事务也不能访问这些 page，所以当判断某一 page 上的 slot 已满时，可以释放掉该 page 上的锁。尽管不满足两段锁协议，但该事务并没有对 page 进行任何操作，所以并不会有任何影响，而且也可以让其他事务访问这些 page，于是在 HeapFile 的 insertTuple 方法中，需要改写这一段：

```

1  for(int i=0;i<numPages();i++){
2      HeapPage page = (HeapPage) bp.getPage(tid, new
          HeapPageId(this.getId(),i),Permissions.READ_WRITE); //读取对应页
3      if(page.getNumEmptySlots()==0) { //full
4          //added in lab4,when there's no empty slots,we could unlock the page

```



```

5         bp.releasePage(tid,new HeapPageId(this.getId(),i));
6         //this will do no harm to 2PL lock,cuz we didn't read any data
7         continue;
8     }
9     page.insertTuple(t);
10    PageList.add(page);
11    //page.markDirty(true,tid); // added in lab4
12    return PageList;
13 }

```

可见，当某 page 的空 slots 数为 0 时，将释放事务 t 对此页的读锁。

2.3 Exercise 3 实现 NO STEAL 策略

在此前实现的汰换策略 evictPage() 中，会将最近最少使用的页面刷新到磁盘上然后从 BufferPool 中删除，然而在 Lab4 中，为了使事务运行正常，被事务修改过的 dirtypages 只有在 commit 之后才会写入到磁盘，因此 Exercise3 中要对 evictPage() 做一些改动，使得脏页不会被汰换：

```

1 private synchronized void evictPage() throws DbException {
2     int length = recentUsedPages.size()-1;
3     while (recentUsedPages.get(length).isDirty()!=null){ //NO STEAL policy
4         length--; //if it is a dirty page,do not evict it
5         if(length<0)
6             throw new DbException("all pages are dirty");
7     }
8     Page page = recentUsedPages.remove(length);
9     discardPage(page.getId()); //remove it from the BufferPool
10
11 }

```

2.4 Exercise4 实现 transactionComplete

BufferPool 中 transactionComplete() 有重载，一个带有布尔参数（为 true 时，进行提交；false 时进行回滚），另一个无参数，总是提交 committed。

在提交事务 t 时，所有被 t 修改的脏页将全部刷新至磁盘（调用 flushpages），然后 t 拥有的所有锁将全部释放；如果进行回滚，那么将先从内存中删去所有 t 修改过的脏页，然后从磁盘中重新读取这些内存页，具体代码如下：

```

1 public void transactionComplete(TransactionId tid, boolean commit)
2     throws IOException {
3     // some code goes here
4     // not necessary for lab1|lab2
5     if(commit){
6         try {
7             flushPages(tid);
8         }catch (IOException e){
9             e.printStackTrace();
10        }

```

```

11     }
12     else{ //recovery
13         for(Page page : idToPage.values()){
14             PageId pid = page.getId();
15             if(tid.equals(page.isDirty())){ //if tid has modified this page,we try to recover it
16                 int tableId = pid.getTableId();
17                 DbFile file = Database.getCatalog().getDatabaseFile(tableId);
18                 Page recoverPage = file.readPage(pid); //reload the page
19                 idToPage.put(pid,recoverPage);
20                 recentUsedPages.add(0,recoverPage); //reput it
21             }
22         }
23     }
24     manager.TransactionCommitted(tid);
25 }

```

其中，在 LockManager 中实现的 TransactionCommitted 就起到了作用，用于把事务 t 的所有锁释放。

2.5 重难点

本次实验的重难点主要在：

1. 获取锁的逻辑：获取锁时不仅要判断页面是否有锁，锁的类型，有时候甚至要通过页面上锁的数量来进行分支操作，具体可以见2.2；
2. 死锁的判断和解决：对于如何判断事务间发生死锁也是重难点，因为此次实现的两段锁并没有保证死锁不会发生。而 Lab4 中我采用了超时检测死锁的方法，如果某事务在 500ms 内没有得到需要的锁，说明可能发生了死锁，则停止该事务并抛出 TransactionAbortedException 异常；
3. NO STEAL 策略也是重点之一，具体可以参见Exercise 3；
4. 多线程并发读写时的管理：在此次的 test 中，包括多线程对于 BufferPool 的访问，其中 LinkedList 类型的用以存储最近使用页面的 recentUsedPage 在最初面临多线程访问和修改时常常报错，因此我对其的 add 方法和 remove 方法在外层又进行了包装，用 synchronized 关键字修饰，使得多线程访问的安全：

```

1     private synchronized void moveToHead(PageId pid){
2
3         String thread = Thread.currentThread().getName();
4         //System.out.println("modify from "+thread);
5         for(int i = 0;i<recentUsedPages.size();i++){
6             Page page = recentUsedPages.get(i);
7             if(page.getId().equals(pid)) {
8                 recentUsedPages.remove(i);
9                 recentUsedPages.add(0,page); //find the required page,and put it at the top as the recent used
                page
10             break;
11         }
12     }
13     //System.out.println("done from "+thread);

```

```

14     }
15
16     private synchronized void addRecentUsed(Page page){
17         recentUsedPages.add(0,page);
18     }
19
20     private synchronized void delRecentUsed(PageId pid){
21
22         for(int i = 0;i<recentUsedPages.size();i++){
23             Page page = recentUsedPages.get(i);
24             if(pid.equals(page.getId())) {
25                 recentUsedPages.remove(i);
26                 break;
27             }
28         }
29     }
30 }

```

3 改动部分

此次由于新加入的 Transaction 机制，和为了应多多线程的并发访问，做了一些改动：

1. 此前用 HashMap 存储 pageID 到 Page 的映射，此次为了更好的处理多线程的事务，将其也更改为了 ConcurrentHashMap 类型；
2. recentUsedPages 的方法的包装，如同在上一节所述，为了应对多线程并发访问并防止出错，将其一些方法用 synchronized 进行修饰；
3. 其余诸如 evictPage 等等，在 Lab4 的实验要求上进行了改动，具体可以参见上文。

4 测试结果

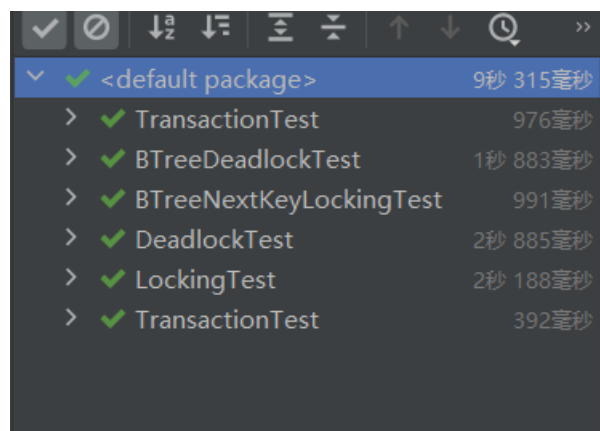


图 4.4: 测试结果