南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

数据库系统实验报告

SimpleDB Lab4

姓名：韩佳迅

学号：2012682

年级：2020 级

专业：计算机科学与技术

2022 年 6 月 10 日

摘要

本次实验完成了 SimpleDB 的 Lab4, 并通过了测试。通过完成 Lab4 实现了 SimpleDB 数据库的事务和锁的管理。

本报告中给出了每个 exercise 的具体设计和实现思路、重难点，借此实验和报告理清了数据库的事务和锁。

目录

一、 Lab4 总览

GitLab 仓库地址：https://gitlab.com/hanmaxmax/SimpleDB

Lab4 主要是实现数据库中的事务、锁和并发控制。在我们的代码实现中，需要考虑以下几方面的问题：

(一) Transactions

transaction 是一组数据库操作的集合进行原子执行，要么这些操作全部执行完成，要么一条都不执行。事务需要满足 ACID 特性（原子性、一致性、隔离性、持久性）。

为了满足上述要求，在编写代码时，需要实现 buffer 管理策略：

1. NO STEAL：如果 dirty pages 正被一个没有提交的 transaction 加锁，不要从 bufferpool 淘汰 dirtypages。

2. FORCE：当 transaction 提交时，应该强制对磁盘写入 dirty pages。

（SimpleDB 中不需要考虑崩溃处理，即不需要日志恢复和撤销重做操作）

(二) Locks

在本实验中，实现的是在页的粒度上加锁和解锁。为了对事务和锁进行管理，我们创建了一种数据结构，跟踪每个事务上了什么锁，并管理应该给它上什么锁，锁的类型等。

SimpleDB 的锁有两种，shared lock 和 exclusive lock。它们满足下面几点要求：

（1）在一个 transaction 能够读资源前，必须获得资源的 shared lock。

（2）在一个 transaction 能够写资源前，必须获得资源的 exclusive lock。

（3）多个 transaction 可以同时获得资源的 shared lock。

（4）同一时间，只有一个 transaction 可以获得资源的 exclusive locks。

（5）如果某个事务是唯一一个拥有某资源的 shared lock 的 transaction，则该事务可以将它对该资源的锁更新为 exclusive lock。

（6）如果一个 transaction 获得了它本不该被授权的锁，代码会进入等待状态，等到那个锁恢复可用状态。

(三) 锁的建立与释放、死锁

transactions 应该在访问对象资源前，请求这部分资源的锁，并且在 transaction commits 之后才能开始释放锁。因此，我们需要实现如何加锁、解锁，以及在何时加锁、解锁。

事务可能出现死锁现象，本实验中考虑实现一个简单的超时策略，如果在既定时间内一个 transaction 没有完成，就抛弃这个 transaction。

二、 设计思路及重难点

(一) Exercise 1

exercise 1 需要在 BufferPool 实现加锁和解锁的方法：

（1）修改 getPage()，在 getpage 之前实现对该 page 的加锁；

（2）完成 releasePage()，对 page 解锁；

（3）完成 holdsLock()，判断 page p 是否已经被事务 t 上锁。

实现思路：

- **在 BufferPool 中新建一个 class Lock，表示锁**

类的结构：

成员属性：

TransactionId tid：事务 id，用于表示该锁是哪个事务上的

int lockType：锁的类型（0：shared clock 1：exclusive clock）

- **在 BufferPool 中再新建一个 class ControlForLock，用于对锁进行管理控制**

类的结构：

成员属性：

id2lock：从 PageId 到 Vector<Lock> 的哈希，用于映射 page 和它上面所有锁的列表

成员变量：

1. **getLock(PageId pid, TransactionId tid, int lockType)**

类型：public synchronized boolean，其中 synchronized 是 java 的关键字，是一种同步锁，一个线程调用方法时，其它线程试图使用该方法的线程将被阻塞

参数：pid，标识是哪个 page 的锁；tid 标识哪个事务；LockType 标识锁的类型

返回：上锁成功 or 失败

思路：如果该页原本就没有锁，则根据 tid 和 lockType 新建一个锁给它，并给它新建一个 vector；否则遍历现有的锁列表，找到在该事务上已经有的锁，其实对应四种情况：1. 请求 shared，已有 ex；2. 请求 ex，已有 shared；3. 请求 ex，已有 ex；4. 请求 shared，已有 shared。其中 3 和 4 说明不需要再上锁了，直接返回 true，1 这种情况已经有了 ex 锁，不需要再上 shared 锁，也返回 true，而 2 的情况需要判断该页是否只有这一把锁，若是，则升级，否则加锁失败。

2. **unLock(PageId pid,TransactionId tid)**

参数：pid，标识是哪个 page 的锁；tid 标识是哪个事务

返回：解锁成功 or 失败

思路：先判断该页的所列表是否为空，若不为空，则遍历锁列表，找到 tid 在 pid 中的锁，并把它从锁列表移除。若列表是空的或找不到对应的锁，则解锁失败。

3. **holdsLock(PageId pid,TransactionId tid)**

参数：pid，标识是哪个 page 的锁；tid 标识是哪个事务

返回：事务 tid 是否对页 pid 上锁的 bool 值

思路：遍历锁列表即可。

- **修改 getPage()**

思路：

（1）先在 BufferPool 类中添加一个 ControlForLock 成员变量 controlforlock

（2）在原有代码的前面添加上锁的代码，调用 controlforlock 的 getLock 函数，根据 getLock 的参数 Permissions perm 决定上锁类型 locktype，若 perm 是 READ_ONLY，则上 shared 锁，否则上 exclusive 锁。

- **完成 releasePage(TransactionId tid, PageId pid)**

直接调用 controlforlock 的 unlock 函数进行解锁。

- **完成 holdsLock(TransactionId tid, PageId p)**

直接调用 controlforlock 的 holdsLock 函数即可。

重难点:

需要自己设计 Lock 类及其管理类，重点在于对类的实现以及设计。参考之前对 BufferPool 的散列表的设计思路，在锁的管理时也建立散列表，实现从 pageId 到其锁列表的映射。并据此实现加锁、解锁等控制措施。

(二) Exercise 2

exercise 2 需要检查并实现在数据库运行的每个环节和步骤都考虑加锁和解锁，具体体现在以下方面:

(1) 遍历时：完成 Lab4 之后明白了为什么当时在实现 HeapFile.iterator() 需要调用 BufferPool.getPage()，这是因为 getPage() 中封装了加锁环节，这样在迭代器中调用 getPage() 自然而然就实现了加锁。

(2)插入和删除时:同上述遍历,因为在插入删除之前也需要加锁,因此,HeapFile.insertTuple() 和 HeapFile.deleteTuple() 在实现时也需要调用 getPage()。

加锁与解锁的情景:

(1) 向 HeapFile 中添加新页的时候：当 insert 的时候没有空位置了，需要新建一个 page，对新建的 page（空 page），要先把空的它写到磁盘中，等到之后的事务提交 flushPage 的时候，写入磁盘的才是有数据的该 page。

(2) insertTuple 和 deleteTuple 需要调用 getpage()，为了避免与其他事务的竞争，要加 exclusive 锁，因此，在调用 getPage 的时候，permission 要选择 READ_WRITE;

(3) 当遍历到某个 page 没有空 slot 时，需要释放该页的锁。因为我们后续都不会使用到这个 page 了，所以这么做不会违反两端锁协议，也可以让其他事务能够访问该 page 了。

实现思路:

- **HeapFile.insertTuple()**

代码思路:

整体思路跟之前一样：遍历整张表的数据页，寻找空 slot，找到了就在空 slot 所在 page 中调用 HeapPage 的 insertTuple 方法进行插入；若没有，则新建一个空页。

不同之处:

1. 在新建空页时，先把该空页写到磁盘中，使得磁盘中，该页的那部分为空，等到它后面插入了数据并且该事务 commit 后，再统一调用 flushpage 将所有更新后的页写入磁盘。

2. 遍历时若发现某页没有空位置，则直接给它解锁。因为我们后续都不会使用到这个 page 了，所以这么做不会违反两端锁协议，也可以让其他事务能够访问该 page 了。

3. 插入时的 Permission 权限要选择 READ_WRITE。

- **HeapFile.deleteTuple()**

代码思路：与之前相同，只需要注意 Permission 权限也要选择 READ_WRITE。

(三) Exercise 3

exercise 3 需要实现 NO STRAL 策略。事务带来的修改只会在 commits 之后写入磁盘, 这意味着可以通过丢弃 dirty pages 并将磁盘的内容读取到这些 dirty pages 来抛弃一次 transaction。

在这种情形下，我们不能淘汰 dirty pages，这样的策略称为 NO STEAL。在实现时，需要修改以下几方面:

（1）事务完成（commit）或抛弃（abort）时，对 dirty pages 的处理，这部分已经在上面完成了。

（2）还需要注意到的一个点是 evictPage 方法。在我们之前的实现过程中，只是随意驱逐出了一个页面。但事实上，我们驱逐出去的这个页面不能是 dirty page，否则就会违背 NO STEAL 原则。

实现思路：

- **BufferPool.evictPage()**

代码思路:

（1）由于在驱逐的时候需要找到最“老”的 page，因此我们需要创建能够记录页面 age 的类成员。

在 BufferPool 中添加成员变量 *int age* 和 *ConcurrentHashMap<PageId,Integer> id2age*，用于记录每页的 age。

（2）在 getPage() 中，维护每页的 page：每次将某 page 放入缓存的时候，同时将此时缓存的 age 存入该 page 对应的 id2age 哈希中，这里的 age 相当于时钟，表示该 page 是在 buffer 的 age 时存入的，然后更新 age，使得 age++。因此，age 越小，证明该页存入缓存越早，越应该被驱逐。

（3）在 evictPage() 中，通过遍历 id2age 哈希表，找到 age 最下的非 dirty page，然后将它从缓冲池中驱逐掉即可。若没找到这样的一个 page，则抛出异常。

重难点:

需要考虑如何设置驱逐策略。本程序选择了通过记录每个页放入缓存的时间来决定驱逐出最老（age 最小）的 page。（这里的设置需要转个弯，age 相当于一个时钟）

(四) Exercise 4

exercise 4 需要实现事务完成的操作。

在 SimpleDB 中，一个 TransactionId 对象是在每个 query 开始时创建的。这个对象被传到这个 query 的每个 operators 上，当 query 完成的时候，BufferPool 中的 transactionComplete 方法被调用。

在 commit 或者 abort 时 transaction 的时候调用这个方法。在执行的任何时候,一个 operator 都可能会抛出 TransactionAbortedException 错误，这说明一个内部错误或者死锁已经发生了。

实现思路：

- **BufferPool.transactionComplete()**

代码思路：commit 或 abort 给定事务，释放与事务关联的所有锁。

首先，根据传来的参数 commit 判断，若 commit 为 true，调用 flushPages(tid)，将该事务的所有 dirty pages 都写入磁盘；否则要恢复该事务处理过的页。

然后，用 holdsLock 遍历所有该 tid 锁住对应的 page，调用 releaseLock 为其解锁。

注：恢复处理的思路如下。

遍历该事务 tid 处理过的 pages，记遍历到的页的 id 为 pid，用 pid 从磁盘上取该页，再把磁盘上的该页放到缓冲池里。相当于，将没有改动过的、存在磁盘中的该 page，再从磁盘中拿出来替换缓冲池中被改过的页。

（五） Exercise 5

exercise 5 需要实现死锁相关操作。

对死锁的检测：在本实验中，对死锁的检测使用了超时策略：如果在既定时间内一个 transaction 没有完成，就抛弃这个 transaction。

实现思路：

- **BufferPool.getPage()**

代码思路：

在原来的代码前面添加时间检测代码，使用 *System.currentTimeMillis()* 获取时间。若多次上锁不能成功，就会导致时间超时，然后抛出异常，事务中断。

重难点：

整个 lab4 的重难点在于对管理锁的结构的设计，以及对每个事务和锁的追踪。再实现过程中，由于不像前几个 lab，每个 exercise 只用实现固定的类或函数，所以 lab4 的实现需要更强的逻辑思路来理清事务的执行过程，否则就会很混乱或者丢掉什么。

三、 Git Commit History

GitLab 仓库地址：https://gitlab.com/hanmaxmax/SimpleDB

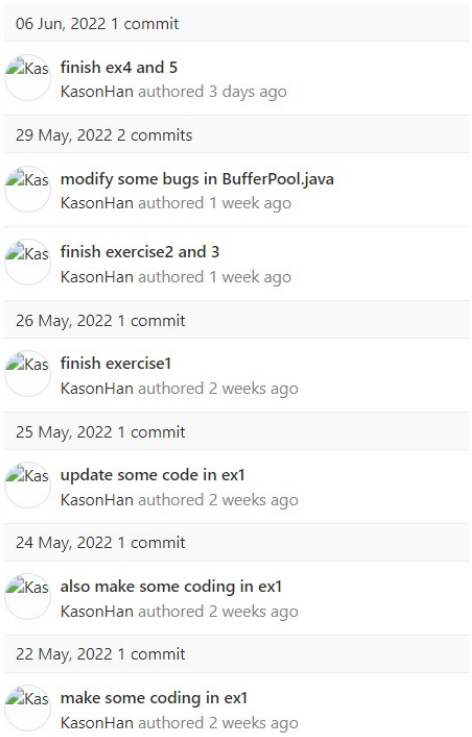

图 1: 提交记录