浙江大学

MiniSQL设计报告

个人模块报告

课程名称：数据库系统

指导教师：周　波

学生姓名：肖瑞轩

学生学号：3180103127

2020 年 7月 1 日

Minisql个人实验报告

3180103127　肖瑞轩

一、 实验目的

（1）设计并实现一个精简型单用户SQL引擎(DBMS)MiniSQL，允许用户通过字符界面输入SQL 语句实现表的建立/删除；索引的建立/删除以及表记录的插入/删除/查找。

（2）通过对MiniSQL 的设计与实现，提高系统编程能力，加深对数据库系统原理的理解。

二、 系统需求

`IndexManager`: 通过B+树的结构，建立索引，实现对关键字的查找、插入和删除操作，

`ErrorReporter`: 在输入格式等出现错误时，判断错误类型并根据不同的类型进行错误信息的提示。

`main.cpp`：调用API和interpreter，完成初始页面与系统不同界面逻辑跳转的设计

`global timer`：调用内置的计时板块，进行程序计时以及全局变量定义、全局文件的资源管理

三、 实验环境

IDE&编译器：Microsoft Visual Studio 2019

四、板块设计

4.1 `BPlusTree`板块

板块目的：

B+树是平衡树数据结构，通常用于数据库和操作系统的文件系统中，其插入与修改具有较稳定的对数时间复杂度，因为数据访问时磁盘I/O耗时更加显著，B+树可以最大化内部结点的储存数据量，从而显著减少I/O耗时。

基本功能描述：

（1）创建和读写索引文件，负责B+树索引的实现，实现B+树的创建和删除，该操作由索引的定义和删除引起。

（2）通过B+树的结构，建立索引，实现对关键字的查找、插入和删除操作，实现查找记录的位置与提供范围查找功能，对外提供对应的接口。

（3） 打印B+树结点信息，保证建立的索引的稳定性，实时将建立的索引写入硬件。

程序文件：

`bplustree.h bplustree.cpp`

调用其他模块：

`BufferManager`模块、`RecordManager`模块

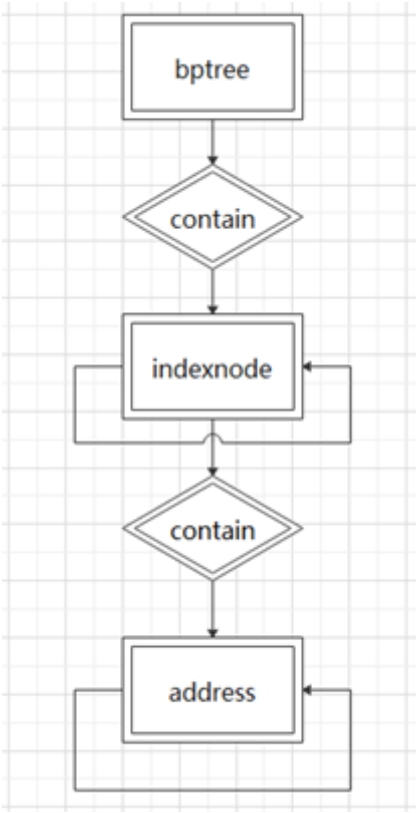
数据结构：

（1）索引头结点 `class IndexHeadNode`

（2）B+树内部节点 `class BPlusTNode`

（3）B+树结构 `class BPlusTree`

模块内关系：



具体接口：

(1)BPlustree类与接口

bplustree记录了整棵树的各项属性，由于一棵树对应一个完整的index索引，因此bplustree内有一个indexhead的索引头结点，也是b+树的根节点，此外我们还需要记录索引对应的属性、文件ID等重要的整体信息，其中数据主要存储在IndexHeadNode中，而对B+树的查找、插入、分裂、删除等操作则在BPlusTree中进行实现，使用到的接口如下所示：

```
class BPlusTree
{
    friend std::vector<RecordHead> ShowTable(std::string table_name, std::string path);
    friend RecordHead GetDbfRecord(std::string table_name, FileAddr fd, std::string path);
public:
    int File_ID;
    string IndexName;
    IndexHeadNode IndexHead;

    BPlusTree(std::string idx_name);
    // 参数：索引文件名称， 关键字类型， 记录各个类型信息数组， 记录各个字段名称信息数组
    BPlusTree(const std::string idx_name, int KeyTypeIndex, char(&_RecordTypeInfo)[RecordColumnCount], char(&_RecordColumnName)[RecordColumnCount/4*ColumnNameLength]);          // 创建索引文件的B+树
    ~BPlusTree() { }
```

```
    FileAddr Search(KeyAttr search_key);                                    //查找关键字是否已经存在
    bool Insert(KeyAttr k, FileAddr k_fd);                                  //插入关键字k
    FileAddr UpdateKey(KeyAttr a, KeyAttr k);                               // 返回关键字对应的记录地址
    FileAddr Delete(KeyAttr k);                                             //返回该关键字记录在数据文件中的地址
    //void PrintBPlusTree();
    void PrintAllLeafNode();
    IndexHeadNode* GetIdxHeadNodePtr();
    BPlusTNode* FileAddrToMemPtr(FileAddr node_fd);
  // 文件地址转换为内存指针


    FileAddr DeleteKey_Internal(FileAddr x, int i, KeyAttr key);            // x的下标为i的结点为叶子结点
    FileAddr DeleteKey_Leaf(FileAddr x, int i, KeyAttr key);                // x的下标为i的结点为叶子结点
    void InsertNotFull(FileAddr x, KeyAttr k, FileAddr k_fd);
    void NodeSplit(FileAddr x, int i, FileAddr y);                          //分裂x的孩子结点x.children[i] , y
    FileAddr Search(KeyAttr search_key, FileAddr node_fd);                  //判断关键字是否存在
    FileAddr SearchKey_Internal(KeyAttr search_key, FileAddr node_fd);
// 在内部节点查找
    FileAddr SearchKey_Leaf(KeyAttr search_key, FileAddr node_fd);          //在叶子结点查找

};
```

(2) IndexHeadNode类与接口

IndexHeadNode是索引头节点的定义，每个indexheadnode包含两个指针，第一种是root指针，用于指向父节点、右侧的兄弟节点、以及子节点；此外每个IndexHeadNode还需要记录关键字字段的位置以及使用数组来记录字段内部的信息数组，需要记录是unique的关键字字段与哪些unique上的关键字字段建立了索引每个unique索引的名字等信息。最后IndexHeadNode还需要记录这个索引是否是属于主键的索引。

```
class IndexHeadNode
{
public:
    FileAddr    root;                                   // the address of the root
    FileAddr    MostLeftNode;                           // the address of the most left node
    int         KeyTypeIndex;                           // 关键字字段的位置
    char        RecordTypeInfo[RecordColumnCount];      // 记录字段类型信息，记录各个类型信息数组，
    char        RecordColumnName[RecordColumnCount/4* ColumnNameLength];//记录各个字段名称信息数组
    bool        IsPrimary;
    bool        UniqueInfo[RecordColumnCount];
    bool        IsUniqueBuild[RecordColumnCount];
    std::vector<string> UniqueIndexName(int(RecordColumnCount));
};

```

(3) BPlusTNode类与接口

BPlusTNode是B+树结构内部每个节点的定义，在BPlusTNode内部需要存储每一个key对应的address地址或者非叶节点需要存储下一层对应的children，我们还需要存储节点的类型（根节点、内部节点、叶节点）以及总的key的数量等信息。

```
class BPlusTNode
{
public:
    BTNodeType node_type;                    //节点类型
    int key_totalnum;                       //存储的总的key的数量
    FileAddr children[MaxChildNum];         // 非叶节点，连接下一个node
    KeyAttr key[MaxKeyNum];                  //每个key的address
    FileAddr next;                           //叶节点连接下一个node
    void PrintBPlusTNode();
};
```

模块实现

使用的数据结构为b+树、b+树节点、索引头结点等节点。上述的接口部分已经明确给出各个数据结构的功能及内容。其中节点结构和索引头结点的结构属于辅助的结构，真正实现功能的是b+树结构。一个b+树对应着一个索引，我们在新建一个表的时候由于主键的关系需要新建一个索引，而我们可以自定义的给其他unique的属性也进行index的添加与删除功能。

B+树索引的功能实现分成两个层次，底层是基于b+树的实际功能的实现，在模板类里面定义了各个函数；上层函数则是为了方便api和buffer manager的调用，提供了不含模板类的接口函数，这些函数根据实际需要来调用底层函数。

我们从最简单的B+树构造函数开始，我们对于B+树的构造有两种情况，一种是通过索引名称、编号和新建索引需要的信息来新建一颗B+树。另一种情况是通过索引的名称来找到内存中某处的索引，并提取该处的地址，两种构造函数的使用分别如下：

第一种构造函数同样需要检查已有的buffer中是否已经存在这样的索引，如果不存在则进行新建，注意需要对IndexHead的各个成分进行赋值与内存初始化。

```
BPlusTree::BPlusTree(const std::string index_name, int KeyTypeIndex,
char(&mRecordTypeInfo)[RecordColumnCount], char(&mRecordColumnName)
[RecordColumnCount / 4 * ColumnNameLength], bool(&mUniqueInfo)[RecordColumnCount],
bool isprimary, bool(&mIsUniqueBuild)[RecordColumnCount]):IndexName(index_name)
{
    BUFFER &buffer = GetGlobalFileBuffer();
    auto MemFileptr = buffer[IndexName.c_str()];
    // 如果索引文件不存在则创建
    if (!MemFileptr)
    {
        buffer.CreateFile(IndexName.c_str());
        MemFileptr = buffer[IndexName.c_str()];
        // 初始化索引文件的root节点
        BPlusTNode root_node;
        assert(sizeof(BPlusTNode)<(FILE_PAGESIZE-sizeof(PAGEHEAD)));
        root_node.node_type = BTNodeType::ROOT;
        root_node.key_totalnum = 0;
        root_node.next = FileAddr{ 0,0 };
        FileAddr root_node_record= buffer[IndexName.c_str()]-
>AddRecord(&root_node, sizeof(root_node));
        // 初始化indexhead
        IndexHead.root = root_node_record;
        IndexHead.MostLeftNode = root_node_record;
        IndexHead.KeyTypeIndex = KeyTypeIndex;
        IndexHead.IsPrimary = isprimary;
        memcpy(IndexHead.RecordTypeInfo, mRecordTypeInfo, RecordColumnCount);
```

```
        memcpy(IndexHead.IsUniqueBuild, mIsUniqueBuild, RecordColumnCount);
        memcpy(IndexHead.UniqueInfo, mUniqueInfo, RecordColumnCount);
        memcpy(IndexHead.RecordColumnName, mRecordColumnName, RecordColumnCount / 4 * ColumnNameLength);
        // 为结点的地址预留空间
        memcpy(buffer[IndexName.c_str()]->GetFileFirstPage()->GetFileCond()->reserve, &IndexHead, sizeof(IndexHead));
    }
    File_ID = MemFileptr->fileId;
}
```

而下面这张构造函数就是通过索引的名称来获取已有的索引对应的物理file_id从而方便对该索引进行访问

```
BPlusTree::BPlusTree(std::string idx_name)
{
    IndexName = idx_name;
    File_ID = GetGlobalFileBuffer()[idx_name.c_str()]->fileId;
}
```

而关于B+树的操作最简单的为查找，查找主要是分为在内部节点查找与在叶节点中进行查找两部分组成。实现方式是从上往下通过递归来进行调用实现。

```
FileAddr BPlusTree::SearchKey_Internal(KeyAttr search_key, FileAddr node_fd)
{
    FileAddr fd_res{0,0};

    BPlusTNode* pNode = FileAddrToMemPtr(node_fd);
    for (int i = pNode->key_totalnum - 1; i >= 0; i--)
    {
        if (pNode->key[i] <= search_key)
        {
            fd_res = pNode->children[i];
            break;
        }
    }

    if (fd_res == FileAddr{ 0,0 })
    {
        return fd_res;
    }
    else
    {
        BPlusTNode* pNextNode = FileAddrToMemPtr(fd_res);
        if (pNextNode->node_type == BTNodeType::LEAF)
            return SearchKey_Leaf(search_key, fd_res);
        else
            return SearchKey_Internal(search_key, fd_res);
    }
}

FileAddr BPlusTree::SearchKey_Leaf(KeyAttr search_key, FileAddr node_fd)
{

    BPlusTNode* pNode = FileAddrToMemPtr(node_fd);
    for (int i = 0; i <pNode->key_totalnum; i++)
    {
        if (pNode->key[i] == search_key)
        {
```

```
            return pNode->children[i];
        }
    }
    return FileAddr{ 0,0 };
}
```

而将这两种search方式整合在一起，就形成了在整棵B+树内进行search的函数

```
FileAddr BPlusTree::Search(KeyAttr search_key, FileAddr node_fd)
{
    BPlusTNode* pNode = FileAddrToMemPtr(node_fd);
    if (pNode->node_type == BTNodeType::LEAF || pNode->node_type ==
BTNodeType::ROOT)
    {
        return SearchKey_Leaf(search_key, node_fd);
    }
    else
    {
        return SearchKey_Internal(search_key, node_fd);
    }
}
```

B+树的插入函数

B+树的插入需要分为内部节点也叶节点进行分别插入，如果是叶节点则可以选择直接插入，而如果是内部节点，需要找到对应的插入的节点以及该节点是否已满，如果节点满之后需要进行先分裂后插入的操作。如果key以及存在于B+树中，则根据对应的类型进行是否上报重复插入的处理。

```
bool BPlusTree::Insert(KeyAttr k, FileAddr k_addr)
{
    try
    {
        auto key_fd = Search(k);
        if (key_fd != FileAddr{ 0,0 })
            throw SQLError::KEY_INSERT_ERROR();
    }
    catch (const SQLError::BaseError & error)
    {
        SQLError::DispatchError(error);
        std::cout << std::endl;
        return false;
    }
    // 得到根结点的
    FileAddr root_fd = *(FileAddr*)GetGlobalFileBuffer()[IndexName.c_str()]-
>GetFileFirstPage()->GetFileCond()->reserve;
    auto proot = FileAddrToMemPtr(root_fd);
    if (proot->key_totalnum == MaxKeyNum)
    {
        // 创建新的结点 s ,作为根结点
        BPlusTNode s;
        s.node_type = BTNodeType::INTERNAL;
        s.key_totalnum = 1;
        s.key[0] = proot->key[0];
        s.children[0] = root_fd;
        FileAddr s_fd = GetGlobalFileBuffer()[IndexName.c_str()]->AddRecord(&s,
sizeof(BPlusTNode));
        // 将新的根节点文件地址写入
        *(FileAddr*)GetGlobalFileBuffer()[IndexName.c_str()]->GetFileFirstPage()-
>GetFileCond()->reserve = s_fd;
```

```
        GetGlobalFileBuffer()[IndexName.c_str()]->GetFileFirstPage()->isModified = true;
        //将旧的根结点设置为叶子结点
        auto pOldRoot = FileAddrToMemPtr(root_fd);
        if (pOldRoot->node_type == BTNodeType::ROOT)
            pOldRoot->node_type = BTNodeType::LEAF;
        // 先分裂再插入
        NodeSplit(s_fd, 0, s.children[0]);
        InsertNotFull(s_fd, k, k_addr);
    }
    else
    {
        InsertNotFull(root_fd, k, k_addr);
    }
    return true;
}

void BPlusTree::InsertNotFull(FileAddr x, KeyAttr k, FileAddr k_addr)
{
    auto nodex = FileAddrToMemPtr(x);
    int i = nodex->key_totalnum - 1;
    if (nodex->node_type == BTNodeType::INTERNAL)//是内部节点 判断插的位置与是否分裂
    {
        while (i >= 0&&k< nodex->key[i])i--;
        // 如果插入的值比内节点的值还小
        if (i < 0) {
            i = 0;
            nodex->key[i] = k;
        }
        assert(i >= 0);
        FileAddr childaddr = nodex->children[i];
        auto ptrchildaddr = FileAddrToMemPtr(childaddr);
        if (ptrchildaddr->key_totalnum == MaxKeyNum)
        {
            NodeSplit(x, i, childaddr);
            if (k >= nodex->key[i + 1])
                i += 1;
        }
        InsertNotFull(nodex->children[i], k, k_addr);
    }
    else// 如果该结点是叶子结点，直接插入
    {
        while (i >= 0 && k < nodex->key[i])
        {
            nodex->children[i + 1] = nodex->children[i];
            nodex->key[i + 1] = nodex->key[i];
            i--;
        }
        nodex->children[i + 1] = k_addr;
        nodex->key[i + 1] = k;
        nodex->key_totalnum += 1;
    }
}
```

其中内部节点满的时候，需要对满的节点进行分裂的操作，分裂的函数为经典的B+树的节点分裂算法。

```
void BPlusTree::NodeSplit(FileAddr x, int i, FileAddr c)
{
    auto pMemPageX = GetGlobalClock()->GetMemAddr(File_ID, x.filePageID);
```

```
    auto pMemPageY = GetGlobalClock()->GetMemAddr(File_ID, c.filePageID);
    pMemPageX->isModified = true;
    pMemPageY->isModified = true;

    BPlusTNode* nodex = FileAddrToMemPtr(x);
    BPlusTNode* nodei = FileAddrToMemPtr(c);
    BPlusTNode z;          // 分裂出来的新结点
    FileAddr z_fd;     // 新结点的文件内地址

    z.node_type = nodei->node_type;
    z.key_totalnum = MaxKeyNum / 2;

    // 将y结点的一般数据转移到新结点
    for (int k = MaxKeyNum / 2; k < MaxKeyNum; k++)
    {
        z.key[k - MaxKeyNum / 2] = nodei->key[k];
        z.children[k - MaxKeyNum / 2] = nodei->children[k];
    }
    nodei->key_totalnum = MaxKeyNum / 2;
    int j;
    for (j = nodex->key_totalnum - 1; j > i; j--)
    {
        nodex->key[j + 1] = nodex->key[j];
        nodex->children[j + 1] = nodex->children[j];
    }

    j++;
    nodex->key[j] = z.key[0];

    if (nodei->node_type == BTNodeType::LEAF)
    {
        z.next = nodei->next;
        z_fd = GetGlobalFileBuffer()[IndexName.c_str()]->AddRecord(&z, sizeof(z));
        nodei->next = z_fd;
    }
    else
        z_fd = GetGlobalFileBuffer()[IndexName.c_str()]->AddRecord(&z, sizeof(z));
    nodex->children[j] = z_fd;
    nodex->key_totalnum++;
}
```

B+树中的删除操作与查找操作类似，需要分内部节点与叶节点两部分进行考虑，其中每一步的细节较多，需要特别注意。

```
FileAddr BPlusTree::DeleteKey_Internal(FileAddr x, int i, KeyAttr key)//i-th children of the x file
{
    BPlusTNode* nodex = FileAddrToMemPtr(x);
    BPlusTNode* childnode = FileAddrToMemPtr(nodex->children[i]);
    FileAddr fd_res;
    if (childnode->node_type == BTNodeType::LEAF)
    {
        fd_res = DeleteKey_Leaf(x, i, key);
    }
    else
    {
        int j = childnode->key_totalnum - 1;
        while (childnode->key[j] > key)j--;
        assert(j >= 0);
        fd_res = DeleteKey_Internal(nodex->children[i], j, key);
```

```
    }
    if (childnode->key_totalnum >= MaxKeyNum / 2)
        return fd_res;
    // 如果删除后的关键字个数不满足B+树的规定，向兄弟结点借用key
    // 如果右兄弟存在且有富余关键字
    if ((i <= nodex->key_totalnum - 2) && (FileAddrToMemPtr(nodex->children[i + 1])->key_totalnum > MaxKeyNum / 2))
    {
        auto RBrother = FileAddrToMemPtr(nodex->children[i + 1]);
        // 借来的关键字
        auto key_bro = RBrother->key[0];
        auto fd_bro = RBrother->children[0];


        // 更新右兄弟的索引结点
        nodex->key[i + 1] = RBrother->key[1];
        // 跟新右兄弟结点
        for (int j = 1; j <= RBrother->key_totalnum - 1; j++)
        {
            RBrother->key[j - 1] = RBrother->key[j];
            RBrother->children[j - 1] = RBrother->children[j];
        }
        RBrother->key_totalnum -= 1;

        // 更新本叶子结点
        childnode->key[childnode->key_totalnum] = key_bro;
        childnode->children[childnode->key_totalnum] = fd_bro;
        childnode->key_totalnum += 1;

        return fd_res;
    }
    // 如果左兄弟存在且有富余关键字
    if (i > 0 && FileAddrToMemPtr(nodex->children[i - 1])->key_totalnum > MaxKeyNum / 2)
    {
        auto LBrother = FileAddrToMemPtr(nodex->children[i - 1]);
        // 借来的关键字
        auto key_bro = LBrother->key[LBrother->key_totalnum - 1];
        auto fd_bro = LBrother->children[LBrother->key_totalnum - 1];

        // 更新左兄弟结点
        LBrother->key_totalnum -=1;

        // 更新本结点
        nodex->key[i] = key_bro;
        for (int j = childnode->key_totalnum - 1; j >= 0; j--)
        {
            childnode->key[j + 1] = childnode->key[j];
            childnode->children[j + 1] = childnode->children[j];
        }
        childnode->key[0] = key_bro;
        childnode->children[0] = fd_bro;

        childnode->key_totalnum += 1;

        return fd_res;
    }

    // 若右兄弟存在将其合并
    if (i < nodex->key_totalnum - 1)
    {
```

```
        auto RBrother = FileAddrToMemPtr(nodex->children[i + 1]);
        for (int j = 0; j < RBrother->key_totalnum; j++)
        {
            childnode->key[childnode->key_totalnum] = RBrother->key[j];
            childnode->children[childnode->key_totalnum] = RBrother->children[j];
            childnode->key_totalnum++;
        }

        // 更新next
        childnode->next = RBrother->next;
        // 删除右结点
        GetGlobalFileBuffer()[IndexName.c_str()]->DeleteRecord(&nodex->children[i + 1], sizeof(BPlusTNode));
        // 更新父节点索引
        for (int j = i + 2; j < nodex->key_totalnum; j++)
        {
            nodex->key[j - 1] = nodex->key[j];
            nodex->children[j - 1] = nodex->children[j];
        }
        nodex->key_totalnum--;
    }
    else
    {// 将左结点合并
        auto LBrother = FileAddrToMemPtr(nodex->children[i - 1]);
        for (int j = 0; j < childnode->key_totalnum; j++)
        {
            LBrother->key[LBrother->key_totalnum] = childnode->key[j];
            LBrother->children[LBrother->key_totalnum] = childnode->children[j];
            LBrother->key_totalnum++;
        }

        // 更新next
        LBrother->next = childnode->next;

        // 删除本结点
        GetGlobalFileBuffer()[IndexName.c_str()]->DeleteRecord(&nodex->children[i], sizeof(BPlusTNode));
        // 更新父节点索引
        for (int j = i + 1; j < nodex->key_totalnum; j++)
        {
            nodex->key[j - 1] = nodex->key[j];
            nodex->children[j - 1] = nodex->children[j];
        }
        nodex->key_totalnum--;
    }
    return fd_res;
}

// 假设待删除的关键字已经存在
FileAddr BPlusTree::DeleteKey_Leaf(FileAddr x, int i, KeyAttr key)
{
    auto nodex = FileAddrToMemPtr(x);
    auto nodei = FileAddrToMemPtr(nodex->children[i]);
    FileAddr fd_res;
    int j = nodei->key_totalnum - 1;
    while (nodei->key[j] != key)j--;
    assert(j >= 0);
    fd_res = nodei->children[j];
    // 删除叶节点中最小的关键字，更新父节点
    if (j == 0)
    {
```

```
        nodex->key[i] = nodei->key[j+1];
    }

    j++;
    while (j <= nodei->key_totalnum - 1)
    {
        nodei->children[j - 1] = nodei->children[j];
        nodei->key[j - 1] = nodei->key[j];
        j++;
    }
    nodei->key_totalnum -= 1;
    return fd_res;
}

```

4.2ErrorReporter异常处理板块

基本功能描述：

（1）出现错误提示的时候中止现有的工作，调用错误处理函数

（2）对不同的错误进行分类，对应不同的报错提示。

具体接口：

我们使用创建新的namespace的姓名空间的方式来对其中的所有的类进行调用，错误主要分为文件的读写错误、索引的查找、插入与删除错误、内存分配错误与输入格式错误等多类错误，具体使用到的接口如下：

```
namespace SQLError
{

    extern std::fstream log_file;
    class BaseError
    {
    public:
        virtual void PrintError()const;
        virtual void WriteToLog()const;
    protected:
        std::string ErrorInfo;
        std::string ErrorPos;

    };
    // 错误处理函数
    void DispatchError(const SQLError::BaseError &error);
    // 派生类错误
    class LSEEK_ERROR :public BaseError
    {
    public:
        LSEEK_ERROR();
    };
    // 文件读错误
    class READ_ERROR :public BaseError
    {
    public:
```

```
        READ_ERROR();
    };
    // 文件写错误
    class WRITE_ERROR :public BaseError
    {
    public:
        WRITE_ERROR();
    };
    // 文件名转换错误
    class FILENAME_CONVERT_ERROR :public BaseError
    {
    public:
        FILENAME_CONVERT_ERROR();
    };
    // 索引文件插入关键字失败
    class KEY_INSERT_ERROR :public BaseError
    {
    public:
        KEY_INSERT_ERROR();
    };
    // B+树的度偏大
    class BPLUSTREE_DEGREE_TOOBIG_ERROR :public BaseError
    {
    public:
        BPLUSTREE_DEGREE_TOOBIG_ERROR();
    };
    // 关键字名字长度超过限制
    class KeyAttr_NameLength_ERROR :public BaseError
    {
    public:
        KeyAttr_NameLength_ERROR();
    };
    class CMD_FORMAT_ERROR :public BaseError
    {
    public:
        CMD_FORMAT_ERROR(const std::string s = std::string(""));
        virtual void PrintError()const;
    protected:
        std::string error_info;
    };

    class TABLE_ERROR :public BaseError
    {
    public:
        TABLE_ERROR(const std::string s = std::string(""));
        virtual void PrintError()const;
    protected:
        std::string error_info;
    };
}
```

报错输出的汇总表格

错误类型	报错信息
输入格式错误	Command format error,please try again!
读文件错误	(READ_FAILED)Illegal page number (less than zero)

错误类型	报错信息
写文件错误	The file handle is invalid or the file is not opened for writing
文件名转换错误	File name convert failed
插入错误	Key Word Insert Failed! The record that to inset has been excisted!
长度匹配错误	KeyAttr name length flowover,it may be happen in where you set the record's key!
运算符作物	Table Operator Error!

在基本操作是否成功/失败也会出现对应的报错或提醒。

4.3 全局常量与时间板块

基本功能描述：

(1)定义minisql中可能使用到的计时板块

(2)封装一些工程全局使用的常数变量

常数定义：

```
constexpr int FILE_PAGESIZE = 8192;                    // 内存页(==文件页)大小
constexpr int MEM_PAGEAMOUNT = 4096;                   // 内存页数量
constexpr int MAX_FILENAME_LEN = 256;                  // 文件名（包含路径）最大长度
constexpr int RecordColumnCount = 12 * 4;              // 记录字段数量限制,假设所有字段都是字符数组，一个字符数组字段需要4个字符->CXXX
constexpr int ColumnNameLength = 16;                   // 单个字段名称长度限制
constexpr int bptree_t = 40;                           // B+tree's degree, bptree_t >= 2
constexpr int MaxKeyNum = 2 * bptree_t;                // the max number of keys in a b+tree node
constexpr int MaxChildNum = 2 * bptree_t;              // the max number of child in a b+tree node
```

具体接口：

```
enum class CmdType
{
    TABLE_CREATE, TABLE_DROP, TABLE_SHOW, TABLE_SELECT, TABLE_INSERT, TABLE_UPDATE, TABLE_DELETE,
    DB_CREATE, DB_DROP, DB_SHOW, DB_USE,
    QUIT, HELP

};
class SQLTimer;
SQLTimer& GetTimer();                                  // 全局计时器
class SQLTimer
{
public:
    void Start();
    void Stop();
    double TimeSpan();                                 // 返回经过的时间，单位：second
    void PrintTimeSpan();                              // 打印时间
private:
```

```
    steady_clock::time_point start_t;
    steady_clock::time_point stop_t;
    duration<double> time_span;
    const static unsigned int precision = 8;          // 输出时间的小数位精度
};

```

五、模块测试

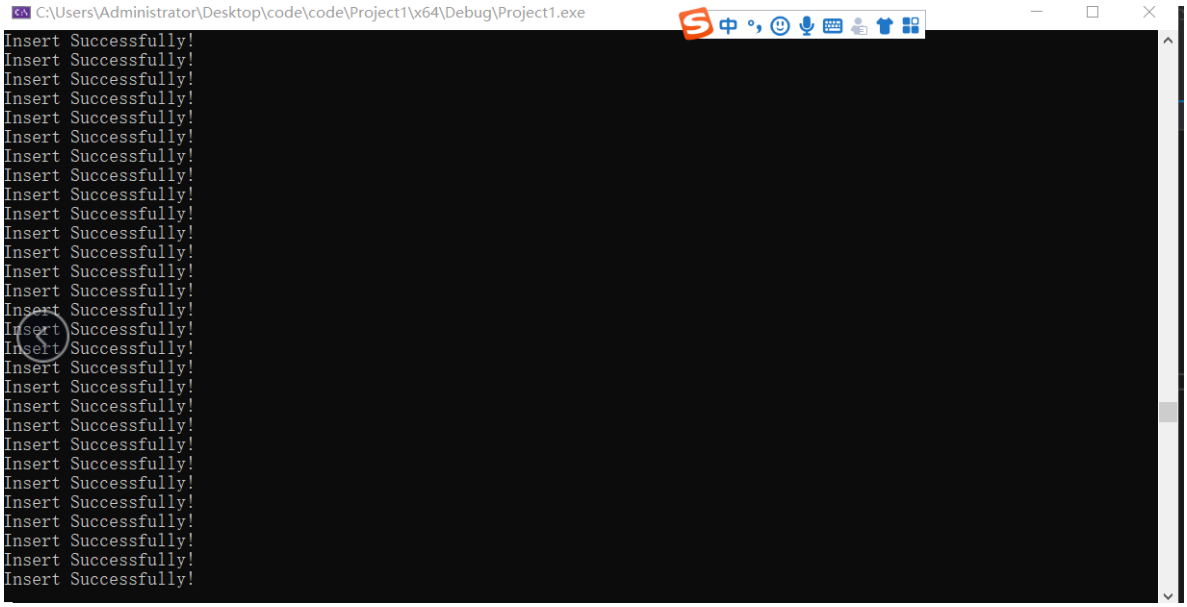
关于create index和drop index的过程，需要得到具体的record文件，因此这部分的测试需要结合.data文件得到。

此外我们需要对index中连续多次insert的性能进行大数据量的压力检测。我们使用连续insert之后再进行查找的方式对`IndexManager`进行检测。我们选取了1500条数据进行对表中的连续插入，然后对其中的数据进行查找操作。

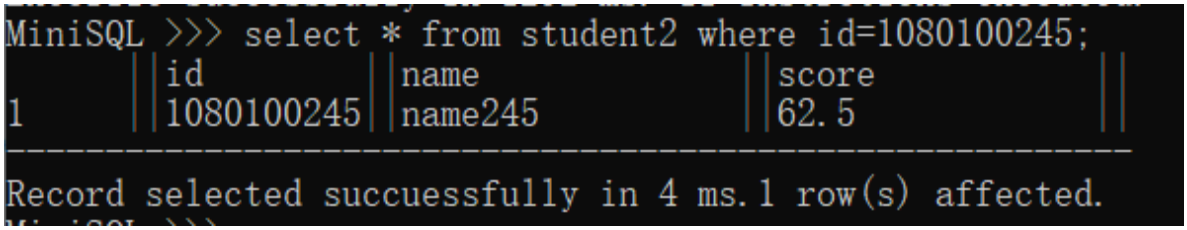
插入数据一览（部分）

```
insert into student2 values(1080103000,'name3000',85.5);
insert into student2 values(1080103001,'name3001',77.5);
insert into student2 values(1080103002,'name3002',86.5);
insert into student2 values(1080103003,'name3003',70);
insert into student2 values(1080103004,'name3004',90.5);
insert into student2 values(1080103005,'name3005',79);
insert into student2 values(1080103006,'name3006',71);
insert into student2 values(1080103007,'name3007',62);
insert into student2 values(1080103008,'name3008',64.5);
insert into student2 values(1080103009,'name3009',75.5);
insert into student2 values(1080103010,'name3010',69.5);
insert into student2 values(1080103011,'name3011',64.5);
insert into student2 values(1080103012,'name3012',61.5);
insert into student2 values(1080103013,'name3013',89.5);
insert into student2 values(1080103014,'name3014',98.5);
insert into student2 values(1080103015,'name3015',74.5);
insert into student2 values(1080103016,'name3016',62.5);
insert into student2 values(1080103017,'name3017',67);
insert into student2 values(1080103018,'name3018',60);
insert into student2 values(1080103019,'name3019',94);
```

插入结果



查找结果



重复多次测试，确保索引操作的正确性和稳定性。

六、实验心得与总结

本次大作业有比较大的挑战性，给我带来了很多的思考和感悟。

这次的编程目标相对以往较大，debug过程时间较长，给我的工程能力和代码能力带来了一定的考验。锻炼了我的编程能力和debug分析处理能力。较大的工程量也增强了我的结构分析和数据结构构造能力，对于错误的调试能力也得到了进一步的锻炼。对于数据库方面，由于index的原理和数据库课程中学到的知识也有所关联，因此我也更进一步地了解到了数据库索引的建立和利用索引处理问题的高效性。并且对如何使用索引有了更深的体会。

在逐步完成工程的过程中，和队友共同处理问题，提升了我的团队协作能力，以及面对问题如何更好沟通的能力。同时，自己出现的问题也让自己意识到了知识方面的不足，这激励着我需要不断拓宽学习面。通过和队友一起探讨和学习别的模块的具体设计和实现，我也对整个`minisql`有了比较清晰地认知，加深了我对数据库一些概念与流程的理解，可以说是收获颇丰。