

浙江大学

数据库系统实验报告

作业名称:	MiniSQL
姓 名:	官泽隆, 王英豪, 郑昊
学 号:	3180103008, 3180102062, 3180101877
电子邮箱:	学号@zju.edu.cn
联系电话:	18888910213, 15370091919, 17547640164
指导老师:	孙建伶

2020 年 6 月 21 日

总体设计报告

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 要求支持三种基本数据类型: int, char(n), float, 其中 char(n) 满足 $1 \leq n \leq 255$.
2. 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. 支持每次一条记录的插入操作。
6. 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

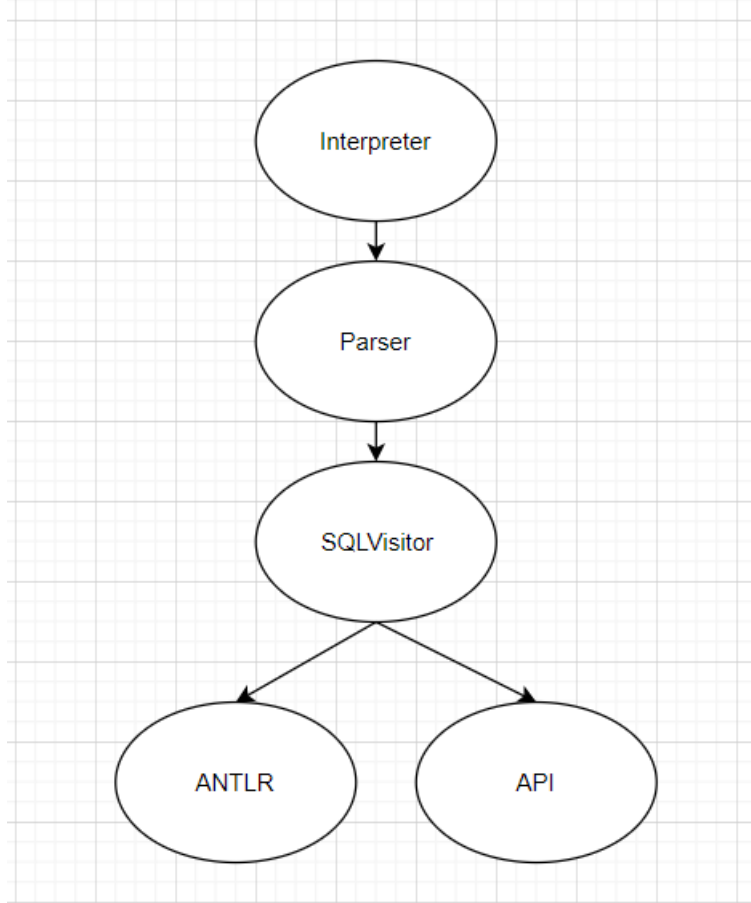
1. Windows 10 Professional
2. Microsoft Visual Studio 2017/2019

四、 系统设计

1. 分工情况
 - i. 王英豪负责模块 Interpreter, API, CatalogManager
 - ii. 官泽隆负责模块 BufferManager, RecordManager
 - iii. 郑昊负责模块 IndexManager

2. Interpreter

- i. 无主要数据结构，使用了 ANTLR 进行 tokenize 和 parsing



- ii.
- iii. 使用 ANTLR 提供的错误处理对语法和词法进行检查，在实际执行中进行语义检查

3. API

```
static SelectResult selectQuery(std::string tableName, std::vector<Condition> conds);  
static QueryResult deleteQuery(std::string tableName, std::vector<Condition> conds);  
static QueryResult createTableQuery(TableDsc info);  
static QueryResult createIndexQuery(IndexDsc info);  
static QueryResult insertQuery(std::string tableName, std::vector<std::string>);  
static QueryResult dropTableQuery(std::string tableName);  
static QueryResult dropIndexQuery(std::string indexName);
```

- i.

关于输入和输出类型的解释

Condition: 用于描述 where 提供的条件

TableDsc: 用于描述 table 的元信息

IndexDsc: 用于描述 Index 的元信息

Insert 中的 string 数组: 插入的 values

SelectResult: select 的结果，包含所查询 table 的元信息和返回的查询结果

QueryResult: 执行结果，包含执行状态和所用时间

- ii. 实现各种查询
- iii. SelectResult:

```

class SelectResult :
public QueryResult {
public:
    std::string print() override;
    SelectResult(int);
    void setSuccess(TableDsc tableInfo, std::vector<std::vector<std::string>> data);
private:
    TableDsc tableInfo;
    std::vector<std::vector<std::string>> data;
};

```

iv.

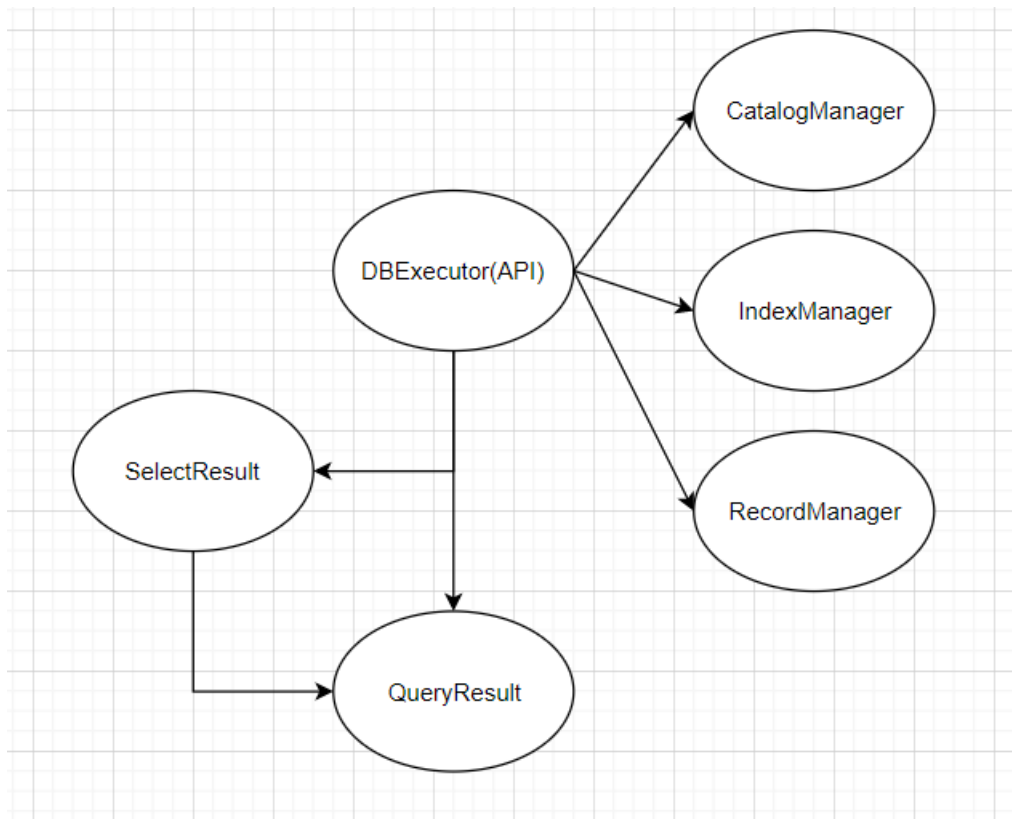
QueryResult

```

class QueryResult:public Result {
public:
    QueryResult();
    QueryResult(int);
    void setSuccess(int cnt);
    int getCount() const;
    int getDuration() const;
    std::string print() override;
protected:
    int count;
    std::chrono::time_point<std::chrono::system_clock> startTime;
    int duration;
};

```

Count 用于给出最后的统计



v.

4. Catalog Manager

- i. 记录数据库中各个 table 以及 index 的元信息
- ii. Column

```
struct Column {
    std::string name;
    std::pair<std::string, int> type;
    bool unique;
    static Column parseString(std::istream& s) {
        Column ret;
        s >> ret.name;
        s >> ret.type.first;
        s >> ret.type.second;
        s >> ret.unique;
        return ret;
    }
    std::string toString() {
        return name + " " + type.first + " " + std::to_string(type.second) + " " + std::to_string(unique);
    }
};
```

Table

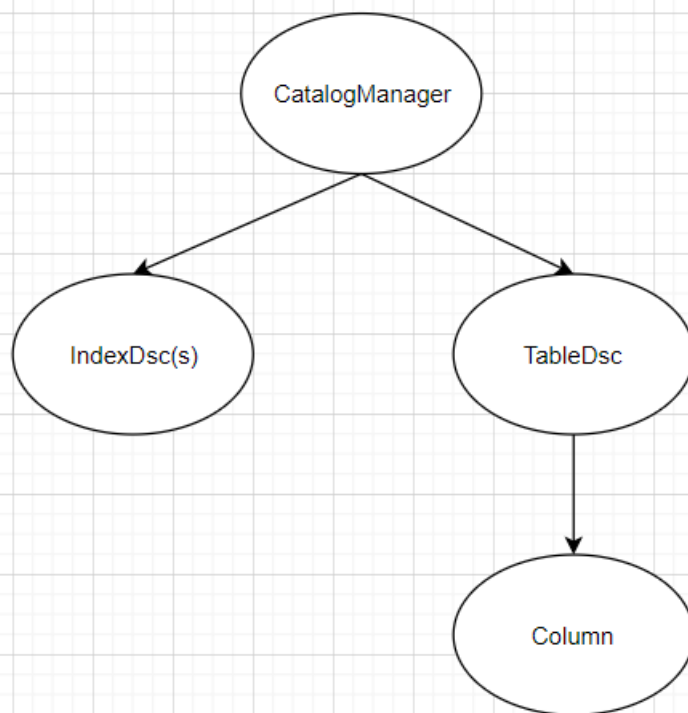
```
struct TableDsc {
    std::string name;
    std::vector<Column> metadata;
    std::string primaryKey;
    static TableDsc parseString(std::istream& s) {
        TableDsc ret;
        int columnCount;
        s >> ret.name;
        s >> columnCount;
        for (int i = 0; i < columnCount; i++) {
            ret.metadata.emplace_back(Column::parseString(s));
        }
        s >> ret.primaryKey;
        return ret;
    }
    std::string toString() {
        std::string s;
        s = name + " " + std::to_string(metadata.size()) + " ";
        for (int i = 0; i < metadata.size(); i++) {
            s += metadata[i].toString() + " ";
        }
        s += primaryKey + " ";
        return s;
    }
};
```

Index

```

struct IndexDsc {
    std::string tableName;
    std::string indexName;
    std::string columnName;
    static IndexDsc parseString(std::istream& s) {
        IndexDsc ret;
        s >> ret.tableName;
        s >> ret.indexName;
        s >> ret.columnName;
        return ret;
    }
    std::string toString() {
        return tableName + " " + indexName + " " + columnName;
    }
};

```



- iii.
- iv. 由于元信息难以分页，以及大小数量都十分有限的特点，catalog manager 自己实现了一套缓存，并不使用 buffer manager 的分页功能。

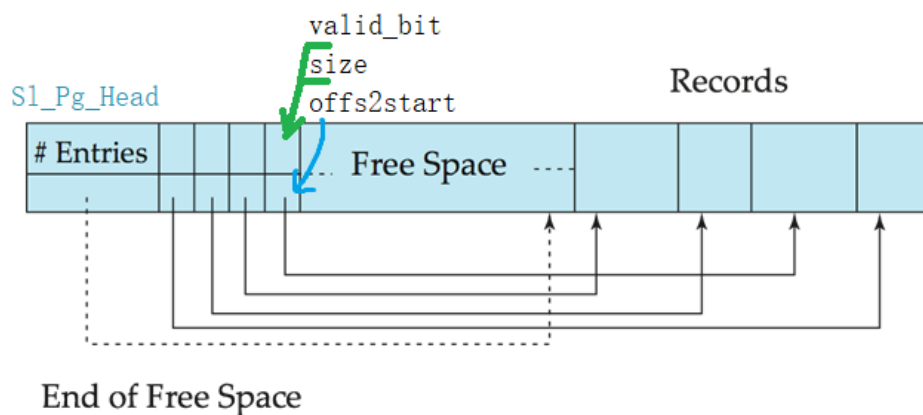
5. Record Manager

- i. 功能描述：实现了设计指导的功能要求“管理记录表中数据的数据文件。主要功能为实现数据文件的创建与删除（由表的定义与删除引起）、记录的插入、删除与查找操作”。具体而言，提供如下接口：

```
int create_table(string tableName); // 0 for success
int drop_table(string tableName); // 0 for success
    // no candidates: full table scan; otherwise index range scan
vector<vector<string>> select(const string &tableName, const vector<Condition> &conds,
    const vector<p_Entry> &candidates = vector<p_Entry>{}); // 返回值是面向显示的
int insert(string tableName, std::vector<std::string> s_vals); // 0 for success
int delete_rec(string tableName, vector<Condition> conds,
    const vector<p_Entry> &candidates = vector<p_Entry>{}); // 返回删除记录条数
void init_index(const string & indexName); // 向新 indx 提供已有键值对，以初始化
```

- ii. 主要数据结构

1. 根据系统需求，把 C++ 对应的三种基本数据类型（int, double, std::string）做了一个 **wrapper** 以适配，提炼一个**抽象基类**（ABC）出来，以使用统一的代码读写各个字段，提高代码可读可维护性。
2. 分槽页（Slotted Page Structure），于课本 ch10 描述过，如下图

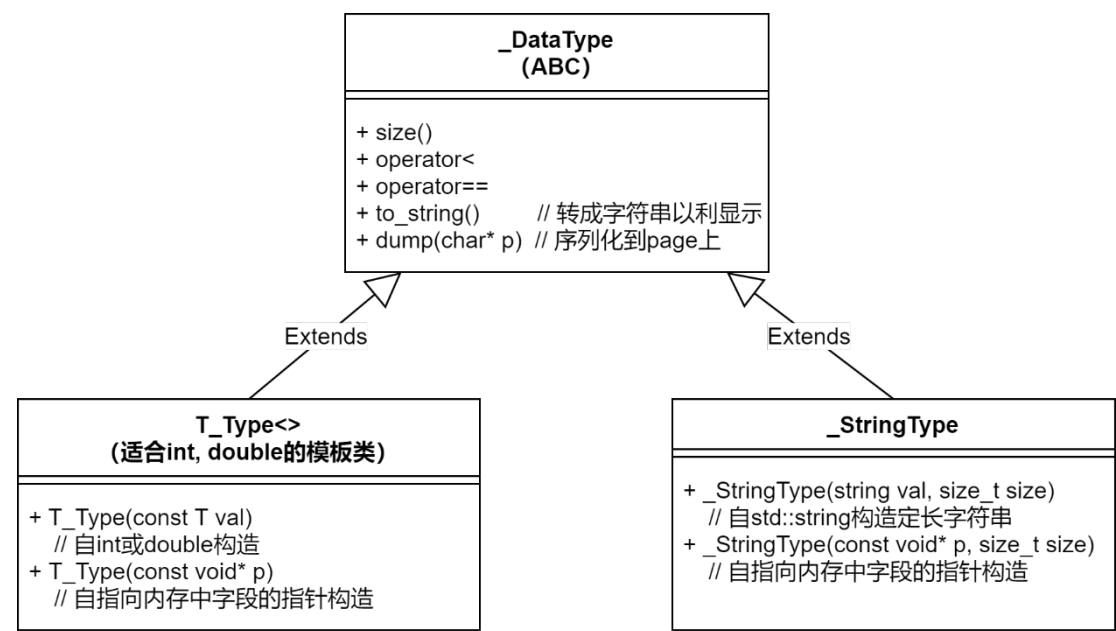


对应 C++ 的结构声明：

```
struct Sl_Pg_Head
{
    unsigned n_entries;
    unsigned end_fspace; // 页内的free space是[第n_entries项 Entry, end_fspace)
    Entry ent;
};
struct Entry
{
    unsigned offs2start; // relative to the start of page
    unsigned short size;
```

```
bool valid;
};
```

3. 接口上使用 (pageNum, offs) 构成的元组 p_Entry 来定位表的 Entry.
- iii. ii.a) 提到的 UML 类图



6. Index Manager

- i. 功能描述: Index Manager 主要负责与索引相关的操作。包括索引的建立与删除(相应地,包括 B+树的建立与删除)以及对键值的插入、删除与修改,同时,IndexManager 提供等值查询与条件查询两种查询方式。提供如下接口:

```
//建立索引
void* CreateIndex(int Type, string IndexName, string TableName, string
Attribute);
//删除索引
int DropIndex(string IndexName);
//搜索 Index
void* GetIndex(string IndexName);
//等值查询
char* IndexSearch(int Type, string IndexName, string Key);
//条件查询,通过 Condition 确定不同条件,0 为小于,1 为小于等于,2 为大于,3 为大于等于
vector<char*> IndexConditionSearch(int Type, string IndexName, string Key,
int Condition);
//插入键值
void IndexInsertion(int Type, string IndexName, string Key, char* Address);
//删除键值
void IndexDeletion(int Type, string IndexName, string Key);
//修改键值
```



```
void IndexUpdate(int Type, string IndexName, string Key, char* Address);
```

ii. 主要数据结构

1. 对于查找到的节点，通过 SearchNode 结构体暂时保存信息：

```
struct SearchNode
{
    Node<Tree>* ptr; //保存节点位置的指针
    size_t position; //保存位置
    bool Exist; //记录是否存在
};
```

2. 对于传入的 Key 值，默认其均为 string 类型，将其转换成所需要的类型时，例如：int、float、char，并保存在 convert 结构体中：

```
//转换 Key 的类型，将其从 string 转换为 int、float、char
struct convert{
    int inttype;
    float floattype;
    string stringtype;
}ck;
```

3. IndexInfo 主要用于保存一个 Index 的信息，包括数据类型（-1 表示 int，0 表示 float，1-255 为 char，具体数字表示其长度）、索引名、表名等：

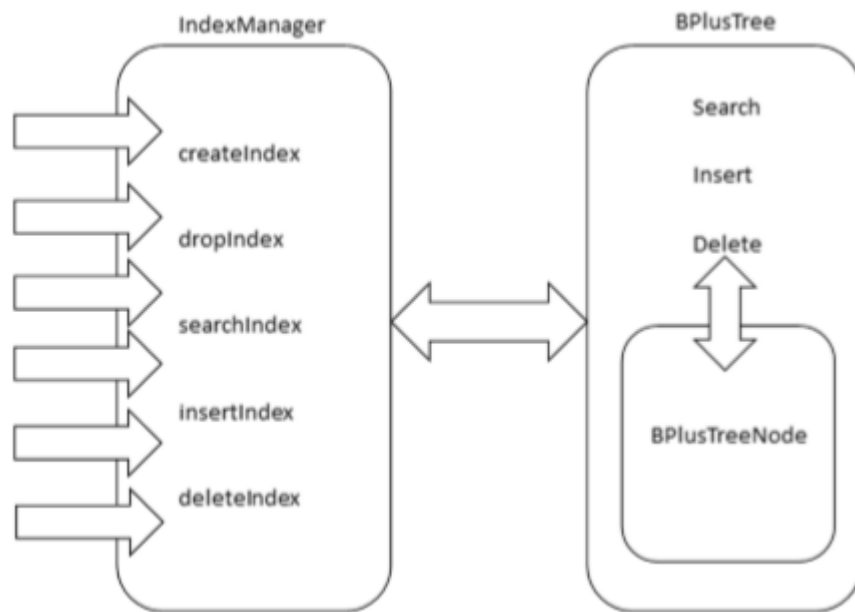
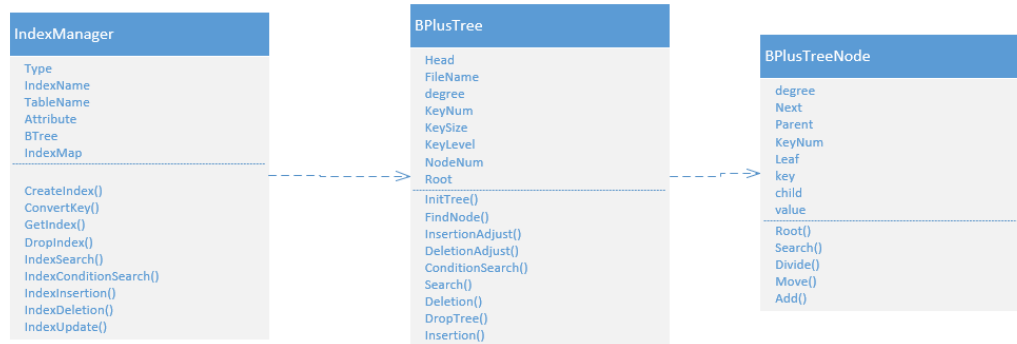
```
struct IndexInfo
{
public:
    //type 表示数据类型，-1 为 int，0 为 float，1-255 为 char
    int Type;
    string IndexName; //索引名
    string TableName; //表名
    string Attribute; //属性名
    void* BTree;
};
```

4. 通过 map，保存所有 Index 的信息：

```
//保存所有 Index 的信息
map<int, IndexInfo*> IndexMap;
```

iii. 说明设计的类(如果有)，以及类间关系

IndexManager 主要提供外部接口，并根据需求通过 BPlusTree 类中的函数对 B+树进行操作。BPlusTree 类用于建立 B+树，它由 B+树结点构成。结点由 BPlusTreeNode 类实现，其中存储键值信息和一些指针。B+树中除了结点外，还需其他的必要信息（如树的根节点、结点数等）。几个类的关系如下图所示：



7. Buffer Manager

- i. 功能描述: Buffer Manager 是一个全局单例。所有使用 Buffer Manager 进行文件数据管理的客户将共享一个由许多内存页组成的缓冲区。但对客户而言,缓冲区是透明的, Buffer Manager 就提供以页为单位的文件读写。具体而言,提供如下接口:

```

static BufferManager& instance() { return bm; }      // 获取全局单例
~BufferManager();                                // 析构函数
const void* getPage_r(p_Page p);                  // 获取用于读的页面
void* getPage_w(p_Page p);                         // 获取用于写的页面
void set_dirty(p_Page p);                          // 调用后,允许在原来只读的页面做写操作
unsigned totalPages(string filename);               // 获取某文件的总页面数
// int: 0 on success below
int pinPage(p_Page p);                             // 锁定缓冲区的页,不允许替换出去
void unpinPage(p_Page p);                           // 解锁缓冲区的页,允许替换
int create_file(string filename);                   // 创建文件
int delete_file(string filename);                   // 删除文件
  
```

```
void flush();
```

```
// 将所有脏页写回磁盘
```

ii. 主要数据结构:

1. 接口上, 使用(filename, pageNum)组成的元组 p_Page 来定位文件中的页
2. 使用哈希表来快速由 p_Page 映射到实际的内存页
`unordered_map<p_Page, Buf_Page, MyHash> pages;`
3. 使用哈希表维护文件的总页数
`unordered_map<string, unsigned> _totalPages;`

8. DB Files

- i. 表元信息, 拓展名为.tbl, 每个表占用一个文件单独保存。格式为:
表名 列数 列信息 主键名, 各部分用空格隔开
其中每一条列信息格式为: 列名 类型 是否为 unique
- ii. 索引元信息, 拓展名为.idx, 所有的索引保存在同一个文件内, 格式为:
索引数量 索引信息
其中每一条索引信息格式为:
表名 索引名 列名
- iii. tableName.mdbf 以二进制形式存储 tableName 表中的记录数据。一瞥

```
00001f90 65 31 33 32 27 00 00 00 00 00 00 00 00 53 40 e132'.....S@
00001fa0 23 05 61 40 27 6E 61 6D 65 31 33 31 27 00 00 00 #.a@'name131'...
00001fb0 00 00 00 00 00 60 50 40 22 05 61 40 27 6E 61 6D .....P@'.a@'nam
00001fc0 65 31 33 30 27 00 00 00 00 00 00 00 00 C0 4D 40 e130'.....M@
00001fd0 21 05 61 40 27 6E 61 6D 65 31 32 39 27 00 00 00 !.a@'name129'...
00001fe0 00 00 00 00 00 E0 58 40 20 05 61 40 27 6E 61 6D .....X@'.a@'nam
00001ff0 65 31 32 38 27 00 00 00 00 00 00 00 00 20 50 40 e128'.....P@
00002000 7F 00 00 00 18 04 00 00 E8 0F 00 00 18 00 01 00 .....
00002010 D0 0F 00 00 18 00 01 00 E8 0F 00 00 18 00 01 00 .....
00002020 A0 0F 00 00 18 00 01 00 88 0F 00 00 18 00 01 00 .....
00002030 70 0F 00 00 18 00 01 00 58 0F 00 00 18 00 01 00 p.....X.....
00002040 40 0F 00 00 18 00 01 00 28 0F 00 00 18 00 01 00 @.....(.
00002050 10 0F 00 00 18 00 01 00 58 0F 00 00 18 00 01 00
```

- iv. BUF_META 以文本文件的形式存储各.mdbf 文件的总页数

```
BUF_META  BufferManager.h
1 student2.mdbf 79
2
```

五、 系统实现分析及运行截图

1. 创建表语句

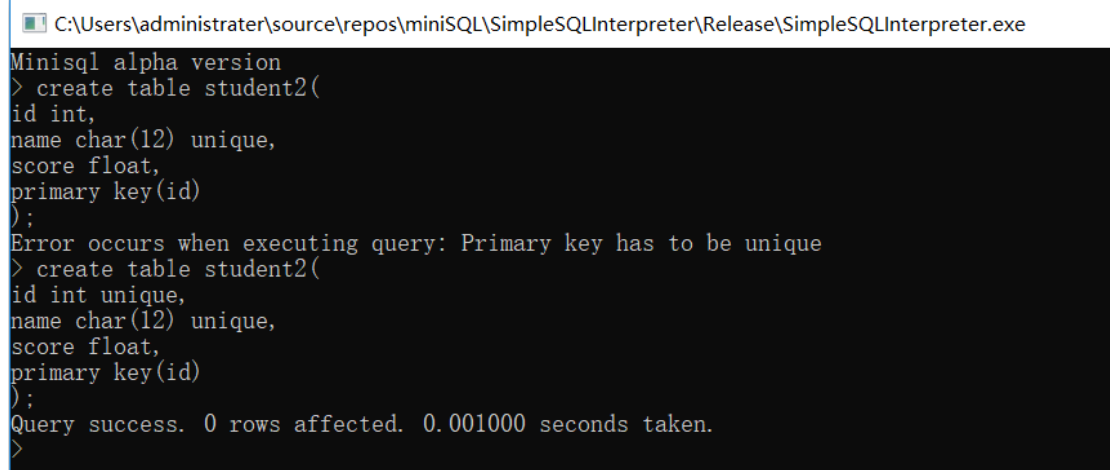
1. 若该语句执行成功, 则输出执行成功信息; 若失败, 必须告诉用户失败的原因。
2. 分析执行该 SQL 所需的模块以及程序执行流程
解释器调用 DBExecutor::createTableQuery(TableInfo info), 其中进行必要的数据完整性检查和重复检查, 最后调用 int RecordManager::create_table(string tableName) 创建表的数据文件, 调用 CatalogManager::updateTableInfo(TableInfo tableInfo) 在 CatalogManager 中写表的元信息, 调用 CatalogManager 和 IndexManager 进行主键索引的建立,

最后将结果和所用时长输出到屏幕上

3. 运行结果截图

示例语句：

```
create table student2(  
    id int,  
    name char(12) unique,  
    score float,  
    primary key(id)  
);
```



```
C:\Users\administrater\source\repos\miniSQL\SimpleSQLInterpreter\Release\SimpleSQLInterpreter.exe  
Minisql alpha version  
> create table student2(  
id int,  
name char(12) unique,  
score float,  
primary key(id)  
);  
Error occurs when executing query: Primary key has to be unique  
> create table student2(  
id int unique,  
name char(12) unique,  
score float,  
primary key(id)  
);  
Query success. 0 rows affected. 0.001000 seconds taken.  
>
```

2. 删除表语句

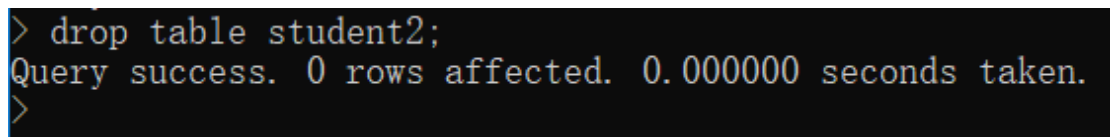
1. 若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。
2. 分析执行该 SQL 所需的模块以及程序执行流程

解释器调用 `DBExecutor::dropTableQuery(std::string tableName)`，其中进行必要的数据库完整性检查和重复检查，最后调用 `int RecordManager::drop_table(string tableName)` 删除表的数据文件，调用 `CatalogManager::dropTable(const std::string& tableName)` 在 `CatalogManager` 中删除表的元信息，调用 `CatalogManager` 和 `IndexManager` 删除主键索引，最后将结果和所用时长输出到屏幕上

3. 运行结果截图

示例语句：

```
drop table student2;
```



```
> drop table student2;  
Query success. 0 rows affected. 0.000000 seconds taken.  
>
```

3. 创建索引语句

1. 若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。
2. 分析执行该 SQL 所需的模块以及程序执行流程

解释器调用 DBExecutor::createIndexQuery(IndexInfo info), 其中进行必要的数据库完整性检查和重复检查, 调用 CatalogManager::updateIndex(IndexInfos indexInfo)在 CatalogManager 中写入索引的元信息, 调用 IndexManager 建立索引, void RecordManager::init_index(const string & indexName)初始化索引, 最后将结果和所用时长输出到屏幕上

3. 运行结果截图

由于 IndexManager 部分未能与系统进行整合, 无法得出运行结果截图, 具体原因请参照 IndexManager 部分的个人实验报告。

示例语句:

```
create index sidx on student ( name );
```

4. 删除索引语句

1. 若该语句执行成功, 则输出执行成功信息; 若失败, 必须告诉用户失败的原因。

2. 分析执行该 SQL 所需的模块以及程序执行流程

解释器调用 DBExecutor::dropIndexQuery(IndexInfo info), 其中进行必要的数据库完整性检查和重复检查, 调用 CatalogManager::dropIndexQuery(std::string indexName)在 CatalogManager 中删除索引的元信息, 调用 IndexManager 删除索引, 最后将结果和所用时长输出到屏幕上

3. 运行结果截图

由于 IndexManager 部分未能与系统进行整合, 无法得出运行结果截图, 具体原因请参照 IndexManager 部分的个人实验报告。

示例语句:

```
drop index sidx;
```

5. 选择语句

1. 若该语句执行成功且查询结果不为空, 则按行输出查询结果, 第一行为属性名, 其余每一行表示一条记录; 若查询结果为空, 则输出信息告诉用户查询结果为空; 若失败, 必须告诉用户失败的原因。

2. 分析执行该 SQL 所需的模块以及程序执行流程

解释器调用 DBExecutor::selectQuery(std::string tableName, std::vector<Condition> conds), 其中进行必要的数据库完整性检查和重复检查, 调用 vector<vector<string>> RecordManager::select(const string & tableName, const vector<Condition>& conds, const vector<p_Entry>& candidates)返回字符串形式的多行记录 (每行记录有固定多个字段). 格式化后打印到屏幕上

3. 运行结果截图

示例语句:

```
select * from student2;  
select * from student2 where id = 1080100978;  
select * from student2 where score > 20.0 and name <> 'name123';
```

C:\Users\administrater\source\repos\miniSQL\SimpleSQLInterpreter\Release\Sir

```
1080109973 'name9973' 76.500000
1080109974 'name9974' 52.500000
1080109975 'name9975' 97.000000
1080109976 'name9976' 50.000000
1080109977 'name9977' 56.500000
1080109978 'name9978' 84.500000
1080109979 'name9979' 99.500000
1080109980 'name9980' 88.000000
1080109981 'name9981' 78.500000
1080109982 'name9982' 53.500000
1080109983 'name9983' 58.500000
1080109984 'name9984' 64.500000
1080109985 'name9985' 71.000000
1080109986 'name9986' 61.000000
1080109987 'name9987' 91.500000
1080109988 'name9988' 75.000000
1080109989 'name9989' 58.000000
1080109990 'name9990' 75.000000
1080109991 'name9991' 69.000000
1080109992 'name9992' 50.500000
1080109993 'name9993' 67.500000
1080109994 'name9994' 97.500000
1080109995 'name9995' 59.500000
1080109996 'name9996' 65.500000
1080109997 'name9997' 61.000000
1080109998 'name9998' 84.500000
1080109999 'name9999' 69.500000
1080110000 'name10000' 80.500000
Query success. 10000 rows in set. 0.440000 seconds taken.
>
```

```
> select * from student2 where id = 1080100978;
id      name      score
1080100978 'name978' 51.500000
Query success. 1 rows in set. 0.048000 seconds taken.
>
```

C:\Users\administrater\source\repos\miniSQL\SimpleSQLInterpreter\Release\Simp

```
1080109973 'name9973' 76.500000
1080109974 'name9974' 52.500000
1080109975 'name9975' 97.000000
1080109976 'name9976' 50.000000
1080109977 'name9977' 56.500000
1080109978 'name9978' 84.500000
1080109979 'name9979' 99.500000
1080109980 'name9980' 88.000000
1080109981 'name9981' 78.500000
1080109982 'name9982' 53.500000
1080109983 'name9983' 58.500000
1080109984 'name9984' 64.500000
1080109985 'name9985' 71.000000
1080109986 'name9986' 61.000000
1080109987 'name9987' 91.500000
1080109988 'name9988' 75.000000
1080109989 'name9989' 58.000000
1080109990 'name9990' 75.000000
1080109991 'name9991' 69.000000
1080109992 'name9992' 50.500000
1080109993 'name9993' 67.500000
1080109994 'name9994' 97.500000
1080109995 'name9995' 59.500000
1080109996 'name9996' 65.500000
1080109997 'name9997' 61.000000
1080109998 'name9998' 84.500000
1080109999 'name9999' 69.500000
1080110000 'name10000' 80.500000
Query success. 9999 rows in set. 0.445000 seconds taken.
>
```


6. 插入记录语句

1. 若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

2. 分析执行该 SQL 所需的模块以及程序执行流程

解释器调用 DBExecutor:: insertQuery(std::string tableName, std::vector<std::string>), 其中进行必要的数据库完整性检查和重复检查, 调用 int RecordManager::insert(string tableName, std::vector<std::string> s_vals) 将记录插入数据文件。RecordManager 内部将进行 unique 键检查, 如果不通过将返回 1; 否则将记录插入并返回 0. 最后将结果和所用时长输出到屏幕上

3. 运行结果截图

示例语句:

```
insert into student2 values (1008612315,'name999',59.9);
```

```
> insert into student2 values (1008612315,'name999',59.9);
Query success. 1 rows affected. 0.048000 seconds taken.
>
```

7. 删除记录语句

1. 若该语句执行成功，则输出执行成功信息，其中包括删除的记录数；若失败，必须告诉用户失败的原因。

2. 分析执行该 SQL 所需的模块以及程序执行流程

解释器调用 DBExecutor:: deleteQuery(std::string tableName, std::vector<Condition> conds), 其中进行必要的数据库完整性检查和重复检查调用 int RecordManager:: delete_rec(string tableName, vector<Condition> conds, const vector<p_Entry> &candidates = vector<p_Entry>{}) 删除符合条件的记录并返回删除的记录数. 最后将结果和所用时长输出到屏幕上。

3. 运行结果截图

示例语句:

```
delete from student2;
delete from student2 where id = 1080100978;
```

```
> delete from student2 where id = 1080100978;
Query success. 1 rows affected. 0.048000 seconds taken.
>
```

```
> delete from student2;
Query success. 10000 rows affected. 0.077000 seconds taken.
>
```

8. 退出 MiniSQL 系统语句

该语句的语法如下:

quit;

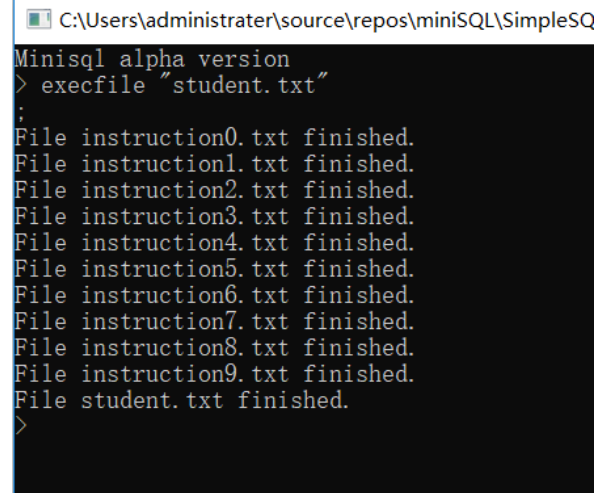
9. 执行 SQL 脚本文件语句

该语句的语法如下:

execfile 文件名 ;

SQL 脚本文件中可以包含任意多条上述 SQL 语句 (包括 execfile 本身), MiniSQL 系统读入该文件, 然后按序依次逐条执行脚本中的 SQL 语句。

1. 若该语句执行成功，则输出执行成功的命令数；若失败，必须告诉用户失败的原因。
2. 运行结果截图



```
C:\Users\administrater\source\repos\miniSQL\SimpleSQ
Minisql alpha version
> execfile "student.txt"
;
File instruction0.txt finished.
File instruction1.txt finished.
File instruction2.txt finished.
File instruction3.txt finished.
File instruction4.txt finished.
File instruction5.txt finished.
File instruction6.txt finished.
File instruction7.txt finished.
File instruction8.txt finished.
File instruction9.txt finished.
File student.txt finished.
>
```