浙江大学

题　　目　Minisql设计报告

——总体设计报告

课程名称　数据库系统

指导老师　周　波　李奇莲

小组成员　肖瑞轩　3180103127

傅淄洋　3180105493

孙家阳　3180103481

目录

Minisql设计报告

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，提高学生的系统编程能力，加深对数据库系统原理的理解。

二、系统需求

（1）数据类型：只要求支持三种基本数据类型：int，char(n)/varchar，float，其中char(n)满足 1 <= n <= 255 。

（2）表定义：一个表最多可以定义32 个属性，各属性可以指定是否为unique；支持unique 属性的主键定义。

（3）索引的建立和删除，对于表的主键自动建立B+树索引，对于声明为unique 的属性可以通过SQL 语句由用户指定建立/删除B+树索引（因此，所有的B+树索引都是单属性单值的）。

（4）查找记录：可以通过指定用and 连接的多个条件进行查询，支持等值查询和区间查询。

（5）插入和删除记录：支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。(where 条件是范围时删除多条)

三、实验环境

编程语言：C/C++

编译环境：Visual studio 2019

四、小组分工

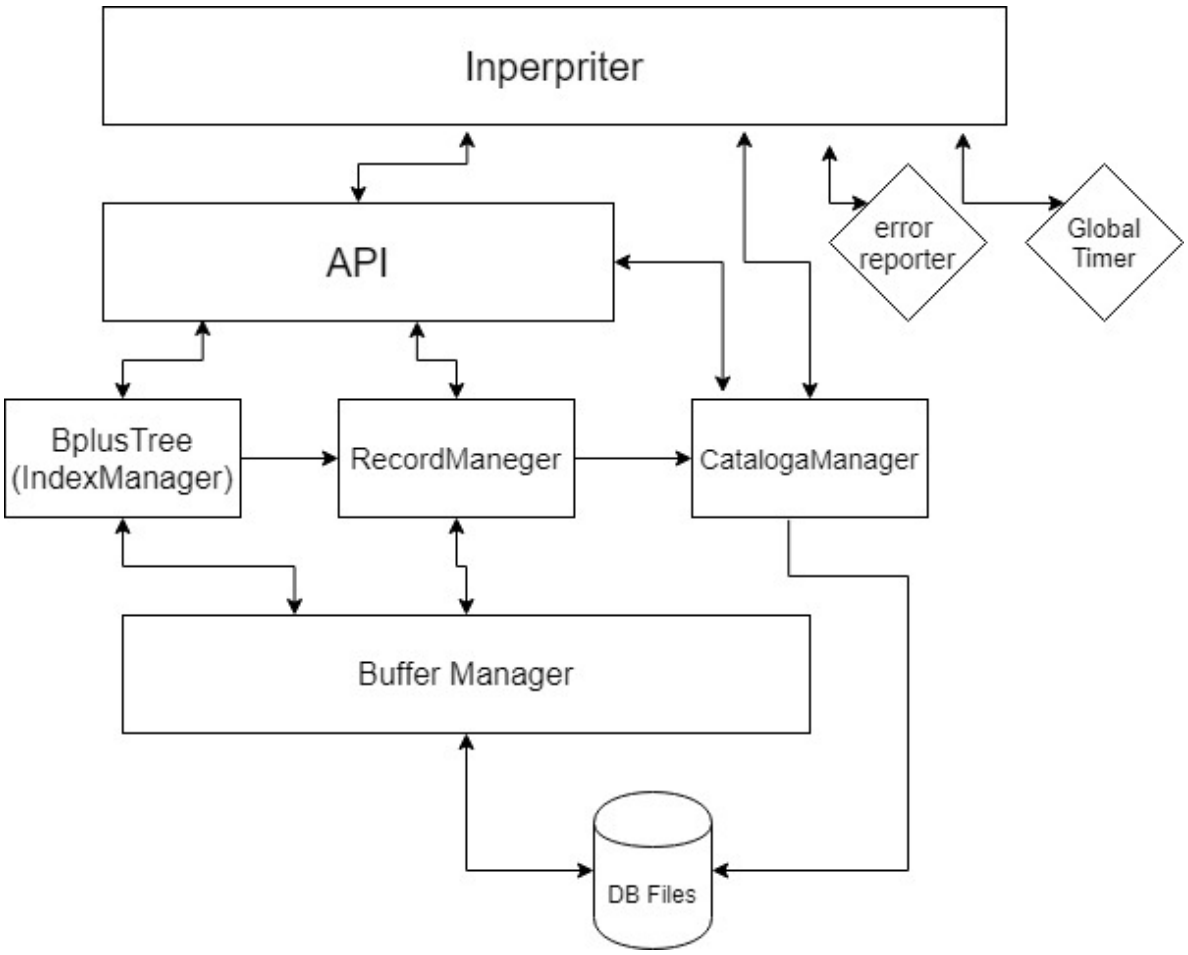
肖瑞轩：IndexManager、ErrorReporter、计时器与main.cpp模块，API和Interpreter部分接口，总体报告的编写与排版。

傅淄洋：BufferManager、CatalogManager、RecordManager模块的编写，API和Interpreter部分接口的设计与调试改进。

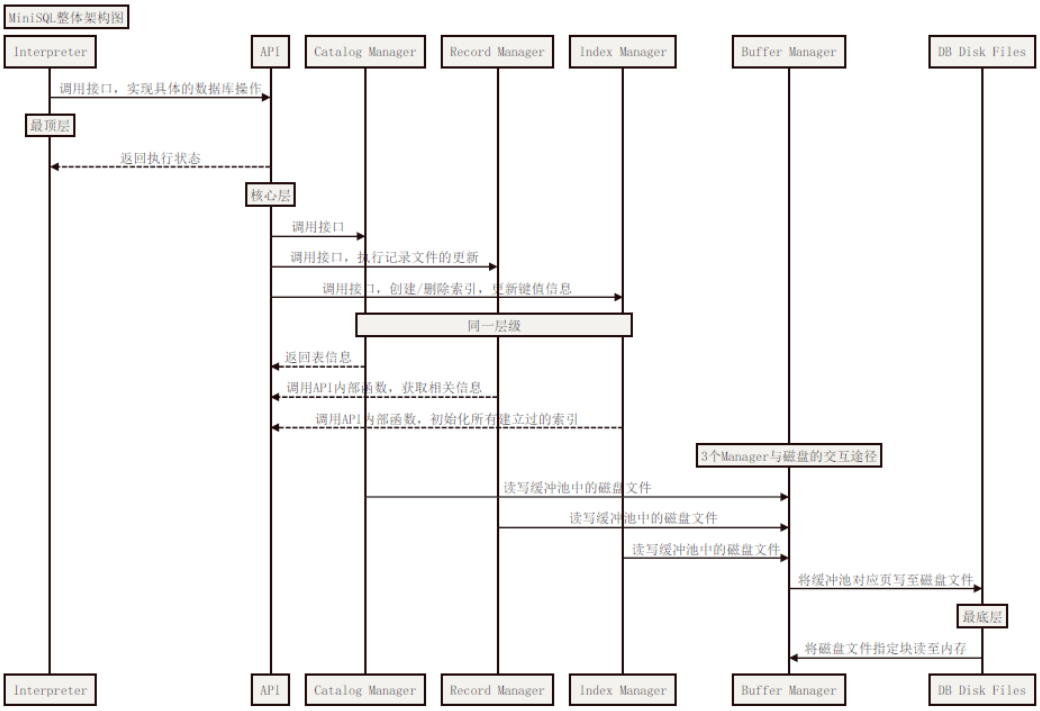
孙家阳：API和Interpreter板块的进一步修改、调试。

五、系统设计

5.1 系统总体结构



我们的Minisql主要由API、 Interpreter、 RecordManager、 BplusTree （即IndexManager）、CatalogManager、BufferManager、ErrorReporter、Globaltimer等几个板块构成，板块直接的关系如上图所示，各模块的交互关系如下图所示。



接下来我们将对系统中的各版块进行具体的阐释。

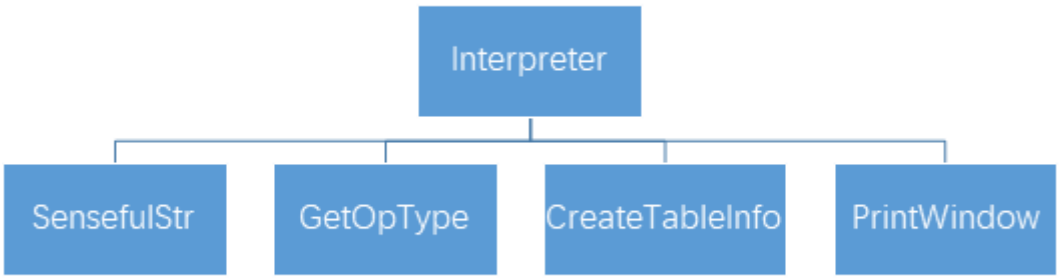
5.2 Interpreter板块

基本功能描述:

（1）将用户输入的语句切割成字符串数组SensefulStr作为指令，并剔除无关信息（如，；等），

（2）根据字符串数组用GetOpType得到该指令类型，创建对应指令的信息存储结构CreateTableInfo，然后调用API执行指令Interpreter，最后反馈执行结果PrintWindow

模块框架:



程序文件: `Interpreter.h Interpreter.cpp`

具体接口:

（1）字符串数组`SensefulStr`

```
class SensefulStr
{
```

```
public:
    SensefulStr(std::string srcstr = "");
    void SetSrcStr(std::string _srcstr);

    std::vector<std::string> GetSensefulStr()const;
private:
void Parse();                                                    // 解析命令为有意字串

    std::string src_str;                                         // 原始命令字符串
    std::vector<std::string> sen_str;                            // 解析后的又一字串
    std::string key_char = ";,()=<>\012\015\040";
    bool IsKeyChar(char c);
};

```

（i）成员变量：

src_str：包含一个命令原始命令字符串，调用getline（），若一条命令包含多行，则仍将其整合为一个字符串

sen_str：经过切割处理后的字符串数组，包含一组命令

key_char：命令中的无关字符，如“,;”等都会被过滤，注意只支持英文符号

（ii）构造函数

调用parse（）函数对原命令字符串进行切割，过滤无关信息，并转化为数组

（iii）其他成员函数

GetSensefulStr：返回sen_str

Parse：去除原字符串中的";,()=<>"及单/双引号和空白符，将其转化为字符串数组并保存在sen_str中

SetSrcStr：对src_str赋值

（II）GetOpType判断指令类型函数

根据sen_str字符串数组，并结合所要求支持的SQL指令，判断指令类型，目前支持以下类型

```
enum class CmdType
{
    TABLE_CREATE, TABLE_DROP, TABLE_SELECT, TABLE_INSERT, TABLE_DELETE,
    DB_CREATE,DB_USE,
    INDEX,EXECFILE, QUIT, HELP
};

```

表的创建/删除，记录的选择/插入/删除，数据库的创建/使用，索引（含创建与使用），退出，提供帮助信息。若用户输入指令不属于上述类型，则抛出异常

创建指令的信息存储结构CreateTableInfo模块

字符串数组SensefulStr是初步加工的指令，接下来要对其进一步修饰加工，提供API能接受执行的指令信息，即根据不同类型的指令创建合适的结构存储。主要包含以下函数：

```
//数据库操作相关信息,返回数据库名称
```

```
// 创建数据库
std::string CreateDbInfo(std::vector<std::string> sen_str);
// 使用数据库
std::string UseDbInfo(std::vector<std::string> sen_str);

// 表操作相关信息，返回存储有效信息的struct
//创建表
TB_Create_Info CreateTableInfo(std::vector<std::string> sen_str);
//删除表
std::string DropTableInfo(std::vector<std::string> sen_str);

//记录操作相关信息，返回存储有效信息的struct
//插入记录
TB_Insert_Info CreateInsertInfo(std::vector<std::string> sen_str);
//选择记录
TB_Select_Info TableSelectInfo(std::vector<std::string> sen_str);
//删除记录
TB_Delete_Info TableDeleteInfo(std::vector<std::string> sen_str);

//索引操作相关信息，生成所需创建/删除的索引信息
Index_Info IndexInfo(std::vector<std::string> sen_str);

```

在创建信息存储结构时主要考虑以下两方面：

1）指令格式是否完整且正确，若不是，抛出自定义的SQLError:: CMD_FORMAT_ERROR异常

2）初步检查该指令的操作是否满足某些字段的属性约束，如primary key约束，unique key约束等，若不是，也会抛出上述异常

反馈执行结果PrintWindow接口

```
#define PRINTLENGTH 63
class PrintWindow
{
public:
    void CreateTable(bool is_created);
    void DropTable(bool is_dropped);
    void SelectTable(SelectPrintInfo select_table_print_info);
    void InsertRecord(bool is_inserted);
    void CreateDB(bool is_created);
    void UseDB(bool isUsed);
    void DeleteTable(bool isDeleted);
    void doIndex(bool isDone);
private:
    void Print(int len, std::string s); // 打印 |xxxx        | 其中竖线内长度为 len
    int GetColumnLength(std::string name,std::vector<std::string> col_name,
std::vector<int> col_len);
};
```

相关接口：

DropTable：判断表是否删除成功，

SelectTable：判断记录是否筛选成功

InsertRecord：判断记录是否插入成功

CreateDB：判断创建当前数据库是否成功

UseDB：判断使用当前数据库是否成功

DeleteTable：判断记录是否删除成功

doIndex：判断索引的建立/删除操作是否成功

Print：格式化打印，使指令的返回结果分栏并列对齐，每栏长度为len，同时打印栏分隔符“|”

GetColumnLength：获得当前记录该字段的值的长度，若大于该字段所允许的最大长度，则返回后者

5.3 API板块:

基本功能描述:

向下：根据Interpreter层解释生成的命令内部形式，调用Catalog Manager提供的信息进行进一步的验证及确定执行规则，然后调用Record Manager、Index Manager、Catalog Manager及B+树模块提供的相应接口执行各SQL语句及命令语句。

向上：API模块进一步进行语法与逻辑检查，抛出可能的异常，提供执行 SQL 语句的接口供 Interpreter 层调用。

调用其他模块:

Buffer模块 B+树模块 Record模块 Check模块

具体接口:

```
/ 创建数据库
bool CreateDatabase(std::string database_name, CatalogPosition &cp);

// 选择数据库
bool UseDatabase(std::string db_name, CatalogPosition &cp);

// 创建表bool CreateTable(TB_Create_Info tb_create_info, std::string path =
std::string("./"));

// 删除表
bool DropTable(std::string table_name, std::string path = std::string("./"));

// 插入记录
bool InsertRecord(TB_Insert_Info tb_insert_info, std::string path =
std::string("./"));

// 选择记录
SelectPrintInfo SelectTable(TB_Select_Info tb_select_info, std::string path =
std::string("./"));

// 删除记录
bool DeleteTable(TB_Delete_Info tb_delete_info, std::string path =
std::string("./"));

//插入/删除索引
bool doIndex(Index_Info index_info, std::string path = std::string("./"));

```

其余接口为API内部调用，会在介绍主要接口时附带介绍

（I）CreateDatabase创建数据库

判断当前路径下数据库是否存在，若存在，则返回不成功，并输出错误信息；若不存在则创建子目录作为数据库

（II）UseDatabase使用数据库

先判断数据库是否存在，若存在，将数据库根目录路径定位到此数据库下，若不存在，返回不成功并输出错误信息

（III）CreateTable创建表

先检查信息，调用Record模块提供的Check_TB_Create_Info，检查检查创建信息（如字段数量是否溢出，primary key是否唯一，字段类型是否支持等），以及当前目录是否在某个数据库中。

然后调用B+树模块创建索引文件.idx（primary key会默认创建索引，该索引不能被用户创建或删除，若表没有指定primary key，则指定第一个字段为primary key），并调用buffer模块创建数据文件.dbf

（IV）DropTable删除表

先判断所需删除的表是否存在，然后判断该表对应的数据文件是否已经被打开使用（若已打开，则需要先丢弃在内存中的文件页，避免程序结束再次写回）。

找到所有本表数据文件.dbf及本表上的索引文件.idx，删除所有文件

（V）InsertRecord插入记录

先检查信息，调用Record模块提供的Check_TB_Insert_Info，检查检查创建信息以及所操作的表是否在当前目录中。

（VI）SelectTable选择记录

先检查信息，调用Record模块提供的Check_TB_ Select_Info，检查检查创建信息以及所操作的表是否在当前目录中。

判断是否有where条件，若无则遍历，若有则调用Search函数。

最后函数返回select后得到的所有结果。

（VII）DeleteTable删除记录

操作类似SelectTable，删除所有指定结果，在查找时调用Search函数，同样区分索引查找与遍历查找。

（V）doIndex创建/删除索引

首先进行异常处理，试图删除不存在的索引和创建已存在的索引都会抛出异常并返回。然后判断该指令为创建还是删除，在对应目录下创建/删除对应表的索引。

5.4 CatalogManager板块

基本功能描述：

（1）负责管理数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。

（2）负责管理表中每个字段的定义信息，包括字段类型、是否唯一等。

（3）负责管理数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

程序文件：

catalog.h catalog.cpp

调用其他模块：

Record模块 B+tree 模块

具体接口：

（1）数据库信息类

这个类主要管理各个数据库的路径，同时直接用_access函数判断是否存在相应数据库，然后返回值。而关于表信息的管理我们分为了几类： 创建表信息，插入表信息，删除表信息，更新表信息，查找表信息。

```
class CatalogPosition
{
    friend bool UseDatabase(std::string db_name, CatalogPosition &cp);
public:
    CatalogPosition();
    bool ResetRootCatalog(std::string root_new);  // 重置根目录
    void SwitchToDatabase();// 转到数据库列表目录下
    bool SwitchToDatabase(std::string db_name);// 转到具体的数据库下
    std::string GetCurrentPath()const;
    std::string GetRootPath()const;
    std::string SetCurrentPath(std::string cur);
    bool GetIsInSpeDb() { return isInSpeDb; }
    bool SetInInSpeDb(bool new_b) { isInSpeDb = new_b; return new_b; }
private:
    static bool isInSpeDb;          //是否在某个具体的数据库目录下
    std::string root; // 根目录，数据库文件的保存位置
    std::string current_catalog;
};
```

（2）索引信息管理类

该索引文件信息类包含了所有需要包含的内容，其实不止索引，也包含了其他内容。如字段名称等，也可以判断一个字段是否为主键。通过record，获得各个字段的信息。实现相对较为简单，大多的获取信息和判断是否存在，没有特别的算法需要说明。

```
// 索引文件头信息管理类
class TableIndexHeadInfo
{
public:
    TableIndexHeadInfo(BPlusTree &_tree) :tree(_tree) {}
    // 表的字段个数
    size_t GetColumnCount()const;
    // 各个字段名字
    std::vector<std::string> GetColumnNames()const;
    // 各个字段类型
    std::vector<Column_Type> GetColumnType()const;
    Column_Type GetColumnType(std::string column_name)const;
    // 各个字段的大小
    std::vector<int> GetColumnSize()const;
    // 第i个字段的数据大小
    int GetColumnSizeByIndex(int i)const;
    // 主键字段的索引
    int GetPrimaryIndex()const;
    // 判断该字段名是不是表的字段
```

```
    bool IsColumnName(std::string column_name)const;
    // 返回该字段在所有字段中的索引位置
    int GetIndex(std::string column_name)const;
    // 判断字段名是否为主键字段
    bool IsPrimary(std::string column_name)const;
    // 返回给定字段名字段距离数据头地址的偏移
    int GetColumnOffset(std::string column_name);
private:
    BPlusTree &tree;
};

```

5.5 RecordManager板块

基本功能描述：

（1）实现数据文件的创建与删除（由表的定义与删除引起）。

（2）实现记录的插入、删除、更新与查找操作。

程序文件：

Record.h Record.cpp

调用其他模块：

BufferManager模块

数据结构：

（1）KeyAttr 该类用来储存不同类型的变量，同时可以进行各个变量之间的比较操作。

（2）Column_Cell 定义每个字段的单元数据字段单元记录了该字段的数据类型、数据的值、以及该字段的下一个字段指针，如果保存的字段是字符串类型，则字符串的前三个字符表示表创建时定义好字符串长度。

（3）RecordHead 定义一条记录的头结构，唯一标志一条记录，记录的头记录了该记录的第一个字段的地址，记录各个字段以链表的形式形成整条记录，若是字符串类型字段，字符串前三个字符是表示该字段字符串的长度。

（4）Record 定义所有的记录数据操作。记录读写包括索引部分(.idx)和数据部分(.dbf)。索引部分的修改由B+树对应的模块负责，本模块的所有操作只修改记录的实际数据部分。该类所有的记录均以文件地址唯一标志。删除和查找记录的地址由索引树提供

（5）CompareCell 储存比较的字段名称，比较的字段类型，比较关系，比较的值。

（1） CompareCell类：

该类为字段的比较类，储存的内容为Operator_Type与Column_Cell，Operator_Type是之前定义的枚举类，分别代表bigger，bigger and equal,less,less and equal,equal,not equal七种所有比较关系，

```
class CompareCell                    //一个字段比较单元
{
public:
    CompareCell(Operator_Type t, Column_Cell cc) :OperType(t), cmp_value(cc) {}
    bool operator()(const Column_Cell &cc);
    Operator_Type        OperType;          //比较关系关系运算符
    Column_Cell     cmp_value;
};

```

(2) KeyAttr类

我们使用联合体来降低占用内存，提高代码复用。除此之外进行了运算符重载，使得该类可以进行比较。

```
class KeyAttr
{
public:
    using Key_Value = union {
        char                StrValue[ColumnNameLength];      //字符串指针
        int                 IntValue;                        //整形值
        double              DoubleValue;                     //浮点型值
    };
    Column_Type type;
    Key_Value value;

    bool operator<(const KeyAttr &rhs)const;
    bool operator>(const KeyAttr &rhs)const;
    bool operator==(const KeyAttr &rhs)const;
    bool operator<=(const KeyAttr &rhs)const;
    bool operator>=(const KeyAttr &rhs)const;
    bool operator!=(const KeyAttr &rhs)const;

};

```

(3) Column_Cell类

```
class Column_Cell
{
public:
    Column_Cell();
    Column_Cell(KeyAttr key);
    Column_Cell(const Column_Cell& rhs); // 拷贝构造
    Column_Cell& operator=(const Column_Cell&rhs); // 拷贝赋值
    Column_Cell(Column_Cell&& rhs) = delete; // 移动构造
    Column_Cell& operator=(Column_Cell&&rhs) = delete; // 移动赋值
    size_t size()const;
    void* data()const;
    ~Column_Cell();
    Column_Type column_type;
    std::string columu_name;
    Column_Value column_value;
    Column_Cell *next;
    size_t sz = 0;   // 保存字符串字段的长度
    // 类型转换
    operator KeyAttr()const;
};

```

（4）RecordHead类

该类定义一条记录的头结构，唯一标志一条记录记录的头记录了该记录的第一个字段的地址，记录各个字段以链表的形式形成整条记录。

```
class RecordHead
{
public:
    RecordHead();
    RecordHead(const RecordHead &rhs); // 拷贝构造
    RecordHead& operator=(const RecordHead&rhs);  // 拷贝赋值
    RecordHead(RecordHead &&rhs); // 移动构造
    RecordHead& operator=(RecordHead&&rhs);  // 移动赋值
    ~RecordHead();

    void AddColumnCell(const Column_Cell &cc);
    size_t size()const;                          // 返回整条记录的大小
    Column_Cell* GetFirstColumn()const;

private:
    Column_Cell *phead;  // 指向记录的第一个字段
    Column_Cell *pLast;

};

```

（5）Record类

```
class Record
{
public:
    // 插入新的记录，返回新插入记录的所在数据文件的地址
    FileAddr InsertRecord(const std::string dbf_name, const RecordHead &rd);

    // 删除记录，返回删除的记录所在数据文件的地址
    FileAddr DeleteRecord(const std::string dbf_name, FileAddr fd, size_t);

    // 更新整条记录,若是字符串类型字段，字符串前三个字符是表示该字段字符串的长度
    // 实际写入时并不将字符串的长度写进文件
    bool UpdateRecord(const std::string dbf_name, const RecordHead &rd, FileAddr
fd);

private:
    // 将以链表的形式传入的记录数据读取并保存为一个整数据块以便于写入数据文件
    // tuple的第一个元素为记录数据的大小，第二个元素为数据的指针
    std::tuple<unsigned long, char*> GetRecordData(const RecordHead &rd);
};

```

5.6 IndexManager（BPlusTree）板块

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

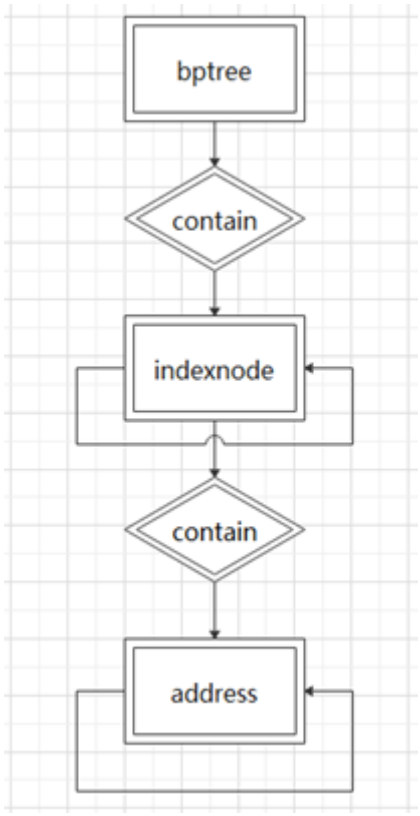
数据结构：

（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，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];
    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();
};
```

5.7 BufferManager板块

基本功能描述：

（1）实现物理文件的读写，包括两类接口：

（2）一类实现对任意文件任意位置读写任意长度的数据

（3）另一类实现对固定长度数据的读写操作，并自动管理文件空间

程序文件：

Buffer.h Buffer.cpp

数据结构

Buffer模块主要由以下几个类实现：

（1）PAGEHEAD，作为页头信息，用以标识文件页。

（2） FileAddr，作为文件地址，定位在文件中的位置。存储页的编号和页内偏移量。

（3） FILECOND，作为文件头，用以存储文件中各页存储状况。

（4）MemPage，作为内存页，即一个MemPage为一个固定内存页。存储文件号和文件页号，即指向的文件和在文件中为第几页，并且储存是否是最近一次访问内存和是否为脏页信息，以完成页替换功能，同时该类存储实际物理文件地址。以及是否钉住的信息，实现钉住功能。

（5）PageManager，作为内存页管理类，即该类存储所有页，当文件申请新页时，即从该类中提取页分配给文件。在该类中实现页置换算法。

（6） MemFile，作为文件类，即将已经打开的物理文件提取到内存中，放入缓冲区。在该类中实现读写删除更新数据等功能。并且与PageManger一起实现申请页并将数据写入该页。

（7） BUFFER，作为文件类的汇总，负责打开文件并存储已经打开文件，即所有其他模块均从该类进入，获得想要操作的文件。实现缓冲区的功能。

具体接口：

（1）MemPage类

```
class MemPage
{
  friend class MemFile;
  friend class PageManager;
  friend class BUFFER;
public:
  MemPage();
  ~MemPage();
private:
  void BacktoFile() const;       // 把内存中的页写回到文件中
  bool SetModified();          // 设置为脏页
  bool SetPinned();
public:
  unsigned long fileId;         // 文件指针，while fileId==0 时为被抛弃的页
  unsigned long filePageID;      // 文件页号
  mutable bool bIsLastUsed;      // 最近一次访问内存是否被使用，用于PageManager算法
  mutable bool isModified;       // 是否脏页
  mutable bool isPinned;        // 是否钉住
  void *PtrtoPageBegin;         // 实际保存物理文件数据的地址
  PAGEHEAD *pageHead;          // 页头指针
  FILECOND* GetFileCond()const;   // 文件头指针（while filePageID == 0）
};
```

以上是该类的定义。其中解释一下mutable去定义两个变量的原因。是因为在BacktoFile时加了const关键字，但是需要改变上述两个变量，所以需要有可变定义。

（2）MemFile类

该类实现较为复杂，首先从其存储数据看起。其存储filename，fileid，和文件一共有多少页等信息。这些应该都很好理解。函数ReadRecord即获得一个文件内容的地址，这里不是指单独的文件，而是文件中的一些内容的地址，代码也非常简单。ReadWriteRecord这个函数不同在于它读取的是将要用于改变的数据的地址，所以会多一步，即将该页设置为脏页，其余完全与上一函数相同。MemRead也与上述函数相同，其主要作用是在插入删除时读取作用，故需要将该页设置为lastused。

```
class MemFile
{
    friend class BUFFER;
    friend class BTree;
    friend bool DropTable(std::string table_name, std::string path);
public:
    const void* ReadRecord(FileAddr *address_delete)const;          // 读取某条记录，
返回记录指针(包括记录地址数据)
```

```
    void* ReadWriteRecord(FileAddr *address_delete);          // 读取某条记录,返回记录指针(包括记录地址数据)
    FileAddr AddRecord(const void* const source_record, size_t sz_record);
                // 返回记录所添加的位置
    FileAddr DeleteRecord(FileAddr *address_delete, size_t);                // 返回删除的位置
    bool UpdateRecord(FileAddr *address_delete, void *record_data, size_t record_sz);
    // 构造
    MemFile(const char *file_name, unsigned long file_id);
    // 写入数据
    void* MemRead(FileAddr *mem_to_read);                                   // 读取内存文件,返回读取位置指针
    FileAddr MemWrite(const void* source, size_t length);                   // 在可写入地址写入数据
    FileAddr MemWrite(const void* source, size_t length, FileAddr* dest);
    void MemWipe(void*source, size_t sz_wipe, FileAddr *fd_to_wipe);
    MemPage * AddExtraPage();                                               // 当前文件添加一页空间
    MemPage* GetFileFirstPage();                                            // 得到文件首页
    char fileName[MAX_FILENAME_LEN];
    unsigned long fileId;                                                  // 文件指针
    unsigned long total_page;                                              // 目前文件中共有页数
};

```

(3) PageManager类

该类是储存并管理整个内存页的类，其关键部分在于获取一个页与页替换的算法。

```
class PageManager
{
  friend class MemFile;
  friend class BUFFER;
  friend class BTree;
  friend bool InsertRecord(TB_Insert_Info tb_insert_info, std::string path);
  friend bool DropTable(std::string table_name, std::string path);
public:
  PageManager();
  ~PageManager();
  // 返回磁盘文件内存地址
  MemPage* GetMemAddr(unsigned long fileId, unsigned long filePageID);
  // 创建新页，适用于创建新文件或者添加新页的情况下
  MemPage* CreatNewPage(unsigned long fileId, unsigned long filePageID);
private:
  // 返回一个可替换的内存页索引
  // 原页面内容该写回先写回
  unsigned int GetReplaceablePage();
  // 如果目标文件页存在内存缓存则返回其地址，否则返回 nullptr
  MemPage* GetExistedPage(unsigned long fileId, unsigned long filePageID);
  MemPage* LoadFromFile(unsigned long fileId, unsigned long filePageID);
  // PageManager置换算法
  unsigned long PageManagerSwap();
  MemPage* MemPages[MEM_PAGEAMOUNT+1]; // 内存页对象数组
```

5.8 ErrorReporter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;
    };
}
```

5.9 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

(3)进行程序欢迎界面、帮助界面、基本指令跳转的实现

程序文件:

globaltimer.h

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：

进行计时的接口如下所示：

```
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
};

```

程序的欢迎界面和帮助界面的接口如下：

```
void ShowStarter()
{

    cout << "(+---------------------------------------------------------------------+)" << endl;
    cout << "(|                  Welcome to our Minisql!                            |)" << endl;
    cout << "(|                                                                     |)" << endl;
    cout << "(|*You can type a standard SQL statement that ends with; one at a time.|)" << endl;
    cout << "(|*You can type \'quit;\' to quit the minisql console.                  |)" << endl;
    cout << "(|*You can type \'help;\' to get help                                  |)" << endl;
```

```
    cout << "(|*Have fun and enjoy yourself!
|)" << endl;
    cout << "(|*Currently supported data types are int,double and char.
  |)" << endl;
    cout << "(|
  |)" << endl;
    cout << "(|Provided by ZJU DBS group:Fu Ziyang,Xiao Ruixuan and Sun Jiayang
  |)" << endl;
    cout << "(+---------------------------------------------------------------
--+)" << endl;

}
```

minisql的逻辑跳转接口如下：

```
void RunMiniSQL()
{
    SensefulStr senstr;
    PrintWindow print_window;
    while (true)
    {
        try
        {
            string cmd = GetCommand();
            senstr.SetSrcStr(cmd);
            auto cmd_type = GetOpType(senstr.GetSensefulStr());

            if (cmd_type == CmdType::QUIT)break;
            if (cmd_type == CmdType::HELP)
            {
                ShowHelp();
                continue;
            }
            if (cmd_type == CmdType::EXECFILE)
            {
                std::vector<std::string> tmp = senstr.GetSensefulStr();
                std::freopen(tmp[1].c_str(), "r", stdin);
                continue;
            }
            Interpreter(senstr.GetSensefulStr(), cmd_type, print_window);
        }
        catch (SQLError::BaseError &e)
        {
            SQLError::DispatchError(e);
            cout << endl;
            continue;
        }

    }
}
```

六、系统运行效果展示

Minisql初始化欢迎页面

```
(+---------------------------------------------------------------------+)
(|                  Welcome to our Minisql!                            |)
(|                                                                     |)
(|*You can type a standard SQL statement that ends with; one at a time.|)
(|*You can type 'quit;' to quit the minisql console.                  |)
(|*You can type 'help;' to get help                                   |)
(|*Have fun and enjoy yourself!                                       |)
(|*Currently supported data types are int,double and char.            |)
(|                                                                     |)
(|Provided by ZJU DBS group:Fu Ziyang,Xiao Ruixuan and Sun Jiayang     |)
(+---------------------------------------------------------------------+)
MiniSQL:
```

输入help之后的帮助页面

```
MiniSQL>help;
+-----------------------------------------------------------------------------------------------+
|A simple example to create a student databae named STU                                         |
+-----------------------------------------------------------------------------------------------+
|Create database  : create database DBS;                                                        |
|Use database     : use database DBS;                                                           |
|Show database    : show databases;                                                             |
|Create Table     : create table student(id int primary, socre double, name char(20));          |
|Insert Record(1) : insert into student(id,score,name)values(1,95.5,ZhangSan);                  |
|Insert Record(2) : insert into student(id,name)values(2,LiSi); Note:LiSi has no score          |
|UPDATE Table     : update student set score = 96.5 where name = LiSi;                          |
|Delete Table     : delete from student where id = 1; Note: ZhangSan is deleted                 |
|Select Table(1)  : select * from student where id = 2;                                         |
|Select Table(2)  : select * from student where id > 1 and score < 98;                          |
|Select Table(3)  : select id,score from student where id > 1 and score < 98;                   |
|Drop database    : drop database DBS;                                                          |
|Quit             : quit;                                                                       |
+-----------------------------------------------------------------------------------------------+
|Note             : Anytime you want to end MiniSQL use "quit;" command please.                 |
+-----------------------------------------------------------------------------------------------+
```

6.1 Create创建语句

6.1.1 database的创建

```
MiniSQL>create database test1;
Create succeed!
```

```
MiniSQL>use database test1;
database changed!
```

6.1.2 table的创建

成功建表

```
MiniSQL> create table student2(id int,name char(12) unique,score float,primary key(id));
Create table student2 successfully!
(0.06 sec)
MiniSQL>
```

如果我们次数再次要求建立同名的表，则会因为存在重名的表而建表失败：

```
MiniSQL> create table student2(id int,name char(12) unique,score float,primary key(id));
Table student2 already exists!
(0.00 sec)
MiniSQL>
```

当我们使用错误的数据类型建表的时候，则会因为数据类型不能识别而建表失败，如下图所示

```
(0.00 sec)
MiniSQL> create table student2(id int,string a,primary key(id));
unknown data type
MiniSQL>
```

6.1.3 Index的创建

我们可以选择在unique的属性上创建新的索引

```
MiniSQL> create index nameindex on student2(name);
Create index nameindex on table student2 successfully!
(0.03 sec)
```

6.2 insert插入语句

6.2.1 成功插入单条记录:

```
MiniSQL> insert into student2 values(1080109019,'name9019',73.5);
insert result:####################
insert 1 record(s) on student2 successfully!
##################################
(0.01 sec)
```

6.2.2 成功插入多条记录

sql语句如下:

```sql
insert into student2 values(1080109097,'name9097',82.5),
(1080109098,'name9098',89),
(1080109099,'name9099',96.5),
(1080109100,'name9100',61.5),
(1080109101,'name9101',61.5),
(1080109102,'name9102',76),
(1080109103,'name9103',57.5);
```

```
insert 7 record(s) on student2 successfully!
##################################
(0.05 sec)
```

6.2.3 插入的类型不匹配时

插入语句

```sql
insert into student2 values(10801090,'name9097','a');
```

插入结果:

```
insert syntax error!
```

6.2.4 插入重复的主键或unique值时

出现重复的主键时，插入失败

```
MiniSQL> insert into student2 values(1080109097,'name9097',82.5);
[debug]Error: In IndexManager::insertIndex(); Index insertion failed because of duplicate key!
```

当unique的属性出现重复时，同样会出现插入失败的提示

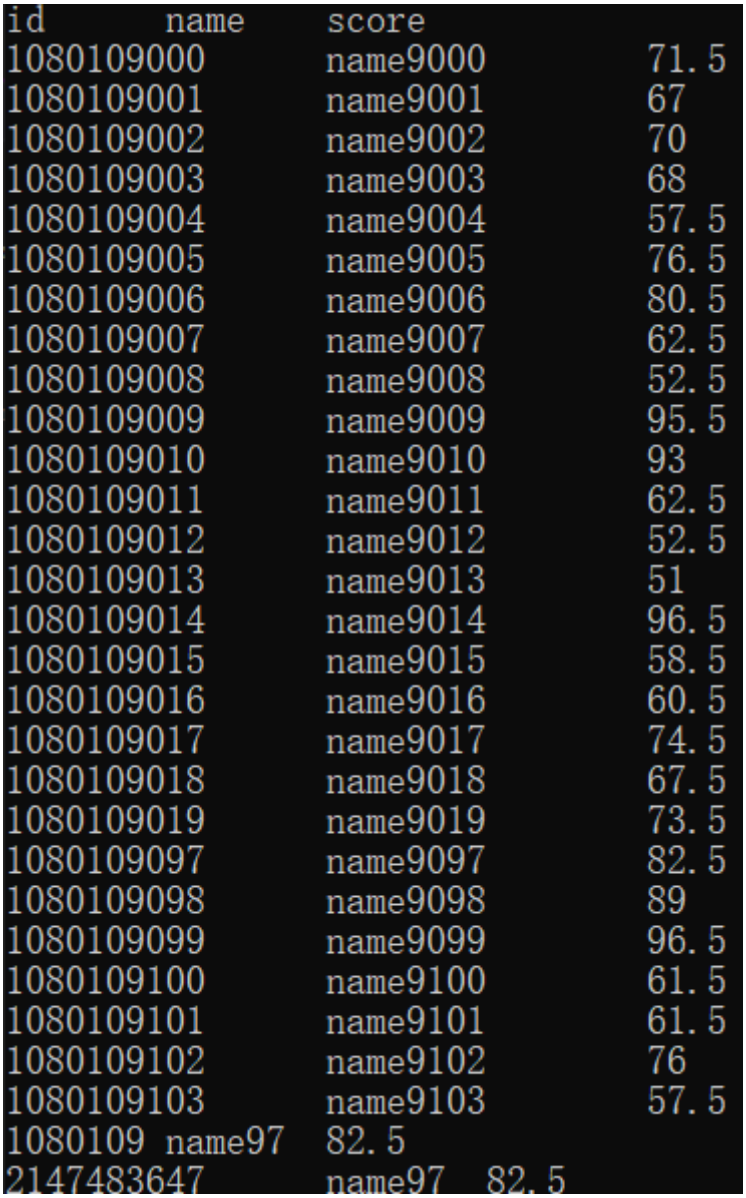
```
MiniSQL> insert into student2 values(108098,'name9098',89);
[debug]Error: In IndexManager::insertIndex(); Index insertion failed because of duplicate key!
```

6.3 select查询语句

6.3.1无条件全表查询

```
Insert succeed!
MiniSQL>select * from student2;
+-----------------------------------+
|id         |name         |score    |
+-----------------------------------+
|1080       |naddsdse1    |229      |
|108001     |nadsdse1     |19       |
|10800001   |nadsme1      |99       |
|1080100001 |name1        |99       |
+-----------------------------------+
4 row in set.[0.00011840 seconds used.]
```

6.3.2 单条件查询



id	name	score
1080109000	name9000	71.5
1080109001	name9001	67
1080109002	name9002	70
1080109003	name9003	68
1080109004	name9004	57.5
1080109005	name9005	76.5
1080109006	name9006	80.5
1080109007	name9007	62.5
1080109008	name9008	52.5
1080109009	name9009	95.5
1080109010	name9010	93
1080109011	name9011	62.5
1080109012	name9012	52.5
1080109013	name9013	51
1080109014	name9014	96.5
1080109015	name9015	58.5
1080109016	name9016	60.5
1080109017	name9017	74.5
1080109018	name9018	67.5
1080109019	name9019	73.5
1080109097	name9097	82.5
1080109098	name9098	89
1080109099	name9099	96.5
1080109100	name9100	61.5
1080109101	name9101	61.5
1080109102	name9102	76
1080109103	name9103	57.5
1080109	name97	82.5
2147483647	name97	82.5

查询语句

```
1  select * from student2 where score>80
```

查询结果

```
MiniSQL> select * from student2 where score>80;
select result: ####################
id       name     score
1080109006       name9006         80.5
1080109009       name9009         95.5
1080109010       name9010         93
1080109014       name9014         96.5
1080109097       name9097         82.5
1080109098       name9098         89
1080109099       name9099         96.5
1080109 name97   82.5
2147483647       name97   82.5
2147483647       name97   82.5
1080102 name97   82.5
1111     name97   82.5
1080109097       name9097         82.5
108098   name9098         89
select 14 record(s) on student2 successfully!
```

6.3.3复合条件查询

查询语句

```
select * from student2 where score>80 and name='name9006';
```

查询结果

```
(0.02 sec)
MiniSQL> select * from student2 where score>80 and name='name9006'
select result: ####################
id       name     score
1080109006       name9006         80.5
select 1 record(s) on student2 successfully!
```

6.3.4 没有符合要求的查询

查询语句

```
select * from student2 where score>90 and name='name9006'
```

查询结果

```
MiniSQL> select * from student2 where score>90 and name='name9006';
select result: ####################
id       name     score
No record is selected
```

6.4 delete删除语句

6.4.1没有符合条件的删除

删除语句

```
delete * from student2 where score>90 and name='name9006'
```

删除结果

```
MiniSQL> delete from student2 where score>90 and name='name9006';
delete result:####################
delete 0 record(s) on student2 successfully!
##################################
```

6.4.2 单一条件删除

删除语句

```sql
delete from student2 where score>90;
```

删除结果

```
delete result:####################
delete 4 record(s) on student2 successfully!
```

6.4.3 复合条件删除

删除语句

```sql
delete from student2 where score>80 and name='name9006';
```

删除结果

```
MiniSQL> delete from student2 where score>80 and name='name9006';
delete result:####################
delete 1 record(s) on student2 successfully!
```

6.4.4 整表删除

删除语句

```sql
delete * from student2;
```

删除结果

```
delete result:####################
delete 29 record(s) on student2 successfully!
```

6.5 drop语句

6.5.1 成功删除索引

drop语句

```sql
drop index nameindex on student2(name);
```

执行结果

```
MiniSQL> drop index nameindex on student2(name);
Drop index nameindex on table student2 successfully!
(0.05 sec)
```

6.5.2 成功删除表格

drop语句

```
drop table student2;
```

执行结果

```
MiniSQL> drop table student2;
Drop table student2 successfully!
(0.05 sec)
```

6.5.3 删除不存在的（失败删除）

drop语句

```
drop table student3;
```

执行结果

```
MiniSQL> drop table student3;
Table student3 not exists!
(0.01 sec)
```

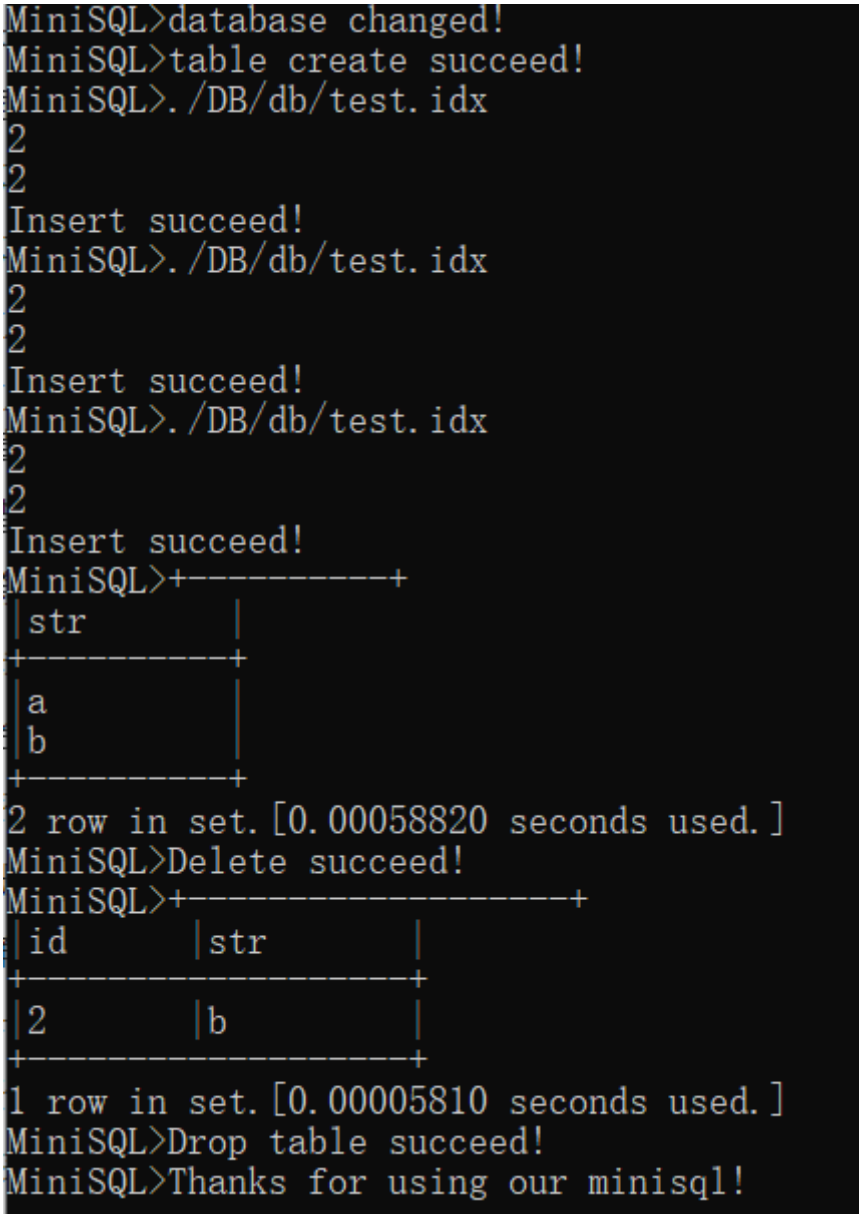
可以看出，因为想要删除的表不存在，因此删除失败。

6.6 execfile执行sql脚本语句

test.txt代码如下：

```
create database db;
use database db;
create table test(id int, str char(10) unique, primary key(id));
insert into test(id, str) values (1,'a');
insert into test(id, str) values (2,'b');
insert into test(id, str) values (3,'c');
select str from test where id>=1 and id<3;
delete from test where str<>'b';
select * from test;
drop table test;
quit;
```

执行execfile后运行结果如下：



如图所示，可以成功运行execfile指令。