MiniSQL 实验报告

专业：混合班

姓名：黄彦玮

学号：3180102067

课程：数据库系统

目录

一、MiniSQL总体框架

1.1 MiniSQL实现功能分析

本次实验的目的是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。MiniSQL支持以下操作：

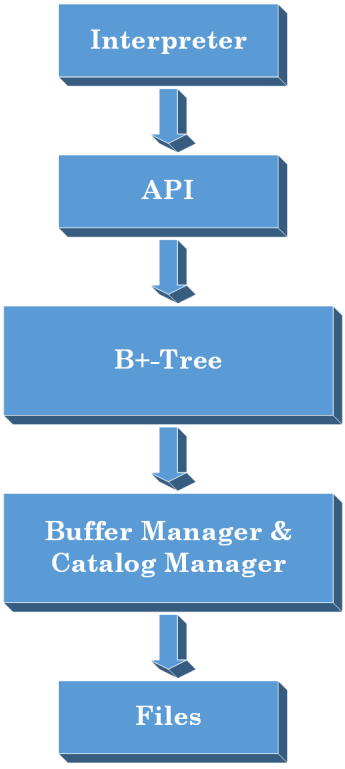
1. 数据类型：支持三种基本数据类型：integer，char(n)，float。
2. 表定义：一个表可以定义多达32个属性，各属性可以指定是否为unique；支持单属性的主键定义。
3. 索引定义：对于表的主属性自动建立B+树索引，对于声明为unique 的属性可以通过SQL 语句由用户指定建立/删除B+树索引，所有的B+树索引都是单属性单值的。
4. 数据操作: 可以通过指定用and连接的多个条件进行查询，支持等值查询和区间查询。支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。

此外，在上述基本功能上，我们还加入了以下功能：

1. 支持select * from table操作。
2. 当SQL语句语法错误时，会进行准确的报错，错误信息准确反映错误原因。
3. 支持单条语句执行和文件执行两种模式，并可搭配使用。

1.2 MiniSQL系统体系结构

MiniSQL的体系结构如下图所示。



1.3 设计语言与运行环境

设计语言：C++ 11.0

开发环境：Microsoft Visual Studio 2019

二、MiniSQL各个模块实现功能

2.1 Interpreter实现功能

本模块是一个用于与用户交互的前端模块，用户可以在dos窗口内输入命令，Interpreter模块会对用户输入的命令进行语法解析。本模块的具体功能是：

1. 对用户输入的指令进行判断：①若为quit，退出程序。②如为"-f"开头，表示从文件读入指令，则检查对应的文件是否存在，若存在，则从对应文件内读入指令并进入步骤2，若不存在则终止命令的执行（但不终止程序执行），并在窗口中显示错误信息，包括错误类型与错误原因。③如用户输入的指令不为“-f"开头，进入步骤2.
2. 先判断用户输入指令的类型，再判断用户输入的命令是否符合SQL语言中该种指令的语法。若两者有任意一者出现错误，则终止命令显示错误信息。否则进入步骤3.
3. 根据指令的类型，将用户指令中的元素（如表名、索引名等）用已有数据结构封装，然后调用API的相关函数进行执行。

对于超过一行的指令，用户可以在一行语句的最后加上”^"号，并在下一行继续输入，Interpreter模块会自动过滤行末的"^"号，并将多行的输入当作一条指令处理。

2.2 API实现功能

本模块用于各种类型的指令的执行，根据不同语句类型，采取不同的执行策略，调用index manager & record manager模块内的相关API进行执行。

2.3 B+-Tree实现功能

B+-Tree模块整合了Index Manager和Record Manager。

Index Manager主要包含了B+树索引，实现B+树的新建、插入节点、删除节点、寻找节点、删除整棵树等功能。B+树的Order是参数，可以根据需要改变。

Record Manager实现记录的插入、删除与查找操作。

2.4 Buffer Manager & Catalog Manager实现功能

Buffer Manager是管理内存的核心模块。这一模块主要实现了加载/更新/删除内存块，如果需求的块不在内存中，则从硬盘中进行相应的读写，否则直接对内存进行读写，但不写回硬盘，待程序运行结束后再写回硬盘，以加快运行速度。当内存中储存的块的个数达到上限时，会对已有的块进行替换，这里采用了先进先出（First in first out, FIFO）的替换策略。

Catalog Manager储存了所有index和table的信息。对于index，它储存了索引名、对应的表名、对应的属性、对应B+-Tree的根节点及大小等信息。对于table，它储存了表名、主码、属性名、属性类型、是否唯一等信息。在进行指令的执行时，通过调用Catalog Manager的API，可以很方便地获取表格或索引的基本信息。

三、内部数据形式及各模块提供的接口

3.1 内部数据形式

为了储存各种类型的数据，我们根据需要设计了多种数据结构，具体如下：

1. user：用于储存数据库用户系统中的用户信息。定义如下：

```
class user {
    string account; //账号
    string pwd; //密码
    bool status; //登录状态
}
```

2. cond：用于储存“select"和”delete"语句中“where"后的表达式（每个cond对象储存单个表达式）。定义如下：

```
class cond { //用于表达式的判断，如 a = 5
    string attr; //被比较的属性名
    int op; //比较符号（0：等于 -1：小于等于 1：大于等于）
    string para; //被比较的参数
}
```

3. where_c：用于储存“select"和”delete"语句中“where"后的表达式（每个where_c对象储存一组表达式）。定义如下：

```
class where_c {  //用于where约束
    int conlist_size; //where约束数量（0表示没有）
    cond* con_list; //表达式列表
}
```

4. select_c：是where_c的继承类，用于储存"select"语句中的信息，包括被选择的属性以及"where"约束的表达式列表。定义如下：

```
class select_c :public where_c {
    int attr_size; //被选择的属性数量（0表示全部）
    string* attr_list;
}
```

5. rec：用于储存table中的单条记录，用于"insert"语句的插入数据或"select"语句的结果返回。定义如下：

```
class rec {  //单条记录
    int attr_size;  //属性个数
    string* attr_list;  //属性名
    string* val_list;   //属性值
}
```

6. rec：用于储存记录表（含一条或多条数据），用于"insert"语句的插入数据或"select"语句的结果返回。定义如下：

```
class data_c {  //记录表，用于insert操作的传参和select操作的返回
    int rec_size; //记录个数
    vector<rec> rec_list; //动态数组储存记录
}
```

7. attr_c：用于储存表格的属性，每个attr_c对象对应一条属性。每个属性包括属性名、种类、长度（仅当属性类型为char时才有意义）、是否唯一。定义如下：

```
class attr_c {
    string attr_name; //属性名
    int type;//0 int 1 float 2 char
    int length;//0 for int&float, length for char
    bool isunique;//1 if unique
}
```

8. table_c：用于储存表格的各项信息。定义如下：

```
class table_c {
    string table_name;//表名
    int attr_size;//属性个数
    attr_c* attr_list;//属性名
    int primkey;//主码的index，只有一个
    int element_size;//记录大小
    int rec_per_page;//每页记录数
}
```

9. catalog_c：用于catalog manager中储存索引的各项信息。定义如下：

```
class catlog_c {
    string index_name;//索引名
    string table_name;//表名
    int tree_index;//属性编号
    int index_size;//索引大小
    int index_root_no;//索引根节点编号
}
```

10. BNode：用于储存B+-Tree的节点信息。定义如下：

```
struct BNode {
    string tablename;//表名
    int tree_index;//索引对应属性编号
    int BT_No;//节点编号
    valtype val[ORDER];//结点的键值，按递增的顺序排
    int child[ORDER];//指向孩子的指针
    int num;//键值的数量
    bool isleaf;//是否为叶结点
    bool isroot;//是否为根结点
    Vector<rec>* recs[ORDER];//记录
};
typedef struct BNode* BTree;
```

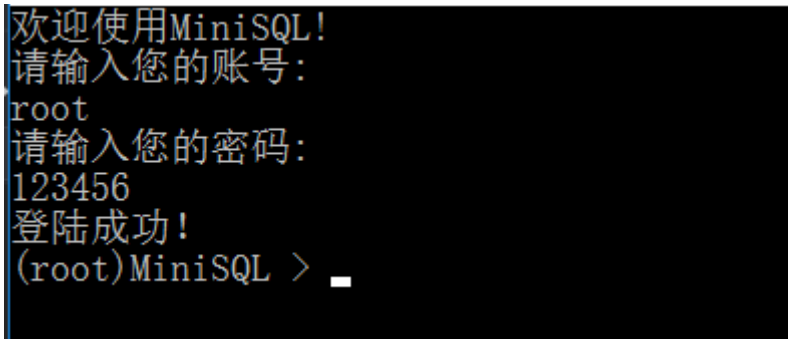
11. file_exception：用于按文件读入指令。（这一功能在实现时用异常处理的方法实现，当interpreter检测到"-f"时，通过报告此类异常来直接返回上一层进行文件读写，具体细节见第4章）定义如下：

```
class file_exception {
    string in_file;//输入文件名，必选
    string out_file;//输出文件名，可选
}
```

12. 在此基础上，我们还实现了Vector，其功能与c++ stl中的vector基本类似，但增加了按下标删除的功能。

3.2 主窗口及主函数设计

MiniSQL采用字符界面窗口，用户需要先行登陆，登陆后可以在"(用户名)MiniSQL > "的提示后输入指令，如下图所示。



3.3 Interpreter接口

```
void exec(string o);
void select_exec(string o, int i);
void insert_exec(string o, int i);
void delete_exec(string o, int i);
void create_exec(string o, int i);
void drop_exec(string o, int i);
void create_table_exec(string o, int i);
void create_index_exec(string o, int i);
void drop_table_exec(string o, int i);
void drop_index_exec(string o, int i);
void getwhere(string o, int i, where_c* data);
void ssplit(const string& s, vector<string>& sv);
```

接口说明：

void exec(string o)：判断指令o的指令类型，并调用对应的api。其中，select指令调用”select_exec"，insert指令调用“insert_exec"，delete指令调用”delete_exec"。对于create table / create index指令和drop table / drop index指令，先分别调用"create_exec"和”delete_exec"判断新建/删除的是表还是索引，然后分别调用"create_table_exec"（或"drop_table_exec"）和“create_index_exec"（或"drop_index_exec"）进行执行。

void select_exec(string o, int i)：将select指令o从下标i开始的部分中的元素提取出来，如表名、属性名、表达式等，用现有数据结构封装后传给API模块进行执行。

void create_exec(string o, int i)：先判断新建的是表还是索引，然后分别调用"create_table_exec"和“create_index_exec"进行执行。

void drop_exec(string o, int i)：先判断删除的是表还是索引，然后分别调用"drop_table_exec"和“drop_index_exec"进行执行。

void insert_exec(string o, int i) ，void delete_exec(string o, int i)，

void create_table_exec(string o, int i)，void create_index_exec(string o, int i)，

void drop_table_exec(string o, int i)，void drop_index_exec(string o, int i)： 类同void select_exec(string o,int i)。

void getwhere(string o, int i, where_c* data)：从指令o的下标i开始读表达式（即"select"和”delete"语句中的“where"关键字后的表达式），并将结果储存进data指向的内容中。

void ssplit(const string& s, vector& sv)：将字符串s按空格划分成若干个子字符串，并储存到动态数组sv中。

3.4 API接口

```
bool con_satisfy(rec r, int con_size, cond* con_list, table_c *tb);
data_c* select_from_table(const string table_name, const select_c* data);
bool insert_into_table(const string table_name, data_c* data);
bool delete_from_table(const string table_name, const where_c* data);
bool create_index(const string table_name, const string attr_name, const string
index_name);
bool drop_index(const string table_name, const string index_name);
bool create_table(table_c data);
bool drop_table(string table_name);
```

bool con_satisfy(rec r, int con_size, cond* con_list, table_c *tb)：判断给定的记录r是否能满足"where"后的一组约束条件。其中con_size是约束表达式个数，con_list对应表达式数组，tb指向的是表格的基本属性。

data_c* select_from_table(const string table_name, const select_c* data)：根据表名和data中的信息（包括被选择的属性名、"where"后的表达式约束等）从对应的表中选择出符合条件的结果。

bool insert_into_table(const string table_name, data_c* data)：根据表名和data中的信息（包括被插入的记录）向对应的表插入记录。

bool delete_from_table(const string table_name, const where_c* data)：根据表名和data中的信息（包括"where"后的表达式约束等）从对应的表中删除记录。

bool create_index(const string table_name, const string attr_name, const string index_name)：根据表名、属性名、索引名创建索引。

bool drop_index(const string table_name, const string index_name)：根据表名、索引名删除索引。

bool create_table(table_c data)：根据data中的表信息创建表。

bool drop_table(string table_name)：根据表名删除表。

3.5 B+-Tree接口

```
BTree CreateBTree(string tn, int ti);
bool Insert(valtype key, BTree* T, bool dupl, rec& data);
bool Delete(valtype key, BTree* T);
bool DeleteATree(BTree& T);
BTree key_where(BTree *T, valtype &key, int &index);
```

BTree CreateBTree(string tn, int ti)：在表格tn上的属性ti上创建B+-Tree索引。

bool Insert(valtype key, BTree* T, bool dupl, rec& data)：向T这棵B+-Tree插入data对应的记录，其中key为索引的键值，dupl为是否允许重复。

bool Delete(valtype key, BTree* T)：向T这棵B+-Tree插入key键值对应的记录。

bool DeleteATree(BTree& T)：删除T这棵B+Tree的所有内容。

BTree key_where(BTree *T, valtype &key, int &index)：在T这棵B+-Tree中寻找键值key所在的位置并返回，其中index为索引的属性。

3.6 Buffer Manager & Catalog Manager 接口

```
void updatetable(table_c data);
void inserttable(table_c data);
bool load_one_table(string table_name, table_c* & data);
```

```
bool deletetable(string table_name);
void insertindex(string index_name, string tablename, int tree_index);
bool load_one_index(string index_name, string& tablename, int& tree_index);
bool deleteindex(string index_name);
bool deleteindex_tb(string table_name);
bool createBTfile(BTree T);
bool deleteBTfile(string tablename, int tree_index, int BT_No);
BTree getBTfile(string tablename, int tree_index, int BT_No);
BTree getRootfile(string tablename, int tree_index);
int getnewNo(string tablename, int tree_index);
bool upd(string table_name, int tree_index, int new_root);
void updateindex();
void quit_upd();
```

注：因为储存不同类型数据的块被替换的频率不同，所以在管理内存中的块的时候，我们对存放不同类型的数据的块进行分开管理，每中类型的块的数量的上限是可以修改的参数，这样做的好处是减少了块的替换次数进而减少了硬盘读写（如Catalog Manager中的内存块的替换频率通常较低，避免了反复替换），可以提高程序运行的效率。

void updatetable(table_c data)：将内存中对管理文件的更新写入硬盘

void inserttable(table_c data)：将data各属性写到管理文件的末尾

bool load_one_table(string table_name, table_c* & data)：从硬盘中读取一张表到data，table_name是表名

bool deletetable(string table_name)：删除table_name对应的表

void insertindex(string index_name, string tablename, int tree_index)：根据索引名、表名、属性名向Catalog Manager插入索引

bool load_one_index(string index_name, string& tablename, int& tree_index)：根据索引名将信息从Catalog Manager中查出后写到后面两个信息里,没有则返回0

bool deleteindex(string index_name)：根据索引名从Catalog Manager中删除索引

bool deleteindex_tb(string table_name)：根据表名从Catalog Manager中删除该表下的所有索引

bool createBTfile(BTree T)：在内存中创建B+-Tree节点

bool deleteBTfile(string tablename, int tree_index, int BT_No)：根据表名、属性名、节点编号从内存和硬盘中删除B+-Tree节点

BTree getBTfile(string tablename, int tree_index, int BT_No)：根据表名、属性名、节点编号从硬盘中获取B+-Tree节点的信息

BTree getRootfile(string tablename, int tree_index)：根据表名、属性名，获取建立在该属性上的索引的根节点的编号，并返回该节点对应指针

int getnewNo(string tablename, int tree_index)：根据表名、属性名，从建立在该属性上的索引申请新的节点，返回该新节点的编号

bool upd(string table_name, int tree_index, int new_root)：根据表名、属性名，将建立在该属性上的索引的根节点改为new_root。

void updateindex()：将所有在内存中修改过但未同步到硬盘的Catalog Manager中的索引信息（即脏信息）写入到硬盘。

void quit_upd()：用户输入quit指令时，更新Buffer Manager中的所有脏信息。

四、参与设计部分实现思路&代码实现

4.1 数据结构设计

3.1节中给出了本项目用到的全部数据结构，其中除BNode由负责B+-Tree的同学设计外，其他数据结构均由我设计。数据结构的设计考虑了各种类型指令中的元素，对每种类型指令设计了合理的数据结构。此外，利用面向对象的编程思想，在设计时注重类之间的继承关系，并对每一个类进行了较为完善的封装。

4.2 简易用户系统与前端界面设计

由于用户系统非本课程与本项目的重点，因此在设计上采取的简单易用的原则，直接将用户信息预置于内存中。前端界面的设计也本着简洁、友好、易用的原则，但充分考虑不同用户的习惯（即不同风格的输入），并在错误信息的设计上保持了细致、严谨的方式。返回结果较为清晰，易于观察。

这一部分的亮点在于指令输入上支持了单行输入/换行输入/从文件中读入等多种模式的切换。其中文件读入的格式为

```
-f [input_name] [output_name] //其中[output_name]可选
```

对于文件读入的方式，在interpreter检测到”-f"关键字后，采用抛出异常的方式将输入文件名和输出文件名返回到前端，在异常处理部分中实现从文件中读入指令的方式。

4.3 Interpreter

在这一模块中，我们需要对用户输入的指令进行语法分析和元素提取。基本的步骤在2.1节已有叙述。这一模块的实现的关键在于如何对输入的字符串进行高效、简洁的分析。这里我采用的方法是用一个split函数把字符串分割成若干子字符串，再进行分析。为适应不同用户的输入习惯（例如变量与符号间不加空格等），我们首先要对一些关键字特殊处理，即在“,"、”=“、”>="、“<=”等符号前后先全部加上空格，然后再以空格为关键字对字符串进行分割。分割后的结果用一个vector进行处理。这样，通过对应元素之间的比较，就可以简洁高效地分析指令的类型及成分，从而对元素进行封装。另一个关键点是需要处理char(n)类型的量中含上述关键字的情况，这种情况我们强制用户在关键字前加上转义字符“\"来进行区分，在代码实现时需要特别判断来加以处理。在实现时还需要特别注意大小写的判断。

split函数的实现如下：

```
void ssplit(const string& s, vector<string>& sv) {
    sv.clear();
    string temp;
    int flag1 = 0,  flag2 = 0;
    for (int i = 0; i < s.length(); i++)
    {
        if (s[i] == '\'')flag1 = !flag1;
        if (s[i] == ' ' && !flag1)
        {
            if (flag2)
            {
                sv.push_back(temp);
                temp = "";
                flag2 = 0;
            }
        }
        else
        {
            temp = temp + s[i];
```

```
            flag2 = 1;
        }
    }
    if (temp != "")sv.push_back(temp);
}
```

有了split函数后，提取元素得到了很大的便利。以select操作为例，分别经历了寻找from关键字、提取被选取的属性名、提取被选取的表名、获取"where"关键字后的约束四个阶段。当属性名或表名为空时，通过抛出异常显示错误信息。四个阶段结束后，调用API模块的接口执行指令，再将返回的结果输出到字符界面。代码如下：

```
void select_exec(string o,int i)
{
    int j, k;
    for (; i < o.length(); i++)if (o[i] != ' ')break;
    select_c* data = new select_c();
    //寻找from关键字
    for (j = i; j < o.length(); j++)
    {
        if (j + 3 < o.length() && o[j - 1] == ' ' && tolower(o[j]) == 'f' &&
tolower(o[j + 1]) == 'r' && tolower(o[j + 2]) == 'o' && tolower(o[j + 3]) == 'm'
&& (j + 4 >= o.length() || o[j + 4] == ' ' || o[j + 4] == '\n'))
        {
            break;
        }
    }
    if (j == o.length())
    {
        throw "Error: 'From' Not Found!";
    }
    //提取被选取的属性名
    while (i < j)
    {
        string attr_p = "";
        for (; i < j; i++)if (o[i] != ',' && o[i] != ' ')break;
        for (; i < j; i++)if (o[i] == ',' || o[i] == ' ')break; else attr_p =
attr_p + o[i];
        if (attr_p == "")throw "Error: Attribute Not Found!";
        data->ins_attr(attr_p);
        if (o[i] == ',')
        {
            i++;
            continue;
        }
        else
        {
            for (++i; i < j; i++)
            {
                if (o[i] != ' ')
                {
                    throw "Error: ',' omitted in attributes!";
                }
            }
        }
    }
    if (data->get_attr_size() == 0)
    {
```

```
        throw "Error: Attribute CANNOT be NULL!";
    }
    //获取表名
    i = j + 4;
    for (; i < o.length(); i++)if (o[i] != ' ')break;
    string table_name = "";
    for (; i < o.length(); i++)if (o[i] != ' '&&o[i]!='\n')table_name =
table_name + o[i]; else break;
    if (table_name == "")throw "Error: Table name CANNOT be NULL!";
    //获取where关键字后的约束表达式
    int flag = 1;
    for (int j = i; j < o.length(); j++)if (o[j] != ' ' && o[j] != '\n')flag =
0;
    if (!flag)
    {
        getwhere(o, i, data);
        if (data->get_conlist_size() == 0)
        {
            throw "Error: Expressions After 'WHERE' CANNOT be NULL!";
        }
    }
    //调用API模块中的接口执行指令
    data_c* ans = select_from_table(table_name, data);
    //将返回的结果输出到用户端
    if (ans->get_rec_size() == 0)
    {
        cout << "No Record Found." << endl;
        cout << "----------------------------" << endl;
        delete ans;
        return;
    }
    Vector<rec>* tmp = ans->get_rec_list();
    for (int j = 0; j < (*tmp)[0].get_attr_size(); j++)
    {
        cout << std::left << setw(15) << (*tmp)[0].get_attr_list(j) << " ";
    }
    cout << endl;
    for (int i = 0; i < ans->get_rec_size(); i++)
    {
        for (int j = 0; j < (*tmp)[i].get_attr_size(); j++)
        {
            cout << std::left << setw(15) << (*tmp)[i].get_val_list(j).c_str()
<< " ";
        }
        cout << endl;
    }
    cout << "----------------------------" << endl;
    delete ans;
}
```

4.4 Buffer Manager & Catalog Manager

Buffer Manager主要实现Block的管理。对于存放数据类型不同的block，采取不同的存放格式。存放BNode（即B+-Tree节点）的块中的结构为：（各元素的含义同3.1节BNode数据结构）

```
table_name tree_index BT_No num isleaf isroot
val[0] val[1] ... val[order-1]
child[0] ... child[order]
rec_size //记录数量
attr_size //属性数量
attr_name[0] attr_name[1] ... //属性名
rec[0].val[0] rec[0].val[1] ...
rec[1].val[0] rec[1].val[1] ...
...
```

存放Catalog Manager中每张表的基本信息的块的结构为：（各元素的含义同3.1节table_c数据结构）

```
table_name element_size rec_per_page
//element_size：每个元素大小 rec_per_page：每块的元素个数
attr_size primary_key
attr[0].name attr[0].type attr[0].len attr[0].isunique
attr[1].name attr[1].type attr[1].len attr[1].isunique
...
```

存放Catalog Manager中每个索引的基本信息的块，每行为一个索引的信息，结构为：（各元素的含义同3.1节Catalog_c数据结构）

```
index_name table_name tree_index index_size index_root
```

当Buffer Manager中的块满了时，采用先进先出（First in first out, FIFO）的策略对块进行替换，实现时用一个队列记录所有块加载的顺序。

Buffer Manager的内部用stl map实现了块的信息与块的编号间的映射，查找时根据需要查找的索引名/表名等信息检索出块的编号，再进行块的读写，方便了信息的查找。

以创建B+-Tree节点、插入表、删除表为例，代码如下：

```
bool createBTfile(BTree T) //对应table的对应B+树创造结点
{
    string file_name = (string)"bin/Ind_" + T->tablename.c_str() + (string)"_" +
to_string(T->tree_index) + (string)"_" + to_string(T->BT_No) + (string)".txt";
    //若Buffer manager中的块的个数已满，替换旧的块后再写入
    if (buf_mp[file_name] == NULL)
    {
        if ((int)buf_queue.size() == max_page)
        {
            writeBTfile(buf_mp[buf_queue.front()]);
            delete buf_mp[buf_queue.front()];
            buf_mp.erase(buf_queue.front());
            buf_queue.pop();
        }
        buf_mp[file_name] = T;
        buf_queue.push(file_name);
    }
    else
    {
        buf_mp[file_name] = T;
    }
    return 1;
}
```

```
void inserttable(table_c data)
{
    string table_name = data.get_table_name();
    //表名已存在，直接替换，否则新建一条信息
    for (int i = 0; i < buf_table.size(); i++)
    {
        if (buf_table[i].get_table_name() == table_name)
        {
            buf_table[i] = data;
            return;
        }
    }
    if (buf_table.size() == max_table)
    {
        updatetable(buf_table[0]);
        buf_table.deletei(0);
    }
    buf_table.push_back(data);
}
```

```
bool deletetable(string table_name)//删除名为tablename的table
{
    //从buffer manager中删除对应的内存块
    for (int i = 0; i < buf_table.size(); i++)
    {
        if (buf_table[i].get_table_name() == table_name)
        {
            buf_table.deletei(i);
        }
    }
    //从硬盘中删除对应的文件
    string file_name = (string)"bin/Cat_" + table_name.c_str() + (string)".txt";
    return remove(file_name.c_str());
}
```

五、MiniSQL系统测试

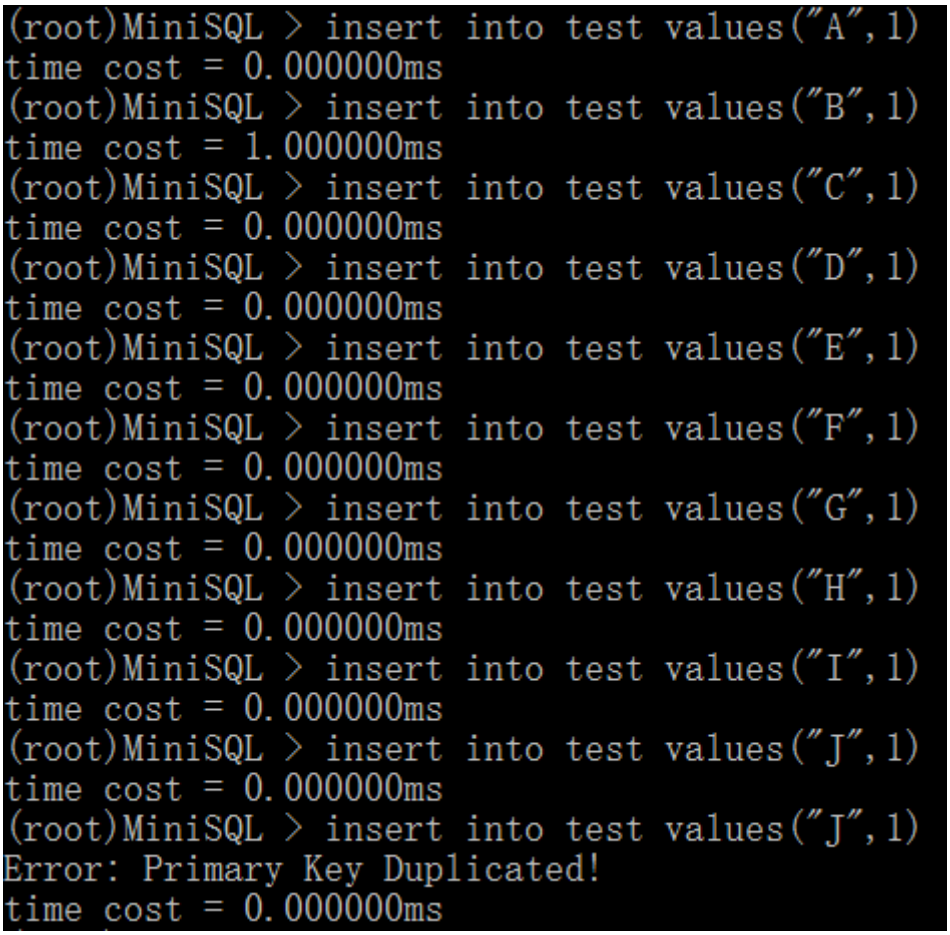
5.1 创建表

```
create table test(name char(5), id int, primary key(name))
```

```
(root)MiniSQL > create table test(name char(5), id int, primary key(name))
time cost = 1.000000ms
```

5.2 插入数据

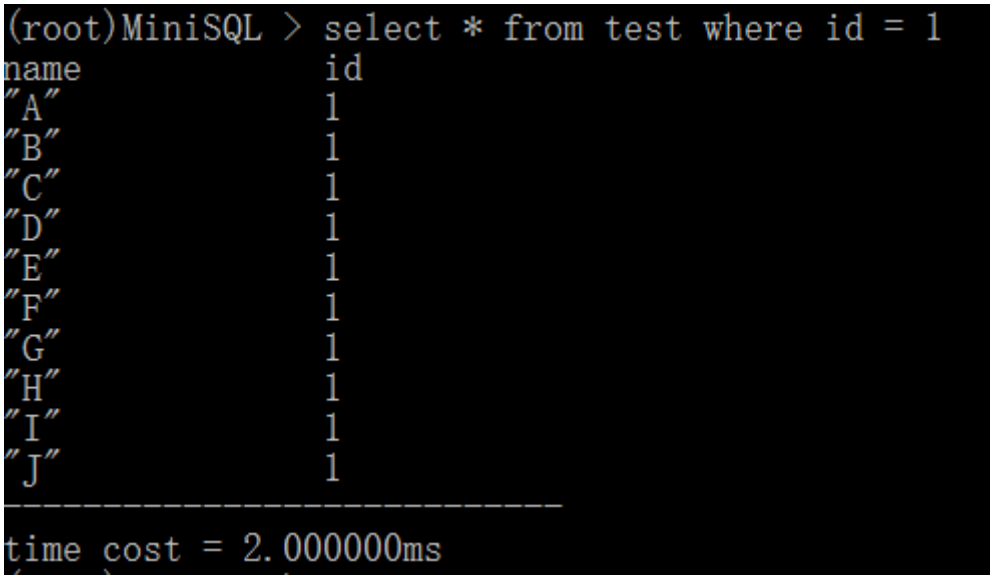
```
insert into test values("A",1)
insert into test values("B",1)
insert into test values("C",1)
insert into test values("D",1)
insert into test values("E",1)
insert into test values("F",1)
insert into test values("G",1)
insert into test values("H",1)
insert into test values("I",1)
insert into test values("J",1)
```



5.3 查找数据

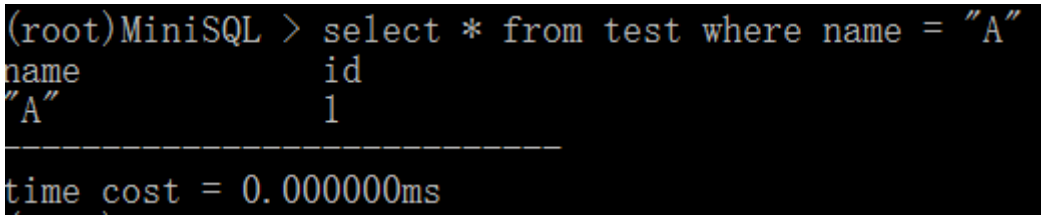
无索引：

```
select * from test where id = 1
```



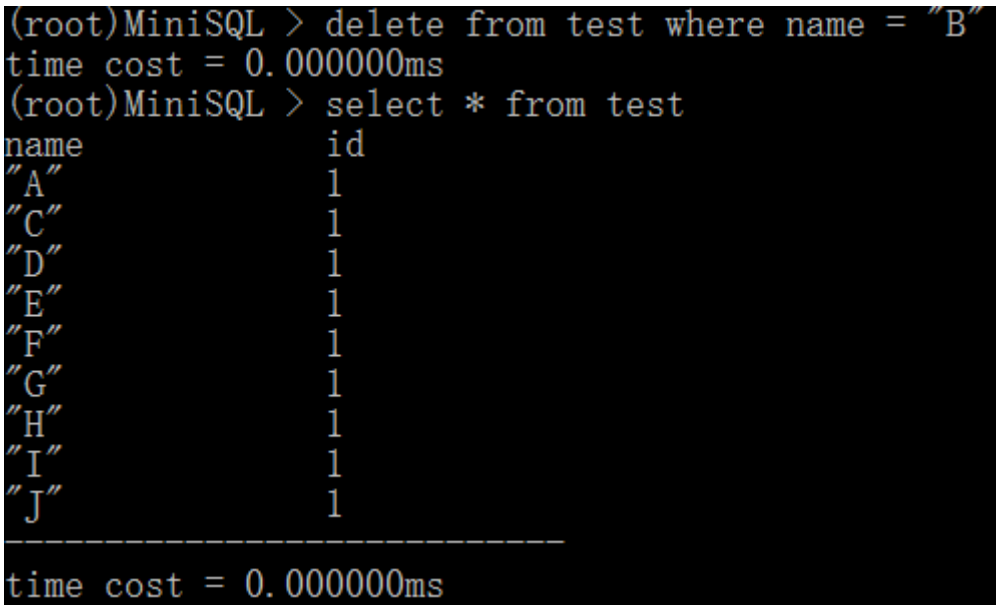
有索引：

```
select * from test where name = "A"
```



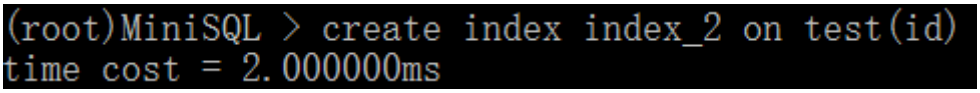
5.4 删除数据

```
delete from test where name = "B"
select * from test
```



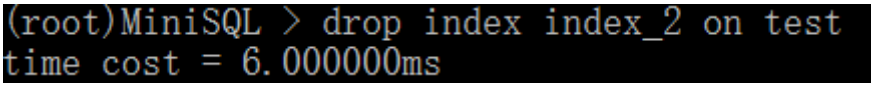
5.5 新建索引

```
create index index_2 on test(id)
```



5.6 删除索引

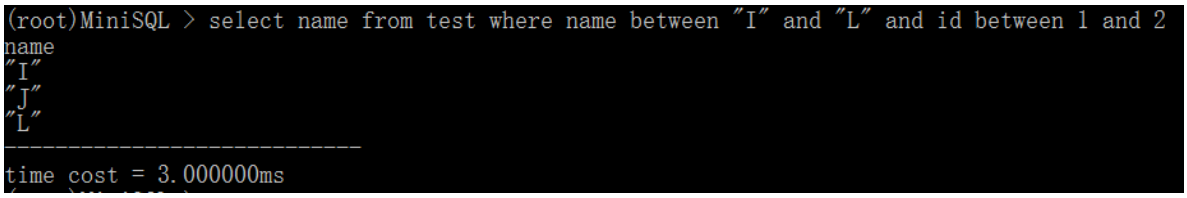
```
drop index index_2 on test
```



5.7 综合查找

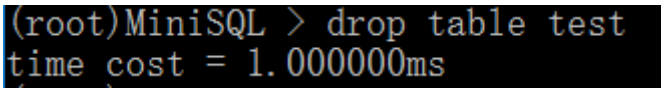
包括单列查找、多条件查找、区间查找

```
insert into test values("K",3)
insert into test values("L",2)
select name from test where name between "I "and "L" and id between 1 and 2
```



5.8 删除表

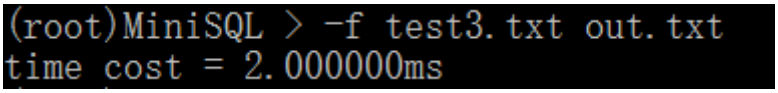
```
drop table test
```



5.9 文件输入

```
//test3.txt
create table test(id int, age int, primary key(id))
insert into test values(7681,25692)
insert into test values(5498,483)
insert into test values(10277,2028)
select id from test where id=48
select id from test where id=49
select id from test where id=50
drop table test
```

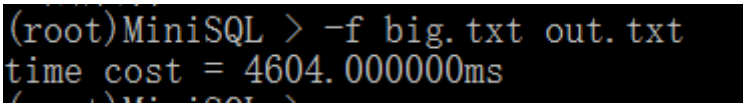
```
-f test3.txt out.txt
```



```
//out.txt
id
7681
----------------------------
id
5498
----------------------------
id
5498
----------------------------
```

5.10 大数据测试

big.txt内共有1条create table语句，10000条insert语句，5000条select语句，1条drop table语句。运行结果如下：



5.11 异常测试

注：因异常种类较多，以下仅截取了部分测试结果：

```
欢迎使用MiniSQL!
请输入您的账号:
> root
请输入您的密码:
> 1234
账号或密码错误，请重新登录
欢迎使用MiniSQL!
请输入您的账号:
> root
请输入您的密码:
> 123456
登陆成功!
(root)MiniSQL > -f
Error: File Not Exists!
(root)MiniSQL > create table table_name(name int)
Error: Primary Key Not Found!
(root)MiniSQL > create table table_name(name int, key char(20), primary
key(null))
Error: Primary Key Not Found!
(root)MiniSQL > create table table_name(name int, key char(20), primary
key(name))
(root)MiniSQL > drop table null
Error: Table Not Exists!
(root)MiniSQL > create index index_name on table_name
Error: Bracket(s) Not Found!
(root)MiniSQL > create index index_name on table_name(null)
Error: Attribute Not Exists!
(root)MiniSQL > drop index null
Error: 'On' Not Found!
(root)MiniSQL > drop index null on table_name
Error: Index Not Exists!
(root)MiniSQL > insert into table_name values(1,1,1)
Error: Number of Attributes is wrong!
```

```
(root)MiniSQL > select * from table_name
No Record Found.
----------------------------
(root)MiniSQL > select from table_name
Error: Attribute CANNOT be NULL!
(root)MiniSQL > select * from null
Error: Table Not Exists!
(root)MiniSQL > select * from table_name where
Error: Expressions After 'WHERE' CANNOT be NULL!
(root)MiniSQL > select * from table_name where name = "2"
No Record Found.
----------------------------
```

六、组内分工

MiniSQL交互界面、主要数据结构、Interpreter模块、Buffer Manager & Catalog Manager模块 ：黄彦玮

API模块、B+-Tree数据结构、 B+-Tree模块、改进Vector容器：付仁泓

调试及测试部分由两人共同完成，每个人主要调试自己所负责的模块。

七、实验心得及改进

本次实验是建立在对课程内容的深入理解上的，前期由于我对于文件存储和索引的理解不够深入，因此走了不少弯路，浪费了不少时间，后来对书上相关内容研读并深入理解后，最终成功的实现了，还是非常有成就感的。本次实验前后耗时约2周，其中主要代码耗时1周，调试及测试耗时1周，本人负责部分大约在1800行左右，极大地锻炼了我的工程能力。

这次实验给我的教训是今后要在实验开始将整个项目的内容及实现方式想清楚，不能只想一个框架就草率分工，特别是数据结构一定要经过组内所有成员仔细确认后才可以开始动手写代码，不然后期如果出了问题解决起来将会非常耗费时间。

我们的项目还有很多地方可以改进。例如，在删除记录之后，可以通过加入free list来将被删除的空间用链表连接起来，将新插入的记录插入到这些空间里去，从而节省内存。又如，我们可以根据数据量的大小动态调整B+-Tree的Order，这样也可以使我们的程序对于不同规模的数据都有不错的运行效率。