哈尔滨工业大学

<<数据库系统>>

实验报告三

(2023 年度春季学期)

姓名:	刘天瑞
学号:	7203610121
学院:	计算学部
教师:	李东博

实验三（project2）

一、实验目的

1. 掌握数据库管理系统的存储管理器的工作原理；
2. 掌握数据库管理系统的缓冲区管理器的工作原理；
3. 使用 C++面向对象程序设计方法实现缓冲区管理器。

二、实验环境

该实验是在 VMware 虚拟机的 Linux 系统下来实现的。其版本为：Linux version5.15.0-69-generic(Ubuntu20.04)gccversion9.4.0(Ubuntu9.4.0-1ubuntu1~20.04.1) GNU Make 4.2.1

三、实验过程及结果

在本次整个数据库实验所给出的源代码中，我只需要补充完整众多代码中的 buffer.cpp，实现缓冲区管理器类就可以了。同时也可以在主函数 main.cpp 中加入更多的测试用例。buffer.cpp 的具体实现过程如下所示：

其中的 BufMgr(const int bufs)和~BufMgr()已经给出，他们的作用是为缓冲池分配一个包含 bufs 个页面的数组，并为缓冲池的 BufDesc 表分配内存。当缓冲池的内存被分配后，缓冲池中所有页框的状态就被设置为初始状态。接下来将记录缓冲池中当前存储的页面的哈希表被初始化为空。除此之外，还需将缓冲池中所有的脏页重新写回磁盘，最后释放缓冲池、BufDesc 表以及哈希表所占用的内存。

1. void advanceClock()的实现：

由于该函数的作用是顺时针旋转时钟算法中的表针，将其指向缓冲池中下一个页框。因此只需要对页框序号 clockHand 不断进行加一操作，并且对总页框数进行取模操作即可。具体的函数代码实现方法如下图 1 所示：

```
/*3.顺时针旋转时钟算法中的表针，将其指向缓冲池中下一个页框。*/
void BufMgr::advanceClock()
{
    clockHand = (clockHand + 1) % numBufs;//对页框编号不断加一，对总页框数进行取模
}
```

图 1

2. void allocBuf(FrameId& frame)的实现：

首先判断该页框的 valid 值，若 valid=false，则说明该页框为空，可以使用，便将页框序号返回即可跳出循环，结束查找；若 valid=1，继续判断该页框的 refbit 值，若该值为 1，说明该页框在最近时间内有被读取，因此将其 refbit 值设置为 0 后，进入下一次循环（将页框序号进行加一后继续重复本次判断过程）。如果其 refbit 值为 0，则继续判断其是否被固定（pinned）。如果其已经被固定（pinned），则将记录页框被固定（pinned）的变量值加一。若该变量值已经达到页框总数，

则表示所有的页框都处于（固定）pinned 状态，此时则会报错。如果该页框并未被（固定）pinned，则要判断该页框有没有处于被重写过但没写入磁盘的状态（即判断 dirty 值是否为 1）。若 dirty 值为 1，则说明该页框中页面是脏的，故要将其先写回磁盘后再将 dirty 值更改为 0；若 dirty 值为 0 则可以将该页框作为要写入的页框（即使该页框为非空）。当然，由于该页框含有有效页面，所以必须将该页面从哈希表中删除。最后，分配的页框编号通过参数 frame 返回。

其实这个函数的具体实现思路就是根据实验指导书中的那个程序框图。只需要将它看懂并且按照程序框图将代码写出来就成功编译了。具体的函数代码步骤如下图 3 所示：

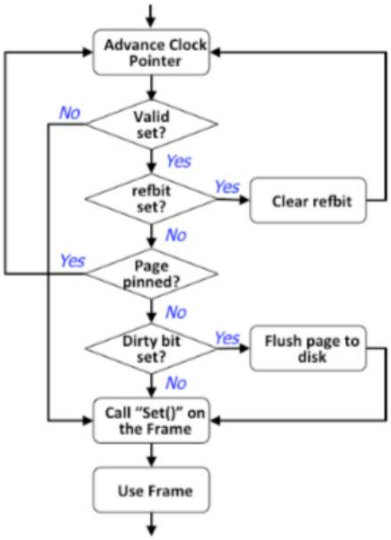

图 2

```
/*4.使用时钟算法分配一个空闲页框。如果页框中的页面是脏的，则需要将脏页先写回磁盘。如果缓冲池中所有页框
都被固定（pinned）了，则抛出BufferExceededException异常。如果被分配的页框中包含一个有效页面，则必须将
该页面从哈希表中删除。最后分配的页框编号通过参数frame返回*/
void BufMgr::allocBuf(FrameId & frame)
{//首先判断是否为有效空闲页，并且是否所有页面都被已被固定（pinned）
    int Count_pinned = 0;
    for (int page = 0; page < numBufs; page++)
    {
        if (bufDescTable[page].pinCnt > 0)
        {
            Count_pinned++;
        }
    }
    if (Count_pinned == numBufs)
    {
        throw BufferExceededException();
    }
    while (true)//然后进行分配
    {
    advanceClock();

    if (!bufDescTable[clockHand].valid)  //当找到有效空闲页时；；，直接分配使用
    {
     frame = clockHand;
     return;
    }

    if (bufDescTable[clockHand].refbit)  //当表针指向每一个页框时，使用时钟算法检查该页框的refbit值，设置为0
    {
     bufDescTable[clockHand].refbit = false;
     continue;
    }

    if (bufDescTable[clockHand].dirty)  //当遇到脏页时，先写回磁盘然后再重新设置为0
    {
     bufDescTable[clockHand].file->writePage(bufPool[clockHand]);
     bufDescTable[clockHand].dirty = false;
    }
    try
    {
     hashTable->remove(bufDescTable[clockHand].file, bufDescTable[clockHand].pageNo);  //如果当前页框含有有效页面，则将其从哈希表中删除
    }
    catch (HashNotFoundException &)//分配的页框编号通过参数frame返回
    {
    }
    bufDescTable[clockHand].Clear();
    frame = clockHand;
    break;
    }
}
```

图 3

3. void readPage(File* file, const PageId PageNo, Page*& page) 的实现：

根据实验指导书提示，首先调用哈希表的 lookup()方法检查待读取的页面(file, PageNo)是否已经在缓冲池中。如果页面在缓冲池中，那么在这种情况下，需要将页框的 refbit 设置为 true，并将 pinCnt 值加一。最后，通过参数 page 返回指向该页框的指针。如果页面不在缓冲池中，这种情况下需要调用 allocBuf()方法分配一个空闲的页框。然后，调用 file->readPage()方法将页面从磁盘读入刚刚分配的空闲页框。接下来，将该页面插入到哈希表中，并调用 Set()方法来正确设置页框的状态，Set()会将页面的 pinCnt 值设置为 1。最后，通过参数 page 返回指向该页框的指针。具体的函数代码实现方法如下图 4 所示：

```
/*5.调用哈希表的lookup（）方法检查待读取页面（file，pageNo）是否已经在缓冲池中。如果该页面已经在缓冲池中
通过参数page返回指向该页面所在的页框指针；如果该页面不在缓冲池中。则哈希表的lookup（）方法会抛出HashNotFoundException
异常。*/
void BufMgr::readPage(File* file, const PageId pageNo, Page*& page)
{
    FrameId frame;  //首先判断是缓存池中的哪个页面
    try           //若要访问的页面在缓存池中
    {
        hashTable->lookup(file, pageNo, frame);
        bufDescTable[frame].pinCnt++;
        bufDescTable[frame].refbit = true;
        page = bufPool + frame;
    }
    catch(HashNotFoundException e)  //若要访问的页面不在缓存池中
    {
        allocBuf(frame);  //在缓存池中分配一个新的空闲页面
        bufPool[frame] = file->readPage(pageNo);  //从磁盘中读入刚分配的空闲页面
        hashTable->insert(file, pageNo, frame);  //在哈希表中插入表项
        bufDescTable[frame].Set(file, pageNo);  //设置页框状态
        page = bufPool + frame;
    }
}
```

图 4

4. void unPinPage(File* file, const PageId PageNo, const bool dirty)的实现：

首先通过调用哈希表的 lookup()方法来判断检查待读取的页面是否在缓冲池中，如果在，就将该页面所在页框的 pinCnt 值减一。如果参数 dirty 等于 true，则需要将页框的 dirty 设置为 false。如果 pinCnt 值已经是 0 了，则需要抛出 PAGENOTPINNED 异常。具体的函数代码实现方法如下图 5 所示：

```
/*6.将缓冲区中包含（file，PageNo）表示的页面所在的页框的pinCnt值减1.如果参数dirty等于true，则
将页框的dirty位置为true。如果pinCnt的值已经是0，抛出PAGENOTPINNED异常。如果该页面不在哈希表
中，什么都不用做*/
void BufMgr::unPinPage(File* file, const PageId pageNo, const bool dirty)
{
    FrameId frame;  //首先判断是缓存池中的哪个页面
    try           //若要访问的页面在缓存池中
    {
        hashTable->lookup(file, pageNo, frame);
        if (bufDescTable[frame].pinCnt == 0)  //pinCnt为0则抛出PAGENOTPINNED异常
        {
            throw PageNotPinnedException(bufDescTable[frame].file->filename(), bufDescTable[frame].pageNo, frame);
        }
        bufDescTable[frame].pinCnt--;  //将pinCnt减一
        if (dirty)
        {
            bufDescTable[frame].dirty = true;
        }
    }
    catch(HashNotFoundException e)  //若要访问的页面不在缓存池中
    {
        return;
    }
}
```

图 5

5. void allocPage(File* file, PageId& PageNo, Page*& page)的实现：

恰如实验指导书所说，首先调用 file->allocatePage()方法在 file 文件中分配一个空闲页面，file->allocatePage()返回这个新分配的页面。然后，调用 allocBuf()方法在缓冲区中分配一个空闲的页框。接下来，在哈希表中插入一条项目，并调用 Set()方法正确设置页框的状态。该方法既通过 pageNo 参数返回一个人新分配

的页面的编号，还通过 page 参数来返回指向缓冲池中包含该页面的页框的指针。该指针即为 bufPool + frame。具体的函数代码实现方法如下图 6 所示：

```
/*8.首先调用file->allocatePage（）方法在file文件中分配一个空闲页面，file->allocatePage（）返回这个新的分配页面，然后调用allocBuf（）方法在缓冲区分配一个空闲的页框。接下来在哈希表中插入一条项目，并调用set（）方法正确设置页框状态。该方法既通过pageNo参数返回新分配的页面的页号，还通过page参数返回指向缓冲池中包含该页面的页框的指针。*/
void BufMgr::allocPage(File* file, PageId &pageNo, Page*& page)
{
    FrameId frame;
    Page p = file->allocatePage();  //分配一个新的空闲页面
    allocBuf(frame);  //分配一个新的空闲页框
    bufPool[frame] = p;
    pageNo = p.page_number();
    hashTable->insert(file, pageNo, frame);  //插入一条哈希表条目
    bufDescTable[frame].Set(file, pageNo);  //调用set（）方法正确设置页框状态
    page = bufPool + frame;
}
```

图 6

6. void disposePage(File* file, const PageId pageNo)的实现：

判断页号为 pageNo 的页面是否在缓冲池中，如果在，则将该页面所在的页框清空并从哈希表中删除该页面。具体的函数代码实现方法如下图 7 所示：

```
/*9.该方法从文件file中删除页号为pageNo的页面。删除之前如果该页面在缓冲池中，需要将该页面所在的页框清空并从哈希表中删除该页面*/
void BufMgr::disposePage(File* file, const PageId PageNo)
{
    FrameId frame;
    try
    {
        //判断页号为pageNo的页面是否在缓存池中
        hashTable->lookup(file, PageNo, frame);
        hashTable->remove(file, PageNo);
        bufDescTable[frame].Clear();
    }
    catch(HashNotFoundException e)
    {
    }
    file->deletePage(PageNo);
}
```

图 7

7. void flushFile(File* file)的实现：

扫描 bufTable，检索缓冲区中所有属于文件 file 的页面。首先判断该页面是否有效，除此之外该页面是否被固定（pinned）住。如果该页面无效或者被固定（pinned）住，则抛出 BadBufferException 异常。这之后再判断该页面 dirty 是否为 1，若为 1 则将该页面内容写回磁盘后将 dirty 值变为 false。最后将页面从哈希表中删除后，调用 BufDesc 类的 Clear()方法将页框的状态进行重置。具体的函数代码实现方法如下图 8 所示：

```
/*7.扫描bufTable，检索缓冲区中所有属于文件file的页面。对于每个检索到的页面进行如下操作：
 （a）如果文件是脏的，则调用file->writePage（）将页面写回磁盘，并将dirty值为false；
 （b）将页面从哈希表中删除；
 （b）调用BufDesc类的Clear（）方法将页框状态重置。
如果文件file的某些页面被固定（pinned）住，抛出BadBufferException异常。
检索到文件file的某个无效页面，抛出BadBufferException异常*/
void BufMgr::flushFile(const File* file)
{
    for (FrameId page = 0; page < numBufs; page++)
    {
        if (bufDescTable[page].file == file)
        {
            if (bufDescTable[page].pinCnt > 0)
            {
                throw PagePinnedException(bufDescTable[page].file->filename(), bufDescTable[page].pageNo, page);

            }
            if (bufDescTable[page].valid == false)  //如果页面无效或者被固定住，则抛出BadBufferException异常
            {
                throw BadBufferException(page, bufDescTable[page].dirty, bufDescTable[page].valid, bufDescTable[page].refbit);
            }
            if (bufDescTable[page].dirty)  //如果是脏页要写回
            {
                bufDescTable[page].file->writePage(bufPool[page]);
                bufDescTable[page].dirty = false;
            }
            hashTable->remove(bufDescTable[page].file, bufDescTable[page].pageNo);  //将页面从哈希表中删除
            bufDescTable[page].Clear();  //重置页框状态
        }

    }
}
```

图 8

实验结果先以 test4 为例：

```
void test4()
{

    bufMgr->allocPage(file4ptr, i, page);
    bufMgr->unPinPage(file4ptr, i, true);
    try
    {
        bufMgr->unPinPage(file4ptr, i, false);
        PRINT_ERROR("ERROR :: Page is already unpinned. Exception should have been thrown before execution reaches this point.");
    }
    catch(PageNotPinnedException e)
    {
    }

    std::cout << "Test 4 passed" << "\n";

}
```

图 9

当正在执行 unPinPage()这个函数时，首先判断 file4ptr 是否在对应的页框中，如果不在，则直接抛出异常。如果在对应页框中，则需要继续判断其 pinCnt 值，如果其 pinCnt 值大于 0，则说明此时有应用正在使用它，结束使用的时候利用 unPin 来使 pinCnt 减一；如果 pinCnt 小于等于 0，则不能进行 unPin 操作，然后抛出异常。在这个 test4 中，file4ptr 的 pinCnt 值等于 0，故不能进行 unPin 操作，所以最后会抛出异常。

然后，在 BufMgr 文件夹下打开终端，输入 make 执行 Makefile，再执行 src 文件夹下的 badgerdb_main 文件来查看测试结果。对所有的测试样例都通过的结果如下图 10 所示：

```
hiterltr@ubuntu:~/BufMgr/src$ ./badgerdb_main
Third page has a new record: world!

Test 1 passed
Test 2 passed
Test 3 passed
Test 4 passed
Test 5 passed
Test 6 passed

Passed all tests.
```

图 10

最后，在本次实验给定的 6 个测试样例之外，我还新增了 1 组测试样例即 test7，它用于测试 pageNo 是否过大从而造成 InvalidPageException 异常的情况。其具体测试代码如下图 11 所示：

```
void test7()
{
    //pageNo过大，造成InvalidPageException
    try
    {
        bufMgr->readPage(file1ptr, 200, page);
        PRINT_ERROR("ERROR :: PageNo out of range, Exception should have been thrown before execution reaches this point.:");
    }
    catch(InvalidPageException e)
    {
    }

    std :: cout << "Test 7 passed" << "\n";
```

图 11

为方便起见，我还在 BufMgr 文件夹下新建了 run.sh 脚本，执行该脚本文件即可同时完成 make 编译与执行 badgerdb_main 的功能（实验并未要求故省略）。所有的 7 个测试样例包括新增的样例都通过的结果如下图 12 所示：

```
hiterltr@ubuntu:~/BufMgr$ ./src/badgerdb_main
Third page has a new record: world!

Test 1 passed
Test 2 passed
Test 3 passed
Test 4 passed
Test 5 passed
Test 6 passed
Test 7 passed

Passed all tests.
```

图 12

四、实验心得

列举遇到并解决的问题等。

答：本次数据库实验编写过程我做起来还算比较轻松，并未遇到很困难无法

解决的问题。实验要求很清晰且思路也很明确。实验指导书的讲解很详细也很透彻。

通过这次实验我对缓冲区管理这方面的知识有了更加深刻的理解，对知识点也有了更加牢固地掌握。相当于是在实验中不断地复习学过的知识了。