MiniSQL 设计报告

一、总体开发思路

1、任务概述

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。

关于数据的定义，数据类型要求支持三种基本数据类型：int，char(n)，float，其中 char(n) 满足 1 <= n <= 255 。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。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）。

关于命令方式，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。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。

语法要求：

创建表：create table 表名(
列名,类型
primary key(列名));

删除表：drop table 表名;

创建索引：create index 索引名 on 表名(列名);

删除索引：drop index 索引名;

选择语句：select * from 表名; 或 select * from 表名 from 条件

插入语句：insert into 表名 values (值 1,值 2,…,值 n);

删除语句：delete from 表名 或 delete from 表名 where 条件

退出语句：quit;

执行 SQL 脚本文件：testcfile 文件名;

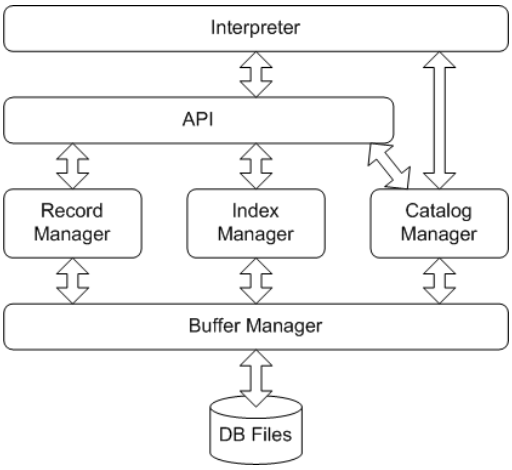
其中条件的格式为：列 op 值 and 列 op 值 and …

op 为=,<,>,<>,<=,>=之一

2、总体框架

在设计模块之前，我们创建了 base 模块，其中定义了所有数据，属性，表的类型，方便了所有模块的实现。

MiniSQL 总共分为六个模块设计：Interpreter，API，Record Manager，Index Manager，Catalog Manager，Buffer Manager。



Interpreter 模块与用户进行交互，接受用户输入的 SQL 命令，并对命令进行解析，根据解析的结果对调用不同的模块（API，Catalog），同时将返回的结果回馈给用户。其中返回结果有四个方面：选择命令返回选择结果，成功执行命令时的提示，命令具有语法问题的提示以及数据库错误的提示。

API 模块接收 Interpreter 解析命令后提供的参数，如表的属性，选择条件等，根据需要调用不同的模块（Record，Index，Catalog），如果是查询指令，需要先从 Catalog 中获取表的属性是否属于主键，如果是主键，则调用 Index 查询，反之则调用 Record 查询。

Record 模块接收 API 提供的参数，实现表中记录的查询，插入，删除功能。通过调用 Buffer 模块读取数据

Index 模块接收 API 提供的参数，实现表中索引的创建，查询，插入，删除。其中索引通过 B+树来储存。通过调用 Buffer 模块读取数据。

Catalog 模块同时接收 API 和 Interpreter 提供的参数，实现表，属性，索引的定义以及判断查询方式的功能。当接收 Interpreter 提供的参数时，Catalog 根据表的信息建立表，并将表和索引的定义信息通过 Buffer 储存在文件中，当接收 API 提供的参数时，Catalog 根据属性是否属于主键确定查询语句的查询方式，返回信息给 API。

Buffer 模块接收来自 Catalog，Record 和 Index 的参数，主要功能是将文件中的记录信息返回。Buffer 作为缓冲区完成了数据的暂时存储，加快了查询速度。

二、Base 模块

1.数据结构

为了实现表的表示，我们分别对于表的属性的数据类型，属性名称，表的名称，表的元组数量进行设计。

（1）数据类型：

由于有整型，双精度型和字符串型三种数据类型，我们建立基类和继承类来构建数据类型。

```
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;
    };
    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;
};
```

（2）表的属性

设置最多有 32 个属性，存储属性的名称，数据类型和是否唯一和总数。

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

（3）索引

设置最多有十个索引，存储索引的名称，在属性列的位置和总数。

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

（4）元组类

按照顺序存储元组中所有的数据，与属性顺序对应。

```
class tuper{
public:
    std::vector<Data*> data;
};
```

元组中还含有一些成员函数，方便对于元组中数据的插入和读取。

```
int length() const
作用：返回元组数据个数。
void addData(Data* d)
作用：添加新数据到元组中
Data* operator[](unsigned short i)
作用：返回具体某个属性的数据
void disptuper()
作用：输出整个元组。
```

（5）表类

存储表的名称，索引，属性名称，元组，主键在属性中的位置以及在文件中的块位置

```
class Table{
public:
    int blockNum;
    std::vector<tuper*> T;
    short primary;
    Index index;
private:
    std::string Tname;
    Attribute attr;
};
```

2.谓语结构

我们统一定义谓语中的词，方便不同模块间的参数传递。

其中 eq-> =，leq-> <= ，l-> <，geq-> >=，g-> >，neg-> <>。

```
typedef enum{
    eq,leq,l,geq,g,neq} WHERE;
struct where{
    Data* d;
    WHERE flag;
};
```

3.异常类结构

我们定义异常类，将程序运行过程中的问题交给 Interpreter 输出。

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

```
    std::string text;
};
```

三、Interpreter 模块

1.模块概述

InterPreter 模块又称解释器模块，它相当于整个 MiniSql 的前端，模块的角色定位就是接受用户的指令，然后对其进行规范化处理然后进一步的解读。解读的基本流程可以理解为：首先进行语法检查，如果不符合语法，则抛出合理的提示提醒用户语法错误；如果语法检查通过那么解释器会针对用户的输入进行不同的任务，分别调用 API 或者 Catalog 中的功能函数。进而层层调用直到底部已完成用户的输入指令。

2.主要功能

Main 函数不断的执行读入用户输入，每一次输入通过 “；”结束，每次读入一个语句，解释器首先检查其语法问题，如果有，则抛出提示“invalid query format”.如果没有则进一步解读：针对输入语句的第一个 string 分流至不同的功能，大致分为：Select; Drop; Insert; show table; create table ; create index ; Exit ; testfile 等;主函数不断地读入用户输入以系统对输入的返回值作为 while（re）的条件，当且仅当用户输入 Exit，系统 return 0 ； 主函数结束，软件退出。

3.接口设计

```
class InterManager{
private:
    std::string qs;
public:
    int EXEC();
    //解释器最前端
    //解释器 解读到用户输入第一个字段后 分流进行进一步解读
    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 Run(int pos);
```

```
    //针对一条以；结尾的语句，找到两个空格之间的字段 比如 select * from 从头开始
    //执行一次 Run（0），那么返回的是第一个空格所在位置 6 ，那么[0,6]就是 select 字段 。
    void interwhere(int& pos1, std::vector<int> &attrwhere, std::vector<where> &w, Attribute A,
                    Table* t);

    void GetQuery();
    //获得一条语句
    void Normolize();
    //对语句进行规范化 比如 select   * from   取消多余的空格 变成 select * from ；
};
```

4.设计思路

成员变量 qs 贯穿整个解释器，用于存储待处理的字符串。在 main 函数中，解释器通过 Getquery()获得用户输入的字符串，存入 qs 中，然后调用 EXEC 进行解析执行，首先调用 Normolize 对 qs 进行标准化处理，然后根据不同的 qs 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值确定是否要继续读取用户输入。

5.语法及功能说明

在程序开始运行时，出现下面的界面：

```
MiniSQL By Zhb & Hdc!
>>>
```

- 创建表

```
MiniSQL By Zhb & Hdc!
>>>create table t1(age int) ;
Interpreter : successful create!
>>>
```

- 创建索引

```
MiniSQL By Zhb & Hdc!
>>>create table t1(age int) ;
Interpreter : successful create!
>>>create index T1 on t1(age) ;
This attribute is not unique!
>>>
```

因为在 t1 中 age 不是 unique 的，所以利用 age 进行索引的创建会失败。

- **插入记录**

```
MiniSQL By Zhb & Hdc!
>>>create table t1(age int) ;
Interpreter : successful create!
>>>create index T1 on t1(age) ;
This attribute is not unique!
>>>insert into t1 values(111);
>>>
```

正确的插入记录，插入成功则如上图所示没有返回值。

- **选择查询**

```
>>>select * from t1;
age
111
Interpreter: successful select!
>>>
```

根据选择条件查询表中已有的数据，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 后面跟 and 连接的判断词，其中包括等值和不等值查询，能够接受> ;<;=;>=;<=;<> ; 最后返回输出的表。

- **删除记录**

删除操作用于删除表中满足指定条件的元祖，其输入格式为：

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

```
>>>select * from t1;
age
111
Interpreter: successful select!
>>>delete from t1 where age =111 ;
Interpreter : successful delete!
>>>select * from t1;
age
Interpreter: successful select!
>>>
```

- **执行脚本文件：**

```
MiniSQL By Zhb & Hdc!
>>>testfile:0_create.txt;
Interpreter : successful create!
Interpreter : successful create!
) ;
ERROR : invalid query format!
>>>
```

除了一些基础的操作意外，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，以分号为标识进行分割，分别存入 qs 变量当中，然后循环调用解析函数分别进行解释执行将每一个命令的返回信息打印在前端上。（这里的 0_create.txt 中内容如下图所示）

```
create table City (
  ID int,
  Name char(35) unique,
  CountryCode char(3),
  District char(20),
  Population int,
  primary key(ID)
);

create table CountryLanguage (
  CountryCode char(3),
  Language char(30),
  IsOfficial char(1),
  Percentage float,
  primary key(CountryCode)
);
```

• 退出系统

```
MiniSQL By Zhb & Hdc!
>>>exit;
```

在执行完一些命令后，用户可以通过输入 exit；退出系统，首先会将缓冲区中的文件强制写入文件，然后再结束程序。

• 显示表的信息

在解释器模块中我们新增了一个 show table 功能，用于显示当前表的属性。

```
MiniSQL By Zhb & Hdc!
>>>show table City ;
City:
ID int unique primary key
Name char(35) unique
CountryCode char(3)
District char(20)
Population int
index: ID(ID)
>>>
```

语法为 show table 表名；他会返回该表的所有属性信息。

四、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个表或多条记录时，就需要调用 RecordManager 来管理表中的数据，该模块负责处理数据的插入，删除，以及表的建立和删除，以及从表中读取数据。
之所以我们把 RecordManager 定位为一个桥梁模块，是因为这个模块上承 API 和解释器，下接底层 BufferManager，起到了一个桥梁的作用，前端解释器接到指令解析执行，调用 API，API 又调用 RecordManager 中的具体函数对该指令进行响应。所以 Record 模块在整个处理的过程中就处于中间过程。

2. 主要功能

1.创建和删除表：每次建立或者删除表的时候，都要调用 RecordManager，删除相关的数据，并且把相对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 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，也就是把记录写入到内存中相对应的位置，再由 Buffer 完成数据的持久化。

3.数据的删除：删除数据时，采用懒惰删除法，首先判断这个要删除的键是否为主键，如果有那么利用 index 进行搜索，找到相应的数据，否则就遍历所有的数据判断是否有要删除的数据，然后将其标记为“被删除状态“，表征该记录已经被删除。

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符合查找的条件，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，最后返回最终生成的表。

3. 接口设计

```
class RecordManager
{
public:
    RecordManager(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 SelectWithIndex(Table& tableIn, vector<int>attrSelect, vector<int>mask,
vector<where>& w);
    Table Select(Table& tableIn, vector<int>attrSelect);
    int FindWithIndex(Table& tableIn, tuper& row, int mask);
    void Insert(Table& tableIn, tuper& singleTuper);
    void InsertWithIndex(Table& tableIn, tuper& singleTuper);
    char* MakeAstring(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);

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

4. 设计思路

设计 RecorManager 是为了让程序可以将数据记录到文件当中，并且下次可以根据表的名字读取出相应的数据。考虑到如果在文件中采取彻底删除的方式，那么在一个大文件中删除前排（比如说第一条记录），那么后面的记录前移将是一个代价很高的操作，所以我们选择惰性删除，对于要删除的记录，我们只是将其标记为已经被删除，这样的话就不需要将被删除记录后面的记录依次前移。

对于第一个问题，我们采用的方法就是再书存储转化的时候直接拷贝。我们用 char 的方法来表示最后的数据，把其他类型的数据直接拷贝到长度相对应的字符串中，对于字符串，每条记录都用所定义的最大长度进行转化，这样的话每一条记录都是定长的，这样记录就很整齐的堆放在文件当中，方便我们的读取与查找。

还有一个小问题，在每一条记录的第一位是标记位，标记该条记录的状态，对于 getInsertPosition 函数找到数据块中可用的空位值，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个位置。

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，第二个输入是选择哪些属性，要选择的属性的位置（以整数的形式放在一个容器当中），第三个是 where 中涉及的属性（也以整数的形

式存放在第一个容器当中），最后一个是指选择的条件，也就是 where 后面的条件。

在实现 Select 时，首先判断 where 后面的条件是否为空，如果是空的，那么直接调用没有 where 的 Select 直接将所有的记录全部顺序遍历，然后通过投影选取出所需要的几个属性，如果有 where 条件，又要先判断条件所涉及属性是否为 primary key，如果是，则调用 Select with Index，通过 index 查找出所需要的记录，如果不是 index，则通过暴力搜索所有记录，一一对比是否满足所有 where 的筛选条件。

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，首先判断表中是否存在主键，以及有没有 unique 属性的键。如果有这些键，就要先进行防重复检查。所以我们给出的解决方案是首先以要插入的信息的主键作为 where 条件进行搜索，如果搜索到值则代表这个值是存在的，那么就会抛出插入数据冗余，插入失败。如果查找不到，则可以插入。如果没有主键，也没有 unique 的键那么就直接插入。注意：如果有两个及以上主键：就要分别进行搜索以确保各个键都不重复。

3.Delete 语句：

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

实现方法：

和 Select 一样，等同于根据后面的 where 先 select 出一些满足条件的列表，不一样的是这次不对他进行打印，而是将这些元祖标记为删除，也就是懒惰删除。

4.Create Table 语句：

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

实现方法：

Create Table 就是创建一个以表名+“.table“为文件名的数据文件。如果有数据插入在调用 insert 函数。

5.Drop Table 语句：

函数原型：bool DropTable(Table& tableIn);

实现方法：删除一个表，首先遍历缓冲区中所有的这个表的所有数据块全部赋值为 Invalid，然后再调用删除硬盘上有关指定表的函数。

五、Index 模块

1、设计思路

Index 模块的功能是建立，删除索引以及根据索引查询数据，提高查询速度。索引通过 B+树实现。我们建立 IndexManager 来对索引功能进行管理，建立 BpTree 实现 B+树的初始化、插入、删除、查找工作。

IndexManager 类

```
class IndexManager{
public:
    IndexManager(){};
    void Establish(string file);
    作用：建立索引文件
```

```
    void Insert(string file, Data* key, int addr);
    作用：向 B+树中插入数据
    int Search(string file, Data* key);
    作用：根据给定的数据进行查找
    void Delete(string file);
    作用：删除索引
};
```

BpTree 类

```
class BpTree {
public:
    int NodeNumber;
private:
    int type;
    string name;
    const int keylength[3]={4,4,10};
public:
    BpTree(string bname);
    作用：建立索引文件
    virtual ~BpTree(){};
    void Initialize(Data* key, int addr);
    作用：初始化索引文件，即插入第一个数据
    int Search(Data* key);
    作用：查询数据
    void Insert(Data* key, int addr);
    作用：插入数据
};
```

分别记录索引的个数，结点类型（叶结点/内结点），索引文件名，数据长度

BpTree 类中还有一些内部函数，方便插入函数的编写

```
void split_leaf(char* node,char* node1);
作用：分裂叶结点
void split_internal(char* node,char* node1, int son1,int son2);
作用：分裂内结点
void insert_internal(char* node,int son1,int son2);
作用：向内结点中插入数据
int find_leaf(Data* key);
作用：找到数据所在的叶结点的块位置
void Initialize(char* node);
作用：初始化结点
```

2.接口设计

与 API 的接口

```
void Establish(string file);
```

作用：建立索引文件
void Insert(string file, Data* key, int addr);
作用：向 B+树中插入数据
int Search(string file, Data* key);
作用：根据给定的数据进行查找
void Delete(string file);
作用：删除索引

六、Catalog 模块

1、设计思路

Catalog 模块的功能是管理所有模式信息，包括表，属性，索引的定义等。我们将这个信息全部放在同一个文件里。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。

2、接口设计

与 Interpreter 的的接口

void create_table(string s, Attribute atb, short primary, Index index);
作用：创建表
bool hasTable(std::string s);
作用：判断是否存在表
Table* getTable(std::string s);
作用：得到一个指定名字的空表
void create_index(std::string tname, std::string aname, std::string iname);
作用：在指定属性上创建索引
void drop_table(std::string t);
作用：删除表
void drop_index(std::string tname, std::string iname);
作用：删除索引
void show_table(std::string tname);
作用：显示表的所有模式信息

与 Record 模块的接口

void changeblock(std::string tname, int bn);
作用：改变表所在的数据块

七、Buffer 模块

1.模块概述：

Buffer Manager 是一个用于管理缓冲区的模块，也就是说我们要用软件来模拟并实现这

个硬件模块的功能。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和写入都是直接再内存中的缓冲区进行，对于任何需要创建或者是添加信息的操作，首先在缓冲区中操作，然后在通过 Write Back()将缓冲区中变动过的块写回到文件当中。

BufferManager 的底层是一个 MAXBLOCKNUM 大小的一个内存池。下图是 Buffer 类的实现方式。

```
class buffer{
public:
	bool isused;
	bool isWritten;
	string filename;
	int blockOffset;	//block offset in file, indicate position in file
	int recent_time; //show the recent visit time
	char values[BLOCKSIZE + 1];	//一个 buffer 开数组

	buffer()
	{
		initialize();
	}
	i
	void initialize(){
		isWritten = 0;
		isused = 0;
		filename = "NULL";
		blockOffset = 0;
		LRUvalue = 0;
		memset(values,EMPTY,BLOCKSIZE);	// 清空
		values[BLOCKSIZE] = '\0';
	}

	string getvalues(int pos, int pos1){
		string result = "";
		if (pos >= 0 && pos <= pos1 && pos1 <= BLOCKSIZE)
			for (int i = pos; i < pos1; i++)
				result += values[i]; // 把内存里的东西取出来
		return result;
	}

	char getvalues(int pos){
		if (pos >= 0 && pos <= BLOCKSIZE)
			return values[pos];
		return '\0';
	}
};
```

每一个 buffer 块自带初始化函数 initialize(),然后对外接口 getvalues(int pos ,int pos1) ;返回第 pos bytes 和 pos1 bytes 之间的值 。

处理软件和和数据文件的交互时，每次会从用户处理获取要处理的表，然后通过 Buffer Manager 将指定的指定偏移量的数据读入到内存当中。而 Buffer 模块新开辟了一块特殊的缓冲区空间，可以通过 BufferNum 进而访问特定的缓冲区空间进行操作。

2.主要功能

1.当作整个程序和硬盘数据的交互渠道，程序中一切所需新建或修改硬盘上的文件都现在缓冲区操作，然后由 Buffer 写入到硬盘上的文件中，亦或程序中一切所需要的对文件的访问，都由 Buffer 从文件中寻找到相应的数据然后读入到缓冲区中以供程序调用。
2.负责缓存区的管理，当某一个缓存块要被替换式，要负责判断该块中的内容是否要被写回文件，负责加载新的模块并在程序结束时也要将缓冲区中的数据写回文件实现数据持久化。
3.模拟 LRU 算法，每次替换 Block 时都会选择未被访问或者时最长时间没被访问的块。

3.接口设计

```
class BufferManager
{
public:
	BufferManager();
	~BufferManager();
	void writeBack(int bufferNum);
	 //将内存中的东西写回到文件当中
	int returnBufferNum(string filename, int blockOffset);
	//传入一个文件名字以及偏移量　将其读入内存中并返回此内存块的编号
	void putSpecialFileIntoBlock(string filename, int blockOffset, int bufferNum);
	//指定一个文件以及偏移量 用指定 BufferNum 的内存块将其读入
	void writeBlock(int bufferNum);
	 // 标记这个 BUfferNUm 的内存块被读操作　　*可考虑精简删除
	void useBlock(int bufferNum);
	//标记这个 BufferNum 的内存块被使用　　* 可考虑精简删除
	int find_EmptyBuffer();
	//从开辟的内存池当中找到一个 空余的未被使用的 如果全被使用则找到最近一次使用最久远的内存块　返回其 BUfferNum
	int find_EmptyBufferExcept(string filename);
	//找到一个空的内存块　在读入一个文件时使用　因此寻找的条件加上一个
	//原内存块读入的不能是这个文件　防止重复读入
```

```
    insertPos getInsertPosition(Table& tableinfor);
    //在一个.table 当中找到一个合适的位置,即找到最后一条记录的下一条插入位置
    int addBlockInFile(Table& tableinfor);
     //在对一个表执行拆入操作时 碰到原来最后一个块不够用 那么新增一个新的 Block 将其偏移量在于那里基础上+1
    //外部调用 API
    int checkWhetherInBuffer(string filename, int blockOffset);
    //检查当前该文件该偏移量的信息是否已经在内存块中  如果是返回 BufferNUm 否返回-1
    void scanIn(Table tableinfo);
    // 顺寻扫描一个.table 文件 将其全部读入内存中
    void setInvalid(string filename);
    //  将一个文件标记为 Invalid
    int FindABlock_for_theFile(string filename, int blockOffset);
    // 为一个文件制定偏移量的信息找到一个新的内存块
    friend class RecordManager;
    friend class CataManager;
//private:
    buffer bufferBlock[MAXBLOCKNUM];                //开辟 内存池 100 个 4kb 大小的内存块
};
```

4.设计思路

在完成整个 BufferManager 之前，我们先设计单个缓存块的结构，如上图所示 Buffer 类实现方法。我们首先开辟了一个 4096+1 大小的空间用来存储整个 block 中的数据，并且立下了 isWritten，isvalid 两个标记以及 filename，blockoffest，recent_time 等属性，还设计了内部初始化函数用于整个块的数据初始化。

在 BufferManager 中有一个 Block 的数组，当中存放了 MaxBlockNum 个 Block，程序的其余部分都需要通过 BufferManager 来申请调用 Buffer。

BufferManager 中 recent_time 在每一次块被访问，被修改时记录下当前的时间，并通过这个量来实现查找最长时间没被访问的块这一功能。

5.实现方法

1）FindABlock_for_the_file 实现方法

函数原型 int FindABlock_for_theFile(string filename, int blockOffset);

函数的功能是接受调用者传入的文件名以及文件偏移量，首先检验其是否已经在缓冲区中，如果是则直接返回对应的 BufferNum，如果不是则新开辟一个内存块将其读入，然后返回新开辟的内存块的 BufferNum；

2）useBlock 实现方法

函数原型 void useBlock(int bufferNum);

该函数提供给外部模块调用，其他模块每次访问指定内存时都会调用该函数，在对相应的 Block 最初修改后标记其最近被访问并且更新这个块的最近访问时间。

3）getInsertPosition 实现方式

函数原型 insertPos getInsertPosition(Table& tableinfor);

该函数返回插入数据段可以插入的位置。调用情景：当需要在表中插入一个新的数据时，调用该函数，读取这个表对应的内存块，然后找到一个空的位置，返回该内存块的编号和内存块中的偏移量，函数根据表的属性得到一条记录的所占空间大小来实现遍历，如果新插入的这条记录恰好需要新增一个 block 来存放，那么就会调用 addBlockInFile 函数，将指定的.table 文件后面新增一个 Block。

4）writeBlock 实现方式

函数原型 void writeBlock(int bufferNum);

函数提供给其他模块使用，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这个 block 做出了修改。调用之后这个 block 被标记被修改过，同时调用 useBlock 函数，标记这个 block 被访问并更新他的 recent_time。

八、小数据测试（200 条插入记录）

1）通过读入文件建立两个表：

```
MiniSQL By Zhb & Hdc!
>>>testfile:0_create.txt;
Interpreter : successful create!
Interpreter : successful create!
) ;
ERROR : invalid query format!
>>>
```

图：0_create.txt

```
create table City (
  ID int,
  Name char(35) unique,
  CountryCode char(3),
  District char(20),
  Population int,
  primary key(ID)
);

create table CountryLanguage (
  CountryCode char(3),
  Language char(30),
  IsOfficial char(1),
  Percentage float,
  primary key(CountryCode)
);
```

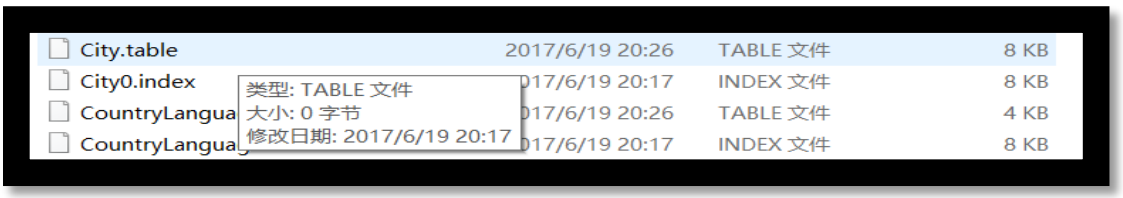
2）通过文件读入测试插入两个小数据

```
>>>execfile:2_insert.txt;
insert into CountryLanguage values ( 'CAF' , 'Sara' , 'F' , 6.4 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into CountryLanguage values ( 'CHE' , 'Romansh' , 'T' , 0.6 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into CountryLanguage values ( 'DEU' , 'Turkish' , 'F' , 2.6 ) ;
Unique Value Redundancy occurs, thus insertion failed
>>>
```

向刚创建的新表中插入小数据，找到三个因为主键或者 unique 属性冲突的插入语句，而其余不冲突的插入信息成功插入。

```
>>>select * from City ;
ID      Name    CountryCode     District        Population
1       Kabul   AFG     Kabol   1780000
2       Qandahar        AFG     Qandahar        237500
3       Herat   AFG     Herat   186800
4       Mazar-e-Sharif  AFG     Balkh   127800
5       Amsterdam       NLD     Noord-Holland   731200
6       Rotterdam       NLD     Zuid-Holland    593321
7       Haag    NLD     Zuid-Holland    440900
8       Utrecht NLD     Utrecht 234323
9       Eindhoven       NLD     Noord-Brabant   201843
10      Tilburg NLD     Noord-Brabant   193238
11      Groningen       NLD     Groningen       172701
12      Breda   NLD     Noord-Brabant   160398
13      Apeldoorn       NLD     Gelderland      153491
14      Nijmegen        NLD     Gelderland      152463
15      Enschede        NLD     Overijssel      149544
16      Haarlem NLD     Noord-Holland   148772
17      Almere  NLD     Flevoland       142465
18      Arnhem  NLD     Gelderland      138020
19      Zaanstad        NLD     Noord-Holland   135621
20      s-Hertogenbosch NLD     Noord-Brabant   129170
21      Amersfoort      NLD     Utrecht 126270
22      Maastricht      NLD     Limburg 122087
23      Dordrecht       NLD     Zuid-Holland    119811
24      Leiden  NLD     Zuid-Holland    117196
25      Haarlemmermeer  NLD     Noord-Holland   110722
26      Zoetermeer      NLD     Zuid-Holland    110214
```

插入小数据之后生成的.table 和.index 文件。



九、大数据测试（4500 条插入记录）

1）通过文件读入 1_insert.txt ;

```
>>>execfile:1_insert.txt;
Interpreter: successful create!
insert into City values ( 649 , 'San Miguel' , 'SLV' , 'San Miguel' , 127696 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 664 , 'Crdoba' , 'ESP' , 'Andalusia' , 311708 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 734 , 'Newcastle' , 'ZAF' , 'KwaZulu-Natal' , 222993 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 802 , 'San Fernando' , 'PHL' , 'Central Luzon' , 221857 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 826 , 'Valencia' , 'PHL' , 'Northern Mindanao' , 147924 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 828 , 'Santa Maria' , 'PHL' , 'Central Luzon' , 144282 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 832 , 'Toledo' , 'PHL' , 'Central Visayas' , 141174 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 843 , 'San Miguel' , 'PHL' , 'Central Luzon' , 123824 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 849 , 'San Carlos' , 'PHL' , 'Western Visayas' , 118259 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 865 , 'San Jose' , 'PHL' , 'Central Luzon' , 108254 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 876 , 'San Fernando' , 'PHL' , 'Ilocos' , 102082 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 1001 , 'Depok' , 'IDN' , 'Yogyakarta' , 106800 ) ;
Unique Value Redundancy occurs, thus insertion failed
insert into City values ( 1820 , 'London' , 'CAN' , 'Ontario' , 339917 ) ;
```

插入 4500 条记录：对于主键重复的插入请求给出提示。

2）从插入的 City 表中 Select * 查看插入情况。

```
4037    Burbank USA     California      100316
4038    Clearwater      USA     Florida 99936
4039    Midland USA     Texas   98293
4040    Davenport       USA     Iowa    98256
4041    Mission Viejo   USA     California      98049
4042    Miami Beach     USA     Florida 97855
4043    Sunrise Manor   USA     Nevada  95362
4044    New Bedford     USA     Massachusetts   94780
4045    El Cajon        USA     California      94578
4046    Norman  USA     Oklahoma        94193
4048    Albany  USA     New York        93994
4049    Brockton        USA     Massachusetts   93653
4050    Roanoke USA     Virginia        93357
4051    Billings        USA     Montana 92988
4052    Compton USA     California      92864
4053    Gainesville     USA     Florida 92291
4054    Fairfield       USA     California      92256
4055    Arden-Arcade    USA     California      92040
4057    Visalia USA     California      91762
4058    Boulder USA     Colorado        91238
4059    Cary    USA     North Carolina  91213
4060    Santa Monica    USA     California      91084
4061    Fall River      USA     Massachusetts   90555
4062    Kenosha USA     Wisconsin       89447
4063    Elgin   USA     Illinois        89408
4064    Odessa  USA     Texas   89293
4065    Carson  USA     California      89089
4066    Charleston      USA     South Carolina  89063
4067    Charlotte Amalie        VIR     St Thomas       13000
4068    Harare  ZWE     Harare  1410000
4069    Bulawayo        ZWE     Bulawayo        621742
4070    Chitungwiza     ZWE     Harare  274912
4071    Mount Darwin    ZWE     Harare  164362
4072    Mutare  ZWE     Manicaland      131367
4073    Gweru   ZWE     Midlands        128037
4074    Gaza    PSE     Gaza    353632
4075    Khan Yunis      PSE     Khan Yunis      123175
4076    Hebron  PSE     Hebron  119401
4077    Jabaliya        PSE     North Gaza      113901
4078    Nablus  PSE     Nablus  100231
4079    Rafah   PSE     Rafah   92020
Interpreter: successful select!
>>>
```

十、亮点：创新的 index 实现方式获得高速插入；

超大数据测试（验收所给出测试数据）

1）文件读入 student.txt 以批量实现导入（每一个子.txt 文件为 1000 条 insert 记录）

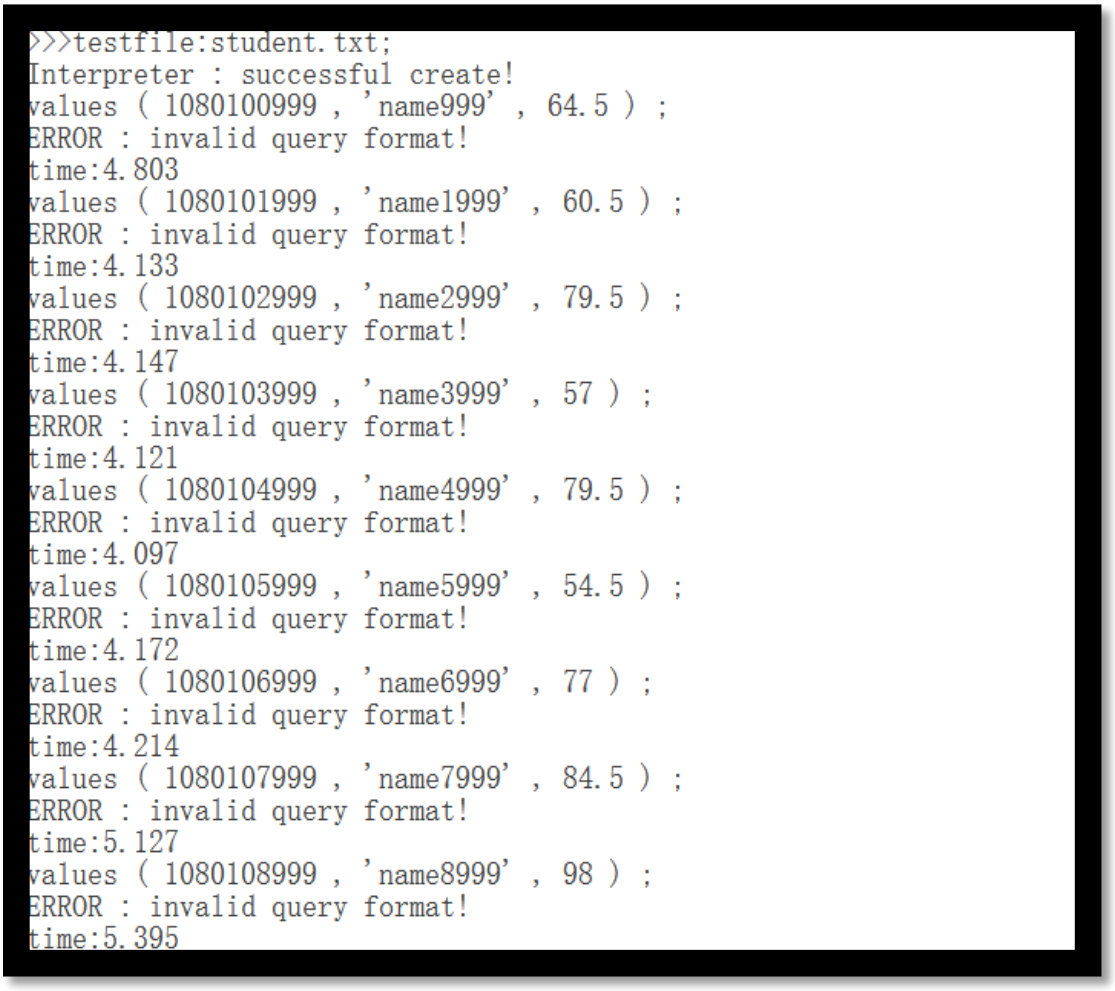
```
create table student2(
          id int,
          name char(12) unique,
          score float,
          primary key(id)
);

execfile:instruction0.txt;
execfile:instruction1.txt;
execfile:instruction2.txt;
execfile:instruction3.txt;
execfile:instruction4.txt;
execfile:instruction5.txt;
execfile:instruction6.txt;
execfile:instruction7.txt;
execfile:instruction8.txt;
execfile:instruction9.txt;
```

2)检查插入情况：

```
1080109974      name9974        52.5
1080109975      name9975        97
1080109976      name9976        50
1080109977      name9977        56.5
1080109978      name9978        84.5
1080109979      name9979        99.5
1080109980      name9980        88
1080109981      name9981        78.5
1080109982      name9982        53.5
1080109983      name9983        58.5
1080109984      name9984        64.5
1080109985      name9985        71
1080109986      name9986        61
1080109987      name9987        91.5
1080109988      name9988        75
1080109989      name9989        58
1080109990      name9990        75
1080109991      name9991        69
1080109992      name9992        50.5
1080109993      name9993        67.5
1080109994      name9994        97.5
1080109995      name9995        59.5
1080109996      name9996        65.5
1080109997      name9997        61
1080109998      name9998        84.5
1080109999      name9999        69.5
1080110000      name10000       80.5
Interpreter: successful select!
```

3）每次插入 1000 条记录所用的时间测试：



4）一万条记录所用总时间

5）实现方法：index 的存储结构（细节）

Index 文件的存储结构：

Index 是以 B+树的方式储存记录的地址。B+树的每一个结点（包括内结点和叶结点）大小都等于一个块的大小。我们将每一个结点标号，通过识别它们的标号实现对不同结点的访问。

内结点中的内存分配方式为前 16 位记录结点的信息，剩余的部分平均分给每一个数据以及它们的指针，具体分配方式如下：

0-4 位记录结点中当前数据数量，4-8 位记录最大数据数量（以便判断结点是否需要分裂），8-12 位记录结点类型（内结点/叶结点），12-16 位记录父节点的标号。我们分配 keylength[type]+16 位给每一个数据，前 keylength[type]位指的是 int，float，string 型需要分配的内存大小，接下来 16 位记录数据的当前位置（4 位），下一个数据的位置（4 位），比数据小的下一层结点的标号（4 位）， 比数据大的下一层结点的标号（4 位）。

叶结点分配方式基本与叶结点相同，不同之处为：对于内结点中记录比数据小的下一层结点的标号的位置，其在叶节点中记录数据的内存地址，而对于内结点中记录比数据大的下一层结点的标号的位置，其在叶结点中置 0（不使用）。

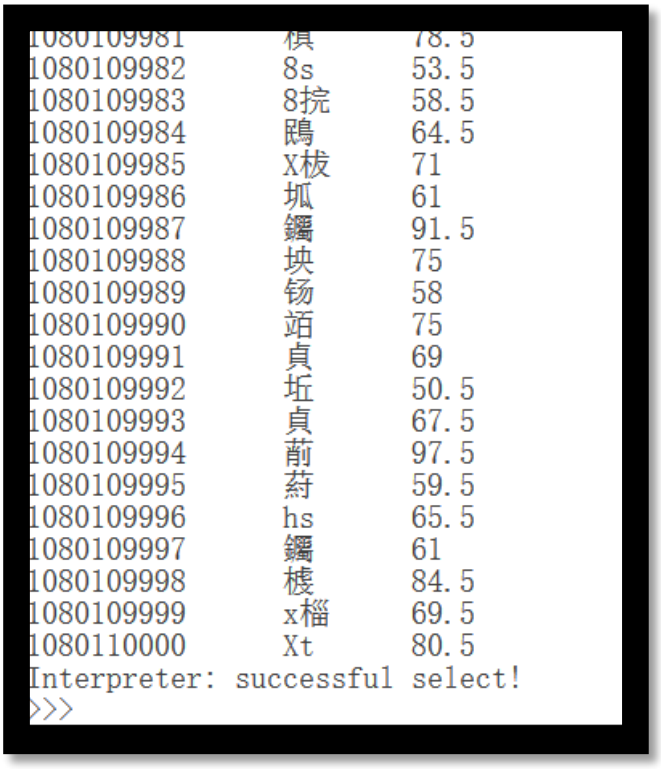
为了能够方便访问到根结点，我们在 index 文件的第一个块中不存储结点，而是记录索引文件信息，0-4 位记录数据类型（int，float，string），4-8 位记录最大结点数量，8-12 位记录当前结点数量，12-16 位记录根结点的标号，剩余的内存空间存放索引的名字。

通过上述定义索引文件的结构，我们可以方便建立索引，并通过不断向下层访问结点的方式快速查询数据的内存地址。

十一、开发以及验收过程中遇到的一些 Bug；

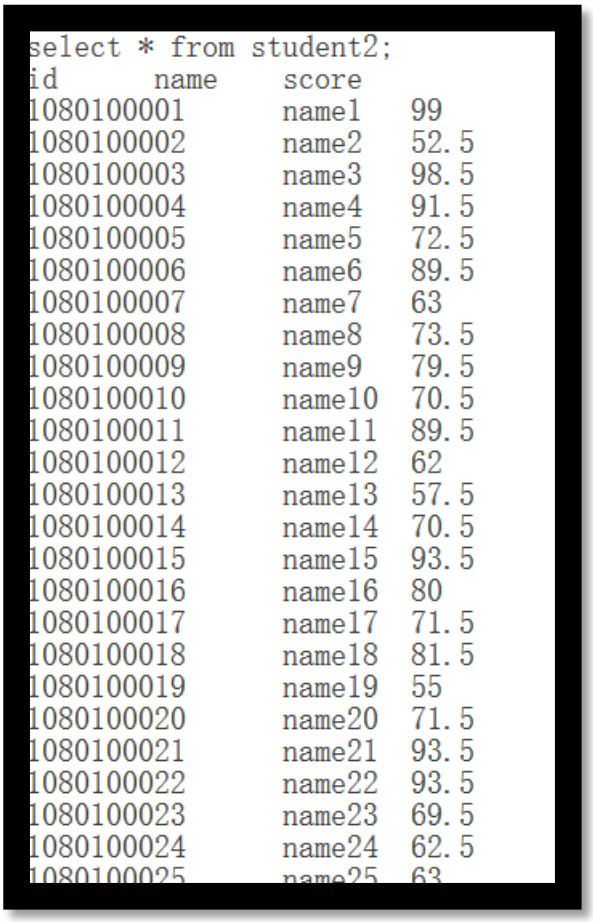
中文乱码 Bug：（已解决）

在开发过程中我们碰到了一个小 bug，就是当讲一条条记录 insert 之后，在调用 select * 指令将其打印出来，会出现 char 类型乱码的问题，如图所示：



这个问题也在验收时遇到了，在验收之后我们重新调整了程序，发现解释器在读入 char 类型的数据时，解释器正确地将其以二进制写入到文件当中，但是应对 select 语句，将 char 类型打印出来时，错误的将第一位（用来标记属性值是否为空的一字节读入也打印了出来），这就导致了 char 类型打印时出现了乱码。意识到这个问题之后，我们修改了

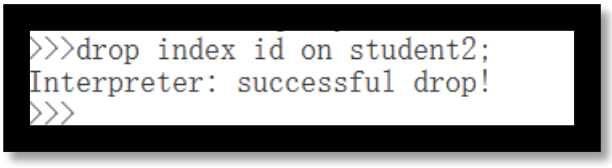
对应的打印结果的部分，更改之后解决了这个问题；



```
select * from student2;
id      name    score
1080100001      name1   99
1080100002      name2   52.5
1080100003      name3   98.5
1080100004      name4   91.5
1080100005      name5   72.5
1080100006      name6   89.5
1080100007      name7   63
1080100008      name8   73.5
1080100009      name9   79.5
1080100010      name10  70.5
1080100011      name11  89.5
1080100012      name12  62
1080100013      name13  57.5
1080100014      name14  70.5
1080100015      name15  93.5
1080100016      name16  80
1080100017      name17  71.5
1080100018      name18  81.5
1080100019      name19  55
1080100020      name20  71.5
1080100021      name21  93.5
1080100022      name22  93.5
1080100023      name23  69.5
1080100024      name24  62.5
1080100025      name25  63
```

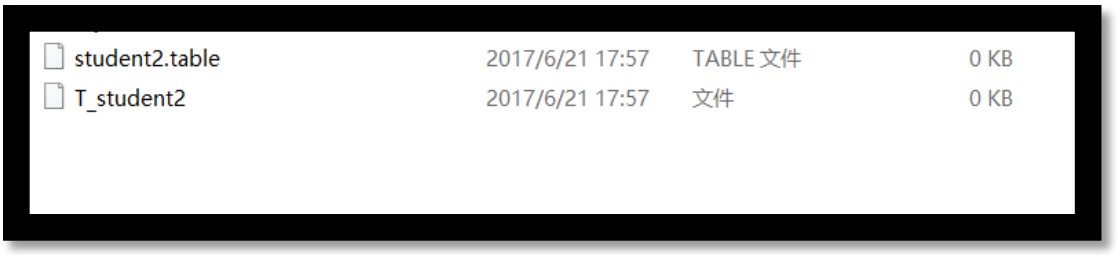
Delete index Bug：（已解决）

在验收过程中无法实现 delete index 功能，验收过后才想起来我们的解释器当中对于 Delete index 语法的定义和验收时的语句不太一样，这就导致了使用验收流程当中的语句无法实现 delete index 反而一直提示语法错误。使用我们的语法是可以完成 index 的删除操作的。我们对于不用的 index 直接 drop，所以我们的语法为 Drop index name on 表名 。



```
>>>drop index id on student2;
Interpreter: successful drop!
>>>
```

相应的.index 文件就被删除。



十一、关于暴力搜索（没有 index） 以及 select with index 两种情况下的时间优化问题：

如果针对单条记录并不能很直观的感受到两种 select 方式的时间差异，于是我们打算利用大量 select 操作进一步使得两者的时间差异扩大，从而得到两种 select 方法比较明显的差距。基于没有如此大量的 select 操作语句，我们观察到在执行有 primary key 的记录插入时，会先把要插入的记录当中的 primary key 当作查询条件进行一次 select 以检查是否有冗余，是否能够插入，这样的话我们就可以使用 100000 条插入数据当作 100000 条 select 数据从而进行两种不同的 select 方式：暴力搜索和 select with index 之间的比较，首先我们将 select with index 功能注释掉重新插入 10000 条数据，在用时 10min 中时插入到第 8999 条记录，之后长时间进程无法截止，于是我们便以（10min，8999 条记录）作为暴力搜索的查询成绩。

然后我们将 select with index 取消注释，也就是使用 select with index 再来进行实验：

```
>>>testfile:student.txt;
Interpreter : successful create!
values ( 1080100999 , 'name999' , 64.5 ) ;
ERROR : invalid query format!
time:4.803
values ( 1080101999 , 'name1999' , 60.5 ) ;
ERROR : invalid query format!
time:4.133
values ( 1080102999 , 'name2999' , 79.5 ) ;
ERROR : invalid query format!
time:4.147
values ( 1080103999 , 'name3999' , 57 ) ;
ERROR : invalid query format!
time:4.121
values ( 1080104999 , 'name4999' , 79.5 ) ;
ERROR : invalid query format!
time:4.097
values ( 1080105999 , 'name5999' , 54.5 ) ;
ERROR : invalid query format!
time:4.172
values ( 1080106999 , 'name6999' , 77 ) ;
ERROR : invalid query format!
time:4.214
values ( 1080107999 , 'name7999' , 84.5 ) ;
ERROR : invalid query format!
time:5.127
values ( 1080108999 , 'name8999' , 98 ) ;
ERROR : invalid query format!
time:5.395
```

一万条记录成功插入花费总时间：

```
time:46.82
>>>
```

在其他条件保持一致的基础之上，唯一的变量就是 每次插入 index 之前进行的主键查询所用的方式，第一次查询使用的是暴力搜索，第二次使用的是 select with index ，所以两个数据表征了 10000 次查询所用时间。由这两个数据可以明显的看出暴力查找和 select with index 之间存在着巨大的速度差异。Select with Index 在查找过程中对于降低成本具有重大意义。

十二、关于团队分工以及承诺；

miniSQL 开发模块分工如下：

Interpreter ，recordManager，BufferManager，API：黄栋成

Bptree，index，catalog： 占涵冰

report： 占涵冰，黄栋成

commitment：

We are here to pledge that this Project is a result of our own efforts.