**《高级数据库系统》课程实验**

Storage and Buffer Manager

实验报告

指导教师： 金 培 权

姓 名： 赵 杭 天

学 号： SC18023140

电子邮件： [htzhao@mail.ustc.edu.cn](mailto:htzhao@mail.ustc.edu.cn) 实验日期： 2019年1月1日

# 一、实验背景

为了了解数据库buffer 管理器的工作原理，对数据库底层结构有更进一步的了解，我们将实现一个简单的存储和缓存管理器。该实验涉及存储和缓存管理器的缓存技术、散列技术、文件存储结构、磁盘空间和缓存模块的接口等功能。

# 二、实验环境

硬件平台：OMEN HP Laptop 15-dc [i7-8750@2.2Ghz](mailto:i7-8750@2.2Ghz) 16.0GB RAM

开发环境：操作系统：Windows 10 Pro 64bit

开发语言：C++ 14

集成开发环境：Visual Studio 2017，Windows SDK 10.0.17134.0

# 三、实验内容

1、实现基于对磁盘进行字节流操作的数据库读写函数；

2、实现基于哈希+双向链表的缓存管理器器设计；（主要）

3、使用trace文件data-5w-50w-zipf.txt验证系统；

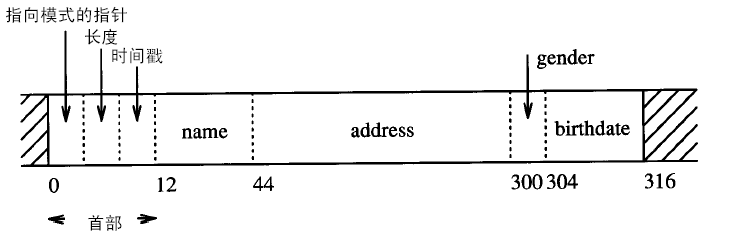
4、更改缓存器大小buffersize对比分析实验结果；

# 三、设计思路与实现概要

## 1、实现基于对磁盘进行字节流操作的数据库读写函数

为了更好地使用trace文件验证系统，首先建立一个真实的数据库，这就需要首先实现对数据库文件进行读写的函数。

根据实验文档所描述的，建立的数据库名称设为data.dbf，若不存在则新建，存在则覆盖。参考上课讲义adb04.pdf与教材，设计的记录类型如下：



每条记录长度为316字节，每个磁盘页大小（pagesize）设定为4KB，，每个磁盘页（这里也称磁盘块）可放入12条记录，在磁盘页首部设计块号（4byte）、块时间戳（4byte）、块偏移量表（索引，24byte），磁盘页尾部填充272个字节。正好实现对齐。文件操作基于C++标准库fstream实现，主要部分如下：

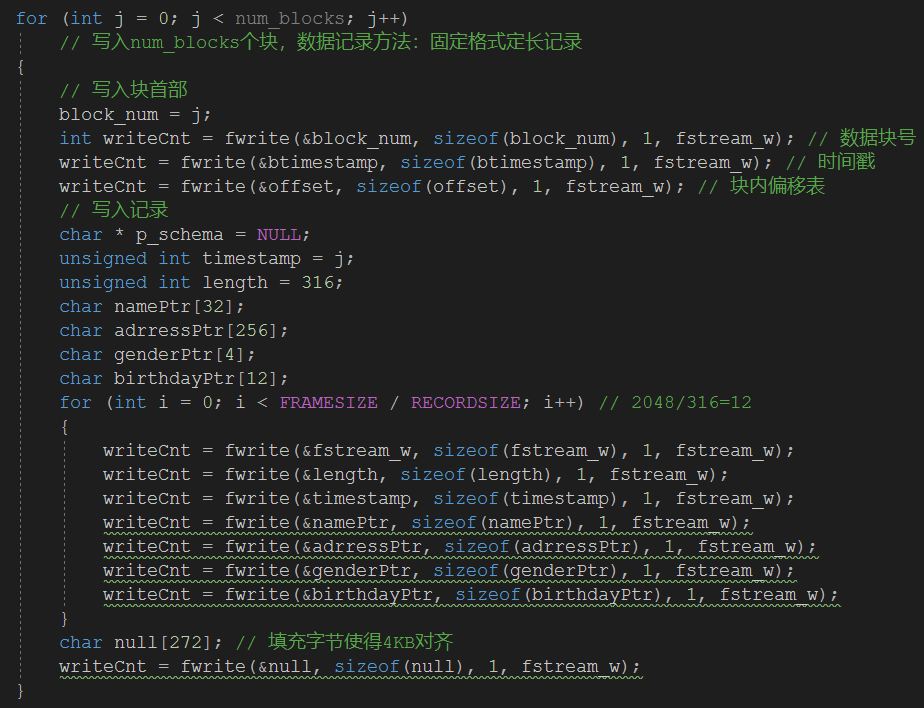


图1.1 数据库文件建立函数

根据trace文件要求，设置num\_blocks=50000，即向磁盘连续写入50000个页，使用目录式堆文件组织， 建立索引。

根据块号返回值确定数据库的正确性：

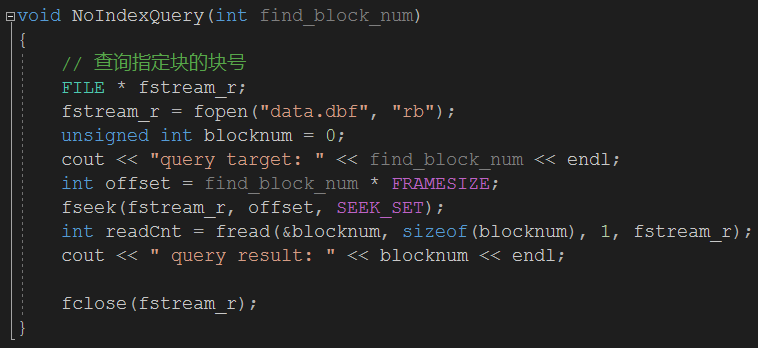


图1.2 数据库文件访问测试

数据库只需建立一次，得到数据库文件data.dbf后可以将CreateBlockWRTest(int num\_blocks)函数的调用注释掉，并备份测试数据库。

## 2、缓存管理器器设计

**2.1确定缓存区的结构**

根据业界标准，缓存页的大小（framesize）一般等于磁盘页大小（pagesize），两者均设为4KB。由上小节讨论，磁盘（disk）data.dbf数据库文件有50000个磁盘页（page），缓存区（buffer）中应存在多个缓存页（frame），缓存区中frame的数量用BUFFSIZE定义，初始设定为1024。在后面的实验中可以看到随缓存区大小（BUFFSIZE）的增大带来的变化。

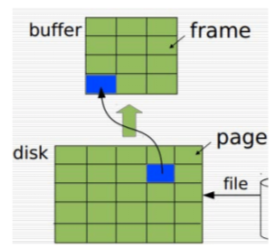


图2.1

**2.2缓存控制块设计（Buffer Control Blocks，BCB）**

一般地，1个BCB维护着1个磁盘页在缓存（内存）中的信息，包括磁盘页号page\_id、此块号对应的缓存页号frame\_id、reference位、用户占用计数count、时间戳time、脏位dirty，BCB结构体如图2.2。在主流的实现方式中，BCB作为数组或者链表的结点，组成便于管理的数据结构，提供快速的查找、更新、置换等功能。

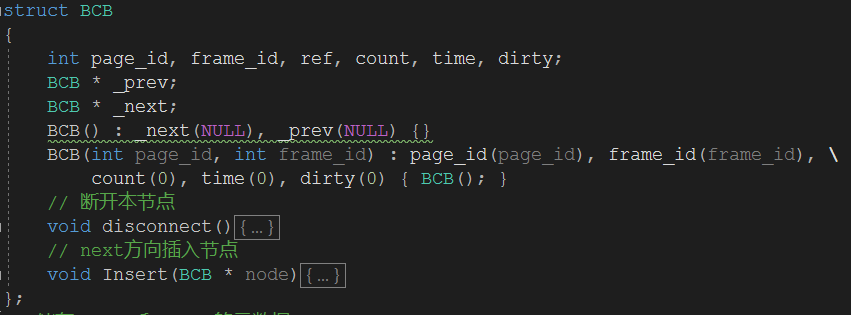


图2.2 BCB结构设计

**2.3使用哈希表+双向链表实现LRUCache**

在大多数场景下，page数远大于BUFFSIZE，所以缓存器必须有一个调度机制，用以实现内存中缓存页frame的置换、淘汰，这里使用了LRU（Least Recent Used）算法。

关于LRU的实现主要有3种方式：

①用一个数组来存储数据，给每一个数据项标记一个访问时间戳，每次插入新数据项的时候，先把数组中存在的数据项的时间戳自增，并将新数据项的时间戳置为0并插入到数组中。每次访问数组中的数据项的时候，将被访问的数据项的时间戳置为0。当数组空间已满时，将时间戳最大的数据项淘汰。这种实现思路比较简单，但缺点也很明显：需要不停地维护数据项的访问时间戳，另外，在插入数据、删除数据以及访问数据时，时间复杂度都是O(n)。

②利用一个链表来实现，每次新插入数据的时候将新数据插到链表的头部；每次缓存命中（即数据被访问），则将数据移到链表头部；那么当链表满的时候，就将链表尾部的数据丢弃。这种方法效率相对第一种稍高，但链表在定位数据的时候时间复杂度为O(n)。

③基于哈希表和双向链表。当需要插入新的数据项的时候（哈希查询），如果新数据项在链表中存在（命中），则把该节点移到链表头部，如果不存在，则新建一个节点，放到链表头部，若缓存满了，则把链表最后一个节点删除（哈希删除）即可。这样一来在链表尾部的节点就是最近最久未访问的数据项。基本所有操作可以实现时间复杂度O(1)。

本实验基于C++ 14 STL容器，使用方法3实现LRUCache：

std::list，相当于双向链表。splice()，push\_front()，pop\_back()的复杂度为O(1)。

std::unordered\_map，相当于哈希表。find()，insert()，erase()的复杂度为O(1) 。

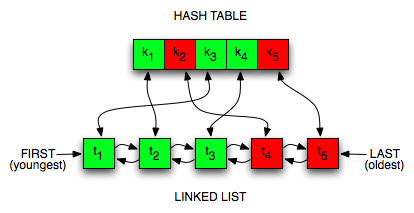


图2.3 哈希表+双向链表实现LRU Cache

LRUCache类具有哈希表+双向链表数据结构。

向BCB添加前向（prev）、后向（next）指针以及断开（disconnect）、插入（insert）函数，使其组成双向链表。

磁盘页号page\_id作为哈希表的key，frame\_id, ref, count, time, dirty作为哈希表的value。其中最主要的value是缓存页号frame\_id。可以以时间复杂度O(1)实现page\_id到frame\_id的映射以及LRU置换。

LRUCache的private成员主要包含：（如图2.4）

①STL容器如哈希表hashmap、栈frame\_ids；

②DiskSpaceManager类实例化对象，用于实现对磁盘文件的读写；

③整型数组f2p[BUFFSIZE]，用于实现frame\_id到page\_id的映射；

④size， capacity，hitNum，IONum作为LRUCache元数据，size为当前LRUCache的大小；capacity为LRUCache容量上限，size到达此值后，未命中的磁盘页请求操作将带来缓冲页置换；hitNum为请求命中数，即请求磁盘页已经在缓存（内存）中，不会发生IO操作；IONum为发生IO操作的累积值。

其中frame\_ids栈主要用于size < capacity时，对新进入的page\_id快速分配frame\_id。

需要注意的是，所有磁盘页的第一次请求肯定会发生IO操作，因为初始的LRUCache不含有任何缓存页的，所以100%的命中率（hit ratio）不会发生。同时也可以预见，随着LRUCache的capacity增加（即BUFFSIZE的增大），命中率应该是上升的，上升趋势应该与请求磁盘页号的统计分布有关。

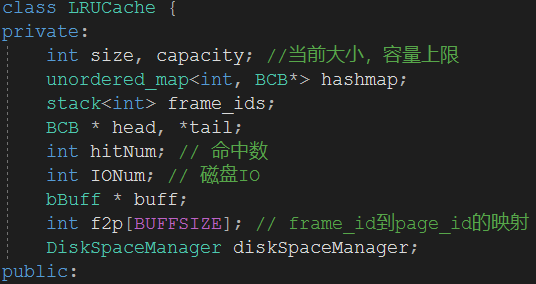


图2.4 LRUCache的private成员

LRUCache的public成员主要包含一系列函数：（如图2.5）

① read(int page\_id)，读出LRUCache对应的frame\_id，并把该节点移到链表头部；

②read\_id(int page\_id)，仅读出LRUCache对应的frame\_id；

③ put(int page\_id, int frame\_id = 0)，将指定磁盘页page写入LRUCache对应的缓存页frame中。在需要时采取置换操作，置换过程中检查脏位并将修改的数据写回磁盘；若指定的磁盘页号在磁盘中不存在，则新建磁盘页。

④selectVictim()，查看要换出的frame\_id；

⑤remove(int page\_id)，删除指定的磁盘页的缓存；

⑥setDirty(int page\_id)，设置指定磁盘页的脏位；

⑦setCount(int page\_id)，设置指定磁盘页的用户数；

⑧readData()，读取指定磁盘页的数据；writeData()，写入指定磁盘页的数据；

⑨getIOnum()，获取LRUCache的IO操作累计值； getHitNum()，获取LRUCache的命中数；

⑩saveDirty2Disk()，完成对象LRUCache释放前的一些工作，主要将所有脏数据写回磁盘。

需要说明的是，本实验设计的程序为了减少IO开销，并不会立即将被修改的缓存页立即写入对应的磁盘页中，而是选择该页被置换时写回磁盘或者LRUCache即将被释放时，将所有脏数据一起写回磁盘。

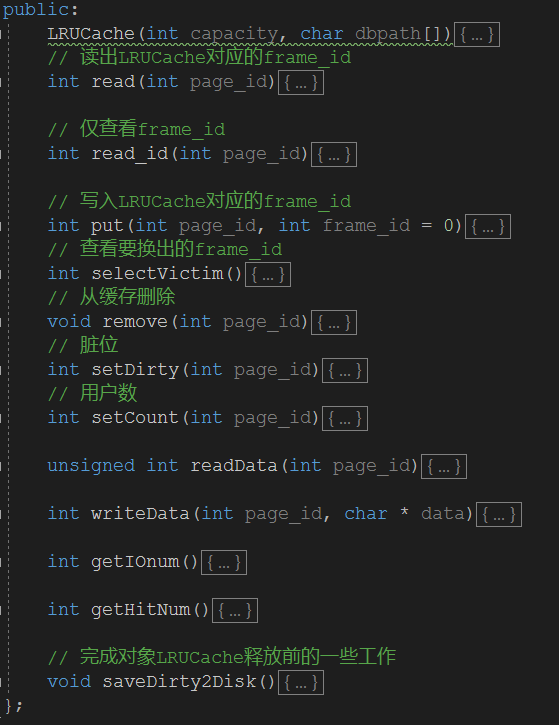


图2.5 LRUCache的public成员

**2.4 实现磁盘文件操作**

为了对磁盘上的数据库文件进行读写，本实验基于标准库stdio.h构建了DiskSpaceManager类以实现文件的流操作，其类方法有：

①openFile(char filepath[])，二进制形式打开文件；

②readDisk(int page\_id, int frame\_id, bBuff \* buff)，将page\_id对应磁盘页读入指定缓存页中；

③writeDisk(int page\_id, int frame\_id, bBuff \* buff) ，将frame\_id对应缓存页写入指定磁盘页中；

需要注意的是，本实验对磁盘IO操作均采用二进制流形式、以磁盘块为单位进行读写。即数据在磁盘中以二进制流表示，被读取到缓存（内存）后还原其数据结构。

添加VERBOSE编译开关控制调试输出，便于进行测试。

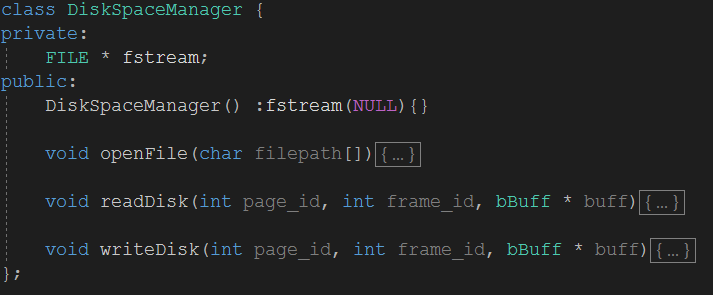
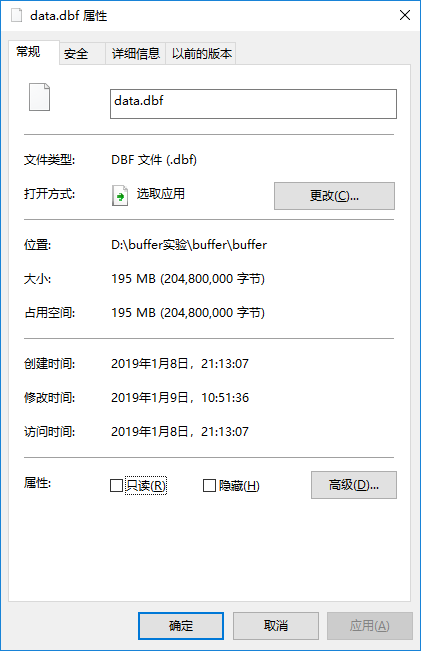


图2.6 DiskSpaceManager的实现概要

# 五、实验结果及分析

## 1、生成数据库文件

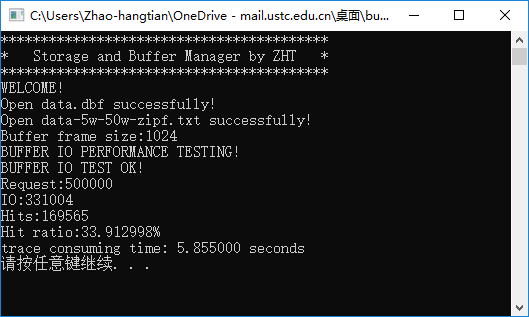
使用CreateBlockWRTest函数生成含50000个磁盘页的数据库文件data.dbf，，如图



## 2、对不同BUFFSIZE进行实验

按行读取设计要求中的trace文件data-5w-50w-zipf.txt，初步处理字符串（分割、类型转换）并调用processTrace函数逐条处理。该trace对0到49,999的page\_id进行500,000次满足Zipf的读写操作。

①以默认值BUFFSIZE=1024进行测试，结果如下图：

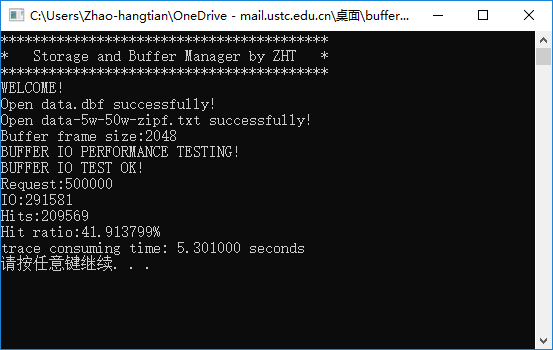


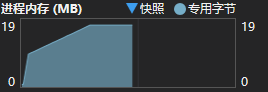


控制台输出信息包含缓冲区大小Buffer frame size、总请求量Request、IO总发起量IO、总命中数Hits、命中率Hit ratio、完成trace总消耗时间trace consuming time。另外，本实验还通过Visual Studio的诊断工具记录了程序的内存占用情况。可以看出，本实验编写程序没有出现明显的内存泄漏。

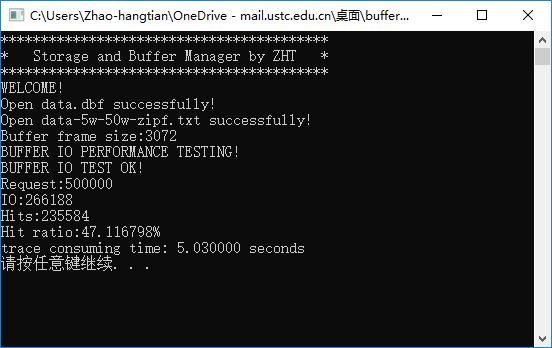
下面将更改BUFFSIZE并分别进行测试，最后汇总于表格1中，并结果进行分析。

②设置 BUFFSIZE=2048进行测试，结果如下图：



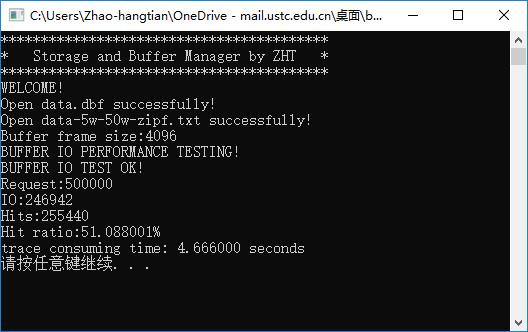


③设置 BUFFSIZE=3072进行测试，结果如下图：



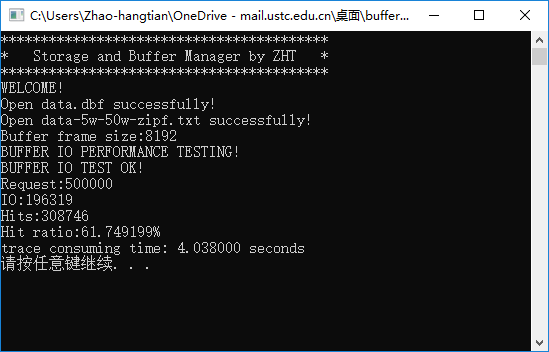


④设置 BUFFSIZE=4096进行测试，结果如下图：



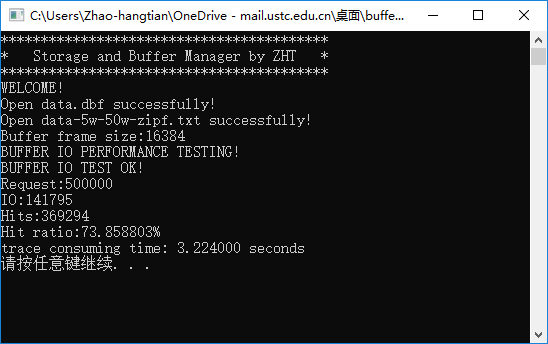


⑤设置 BUFFSIZE=8192进行测试，结果如下图：



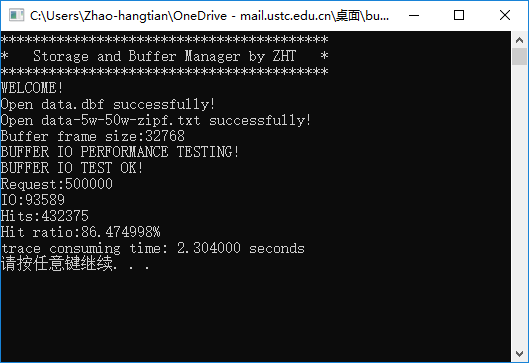


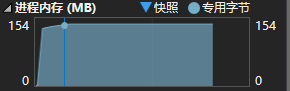
⑥设置 BUFFSIZE=16384进行测试，结果如下图：



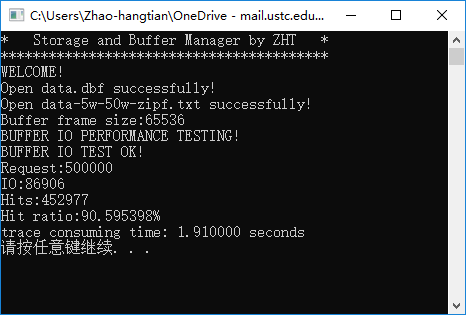


⑦设置 BUFFSIZE=32768进行测试，结果如下图：



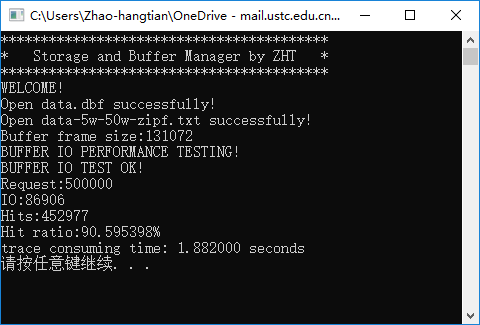


⑧设置 BUFFSIZE=65536进行测试，结果如下图：





⑨设置 BUFFSIZE =131072进行测试，结果如下图：

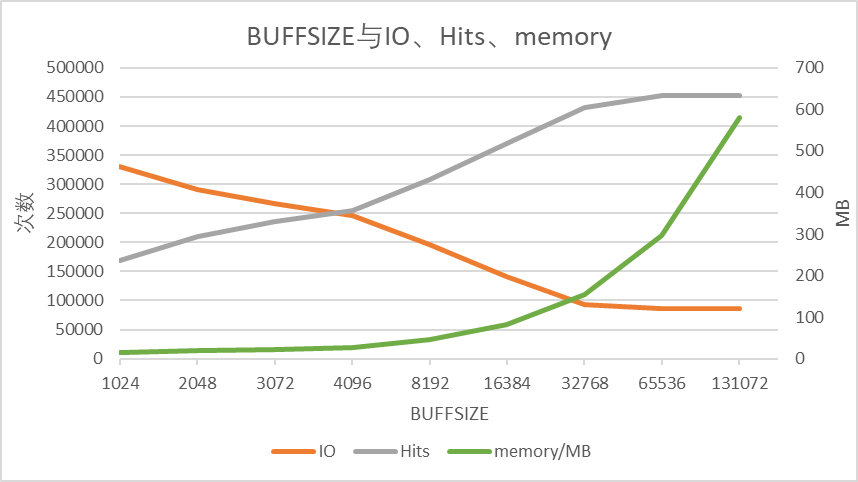


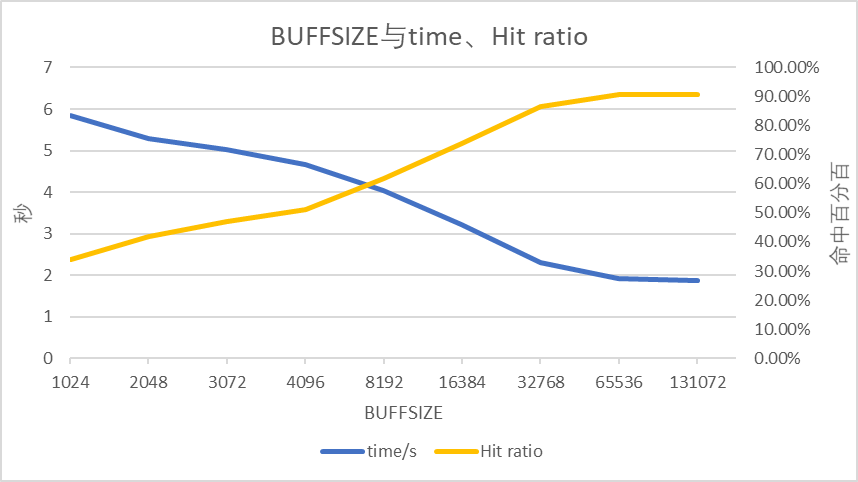


为直观对比，将上述数据整理于下表中：

表1 不同BUFFSIZE的buffer性能表现（Requset=500,000）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| BUFFSIZE | IO | Hits | Hit ratio | time/s | memory/MB |
| 1024 | 331004 | 169565 | 33.91% | 5.855 | 14 |
| 2048 | 291581 | 209569 | 41.91% | 5.301 | 19 |
| 3072 | 266188 | 235584 | 47.12% | 5.03 | 23 |
| 4096 | 246942 | 255440 | 51.09% | 4.666 | 28 |
| 8192 | 196319 | 308746 | 61.75% | 4.038 | 46 |
| 16384 | 141759 | 369294 | 73.86% | 3.224 | 82 |
| 32768 | 93589 | 432375 | 86.48% | 2.304 | 154 |
| 65536 | 86906 | 452977 | 90.60% | 1.910 | 297 |
| 131072 | 86906 | 452977 | 90.60% | 1.882 | 580 |





## 3、分析与结论：

1、观察BUFFSIZE与IO、Hits的关系可以看出，随着缓存页面总数BUFFSIZE的增大，缺页现象逐渐减少，发生磁盘IO的数量稳定减少、命中数Hits稳定增加，IO、Hits两者变化趋势（方向、斜率）是几乎是相反的。

2、随着BUFFSIZE的增大，IO数量将下降到一个稳定值，命中数、命中率页上升到了一个稳定值，即BUFFSIZE继续增大将不会继改善IO性能（命中率）。造成这点的原因分析如下：

①首先，是缓存区的大小已经超过了数据库本身，在将数据库完全读入内存之后、释放LRUCache将脏数据写回磁盘之前，程序将不会因trace产生任何磁盘IO操作；

②如前所述，这是因为初始化的缓存器是不含任何缓存页的，每个磁盘页第一次读入缓存必将产生一次IO；除此之外，在程序即将退出，释放LRUCache时，必须将脏数据写回磁盘，这也必然产生磁盘IO操作。这决定了IO总数绝不可能降至0。

3、从图表中我们可以发现一个有趣的现象，即IO总数下降停留在了86906。固定的86906次磁盘IO是由trace中所有的磁盘号（trace的page\_id从0到49,999，共发起500,000次请求，但是这500,000次请求并未涵盖page\_id从0到49,999的所有page）的第一次读写请求，与写请求之后脏数据写回磁盘所产生的。设trace中出现过的所有的不同的读请求page\_id数为R、trace中出现过的所有的不同的写请求page\_id数为W，则有

4、从命中Hits最终停留在452977次，我们也可以看出，足够大LRUCache将数据库全部读入缓存后，所有请求全部在内存中实现命中。可以推出trace请求的不同的page\_id数，即trace共出现了47023个不同的page\_id。

5、进一步地，由和，我们可以推出：

trace中出现的不同的读请求page\_id数

trace中出现的不同的写请求page\_id数

6、观察IO和Hits的值可以发现，在所有的情况下，IO和Hit的总数都是超过trace发起的请求数500,000的。原因是：

当LRUCache进行1次缺页写操作时，必然带来0次Hit和 2次IO，一次是缺页置换，另一次是脏数据写回磁盘；LRUCache进行1次缺页读操作时，必然带来0次Hit和 1次IO。此外，LRUCache在对同一page\_id进行n次不缺页的读操作时，产生n次Hit和0次IO；LRUCache在对同一page\_id进行n不缺页的写操作时，产生n-1次Hit和1次IO。

综合来看，每1次的读写请求会产生三者情况：1次IO和0次Hit、0次IO和1次Hit、2次IO和0次Hit，所以IO和Hit的总数必然会超过trace发起的请求数。

7、观察BUFFSIZE、memory、time三者的关系，可以看出随着BUFFSIZE的增大，内存占用memory快速上升，而执行trace的时间time逐渐减少。这本质上是采取了“空间换时间”策略。可以看出，分配更大的BUFFSIZE，time会逐渐减少，但减少的速度逐渐放缓，这也符合了“边际效用递减法则”。

# 六、总结

本次实验从金老师布置的第二周便开始着手准备，参考实验描述文档《Storage and Buffer Manager Lab Description》和上课讲义，结合自己的想法，回忆着数据结构、数据库和算法方面的知识，在两个星期里用C++一步步的实现，但半期之后课业逐渐繁重，最近才开始撰写报告。期间也向助教老师进行过请教，在此感谢助教老师和金老师提供的指导。

本次实验最具挑战性的地方应该是如何把握数据结构的组织，必须清晰地认识buffer管理器的功能、原理和定位，以及灵活运用哈希表和双向链表数据结构，熟悉利用C++STL可以起到事半功倍的效果。

通过这次实验，我对于数据库buffer管理器的工作原理有了更深的认识。本实验使用C++实现，期间使用了大量的指针操作、流操作、STL。本次实验，使我对C++指针有了更深刻的认识、动手实践了二进制字节流磁盘保存与在内存中还原其数据结构的操作、动手实现了“哈希+双向链表”这一经典数据结构、了解熟悉了unordered\_map、stack、vector等C++标准模板，收获颇丰。

# 六、源码

本实验源码（sln工程包，release编译）附在ftp服务器提交的压缩包中；

亦可1.16号后访问<https://github.com/Zhao-hangtian/adb-buffer>获取；