

《高级数据库系统》课程实验报告

刘思理 SA20011902

目录

目录	
实验简介	
实验环境	
实验设计	
设计思路	
系统流程	
功能类的设计与实现	
LRU 类 LRU	
数据存储管理器(Data Storage Manager)类 DSMgr	
缓存管理器(Buffer Manager)类 BMgr	
主函数	
实验结果	
项目文件结构	
程序运行结果和分析	
运行结果和 UI 展示	
Buffer 中 frame 数量的影响	
总结与收获	

实验简介

本实验的主要内容是完成一个缓存和数据存储管理器，其中主要涉及到缓存和数据存储管理器的数据结构，Hash表的应用，块在文件中的组织，以及数据在内存与磁盘之间的读写操作等内容。本实验通过构建缓存和数据存储管理器，并改变缓存中帧的数目，探讨不同大小的缓存对读写数据库效率的影响。

实验环境

- 硬件平台：MacBook Pro (Retina, 13-inch, Early 2015)
 - 处理器：2.7 GHz 双核Intel Core i5
 - 内存：16 GB 1867 MHz DDR3
- 操作系统：macOS Catalina 10.15.4
- 集成开发环境：Clion 2019.3

- 编程语言：C++

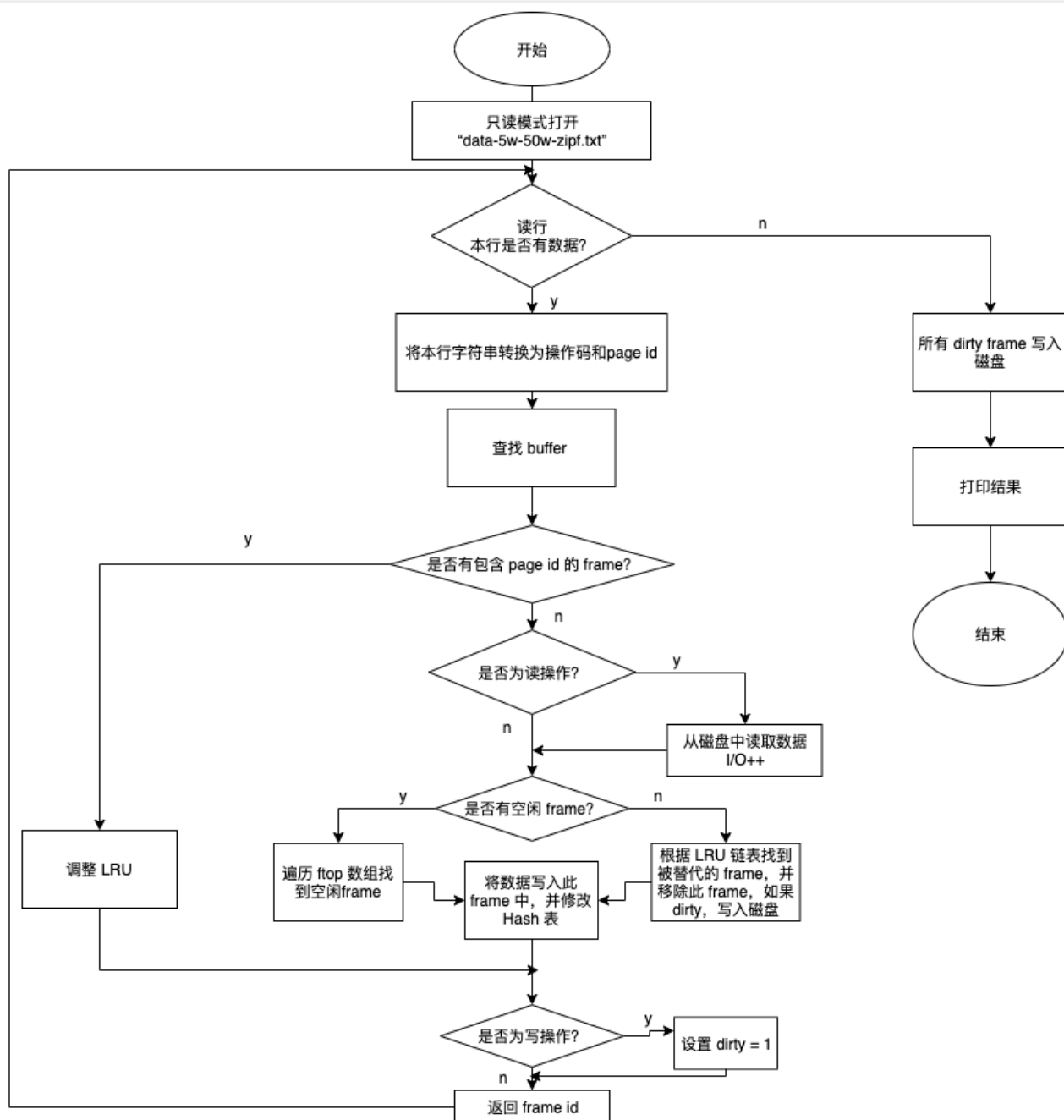
实验设计

设计思路

Buffer中每个frame、磁盘中每个页的大小相等且都为4096 Byte = 4 KB，统一用 `#define FRAMESIZE 4096` 表示。buffer中包含的frame个数用 `DEFFBUFSIZE` 表示。上层首先传递读写记录请求，如果是写请求且需要一个空页，Data Storage Manager 会返回一个没有用过的页号，否则返回记录所存储页对应的虚拟页号。Data Storage Manager 中的虚拟页号需要与磁盘中数据块在目录块中的序号所对应，虚拟页号用 `page_id` 表示，磁盘块在目录块中的序号用 `block_num` 来表示。得到虚拟页号之后，便可以从 Buffer Manager 找到对应的 frame。如果这个 buffer 中存在这个 frame，则记为命中一次，且可以直接在内存中完成读写；否则，如果用户申请读操作的话，需要通过 Data Storage Manager 对磁盘进行读操作（用户如果申请写操作则不需要读磁盘数据，因为只需要在内存上把对应的 frame 进行写操作即可），这样一次读写操作就完成了。在最后，需要将 buffer 中所有 dirty 的 frame 全部写入磁盘，一次读写流程结束。

系统流程

本程序执行的流程图为



功能类的设计与实现

本项目一共设计并实现了三个类，分别是缓存管理器 `BMgr`，数据存储管理器 `DSMgr` 以及 `LRU`。下面分别介绍这几个类的成员变量、方法和功能。

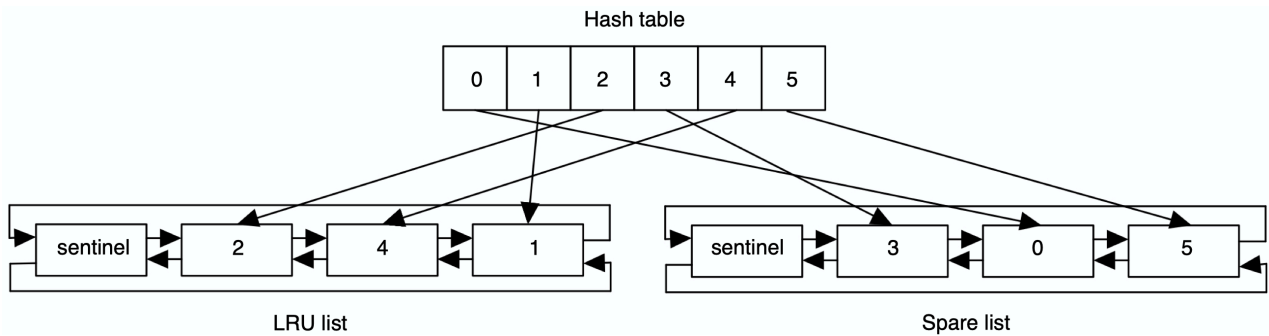
LRU 类 `LRU`

当 buffer 中有 frame 需要被置换时，采用 LRU 策略来寻找被替换 frame 的 frame id。LRU 策略需要维护一个 LRU 链表，这个链表的结点包含的信息是 frame id，结点按照 frame id 对应的 buffer 中的 frame 截至目前最后一次被访问的时间先后顺序排序连接。最后一次被访问的时间离当前时间越近，越靠近链表尾端，否则越靠近链表头端。由于结点包含的 frame id 范围是 0~1023 且每个结点的 frame id 都不相同，因此最多需要 1024 个结点。若结点 frame id 在 buffer 中的 frame 正在被使用，则这个结点在 LRU list；相反，包含的 frame id 在 buffer 中没被使用的话，存储在备用链表 spare list 上。每个结点的内存地址都用散列表的方式来实现索引，以便能以 $O(1)$ 的时间复杂度查找给定的 frame

id 所对应结点的位置。

设置散列表的目的是能以 $O(1)$ 的时间复杂度找到包含特定 `frame_id` 的链表中的结点。当 buffer 大小 `DEFBUFSIZE` 很大的时候，LRU 链表会非常长，因此设置散列表辅助查找结点地址是很有必要的。设置 spare list 的目的是当 buffer 需要空闲 frame 的时候，也可以以 $O(1)$ 的时间复杂度返回一个空闲的 frame id。每次 buffer 申请一个空闲 frame id 时，LRU 会将 spare list 中的头结点（哨兵结点的后继结点）的 frame id 返回给 buffer。

上述数据结构的示意图如下所示。



LRU 类包含一个散列表、两个双向链表、LRU 链表大小和相关方法。链表中一个 LRU 结点的结构如下

```
typedef struct _LRUNode {
    int frame_id;
    struct _LRUNode *previous;
    struct _LRUNode *next;
} LRUNode;
```

，LRU 类的方法和成员变量如下。

```
class LRU {
public:
    LRU();

    void RemoveFirst();

    void AddLast(int frame_id);

    void Remove(int frame_id);

    int GetSize();

    int GetFirst();

    int GetFreeFrame(); // returns a free frame id from spare list

    void FreeLRU();

private:
```

```

LRUNode *use_sentinel, *free_sentinel;
LRUNode *hash_table[DEFBUFSIZE];
int size;          // first node of LRU
};

```

成员变量中，`use_sentinel` 是 LRU list 的“哨兵结点”，只存在充当头结点的作用 `use_sentinel` 是 LRU list 的“哨兵结点”；同理，`free_sentinel` 是 spare list 的哨兵结点；`hash_table` 数组用于实现一个从 `frame_id` 到结点散列表，即 `hash_table[i]` 指向 `frame_id` 为 `i` 的结点。

成员方法中，`size` 是 **LRU** 链表中除了哨兵结点以外剩下结点的个数。成员方法中，`RemoveFirst()` 是将 LRU 链表中的哨兵结点的后继结点删除，相当于删除 LRU 链表中的 LRU 端，每次在 buffer 中替换 frame 的时候都会用到这个方法；`AddLast(int frame_id)` 表示在 LRU 链表尾部插入一个包含 `frame_id` 的结点；第三个方法表示删除包含指定 `frame_id` 的结点；`GetSize()` 返回链表的 `size` 变量；`GetFirst()` 返回链表哨兵结点的后继结点，即 MRU 端；`GetFreeFrame()` 方法返回一个空闲的 frame 的 `frame_id`；`FreeLRU()` 删除动态分配给链表中所有结点的内存空间，以防内存泄漏。

下面介绍下 LRU 链表中的删除和插入结点的实现。LRU 链表的删除本质上是把 LRU list 中的被删除结点利用头插法插入到 spare list 的头结点，具体实现时通过 `hash_table` 以 $O(1)$ 的时间复杂度找到要被删除的结点，然后将其从 LRU list 中删除并插入到 spare list 中哨兵结点的后面。具体实现的代码如下所示。

```

void LRU::Remove(int frame_id) {
    LRUNode *p, *q;
    // get the node to delete with hash table and then get its previous node
    q = hash_table[frame_id];
    p = q->previous;

    // delete the node from LRU list
    p->next = q->next;
    p->next->previous = p;
    size--;

    // add the node to the spare list as the first node
    q->next = free_sentinel->next;
    q->previous = free_sentinel;
    free_sentinel->next = free_sentinel->next->previous = q;
}

```

插入方法的实现同理，只不过是 spare list 中的结点移动到 LRU list 中去。具体代码如下所示。

```

void LRU::AddLast(int frame_id) {
    // get the node of given frame id in spare list and its previous node
    LRUNode *p = hash_table[frame_id];
    LRUNode *q = p->previous;

    // delete the node from spare list

```

```

q->next = p->next;
p->next->previous = q;

// add the node to the tail of LRU list
use_sentinel->previous->next = p;
p->previous = use_sentinel->previous;
p->next = use_sentinel;
use_sentinel->previous = p;
size++;
}

```

数据存储管理器(Data Storage Manager)类 **DSMgr**

在实验文档“buffer.pdf”的基础上，增加了以下成员变量和方法。

```

public:

    int GetNumIOs();    // returns number of I/O operations

    int GetBlockNum(int page_id);    // returns page id in file

    void SetIndex(int page_id, int block_num);    // set index of memory virtual
page id to file page id

    int BlockNumToOffset(int block_num);    // returns offset from the
beginning of file

    int LookForFreePage();    // returns the id of a free virtual page of memory

    int LookForFreeBlock();    // returns the id of a free page of file

    void FreeDSMgr();    // deletes all allocated space

private:

    int blocks[MAXPAGES];    // page id in file manager
    int pages[MAXPAGES];    // page id in memory
    int index[MAXPAGES];    // page id of memory to page id of file

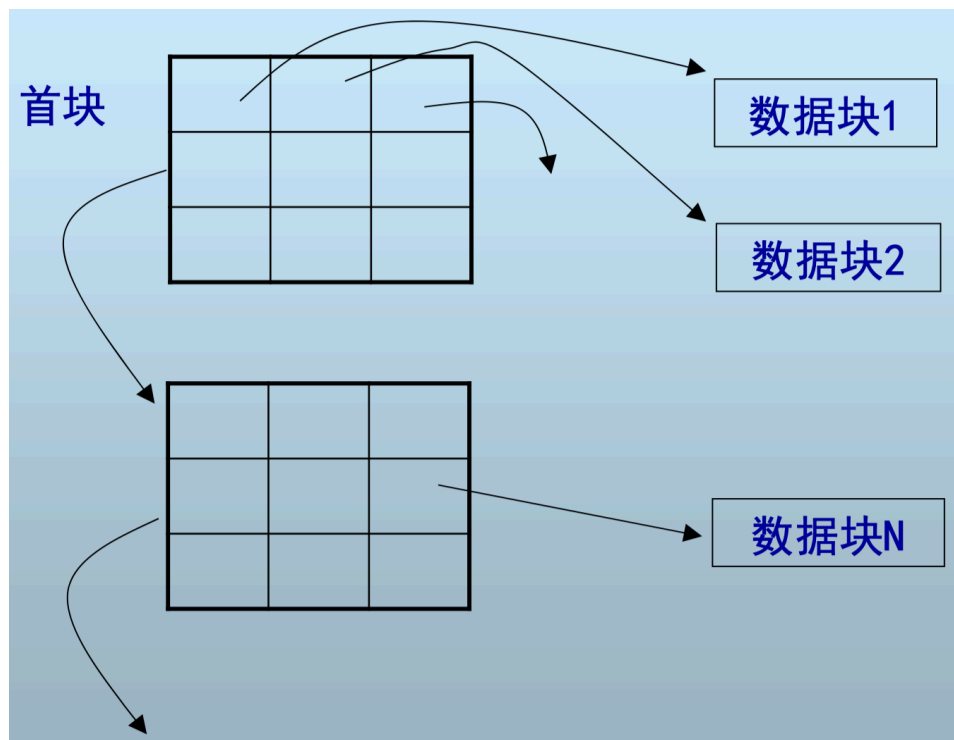
    int numIOs;

    static void GenerateDatabase();

```

在 `DSMgr` 中，涉及到两种页号：一个是内存管理中的虚拟页号，这个页号对应着内存中的物理地址；另一个是文件管理中的虚拟页号，这个页号可以对应磁盘中的存储空间地址。因此，需要建立一个从内存管理中的虚拟页号到文件管理中的虚拟页号索引，这就是 `index` 数组。`numIOs` 统计了总的 I/O 操作次数。

成员方法中，`GenerateDatabase()` 方法在 `DSMgr` 初始化的时候生成了数据文件 "data.dbf"。这个数据文件的组织方式是目录式堆文件组织，这种组织结构的示意图如下所示。



根据 trace 文件中的 page id 分布可知，数据块至少有 50000 块。一个数据块的大小为 4096 字节，目录块存储了所有数据块的偏移量，一个偏移量（整型变量）大小为 4 个字节，则一个目录块可以存放 1024 个数据块的偏移量，远小于数据块的个数 50000，因此需要多个块来构成目录。每个目录块的头部存储了下一个目录块的地址，因此每个块还剩下 1023 个数据块偏移量，因此 50000 个数据块需要 $50000/1023 = 49$ 个目录块。因此，目录式堆文件最终由 49 个目录块和 50000 个数据块构成。生成 "data.dbf" 的代码如下。

```
/**
 * This is a helper method used for initialization to generate data.dbf. In the
 * file "data.dbf", there are 49
 * directory pages and 50,000 data pages. The directory pages store the
 * pointers, each of which points to a data page.
 */
void DSMgr::GenerateDatabase() {
    FILE *data_file = fopen("../data/data.dbf", "wb");

    int content[1024];
    int i, j, offset;
    for (i = 0; i < 1024; i++) content[i] = -1;

    // initialize the directory pages
    for (i = 0; i < 49; i++) {
```

```

        content[0] = (i + 1) * FRAMESIZE;
        offset = (i * 1023 + 49) * FRAMESIZE;
        for (j = 1; j < 1024; j++) {
            content[j] = offset;
            offset += FRAMESIZE;
        }

        fwrite(content, sizeof(int), 1024, data_file);
    }

    // initialize the data pages
    for (i = 49; i < 50049; i++) {
        for (j = 0; j < 1024; j++) content[j] = i - 49;

        fwrite(content, sizeof(int), 1024, data_file);
    }

    fclose(data_file);
}

```

`BlockNumToOffset(int block_num)` 方法目的是找到文件页号对应的偏移量（物理地址）。首先确定数据块号的地址存储在第几个目录页及页内地址，然后迭代访问目录页并找到相应的目录页，在这个目录页中最终可以找到对应数据块的偏移量并返回。代码如下。

```

int DSMgr::BlockNumToOffset(int block_num) {
    int dir_num = block_num / 1023, offset_in_dir = block_num % 1023 + 1;
    int i, offset = 0;

    // find the directory page
    for (i = 1; i <= dir_num; i++) {
        fread(&offset, sizeof(int), 1, currFile);
        Seek(offset, 0);
    }
    offset += offset_in_dir * 4;    // get the offset of the pointer that
    points to the block we want
    Seek(offset, 0);

    fread(&offset, sizeof(int), 1, currFile);    // read the pointer and get the
    offset
    Seek(0, 0);    // rewind the file pointer
    return offset;
}

```

`LookForFreePage()` 方法会返回一个空闲的内存逻辑页号，这个方法会遍历 `pages` 数组并找到一个 `use_bit` 为 0 的 id，buffer manager 类中的 `FixNewPage()` 会调用这个函数来获取 `page_id`。`LookForFreeBlock` 方法和这同理。

在文档中提到的方法 `WritePage(int frame_id, bFrame frm)` 中，将缓存数据写入磁盘之前，需要找个文件管理中的一个虚拟页并将数据写入其对应的磁盘块，因此需要先查询索引。如果索引为空，就需要调用 `LookForFreeBlock` 来找到一个空的磁盘块号，在进行写入操作，具体代码如下。

```
int block_num = GetBlockNum(frame_id);
if (block_num == -1) {
    block_num = LookForFreeBlock();
    SetIndex(frame_id, block_num);
    IncNumPages();
}
```

写磁盘后需要将 `numIOs` 加一。

`DSMgr` 的初始化包括生成一个数据库文件 "data.dbf"，将文件指针 `currFile` 指向数据文件开头，然后在建立虚拟页号和磁盘块号的使用数组以及从虚拟页号到磁盘块号的索引数组。具体代码如下。

```
DSMgr::DSMgr() {
    GenerateDatabase();

    OpenFile("../data/data.dbf");    // open the database file

    numPages = MAXPAGES;    // initially, all the pages ids are used
    int i;
    for (i = 0; i < MAXPAGES; i++) {    // set the use array of page ids, block
ids and the index table
        blocks[i] = 1;
        pages[i] = 1;
        index[i] = i;
    }

    numIOs = 0;
}
```

需要注意的是，本程序第一次运行之后就会生成文件 `data.dbf`，因此在之后运行测试的过程中可以把 `GenerateDatabase()` 这一行注释掉。

缓存管理器(Buffer Manager)类 `BMgr`

在实验文档 "buffer.pdf" 的基础上，加了以下成员变量和方法。

```
public:

    int GetHits();    // returns hits

    void FreeBMgr();    // delete all allocated space
```

```

int GetNumIOs();    // returns I/Os

int GetOffset(int block_num);    // returns offset

private:

bFrame *buffer[DEFBUFSIZE]; // frame array

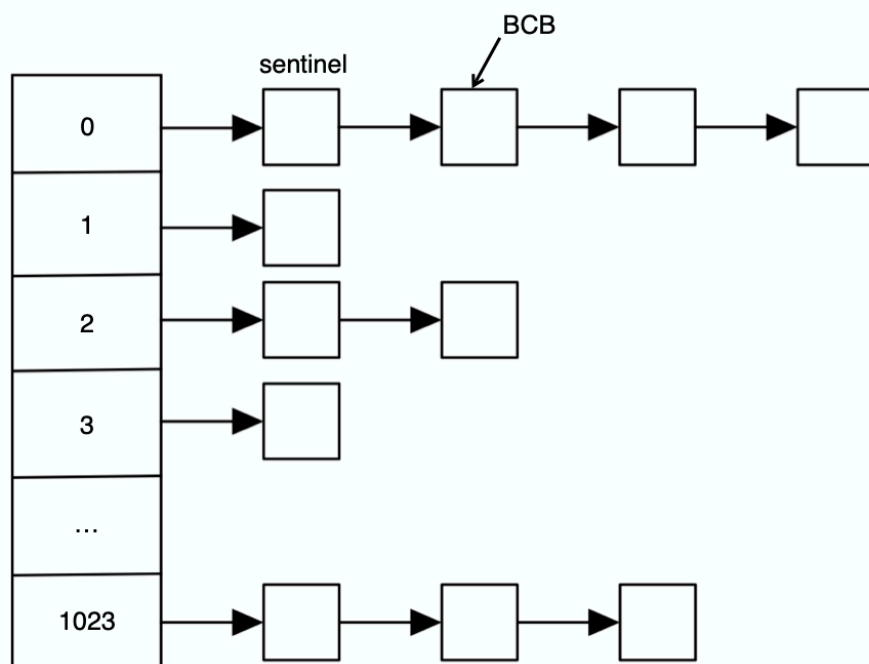
LRU *lru;    // LRU object
int numFreeFrame;    // number of free frames
int hits;    // number of hits
DSMgr *dsmgr;    // data storage manager object

int LookForFreeFrame(); // returns a free frame id

```

在新增的成员变量中，`buffer` 中每个元素都是一个 buffer frame，对应的数组下标表示对应的 `frame_id`。`lru` 是一个 LRU 对象，包含了这个 buffer manager 的各 frame 访问时间先后信息。`numFreeFrame` 表示空闲 frame 的个数，`hits` 表示请求页面的命中次数，`dsmgr` 是一个 data storage manager 的对象。

`frame_id` 和 `page_id` 之间的哈希表为 `ftop` 和 `ptof` 两数组。其中，`frame_id` 范围是 0 到 1023，因此每个 `frame_id` 仅对应一个 `page_id`，因此 `ftop` 是一个整型数组。而 `page_id` 的范围是 0 到 50000，远大于 `frame_id` 的范围，因此需要设置一个 Hash 映射函数，实验文档中将这个映射函数设为 $H(k) = \text{page_id} \% \text{buffer_size}$ 。`ptof` 中的每个元素是一个 BCB 单链表，这个链表头部也设有哨兵结点，剩下所有结点的 `page_id` 的 Hash 函数值都等于 `ptof` 数组的下标。`ptof` 数组结构示意图如下所示。



`FixPage` 方法返回输入的 `page_id` 的对应 `frame_id`。当 buffer 中存在这个 page 对应的 frame 的时候即为命中，此时需要 `hits++`，并调整 LRU 链表；否则此请求没有命中。当没有命中的时候，只有是读操作的时候，Data Storage Manager 才会去读磁盘，导致 `numIOs++`。而当写磁盘块时，只需要将利用写的数据而不需要数据库中的记录数据，因此不需要读磁盘块，`numIOs` 不会增加。

`FixNewPage` 方法会调用对象 `dsmgr` 中的 `LookForFreePage` 方法来从 `pages` 数组中寻找没有正在被使用的 `page_id`，同时也会寻找一个空闲的 frame。如果发现 `numFreeFrame = 0`，即所有 frame 都正在被使用，此时需要使用 LRU 置换策略，调用 `SelectVictim` 方法来置换一个 frame 并得到此 frame 的 `frame_id`。

主函数

在主函数 `main` 中，首先打开 trace 文件 `data-5w-50w-zipf.txt`。按行读取文件，并调用标准库 `stdlib.h` 中的 `char *strtok(char *str, const char *delim)` 函数，它可以将字符串 `str` 分成若干部分，分割符是 `delim` 字符串中的所有字符。在本实验中分隔符字符串定义为 `const char *IGNORE_CHARS = "\f\n\r\t\v,"`。因此每行可以被分为两个字符串，分别代表操作码（读或写）和页号。之后又调用了 `stdlib.h` 标准库中的 `long int strtol(const char *str, char **endptr, int base)` 函数，它可以将字符串 `str` 转换成 `base` 进制的数字。`base` 也可以取特殊值 0。通过这两个库函数，每行字符串便以逗号为分割提取出两个整数，再将这两个整数输入 buffer 中的 `FixNewPage` 方法即可。在所有测试结束之后，需要调用 buffer 中的 `WriteDirty` 方法将所有的脏数据写入磁盘。

主函数中对每个 `FixNewPage` 和最后的 `WriteDirty` 进行计时，并输出以秒为单位的总时间。主函数代码的关键部分如下。

```
while (fgets(line, 20, f) != NULL) {
    num_lines++;
    // divide the line by comma and store operation code and page id in array
    args

    token = strtok(line, IGNORE_CHARS);
    args[0] = token;
    token = strtok(NULL, IGNORE_CHARS);
    args[1] = token;

    // transform strings to integers
    op = strtol(args[0], &token, 0);
    page_id = strtol(args[1], &token, 0);

    // request of reading or writing data and calculate total time
    start = clock();
    bmgr->FixPage((int) page_id, (int) op);
    end = clock();
    time_cost += (end - start) * 1.0 / CLOCKS_PER_SEC;
}
fclose(f);

// write all dirty frame to the disk and calculate total time
```

```
start = clock();
bmgr->WriteDirtyys();
end = clock();
time_cost += (end - start) / CLOCKS_PER_SEC;
```

实验结果

项目文件结构

本项目目录下主要包含两个文件夹，分别是存放数据文件和测试文件的 data/ 目录以及存放源代码的 src/ 目录，这两个目录的树形结构如下所示。

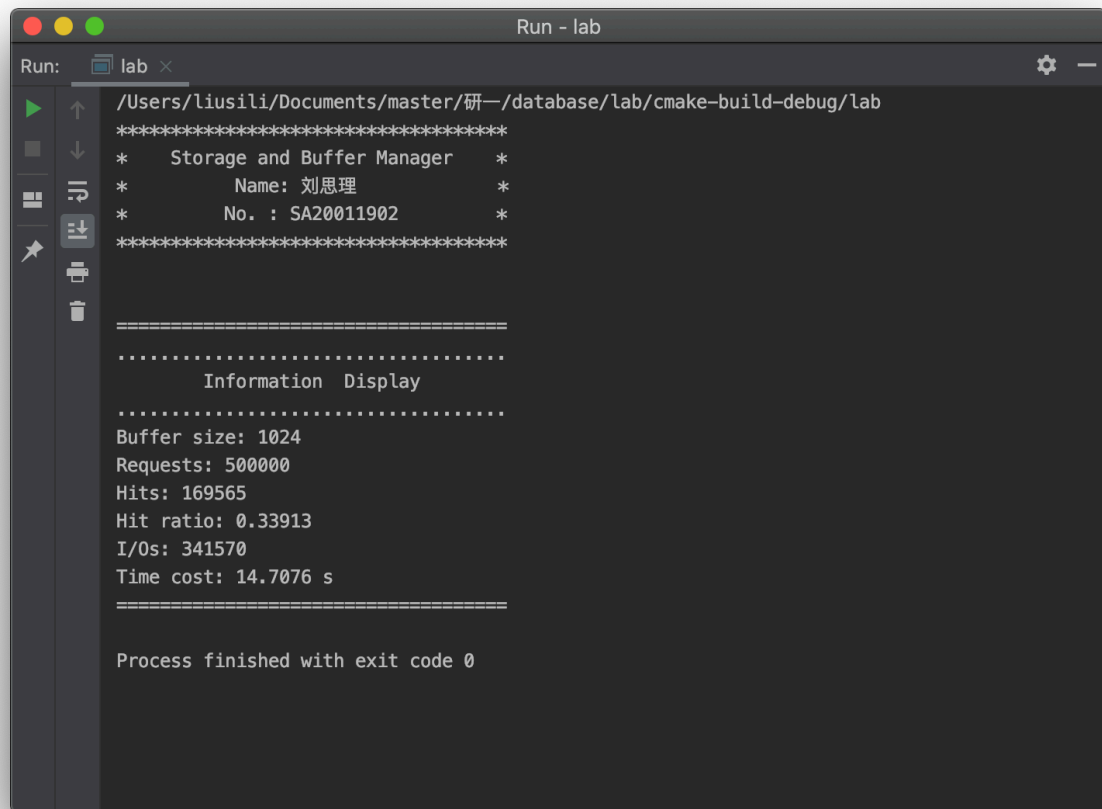
```
src
├── [ 6526]  BMgr.cpp
├── [ 1447]  BMgr.h
├── [ 5622]  DSMgr.cpp
├── [ 1400]  DSMgr.h
├── [ 1502]  LRU.cpp
├── [ 477]   LRU.h
└── [ 2742]  main.cpp
data
├── [ 3690270] data-5w-50w-zipf.txt
└── [ 205000704] data.dbf
```

其中中括号里的内容是文件大小，以字节为单位。可以看到，data.dbf 文件大小是 205000704 字节。在 Data Storage Manager 的 GenerateDatabase 方法中，生成了 49 个目录块和 50000 个数据块，共计 50049 个文件块。每个文件块有 4096 个字节，因此文件总大小为 $50049 \times 4096 = 205000704$ bytes，与实际文件大小吻合。

程序运行结果和分析

运行结果和 UI 展示

当 `DEFBUFSIZE` 为 1024 时，程序运行结果截图如下所示。



```
Run: lab x
/Users/liusili/Documents/master/研一/database/lab/cmake-build-debug/lab
*****
*   Storage and Buffer Manager   *
*   Name: 刘思理                 *
*   No. : SA20011902            *
*                               *
*****

=====
.....
      Information Display
.....
Buffer size: 1024
Requests: 500000
Hits: 169565
Hit ratio: 0.33913
I/Os: 341570
Time cost: 14.7076 s
=====

Process finished with exit code 0
```

其中，Buffer size 表示 buffer 中 frame 的数量，Request 表示 trace 文件中请求的次数，Hits 表示命中的请求的次数，Hit ratio 表示所有请求中命中的比例，即 Hits / Request，I/Os 表示 I/O 操作总次数，Time cost 表示所有请求和最后把脏页面写入文件的总时间花费。

可以看到，当 buffer 中 frame 数量为 1024 的时候，命中率只有 33.913%，且需要 341570 次 I/O 操作。

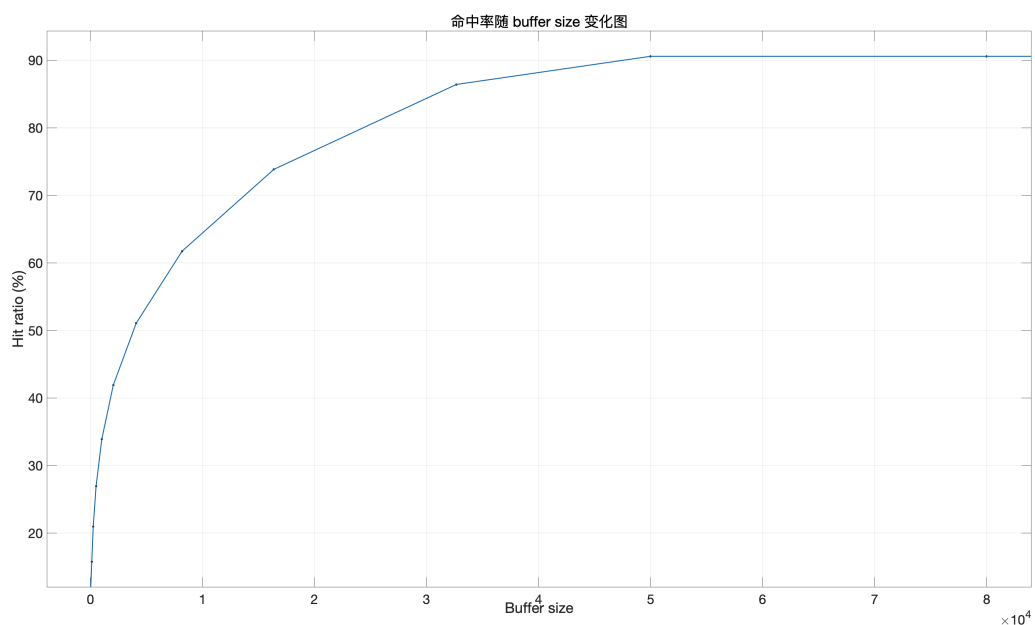
Buffer 中 frame 数量的影响

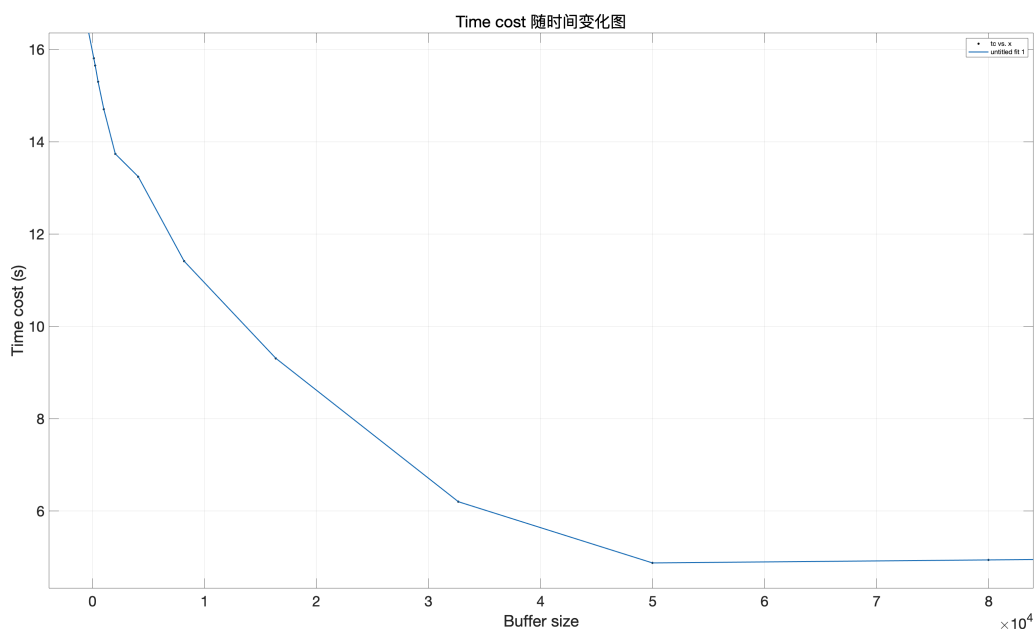
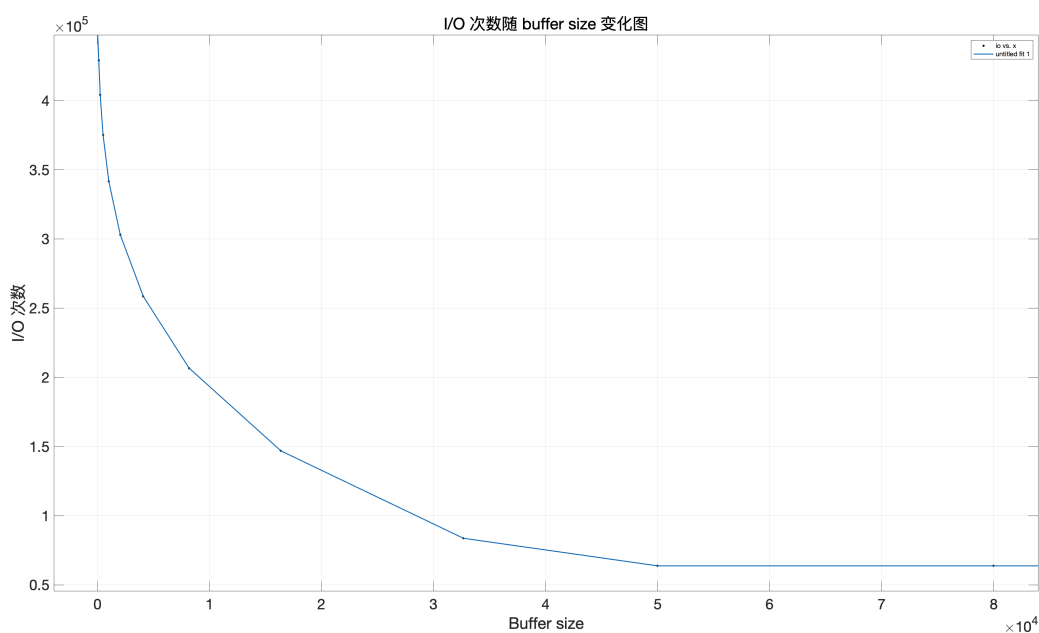
在本实验中，buffer 中 frame 的数量是可调整的。我设置了 11 个不同的 `DEFBUFSIZE` 并分别得到各指标的结果，并探讨了 frame 数量的 buffer 表现效果的影响。

buffer size 及相关指标的数据写在下表中

Buffer size	Hit ratio (%)	I/Os	Time cost (s)
128	15.7472	429074	15.8119
256	20.9354	404161	15.6552
512	26.9588	375072	15.3040
1024	33.9130	341570	14.7076
2048	41.9138	302977	13.7398
4096	51.0880	258588	13.2458
8192	61.7492	206668	11.4146
16384	73.8588	146904	9.30795
32678	86.4296	83741	6.20076
50000	90.5954	63838	4.87483
80000	90.5954	63838	4.94003

画 Hit ratio, I/Os, Time cost 这三个指标随 Buffer size 的折线图，如下所示。





可以看到随着 buffer size 增大，各指标的变化趋势为：命中率上升，I/O 次数下降，时间花费减少。另外，当 buffer size 上升到 50000 后，各指标基本不再变化。其中的原因是显而易见的：由于 page id 的范围是 1-50000，因此当 buffer size 涨到 50000 以上后，buffer 能缓存所有的记录，即不会在中途出现 buffer 已满导致需要选择一个 frame 进行置换，从而不会发生脏页面置换产生的写磁盘的操作、增加 I/O 次数的情况。只要某个请求的 page id 在之前的请求出现过，都会在 buffer 中命中，因此对于同样的一个请求序列，buffer size 大于 50000 后，命中率也会保持不变。

以下是在请求 frame 的过程中，可能会出现的事件及其时间复杂度：

- 查询 page id 对应的 BCB 块： $O(m)$. 其中 m 是散列表中对应的链表长度.
- 调整 LRU 链表： $O(1)$. 由于有散列表的存在，寻找、插入、删除链表结点的时间复杂度都为 $O(1)$.
- 寻找空闲的 frame： $O(1)$. 由于有 spare list 的存在，寻找空闲 frame 的时间复杂度也为 $O(1)$.
- 读或写磁盘页：一次 I/O 时间.

当 buffer size 增大时，平均链表长度会越来越短，m 变小，因此查找 BCB 块的时间会减小。特别地，当 buffer size > 50000 时，m = 1；由于命中率变高，需要读操作的次数也减少；同时，LRU 置换次数也会减少，导致脏页面的写操作的次数也减少。综上所述，buffer size 的增大确实可以减少时间花费。

但 buffer size 并不是越大越好，buffer size 越大，占用的内存空间也就越大。所以当 buffer size 等于 80000 的时候，时间花费会稍稍高于 buffer size = 50000 的时候。这是因为二者虽然理论的时间花费相同，但前者占用的内存资源更大，对运行速度也会有一定的影响。因此要确定一个最佳的 buffer size，需要平衡时间和空间来考虑。

总结与收获

通过本实验，我亲手实现了一个数据库的缓冲和数据存储管理器，学到了提高数据库查询效率的底层方法及其并模拟了实现过程。同时，我也通过本实验学习和巩固了很多操作系统和计算机组成的知识，比如块在文件中的组织方式，内存中的页、文件管理中的页、磁盘页之间的关系等等。编程实现的过程提升了我 C++ 的编程能力，巩固了对面向对象思想的理解，同时也熟练练习了通过 C++ 实现对文件读写的操作。