simpleDB 实验报告

Lab4：SimpleDB Transactions

作者：曹珉浩 2113619

时间：May 22, 2023

目录

第 1 章 Exercise-1 **1**

1.1 transaction & 2PL 1

1.2 modify BufferPool 1

1.2.1 Lock & LockManager 2

1.2.2 modify getPage() 4

1.2.3 relaesePage() 4

1.2.4 holdsLock() 5

第 2 章 Exercise-2 & Exercise-3 **6**

第 3 章 Exercise-4 **7**

第 4 章 Exercise-5 **8**

第 5 章 Design alternatives **9**

第 6 章 Commit & Results **10**

第 1 章 Exercise-1

1.1 transaction & 2PL

数据库中的事务是指一组数据库操作组成的逻辑单元，这些操作要么全部成功执行，要么全部回滚（撤销）。事务提供了一种可靠的方式来处理数据库中的并发操作和故障恢复。其具有四个特性 (ACID)：

- 原子性（Atomicity）：事务是一个不可分割的操作单元，要么全部执行成功，要么全部回滚。如果事务中的任何一个操作失败，所有的操作都会被回滚到事务开始之前的状态。
- 一致性（Consistency）：事务执行的结果必须使数据库从一个一致的状态转换到另一个一致的状态。在事务开始之前和结束之后，数据库的完整性约束没有被破坏。
- 隔离性（Isolation）：每个事务的操作应该与其他事务相互隔离，使它们彼此独立。并发执行的多个事务之间不应该相互干扰，每个事务应该感知不到其他事务的存在。
- 持久性（Durability）：一旦事务提交成功，其对数据库的修改应该是永久性的，即使在系统故障或重启之后，修改的数据也应该能够被恢复。

2PL(Two-Phase Locking)：两阶段锁定是数据库管理系统中的一种并发控制协议，用于确保事务的隔离性和一致性，它是一种基于锁的并发控制方法。这两个阶段指的是：

- 加锁阶段（Growing Phase）：在这个阶段，事务可以获取锁，但不能释放锁。事务在执行过程中需要获取所需的全部锁，并且在获取锁之前会等待其他事务释放对应的锁。一旦事务获得了锁，它就可以执行操作。
- 解锁阶段（Shrinking Phase）：在这个阶段，事务只能释放锁，不能再获取新的锁。事务执行完成后，会释放它持有的所有锁，使其他事务能够获取这些锁并执行操作。

2PL 协议的核心思想是在事务执行期间使用锁来保护共享资源，以避免并发访问造成的数据不一致性问题。通过对资源的加锁和解锁操作，2PL 协议保证了事务的隔离性，即每个事务在执行期间能够独立地访问资源，并且事务之间不会相互干扰或产生冲突。但要注意 2PL 协议并不能解决死锁的问题，因此要手动采取一些机制，比如超时机制。

1.2 modify BufferPool

根据实验指导，我们应该采取 NO STEAL/FORCE 的缓冲区管理策略。它的基本思想是：

- NO STEAL(不允许抢占)：如果一个事务对某个数据块进行了修改，但是该事务还未提交，那么其他事务不允许将这个修改的数据块写入磁盘，即不允许抢占。这样可以避免频繁的磁盘写入，提高性能。
- FORCE(强制写入)：当一个事务提交时，它所做的修改必须被强制写入磁盘，以确保数据的持久性。这样可以避免数据在内存中丢失，即使系统崩溃，也能够恢复到提交事务之后的状态。

我们还需要实施共享锁和排他锁，它们的工作原理如下：

- 在一个事务可以读取一个对象之前，它必须有一个共享锁。
- 在一个事务可以写一个对象之前，它必须有一个独占锁。
- 多个事务可以在一个对象上拥有一个共享锁。
- 只有一个事务可以对一个对象拥有独占锁。

- 如果事务 t 是唯一持有对象 o 共享锁的事务，则 t 可以升级 o 对象锁为独占锁。
- 如果一个事务请求本不应被授权的锁，其代码会被阻塞直至锁恢复可用状态。

1.2.1 Lock & LockManager

为了方便实现与管理，新增 Lock 类如下：

```
public class Lock {
    public TransactionId tid; //事务 id
    public boolean isShared; //是不是共享锁
    public Lock(TransactionId tid,boolean isShared){
        this.tid = tid;
        this.isShared = isShared;
    }
}
```

再新增一个类 LockManager 实现对锁的管理，后续三个方法的修改都会基于此类。首先在此类中，需要有一个数据结构，用于描述页面以及页面上锁的情况 (我们实现页面粒度的锁)，由于共享锁的存在，一个页面上可能有多个锁，因此我们选择建立一个 PageId 到 Lock 数组的映射，而 HashMap 线程不安全，我们选择线程安全的 ConcurrentHashMap：

```
ConcurrentHashMap<PageId,ArrayList<BufferPool.Lock>> lockMap;
```

接下来我们实现获取锁的逻辑，即 acquireLock 方法，很显然这是一个同步方法：

首先，如果调用方法时，该页面上没有任何的锁，那么直接对该页面上锁即可，由于 synchronized 关键字的修饰，不会存在并发修改的情况：

```
if (lockMap.get(pid) == null) {
    Lock lock = new Lock(tid, isShared);
    ArrayList<Lock> locks = new ArrayList<BufferPool.Lock>();
    locks.add(lock);
    lockMap.put(pid, locks);
    return true;
}
```

如果调用方法时该页面上已经有锁了，那么就会有很多种可能，这时我们需要遍历 LockManager 中维护的 Lock 列表。如果在锁列表中能找到 tid 相同的锁，证明该事务已经在此页面上得到过锁了，这时有以下几种可能：

- 两把锁不仅 tid 相同，它们的属性 (isShared) 也相同，则这两把锁就是完全相同的 (锁重入)，认为获取锁成功，且不对原有数据结构做任何改动

- 两把锁的属性不同时，如果原有锁为独占锁，那么请求锁就是共享锁，由于独占锁拥有共享锁的性质，所以在拥有独占锁时请求一个共享锁，我们也认为是获取锁成功的
- 经过上述两重条件过滤，现在的情况是原有锁为共享锁，而请求锁为独占锁。这个时候需要判断 Lock 列表的大小，如果 Lock 列表中只有这一把锁，那么可以把共享锁升级为独占锁
- 最后剩下的一重逻辑是，该页面上有多个事务正在共享锁，而正在请求一把独占锁，那么判定获取锁失败

而如果遍历 Lock 列表，没有发现 tid 与传入 tid 相同的锁，且由于我们最早判断了页面上无锁的情况，则此时只能获取一个共享锁，主要有以下几种情况：

- 如果 Lock 列表的大小为 1，且为独占锁，那么无论想要获取什么锁都会失败
- 经过上述逻辑过滤，页面上不是独占锁，那么如果想要获取的锁是共享锁，就可以成功获取，并修改 LockManager 的 Lock 列表
- 想要获取独占锁，而页面只能被获取共享锁，则判定失败

```
ArrayList lockList = lockMap.get(pid);
for (Object o : lockList) {
    Lock lock = (Lock) o;
    if (lock.tid == tid) {
        if (lock.isShared == isShared)
            return true;
        if (!lock.isShared)
            return true;
        if (lockList.size() == 1) {
            lock.isShared = false;
        }
        else
            return false;
    }
}
if (lockList.size() == 1 && !((Lock) lockList.get(0)).isShared)
    return false;
if (isShared) {
    Lock lock = new Lock(tid, true);
    lockList.add(lock);
    lockMap.put(pid, lockList);
    return true;
}
return false;
```

接下来实现释放锁的逻辑，即 releaseLock 方法：这个方法是实现相对容易，只需要注意一点，如

果删除后 Lock 列表的大小为 0，就说明这个页面上已经没有锁了，为了与我们上面获取锁的逻辑吻合，我们需要把这个映射从 Lockmap 中删除。

```
public synchronized boolean releaseLock(PageId pid,TransactionId tid){
    if(lockMap.get(pid) == null)
        return false;
    ArrayList<Lock> locks = lockMap.get(pid);
    for(int i=0;i<locks.size();++i){
        Lock lock = locks.get(i);
        if(lock.tid == tid){
            locks.remove(lock);
            if(locks.size() == 0)
                lockMap.remove(pid);
            return true;
        }
    }
    return false;
}
```

1.2.2 modify getPage()

我们之前实现的逻辑，如果要获取的页正在被修改或删除，那么实际上我们不应该取得这个页，在引入锁对页面进行锁定后就可以有效解决这个问题：

首先可以根据传入的权限值，判定要获取锁的类型，然后定义一个布尔值 lockAcquired，令其初始值为 false，表示还没有获取锁，然后进入一个 while 循环，退出的条件是 lockAcquired 为 true，即成功的获取了锁。在本小节的设计中，没有考虑死锁的问题，即如果产生死锁现象，getPage 方法会一直等待下去，无法跳出循环，解决死锁的策略将于后续小节给出。

```
boolean isShared = (perm==Permissions.READ_ONLY ? true : false);
while(!lockAcquired)
    lockAcquired = lockManager.acquireLock(pid,tid,isShared);
```

1.2.3 relaesePage()

该方法实现较为简单，直接让 LockManager 对指定页面释放锁即可：

```
public void releasePage(TransactionId tid, PageId pid) {
    lockManager.releaseLock(pid,tid);
}
```

1.2.4 holdsLock()

我们依然通过 LockManager 来简化实现，因此先在 LockManager 中定义方法 holdsLock()，具体的实现方式就是如下所示的遍历：

```
public synchronized boolean holdsLock(PageId pid,TransactionId tid){
    if(lockMap.get(pid) == null)
        return false;
    ArrayList<Lock> locks =  lockMap.get(pid);
    for (int i = 0; i < locks.size(); i++)
        if(locks.get(i).tid == tid)
            return true;
    return false;
}
```

然后在外部类的方法中，直接调用内部类 LockManager 的方法即可：

```
public boolean holdsLock(TransactionId tid, PageId p) {
    return lockManager.holdsLock(p,tid);
}
```

第 2 章 Exercise-2 & Exercise-3

Exercise-2 要求我们在整个 SimpleDB 中获取和释放锁，根据实验指导我们要检查几点：

- HeapFile.insertTuple：我们应该通过调用 BufferPool 的 getPage 方法来获取锁，容易知道想实现插入的操作，我们应该获取的是一把独占锁，拥有读写权限，因此构造的临时页对象 page 要传入参数 Permissions.READ_WHITE
- HeapFile.deleteTuple：这也应该是一个同步方法，通过 Database.getBufferPool().getPage() 获取一把独占锁，同上，传入权限应该为 Permissions.READ_WHITE
- HeapFile.iterator：在迭代器中我们也使用了 BufferPool 中的 getPage() 方法，因此满足实验要求

Exercise-3 要求我们在驱逐策略中实现不驱逐脏页的必要逻辑，特别需要注意，如果缓冲池中的所有页面都是脏的，我们应该抛出 DbException 而非随机驱逐一个脏页。

```
1  private synchronized void evictPage() throws DbException {
2     boolean allDirty = true;
3     for(Map.Entry<PageId, Page> entry : pages.entrySet()) {
4        if(entry.getValue().isDirty() != null)
5           continue;
6           try {
7               allDirty = false;
8               flushPage(entry.getKey());
9           } catch (IOException e) {
10              e.printStackTrace();
11          }
12          discardPage(entry.getKey());
13          break;
14       }
15       if(allDirty)
16         throw new DbException("All pages are dirty");
17   }
```

在方法中，我们先维护了一个布尔值 allDirty，用以记录是否所有页均为脏页，然后遍历缓冲池中的页，如果是脏页就跳过，否则标记 allDirty 为 false，表示不是所有页均为脏页，然后通过 Lab-2 实现的 flushPage 方法把这个页刷新到磁盘，最后从缓冲池中删除这个页。如果遍历完没有找到非脏的页，最后抛出 DbException 的异常，标记所有页均为脏页。

第 3 章 Exercise-4

在 SimpleDB 中，TransactionId 对象在每次查询开始时创建。该对象被传递给查询中涉及的每个运算符。查询完成后，将调用 BufferPool 方法 transactionComplete。调用此方法将提交或中止事务，由参数标志 commit 指定。

在该实验中我们要实现两个版本的 transactionComplete 方法，一个接收 commit 而一个不接收，不接受参数的版本应该始终提交，因此可以看作是 commit=true 的普通方法，代码实现如下：

```
public void transactionComplete(TransactionId tid, boolean commit)
    throws IOException {
    if(commit)
        flushPages(tid);
    else{
        for (PageId pid : pages.keySet()) {
            Page page = pages.get(pid);
            if (page.isDirty() == tid) {
                DbFile file =  Database.getCatalog().getDatabaseFile(pid.getTableId());
                Page pageFromDisk = file.readPage(pid);
                pages.put(pid, pageFromDisk);
            }
        }
    }
    for(PageId pid:pages.keySet()){
        if(holdsLock(tid,pid))
            releasePage(tid,pid);
    }
}
```

如果 commit=true，表示要提交事务，那么把 tid 对应的所有页面刷新到磁盘即可。否则表示要中断事务，那么就要把页面恢复到它的磁盘状态来恢复事务所做的任何更改：首先遍历缓冲池中的页面，找到和 tid 对应的页面，然后获取它的磁盘状态 (即从数据库目录中获取)，然后把它的磁盘状态重新 put 进缓冲池，表示事务回滚。最后无论提交还是终止，都应该释放 BufferPool 中保持的关于事务的状态，即锁。

第 4 章 Exercise-5

在本小节中，将实现使用超时策略检测并解决死锁问题，即若在一个规定的时间内没有完成这个事务，就认为其产生了死锁，并回滚这个事务。

在 simpleDB 的设计实现中，有关锁的操作都是基于 BufferPool.getPage() 方法实现的，因此，我们对死锁问题的解决也应该从这个方法入手。我们定义一个事务完成上限时间 limit=5000ms，在获取锁之前记录开始时间，在获取锁的方法内部获取当前时间，如果当前时间-开始时间超过了事务的最大运行时间，我们就认为产生了死锁，并抛出异常。代码实现如下：

```
long start = System.currentTimeMillis();
long limit = 1000;
while(!lockAcquired){
   long now = System.currentTimeMillis();
   if(now-start > limit)
      throw new TransactionAbortedException();
   lockAcquired = lockManager.acquireLock(pid,tid,isShared);
}
```

但上述代码的运行效率极差，如图4.1所示，原因是可能多个线程在同一时刻同时超时并发起新的尝试。如果我们将这个极限时间设置为 1000 加上一个随机值，那么每个线程在获取超时时间时会得到一个略微不同的值，使得它们在发起新的尝试时的时间稍有差异。能够分散并发操作的压力，降低因同时发起大量尝试而导致的资源争夺和竞争情况。采用随机时间得到的运行结果如图4.2所示。

图 4.1: 采用固定时间得到的运行效率

图 4.2: 采用随机时间得到的运行效率

第 5 章 Design alternatives

依赖图机制通过维护锁之间的依赖关系来解决死锁。当系统检测到存在循环依赖的锁关系时，可以主动中断其中一个或多个锁的持有者，打破死锁状态。依赖图机制对比我们之前实现的超时机制的一个优点是它能更准确的识别死锁，并能舍弃一些无谓的等待时间。

碍于时间关系，在此只列出实现思路和算法伪码 (大作业和考试太多了.jpg)，未完成具体测试。

- 首先引入依赖图数据结构：创建一个图来记录锁之间的依赖关系
- 在获取锁之前，检查当前事务是否已经持有其他锁。如果当前事务持有其他锁，并且存在依赖关系，需要中断其中一个锁的持有者，以打破死锁状态
- 在释放锁之前，更新依赖图，移除当前事务与其他锁之间的依赖关系

把我们之前实现的 Lock 和 LockManager 辅助类替换为依赖图类，并修改核心方法 getPage：

```
1  public Page getPage(TransactionId tid, PageId pid, Permissions perm)
2      throws TransactionAbortedException, DbException {
3   boolean isShared = (perm == Permissions.READ_ONLY ? true : false);
4   boolean lockAcquired = false;
5   if (lockManager.hasDeadlock(tid, pid, isShared)) { //检查是否有依赖关系
6       lockManager.breakDeadlock(tid, pid, isShared); //如果有就中断一个持有者
7       throw new TransactionAbortedException();
8   }
9   lockAcquired = lockManager.acquireLock(pid, tid, isShared);
10  if (tid == null)
11      throw new TransactionAbortedException();
12  if (this.pages.containsKey(pid))
13      return pages.get(pid);
14  if (pages.size() >= numPages)
15      evictPage();
16  Page page = Database.getCatalog().getDatabaseFile(pid.getTableId()).readPage(pid);
17  pages.put(pid, page);
18  // 更新依赖图，添加当前事务与获取的锁之间的依赖关系
19  lockManager.addDependency(tid, pid, isShared);
20  return page;
21
```

第 6 章 Commit & Results