

MInSQL

设计报告

专业：计算机科学与技术

学号：3180105566

姓名：刘振东

组员：任浩然、秦兆祥

2020 年 6 月 25 日

目录

1.实验目的.....	3
2.实验需求.....	3
2.1 需求概述.....	3
3.设计框架.....	3
3.1 系统体系结构.....	3
3.2 模块概述.....	4
3.2.1 Interpreter.....	4
3.2.2 API.....	4
3.2.3Catalog Manager.....	4
3.2.4Record Manager.....	5
3.2.5 Index Manager.....	5
3.2.6 Buffer Manager.....	5
4.具体实现.....	5
4.1 语法说明.....	5
4.2 各个模块的函数接口说明.....	7
4.2.1 Interpreter.....	7
4.2.2 API.....	8
4.2.3Catalog Manager.....	10
4.2.4Record Manager.....	11
4.2.5 Index Manager.....	12
4.2.6 Buffer Manager.....	13
4.2.7 精妙的 database 设计.....	14
4.3 操作演示.....	17
4.3.1 建立表.....	17
4.3.2 插入数据.....	18
4.3.3 删除数据.....	19
4.3.4 创建索引.....	20
4.3.5 删除表.....	21
4.3.6 删除索引.....	21
4.3.7 查询.....	22
4.3.8 离开.....	22
5.分工与心得.....	23
5.1 分工.....	23
5.2 心得.....	23

1.实验目的

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。

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。

2.实验需求

2.1 需求概述

数据类型

只要求支持三种基本数据类型: int, char(n), float, 其中 char(n)满足 $1 \leq n \leq 255$ 。

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; 支持单属性的主键定义。

索引的建立和删除

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 (因此, 所有的 B+树索引都是单属性单值的)。

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。

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。

3.设计框架

3.1 系统体系结构

MiniSQL 体系结构如下:

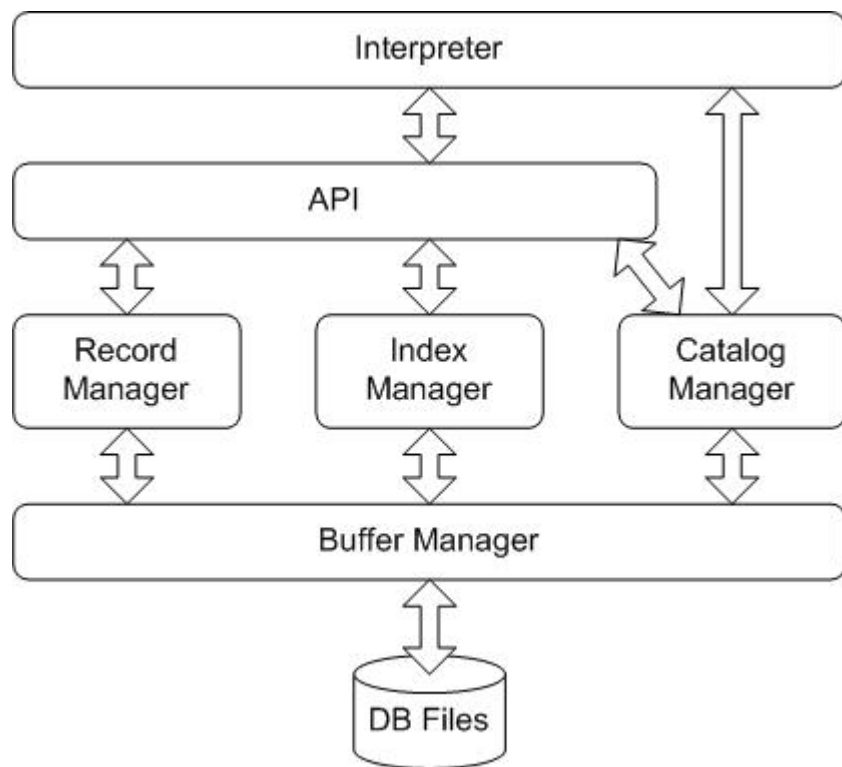


图 1 MiniSQL 体系结构

3.2 模块概述

3.2.1 Interpreter

Interpreter 模块直接与用户交互，主要实现以下功能：

- 1.程序流程控制，即“启动并初始化 → ‘接收命令、处理命令、显示命令结果’循环 → 退出”流程。
- 2.接收并解释用户输入的命令，生成命令的内部数据结构表示，同时检查命令的语法正确性和语义正确性，对正确的命令调用 API 层提供的函数执行并显示执行结果，对不正确的命令显示错误信息。

3.2.2 API

API 模块是整个系统的核心，其主要功能为提供执行 SQL 语句的接口，供 Interpreter 层调用。该接口以 Interpreter 层解释生成的命令内部表示为输入，根据 Catalog Manager 提供的信息确定执行规则，并调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口进行执行，最后返回执行结果给 Interpreter 模块。

3.2.3 Catalog Manager

Catalog Manager 负责管理数据库的所有模式信息，包括：

1. 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. 表中每个字段的定义信息，包括字段类型、是否唯一等。
3. 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。

Catalog Manager 还必需提供访问及操作上述信息的接口，供 Interpreter 和 API 模块使用。

3.2.4 Record Manager

Record Manager 负责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，并对外提供相应的接口。其中记录的查找操作要求能够支持不带条件的查找和带一个条件的查找（包括等值查找、不等值查找和区间查找）。

数据文件由一个或多个数据块组成，块大小应与缓冲区块大小相同。一个块中包含一条至多条记录，为简单起见，只要求支持定长记录的存储，且不要求支持记录的跨块存储。

3.2.5 Index Manager

Index Manager 负责 B+树索引的实现，实现 B+树的创建和删除（由索引的定义与删除引起）、等值查找、插入键值、删除键值等操作，并对外提供相应的接口。

B+树中节点大小应与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。

3.2.6 Buffer Manager

Buffer Manager 负责缓冲区的管理，主要功能有：

1. 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文件
2. 实现缓冲区的替换算法，当缓冲区满时选择合适的页进行替换
3. 记录缓冲区中各页的状态，如是否被修改过等

为提高磁盘 I/O 操作的效率，缓冲区与文件系统交互的单位是块，块的大小应为文件系统与磁盘交互单位的整数倍，一般可定为 4KB 或 8KB。

4. 具体实现

4.1 语法说明

MiniSQL 支持标准的 SQL 语句格式，每一条 SQL 语句以分号结尾，一条 SQL 语句可写在一行或多行。为简化编程，要求所有的关键字都为小写。在以下语句的语法说明中，用黑体显示的部分表示语句中的原始字符串，如 **create** 就严格的表示字符串“create”，否则含有特殊的含义，如 表名 并不是表示字符串“表名”，而是表示表的名称。

创建表语句

该语句的语法如下：

```
create table 表名 (  
    列名 类型 ,  
    列名 类型 ,  
  
    列名 类型 ,
```

```
primary key ( 列名 )  
);
```

其中类型的说明见第二节“功能需求”。

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。

示例语句：

```
create table student (  
    sno char(8),  
    sname char(16) unique,  
    sage int,  
    sgender char (1),  
    primary key ( sno )  
);
```

删除表语句

该语句的语法如下：

```
drop table 表名 ;
```

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。

示例语句：

```
drop table student;
```

创建索引语句

该语句的语法如下：

```
create index 索引名 on 表名 ( 列名 );
```

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。

示例语句：

```
create index stunameidx on student ( sname );
```

删除索引语句

该语句的语法如下：

```
drop index 索引名 ;
```

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。

示例语句：

```
drop index stunameidx;
```

选择语句

该语句的语法如下：

```
select * from 表名 ;
```

或：

```
select * from 表名 where 条件 ;
```

其中“条件”具有以下格式：列 op 值 and 列 op 值 … and 列 op 值。

op 是算术比较符： = <> < > <= >=

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。

示例语句:

```
select * from student;  
select * from student where sno = '88888888';
```

插入记录语句

该语句的语法如下:

```
insert into 表名 values ( 值1 , 值2 , ... , 值n );
```

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。

示例语句:

```
insert into student values ('12345678', 'wy', 22, 'M');
```

删除记录语句

该语句的语法如下:

```
delete from 表名 ;
```

或:

```
delete from 表名 where 条件 ;
```

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。

示例语句:

```
delete from student;  
delete from student where sno = '88888888';
```

退出 MiniSQL 系统语句

该语句的语法如下:

```
exit;
```

4.2 各个模块的函数接口说明

4.2.1 Interpreter

这个模块调用底层会接受中断信息, 如果 `catch` 则执行失败, 否则成功

```
API my_api;
```

1) `BufferManager` 的设置引用。

为 API 中的 `RecordManager`、`CatalogManager`、`BufferManager`, 分配一个共用的 `buffer` 数组, 分别通过各个模块的 `set_buffer` 函数实现。

```
void set_buffer(BufferManager& test_buffer);
```

2) `insert` 语句的函数调用接口。

主要是用于对输入的语句进行解释, 处理语法错误, 传给 API, 根据指令不同, 实现不同的函数

```
void insert_sql(string tablename, MyTuper& temptuple); //insert
```

3) 处理 WHERE 条件 ($\geq, >, =, <, \leq$) , 从 string 类来判断属于哪一个 WHERE 类型。其中 WHERE 是一个枚举, 实现五种比较。WHERE 的定义在 database.h 里面

```
WHERE get_where(string test_str);
```

4) where 语句后面的信息处理, type 决定 select 还是 delete
处理后调用 api 实现 select or delete

```
void process_where(string test_str, int t,int type);
```

5) create table 语句接口, 调用 API

```
void create_table_sql  
(string tablename,Attribute& temp_Attribute,string Pri_key);
```

6) create index 语句接口, 调用 API

```
void create_index_sql  
(string indexname,string tablename,string attrname); //create index
```

7) 对 create table 语句下面的几行数据进行处理
包含属性定义以及主码定义

```
void test_str_process(string test_str, Attribute& temp_Attribute);
```

8) drop table / drop index 接口, 调用 API

```
void drop_table_sql(string tablename); //drop table  
void drop_index_sql(string indexname); //drop index
```

9) delete 语句接口 (无条件 / 有条件)
value 是具体值, 三种数据类型都先用 string 表示
AttriName 表示具体是哪一个属性名

```
void delete_tuple_sql(string tablename);  
void delete_tuple_sql(string tablename,  
                        WHERE tempwhere,string value,string AttriName);
```

10) select *语句接口 (无条件 / 有条件)

```
void select_tuple_sql(string tablename);  
void select_tuple_sql(string tablename,  
                        WHERE tempwhere, string value, string AttriName);
```

4.2.2 API

```
private:  
    CatalogManager API_Catalog;  
    RecordManager API_Record;  
    IndexManager API_Index;  
    vector<MyTable*> MyTableDef;
```

1) 为 CatalogManager、RecordManager、IndexManager 引用 BufferManager

```
void set_buffer(BufferManager& temp_buffer) {
```



```
API_Catalog.set_Buffer(temp_buffer);
API_Record.set_Buffer(temp_buffer);
API_Index.set_Buffer(temp_buffer);
}
```

2) api 中的 insert into 语句的接口。

首先判断表是否存在，若表不存在发出中断；

如果存在 **PrimaryKey** 和 **Unique** 的属性，根据 **index** 的有无，调用 **Index** 模块判断是否重复，然后调用 **RecordManager** 插入 **buffer** 数组。**api** 获取一个插入位置以及相关的信息，调用 **IndexManager**，更新 **B+ tree**；

```
void api_insert_table(string tablename, MyTuper&tempmytupler);
```

3) **api** 中判断表的存在位置，是指内存中的一个 **MyTable** 的 **vector** 的位置，因为 **api** 中有 **vector<MyTable*> MyTableDef**，从中判断表有没有存在。

```
int Istable(string tablename);
```

4) api 中创建表语句的接口

首先从 **vector** 里面检查表是否存在，如不存在则从硬盘读取，若存在，**throw** 中断；若仍然不存在，则表名未重复，调用 **CatalogManager**，同时调用 **IndexManager** 生成主码的索引文件。

```
void api_create_table
(string tablename, Attribute &temp_attribue, string Pri_key);
```

5) api 中创建索引语句的接口，调用 CatalogManager 和 IndexManager

会首先判断 **indexname** 是否已经存在，若存在发出中断

```
void api_create_index
(string indexname, string tablename, string attrname);
```

6) api 中 drop table 语句的接口，后者是从一个 MyTable 的 vector 里面删除

会判断 **table** 是否存在（先从 **vector**，若没有再到内存），若不存在则发出中断

```
void drop_table(string tablename);
void drop_table_fromDef(string TableName);
```

7) api 中 drop index 语句的接口

会判断 **index** 是否存在，若不存在则发出中断

```
void drop_index(string indexname, string tablename);
```

8) api 中查询语句的接口（有条件 / 无条件）

若是有条件则首先检查判断条件是否存在索引，若存在，则从 **IndexManager** 里面 **select**；

若不存在索引，则从硬盘中读取，此时是线性的查找。

```
void select_tuple_sql(string tablename);
void select_tuple_sql
(string tablename, WHERE tempwhere, string value, string AttriName);
```

9) api 中删除语句的接口 (有条件 / 无条件)

若是有条件则首先检查判断条件是否存在索引, 若存在, 则从 **IndexManager** 里面查询符合条件的 **tuple**, 同时对 **buffer** 进行操作, **delete** 数据;

若不存在索引, 则从硬盘中读取, 此时是线性的查找再删除。

```
void delete_tuple_sql(string tablename);  
void delete_tuple_sql  
(string tablename, WHERE tempwhere, string value, string AttriName);
```

10) 从 **index** 获取表的名字, 以便于 **drop index**

api 删除索引时需要的接口, 调用 **IndexManager** 实现

```
string get_tablename_fromindex(string idnexname);
```

4.2.3 Catalog Manager

成员: **BufferManager*** **m_Buffer_manager**;

1) **get theTable** 从 **table** 的名字, 在 **api** 中 **push back**

主要是调用 **bufferManager** 的 **getbuffenum**, 从硬盘中的表头文件获取到表的定义信息, 返回一个 **table** 类给 **api**。

表头的定义是:

第一行: 数据文件对应的块数

第二行: 属性数 **AN**

接着 **AN** 行: 属性名 属性类型 是否 **unique**

第 **AN+3** 行: 主码的位置 (没有则-1)

第 **AN+4** 行: **index** 数 **IN**;

接着 **IN** 行: **index** 名称 属性索引

```
MyTable* Get_My_Table(string tablename);
```

2) **CatalogManager** 中的 **BufferManager** 引用

```
void set_Buffer(BufferManager& temp_buffer);
```

3) **CatalogManager** 建立表, 包含 **tablename_head.table** 和 **tablename.table** 文件, 调用底层的 **BufferManager**, 从而实现给文件分块的操作。

```
void create_table(MyTable& temp_table);
```

4) 从一个 **Mytable** 类里面读取包含表定义信息的 **string**

此时的 **table** 是一个 **table** 类。在 **database.h** 中实现

```
string get_table_define(MyTable& temp_table);
```

5) **drop table** 实现

首先设置 **BufferManager** 的相关的块无效, 然后用 **remove** 函数删除文件 (包含表头文件、数据文件、**index** 文件)。

```
void drop_table(string tablename);
```

6) create index 实现

只是通过 BufferManger 增加 index 的定义, 通过一个 table 的引用实现 addindex, 在将表 table 对应的定义信息转换为 char*, 通过 BufferManager 写入到表头文件。

```
void create_index  
(MyTable& temp_table, string indexname, int indexAttri);
```

7) drop index 实现

只是通过 BufferManger 去掉相关的 index 的定义, 通过一个 table 类, 去除 index, 然后更新 buffer 得以实现。

```
void drop_index(MyTable& temp_table);
```

4.2.4 Record Manager

成员:

```
BufferManager* m_Buffer_manager;  
My_place the_place;
```

1) RecordManager set_buffer

同 CatalogManager

```
void set_Buffer(BufferManager& temp_buffer) ;
```

2) 自定义比较方式, 返回值 1 满足条件, 针对 char(n)

```
int compare(WHERE tempwhere, char A[], char B[], int size);
```

3) 从 tablename 获取 blocknum

通过 BufferManager 中的存的表头文件实现读取 blocknum

```
int get_blocknum(string tablename);
```

4) 从 tablename 获取属性数

通过 BufferManager 中的存的表头文件实现读取属性数量

```
int get_Attrinum(string tablename);
```

5) insert 接口

调用底层的 BufferManager 实现, 首先获取一个插入位置, 然后将 tuple 转换为字节流插入。具体有 int / float / char(n) 到字节流的转换。存在 BufferManager 的 buffer 数组的 blockcontent 里面, 退出程序, 调用析构函数函数的时候实现。

```
void insert_tuple_table(MyTable& temp_table, MyTuper& temp_tuper);
```

6) 无条件删除

清空 .table 文件, 文件保留。

直接对 BufferManager 的 buffer 数组的 blockcontent 里面的所有 block 进行置 0; (memset 函数)

```
void delete_from_table(string tablename, int blocknum);
```

7) 删除 (select 为 0) / 查询 (select 为 1)

如果是删除, 找到符合条件的 tuple, 对 onetuplesize+1 的空间置 0

```
void select_or_delete_from_table(bool select, MyTable& test_table, WHERE  
tempwhere, string value, string Attriname);
```

8) 无条件 select

线性输出, 调用 print 函数, 对每一个 tuple 的输出

```
void serach_in_table(string tablename, MyTable& temp_table);
```

9) 每一条元组的输出

针对给出的属性的类, 进行输出, 从字节流到 int / float / char (n) 的转换

```
void print(char test_char[], Attribute& test_Attri);
```

10) check primary key 以及 unique

线性 check, 如果发现已经存在, paochu 中断。

```
void check(MyTable& temp_table, MyTuper& temp_tuper);
```

11) 对于 int 和 float 的比较函数, 利用模版

```
template<class T>  
int compare_int_float(WHERE tempwhere, T A, T B);
```

4.2.5 Index Manager

1) 创建索引

```
void CreateIndex(MyTable& table, string attr, string indexname);
```

2) B+树 search

```
void Search(char* value, MyTable& table, string& attr, string indexname,  
WHERE con, bool set);
```

3) 从文件读取 B+树

```
void BuildBTreeFromFile(string indexname);  
// indexname not have ".txt", just name
```

4) 从 B+树获取节点

```
PtrtoNode GetOneTNode(ifstream& inf, int flag);
```

5) DeleteALL 删除所有 B+树节点的接口

```
void DeleteAll(string indexname); // delete all data in this index
```

6) 删除索引

```
void DropIndex(string indexname); // delete this index
```

7) buffermanager 引用

```
void set_Buffer(BufferManager& abuffer) { this->buffer = &abuffer; }
```

8) B+树插入和删除

```
void InsertBtree(MyTable& table, My_place& data, string indexname);
```

9) print 函数, 输出元组

```
void print(char* test_char, Attribute& test_Attri);
```

10) primarykey 和 unique 如果存在索引的判读

```
bool IsRepeat(MyTable& table, string attr, char* value, string indexname);
```

11) 删除数据 set invalid

```
void set0(TupleData data, MyTable& table);
```

12) 从 indexname 获取 tablename

```
string gettablename(string indexname)
```

4.2.6 Buffer Manager

1) buffer 的写回到文件的接口, 在 buffermanager 析构函数调用的时候调用

```
void WriteBlock_to_File(int BufferNum);  
//Write the content of block back to file,and initialize the block
```

2) buffer 的写回到文件的接口, 在 buffermanager 析构函数调用的时候调用

```
void WriteBlock_to_File(int BufferNum);
```

3) 从 buffermanager 中找到对应文件的第几个 block 存在的 buffer 索引

```
int GetBufferNum(string filename, int BlockOffset);
```

4) 读取文件中第几块对应的信息

```
void ReadBlock(string filename, int BlockOffset, int BufferNum);
```

5) 读取文件中第几块对应的信息

```
void WriteBlock(int BufferNum);  
//We use it when the block is rewritten, and we symbolize it
```

6) 象征性的写回, 就是设置 iswritten, 但是不写回到文件

```
int GetEmptyBuffer();  
//Get the empty buffer block in memory
```

7) 从 buffer 中取出一个空的 buffer 来使用到文件

```
int GetEmptyBuffer();  
//Get the empty buffer block in memory
```

8) 从 **buffer** 中取出一个空的 **buffer** 来使用到文件，要求不与 **filename** 相同

```
int GetEmptyBufferExcept(string filename);  
//Get the empty buffer block in memory without substitute the  
designated file
```

9) 从 **buffer** 中找到一个可以插入元组的地方，供 **insert** 使用

```
InsertPosition GetInsertPosition(MyTable& table);  
//Return the available position of inserting data
```

10) 申请新的模块，内部调用，如果块数不够，则块++

```
int AddNewBlock(MyTable& table);  
//After inserting a new table file, return the updated index of block  
in memory
```

11) 已经存在 **buffer** 里面的 **index** 的获取

```
int Index_in_Buffer(string filename, int BlockOffset);  
//Get the index of one block in memory, return -1 if not find the block  
void ScanWholeFile(MyTable& table);
```

12) 扫描所有文件

```
void ScanWholeFile(MyTable& table);  
//Read the whole table file into memory(if the file is to large, it  
may cause collapse)
```

13) **buffer** 中 **tuple** 有效无效的设置

```
void SetInvalid(string filename);  
//Set the block of the file to invalid, used for deleting table and  
index
```

14) 重新分配 **block** 给当前的文件，以及对应的块

```
int AllocateBlock(string filename, int BlockOffset);  
//External function call interface
```

4.2.7 精妙的 database 设计

1) **My_place** 结构

这个是 **RecordManager** 插入的时候获取到的位置信息，传给 **IndexManager**，这是通过 **api** 实现的。

```
struct My_place {  
int buffer_num; // buffermanager  
int bolck_num;  
int tuple_offset;  
};
```

2) MyData 类

Flag 表示数据类型 (-2 表示 int, -1 表示 float, >=0 表示 char(n))

```
class MyData{
public:
    int Flag;
    virtual ~MyData() {}
};
```

3) MyInt 类

从 MyData 类派生, 包含 int 型的 value

```
class MyInt: public MyData{
public:
    int value;
    MyInt(int x):value(x){
        Flag = -2;
    };
    virtual ~MyInt() {}
};
```

4) MyFloat 类

从 MyData 类派生, 包含 float 型的 value

```
class MyFloat: public MyData{
public:
    float value;
    MyFloat(float x):value(x){
        Flag = -1;
    };
    virtual ~MyFloat() {}
};
```

5) MyChar 类

从 MyData 类派生, 包含 char(n)型的 value, 以 string 形式保存

```
class MyChar: public MyData{
public:
    string value;
    MyChar(string x):value(x){
        Flag = x.size(); //We use different Flag values to
represent different length
    };
    virtual ~MyChar() {}
};
```

6) Attribute 类

包含属性数, 属性类型, 属性名, 是否 unique

```
struct Attribute{
    int AttrNum; //Represent the number of attributes
    short Flag[MAXATTRIBUTE]; //Represent the flag of every attribute
    string AttributeName[MAXATTRIBUTE];
    //Represent the name of every attribute
```

```
bool IsUnique[MAXATTRIBUTE];
        //Determining whether every attribute is unique
};
```

7) Index 类

包含 index 数, 位置 (在 attribute 中的位置), 索引名

```
struct Index{
    int IndexNum;
    short Position[MAXINDEX];
        //Represent the location of the index in the attribute
    string IndexName[MAXINDEX];
};
```

8) MyTuper 类

一个包含每一个 Data 的容器

AddData 是 api 主要用到的一个函数, 插入一个 data 到 MyTuper 类里面

```
class MyTuper{
public:
    vector<MyData*> Data;

    void AddData(MyData* NewData)
    {
        Data.push_back(NewData);
    }
};
```

9) MyTable 类成员定义

包含块数,

```
class MyTable{
    //friend class CataManager;

public:
    int BlockNum;                //The number of blocks the table occupies
    string TableName;            //The name of the table
    Attribute attributes;        //Attributes in table
    vector<MyTuper*> tupers;      //Tupers in table
    Index indices;               //Indices in table
    int PrimaryLocation;         //The location of primary key, -1
                                //represent that there is no primary key
};
```

10) MyTable 获取一个 tuple 的 size, int 为 4 字节, float 为 4 字节

```
int OneTuperSize();
```

11) MyTable 添加 index 或者 drop index, 会根据有没有这个类型而 throw 中断

```
void SetIndex(short i, string indexname);
void DropIndex(string indexname);
```

12) MyTable 添加元组, 会根据 tuplesize 判断是否符合条件, 不符合条件则发出中断


```
void AddTuper(MyTuper* MT);
```

13) 中断处理机制

向上发出中断，**catch** 接受

```
class TableException: public exception{
public:
    TableException(string s):Message(s){}

    string DisplayMessage()
    {
        return Message;
    }

private:
    string Message;
};
```

4.3 操作演示

4.3.1 建立表

1) 正确示范:

代码:

```
create table test (
first int unique ,
second float ,
thired char(8) ,
primary key ( first )
);
```

```
MinSQL---->create table test (
---->first int unique ,
---->second float ,
---->thired char(8) ,
---->primary key ( first )
---->);
creat table success

input exit(end program) else continue
```

图 1 建表

2) 错误示范:

```
MinSQL---->create table test (  
---->first int unique ,  
---->second float ;  
wrong near ';' at row :2  
input exit(end program) else continue
```

图 2 假设代码不完整

3) 如果表存在:

```
MinSQL---->create table test (  
---->first int unique ,  
---->second float ,  
---->thired char(8) ,  
---->primary key ( first )  
---->);  
Table already Exsit  
input exit(end program) else continue
```

图 3 假设表存在

4.3.2 插入数据

1) 代码

```
insert into test values (0,99.9,'888888');  
insert into test values (123,3.14,'888888');  
insert into test values (133,3.14,'888888');  
insert into test values (143,3.14,'888888');
```

```

MinSQL---->insert into test values (0,99.9,'888888');
insert success

input exit(end program) else continue
MinSQL---->insert into test values (123,3.14,'888888');
insert success

input exit(end program) else continue
MinSQL---->insert into test values (133,3.14,'888888');
insert success

input exit(end program) else continue
MinSQL---->insert into test values (143,3.14,'888888');
insert success

input exit(end program) else continue
MinSQL---->

```

图 4 插入数据

2) 查询

```

select * from test where first = 0 ;
select * from test where first != 0 ;
select * from test where first > 123 ;

```

```

MinSQL---->select * from test where first = 0 ;
first      second      thired
0          99.9        '888888'

input exit(end program) else continue
MinSQL---->select * from test where first != 0 ;
first      second      thired
123        3.14        '888888'
133        3.14        '888888'
143        3.14        '888888'

input exit(end program) else continue
MinSQL---->select * from test where first > 123 ;
first      second      thired
133        3.14        '888888'
143        3.14        '888888'

input exit(end program) else continue

```

图 5 插入数据-查询

4.3.3 删除数据

1) 代码

```
delete from test where first= 133 ;
```

2) 查询

```
select * from test ;
```

```

MinSQL---->delete from test where first= 133 ;
delete from testwhere ... success
delete success

input exit(end program) else continue
s
MinSQL---->select * from test ;
something wrong befor where
input exit(end program) else continue
s
MinSQL---->select * from test;
first      second    thired
0          99.9      '888888'
123        3.14      '888888'
143        3.14      '888888'

```

图 6 删除-查询

4.3.4 创建索引

1) 创建代码

create index indext on test (second) ;

```

MinSQL---->create index indext on test ( second ) ;
no ;

input exit(end program) else continue
create index indext on test ( second )
MinSQL---->create index indext on test ( second ) ;
create index success

input exit(end program) else continue
s
MinSQL---->create index indext on test ( second ) ;
no ;

input exit(end program) else continue
create index indext on test ( second ) ;
MinSQL---->create index indext on test ( second ) ;
the index named indext exist ont the table named test

input exit(end program) else continue

```

图 7 创建索引 (判段索引存在)

4.3.5 删除表

```
MinSQL---->drop table table test ;
table not exist
input exit(end program) else continue
s
MinSQL---->drop table test ;
delete the table head file named test
delete the table head file named test
drop table success

input exit(end program) else continue
s
MinSQL---->select * from test ;
first      second      thired
Table may not exist
input exit(end program) else continue
```

图 8 删除表

4.3.6 删除索引

```
this is an empty B+Tree
creat table success

input exit(end program) else continue
s
MinSQL---->create index indexname on book ( bookprice );
This is an empty B+Tree
This is an empty B+Tree
This is an empty B+Tree
create index success

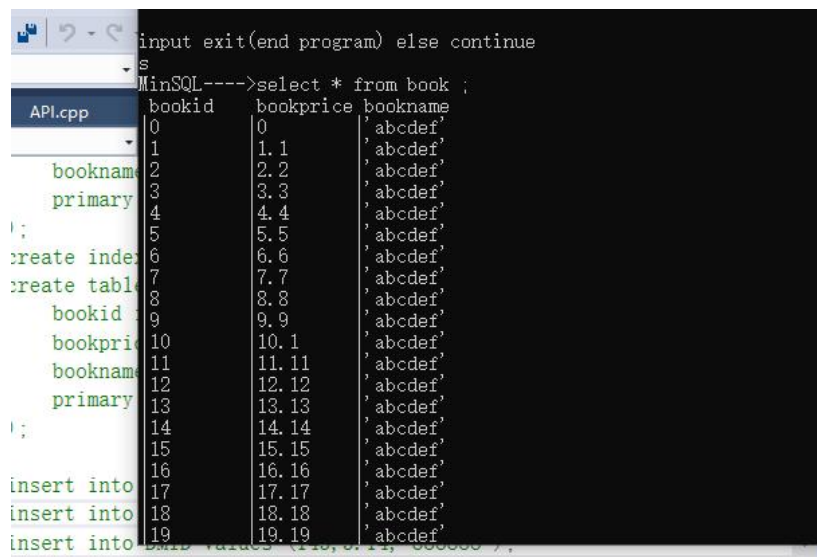
input exit(end program) else continue
s
MinSQL---->create index indexname on book ( bookname );
the index named indexname exist ont the table named book

input exit(end program) else continue
s
MinSQL---->drop index indexname ;
delete the index named indexname
drop index indexname success

input exit(end program) else continue
s
```

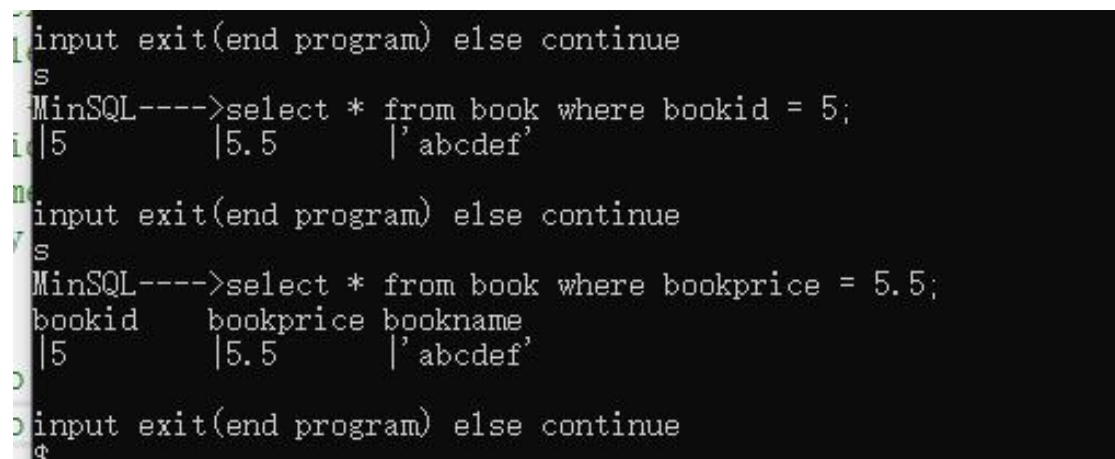
图 9 删除索引

4.3.7 查询



```
input exit(end program) else continue
s
MinSQL---->select * from book ;
bookid  bookprice bookname
0        0         'abcdef'
1        1.1       'abcdef'
2        2.2       'abcdef'
3        3.3       'abcdef'
4        4.4       'abcdef'
5        5.5       'abcdef'
6        6.6       'abcdef'
7        7.7       'abcdef'
8        8.8       'abcdef'
9        9.9       'abcdef'
10       10.1      'abcdef'
11       11.11     'abcdef'
12       12.12     'abcdef'
13       13.13     'abcdef'
14       14.14     'abcdef'
15       15.15     'abcdef'
16       16.16     'abcdef'
17       17.17     'abcdef'
18       18.18     'abcdef'
19       19.19     'abcdef'
```

图 10 不带条件的查询




```
input exit(end program) else continue
s
MinSQL---->select * from book where bookid = 5;
5        5.5       'abcdef'

input exit(end program) else continue
s
MinSQL---->select * from book where bookprice = 5.5;
bookid  bookprice bookname
5        5.5       'abcdef'
```

图 11 带条件的查询

4.3.8 离开



```
MinSQL---->s
wrong sql!(delete / selete * / insert /create / drop)

Maininput exit(end program) else continue
exit

C:\Users\test\source\repos\DBM_past\Debug\DBM.exe (进程 17476)已退出, 返回代码为: 0。
若要在调试停止时自动关闭控制台, 请启用“工具”->“选项”->“调试”->“调试停止时自动关闭控制台”
按任意键关闭此窗口...
```

图 12 exit

5.分工与心得

5.1 分工

刘振东 (API, Interpreter, RecordManager, CatalogManager)

任浩然 (B+Tree, IndexManager)

秦兆祥 (Database, BufferManager)

调试: 全部

5.2 心得

这次大程序，本文作者本来以为本小组做不出来的，最后做出来确实感觉非常开心。团队里面缺一不可。

首先是秦兆祥同学写的 `database` 和 `BufferManager` 模块，给我们奠定了基础，如果没有这两个东西，则我们后续的工作根本无法展开。其中 `database` 模块写的非常秒，`BufferManager` 的设计也让我对于数据库中文件存储的方式有了更加深刻的理解。

然后是任浩然同学写的 B+树模块，降低了插入以及查询的时间复杂度， N 次操作，从原来的 $O(N^2)$ 降低到了现在的 $O(N \log N)$ ，插入 10000 条数据的操作从原来的 1min 以上降低到了现在的 20s。

最后是本人的工作，API、Interpreter、CatalogManager、RecordManager。这几个模块纯属搬砖的模块，就基本是设计上没有很精妙的地方，但是对于整个 MiniSQL 来说，确实是比较重要的一部分，Interpreter 将 SQL 语句翻译为机器可以识别的代码，调用 API，API 实现各个语句的不同处理方式，决定是调用 Index 还是 Catalog 还是 Record。Catalog 包含了表的定义，包含索引定义等等的信息。Record 包含数据的处理等等一些列的操作。

做完之后，感觉自己的能力得到了很大的提升，但同时本人觉得底层的实现是非常的精妙的。这也是我需要学习的地方。