

MiniSQL 开发文档

一、总体开发思路

1. 任务概述

实验要求设计一个简单的 DBMS，能够使用户通过字符界面实现表的建立 / 删除，索引的建立 / 删除，以及记录的插入 / 删除 / 查找。

数据库能够接受的数据类型为 int, float 以及 char(1~255)，表的定义最多有 32 条属性，能够允许 unique 属性定义和单属性主键定义。在索引方面，要求对主键自动建立 B+ 树索引，对 unique 属性可以建立或者删除 B+ 树索引，其中，B+ 树索引均为单属性、单值。

在进行记录查询时，可以允许基于属性的等值查询和范围查询，以及 and 连接的多条件查询。在进行数据更改时，要能够实现单条数据的插入以及单条或者多条记录的删除。

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：

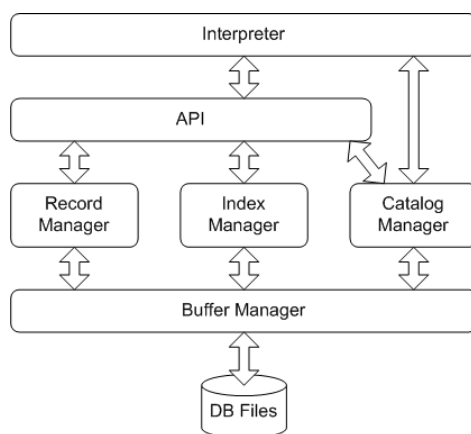
- 创建表：create table t1(tid int, tname char(8));
- 删除表：drop table t1;
- 创建索引：create index T1 on t1(tid);
- 删除索引：drop index T1 on t1;
- 选择查询：select xxx from xxx where xxx;
- 插入记录语句：insert into t1 values(111, 'xx');
- 删除记录语句：delete from t1 where xxx;
- 退出系统：exit;
- 执行脚本语句：execfile:xxx.txt;

另外，还要求设计 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。

2. 总体框架

MiniSQL 的总体框架如下：



Interpreter 模块负责接收并处理用户在前端输入的 SQL 命令, 识别命令的类型, 执行不同的子函数, 分别调用 API 模块和 Catalog 模块的功能函数, 并接收返回值。对于选择命令, 需要进一步在前端输出结果。在处理命令时, 通过规范语法进行识别, 若输入存在语法错误, 则给出提示, 若出现其他数据库错误, 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, 也应给出相应提示。若操作成功, 也返回相应提示。

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, 会调用 Catalog 中相应的函数进行操作, 这些功能函数会调用 Buffer 模块获得相应的数据块, 并在数据块上进行读写。另一方面, 一些其他模块也需要 Catalog 提供信息 (如 B+ 树查找需要索引信息), 则各自通过 API 调用 Catalog 的对应功能函数。

Index 模块负责 B+ 树索引的实现, 其中包括索引的创建, 删除, 等值查询, 与插入数据同步的键值插入, 以及删除键值。Index 模块一方面通过 Buffer 模块读写索引文件, 另一方面为 API 提供接口, 用于数据的查找。

API 模块负责各个子功能模块的相互接口。从解释器中接收非定义性命令和对应参数后, 分别根据需要调用不同模块实现命令。例如, 在进行查询时, API 首先获得解释器提供的查询参数 (表名, 属性名, 范围限制), 然后通过 Catalog 获得该表的索引定义, 根据情况决定是否使用索引查询, 最后交给 Record 模块处理。

Buffer 模块负责缓冲区的管理, 即数据块的写入与写出。其余子模块在有数据文件的访问寻求时, 会向 Buffer 发出请求, Buffer 直接访问数据文件, 获取指定数据块, 并将该数据块的读写权限教给该模块。除此之外, Buffer 还要能够实现缓冲区锁定, 缓冲区替换等一系列功能。

Record 模块主要实现数据的查找操作。包括数据文件的创建、数据文件的删除、记录的插入与删除、记录的查询。具体的查找操作的参数都由 API 传递给 Record, 然后 Record 向 Buffer 请求相应的数据块进行读写的实现。

在实际设计中, 我们首先设计了通用的数据结构与类 (base.h), 然后定好每个模块的具体功能与模块之间的接口, 之后分别进行代码实现。

二、 通用类设计

1. 基础数据结构

在 DBMS 中, 最重要的数据即为表的表示。表的构成非常复杂, 就定义而言, 其包括表的属性名称, 表属性的数据类型, 表的名称, 表的元组数量。除此之外, 表的数据项也是构成表的重要元素。

为了能设计出一个表类, 我们首先定义了如下基础数据结构:

- 属性结构

```
struct Attribute{
    short flag[32]; //data type
    std::string name[32]; //attribute name
    bool unique[32]; //unique
    int num;
};
```

其中, flag 表示每个属性的数据类型 (-1 为 int, 0 为 float, 1—255 为 char(n)), name 数组表示属性名, unique 数组表示属性是否 unique, num 表示总的属性数。

- 索引结构

```
struct Index{
    int num;
    short location[10];
    std::string indexname[10];
};
```

num 表示索引数，location 表示索引在属性列中的位置，indexname 表示索引的名字。

我们分别使用上述一个结构来表示表中的对应信息。除此之外，我们还需考虑一个单独数据项的存储方式。注意到我们事先是不知道数据类型的，因此需要特殊的结构来存储数据。我们设计了如下的多态父类指针：

```
class Data{
public:
    short flag;
    //-1-int 0-float 1~255-char.
};

class Datai : public Data{
public:
    Datai(int i):x(i){
        flag = -1;
    };
    int x;
};

class Dataf : public Data{
public:
    Dataf(float f):x(f){
        flag = 0;
    };
    float x;
};

class Datac : public Data{
public:
    Datac(std::string c):x(c){
        flag = c.length();
    };
    std::string x;
};
```

我们首先定义父类数据项 Data，其中唯一变量 flag 用于标识数据类型。然后分别设计三个子类，存储不同类型的数据。在数据生成时，我们只需要把返回的子类指针作为父类指针保存；而在访问时，可先通过访问父类的 flag 来判断数据类型，再通过指针的强制类型转换访问对应的子类里的数据项。

- Where 结构

在 `select` 和 `delete` 操作中，需要分析 `where` 后面的语句，将其具体转化为结构，方便不同模块之间的参数传递。Where 结构定义如下：

```
typedef enum{
    eq, leq, l, geq, g, neq} WHERE;

struct where{
    Data* d;
    WHERE flag;
};
```

其中 `d` 为数据项，`flag` 为枚举类型，表示比较连词，`eq` 表示等于，`leq` 表示小于等于，`l` 表示小于，`geq` 表示大于等于，`g` 表示大于，`neq` 表示不等于。

- insertPos 结构

在执行选择、插入、删除等操作时，我们需要准确的定位每一个元组所在的位置，尤其是要直到该元组在硬盘中物理文件的位置以及在内存中的虚拟位置，为此，我们定义了 `insertPos` 结构来描述元组所在的位置。该结构的定义如下：

```
class insertPos{
public:
    int bufferNUM; //在内存中的第几个区块
    int position; //在区块中的位置
};
```

该结构主要着重描述了元组在内存中的虚拟地址是多少。第一个整数表示了该元组在缓存区中区块的编号，第二个整数表示了该元组数据的起始地址在数据块中的偏移量（用字节数来表示）。虽然这个是数据在内存中的虚拟地址，但是可以很容易的通过 `BufferManager` 中的地址表来转换成相应的物理地址，两者的互相转换便于文件的读写操作的实习。

除此之外，在将元组插入索引时，还要将其转换成“长地址”，长地址 (`addr`) 的计算方法是：

$$addr = blockOffset \times position$$

在上述公式中，`blockOffset` 指的是元组在物理文件中的块偏移量，这个值可以通过 `BufferManager` 来得到。每次向索引中插入 `Key` 或者获取 `Key` 的地址时都采用的是“长地址”，这样便于索引的存储，也便于数据的流通。

2. tuple 类与 Table 类

为了设计 `Table` 类，我们首先设计了 `tuple` 类，用于表示表中的一个元组。`tuple` 类的定义如下：

```
class tuple{
public:
    std::vector<Data*> data;
public:
    tuple();
    tuple(const tuple& t);
    ~tuple();

    int length() const{
```

```

        return (int)data.size();
    }//return the length of the data.

    void addData(Data* d){
        data.push_back(d);
    }//add a new data to the tuper.

    Data* operator[](unsigned short i);
    //return the pointer to a specified data item.

    void disptuper();
    //display the data in the tuper.

};

```

其数据项由 Data 的指针型向量构成，为了便于后续操作，我们还为 tuper 类设计了几个主要的成员函数：length()用于返回元组长度，addData()用于添加新数据项，disptuper()用于显示元组数据，[]用于寻址。

在前面一系列基础结构下，我们设计了如下的 Table 类：

```

class Table{
public:
    int blockNum;//total number of blocks occupied in data file;
    std::vector<tuper*> T;//pointers to each tuper
    short primary;//the location of primary key. -1 means no
    primary key.
    Index index;
private:
    std::string Tname;
    Attribute attr;//number of attributes
};

```

(成员函数没有写出)

其中，blockNum 用于记录该表所占据的数据块数，T 作为 tuper* 的向量，用于记录表的成员数据，primary 记录主键的位置，index 为索引信息，attr 为属性信息，Tname 为表名。Table 类的构造函数如下：

```

Table(std::string s,Attribute aa, int
bn):Tname(s),attr(aa),blockNum(bn){
    primary = -1;
    for(int i = 0;i<32;i++){ aa.unique[i] = false; }
    index.num=0;
}

```

即只通过表名，属性，块数目初始化。在实际使用时，我们会先构造空表，然后调用其它成员函数进行主键和索引的设置。除一些基本的构造析构函数和信息访问函数外，我们还设计了添加数据函数和显示函数以及用于计算数据大小的函数：

```
void disp();
void addData(tuper* t);
int dataSize()
```

3. 异常类

由于程序的层次关系较为明显，对于程序运行过程中可能遇到的问题，我们不直接输出，而采取抛出异常的方式交由上层处理。不同类中会定义不同类型的异常：在通用类中，我们定义了表异常：

```
class TableException: public std::exception{
public:
    TableException(std::string s):text(s){}
    std::string what(){
        return text;
    };
private:
    std::string text;
};
```

在解释器中，定义了查询异常，用于处理语法错误和逻辑错误：

```
class QueryException:std::exception{
public:
    QueryException(std::string s):text(s){}
    std::string what(){
        return text;
    };
private:
    std::string text;
};
```

三、 Interpreter 模块

1. 设计思路

解释器模块作为整个 MiniSQL 的前端，其角色是接收用户输入的命令进行解读后通过 API 或 Catalog 调用不同的功能函数。解释器类设计如下：

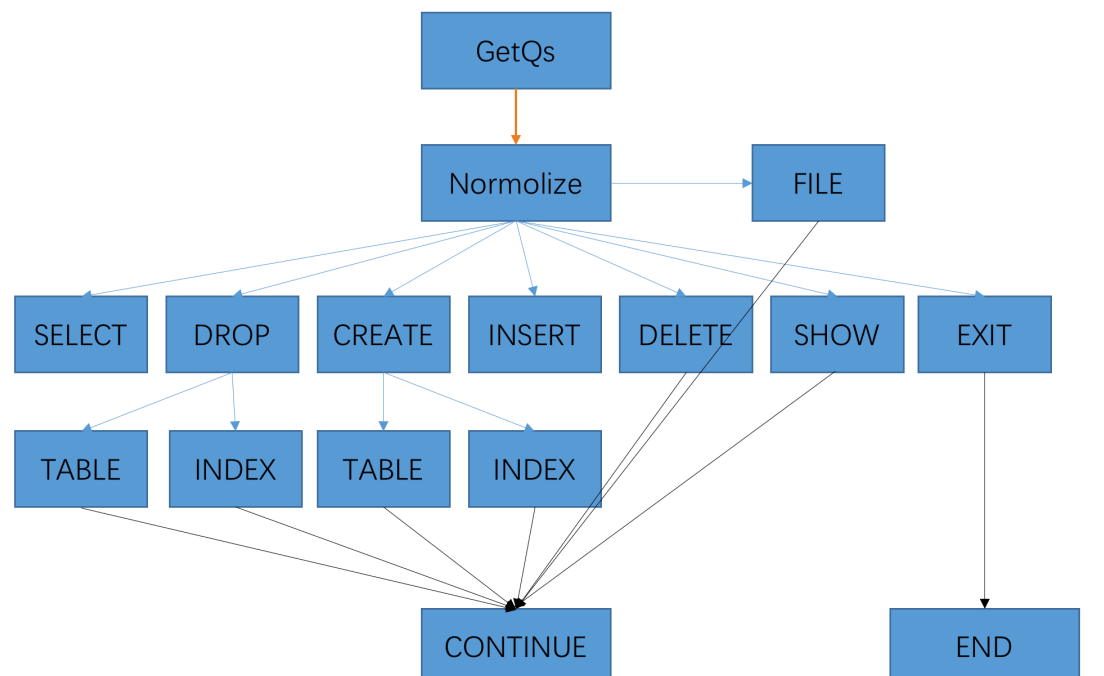
```
class InterManager{
private:
    std::string qs;//query string
public:
    int EXEC();
    void EXEC_SELECT();
    void EXEC_DROP();
```

```

void EXEC_CREATE();
void EXEC_CREATE_TABLE();
void EXEC_CREATE_INDEX();
void EXEC_INSERT();
void EXEC_DELETE();
void EXEC_SHOW();
void EXEC_EXIT();
void EXEC_FILE();
inline int GOGOGO(int pos);
void interwhere(int& pos1, std::vector<int> &attrwhere,
tuper &up, tuper &down, Attribute A,
Table* t);
void GetQs();
//Get the inputting string
void Normolize();
//Normolize the inputting string
};

```

其唯一的成员变量 qs 用于存储待处理的字符串。在 main 函数中，解释器首先通过 GetQs 函数获得用户输入的字符串，存入 qs 中，然后调用 EXEC 进行解释执行。EXEC 函数对当前 qs 字符串进行解读并操作，其首先使用 Normolize 函数对 qs 进行标准化处理，然后根据 qs 的前几个关键字选择不同的子执行函数进行执行，并根据执行的子函数的不同返回不同值。结合 main 函数中的循环体，程序执行流程如下所示：



2. 语法与功能说明

在程序开始运行时，出现如下界面：

Welcome to MiniSQL!

>>>

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。

- 创建表

创建表的格式如下：

create table 表名(属性名 类型名, 属性名 类型名, ... 属性名 类型名, primary key(属性名));
其中，关键词之间可有任意空格或回车（其余语法也相同，不再赘述），可在类型名后添加 unique 性质，主键约束可选，实例如下：

Welcome to MiniSQL!

```
>>>create table student(  
sid int,  
    sname char(8) unique,  
grade int, GPA float,  
primary key (sid)  
);
```

Interpreter: successful create!

>>>

- 创建索引

创建索引的格式如下：

create index 索引名 on 表名(属性名)，创建索引的属性必须是 unique 的，否则无法创建，会返回相应的错误信息，

实例如下：

```
>>>create index T1 on student(sname);
```

Interpreter: successful create!

>>>

- 删除索引

删除索引的格式如下：

drop index 索引名 on 表名;，若索引名不存在或有其它语法错误，则返回错误提示，示例如下：

```
>>>drop index T1 on student;
```

Interpreter: successful drop!

>>>

- 删除表

删除表的格式如下：

drop table 表名;，注意，若表中有索引，则会先删除所有的索引，再删除表，示例如下：

```
>>>drop table student;
```

Interpreter: successful drop!

>>>

- 插入记录

插入记录命令用于插入与表属性声明相符合的数据字段，如下所示：

insert into 表名 values(数据,数据,...数据)。其中，数字数据直接输入，字符串数据必须用两个单引号包含，示例输入如下：


```
>>>insert into student values(111,'hrg',3,3.9);
Interpreter: successful insert!
>>>
```

- 选择查询

选择查询用于查询已有表中的数据，其输入格式如下：

select 多个属性名(*) from 表名 where 属性名 比较运算 值 and ...;

其中，select 后可以跟具体的属性名或者*，表示显示哪些具体的属性列，where 后跟 and 连接的判断谓词，其中包括等值查询和不等值查询，能够接受的比较运算操作有：

“>,<,>=,<=,<>”。最后会输出返回的表，如果有异常则会有对应提示信息，如下所示：

```
>>>select * from student;
sid sname grade GPA
111 hrg 3 3.9
Interpreter: successful select!
>>>
```

- 删除记录

删除操作用于删除表中满足指定条件的元组，输入格式为：

delete from 表名 where 属性名 比较运算 值 and ...;如下所示：

```
>>>delete from student
where sname = 'hrg';
Interpreter: successful delete!
>>>
```

- 执行脚本文件

除了基础操作之外，还可以执行脚本文件。EXEC_FILE 函数会以流的形式读入脚本文件，以分号为标示进行分隔，分别存入 qs 变量中，然后将每个命令打印在前端上，并循环调用 EXEC 函数进行解释执行。如下所示：

```
>>>execfile:jiaoben.txt;
create table teacher( tid int, tname char(10) unique, dept
char(20), telephone int, salary float, primary key(tid));
Interpreter: successful create!
create index T1 on teacher(tname);
Interpreter: successful create!
insert into teacher values(134,'qwertt','Computers',
132434,3000.6);
Interpreter: successful insert!
select* from teacher;
tid tname dept telephone salary
134 qwertt Computers 132434 3000.6
Interpreter: successful select!
delete from teacher where dept = 'CS' and salary>=10;
Interpreter: successful delete!
drop index T1 on teacher;
Interpreter: successful drop!
drop table teacher;
Interpreter: successful drop!
```

```
>>>
```

- 退出系统

在执行完一些命令后，用户可以通过输入 `exit` 退出系统。此时首先会将缓冲区中数据块的内容强制写入文件，然后再结束程序，如下所示：

```
>>>exit;
```

Program ended with exit code: 0

- 部分异常反馈

由于在用户输入层十分容易出现异常：例如输入的语法错误，或者存在访问的不合法性（例如表的重定义，索引重定义，访问不存在的表，不存在的属性等）。下面的示例列出了几个常见的异常反馈：

```
>>>create table sss(
ss in, ss2 float);
ERROR: invalid query format!
>>>select * from sss;
ERROR in getTable: No table named sss
>>>create table student(
ss int, ss2 char(8));
Interpreter: successful create!
>>>create table student( ss int);
ERROR in create_table: redefinition of table: student
>>>create index T1 on student(ss3);
No attribute named ss3
>>>
```

3. 接口设计

解释器作为整个 DBMS 的最顶层，与其直接接口的只有 API 和 Catalog，且绝大多数接口函数都是由解释器向其他模块单方面传递参数，而且其中只有查询命令需要获得表的返回参数：

API 接口：

```
Table Select(Table& tableIn, vector<int> attrSelect,
vector<int>mask, vector<Data*>upper_bound,
vector<Data*>lower_bound);//return a table containing select results
```

（根据查询参数进行查询）

```
int Delete(Table& tableIn, vector<int>mask,
vector<Data*>upper_bound, vector<Data*>lower_bound);
```

（根据删除参数进行删除）

```
void Insert(Table& tableIn, tuple singleTuper);
```

（为一个表插入一个元组）

```
bool DropTable(Table& tableIn);
```

（删除一个表）

```
bool CreateTable(Table& tableIn);
```

（创建一个表）

Catalog 接口：

```
void create_table(string s, Attribute atb, short primary, Index index);
```

(根据基础信息创建一个表)

```
bool hasTable(std::string s);
```

(判断是否存在这样一个表)

```
Table* getTable(std::string s);
```

(得到一个指定名字的空表，只有属性定义没有数据)

```
void create_index(std::string tname, std::string aname, std::string iname);
```

(在指定表的指定属性上建立索引)

```
void drop_table(std::string t);
```

(删除表)

```
void drop_index(std::string tname, std::string iname);
```

(删除索引)

```
void show_table(std::string tname);
```

(显示一个表的各种定义，包括名称，属性类型，主键与索引信息)

```
void changeblock(std::string tname, int bn);
```

(修改一个表所占的数据块)

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 where 组成的向量表示，同时作为传递的参数。

另外，在 main 函数里，设计了如下的总体运行框架：

```
while(re){
    try{
        cout << ">>>";
        itp.GetQs();
        re = itp.EXEC();
    }
    catch(TableException te){
        cout << te.what() << endl;
    }
    catch(QueryException qe){
        cout << qe.what() << endl;
    }
}
```

程序在开始执行后，会循环调用解释器的解释函数并执行命令，直至遇到 exit 命令。其中，会捕捉解释器或者其它下层模块抛出的异常并输出。

4. 模块测试

在解释器模块中，我们新增了功能 show table，用于显示表的当前属性，因此可以通过这一命令在测试其它定义类语句，其中一个测试样例如下：

```

>>>show table ss;
ss:
s1 int unique primary key
s2 float unique
s3 char(8)
index: s1(s1)
>>>create index T1 on ss(s2);
Interpreter: successful create!
>>>show table ss;
ss:
s1 int unique primary key
s2 float unique
s3 char(8)
index: s1(s1) T1(s2)

```

除了定义类语句外，我们还测试了值访问语句。其中主要测试了 select 语句，因为 select 语句要经过很多模块的操作，因此容易出现问題。在调用 API 提供的接口前，我们首先打印出相关的查询参数，检验是否正确，测试结果如下：

```

>>>select s1,s3
from ss
where s1 = 3 and s3<>'ddd';
location of selected attr:
0 2
location of where attr:
0 2
0 5
3 ddd

```

四、 Catalog 模块

1. 设计思路

Catalog 模块的功能是管理数据库的所有模式信息，包括表的定义、属性定义、索引定义等。为了方便存储、访问并更改信息。对于单独的一个表，我们将表定义属性定义和其上的索引定义同意放在一个文件里。从文件头开始依次记录以下信息：属性数，索引数，所占的数据块数，主键位置，所有的属性名，属性的数据类型，属性是否唯一，所有索引名，索引位置。由于 Catalog 模块在程序的中层位置，其不直接访问数据文件。因此在访问时，Catalog 首先调用 Buffer 读取相应的文件，然后获得一块对应的数据块，并直接在其上进行读写。

2. 接口设计

Catalog 与解释器、Buffer、API 都有接口，其中与 API 的接口只有一个，用于其他模块，例如 Record 修改表的数据块。

```

void changelock(std::string tname, int bn);

```

Interpreter 接口：

```
void create_table(string s, Attribute atb, short primary, Index index);
```

(根据基础信息创建一个表)

```
bool hasTable(std::string s);
```

(判断是否存在这样一个表)

```
Table* getTable(std::string s);
```

(得到一个指定名字的空表，只有属性定义没有数据)

```
void create_index(std::string tname, std::string aname, std::string iname);
```

(在指定表的指定属性上建立索引)

```
void drop_table(std::string t);
```

(删除表)

```
void drop_index(std::string tname, std::string iname);
```

(删除索引)

```
void show_table(std::string tname);
```

(显示一个表的各种定义，包括名称，属性类型，主键与索引信息)

Buffer 接口：

```
int getbufferNum(string filename, int blockOffset);
```

(向 Buffer 申请访问某个文件的某个数据块，返回缓冲区中的一个数据块)

```
void readBlock(string filename, int blockOffset, int bufferNum);
```

(读取一个缓冲区的数据块)

```
void writeBlock(int bufferNum);
```

(当对一个数据块执行外写操作后，调用此函数进行记录)

```
void useBlock(int bufferNum);
```

(当对一个数据块执行完读操作后，调用此函数进行记录)

五、 Index 模块

1. 设计思路

Index 模块的主要功能是建立索引，提高表的搜索速度。索引的实现使用的是 b+树算法。在 index 模块中有两个类，一个是 bptree 类，该类的功能是初始化 b+树、实现 b+树的插入、删除、查找工作；另一个是 indexmanager 类，该类的功能是对生成的 b+树进行管理，管理 index 文件等。两个类的定义分别是：

```
class IndexManager{//IndexManager.h
public:
    IndexManager(){};//indexmanager文件建立
    void Establish(string file);//建立索引
    void Insert(string file, Data* key, int Addr);//插入键值
    void Delete(string file, Data* key);//删除键值
    int Find(string file, Data* key);//搜索键值
    void Drop(string file);//删除索引
    int*Range(string file, Data*key1, Data*key2);//范围搜索键值
```

```
~IndexManager() {};//析构函数  
};
```

IndexManager 管理当前目录下的 Index 文件，并对这些 Index 文件进行插入、删除、查找等等操作。

```
class index{//bptree.h  
public:  
    int Number;//记录当前index文件结点数目;  
    int keylength[3];//对应三种类型的键值: int float string  
    BufferManager bf;  
private:  
    int maxchild;//b+树的最大儿子个数  
    int order;//阶数  
    int type;//0 int; 1 double; 2 string//键值类型  
    string name;//index文件名  
public:  
    index(string filename);//建立一个索引  
    ~index() {};//析构函数  
    void initialize(Data* key, int Addr, int ktype);//索引初始化  
    int find(Data* key);//搜索键值  
    void insert(Data* key, int Addr);//插入键值  
    int* split(char* currentBlock, Data* mid, Data* key, int Addr, int  
leftpos, int rightpos);//分裂b+树节点  
    void Internal_insert(char* currentBlock, Data* mid, int leftpos, int  
rightpos);//向中间节点插入数据  
    void SplitLeaf(char* block1, char*block2, char* currentBlock, Data*  
key, int Addr);//分裂b+树叶节点  
    void SplitInternal(char* block1, char*block2, char* currentBlock, Data*  
mid, int leftpos, int rightpos);//分裂b+树中间节点  
    void Delete(Data* key);//删除键值  
    int* Range(Data*key1, Data*key2);//查找范围中的键值  
};
```

Private 变量是记录当前 index 文件中的 b+树的基本信息。如果当前目录没有目标 index 文件，则新建一个文件；如果有目标 index 文件那么就读取 index 文件中的信息并初始化 index 类。由于节点的大小与 Buffer 中的一块大小相同，所以每次读取一个节点，并区分叶节点和非叶节点。非叶节点查找键值应该存在的叶节点块位置，叶节点在相应位置插入或查找键值、地址即可。

2. 接口设计

IndexManager 仅和 BufferManager、API 有接口，在整个函数中调用的部分就只有 BufferManager。通过 extern 引用其他部分定义的 BufferManager，具体的操作方法见 BufferManager。

API 调用方式：

IndexManager接口:

```
IndexManager() {} ; (建立indexmanager类)
void Establish(string file); (建立index文件, 如果存在则初始化
IndexManager信息)
void Insert(string file, Data* key, int Addr); (向index文件中插入键值)
void Delete(string file, Data* key); (在index文件中删除键值)
int Find(string file, Data* key); (在index文件中寻找键值)
void Drop(string file); (删除index文件)
int*Range(string file, Data*key1, Data*key2); (在index文件中进行范围搜索)
~IndexManager() {} ; (析构函数)
};
```

六、 API 模块

1. 模块概述

API 是整个系统的核心, 在整个系统中起到了承上启下的作用。在与上层模块的对接中, API 主要功能是处理来自 interpreter 各种请求, 比如创建表, 删除表之类的任务。在对接下层模块中, API 主要是根据相应的功能来调用不同的模块。也就是说各个模块之间的相互协同工作主要是通过 API 实现的。进过 API 的数据流主要有两个方向。一个是自上而下的, 这是在 interpreter 接受到用户的输入之后, 根据用户的请求直接调用 API 中的相应模块。然后 API 再根据不同功能以及不同模块的需要, 把 interpreter 传入的数据分别传送给不同的模块, 供其实现各自的功能。另一条数据流是自下而上的, 这是当各个模块完成自己的功能返回结果时, 需要把得到的结果数据返回给 interpreter 供其输出, 因此这时也会通过返回值的方式返回得到的数据, 将数据传输到最顶层。

2. 主要功能

1. 创建表: 每次建立表的时候, 都要调用 RecordManager 以及 IndexManager, 通过 RecordManager 创建一个表, 并且通过 indexManager 根据表中的主键来建立相应的索引。

2. 删除表: 每次删除表的时候, 要调用 RecordManager 以及 IndexManager, 分别需要清空缓存区中所有与相应表有关系的数据, 并且需要将在这个表上建立的所有索引都删除, 然后再把硬盘上的相关文件全部删除。

3. 数据的插入: 插入数据时, 直接调用 RecordManager 即可, 把表的信息以及元组的数据传送给 RecordManager, 然后如果没有异常抛出, 就说明插入数据成功。

4. 数据的删除: 删除数据时, 也直接调用 RecordManager 即可, 把表的信息以及要删除的条件送个 RecordManager, 然后会获得一个返回值, 表示给定表中有多少组数据被删除, 将这个值返回给 interpreter, 用于输出, 使得用户可以方便的得到信息。

5. 数据的查找: 查找数据也直接调用 RecordManager 即可, 然后会得到一个表作为一个返回值, 这个表中存放这查询得到的记录, interpreter 可以直接输出这个结果。

6. 创建索引以及删除索引: 这两个与索引直接相关的函数直接调用了 IndexManager 的相应函数接口, 需要注意的是在创建索引的时候, API 需要根据表名以及创建索引对应的位置来给出不同的文件名。

3. 接口设计

API 的接口都是外部接口，而且都是被 interpreter 调用的，所有的 API 接口如下：

```
Table Select(Table& tableIn, vector<int> attrSelect,
vector<int>mask, vector<where> w); //return a table containing select
results
int Delete(Table& tableIn, vector<int>mask, vector<where> w);
//delete table时调用
void Insert(Table& tableIn, tuper& singleTuper); // 向table中插入数据
时调用
bool DropTable(Table& tableIn); // drop table时调用
void DropIndex(Table& tableIn, int attr); // drop index时调用
bool CreateTable(Table& tableIn); // create table 时调用
bool CreateIndex(Table& tableIn, int attr); //创建索引
```

4. 实现方法

1. CreateTable 语句

函数原型：bool CreateTable(Table& tableIn);

实现方法：在创建表的时候，要检查创建的表中是否存在主键，如果存在主键，那么就对主键建立索引。此过程需要调用 CreateIndex 函数。下一步就是调用 RecordManager 中创建表的函数，来创建一个表。

2. CreateIndex 语句

函数原型：bool CreateIndex(Table& tableIn, int attr);

实现方法

该函数的功能就是在指定的表中，对第 attr 个属性建立索引。在实现这个函数的过程中，我们需要根据表以及属性的位置来转换成相应索引文件的文件名。为了保证每一个表中都可以有多个 index，每一个 unique 属性上都可以建立一个索引，我们采用的方法是把表名加上所选属性的位置序号来作为索引文件的文件名。这样就是得每个表可以建立多个索引。准备好文件名之后就可以直接调用 IndexManager 来创建相应的索引了。

3. Insert 语句

函数原型：void Insert(Table& tableIn, tuper& singleTuper);

实现方法

由于在实现 insert 的过程中同时需要调用 RecordManager 与 IndexManager，因此就充分的体现了 API 的中枢作用。在 insert 开始之前，需要先对主键以及创建了索引的具有 unique 属性的键通过 IndexManager 进行等值查找。如果找到了相应的位置，那就说明数据中已经存在了现有的记录，所以就不可以再次插入了，只能抛出异常，提示是主键重复或者是 unique 属性重复。

如果 IndexManager 并没有返回一个有效的地址，就说明数据堆中可能不存在相应的数据，或者是有些具有 unique 属性的键并没有被建立索引。这时候就要判断是否还有多余的、未建立索引的 unique 属性。如果没有，那么就说明可以插入，调用 RecordManager 直接进行数据的插入。如果存在这样的属性，那就要调用 RecordManager 中具有查重功能的 insert 函数。

4. delete 语句

函数原型：int Delete(Table& tableIn, vector<int>mask, vector<where> w);

实现方法

与 insert 语句相似, delete 语句也需要先调用 IndexManager 来删除相应的索引记录, 然后在调用 RecordManager 的功能删除相应的行。

5. select 语句

函数原型: `Table Select(Table& tableIn, vector<int> attrSelect, vector<int>mask, vector<where> w);`

实现方法

在进行 select 时, 需要处理的有等值查询与范围查询, 对应两种不同的查询方式, 我们采用了不同的方法。在程序实际运行的过程中, 先判断是否存在等值查询, 然后在判断是否还需要进行范围查询。

如果刚开始判断有等值查询, 那就判断有没有建立在对应属性上的索引, 如果没有, 那就直接跳过这步。如果有, 那就通过 IndexManager 来查询这条记录对应的地址是多少, 然后, API 把查询到的地址传递给 RecordManager, 通过它来获取该地址上的数据。

如果没有相应的索引信息, 那么查询则需要通过 RecordManager 来遍历所有的记录来实现。但是, 直接遍历的效率过低, 我们采用了“列查询”的方式来遍历数据, 使得遍历的效率极大的提高。具体有关“列查询”的实现方法以及测试将在后文的测试中详细说明。

在查询完成之后, API 就可以把 RecordManager 得到的表返回给 interpreter, 用于最后的输出。

6. DropTable 语句

函数原型: `bool DropTable(Table& tableIn);`

实现方法

在进行该操作时, 直接调用 RecordManager 的 DropTable 函数就可以删除所有该表的数据信息。然后再对所有表中所有的 index 执行 DropIndex 操作, 就可以把整个表的信息全部删除了。

6. DropIndex 语句

函数原型: `void DropIndex(Table& tableIn, int attr);`

实现方法

需要注意的是该函数的第二个参数是索引对应的属性在所有属性中的位置, 也就是指这是第几个索引。这个函数可以直接调用 IndexManager 的相应接口函数, 来实现索引的删除。

七、 Record 模块

1. 模块概述

RecordManager 是一个用于管理数据记录的模块。每当有数据插入一个表或者是删除一条或多条记录时, 就需要调用 RecordManager 来管理表中的数据, 因此, 该模块负责的功能主要是数据的插入、删除, 表的建立与删除, 以及从表中读取数据时也需要该模块的工作。该模块主要负责软件与数据文件的交互, 以及数据文件的解析处理。

处理软件与数据文件的交互时, 每次会从用户处获取要处理的表, 然后通过 BufferManager 将指定的表的数据加载到内存中, 然后将内存中的数据转换成为更加便于操作的数据记录。建立表与删除表时, 也需要通过 RecordManager 建立相应的表文件或者是删除相应的表文件。

在进行数据的文件处理与解析时, 每次会根据用户的指令来执行相应的操作。例如当用户执行 select 操作时, 该模块会把所有满足 select 条件的数据转换成相应的元组, 放到相应的表中, 返回给用户。

2. 主要功能

1. 创建与删除表：每次建立或者删除表的时候，都要调用 RecordManager，删除相关表的数据，并且把相应的索引信息一同删除。

2. 数据的插入：插入数据时，需要先判断插入的数据是否有主键，是否有某些有 unique 属性的键，如果有，则先进行查重，如果没有重复的，则将数据插入到表中，也就是把记录写到内存中的相应位置上，剩下的事情交给 BufferManager 处理。

3. 数据的删除：删除数据时，采用 Lazy deletion 的方式，先判断该表中是否有对应的索引，如果有对应的索引，则通过索引进行查找。如果通过 index 没有找到，那就直接遍历一个 block 中的所有数据，判断是否有要删除的数据。找到记录对应的位置后将该条记录标记为“被删除”状态，表示记录已经被删除。

4. 数据的查找：查找数据时，先判断是否有等值查找，如果有等值查找，并且在对应的表上还建立有索引，那就可以直接通过 index 找到相应的数据，然后插入到结果中返回。否则，就遍历数据文件中的所有表，判断表中的每一条数据是否满足查找的条件，如果满足就插入到结果的表中，直到查询到数据文件的结尾，然后返回最终生成的表。

3. 接口设计

所有的接口分为内部接口与外部接口，外部接口供别的模块调用，而内部接口则是为了实现外部接口而产生的函数。

外部接口

```
Table Select(Table& tableIn, vector<int>attrSelect, vector<int>mask,
vector<where> w);
//执行select语句
void Insert(Table& tableIn, tupler singleTuper);
//插入数据
int Delete(Table& tableIn, vector<int>mask, vector<where> w);
//删除数据
bool DropTable(Table& tableIn); //删除表
bool CreateTable(Table& tableIn); //创建表
```

内部接口

```
Table Select(Table& tableIn, vector<int>attrSelect);
//没有where语句的select
bool isSatisfied(Table& tableInfor, tupler row, vector<int> mask,
vector<where> w);
//判断单条元组是否满足输入的条件
char* Tuper2Char(Table& tableIn, tupler singleTuper);
//将一条记录转换为字符串，用于数据的存储
Table SelectProject(Table& tableIn, vector<int>attrSelect);
//将选择出来的元组进行投影，使得结果只有给定的属性
tupler String2Tuper(Table& tableIn, string stringRow);
//将字符串转换成元组，用于从数据块中读取数据
```

4. 设计思路

设计 RecordManager 就是为了让程序可以将数据记录到文件中，并且下次可以根据表的名字读取相应的数据。这个过程主要有两个要求，一个就是要求稳定性，另一个要求性能要高。稳定性要求每次数据的读取和写入都不能出现数据的丢失以及损坏。因此就要求读写数据时尽量要求保持数据的原样。对于性能的要求，我们采用的方法是 Lazy deletion，该方法在删除数据是只做标记，并不需要把删除后面的全部数据都向前移，因此会给性能带来很大的提升。

对于第一个问题，我们采用的方法是在数据存取转换时直接原样拷贝。我们用 char 的方式来表示最后的数据，把其他所有的类型的数据都直接拷贝到长度相对应的字符串中，这种方法极大的降低了数据丢失与损坏的可能性。比如一个表的属性的类型分别是 int, float, char(10)那么我们就直接用 memcpy 函数把相应的值拷贝到对应的数据区块。其中对于字符串数据，不论字符串的原有长度是多少，我们都按照其最大的长度进行转换，这样可以使得数据块的数据对齐，方便我们在读取数据时进行的操作。以下是一个数据转字符串的实例：

```
Table Select(Table& tableIn, vector<int>attrSelect, int data;
char dataRAW[BLOCKSIEZ];
memcpy(dataRAW, &data, sizeof(int));
```

我们按照这种方法就可以保证数据保存到文件中时一定是按照原来的格式保存的，保证了数据不会丢失。同理，在进行数据读取时，我们也采用这样的方法，例如一下是一个读取浮点数的事例：

```
float data;
char dataRAW[BLOCKSIZE];
memcpy(&data, dataRAW, sizeof(float));
```

通过这样的方法，我们可以把缓存区中的数据读取到 data 变量中。完成了数据的读取。

对于第二个问题，我们采用 Lazy deletion 的方法。由于这种方法需要一个位做标记为，因此我们需要额外在每一条数据前加一位来标识这条记录是否有效。插入数据是通过 getInsertPosition 函数来找到数据块中可用的空位置，这个位置可能是在文件块的末尾，也有可能是之前删除数据后形成的空腔。然后我们就可以直接把数据插入到这些位置。

5. 实现方法

1. Select 语句

函数原型：Table Select(Table& tableIn, vector<int>attrSelect, vector<int>mask, vector<where> w);

实现方法

其中，函数的第一个输入是一个空的表头，第二个输入是要选择哪些属性，要选择的属性的位置一整数的形式放在一个容器中。第三个数据是针对 where 语句而增加的，就是判断哪些属性后面需要有特殊条件的限定。最后一个属性就是选择的条件，也就是 where 子句后面的条件。

在实现 Select 时，要先判断 where 后面的条件是否为空，如果是空的，那就直接调用 SelectProject 函数，把所选则的表投影到 select 后面的指定的几个属性上。如果后面的属性非空，那就需要每次执行时先找到相应的表在硬盘上的文件，然后通过 Buffer-

Manager 将这些文件读取到内存中，接着在内存中通过 `String2Tuper` 函数将字符串转换成相应的数据。并把这些数据插入表中，最后将这个表进行投影，返回投影后的表。

如果 where 条件不为空，那么就需要调用 `isSatisfied` 函数逐条判断该条元组是否满足所给的条件，如果满足，就把该条记录插入到结果的表中。一直这样循环，直到遍历玩所有的元组为止，然后将最后形成的表输出就可以得到结果。

2. Insert 语句

函数原型：`void Insert(Table& tableIn, tuper singleTuper);`

实现方法

在向表中插入数据时，要先判断该表中有没有主键，以及有没有 unique 属性的键。如果有这些键，就要进行防重复检查。那我们就需要进行以此 select 操作，被操作的表就是要被插入数据的表，select 的条件就是等于主键或 unique 键等于要插入的元组的键值。要注意的是，如果一个表中存在两条 unique 键，那就要分别进行两次 select，判断这两次 select 的结果是否都等于空，如果是这样，才可以插入数据。如果表中没有主键，也没有 unique 属性的键，那就可以直接向表中插入元组。

插入时需要先获取可以插入数据的位置，然后调用 `Tuper2Char` 函数，把要插入的元组的信息用之前提到的方法转换成字符串，然后将字符串直接写入缓存区中，并且调用 `writeBuffer` 函数，表示已经对 buffer 进行了写操作。就可以进行其他的操作了。

3. Delete 语句

函数原型：`int Delete(Table& tableIn, tuper singleTuper);`

实现方法

在进行 delete 操作时，需要先遍历所有的数据，每条记录都用 `isSatisfied` 函数判断是否满足所给的条件。如果满足，就把缓存区中的该条元组标记为无效。达到删除的效果。

4. CreateTable 语句

函数原型：`bool CreateTable(Table& tableIn);`//创建表

实现方法

CreateTable 所需要做的事情十分简单，就只需要新建立一个以表名为文件名的数据文件就可以了。之后如果有数据插入再调用 insert 函数。

5. DropTable 语句

函数原型：`bool DropTable(Table& tableIn);`//删除表

实现方法

要想删除一个表，需要把缓存区中所有有关这个表的数据块全都设置为无效，这个可以调用 BufferManager 的 `setInvalid` 函数来实现，这样就可以直接删除这些还没有被写入到文件中的函数，接着还要删除硬盘上有关指定表的函数。这个可以直接调用系统函数实现。

八、 Buffer 模块

1. 模块概述

BufferManager 是一个用于管理缓冲区的模块，被我们定义为一个硬件模块，也就是说我们要用软件来模拟这个模拟实现这个模块的功能。他的主要功能就是负责内存与硬盘上文件和数据的交互。程序的所有的数据读取与写入都是直接在内存中的缓冲区进行的，操作完成之后由 BufferManager 来检查是否需要把相应的缓冲区块写回到文件中。

在每一个 BufferManager 中存放着多个 block，每个 block 对应着一块缓存区，为了保证与磁盘的交互效率达到最大化，我们将每个缓存区的大小设置为 4K，这样每次读取文件或者向缓存区中写数据时，操作的都是 4K 大小的文件块。每次要读取文件或者数据时，都要向 BufferManger 提出申请，如果 BufferManager 发现被请求的文件不在内存中，则将文件加载到内存中，然后在把内存块的编号返回给请求者。如果文件本身就在内存块中，则直接把内存块标号返回。

其他的模块获取到 BufferManager 返回的编号值以后，可以直接凭借这个编号访问对应的缓存取。类似于物理寻址的方式，BufferManager 会开辟一块专门的缓存区位置，然后所有的函数要访问数据时都需要凭借之前得到的内存块标号来在这个区域中存取数据。实现相应的功能。

2. 主要功能

1. 当做整个程序与硬盘上数据的接口，每次需要读取硬盘上的文件或者是向硬盘上写文件时都需要与 BufferManager 进行交互。

2. 负责缓存区的管理，比如当某个缓存块要被替换时，要负责判断该块的内容是否需要被写会文件，并负责将脏数据写会文件，加载新的模块。在程序结束时也要把所有的脏数据写会文件中。

3. 模拟 LRU 算法，每次替换 block 时选择替换掉最近访问时间最早的块。

3. 接口设计

所有的接口分为内部接口与外部接口，外部接口供别的模块调用，而内部接口则是为了实现外部接口而产生的函数。

外部接口

```
int GiveMeABlock(string filename, int blockOffset);  
//输入文件名和块偏移量，获取相应的内存块编号  
insertPos getInsertPosition(Table& tableinfor);  
//返回插入数据的可行位置  
void writeBlock(int bufferNum);  
//block被改写时调用，做标记  
void useBlock(int bufferNum);  
//读block时调用该函数，为该块做标记
```

内部接口

```
void flashBack(int bufferNum);
//将block立刻写入文件
int getbufferNum(string filename, int blockOffset);
//获取指定block在内存中的编号
void readBlock(string filename, int blockOffset, int bufferNum);
//将文件块读取到block中
int getEmptyBuffer();
//寻找内存中空的block
int getEmptyBufferExcept(string filename);
//寻找内存中空的block, 并且不能替换掉给定的文件
int addBlockInFile(Table& tableinfor);
//文件后插入新的block, 返回新的block在内存中的编号
int addBlockInFile(Index& indexinfor);
//在Index文件中插入新的块
int getIfIsInBuffer(string filename, int blockOffset);
//判断指定文件块是否在内存中, 如果在, 则找到指定block在内存中的编号
void scanIn(Table tableinfo);
//把整个表文件全部都读入内存中
void setInvalid(string filename);
//将整个文件涉及到的所有块标记为无效, 在删除表和 index 的时候使用
```

4. 设计思路

在设计整个 BufferManager 之前, 我们要先设计单个缓存块的结构。一个 block 我们定义为一个类, 其中有一个长度为 4096+1 的字符串, 用来存储整个 block 的数据, 还有 isWritten, isValid, filename, blockOffset, recent_time 等属性来判断该块的状态。通过 initial 函数可以将整个 block 初始化。

在 BufferManager 中有一个 block 的数组, 里面存放着 N 个 block, 这个值根据用户的定义不同而不同。我们初始设置一个 BufferManager 中存放这 5 个 block。BufferManager 被定义为软件的“硬件”因此是整个程序的全局变量, 每次执行时都只会生成一个对象。其他的模块不可以在其内部生成 BufferManager, 只能调用全局的 BufferManager。

BufferManager 要实现 LRU 算法, 即每次新的 block 只能替换最长时间没有被访问的块。该算法可以借助 block 中的 recent_time 变量来实现。

5. 实现方法

1) GiveMeABlock 实现方式

函数原型: int GiveMeABlock(string filename, int blockOffset);

该函数的功能是调用者给定一个文件名以及所需要的文件中的第几块 block, 然后返回该块在内存的缓存区中的代号。首先该函数会先调用 getIfIsInBuffer 函数来判断这个文件的指定块是否已经存在在内存中, 如果已经存在, 就直接放回相应的编号, 否则返回-1 表

示不存在。

如果上述函数返回-1，则调用 `getEmptyBuffer` 函数来寻找一个空的 block，如果所有的 block 都是满的，就遍历所有 block，查看他们最近被修改的时间，找到最长时间未被访问的块，然后将该块的编号返回，如果此块被标记为 dirty，就说明被修改过，则需要将该块的内容重新写会文件，否则直接把该块的内容清空即可。

进过上一个步骤，我们就得到了一个空的数据块，然后调用 `readBlock` 函数把文件中的数据读取到内存中。

2) `getInsertPosition` 实现方式

函数原型：`insertPos getInsertPosition(Table& tableinfor);`

该函数的功能就是返回插入数据的位置。当需要在表中插入新的数据时，就会调用该函数，函数会读取这个表对应的内存块，然后找到一个空的位置，返回内存块的编号和内存块中的偏移量。函数会根据给定表的属性来判断一条记录占用的存储空间，以此来一条一条的遍历每一条记录。如果整个块里都没有可行的插入位置，那就调用 `addBlockInFile` 函数，将指定 table 对应的文件后面新增一个 block。

3) `insertPos` 实现方式

函数原型：`void writeBlock(int bufferNum);`

该函数是提供给其他模块使用的，每次其他模块修改过某个 block 中的数据时，就调用这个函数，表示对相应的 block 做出了修改。调用这个函数会把这个 block 标记为 dirty，表示被修改过。同时还会调用 `useBlock` 函数，表示这个 block 已经被访问。

4) `useBlock` 实现方式

函数原型：`void useBlock(int bufferNum);`

该函数是提供给外部的其他模块调用的，每次使用其他模块访问指定内存中的 block 时，尤其是从其中读取数据时，要调用该函数，表示对相应的 block 做出了修改，以便使用 LRU 算法时检查最近访问的时间。

九、 综合测试

1. 表的建立

首先执行第一个关于表建立和索引建立的脚本文件，交互窗口和当前数据文件（.table 为数据文件，T_为 catalog）如下。由于此时程序未结束，因此表的定义信息都存储在 Buffer 中。

```
>>>execfile:0_create_table_scheme.txt;
create table City ( ID int, Name char(35) unique, CountryCode
char(3), District char(20), Population int, primary key(ID) );
Interpreter: successful create!
create table CountryLanguage ( CountryCode char(3), Language
char(30), IsOfficial char(1), Percentage float, primary
key(CountryCode) );
Interpreter: successful create!
>>>
```

 City.table	今天 下午5:03	0 字节
 CountryLanguage.table	今天 下午5:03	0 字节
 MiniSQL	前天 下午5:32	466 KB
 T_City	今天 下午5:03	0 字节
 T_CountryLanguage	今天 下午5:03	0 字节

2. 数据的插入、查找、删除

下面执行插入数据的脚本文件，程序运行结果如下所示（由于插入数据较多，仅选取一部分打印结果）：

```
>>>execfile:1_insert_small_data.txt;
insert into City values (1,'Kabul','AFG','Kabol',1780000);
insert into City values
(2,'Qandahar','AFG','Qandahar',237500);
insert into City values (3,'Herat','AFG','Herat',186800);
insert      into      City      values      (4,'Mazar-e-
Sharif','AFG','Balkh',127800);
...
insert into CountryLanguage values ('DEU','Turkish','F',2.6);
Unique Value Redundancy occurs, thus insertion failed
insert into CountryLanguage values ('DJI','Arabic','T',10.6);
insert      into      CountryLanguage      values      ('DMA','Creole
English','F',100.0);
>>>
```

注意到倒数第二条语句插入时，系统在进行 unique 检查时发现该条数据存在重复，故拒绝了数据插入。

然后我们进行数据查找，此处仅列出两条查询结果，如下所示：

```
>>>select Name, Population from City where ID=3;
Name  Population
Herat 186800
Interpreter: successful select!
>>>select* from CountryLanguage
where Percentage>=70.0;
CountryCode Language IsOfficial Percentage
ALB Albaniana T 97.9
ARM Armenian T 93.4
ATG Creole English F 95.7
BGD Bengali T 97.7
BGR Bulgariana T 83.2
BHS Creole English F 89.7
BIH Serbo-Croatian T 99.2
BMU English T 100
BRB Bajan F 95.1
CHN Chinese T 92
COM Comorian T 75
CPV Crioulo F 100
CUB Spanish T 100
DMA Creole English F 100
Interpreter: successful select!
>>>
```


最后我们测试数据的删除，执行如下两条命令，结果如下：

```
>>>delete from City where ID=3;
Interpreter: successful delete!
>>>select ID,Name from City where ID<=5;
ID  Name
1   Kabul
2   Qandahar
4   Mazar-e-Sharif
5   Amsterdam
Interpreter: successful select!
>>>
```

3. 索引测试

首先我们检查当前目录下文件：

 City.table	今天 下午5:09	0 字节	Xcode :
 City0.index	今天 下午5:09	8 KB	文稿
 CountryLanguage.table	今天 下午5:03	0 字节	Xcode :

可以看到一个 City0 的索引文件，实际上这是表建立时为主键创建的索引。但是在表刚建立时这个文件不存在，因为此时索引没有内容，而当我们进一步写入数据时，索引文件就会被创造。




下面我们建立一个索引并打印此时的表信息，如下所示。

```
>>>create index name_idx on City(Name);
Interpreter: successful create!
>>>show table City;
City:
ID int unique primary key
Name char(35) unique
CountryCode char(3)
District char(20)
Population int
index: ID(ID) name_idx(Name)
>>>
```

4. 表的删除

最后我们使用 drop 语句删除表，删除后文件目录中已经不包括任何该表的数据：

```
>>>drop table City;
Interpreter: successful drop!
>>>exit;
```

 CountryLanguage.table	今天 下午5:03
 MiniSQL	前天 下午5:32
 T_CountryLanguage	今天 下午5:05

5. *查重优化（列查询）

原理介绍

“列查询”的主要原理就是：不查询所有的数据，只查询要筛选的数据。这个查询的实现主要是依靠数据文件的存储方式而实现的。在创建表之后，我们就可以直接计算出每条记录的最大长度，在文件中存储时，我们总是按照这个最大的长度存储。这样就实现了数据记录的对齐。既然所有的数据记录都已经对齐，那么我们要查询记录时，就只需要查询我们需要的那部分内容就可以了。

例如，一个表的属性有一个整数 ID,一个长度为 10 的字符串 Name。根据这些数据，我们就可以计算出一条记录的总长度为 15（还有一个字节标记是否有效）。那如果查询的条件是 $ID > 500$ and $ID < 1000$ 时，我们只要遍历没 15 个字节的前 4 个字节就可以判断这条记录是否满足要求筛选条件。如果满足，就可以直接选出这条记录，当作返回值。如果不满足，就直接跳过。这样会减少很多数据转换成元组的操作，同时，就会比较大的提高数据库的效率。

效果对比

测试平台：windows 10 操作系统，Intel i5-4210U CPU@1.7GHz

数据规模：一共 4000 组数据，每组数据有 5 个属性，其中一个为主键，还有一个属性有 unique 约束。

测试效果：

在进行大数据插入时，我们采用了助教提供的数据样例，其中总共有 4000 组的数据。采用“列查询”的方式，我们可以在 27 秒的时间内完成全部的插入工作。但如果采用之前的方法，则需要至少 30 分钟的时间。因为我们发现当数据插入到 500 条以后时，基本上每秒之能插入 2 条记录，而且随着记录总数的增多，插入的速度会越来越慢。在测试进行 10 分钟后一共插入了 1800 条数据，与 4000 条数据相距甚远。

原因分析：

在使用原始的插入方式时，我们每次插入都要进行以此 select，选择的条件就是主键与插入元组的主键相等，unique 约束的键与插入元组的 unique 键相等。如果选择得到的结果非空，那就说明新的元组已经重复，不能进行插入。然而，每一次调用 select 的开销是及其巨大的，需要调用多个函数，包括表的构造函数与析构函数等。还会调用 new 与 delete 从系统获得空间存放临时数据，或是释放数据。这都会产生十分巨大的开销。因此会耗费很多的时间与系统资源。

然而，在使用“列查询”的方式进行查询时，我们只需要在查询第一条记录时，计算出要查询的记录在文件中一条记录的第几位到第几位，以及每条记录的长度是多少，这样，我们在查询时并不需要分配或获取额外的空间，每次只要将指针的位置向后移动一定的位数，就可以准确的找到记录的位置。这种仅靠指针移动来比较信息的方式当然会比调用多个函数的比较方式节约了很多时间。

而且，在原始的比较方法中，我们必须把元组中的所有数据都从数据文件中还原出来，然后才能进行比较。而“列查询”的方法只要还原出要查询的数据，就可以完成比较。因此还原数据所用的时间以及内存开销也会减少很多。从而大大的提高了效率。