

MiniSQL实验报告

1 实验要求

本次实验要求设计并实现一个精简型单用户SQL引擎，允许用户通过字符界面输入SQL语句实现表的建立和删除；索引的建立和删除以及表记录的插入、删除和查找。

1.1 详细功能要求

1.1.1 数据类型

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

1.1.2 表定义

MiniSQL的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。

1.1.3 索引的建立和删除

MiniSQL会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）。

1.1.4 查找记录

MiniSQL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。

1.1.5 插入和删除记录

MiniSQL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。

1.2 语法要求

1.2.1 创建表

该语句的语法如下：

```
create table tableName (  
    列名 类型 ,  
    列名 类型 ,  
    .....  
    列名 类型 ,  
    primary key ( 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：

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

1.2.2 删除表

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：

```
drop table student;
```

1.2.3 创建索引

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：

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

1.2.4 删除索引

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：

```
drop index stunameidx;
```

1.2.5 查找记录

格式：

```
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。

实例：

```
select * from student;
select * from student where sno = '88888888';
select * from student where sage > 20 and sgender = 'F';
```

1.2.6 插入记录

格式：

```
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：

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

1.2.7 删除记录/全表

格式：

```
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：

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

1.2.8 退出

格式:

```
quit;
```

1.2.9 执行SQL脚本

格式:

```
execfile 文件名;
```

实例:

```
execfile file.txt;
```

SQL脚本文件中可以包含任意多条上述8种SQL语句，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。

2 总体设计

2.1 成员分工

段皞一: Interrupter和api

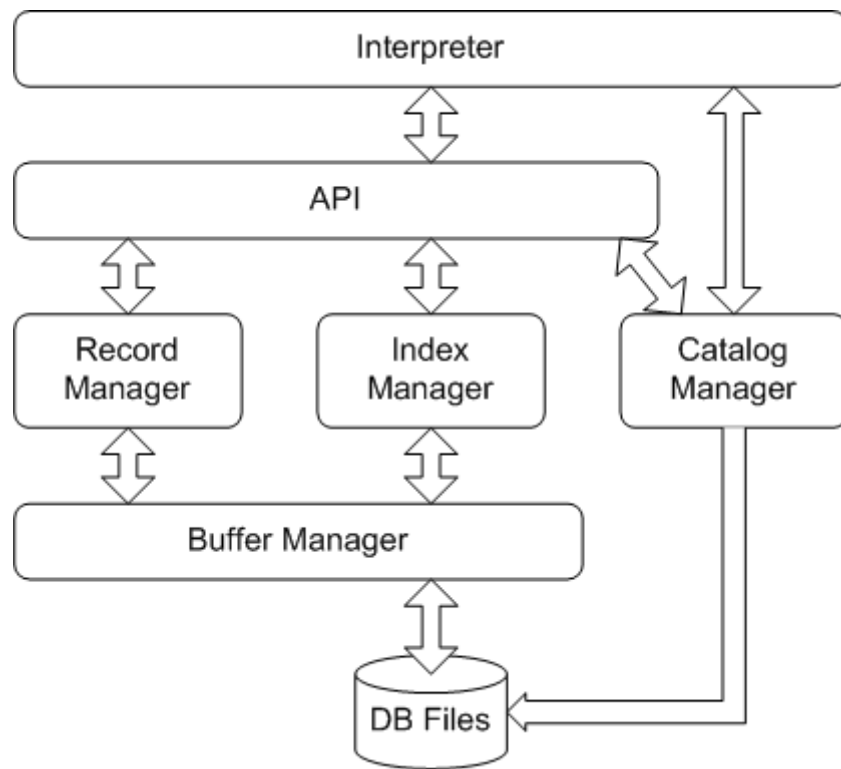
刘逸飞: Record Manager

沈尹祺: b+ tree 和 Index manager

杨林涛: Catalog Manager、Buffer Manager

2.2 MiniSql结构

MiniSql的基本结构如下图所示



Interpreter 模块负责接收并检查、解释用户在前端输入的SQL命令，识别命令的正确性，执行不同的子函数。**Interpreter**会调用 **API** 模块进行实际操作，调用**Catalog**模块修改表和索引的信息，并接收返回值。对于选择命令，需要进一步在前端输出结果。在检查命令时，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，也会给出相应提示。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**Interpreter** 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模块的功能为通过b+树建立索引，并实现对索引的管理，能够支持对于索引的单条插入与删除，查找等功能。**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。

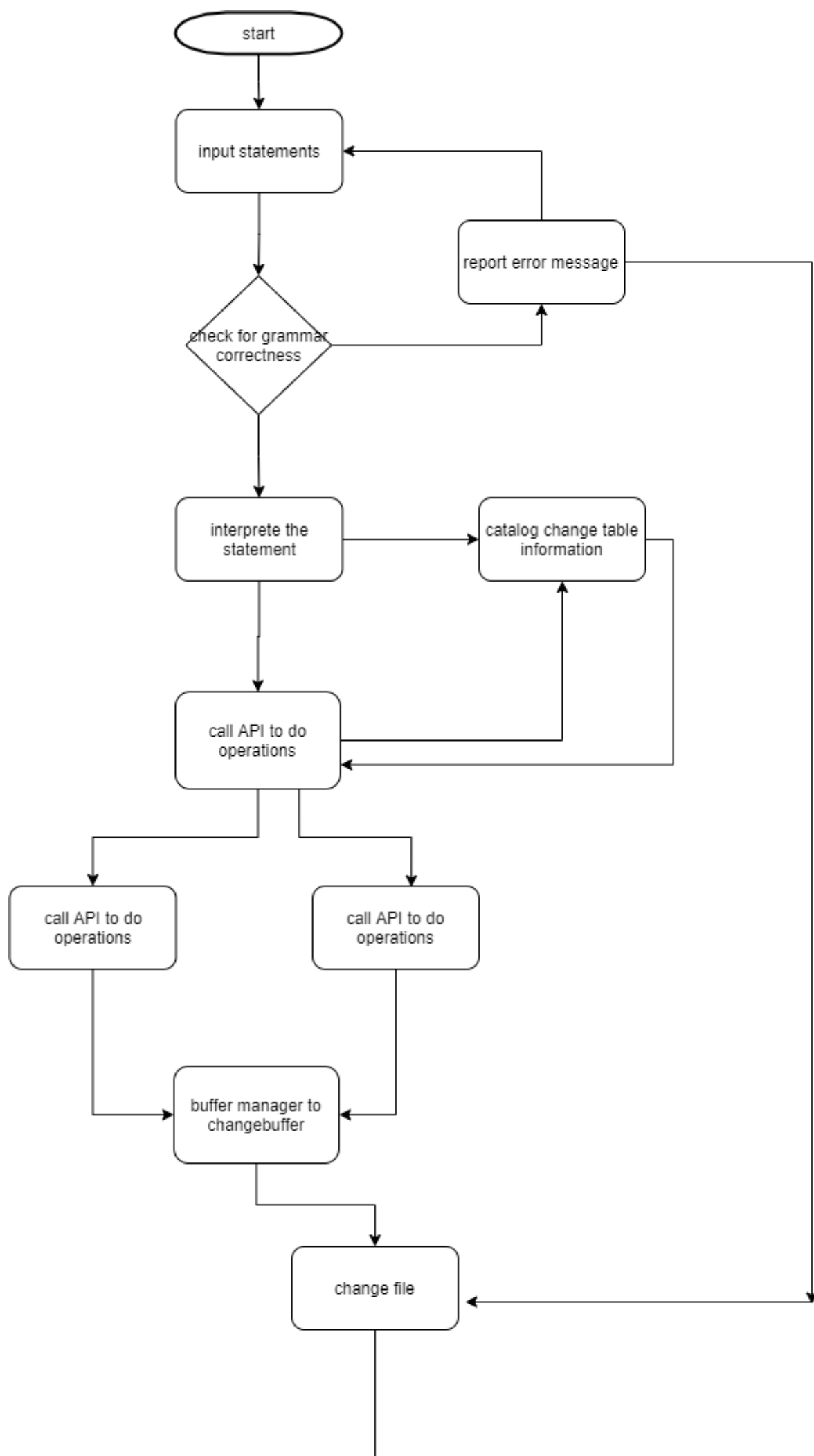
API模块负责各个子功能模块的相互接口。从解释器中接收非定义性命令和对应参数后，分别根据需要调用不同模块实现命令。例如，在进行查询时，**API** 首先获得解释器提供的查询参数（表名，属性名，范围限制），然后通过 **Catalog** 获得该表的索引定义，根据情况决定是否使用索引查询，最后交给 **Record** 模块处理。

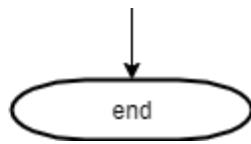
Buffer Manager模块负责缓冲区的管理，即数据块的写入与写出。其余子模块在有数据文件的访问寻求时，会向 **Buffer** 发出请求，**Buffer**直接访问数据文件，获取指定数据块，并将该数据块的读写权限教给该模块。除此之外，**Buffer** 还要能够实现缓冲区锁定，缓冲区替换等一系列功能。

Record Manager模块主要实现具体的操作。其中包括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。具体的操作的参数都由 **API** 传递给 **Record Manager**，**Record Manager**向**Buffer**请求相应的数据块进行读写来进行实现。

在实际设计过程中，我们首先设计了通用的数据结构与类（**base.h base.cpp**），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。

2.3 MiniSql 运行流程





用户输入指令后，先通过解释器判断指令的正确与否，如果指令不正确，会输出错误信息；如果指令正确，解释器会调用API实现操作，同时会调用Catalog修改表的信息。Catalog会改变定义的文件信息的值。API又会调用Record Manager和Index Manager来进行具体的操作。Record Manager和Index Manager会调用Buffer Manager来改变文件的信息。成功储存后一次指令操作就结束了

3 详细设计

3.1 数据结构

3.1.1 基础数据结构

在DBMS中，表的表示是十分重要的。就定义而言，表的构成非常复杂，其包括表的属性名称，表属性的数据类型，表的名称，表的元组数量等等。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
    string name[32]; //attribute name
    bool unique[32]; //unique
    int num;
};
```

在该结构中各个参数的定义：

参数名	作用
flag	属性数据类型(-1:int, 0:float, 0-255:char(n))
name	属性名
unique	各个属性是否unique
num	总的属性数目

- 索引结构

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

参数名	作用
num	索引数
location	索引在属性列中的位置
indexname	索引的名字

此外，我们还要考虑设计一个存储单个数据项的方式，因为这在查找等操作中是十分有用的。注意到我们事先是不知道数据的类型的，因此需要特殊的结构来存储数据，我们定义了如下的多态父类指针：

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

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

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

```

class Datac : public Data{
public:
    Datac(std::string c):x(c){
        flag = c.length();
        if(flag==0)
            flag = 1;
    };
    virtual ~Datac(){}
    std::string x;
};

```

首先，Data父类数据项唯一变量flag用于记录数据的类型，然后分别设计3个子类，存储不同类型的数据。在数据生成的时候，我们只需要把返回的子类指针作为父类指针保存；而在访问的时候，可以先通过flag来判断数据类型，在通过强制类型转换得到想要的子类指针，进而获取数据。

- where结构

```

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

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

```

定义了枚举WHERE:

值	0	1	2	3	4	5
符号	eq	leq	l	geq	g	neq
意义	=	<=	<	>=	>	<>(!)

- insertPos结构

在执行select, insert, delete等操作的时候，我们需要准确地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一级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。该结构的定义如下所示：

```
class insertPos {
public:
    int bufferNum; // the index of blocks in the memory
    int position;  // the index of tuple in the block
};
```

参数名	作用
bufferNum	该元组在缓冲区中区块的编号
position	该元组数据的起始地址在区块中的偏移地址

这个地址是数据在内存中的虚拟地址，但是可以通过BufferManager中的地址表来转换成相应的物理地址，两者的相互转换便于文件的读写操作的实现。

3.1.2.tuper类与Table类

- tuper类

表示表中的一个元组：

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

    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.
```

```
};
```

成员	作用
<code>vector<Data*> data</code>	存储元组中的数据项
<code>length()</code>	返回元组的长度
<code>addData()</code>	添加新数据项
<code>distuper()</code>	用于显示元组数据
<code>operator[]</code>	索引符号重载，找索引中特定的数据项

- Table类

```
class Table{
    friend class CataManager;
public:
    int blockNum;//total number of blocks occupied in data file;
    int dataSize(){ //size of a single tuper;
        int res = 0;
        for (int i = 0; i < attr.num;i++){
            switch (attr.flag[i]){
                case -1:res += sizeof(int); break;
                case 0:res += sizeof(float); break;
                default:res += attr.flag[i]+1; break; //多一位储
存'\0'
            }
        }
        return res;
    }
public:
    std::string Tname;
    Attribute attr;//number of attributes
public:
    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;
    }
```

```

//Construct with Tname and column.
Table(const Table& t);
~Table();
std::vector<tuper*> T;//pointers to each tuper
short primary;//the location of primary key. -1 means no
primary key.
Index index;
Attribute getattribute(){
    return attr;
}
void setindex(short i, std::string iname);
void dropindex(std::string iname);
void Copyindex(Index ind){
    index = ind;
}
Index Getindex(){
    return index;
}
void setprimary(int p){
    primary = p;
};//set the primary key
void disp();
std::string getname(){
    return Tname;
};
int getCsize() const{
    return attr.num;
};
int getRsize() const{
    return (int)T.size();
};
void addData(tuper* t);
};

```

成员	作用
blockNum	记录该表所占据的数据块数
vector<tuper*> T	存储表的成员数据
primary	记录主键的位置
index	索引信息
attr	属性信息

成员	作用
Tname	表名

- 异常类

由于程序的层次关系比较复杂，在程序运行的过程中，如果遇到问题，直接输出返回的形式显然是不可取的，我们定义了异常类，采用抛出异常的方式交由程序的顶层模块进行处理。不同类，不同操作，会有不同的异常。在通用类中，我们定义了表异常TableException:

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

而在之后将要介绍到的解释其中，定义了查询异常QueryException:

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

3.2 Interpreter

作为整个miniSQL程序的绝对前端，Interpreter主要是负责对用户输入的命令进行解析后采取相应的操作：如果错误返回输入错误信息，如果语法正确，则调用API或者Catalog模块的函数，进一步判断语句执行的可行性。

解释器的类设计代码如下所示：

```

class InterManager
{
public:
    string qs;
    void getQueryString();
    void normolize();

    bool Exec();
    void ExecDrop();
    void ExecCreate();
    void ExecCreateTable(int lastPos);
    void ExecCreateIndex(int lastPos);
    void ExecDelete();
    void ExecSelect();
    void ExecInsert();
    void ExecFile();
    void ExecQuit();
    inline int readElement(int pos);
    void interWhere(int& pos1, vector<int> &attrwhere,
vector<struct where> &w, struct Attribute A, class Table* t);
};

```

InterManager类中的成员变量qs存储当前正要处理的SQL语句，解释器首先通过getQueryString获得用户字符串，存入qs中。这里注意用户输入的SQL语句是非常灵活的，可以无限换行，无限空格，只要是正确的SQL语句，并且用分号结尾，解释器都能够正确解析。这里主要依靠标准化函数的处理和之后分析字符串的现金策略。

3.2.1 处理语句函数

- Normolize标准化函数

```

for (int i = 0; i < qs.length(); i++)
    if (isCharacter(qs[i]))
    {
        if (qs[i-1] != ' ') qs.insert(i++, " ");
        if (qs[i+1] != ' ') qs.insert(++i, " ");
    }

```

如果是特殊符号，在前后插入空格，方便之后进行语义分析。

- readElement函数

分析策略，以空格为分隔，一个“单词”一个“单词”地进行读取：

```
int InterManager::readElement(int pos)
{
    while (pos < qs.length() && qs[pos] != ' ') pos++;
    return pos;
}
```

3.2.2 实际操作语句

程序开始时候的执行界面：

```
Welcome to MiniSQL!
>>>_
```

- 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, , **primary key**(属性名));

可在类型名后面添加**unique**性质，主键约束可以选：

```
Welcome to MiniSQL!
>>>create table student2(
id int,
name char(12) unique,
score float,
primary key(id)
);
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。

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

- 删除索引

删除索引的格式如下：

drop index 索引名 on 表名；

若索引名不存在或者有其他语法错误，则不能正确执行。

```
>>>drop index stunameidx on student2;  
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。

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

- 插入记录

插入记录的格式如下：

insert into 表名 values (数据, 数据, ...);

其中数字直接输入，字符串用双引号或者单引号包含。

```
>>>insert into student2 values(1080105009,'name5009',61.0);  
0 Interpreter: successful insert!  
>>>
```

- 选择查询

```
>>>select name, score from student2 where score = 97;
name      score
name184 97
name211 97
name256 97
name266 97
name492 97
name614 97
name733 97
name748 97
name782 97
name794 97
name865 97
name887 97
name889 97
name977 97
Interpreter: successful select!
>>>
```

- 删除记录

```
>>>delete from student2 where score = 97;
Interpreter: successful delete!
```

之后再查寻记录，记录就没有了。

```
>>>select name, score from student2 where score = 97;
name      score
Interpreter: successful select!
>>>
```

- 执行脚本文件

execfile 文件名;

```
>>>execfile instruction2.txt;
```

```
Interpreter: successful execfile!
>>>
```

- 退出系统

quit;

输入quit;

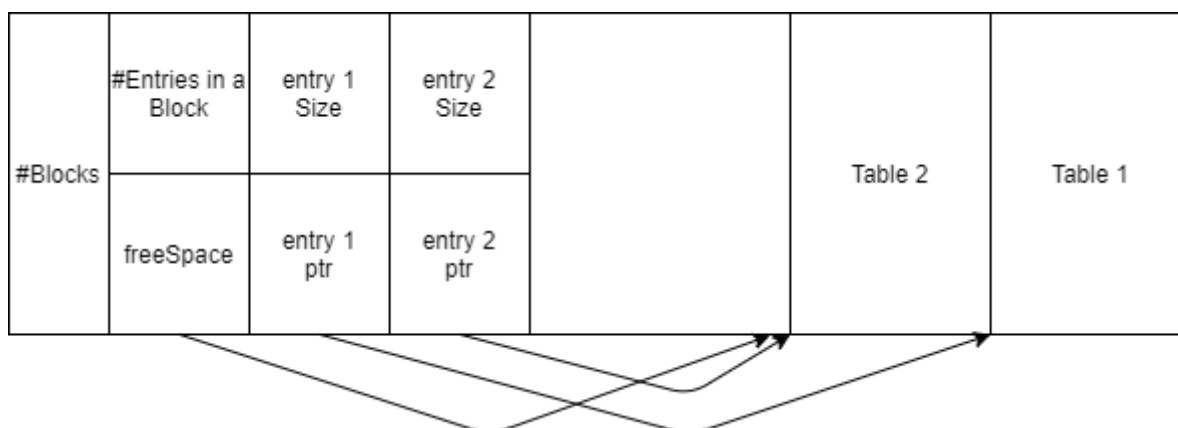
系统保存相关数据，然后退出。

3.3 Catalog Manager

```
class CataManager {
public:
    CataManager(){};
    void create_table(string s, Attribute atb, short primary, Index
index);
    bool hasTable(std::string s);/**检测重复定义，若返回1，则已经定义同名
表
    Table* getTable(std::string s);/**向文件读取并返回Table
    void create_index(std::string tname, std::string aname,
std::string iname);
    void drop_table(std::string t);
    void drop_index(std::string tname, std::string iname);
    void show_table(std::string tname);/**打印表的定义
    void changeblock(std::string tname, int bn);/**存储record的块变多了，表的定义（占了多少空间）也修改

private:/**内部函数
    void create_table_NoApi(string s, Attribute atb, short primary,
Index index, int bn);
    void drop_table_NoApi(std::string t);
    void LoadInfo(string s, Attribute atb, short primary, Index
index, char* head,int bn);
    char* FindPtrPosition(string s, char*& base, short& recIdx);
    short FindLocation(string aname, string tname);
};
```

我们在这里使用了SlottedPage来储存表的定义。



这里是我们储存的一张表的具体内容，**bn**是为储存表的**record**占了多少个块。

bn	attr.num	attr1		primary Key	index.num	index1	
----	----------	-------	--	----------------	-----------	--------	--

我们将所有的表放到Table_Defiintion文件里，我们先计算占用的空间然后从尾向头插入。

我们在**create_table()**，**create_index()**时会先计算可用空间，如果不够就申请新的block。

在**drop_table()**，**drop_index()**时会立刻缩小占用空间以保证Definition块内数据定义的一致性。

3.4 Index Manager

该模块的功能为通过b+树建立索引，并实现对索引的管理，能够支持对于索引的单条插入与删除，查找等功能。该模块中建立的b+树的结点大小为8k字节，能够较好且较为高效的实现其功能。

IndexManager 设计如下：

```
class IndexManager{
public:
    IndexManager(){};
    void Establish(string file);//建立索引
    void Insert(const string &file, Data* key, int Addr);//对现有索引
    进行插入
    void Delete(string file, Data* key);//对现有索引进行删除
    int Find(string file, Data* key);//对现有的索引进行等值查找
    void Drop(string file);//删除索引
    ~IndexManager(){};
};
```

下面逐个介绍各个函数的作用

3.4.1 建立索引Establish(string file)

根据传入的文件名建立名为“file”.index的b+树

3.4.2 删除索引Drop(string file)

根据传入的文件名删除名为“file”.index的b+树

3.4.3 对现有索引进行插入 Insert(const string &file, Data* key, int Addr)

将key值以及该key对应的地址插入名为“file”.index的b+树中。首先进行检查，若没有该b+树，或者b+树中已有该key值，则会返回错误信息，否则，将key与addr插入b+树中。最后调用BufferManager将改变的block重新写入buffer中。

3.4.4 对现有索引进行删除 Delete(string file, Data* key)

传入待删除的key值，调用b+树进行遍历与查找，若无法找到则会返回错误信息，若能找到则会通过b+树进行删除。

3.3.5 对现有的索引进行等值查找 Find(string file, Data* key)

传入待查找的key值，通过b+树进行遍历与查找，若能找到，则返回该key的addr，若不能找到，返回-1。

3.5 API

API模块是整个系统的顶层调度模块，在整个系统中起到串联前端和后端的作用，在于上层模块的对接中，API主要是相应Interpreter各种请求，比如说创建表，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进行的。

3.5.1 主要功能

- 创建表

每次建立表的时候，都要调用RecordManager和IndexManager，通过RecordManager创建一个表，并且通过IndexManager根据表中的主键和Unique属性创建相应的索引。

- 删除表

每次删除表的时候，都要调用RecordManager以及IndexManager，分别清空缓存区中所有与相应表有关系的数据，并且需要把这个表上建立的所有索引全部删除，然后再把硬盘上的相关文件全部删除。

- 数据的插入

插入数据的时候，直接调用RecordManager，表的信息传过去。

- 数据的删除

删除数据时，直接调用RecordManager，返回相应的信息给用户界面。

- 数据的查找

查找数据也调用RecordManager即可，然后会得到一个表作为一个返回值，这个表中存放着查询得到的记录，用table类自带的disp()成员函数能够在用户界面端输出查询到的元组信息。

- 创建索引以及删除索引

这两个与索引直接相关的函数也直接调用了IndexManager的相应函数接口，需要注意的是，创建索引的时候，API创建索引的名字是不同的，用户创建的索引命名一律用"索引名"+"index"，而创建表的时候由primary和unique自动生成的索引命名格式是"表名"+"属性序号"+"index"，之后处理的时候，便于进行区分。

3.5.2 接口设计

API的接口定义如下：

```
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);
void Insert(Table& tableIn, tuple& singleTuper);
bool DropTable(Table& tableIn);
void DropIndex(Table& tableIn, int attr);
bool CreateTable(Table& tableIn);
bool CreateIndex(Table& tableIn, int attr);
```

3.6 Buffer Manager

buffer manager的类定义如下

```
class BufferManager {
public:
    buffer bufferBlock[MAXBLOCKNUM];
    BufferManager() {
        /*initialize linkedlist, for LRU
        head = new s_LinkedList;
        tail = new s_LinkedList;
        head->next = tail;
        head->prev = NULL;
        tail->prev = head;
        tail->next = NULL;
        };
    ~BufferManager() {
        /*delete linked list
        LinkedList prev = head;
        for (LinkedList ptr = head->next; ptr; prev = ptr, ptr = ptr-
>next) free(prev);
        free(prev);
        for (int i = 0; i < BlockMaxNum; i++) flashBack(i);
        /*全部向磁盘输出
    };
    /*Method
    int GiveMeABlock(string filename, int blockOffset);/*其他模块向
    BufferManager索要空间的统一接口
    class insertPos getInsertPosition(Table& tableinfor);/*向
    RecordManager返回课插入单个元组的内存位置。
    void writeBlock(int bufferNum);/*标记Dirty, 使其可通过LRU写回磁盘
    void useBlock(int bufferNum);/*将一块已分配的内存区域进入到LRU链表中
    /*下面3个是GiveMeABlock的实际调用函数
    int getbufferNum(string filename, int blockOffset);
    int getIfIsInBuffer(string filename, int blockOffset);
    int getEmptyBuffer();
    void readBlock(string filename, int blockOffset, int
    bufferNum);/*硬盘上原本有内容的话调用它写到缓冲区

public:
```



```

    void setInvalid(string filename);

private:
    void InsertLinkedList(int i);
    void flashBack(int bufferNum);
    int getEmptyBufferExcept(string filename);
    void scanIn(Table tableinfo);
    int addBlockInFile(Table& tableinfor);
    int addBlockInFile(Index& indexinfor);
    void InitializeRecordBlock(int bufferNum, Table& tableinfor);
    LinkList head, tail;
    friend class index;
};

```

我们使用LRU算法来排列用到的Block，这是由链表实施的。

```

void BufferManager::useBlock(int bufferNum) {
    /*to perform LRU algo
    /*add the used index to the front of the linked list
    /*may use mutiple, find and place in the front of the linklist
    LinkList tmp;
    for (tmp = tail->prev; tmp != head; tmp = tmp->prev)
        if (tmp->index == bufferNum) {
            /*delete
            tmp->prev->next = tmp->next;
            tmp->next->prev = tmp->prev;
            /*reInsert
            tmp->prev = head;
            tmp->next = head->next;
            head->next->prev = tmp;
            head->next = tmp;
            break;
        }
    if (tmp == head) InsertLinkedList(bufferNum);
}

```

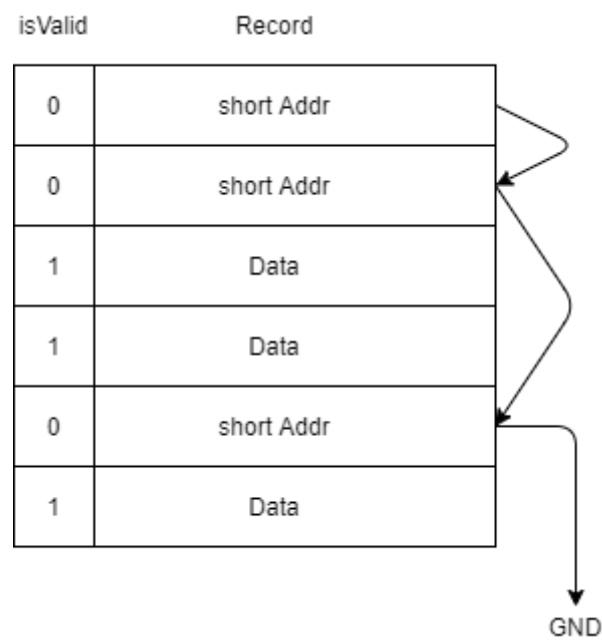
这里将最先用到的块放到链表的前方，如果缓冲区满，则将链表最后非尾节点对应的空间释放。

我们对磁盘文件中的文件定义以下地址：

$$Addr = BlockOffset \times BLOCKSIZE + inBlockAddr \text{ (Byte)}$$

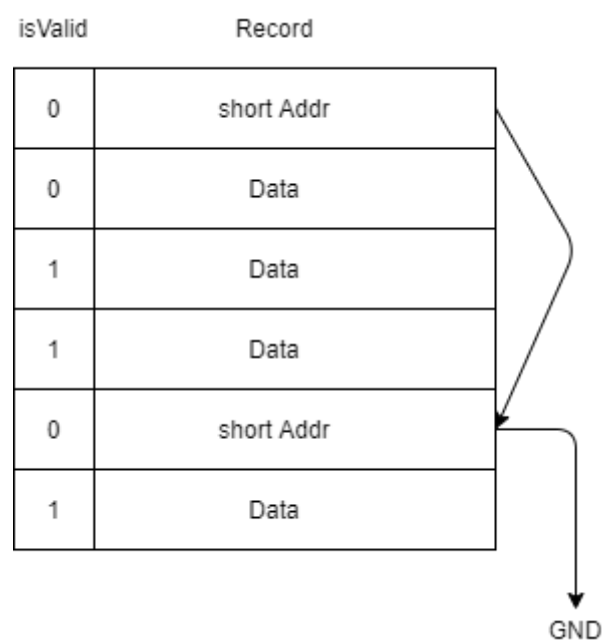
我们用 `flashBack()` 来写入磁盘，在 `main()` 结束时，`bufferManager` 会自动析构向硬盘写入所有内容。

我们与 `RecordManager` 商定用链表串联起文件中被删除的空处：



由于 `BLOCKSIZE` 是在 `short` 范围内，我们将硬盘地址以 `short` 类型存储在空的条目内。

我们返回哑头节点的后继，当 `RecordManager` 向 `BufferManager` 索要空间插入的时候。



其他方法：

GiveMeABlock: 用户将文件名和块的偏移量传给此函数。这个函数首先在被分配的给他的缓冲区中，先寻找这个filename下面的Block offset对应的block。如果找不到的话，就直接去磁盘里找，磁盘里找不到返回空块给用户使用。

writeBlock: 是标记这个block的dirty位，后面到flashback的时候可以把它给血回去。

useBlock: 把这个block进入我们的LRU队列里头。这样子的话，我们就可以统一管理他们的输入和输出，以提高输入和输出的效率。

getbufferNum: GiveMeABlock调用这个函数返回可用的块。GiveMeABlock同时也将改Block送入LRU队列。

getIfIsInBuffer: 在缓冲区里找符合文件名和偏移量的块。

getEmptyBuffer: 磁盘里找不到返回空块给用户使用。

readBlock: 文件中已有内容，写入bufferBlock，这样就可以实现数据库重启也可以保存表的定义和数据以及index。

3.7 Record Manager

Record Manager负责管理记录表中数据的数据文件,能够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。

Record Manager主要包括以下函数

```
class RecordManager {
public:
    RecordManager(class BufferManager* bf) : buf_ptr(bf) {}
    ~RecordManager();
    bool issatisfied(Table& tableInfor, tuper& row, vector<int>
mask, vector<where> w);
    Table Select(Table& tableIn, vector<int> attrSelect,
vector<int> mask, vector<where>& w);
    Table Select(Table& tableIn, vector<int> attrSelect);
    void CreateIndexCatalog(string iname, int attr, string tname,
Table &tableIn);
    int FindwithIndex(Table& tableIn, tuper& row, int mask);
    tuper* Char2Tuper(Table& tableIn, char* stringRow);
    void InsertwithIndex(Table& tableIn, tuper& singleTuper);
    char* Tuper2Char(Table& tableIn, tuper& singleTuper);
    int Delete(Table& tableIn, vector<int> mask, vector<where> w);
```

```

bool DropTable(Table& tableIn);
bool CreateTable(Table& tableIn);
Table SelectProject(Table& tableIn, vector<int> attrSelect);
bool UNIQUE(Table& tableinfo, where w, int loca);

private:
    RecordManager() {}
    class BufferManager* buf_ptr;
};

```

Record Manager 需要使用 buffer manager，所以在类中需要包括 `buf_ptr` 变量来调用 buffer manager。

Record Manager 进行的操作及其原理如下所示

3.7.1 创建与删除

创建表与删除表只需要建立或删除对应的文件即可，不需要进行其他额外的操作。

3.7.2 插入

插入一条新记录时，需要对表属性的 `unique` 约束进行检查也。插入既可以通过遍历表格所有记录来实现，又可以通过检查索引文件来进行实现。为了速度能够更快，我们选择通过查找索引来进行插入。插入时会遍历所有索引，如果有重复值的索引，提示这个元素冗余了，无法插入。如果没有，则可以插入。插入通过调取 `buffer` 的对应函数获得插入位置来进行插入。同时也对现存的索引都进行插入。

3.7.3 删除

删除记录时，首先遍历所有的记录，寻找是否有相应条件的记录，进行删除。同时还要删除掉他们对应的索引。如果没有记录，则不进行操作。

3.7.4 查询

查询和删除记录类似，遍历所有的记录来找到对应的记录，并进行删除。由于索引的结构限制，我们无法实现范围的索引查询。因为除了等值的查询外还有六种关系的查询，如果查询仍然使用等值的索引查询，那么其实和遍历记录的消耗也相差不大。因此在查询部分我们没有使用索引。而是通过遍历所有元素，检查符合条件的记录并添加进表，再将这张表返回

给API的方法来实现。

4 程序展示

4.1 create table:

输入:

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

输出结果:

```
Interpreter: successful create!
```

4.2 insert

输入:

```
insert into student2 values(1080101001,'name1001',84.5);  
insert into student2 values(1080101002,'name1002',88.0);  
insert into student2 values(1080101003,'name1003',90.0);  
insert into student2 values(1080101004,'name1004',88.5);  
insert into student2 values(1080101005,'name1005',74.5);
```

输出结果:

```
Interpreter: successful insert!  
Interpreter: successful insert!  
Interpreter: successful insert!  
Interpreter: successful insert!  
Interpreter: successful insert!
```

4.3 select

输入:

```
select * from student2;
```

输出结果:

```
id      name      score
1080101001  name1001    84.5
1080101002  name1002    88
1080101003  name1003    90
1080101004  name1004    88.5
1080101005  name1005    74.5
Interpreter: successful select!
```

输入:

```
select * from student2 where name = "name1002" and score = 88;
```

输出结果:

```
id      name      score
1080101002  name1002    88
Interpreter: successful select!
```

4.4 delete

输入:

```
delete from student2 where name = "name1001";
```

输出:

```
Interpreter: successful delete!
```

此时再输入:

```
select * from student2;
```

输出：

```
id      name      score
1080101002  name1002      88
1080101003  name1003      90
1080101004  name1004      88.5
1080101005  name1005      74.5
Interpreter: successful select!
```

可见能够成功地删除

4.5 create index




输入：

```
create index stunameidx on student2 ( name );
```

输出：

```
Interpreter: successful create!
```

此时文件夹内多出了一个index文件：

 student20.index	2021/6/27 11:44	INDEX 文件	16 KB
 student21.index	2021/6/27 11:44	INDEX 文件	16 KB
 stunameidx.index	2021/6/27 11:49	INDEX 文件	16 KB

4.6 drop index




输入：

```
drop index stunameidx on student2;
```

输出：

```
Interpreter: successful drop!
```

此时文件夹内已经没有了相应的文件：

 student20.index	2021/6/27 11:44	INDEX 文件	16 KB
 student21.index	2021/6/27 11:44	INDEX 文件	16 KB
 Table_Definition	2021/6/27 11:41	文件	0 KB

4.7 drop table

输入：

```
drop table student2;
```

输出：

```
Interpreter: successful drop!
```

此时文件夹内相应的.index文件与.table文件已经没有了。

4.8 测试总结

经过以上的测试，我们发现表的插入、删除、查找，索引和表的创建和删除都没有问题

5 总结

在本次实验中，我们将数据库系统理论与实践结合，成功地设计了一种数据库软件，能够对数据进行简单的处理，对记录进行增加、删除、查找、建立索引等。通过上手实践，我们能够更好理解了数据库设计的原则与技巧，对老师上课讲授的概念有了更深的体会。同时，我们的编码能力和调试技巧都得到了锻炼。但是我们的数据库还有一些不足，例如在数据量特别大时会出现错误等情况。我们将在未来的学习中，逐渐去学习更加高深的数据库设计理论，并逐渐锻炼自己的编程能力，来解决这些问题。