MINISQL Guideline

学号	姓名	专业
3160104714	熊苗	交叉创新平台1604
3160105705	李易非	交叉创新平台1604
3160104805	王丹尧	交叉创新平台1604

一、MINISQL框架

1.1 背景

1.1.1 设计目的

设计并实现一个精简型单用户 SQL 引擎(DBMS)MiniSQL，允许用户通过字符界面输入 SQL 语句实现表的建立/删除；索引 的建立/删除以及表记录的插入/删除/查找。 通过对 MiniSQL 的设计与实现，提高系统编程能力，加深对数据库系统原理的理解。

1.1.2 项目背景和方案

这个项目分为不同的模块，各个模块之间需要在各司其职的同时紧密联系，才能成功运行。
于是经过分析讨论，我们首先明确了MiniSQL的整体架构和各模块功能以及它们之间的联系。然后根据各模块之间的联系和接口进行了进一步的分工，每个人负责不同的模块，并在个人模块完成后进行模块测试。最后，将所有模块合在一起进行整体项目测试并加以完善。
项目最终实现的MiniSQL可以支持基本的SQL语句，也能针对不同的情况进行处理，提示错误信息，执行成功则将结果和执行时间反馈给用户。

1.2 功能描述

> 数据类型

支持三种基本数据类型：int，char(n)，float，其中 char(n)满足 1 <= n <= 255 。

表定义 一个表最多可以定义 32 个属性，各属性可以指定是否为 unique；支持单属性的主键定义。

> 表定义

一个表最多可以定义 32 个属性，各属性可以指定是否为 unique；支持单属性的主键定义。

> 索引的建立和删除

对于表的主属性自动建立 B+树索引，对于声明为 unique 的属性可以通过 SQL 语句由用户指定建立/删除 B+树索引（因此，所有的 B+树索引都是单属性单值的）。

> 查找记录

可以通过指定用 and 连接的多个条件进行查询，支持等值查询和区间查询。

> 插入和删除记录

支持每次一条记录的插入操作；支持每次一条或多条记录的删除操作。

1.3 运行环境

> 编程语言：C++

> 运行环境：Mac OS 和 Windows 均可

> `CMakeLists.txt` 已在Windows中测试可用。

二、各模块实现功能

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。
```

API

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

Catalog Manager

Catalog Manager 负责管理数据库的所有模式信息，包括👇

```
1．数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
2．表中每个字段的定义信息，包括字段类型、是否唯一等。
3．数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。 Catalog Manager 还必需提供访问及操作上述信息的接口，供 Interpreter 和 API 模块使用.
```

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。

Index Manager

Index Manager 负责 B+树索引的实现，实现 B+树的创建和删除（由索引的定义与删除引起）、等值查找、插入键值、删除键 值等操作，并对外提供相应的接口。
B+树中节点大小应与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。

Buffer Manager

Buffer Manager 负责缓冲区的管理，主要功能如下👇

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。

三、文件存储方式

3.1 设计简介

本次实验中，有3种类型的文件

1. 表的 metadata；
2. 索引的 metadata;
3. 数据文件；
4. 索引文件；

   前三种文件我们进行了**同质化处理**，可以用相同的处理方式来读对应数据，第四种文件因为 B+树的特性，采取了不同的存储方式；

3.2 数据文件基本约定

3.2.1 物理存储说明

- 采用**定长记录**方法，`varchar(n)` 将在数据文件中分配 n 个字节的空间，不会因为具体记录长度的变化而变化；

- 采用 **Heap File** 方法，即文件内部 record 没有固定的存储顺序，通过 **free list** 来标记所有被删除的记录。free list header 在文件中占据4位 (int)，若为 -1 则说明到了 free list 尾部，若不为1，则代表下一条无效 record 在文件中的**绝对地址**。其余被删除记录的前四位与上述逻辑相同，若为 -1 则说明到了 free list 尾部，若不为1，则代表下一条无效 record 在文件中的**绝对地址**；
- 每一条数据后新增一个 **有效位** ，在遍历文件过程中，仅仅通过 **free list** 来判断一个 record 是否被删除非常耗时，最终我们决定 **空间换时间**， 在每一条记录后都添加一个有效位，若为1则表示有效记录，若为0则表示无效记录；
- 记录 **不会跨块存储** ，在一个 Block 不足以容纳一条记录时，直接前往下一个 Block；

3.2.2 逻辑存储说明

- 每个数据文件都有自己的 metadata，存储在所有文件的 **第一个 Block** 中，metadata包含
  - Record Length (物理长度，包含了有效位)
  - free list header，free list 链表头，指向一个被删除的记录；
  - Record Count，代表文件中记录总数，注意这里包含无效记录。这是为了确认文件 **EOF** 的具体位置；
- 其余的 Block 正常存储数据，若为无效记录，则前四位指向 free list 下一条无效记录；若为有效记录，则是一个 record 的正常存储单元；

3.2.3 注意事项

- 逻辑 Record Length 为物理 Record Length 减1；
- 因为要维护一个 free list，我们为每一个指针都分配了四个字节的空间，所以对于 varchar(j) (j < 4) 类型，我们都按照 varchar (4) 进行处理；
- Record Count，代表文件中记录总数，注意这里包含无效记录。这是为了确认文件 **EOF** 的具体位置；

3.2.2 具体数据文件存储内容

3.2.2.1 Table Meta 存储说明

Table Meta 存储在特定文件夹 `TableMeta/` 中，其中包含一个提供所有 Table 名字的 `TableMeta/tables` 文件，以及各个 Table 对应的 Meta Data 文件 `TableMeta/Table`

- `TableMeta/tables` 存储数据库中所有表名 (32 bytes)；
- `TableMeta/Table` 存储特定 table 的属性信息，文件中的一条记录存储属性名 (32 bytes)、数据类型 (2 bytes)、是否主键 (1 byte)、是否唯一 (1 byte)；

  把 Table Meta 又分成 tables 与 特定 table 是为了保持同一个文件的 **record length 一致**，因为不同表的属性个数是不同的，如果存储在一起，很难在不浪费空间的前提下保持记录长度的一致性；

3.2.2.2 Index Meta 存储说明

Index Meta 存储在特定文件夹 `IndexMeta/` 中，其中包含一个提供所有 Index 信息的 `IndexMeta/indices` 文件。

- `IndexMeta/indices` 存储 索引名 (64 bytes) 、表名 (32 bytes)、属性名 (32 bytes)；

👇是一个例子

```
+--- TableMeta
|        |--- tables
|        |--- book
|        |--- student
|
+--- IndexMeta
         |--- indices
```

3.2.2.3 具体记录存储说明

具体记录文件存储在特定文件夹 `data/` 下，每一张表占据一个文件，存储表中真实的记录。

👇是一个例子

```
+--- data
        |--- book
        |--- student
```

四、Interpreter 模块设计报告 👹

> by 王丹尧

3.1 模块概述

Interpreter模块的基本功能是将用户输入的命令进行语法分析和语义解析，得到需要的命令参数，将命令参数封装成对应命令的参数类对象传给API执行相应的操作。同时对于不能识别的命令向用户提供错误信息。

3.2 主要对外接口

> select

·需要查询的表的名字 table_name
·需要查询的属性名字 attribute_name
·需要查询的属性条件 condition
·查询的属性条件的“=”等操作符 operand

> Insert

·插入的表的名字 table_name
·插入条目的信息vector data
·插入条目的信息对应的type datatype

> delete

·要删除的表名 table_name
·需要删除的属性名字 attribute_name
·需要删除的属性条件 condition
·删除的属性条件的“=”等操作符 operand

> drop table

·要创建的表的名字 table_name
·要创建的表包含的属性的vector a
·要创建的表的主键 primary

> create Index

·需要创建索引的表的名字 table_name
·需要创建索引的属性的名字 attribute_name
·要创建的索引的名字 index_name

> execute file

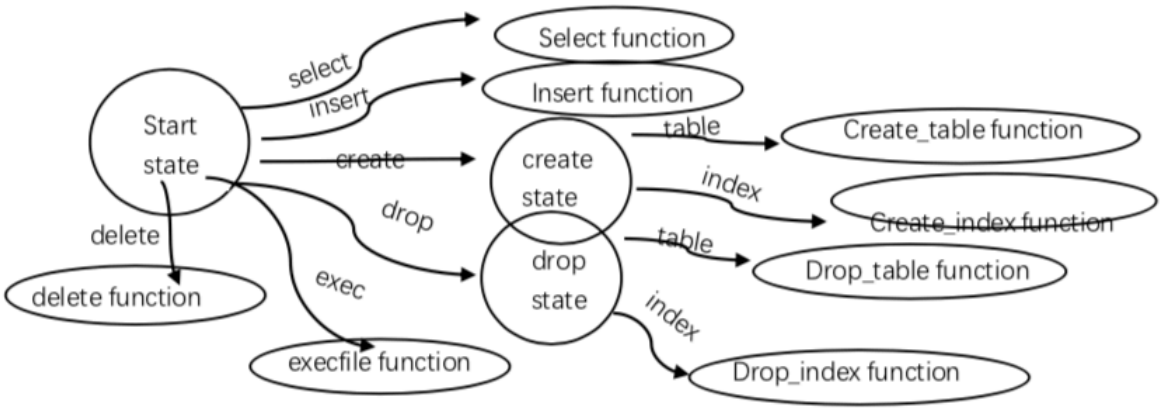
不对外提供接口
读取用户提供的文件并执行其中的语句

3.3 设计思路

1. 读取用户输入的一条语句，以“；”字符作为一条语句结束的标记。将语句解 析成为一个有意字串的vector（具体实现名称为tokens）。例如下列语句：

```
create table tablename
(
    variablename1 type1,
    variablename2 type2
);
“create ”,“table”,“tablename”,“(“,“variablename1”,“type1”
```

2. 对有意字串进行分析状态图



五、API 模块设计报告

> by 李易非

4.1 初步介绍

API是程序架构的中间层，API接受 Interpreter 解释完成、初步检测语法的命令，调用 CatalogManager 检验一致性、确定命令的具体执行规则；调用 RecordManager 来进行记录文件的更新；调用 IndexManager 进行索引文件的更新；在本次实验中，我们的API 类有如下几个函数。

```
class API
{
  public:
    bool create_table(表名, 属性);
    bool drop_table(表名);
    bool insert(表名, 插入数据, 插入数据类型);
    int Delete(表名, 属性名, 条件, 操作数);
    int select(表名, 属性名, 条件, 操作数);
    bool create_index(表名, 属性名, 索引名);
    bool drop_index(索引名, 表名);
};
```

4.2 技术细节

错误管理

本次工程中，共建 `Error` 类来进行错误管理，所有底层函数都利用 **throw** 机制抛出错误，

在 API 层集中进行 **try catch** 。以 Drop Index 为例，API 要做的事很简单，调用对应的函数并处理错误即可。

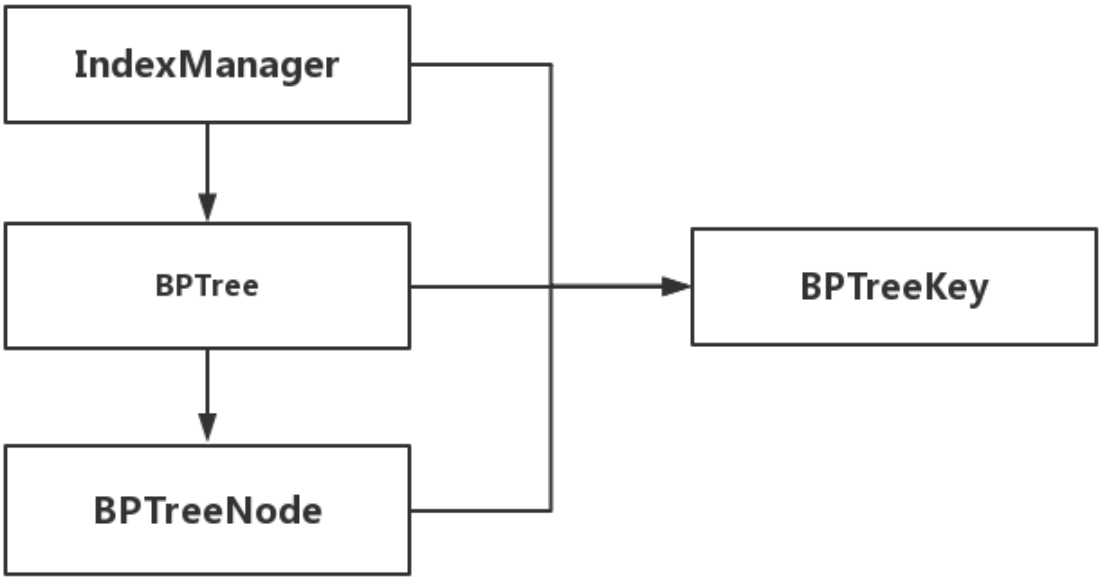
```
bool API::drop_index(const string &index_name, const string &table_name)
{
    try
    {
        CatalogManager &catalogmanager = MiniSQL::get_catalog_manager();
        IndexManager &indexmanager = MiniSQL::get_index_manager();

        catalogmanager.drop_index(index_name, table_name);
        indexmanager.dropIndex(index_name);
        return true;
    }
    catch (Error err)
    {
        err.print_error();
        return false;
    }

    return true;
}
```

六、Index Manager 设计简介 👹

> by 熊苗

1 设计简介

`IndexManager` 实现的是与文件交互的B+树，每一个节点都是存放在文件里4K大小的 `block` 。其主要由4个类： `IndexManager` ， `BPTree` , `BPTreeNode` , `BPTreeKey` 实现，其调用关系 👇



2 实现思路

1. 当API调用IndexManager时，IndexManager会将传入信息封装成一个entry，然后生成一个临时的BPTree读取文件进行操作。
2. BPTree通过Index文件的headerBlock获取基本信息，之后从根结点开始一层层向下搜索，直至搜索到叶结点。
3. 每进入一个Node，都会先从文件里把结点的相应信息和所有的entry读到内存里，然后执行相应的删除查询或者插入操作，最后写会内存。

3 重要组成部分简介

> IndexManager

API和B+树之间的交互平台，整个B+树的对外接口。

主要用于生成临时的B+树和调用相应的操作。

> **BPTree**
>
> 实现B+树的各种操作。

结构图：header，保留block ID = 0；

nodeCount	firstEmptyBlock	firstLeftBlock	mostRightBlock	RootID
记录了B+树的block数目	维护Index文件中第一个为空的block的ID	记录B+树叶节点中最左边一个的blockID，方便范围查询	记录B+树叶节点中最右边一个的blockID，方便范围查询	根结点的Block ID，查找的起点

> **BPTreeNode**
>
> 1个BPTreeNode是B+树的一个结点，实现中采用了4096Byte即一页来存储一个node 。在构造函数时将会通过buffer读取一个完整的node到文件里，析构函数会判断node是否被修改过或者remove，然后把结点写回文件。支持对B+树的结点进行查询删除等各种操作。

isLeaf	nodeSize	nextNodeID	Entry1	Entry2	.....
是否为叶节点	结点中entry的个数	对叶节点，为下一个结点的block ID，方便范围查询；对内子结点，存储第一个指针	第一个entry	第二个entry	

> **BPTreeKey**
>
> 对一个Attribute的封装，由key和它对应的pointer组成。

raw_data	pointer
存储了char*数组，长度为keyLength	存储记录所在的blockID

封装的目的

为了避免实现模版类，通过把Key封装在一个类里，重载各种逻辑比较符号，简化B+树里基于key的各种操作，如搜索查询等。

七、Manager 设计简介 

1 File Manager 设计简介

> by 李易非

由于 Record Manager 与 Catalog Manager 都要对数据文件进行操作，我们在 Record Manager / Catalog Manager 与 Buffer Manager 之间加入 File Manager，理由如下：

- 减少各个 Manager 交互；
- 逻辑意义上更加清晰，Record Manager 与 Catalog Manager 不必亲自提供 / 调用各个文件的块位置，逻辑上只对对应 " 文件 " 进行操作。

  File Manager 具有的功能如下 (简化版) ：

```
class FileManager
{
    private:
    string file_name;
```

```
    int record_length;           // 记录长度
    int first_free_record_addr; // free list 表头
    int record_count;            // 记录总数
    int pointer;                 // 文件中的“记录指针”

    public:

    FileManager(文件名);

    const char* get_record(记录地址); // 通过地址获得记录内容

    const int add_record(记录的rawdata); // 添加一条记录

    bool delete_record_ByAddr(记录地址); // 通过地址删除记录

    int getNextRecord(char * rawdata);  // 获得下一条record


};
```

2 Buffer Manager 简介 

> by 李易非

Buffer Manager 是数据库与数据文件进行交互的中间层，所有数据库调用 / 更新文件的行为，都必须通过 BufferManager 来管理。这样可以尽量减少内存与磁盘的速度鸿沟，把一些较为常用的 Block 存入 Buffer 中，为了尽量减少 I / O 操作，应该把一些 Block 存在 Buffer 中，数据库需要访问速度更快；

我们采取了 **时钟算法** 作为块替换策略，时钟算法的效率接近块替换策略 **LRU算法** ，但是实现却简单很多，所以我选择了时钟算法作为Buffer的块替换策略；为了实现时钟算法，我们在Buffer内部实现了一个循环链表，循环链表的 Node 是 BufferNode 类，BufferNode 中含有一个 Block 实体，一个标记位，以及指向下一个 BufferNode 的指针；

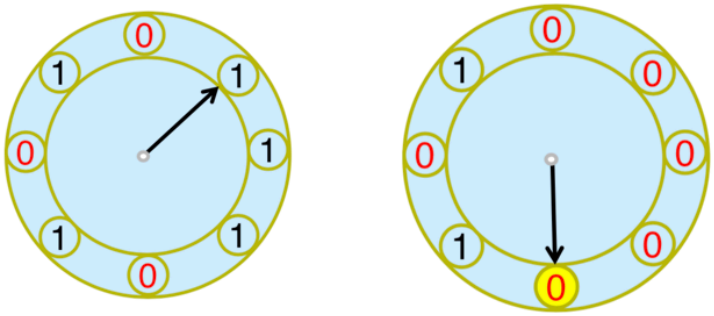
BufferNode 具有的功能如下 (简化版):

```
class BufferNode
{
    Block * block;                  // 实际的Block内容
    bool reference_bit;             // 标记位，来确定时钟算法替换的块
    BufferNode * next;              // 指向下一个BufferNode

    Block * get_block();            // 获得BufferNode中的块
    void set_block(Block * block);  // 重置BufferNode中的块

};
```

Buffer Manager 具有的功能如下 (简化版) :

```
class BufferManager
{
    const int max_block_count;      // Buffer 容量
    BufferNode * clock_ptr;         // ptr of the 时钟算法


    void WriteBlockBack(Block * block);                 // 写回一个 Block
    Block* getBlock(文件名, Block ID);                  // 通过文件名，Block
ID 获得对应块
    void addBlock(Block * block);                       // 在Buffer中加入一个
块
    void DeleteBlockByFile(文件名);                     // 删除所有与文件名相关
的块
    void WriteAllBack();                                // 写回Buffer里的所有
块

};
```

2.1 时钟算法介绍



Buffer中的每一个结点都有一个参考位，在需要替换Buffer中的块时，

- 从时钟算法指针开始遍历，若参考位为1，则将该节点参考位置为0，前往下一块；
- 重复上述过程直到找到一个参考位为0的节点，将其替换；

  每当有新块加入时，将其参考位变为1；

3 Catalog Manager 模块设计报告

> by 李易非

Catalog Manager 管理数据库的 Meta Data，在数据库启动阶段，Catalog Manager 将读取 Meta Data并把它们存入内存当中，这样做的理由如下：

- minisql 应用场景：表/索引的数量不会很多，且表的属性最多只有32个；
- 在程序中我们需要大量访问数据 Meta Data，直接存入内存，可以加快访问速度；

  File Manager 具有的功能如下 (简化版) ：

```
class CatalogManager
{
    unordered_map <string, Table *> Name2Table; // 表名到表的hash
    unordered_map <string, Index *> Name2Index; // 索引名到索引的hash

    CatalogManager();

    bool create_table(表名, 属性集);          // 创建表，更新Meta data
    bool drop_table(表名);                    // 删除表，更新Meta data
    const Table* get_table(表名);             // 通过表名获得表
    bool create_index(索引名, 表名, 属性名);  // 创建索引，更新Meta data
    bool drop_index(索引名, 表名);            // 删除索引，更新Meta data

};
```

4 Record Manager 设计报告

> by 李易非

Record Manager 具体管理各个表的记录，依据 API 提供的需求来进行相应数据文件的更新，因为之前 File Manager 的实现，Record Manager 的实现被大大化简，从物理层操作上升到了逻辑层操作。

Record Manager 具有的功能如下 (简化版) ：

```
class RecordManager
{
    bool create_table(表名);
    bool drop_table(表名);
    int insert(表名, rawdata);
    int select(表名, 属性名, 条件, 操作数);
    int Delete(表名, 属性名, 条件, 操作数);
    const char * GetRecordByAddr(表名, 地址);
    bool DeleteRecordByAddr(表名, 地址);
};
```

八、测试说明 ✅

> by 李易非 王丹尧 熊苗

一、基础测验

1、normal1.sql ✅

指导书中提供的基本例子；

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

create index stunameid on student(sname);

insert into student values('123','wy',22,'M');

select * from student;

select * from student where sgender = 'M';

select * from student where sage = 22;
```

测试基本的建表，建索引，插入，选择是否正常；

2、normal2.sql ✅

```
Create table(
    id int primary key,
    name varchar(50),
    age INT);//错误：没有表名
```

```
Create table student(
    id int primary key,
    name varchar(50),
    age);//错误：age没给类型

Create table student(
    id int primary key,
    name varchar(50) unique,
    age INT);//建表并测试unique属性

create index person on student;//错误反例

create index person in student(id);//错误反例in

create index person on student(id);//创建索引

drop index person;//删除索引反例
drop index person on student;//删除索引

insert into student values(5, 'lyf');

insert into student values(5, 'lyf', 18);//单条记录插入

select * from student where id = 5;//等值查询

select * from student where id > 4;//范围查询

select * from student where id = 0 and name = 'k';//and连接的多条件查询

delete from student where id = 1;//找不到需要删除的

delete from student where id = 5;//删除记录

drop table student;//删除表
```

测试所有指令，报错显示；

3、normal3.sql ✅

```
create table book(id int primary key, name varchar(30) unique);
insert into book values(1, 'LYF');
insert into book values(2, 'LYF');
drop index bookname on book;
insert into book values(3, 'LYF');

create index bookname on book(name);
insert into book values (4, 'LYF');
```

主要测试索引，unique属性在有索引 / 无索引情况下能否正常判断一致性；

二、压力测验

1、testint.sql ✅

`int` 作为表的 `primary key` 插入10w条，检查select / delete 能否正常进行；

```
create table book(id INT PRIMARY KEY, name VARCHAR(30));
insert into book values (0, 'I am upset');
insert into book values (1, 'I am upset');
insert into book values (2, 'I am upset');
...
...
...
insert into book values (99997, 'I am upset');
insert into book values (99998, 'I am upset');
insert into book values (99999, 'I am upset');


select * from book;
select * from book where name = 'I am upset';
select * from book where id > 50000;

delete from book;
drop table book;
```

2、testfloat.sql ✅

`float` 作为表的 `primary key` 插入10w条，检查select / delete 能否正常进行；

```
create table book(id FLOAT PRIMARY KEY, name VARCHAR(30));
insert into book values (0.1, 'I am upset');
insert into book values (1.1, 'I am upset');
insert into book values (2.1, 'I am upset');
...
```

```
...
...
insert into book values (99997.1, 'I am upset');
insert into book values (99998.1, 'I am upset');
insert into book values (99999.1, 'I am upset');


select * from book;
select * from book where name = 'I am upset';
select * from book where id > 50000;

delete from book;
drop table book;
```

3、teststring.sql ✅

`varchar(30)` 作为表的 `primary key` 插入10w条，检查select / delete 能否正常进行；

```
create table book(id varchar(30) primary key, name varchar(30));
insert into book values ('0.1abc', 'I am upset');
insert into book values ('1.1abc', 'I am upset');
insert into book values ('2.1abc', 'I am upset');
...
...
...
insert into book values ('99997.1abc', 'I am upset');
insert into book values ('99998.1abc', 'I am upset');
insert into book values ('99999.1abc', 'I am upset');


select * from book;
select * from book where name = 'I am upset';
select * from book where id > '50000.1abc';

delete from book;
drop table book;
```

4、testunique.sql ✅

主要对unique属性进行大数据测试

```
create table book(id varchar(30) primary key, name varchar(30) unique);
insert into book values ('0.1abc', '0.1');
insert into book values ('1.1abc', '1.1');
insert into book values ('2.1abc', '2.1');
...
```

```
...
...
insert into book values ('99997.1abc', '99997.1');
insert into book values ('99998.1abc', '99998.1');
insert into book values ('99999.1abc', '99999.1');


select * from book;
select * from book where name = '666.1';
select * from book where id > '50000.1abc';

drop index bookname on book;
insert into book values('hh', '0.1'); //没有index条件下一致性检验

create index bookname on book(name);  // 检查能否正常建树
insert into book values('hh', '0.1'); // 具有index条件下一致性检验


delete from book;
drop table book;
```

成员分工

本系统的分工如下:

Interpreter 模块.....王丹尧

API 模块.....李易非

Catalog Manager 模块 .....李易非

Record Manager 模块.....李易非

Index Manager 模块.....熊苗

Buffer Manager 模块.....李易非

总体设计报告.....熊苗、王丹尧

模块汇总.....李易非

测试.....ALL