浙江大学

设计项目 MiniSQL

组长： 沈锴

组员： 蒋晨恺

赵宣栋

指导教师 孙建伶

年级与专业 大二　计算机科学与技术

所在学院 计算机科学与技术学院

目录:

一、 引言

1.1 项目名称

MiniSQL 数据库系统设计与实现。

1.2 实验目的

设计并实现一个精简型单用户 SQL 引擎(DBMS)MiniSQL（Mini Structural Query Language Engine），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。

1.3 实验需求

1.3.1 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 <= n <= 255。

1.3.2 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;支持单属性的主键定义。

1.3.3 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)。

1.3.4 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。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。

1.3.5 功能命令

在输入端口，一条 SQL 语句可以为一行或多行，以分号作为结束标注，命令中的关键字 均为小写。具体而言，需要实现下面的功能命令:

1. 创建表: create table t (tid int, tname char(8));
2. 删除表: drop table t;
3. 创建索引: create index T1 on t (tid);
4. 删除索引: drop index T1 on t;

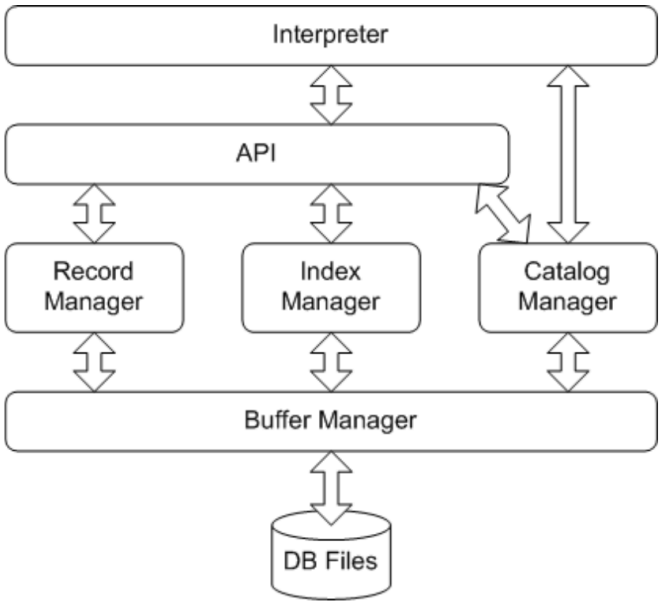
5. 选择查询: select tid from t where tname…;
6. 插入记录语句: insert into t values (…);
7. 删除记录语句: delete from t where …;
8. 提出系统: quit;
9. 执行文件: execfile …;

需要设计 buffer 模块唯一访问数据文件，负责数据块的写入和写出，记录块状态并实现缓冲区的替换算法。数据文件由一个或多个数据块组成，块大小与缓冲区大小相同。一个数据块包含一条至多条记录，支持定长的记录，不要求支持记录的跨块储存。一个数据块能存储多少条记录可以根据块大小和存储的每条记录的长度计算出来。

索引文件中记录数据的 B+树索引信息，B+树中节点大小应与缓冲区块大小相同;B+树中节点中的数据按照顺序存储;B+树的叉数由节点与索引键大小计算得到。

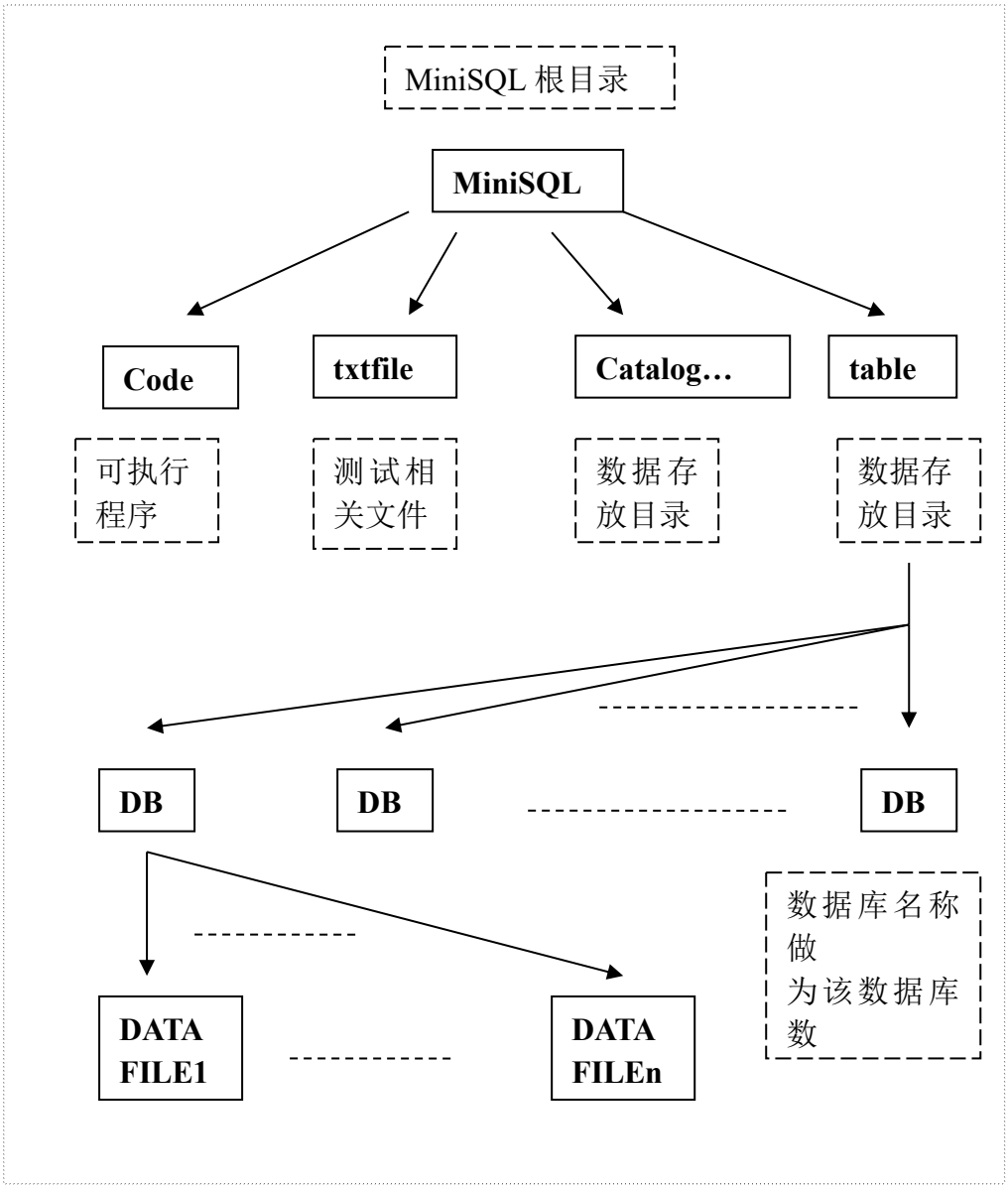
二、 总体框架

2.1 系统体系结构



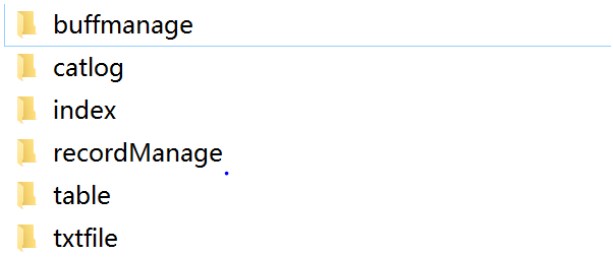
图表 1 MiniSQL 总体框架

2.2 系统文件结构



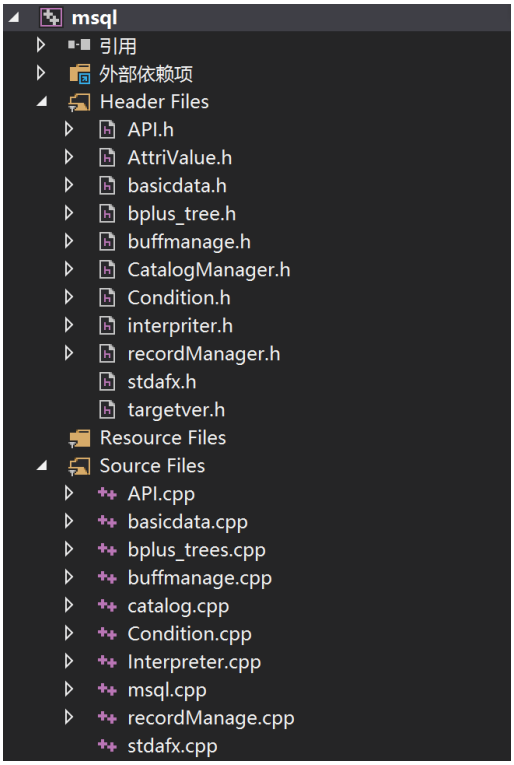
图表 2 系统文件结构

Index 目录下存放索引文件，recordmanage 目录下存放记录文件，talbe 目录下存放表的文件。Txtfile 目录下存放测试所用文件。



图表 3 文件存储

2.3 程序清单



图表 4 程序清单

2.4 模块概述

2.4.1 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。

2.4.2 API（application programming interface）

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。

2.4.3 CatalogManager

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。

2.4.4 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。

2.4.5 IndexManager

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。

2.4.6 BufferManager

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
4. 提供缓冲区页的 pin 功能，及锁定缓冲区的页，不允许替换出去

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。

2.4.7 DB Files

DB Files 指构成数据库的所有数据文件，主要由记录数据文件、索引数据文件和 Catalog 数据文件组成。

三、 通用类设计

3.1 基础数据结构

在 DBMS 中，表的表示是最为重要的。通常在关系数据库中，数据被定义成表的形式进行存储。表的构成有很多要素，如数据类型，表的属性名称，表的名称，表的属性数量，和表相关的 B+树信息等等。

为了能够设计一个表类，我们首先定义如下的最为基础的数据结构：表的属性。

3.1.1 表属性类

表的属性被设定被一个对象，对象的定义如下所示。

```
struct AttriValue
{
    bool isPrimary = 0, isUnique = 0
    int cntintable;
    int length, type;
    std::string AtrributeName;
    std::string strVal;
    int intVal;
    float FloatVal;
};
```

AttributeName 表示属性的名称，Type 表示属性的数据类型。isPrimary 表示该属性是否为主键，isUnique 表示该属性是否为单值。cntintable 表示该属性在表中的序号。Length 表示该属性的数据占有的字节数。

由于 AttriValue 对象既是表的定义的一部分，也是表的数据的一部分。因此该对象中可以存储具体的值。对于 int 数据，则存储在 intVal 中，float 数据则存储于 FloatVal 中，char 数据则存储于 strVal 中。

特别的，对于数据类型，我们定义了以下枚举类型：

```
enum keytype { isint, isfloat, ischar };
```

3.1.2 索引数据类

```
class IndexNode
{
public:
    std::string  IndexName;
    bool valid;
    btree*tree;
    IndexNode() :IndexName(""), valid(0), tree(nullptr) {}

};
```

IndexName 用来表示索引的名称，valid 用来表示索引是否被删除，tree 是

指向具体的 B+树的指针。

3.1.3 缓存区缓存块类

```
struct Block {
    std::string filename;//block's filename in buff
    int type;//0 int 1 float 2 char 3 unknown
    int blockoffset;//the location of block in filename
    char*cblock;//the massage in Block
    bool isdirty;//if it is changed or not
    bool pin;
    int charnum;
    Block*next;
};
```

缓存块用来封装 4k 大小的内存区。Filename 属性为该缓存块所归属的文件名。Type 是缓存块数据类型。Blockoffset 是缓存块在文件中的偏移量。Cblock 是指向 4k 大小的缓存区的指针。Isdirty 标注缓存块是否被修改。Pin 标注缓存块是否被锁定。Charnum 用来标记有效字节数。Next 指针是指向下一个缓存块的指针。

缓存区负责管理文件。文件也被定义成了对象。缓存区需要负责维护每一个文件的最大块偏移量和之前被删除的块的偏移量。

3.1.4 文件类

文件对象定义如下：

```
class File
{
public:
    std::vector<int> emptyBlocks;
    int lastEmptyBlock;
    std::string fileName;
    fileType type;
    File* next;
};
```

EmptyBlocks 用来存储空块的偏移量，lastEmptyBlock 用来维护最大的块偏移量。fileName 用来存储文件名，type 用来表示文件的类型。

这里的文件类型不同于之间的数据类型。我们定义如下的文件枚举类型：

```
enum type { index, record, uncertain };
```

3.1.5 条件类

条件类型。

在 SQL 语句中，常常会用到条件语句。如 select * from A where name<=’name123’ ;这一句中 where 后面的就是条件语句。我们将每一个条件抽象成对应的对象进行封装。对象定义如下所示：

```
class Condition
{
```

```
public:
    enum Operator {
        OPERATOR_EQUAL, OPERATOR_NOT_EQUAL, OPERATOR_LST, OPERATOR_MOT,
        OPERATOR_LESS_EQUAL, OPERATOR_MORE_EQUAL
    };


    std::string attriName;
    std::string value;
    int OP;
};
```

attriName 为条件中的属性名。Value 为具体的值。OP 为条件。条件被定义成了专门的类型，也就是 Operator 枚举类型。分别有等值、不等值、小于、大于、小于等于、大于等于。

3.1.6 缓存块的记录类

在 Minisql 中，我们需要对缓存块中的记录进行定位。因此我们定义了如下的数据结构。

```
class position
{
public:
    int blocknum;
    int offset;
};
```

Blocknum 为缓存块在文件中的偏移量，offset 是记录在缓存快中的位置。

3.1.7 不同数据类型的存储说明

在设计 Minisql 的时候，我们需要处理三种数据类型。Char 型数据类型我们通过 string 类型进行封装。为了统一起见，对于 string 和 int 也用 string 进行封装。但是不同的是，int 和 float 并不是等长的。如果通过 stringstream 转换成 string 按照字符串进行存储的话，会带来一系列缺陷：

1. 记录的长度不定。比如说 float 支持最大 3.4e+38 的数字的表示。因此不定的记录会给存储带来不便。

2. 效率低下。如果用 string 进行存储，那么每一次需要读取的字符串可能会很长。

因此我们采取存储机器码的方式。Int 和 float 实际上都可以用 32 位 4 字节进行存储。我们只要将数字转换成机器码，将机器码存储在 string 对象中，就实现了统一和效率这两个问题。

但与此同时会带来一系列代价，比如说转换麻烦容易出错等等。

具体的转换可以通过以下函数：

```
std::string RecordManager::changekey(const std::string&keyin, int type)
{
    char buf[5] = { 0 };
```

```
    string ans;
    if (type == 0)
    {
        int val = stoi(keyin);
        *reinterpret_cast<int*>(buf) = val;
        ans.resize(4);
        ans[0] = buf[0];
        ans[1] = buf[1];
        ans[2] = buf[2];
        ans[3] = buf[3];
    }
    else if (type == 1)
    {
        float val = stof(keyin);
        *reinterpret_cast<float*>(buf) = val;
        ans.resize(4);
        ans[0] = buf[0];
        ans[1] = buf[1];
        ans[2] = buf[2];
        ans[3] = buf[3];
    }
    else {
        ans = keyin;
    }
    return ans;
}
```

3.2 Table类

```
class TableNode
{
public:
    std::string TableName;
    std::list<AttriValue> AttributeList;
    std::vector<std::string> PrimaryKey;
    std::vector<std::string> Unique;
    //std::map<std::string, std::pair<btree*, bool>>bplustrees;
    std::map<std::string, IndexNode>bplustrees;
    int RecordNum;
};
```

TableNode 存储着表的元信息。TableName 是表的名称，AttributeList 是一个表的属性的链表，PrimaryKey 存储着主键的信息。Unique 存储着不可重复的

键的信息。RecordNum 存储着表的记录数。

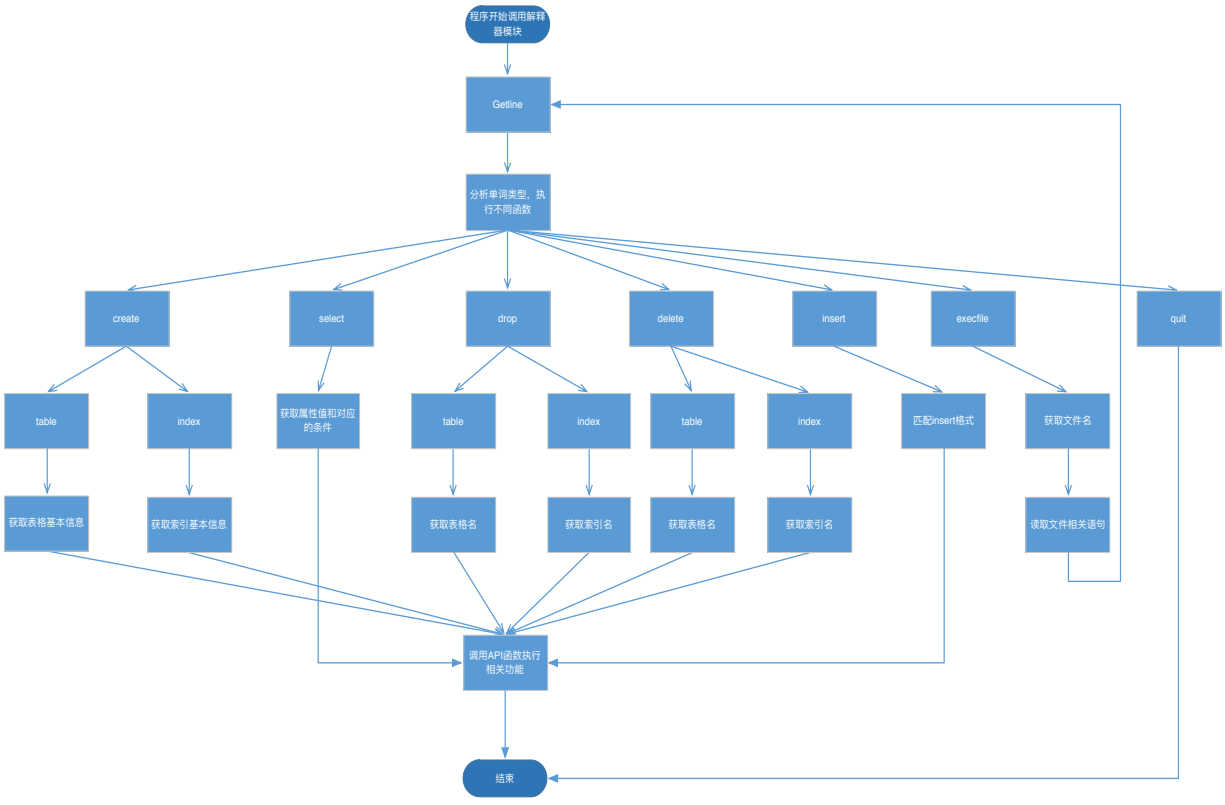
Bplustrees 为一个红黑树，存储着跟表相关的索引信息。第一个参数是建立 B+树的属性名，第二个参数是对应的对应的索引对象。

四、 Interpreter 模块

4.1 模块概述

Interpreter 模块直接与用户交互，主要负责接收并解释用户在前端输入的 SQL 命令，识别命令的类型，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，接收返回值。对不正确的命令，例如表的重定义，使用不存在的属性，数据类型不匹配等问题显示错误信息，若操作成功，也会进行提示。

4.2 设计思路



图表 5 Interpreter 设计流程图

当用户打开程序时，用户在交互界上面输入 SQL 语句，然后通过分号进行隔断，对于每一条语句，交给 Interpreter 部分进行处理。

Interpreter 将整个语句（string）送入 getword 函数，getword 将关键字进行切分，根据不同的关键字，运行不同的函数，并且传递给 API 进行下一步的处理。

Interpreter 还会接收 execfile, quit 等命令，分别执行读取文件和退出的操作。对于读取文件，Interpreter 打开文件，逐行读取并运行 interpreter 函数进行处理。

4.3 Interpreter接口设计

```
class Interpreter {
public:
    API myapi;
    std::string filename;   //use the filename to open the file
    std::string word;       //used to analysis each word in the sentence
    int interpreter(std::string s);                //analysis sentence
    std::string getword(std::string s, int &num);   //fetch the word from
sentence
    bool exec_create(std::string s, int &num);      //do the create function
    bool exec_select(std::string s, int &num);      //do the select function
    bool exec_drop(std::string s, int &num);    //do the drop function
    bool exec_delete(std::string s, int &num);      //do the delete function
    bool exec_insert(std::string s, int &num);      //do the insert function
    Interpreter() {}        //the constructor of Interpreter class
    ~Interpreter() {}       //the destructor of Interpreter class
};
class SyntaxException {};   //exception handling
```

其中 Interpreter 调用 API 类，执行 API 中的函数。Interpreter 包含文件名，单词的数据成员，也包括 `bool exec_create(std::string s, int &num);`等成员函数。

其中，我们着重说明 select 和 delete 的参数传递机制。在解释器中，首先对 select 的基础元素进行解释，抽离出表名，属性名，和 where 后的条件。对于 where 后 and 连接的条件，我们使用自定义结构 attribute 组成的向量表示，同时作为传递的参数.

程序主要通过 `int interpreter(std::string s);`对输入的一串字符进行处理，然后根据关键字的不同分别调用不同的执行函数。在执行过程中，执行函数调用 `std::string getword(std::string s, int &num);`获得下一个单词的字符串变量

4.4 语法功能说明

用户进入 MiniSQL 程序之后可以由用户输入相应的 SQL 命令，可以分行输入，但注意关键词必须正确输入，并以分号结尾。下面就各个语句进行语法功能说明。

4.4.1 创建表

创建表的格式如下:

create table 表名（
列名 类型，
列名 类型，
……
列名 类型，
primary key(列名)
）；

创建表实例：

```
>> create table student2(
id int,
name char(12) unique,
score float,
primary key(id)
);
Interpreter: successful create!
```

若该语句执行成功，则输出执行成功信息；若失败，则告诉用户失败的原因。

```
>> create table (a int);
Sysntax Error! Unknown data type
Interpreter: create failed!
```

——因为没有定义表的名字，输出错误信息

```
>> create table a;
Sysntax Error! Unknown data type
Interpreter: create failed!
```

——没有定义表的具体属性，输出错误信息

4.4.2 删除表语句：

Drop table 表名;

删除表实例：

```
>> drop table student2;
Interpreter: successful drop!
```

删除表成功。

```
>> drop table student;
The table you delete does not exist!
Interpreter: drop failed!
```

——如果已经删除，则再次删除时找不到原来的表

```
>> drop index aaa;
The Index you delete does not exist!
Interpreter: drop failed!
```

——如果删除一个不存在的表，输出错误信息

4.4.3 创建索引

Create index 索引名 on 表名 (列名);

```
>> create index stuidx on student2 ( name );
Interpreter: successful create!
```

若该语句执行成功，则输出执行成功信息

```
>> create index stuidx on student2 ( name );
The index you create does already exist!
Interpreter: create failed!
```

因为已经创建过了，输出重复创建的错误信息

4.4.4 删除索引语句

Drop index 索引名;

```
>> drop index stuidx;
Interpreter: successful drop!
```

若该语句执行成功，则输出执行成功信息;

```
>> drop index studdd;
The Index you delete does not exist!
Interpreter: drop failed!
```

如果索引不存在，输出错误信息;

4.4.5 选择语句

该语句的语法如下:
Select * from 表名;
Select * from 表名 where 条件;
条件可以是 列 op 值 and 列 op 值 …… and 列 op 值
Op 可以是算术比较符: = <> != > < >= <=

```
>> select * from student2 where name='name245';
NO tree
1080100245      name245         62.5
Number of tuples selected: 1
Time: 0.006
Interpreter: successful select!
```

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，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;

```
>> select * from student;
The table you select does not exist!
Interpreter: select failed!
```

如果查询的表格不存在，则输出错误信息提示；

4.4.6 插入记录语句

语句的语法如下：

Insert into 表名 values (值 1，值 2，值 3，•••••••，值 n)；

```
>> insert into student2 values(1080197996,'name97996',100);
Insert
```

若该语句执行成功，则输出执行成功信息 insert

```
>> insert into student2 values(1080100245,'name245',100);
Duplicate entry for key 'PRIMARY'
Interpreter: insert failed!
```

因为 name245 的记录已经存在，再次插入会造成主键重复的错误

4.4.7 删除记录语句

语句语法

Delete from 表名；

Delete from 表名 where 条件；

```
>> delete from student2 where name='name97996';
Number of tuples deleted: 1
Interpreter: successful delete!
```

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；

```
>> delete from student;
The table you delete record does not exist!
Interpreter: delete failed!
```

如果要删除的记录并不存在，则输出不存在的错误信息。

4.4.8 退出 miniSQL 程序

直接输入 Quit 命令，程序关闭。

4.4.9 执行 SQL 脚本文件语句

语句语法：execfile 文件名；

```
>> execfile instruction10.txt;
C://zxd_CS//file//txtfile//instruction10.txt
Insert
Insert
Insert
Insert
Insert
Insert
```

如果文件存在，依据文件中的内容依次执行。

```
>> execfile a.txt;
C://zxd_CS//file//txtfile//a.txt
Can not open the file
```

如果文件不存在，输出错误信息。

五、 API 模块

5.1 模块概述

API 模块负责封装底层的接口给 Interpreter 使用。Minisql 的数据处理是 recordmanager 负责的，表的创建与维护是 catalogmanager 负责的。为了提供一个统一的接口，需要 API 对这些接口进行封装。因此 API 模块具有承上启下的作用，在 Interpreter 模块和底层模块之间负责调整。

在整个系统中，API 承接来自 Interpreter 模块的数据请求，比如建表、删除表、建索引、删索引等等请求。然后 API 会根据得到的请求来调用底层的模块，如调用 Catalog 模块处理建表删表请求，调用 RecordManager 处理插入元组删除元组的请求等等。也就是数据从 Interpreter 从上而下流向底层。然后底层模块会讲操作信息返回，比如删除成功返回 true。API 得到信息会将信息处理。或者返回到 API 模块或者将错误信息输出。于是信息又流通到底层。

5.2 设计思路

缓存区利用的局部性原理。经过统计我们发现，当一个数据块被访问的时候，其附近的数据块很有可能也会被访问，同时这个数据块在将来很有可能被再次访问到。这就是时间局部性和空间局部性。

利用这个局部性原理，我们可以一次读写读入一个 Block 而不是特定的多少 Bytes。这个在操作系统层面就已经为我们实现了。我们需要完成的是如何去管理这些 Block。通常的这个 Block 大小是 4K。

我们采用的是全相连的方式。理由是内存的搜索是比较快的，这种实现代价最小。

当缓存区已经满的时候，我们需要替换。替换的策略采取的是 LRU。根据统计学发现，最近最少使用的块是最不可能再次被访问到的。

5.3 API函数接口

```
class API {
public:
    bool createtable(const std::string&tablename,
std::vector<AttriValue>*valuein);
    bool droptable(const std::string&tablename);
    bool createindex(const std::string&indexname, const std::string&tablename,
const std::string&atrrubename);
    bool dropindex(const std::string&indexname);
    bool insertrecord(const std::string&tablename,
std::vector<std::string>*info);
    bool deleterecord(const std::string&tablename,
std::vector<Condition>*Conditions);
    bool selectrecord(const std::string&tablename,
std::vector<std::string>*selectinfo, std::vector<Condition>*condition);
private:
    RecordManager mr;
    CatalogManager mc;

};
```

5.4 API接口设计说明

1. 创建表：建立表的时候，需要调用 CatalogManager 查询表是否存在。如果可以插入，则调用 CatalogManager 调用相应的创表函数。这时会建立相应的表并且对必要的键建好 B+搜索树。

2. 删除表：删除表的时候，需要调用 CatalogManager 查询表是否存在。如果可以删除，则调用 CatalogManager 删除表，并且调用 RecordManager 删除对应的空块信息。

3. 元组的插入：插入时，需要调用 CatalogManager 查询表是否存在。如果存在直接调用 RecordManager 的插入函数即可。RecordManager 会处理所有的插入请求。然后将插入结果返回即可。

4. 元组的删除：删除时，需要调用 CatalogManager 查询表是否存在。如果存在直接调用 RecordManager 的删除即可。RecordManager 会处理所有的删除请求。然后将删除信息返回即可。

5. 数据查找：数据查找需要调用 CatalogManager 查询表是否存在。如果存在直接调用 RecordManager 的查询函数即可。RecordManager 会处理查询请求并且对查询结果进行输出。

6. 创建索引和删除索引：同样的需要先调用 CatalogManager 查询表是否存在。如果存在直接调用 CatalogManager 的建立索引和删除索引函数即可。然后将信息返回 Interpreter 模块。

六、 Catalog 模块

6.1 模块概述

Catalog 模块负责管理数据库的所有模式信息，包括表的定义信息，属性的定义信息，索引的定义信息。当解释器读到相应的命令需要更改模式信息时，将参数传递到 API，API 会调用 catalog 中相应的函数进行操作，这些功能函数会调用 bplustree 模块在属性上建立 B+树，并且将索引的信息存储到文件中。

我们在建表的时候就将有 unique 限制和 primarykey 限制的属性建立索引，然后在之后的操作中，如果有建立索引的操作，则将 indexnode 的 valid 位置为 true。

6.2 设计思路

其他模块通过调用 Catalog，会获得数据库中的模式信息，所以需要通过一定的数据结构存储这种模式信息，包括属性的定义，索引的定义。

我们主要设计了 IndexNode、TableNode 以及 CatalogManager 三个类，分别存储索引，表和整个 catalog 的信息。表中包含一个或多个索引，catalog 中包含一个或多个表，所以是一个嵌套的设计。

6.2.1 IndexNode 类的定义

```
extern BufferManager m;
class IndexNode
{
public:
    std::string IndexName;      //store the name of index
    bool valid;                //store the status of the index
    btree*tree;                //a pointer to b_plus tree
    IndexNode() :IndexName(""), valid(0), tree(nullptr) {}
                                //constructor
};
```

6.2.2 TableNode 类的定义

```
class TableNode
{
public:
    std::string TableName;                    //store the name of table
    std::list<AttriValue> AttributeList;       //store the attribute
    std::vector<std::string> PrimaryKey;       //store the primarykey
    std::vector<std::string> Unique;           //store the unique
    std::map<std::string, IndexNode>bplustrees;
```

```
    //the correspondence of attribute and btree
    int RecordNum;                              //store the number of record
    btree*gettree(const std::string&AttributeName)//get the btree without
limitation
    btree*gettreeop(const std::string&attributename)//get the btree if it is
valid
    btree*gettreebyname(const std::string&Indexname)//get the btree by its name
    bool setIndexName(const std::string&Attributename, const
    std::string&Indexname);
    //create a bplus tree
    bool setinvalid(const std::string&Attributename, const
std::string&Indexname);
     //delete a bplus tree
    void inserttree(const std::string&AttributeName, btree* tree);
    //insert a bplus tree
    TableNode() :RecordNum(0) {}
    //constructor with recordnumber
    TableNode(std::string table_name, std::list<AttriValue> Attribute_list, int
record_num) :
        TableName(table_name), AttributeList(Attribute_list),
RecordNum(record_num) {}

    //constructor with different value
};
```

6.3 Catalog接口设计

```
class CatalogManager
{
private:
    std::list<TableNode> TableList; //store all the tables
    void ReadCatalog();         //read the catalog
    void WriteCatalog();        //write information in catalog
public:
    //1 for success, 0 for fail, -1 for not exist
    bool CreateTable(std::string table_name, std::list<AttriValue>
Attributelist);     //create a index on a table
    bool CreateIndex(TableNode&a, std::string table_name, std::string
attributename); //create a table
    bool ifTableExist(std::string table_name);          //return if the table
exist
```

```
    bool ifIndexExist(std::string index_name);          //return if the index
exist
    int getRecordNum(std::string table_name);           //return number of
record
    bool deleteTable(std::string table_name);           //delete a table
    bool deleteIndex(std::string index_name);           //delete a index
    bool recordDelete(std::string table_name, int num); //delete a record
    bool recordAdd(std::string table_name, int num);//add a record to  catalog
    TableNode& getTableInfo(std::string table_name);//get information about
table
    CatalogManager();                                   //constructor
    ~CatalogManager();                                 //destructor
};
```

Catalog 模块提供多个接口:CreateTable, CreateIndex 可以创建 table 和 index 的模式信息。

IfTableExist, ifIndexExist 提供便利的查询信息是否存在的接口。

Getrecordnumber 可以便利的获得所有表格中记录的数量，deleteindex 是一种伪删除，仅仅将属性对应的 valid 位置为 false，便于之后的再次使用。

deleteTable 则是将与这个表对应的文件删除。Recorddelete 和 recordadd 是对于 TableNode 的修改，修改对于的 table 的值。Gettalbeinfo 则是返回一个 table 的概要信息。

七、 Record Manager 模块

7.1 模块概述

Record Manager(以下简称 RM)是和 Index Manager、Catalog Manager 并列的管理数据文件的某块。它向上为 API 提供与数据记录相关的函数接口，向下调用 Buffer Manager 的函数接口实现数据记录的读写。

每当要往数据库的表格中插入记录、查询记录、删除记录时，就需要调用 RM. RM 首先从 Catalog Manager 处获取当前数据库中表格信息，调用 Buffer Manager 将硬盘上的文件数据读入 Buffer 中，然后按照操作的需求对于这些记录进行查找、判断或是删除。

主要功能如下：

1) 记录插入

插入记录时，先判断所插入的记录信息中是否有主键或是其他 unique 的属性，如果有则先进行查重操作，若是所查信息在表中已经存在，则进行回滚，不插入。如果满足插入条件，则向 Buffer Manager 索取一个可以写记录的块，将记录写进 Buffer.

2) 记录查找

查找分为普通查找和带索引的查找。如果所查记录中包含主键信息，则默认进行索引查找。对于含有 unique 信息的记录，只有在建立索引之后才会用索引查找，否则为普通查找。

对于所有属性均不是 unique 的记录采用遍历表格的方式查找。

3) 记录删除

先通过查找操作找到所在记录，执行删除。

7.2 设计思路

7.2.1 记录存储

每次插入一条可行的记录都需要把记录存储到一个恰当的地方。在没有进行过删除操作的表格中，默认是按顺序进行存储，一个块存满之后再获取一个空块，Block 的块号依次递增。但是考虑到实际情况会涉及删除操作，这样会使原先充满的一些块留出一些空余。为了节省存储空间以及减少寻找可存储块的时间，我们将所有未满块的块号存在一个队列（queue）中，每次需要存储时，先判断队列是否为空，若不空则弹出一个可存储块；否则，RM 向 Buffer Manager 索取一个新的空块。

7.2.2 块内存储

由于所设计的属性均为定长，因此可以预先计算出整条记录的长度，当块的大小固定（4096）时，就可以算出一个块存储的最大记录数。实际存储的时候，为了方便之后的插入与查找操作，我们空出表格文件的第零块，用于记录表格的元信息。对 RM 而言，只把当前表格中的所有记录数目写在第零块块头 4 位。从第一块开始，块头 4 位记录这一块中的记录数，往后则是一个固定的形式：1 位的有效位，用于判断其后的记录是否有效；L 位长的一条记录，其中 L 为该记录的总长度，等于每个属性长度之和。每个属性存储所占位数在一开始创建表格时定义好。

7.2.3 数据存储

考虑到整形和浮点型数据长度不一，存储字符串类型效率低，我们采取其数值的机器码形式存储。字符串类型则直接使用字符串形式存储。

7.2.4 AttriValue 结构

我们为查找操作中选择的属性定义了一个 AttriValue 类，增加 RM 对外接口的抽象度，方便上层模块调用。该类对象中包含了属性的一些元信息，如是否为主键、是否为 unique 等。

```
struct AttriValue
{
    bool isPrimary = 0, isUnique = 0, isNull = 0;
    int cntintable;
    int length, type;
    std::string AtrributeName;
    std::string strVal;
    int intVal;
```

```
    float FloatVal;
    bool operator ==(const AttriValue &f)
    {
        if (isPrimary != f.isPrimary || isUnique != f.isUnique || isNull !=
f.isNull || AtrributeName != f.AtrributeName)
        {
            return false;
        }
        if (isNull)
        {
            return true;
        }
        if (type == 0)
        {
            return intVal == f.intVal;
        }
        if (type == 1)
        {
            return FloatVal == f.FloatVal;
        }
        if (type == 2)
        {
            return strVal == f.strVal;
        }
    }
```

7.2.5 Condition 类

我们为选择操作的条件专门定义了一个类 Condition，将这个类进行封装是考虑到判断条件较多且涉及了 int/float/string 三个类型的数据的条件判断，将所有判断写在 Condition.cpp 里可以降低程序的复杂度，提高可读性。

Condition.h 中定义了所有比较操作的枚举类型，给出了对于 int/float/string 三个类型数据判断函数。

```
class Condition
{
public:
    enum Operator {
        OPERATOR_EQUAL, OPERATOR_NOT_EQUAL, OPERATOR_LST, OPERATOR_MOT,
        OPERATOR_LESS_EQUAL, OPERATOR_MORE_EQUAL
    };
    std::string attriName;
    std::string value;
```

```
    int OP;
    Condition(void);
    Condition(std::string attr, std::string val, int op);
    bool check(int Int);
    bool check(float Float);
    bool check(std::string Str);
};
```

下面给出 Condition.cpp 中对于 float 类型的比较函数作为例子

```
bool Condition::check(float Float)
{

    float thisFloat;
    char buf[6];
    buf[0] = value[0];
    buf[1] = value[1];
    buf[2] = value[2];
    buf[3] = value[3];
    thisFloat = *reinterpret_cast<float*>(buf);

    switch (OP)
    {
    case OPERATOR_EQUAL:
        return Float == thisFloat;
    case OPERATOR_NOT_EQUAL:
        return Float != thisFloat;
    case OPERATOR_LST:
        return Float < thisFloat;
    case OPERATOR_MOT:
        return Float > thisFloat;
    case OPERATOR_LESS_EQUAL:
        return Float <= thisFloat;
    case OPERATOR_MORE_EQUAL:
        return Float >= thisFloat;
    default:
        return true;
    }
}
```

7.3 接口设计（接口功能参见注释）

7.3.1 外部接口

```
/*RM构造函数*/
RecordManager();
/*RM析构函数*/
~RecordManager();
/*删除记录*/
bool deletemap(TableNode &tablein);
/*遍历查找记录，返回查找到的记录数*/
int Select(TableNode &tableIn, std::vector<AttriValue>*AttrSelected,
std::vector<Condition>*ConditionApplied);
/*插入记录*/
bool Insert(TableNode &tableIn, std::vector<AttriValue>*ValueToInsert);
/*选择查找方式，B+树与遍历相结合*/
bool selectop(TableNode& tablein, std::vector<AttriValue>*AttrSelected,
std::vector<Condition>*ConditionAppied);
/*删除记录，返回被删除的记录数*/
int Delete(TableNode &tableIn, std::vector<Condition>*ConditionApplied);
```

7.3.2 内部接口

```
/*把属性值转换成机器码*/
std::string changekey(const std::string&keyin, int type);
/*返回属性的机器码*/
std::string getkey(char*record, int offset, int length);
/*判断一条记录是否符合条件*/
bool ConditionSatisfied(TableNode &tableIn, char* recordBegin,
std::vector<Condition>* ConditionApplied);
/*在索引中删除*/
void DeleteInBTree(TableNode &tableIn, char* attriValBegin);
/*在每个块中查找记录*/
int SelectRecordEachBlock(TableNode &tableIn,
std::vector<AttriValue>*AttrSelected, std::vector<Condition>*ConditionApplied,
pointer block, int rnum, int op);
/*返回记录总长度*/
int GetRecordSize(TableNode &tableIn);
/*返回属性类别*/
int GetTypeSize(int type);
/*判断记录具体内容是否符合条件*/
bool ContentSatisfied(char* content, int type, Condition* condition);
```

```
/*打印一条记录*/
void PrintARecord(TableNode &tableIn, char* recordBegin,
std::vector<AttriValue>* AttrSelected);
/*打印记录内容*/
void PrintContent(char * content, int type);
/*返回属性值的字符串形式*/
std::string GetKeyString(std::vector<AttriValue>*ValueToInsert, int n);
/*删除一条记录*/
bool DeleteARecord(TableNode &tableIn, char* validBit);
/*更新记录数*/
bool UpdateRecordNum(TableNode &tableIn, pointer block, int delete_amount);
/*在新的空块中插入记录*/
bool InsertInNullBlock(TableNode &tableIn,
std::vector<AttriValue>*ValueToInsert);
/*更新B+树*/
void UpdateBTree(TableNode &tableIn, std::string attrName, std::string attrValue,
int blockOffset, int innerOffset);
/*返回表格目录名全称*/
std::string GetTableName(std::string tableName);
/*表格名与空块队列的映射*/
typedef std::map<std::string, std::queue<int> >Mymap;
Mymap mymap;
```

八、 Index 模块

8.1 模块概述

在数据库中，常常需要查找某些 entry。如果对记录进行顺序查找，其代价是非常大的。B+树是一种 k 叉排序树。不同于一般的不同于一般的 k 叉树，b+树的节点有多个孩子，这些孩子是排好序的。因此 B+树天然适合磁盘文件这类以块为单位进行读写的类型。

通常，通过 B+查询和插入及删除都具有 O(LogN)的复杂度。相比于 O（N）的查询，有着巨大的优势。

8.2 设计思路

索引的设计是基于 B+树的数据结构，其特点非常契合磁盘的读写，而且能够以 logN 的复杂度进行查找。因此非常适合做索引。

8.3 Index模块的定义

```
class btree {
private:
    typedef std::pair<int, int>PII;
    int leaf_least;
    int leaf_most;
    int order_least;
    int nonleaf_most;
    int leafnodesize;
    int nonleafnodesize;
    int order;
    int proot;
};
```

8.4 接口定义与说明

```
class btree {
public:
    btree(const indexinfo&, int);
    ~btree() = default;
    indexinfo index;
    void insert(std::string key, int blockoffset, int offset);
    void deleteonerecord(std::string key);
    std::pair<int, int>searchequal(const std::string& key);
    std::vector<std::pair<int, int>>searchbetween(const std::string& key1,
const std::string &key2);
    std::vector<std::pair<int, int>>searchlessthan(const std::string&key);
    std::vector<std::pair<int, int>>searchlessequalthan(const
std::string&key);
    std::vector<std::pair<int, int>>searchbiggerthan(const std::string&key);
    std::vector<std::pair<int, int>>searchbiggerequalthan(const
std::string&key);
    std::vector<std::pair<int, int>>searchall();
};
```

Index 模块设计了插入、删除、查询、载入接口。

Index();函数可以创建或者载入索引。Index(Indexinfo&,0);是新建索引，Index(Indexinfo&,1);是载入已有索引。

Insert();函数为插入键值。

Deleteonerecord();是删除键值。

Searchequal();为等值查询。
Searchbetween();为区间查询。
Searchlessthan();为查询小于某一键值的区间。
Searchlessequalthan();为查询小于等于某一键值的区间。
Searchbiggerthan();为查询大于某一键值的区间。
Searchbiggerequalthan();为查询大于等于某一键值的区间。
Searchall();为查询全部键值。

九、 Buffer 模块

9.1 模块概述

BufferManager 模块是数据库用来访问文件的模块。我们知道，随着现代处理器的发展，文件读写是制约程序运行效率的很大原因。数据库需要大量读写文件，因此有必要设立缓存区，对磁盘访问进行优化。

9.2 设计思路

缓存区利用的局部性原理。经过统计我们发现，当一个数据块被访问的时候，其附近的数据块很有可能也会被访问，同时这个数据块在将来很有可能被再次访问到。这就是时间局部性和空间局部性。

利用这个局部性原理，我们可以一次读写读入一个 Block 而不是特定的多少 Bytes。这个在操作系统层面就已经为我们实现了。我们需要完成的是如何去管理这些 Block。通常的这个 Block 大小是 4K。

我们采用的是全相连的方式。理由是内存的搜索是比较快的，这种实现代价最小。

当缓存区已经满的时候，我们需要替换。替换的策略采取的是 LRU。根据统计学发现，最近最少使用的块是最不可能再次被访问到的。

9.3 数据结构定义

Buffer 的对象定义如下：

```
class BufferManager
{
private:
    const int MAXBLOCKNUM = 1024;
    struct Block* head;
    Block* tail;
    class File* fileHead;
    int number;
```

```
};
```

9.4 Buffer模块函数接口定义与说明

```
class BufferManager
{
public:
    BufferManager();
    ~BufferManager();
    void flushoneblock(Block* bufferBlock);
    void flush();
    void deletebuff();
    Block* getblock(std::string FileName, int offset);
    std::pair<Block*, int> getnullblock(std::string filename);
    Block* createfile(std::string filename);
    void deleteBlock(std::string fileName, int offset);
    void deletefileblock(std::string filename);
};
```

1. getblock 函数返回一个被申请的块。如果块不存在则返回 nullptr。

2. getnullblock 函数返回一个空块以及空块的编号。如果文件不存在则新建一个文件。

3. createfile 函数新建一个文件，并且返回该文件的第一块。

4. deleteblock 负责删除块。

5. deletefileblock 函数负责将缓存区所有该文件的所有块以及文件信息删除。

6. flush 函数负责将缓存区所有的脏块写回。

7. flushoneblock 函数负责将缓存区指定的脏块写回。

9.5 重要接口实现方法

1. getblock 函数实现方式

当需要调用 getblock 函数时，需要扫描整个缓存区。对比信息，如果能够匹配到，直接返回这个 Block。如果不能匹配到，说明这个数据块不在缓存区。因此需要调用内部的 readnewblock 将这个块读入。

2. getnullblock 函数实现方式

当需要调用 getnullblock 时，需要返回一个被申请文件的一个空块。这个函数需要扫描文件链表，找到相应的文件检查空块的 vector。如果里面有值直接将空块号弹出即可。如果没有就新建一个块。

3. deleteblock 函数实现方法

删除块只需要先获取这个块，找到块后将待删除号偏移量存储在对应文件的空块的 vector 中即可。然后从链表中删除这个块。

十、 综合测试

表的建立

首先建立好运行 minisql 所需要文件层次结构

图表 6 文件层次结构

进入欢迎界面后，运行事先写好的建立表和插入 10000 条记录的脚本文件，显示建立表格、插入记录成功。（由于插入打印提示显示过长，这里只选取部分提示）

```
welcome come to sk, zxd, jck's minisql
>> execfile student.txt;
```

```
Insert Successful!
Insert Successful!
Insert Successful!
```

1. int 类型上的等值条件查询

查询成功，并且含有主键的查询会默认使用索引。

```
>> select * from student2 where id=1080100245;
1080100245      name245         62.5
Selected using btree: 1Time: 0.002
Interpreter: successful select!
```

2. float 类型上的等值条件查询

略去了中间许多的打印结果，可以看到最后共查询到了 166 条记录。

```
>> select * from student2 where score=98.5;
NO tree
1080100003      name3           98.5
1080100046      name46          98.5
1080100085      name85          98.5
1080100137      name137         98.5
1080100201      name201         98.5
1080100242      name242         98.5
```

```
1080109724      name9724        98.5
1080109765      name9765        98.5
1080109828      name9828        98.5
Number of tuples selected: 166
Time: 0.345
Interpreter: successful select!
```

3. char 类型上的等值条件查询，并观察运行时间 t1

在没有建立索引的情况下，查询 name 属性为”name245”的等值查询时间为

0.142sec.

```
>> select * from student2 where name='name245';
NO tree
1080100245      name245         62.5
Number of tuples selected: 1
Time: 0.142
Interpreter: successful select!
```

4. int 类型上的不等条件查询
   查询成功，总计查到 9999 条记录。

```
>> select * from student2 where id<>1080109998;
```

```
1080109995      name9995        59.5
1080109996      name9996        65.5
1080109997      name9997        61
1080109999      name9999        69.5
1080110000      name10000       80.5
Number of tuples selected: 9999
Time: 10.646
Interpreter: successful select!
```

5. float 类型上的不等条件查询
   查询成功，总计查到 9834 条记录。

```
>> select * from student2 where score<>98.5;
```

```
1080109997      name9997        61
1080109998      name9998        84.5
1080109999      name9999        69.5
1080110000      name10000       80.5
Number of tuples selected: 9834
Time: 9.991
Interpreter: successful select!
```

6. 考察 char 类型上的不等条件查询
   查询成功，总计查到 9999 条记录。

```
>> select * from student2 where name<>'name9998';
```

```
1080109995      name9995        59.5
1080109996      name9996        65.5
1080109997      name9997        61
1080109999      name9999        69.5
1080110000      name10000       80.5
Number of tuples selected: 9999
Time: 10.177
Interpreter: successful select!
```

7. 多条件 and 查询
   a) 对于单个 float 属性的多条件

```
>> select * from student2 where score>80 and score<85;
```

```
1080109957      name9957        80.5
1080109978      name9978        84.5
1080109998      name9998        84.5
1080110000      name10000       80.5
Number of tuples selected: 962
Time: 1.307
Interpreter: successful select!
```

   b) 对于多个属性的多条件

```
>> select * from student2 where score>95 and id<=1080100100;
1080100001      name1           99
1080100003      name3           98.5
1080100046      name46          98.5
1080100049      name49          97
1080100052      name52          97
1080100066      name66          99
1080100085      name85          98.5
1080100090      name90          96.5
1080100094      name94          99.5
Selected using btree: 9Time: 0.004
Interpreter: successful select!
```

8. primary key 约束冲突（或 unique 约束冲突）

```
>> insert into student2 values(1080100245,'name245',100);
Duplicate entry for key 'PRIMARY'
Interpreter: insert failed!
```

9. 创建 index

```
>> create index stuidx on student2 ( name );
Interpreter: successful create!
```

10. 有索引的查找，观察运行时间 t2

与之前未用索引查找相比，时间大大缩短，t2=0.001sec, t2<t1

```
>> select * from student2 where name='name245';
1080100245      name245         62.5
Selected using btree: 1Time: 0.001
Interpreter: successful select!
```

11. 在建立 b+树后再插入数据，考察 B+树的 insert

```
>> insert into student2 values(1080197996,'name97996',100);
Insert Successful!
```

12. 有索引的查找，记录时间 t3

t3 = 0.001sec

```
>> select * from student2 where name='name97996';
1080197996      name97996       100
Selected using btree: 1Time: 0.001
Interpreter: successful select!
```

13. 记录和 B+树的 delete

记录和 B+树均做了删除操作

```
>> delete from student2 where name='name97996';
Number of tuples deleted: 1
Interpreter: successful delete!

>> select * from student2 where name='name97996';
Selected using btree: 0Time: 0.002
Interpreter: successful select!
```

14. drop index

```
>> drop index stuidx;
Interpreter: successful drop!
```

15. 删除索引后的查找时间

t4 = 0.139sec, 远大于之前的 0.001sec

```
>> select * from student2 where name='name97996';
NO tree
1080197996      name97996       100
Number of tuples selected: 1
Time: 0.139
Interpreter: successful select!
```

对于 name 的等值查询则恢复到一开始未用索引的时间

```
>> select * from student2 where name='name245';
NO tree
1080100245      name245         62.5
Number of tuples selected: 1
Time: 0.145
Interpreter: successful select!
```

16. 记录的 delete

a) 主键删除

```
>> delete from student2 where id=1080100245;
Number of tuples deleted: 1
Interpreter: successful delete!

>> select * from student2 where id=1080100245;
Selected using btree: 0Time: 0.005
```

b) 等值删除，可以删除多条

```
>> delete from student2 where score=98.5;
Number of tuples deleted: 166
Interpreter: successful delete!

>> select * from student2 where score=98.5;
NO tree
Number of tuples selected: 0
Time: 0.137
Interpreter: select failed!
```

c) 删除表格内所有记录

```
>> delete from student2;
Interpreter: successful delete!

>> select * from student2;
Selected using btree: 0Time: 0
Interpreter: successful select!
```

17. drop table

删除表格后再查询，打印查询错误信息

```
>> drop table student2;
Interpreter: successful drop!

>> select * from student2;
The table you select does not exist!
Interpreter: select failed!
```

十一、 优化拓展

11.1 查询优化

从上面的测试结果可以看出，有无索引对于 minisql 查询的速度影响是非常大的。用索引查询，较之于无索引，可以将查询时间缩小数百倍。因此，在任何一个查询属性有索引的情况下，应尽可能地使用 index 来查询。于是我们在此基础上做出了一些查询优化。

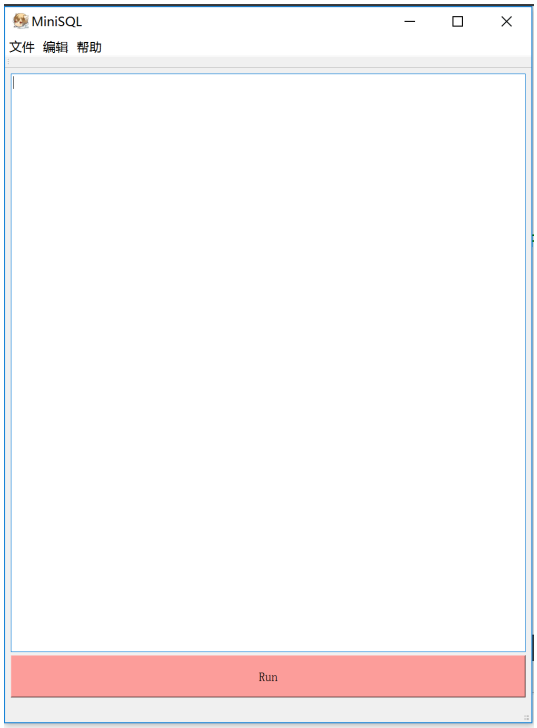
具体思路为，当出现多条件 and 查询时，我们首先对其要查询的属性类型进行分类，已有 index 的为一类，没有 index 的为一类。我们先用 B+树对已有 index 的一类进行查询，然后将查询结果求交集，得到一个预先的查询记录集合。在这个集合中，我们再对没有 index 的属性逐个进行遍历，剔除不合条件的记录后，得到最终的查询结果。

经过分析对比，查询时间得到了明显的缩短，起到了优化效果。

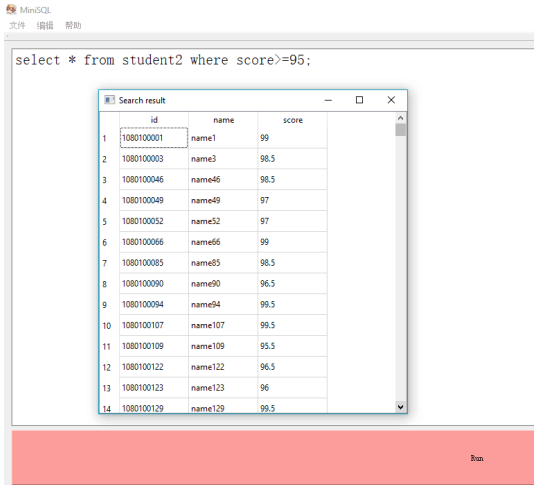
11.2 GUI拓展

我们采用了 QT 作为图形化扩展的 IDE，主要使用 QT 中的 QTextEdit 类进行类似于记事本的输入，然后将 UI 界面获得的字符串交给 MiniSQL 来处理。

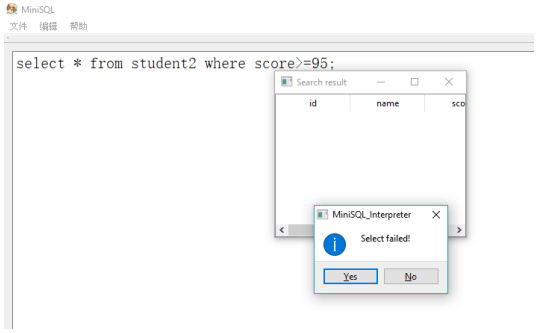
运行中的提示信息会弹出对话框，同时查询的结果会以表格的形式展现。



图表 7 GUI 主界面



图表 8 查询结果显示



图表 9 错误信息提示

十二、 小组分工

总体概念设计	沈锴 蒋晨恺 赵宣栋
通用类设计	沈锴
Interpreter 模块	赵宣栋
API 模块	沈锴
Catalog 模块	赵宣栋
Index 模块	沈锴
Record 模块	蒋晨恺
Buffer 模块	沈锴
GUI 设计	赵宣栋 沈锴
综合测试	沈锴 蒋晨恺 赵宣栋