实验题目	缓冲区管理器实现			实验日期	2022 年 4 月 3 日
班级	1903104	学号	1190201421	姓名	张瑞

CS33503 数据库系统实验

实验检查记录

实验结果的正确性(60%)		表达能力(10%)	
实验过程的规范性(10%)		实验报告(20%)	
加分(5%)		总成绩(100%)	

实验报告

一、实验目的（介绍实验目的）

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（介绍实验使用的硬件设备、软件系统、开发工具等）

1. 硬件设备：Intel(R) Core(TM) i5-7200U CPU @ 2.50GHz 2.71 GHz; 8GB RAM
2. 软件系统：Windows 10
3. 开发工具：8.0.28 MySQL Community Server – GPL

三、实验过程（介绍实验过程、设计方案、实现方法、实验结果等）

1. 阅读源代码

本次实验需要实现一个缓冲区管理器，主要由 3 个类实现，分别为 BufMgr、BufDesc 和 BufHashTbl，缓冲区管理器的底层涉及到文件类（File）和页面类（Page）。在实验源代码中，已经给出了 File、Page 和各种异常的定义与实现，BufMgr、BufDesc 和 BufHashTbl 的全部定义和大部分实现也都已经给出，仅需实现 BufMgr 中的几个方法。

为了实现缓冲区管理器，下面对 BufMgr、BufDesc 和 BufHashTbl 这几个类进行分析。

(a) BufHashTbl：

该类采用链表法实现了一个哈希表，能将一个文件句柄（File*）和页号（PageId）的组合与一个缓冲池中的页框号（FrameId）进行映射，同时提供了插入、查找、删除等方法。

其定义如下：

```
/**
* @brief Hash table class to keep track of pages in the buffer pool
*
* @warning This class is not threadsafe.
*/
class BufHashTbl
{
 private:
  int HTSIZE;  // Size of Hash Table
  hashBucket**  ht;  // Actual Hash table object
  // returns hash value between 0 and HTSIZE-1 computed using file and pageNo
  int    hash(const File* file, const PageId pageNo);
```

```
 public:
  BufHashTbl(const int htSize);  // constructor
  ~BufHashTbl(); // destructor

  // Insert entry into hash table mapping (file, pageNo) to frameNo.
  void insert(const File* file, const PageId pageNo, const FrameId frameNo);
  // Check if (file, pageNo) is currently in the buffer pool (ie. in the hash table).
  void lookup(const File* file, const PageId pageNo, FrameId &frameNo);
  // Delete entry (file,pageNo) from hash table.
  void remove(const File* file, const PageId pageNo);
};
```

其中 hashBucket 定义如下：

```
/**
* @brief Declarations for buffer pool hash table
*/
struct hashBucket {
        File *file;  // pointer a file object (more on this below)
        PageId pageNo;  // page number within a file
        FrameId frameNo;  // frame number of page in the buffer pool
        hashBucket*   next;  // Next node in the hash table
};
```

(b) BufDesc：

该类用于记录缓冲池中每个页框的状态，BufMgr 是它的友元。该类能记录当前页框的页框号、对应的文件句柄和页号、引用次数、是否被修改、是否有效和“最近”是否被访问等信息，并提供设置、清除和打印信息等方法。

其定义如下：

```
/**
* @brief Class for maintaining information about buffer pool frames
*/
class BufDesc {

  friend class BufMgr;

 private:
  File* file;  // Pointer to file to which corresponding frame is assigned
  PageId pageNo;  // Page within file to which corresponding frame is assigned
  FrameId frameNo;  // Frame number of the frame, in the buffer pool, being used
  int pinCnt;  // Number of times this page has been pinned
  bool dirty;  // True if page is dirty;  false otherwise
  bool valid;  // True if page is valid
  bool refbit;  // Has this buffer frame been reference recently

  // Initialize buffer frame for a new user
  void Clear();
  // Set values of member variables corresponding to assignment of frame to a page in the file.
  void Set(File* filePtr, PageId pageNum);
  // Print values of member variables
  void Print();
  BufDesc();  // Constructor of BufDesc class
};
```

(c) BufMgr：

该类为缓冲区管理器的核心，能记录时钟算法当前指向的页框、缓冲区大小、缓冲区各个页框状态、缓冲区中页面所组成的哈希表等信息，并提供读取页面、解除页面固定、分配页面、删除页面、清除缓冲区等方法。

其定义如下：

```
/**
* @brief The central class which manages the buffer pool including frame allocation and deallocation to pages
in the file
*/
class BufMgr
{
 private:
  FrameId clockHand;  // Current position of clockhand in our buffer pool
  std::uint32_t numBufs;  // Number of frames in the buffer pool
  BufHashTbl *hashTable;  // Hash table mapping (File, page) to frame
  BufDesc *bufDescTable;  // BufDesc objects, one per frame
  BufStats bufStats;  // Maintains Buffer pool usage statistics
  void advanceClock();  // Advance clock to next frame in the buffer pool
  void allocBuf(FrameId & frame);  // Allocate a free frame.

 public:
  Page* bufPool;  // Actual buffer pool from which frames are allocated
  BufMgr(std::uint32_t bufs);  // Constructor of BufMgr class
  ~BufMgr();  // Destructor of BufMgr class

  // Reads the given page from the file into a frame and returns the pointer to page.
  void readPage(File* file, const PageId PageNo, Page*& page);
  // Unpin a page from memory since it is no longer required for it to remain in memory.
  void unPinPage(File* file, const PageId PageNo, const bool dirty);
  // Allocates a new, empty page in the file and returns the Page object.
  void allocPage(File* file, PageId &PageNo, Page*& page);
  // Writes out all dirty pages of the file to disk.
  void flushFile(const File* file);
  // Delete page from file and also from buffer pool if present.
  void disposePage(File* file, const PageId PageNo);
  // Print member variable values.
  void  printSelf();
  // Get buffer pool usage statistics
  BufStats & getBufStats();
  // Clear buffer pool usage statistics
  void clearBufStats();
};
```

2. 实现 buffer.cpp 中的方法

(a) ~BufMgr()：

该方法先将缓冲池中所有脏页写回磁盘，然后释放缓冲池、BufDesc 表和哈希表占用的内存。

(b) void advanceClock()：

该方法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。

(c) void allocBuf(FrameId& frame)：

该方法使用时钟算法分配一个空闲页框。如果页框状态 valid 为 false，则可直接分配；如果 refbit 被设为 true，则需要将其改为 false 并寻找下个页框；如果页框被固定住了，则将固定住的页框数加一，当缓冲池中所有页框都被固定了的时候，抛出 BufferExceededException 异常；如果页框中的页面是脏的，则需要将脏页先写回磁盘。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，分配的页框的编号通过参数 frame 返回。

(d) void readPage(File* file, const PageId PageNo, Page*& page)：

该方法将一个页面读入缓冲池中。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。

如果该页面已经在缓冲池中，先将页框的 refbit 置为 true，并将 pinCnt 加 1，最后通过参数 page 返回指向该页框的指针。

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。在这种情况下，捕捉该异常，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()方法正确设置页框的状态。最后通过参数 page 返回指向该页框的指针。

(e) void unPinPage(File* file, const PageId PageNo, const bool dirty)：

该方法将一个在缓冲区中的页面解除固定。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。

如果该页面已经在缓冲池中，将其所在页框的 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 值已经是 0，则抛出 PAGENOTPINNED 异常。

如果该页面不在缓冲池中，则什么都不用做。

(f) void allocPage(File* file, PageId& PageNo, Page*& page)：

该方法分配一个页面并将其放入缓冲池中。首先调用 file->allocatePage()方法在 file 文件中分配一个空闲页面。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。

(g) void disposePage(File* file, const PageId pageNo)：

该方法从文件 file 中删除页号为 pageNo 的页面。在删除之前，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。

(h) void flushFile(File* file)：

该方法清除缓冲池中所有属于文件 file 的页面。对每个检索到的属于文件 file 的页面，如果是无效页，则抛出 BadBufferException 异常；如果被固定住，则抛出 PagePinnedException 异常；其余页面进行如下操作：(a) 如果页面是脏的，则调用 file->writePage()将页面写回磁盘，并将 dirty 位置为 false；(b) 将页面从哈希表中删除；(c) 调用 BufDesc 类的 Clear()方法将页框的状态进行重置。

3. 在 main.cpp 中添加测试用例

实验源代码中包含 6 个测试用例：test1 将内容写入页面后再读出来，测试前后内容是否一致；test2 将多个文件交叉进行写与读的操作，测试前后内容是否一致；test3 读一个不存在的页，测试是否会报错；test4 对一个未被固定住的页面进行解除固定操作，测试是否会报错；test5 在缓冲池已满的情况下申请分配一个页面并放入缓冲池，测试是否会报错；test6 对缓冲池中尚未解除固定的页面进行清除，测试是否会报错。

为了使测试更充分，下面添加 3 个测试用例：

Test7 读取一个不在缓冲池中的页面，待其进入缓冲池中后，再次读取，测试其两次读取是否都能正常执行，且 pinCnt 值为 2，代码如下：

```
void test7()
{
        //reading a page in the buffer pool. The pinCnt should increase
        bufMgr->readPage(file1ptr, 1, page);
        bufMgr->readPage(file1ptr, 1, page);
        bufMgr->unPinPage(file1ptr, 1, false);
        bufMgr->unPinPage(file1ptr, 1, false);
        try
        {
                bufMgr->unPinPage(file1ptr, 1, false);
                PRINT_ERROR("ERROR :: Page is already unpinned. Exception should have been thrown before
execution reaches this point.");
        }
        catch(PageNotPinnedException e)
        {
        }

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

Test8 读取一个已被删除的页面，测试是否会报错，代码如下：

```
void test8()
{
        //reading a disposed page. Should generate an error
        bufMgr->readPage(file1ptr, 100, page);
        bufMgr->disposePage(file1ptr, 100);
        try
        {
                bufMgr->readPage(file1ptr, 100, page);
                PRINT_ERROR("ERROR :: Page should not exist. Exception should have been thrown before execution
reaches this point.");
        }
        catch(InvalidPageException e)
        {
        }

        std::cout << "Test 8 passed" << "\n";
}
```

Test9 在缓冲池中有多个文件对应页面且缓冲池已满时，分配页面，测试是否会报错，清除其中一个文件的页面，再分配页面，测试页面分配是否成功，且页面清除数量是否正确，代码如下：

```
void test9()
{
        //flushing one file with another file left.
        for (i = 1; i <= num/2; i++) {
                bufMgr->readPage(file1ptr, i, page);
                bufMgr->readPage(file5ptr, i, page);
        }

        PageId tmp;
        try
        {
                bufMgr->allocPage(file4ptr, tmp, page);
                PRINT_ERROR("ERROR :: No more frames left for allocation. Exception should have been thrown
before execution reaches this point.");
        }
        catch(BufferExceededException e)
        {
        }

        for (i = 1; i <= num/2; i++)
                bufMgr->unPinPage(file5ptr, i, false);
        bufMgr->flushFile(file5ptr);

        for (i = 1; i <= num/2; i++) {
                bufMgr->allocPage(file4ptr, tmp, page);
        }

        try
        {
                bufMgr->allocPage(file4ptr, tmp, page);
                PRINT_ERROR("ERROR :: No more frames left for allocation. Exception should have been thrown
before execution reaches this point.");
        }
        catch(BufferExceededException e)
        {
        }

        std::cout << "Test 9 passed" << "\n";
}
```

4. 实验结果

首先运行 make 编译项目，然后运行./badgerdb_main，结果如下：

实验题目	缓冲区管理器实现			实验日期	2022 年 4 月 3 日
班级	1903104	学号	1190201421	姓名	张瑞

```
zr@ubuntu:~/shared/Database/BufMgr/src$ ./badgerdb_main
Found record: hello! on page 1
Found record: hello! on page 2
Found record: hello! on page 3
Found record: hello! on page 4
Found record: hello! on page 5
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
Test 8 passed
Test 9 passed

Passed all tests.
zr@ubuntu:~/shared/Database/BufMgr/src$
```

可以看到，原有测试用例和添加的测试用例全部通过。

四、实验结论（总结实验发现及结论）

通过本次实验，我对缓冲区管理器的工作原理有了更深刻的认识与理解，掌握了基于时钟算法的缓冲区页面替换策略，对基于 C++语言的编程进行了首次尝试，提升了代码能力，也在阅读源代码的同时，了解了 BadgerDB 各个类的定义与实现，掌握了数据库管理系统里面存储管理器的工作原理。