数据库实验报告

姓名：董建亮

学号：SA20011013

院系：计算机科学与技术学院

实验内容

参考实验说明文档，设计一个有LRU缓存机制和目录文件结构的数据库存储管理系统。

实验文件目录结构

```
.
├── CMakeLists.txt
├── data-5w-50w-zipf.txt //测试文件
├── Debug //Debug版本
|   ├── bin//存放可执行文件的文件夹
|   ├── CMakeCache.txt
|   ├── CMakeFiles
|   ├── cmake_install.cmake
|   ├── Makefile
|   └── src
├── Release //Release版本
|   ├── bin
|   ├── CMakeCache.txt
|   ├── CMakeFiles
|   ├── cmake_install.cmake
|   ├── Makefile
|   └── src
└── src //源码
    ├── buffer //缓冲区源码
    ├── CMakeLists.txt
    ├── include//头文件
    ├── storage//存储模拟实现源码
    ├── test//主函数源码
    └── UI//输出界面源码
```

实验的编译和运行

release版本

在主目录下：

Linux系统

```
$ mkdir Release
$ cd Release
$ cmake ../
$ make
$ cd bin
$ ./main
```

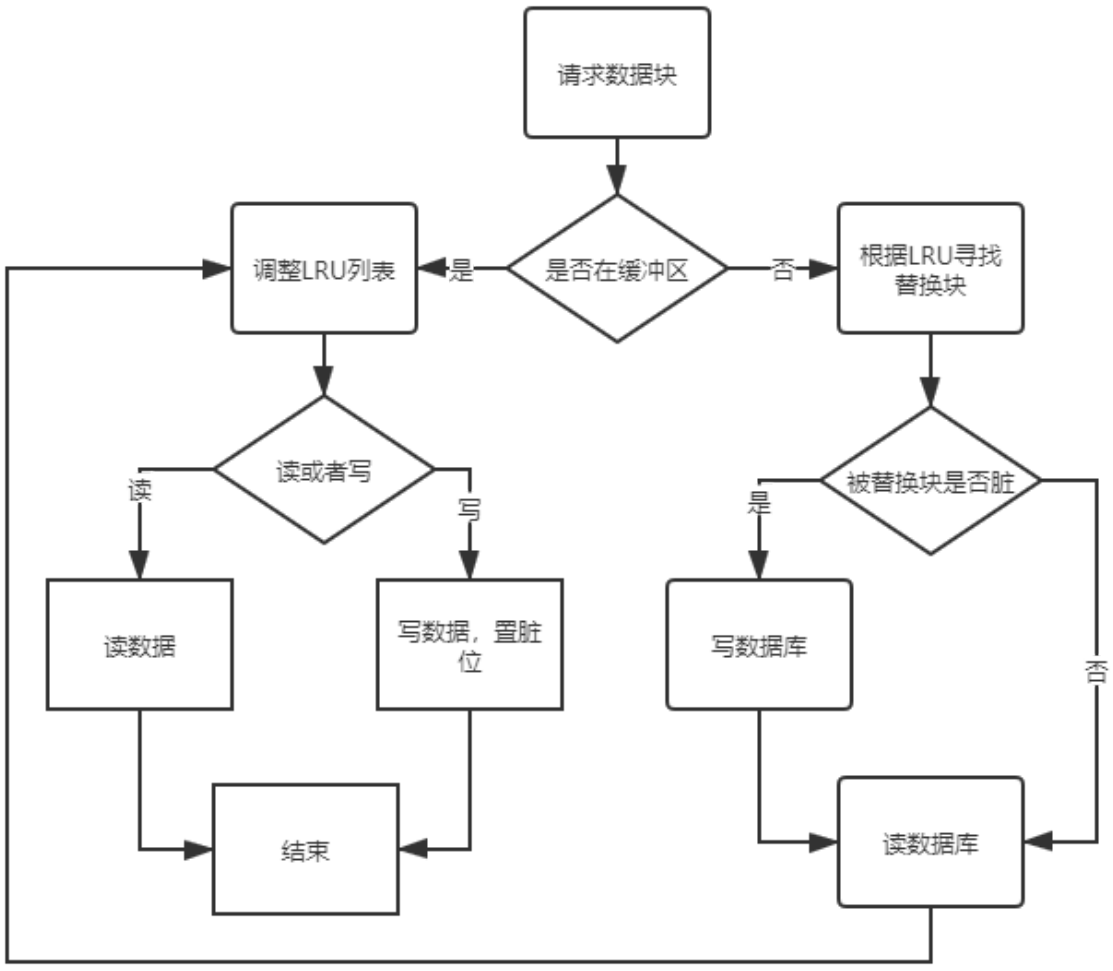
windows系统

```
mkdir Release
cd Release
cmake -G"MinGW Makefiles" ../
mingw32-make
cd bin
main.exe
```

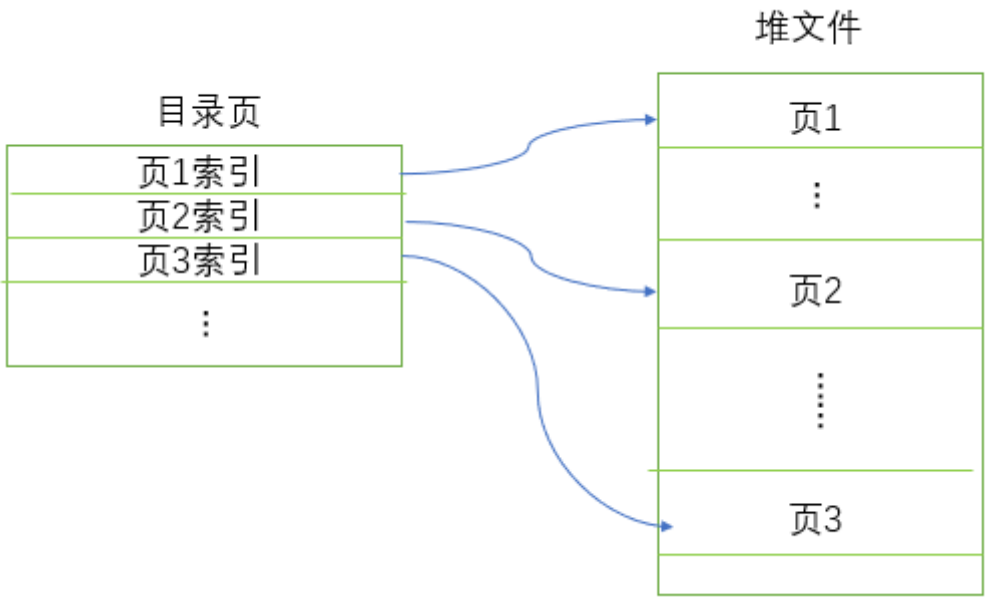
Debug版本

在cmake命令中加入参数-DCMAKE_BUILD_TYPE=Debug，其余同上。

LRU缓存机制流程



基于目录索引的堆文件结构



数据结构设计

常量数据

```
#define FrameSize 4096  //每个Frame的大小，为4096字节
#define BufSize 1024    //缓冲区容量大小，为1024
#define DataNum 50001  //页数据量
#define TestNum 500000  //测试数据量
#define TestFile "../../data-5w-50w-zipf.txt"  //测试数据文件路径
#define DBF_FILE_NAME "data.dbf" //生成的数据库文件名
```

结构体

1. 测试数据，用于保存测试数据

```
struct Test_Data
{
    bool iswrite;
    int page_num;
};
Test_Data test[TestNum];
```

2. LRU节点，采用双向链表节点的设计模式

```
struct LRU_node
{
    bool isHead;//是否为头
    bool isTail;//是否为尾
    int frameID;
    int pageID;
    LRU_node *front;//前指针
    LRU_node *next;//后指针
};
```

3. 缓冲区控制块BCB

```
struct BCB
{
    bool isHead;
    bool isTail;
    int frameID;
    int pageID;
    bool isWrite;     //数据块是否要写回硬盘
    BCB *front;
    BCB *next;
};
```

4. 哈希桶

```
/*
通过hash(pageID) = pageID % buffersize计算存在哪个桶中，每个桶都是双向链表结构。
*/
BCB *HashTable[BufSize];
```

类

LRU缓冲区

1. `void init_LRU()` 用于LRU链表的初始化
2. `bool isfull()` 和 `bool isempty()` 用于链表长度的判断
3. `LRU_element* return_head()` 和 `LRU_element* return_tail()` 用于返回链表的头节点和尾节点
4. `void insert_node(LRU_element *new_node)` 和 `void insert_new_node(int pageID, int frameID)` 用于向链表尾部添加新的节点
5. `void drop_head()` 和 `void drop_node(LRU_element *node)` 用于drop链表中节点
6. `int return_len()` 用于返回LRU链表的长度
7. `bool FULL_LRU()` 用于判断LRU链表是否已满
8. `void adjust_LRU(LRU_element *node)` 和 `void adjust_page(int page_num)` 用于链表节点位置的调整

frame_LRU

继承了LRU的类，用于存储可使用的frame的值。

1. `void init_frame()` 用于frame_LRU链表的初始化

2. `LRU_element* victim_node()` 用于找出第一个可用的frame

Hash哈希桶

1. `init_Hash()` 用于Hash的初始化
2. `int hash` 用于计算hash值，为 `pageID % buffersize`
3. `insert_BCB` 和 `drop_BCB` 用于插入和删除节点
4. `has_page` 用于判断是否在表中，为bool型
5. `find_BCB` 用于在 `has_page` 为真的时候返回该pageID所对应的节点

Disk存储类

1. `void init_disk()` 用于Disk类的初始化
2. `void disk_input(int pageID)` 和 `void disk_output(int pageID)` 用于向磁盘写入或从磁盘读出数据
3. `directory_page` 目录页，用于存放数据库文件的页的索引

DUI界面类

1. `print` 用于正常输出界面
2. `time_result` 用于时间的输出
3. `error` 用于报错界面
4. `debug` 用于调试界面
5. `rule` 用于分隔界面
6. `print_rate` 用于进度界面的显示
7. `getTime` 获取时间

实验运行测试结果

程序运行结果如下

```
dongjl@ubuntu:~/database_lab/BufferandStorageManage/D/bin$ ./main
2021-01-28 23:45:03
================================== init system ==================================
Info: Buffer Size is :1024
Info: LRU has been created.
Info: LRU has been created.
Info: Frame LRU has been created
Info: Hash has been created.
Info: disk has been created.
================================== read file data ==================================
Info: read the file done.
================================ handle data request ================================
[0%] begin to handle data.
[5%] LRU is full now.
[50%] handling data...
[100%] handle data done.
================================== print result ==================================
[Result] Buffer_Hit is: 169565
[Result] Read_IO is: 330435
[Result] Write_IO is: 172386
=================================== end ===================================
Time: time is: 0.389197s
Info: Hash has been dropped.
Info: Frame LRU has been dropped
Info: LRU has been dropped.
Info: LRU has been dropped.
```

在Release版本测试数据量的大小对运行结果的影响，得到如下数据表格

测试数据量	100k	200k	300k	400k	500k
Buffer_Hit	33675	67456	101403	135569	169565
Read_IO	66325	132544	198597	264431	330435
Write_IO	34236	68844	103393	137812	172386
Run_Time(s)	0.077297	0.154037	0.223976	0.298491	0.389197