

**本科实验报告**

|  |  |
| --- | --- |
| 课程名称： | 数据库系统 |
| 姓 名： | 汤烨春 成菲尔 徐雅涵 周益聪 李明光 |
| 学 院： | 计算机学院 |
| 专 业： | 软件工程 |
| 学 号： | 3160102212 3160101782 3160102199 3160102442 3160104441 |
| 指导教师： | 陈岭 |

2018年 7 月 1 日

目录

[一、Minisql总体设计报告 4](#_Toc518233661)

[1、引言 4](#_Toc518233662)

[1.1 编写目的 4](#_Toc518233663)

[1.2 项目背景 4](#_Toc518233664)

[1.3定义 4](#_Toc518233665)

[1.4 参考资料 4](#_Toc518233666)

[2、任务概述 4](#_Toc518233667)

[2.1功能概述 4](#_Toc518233668)

[2.2 运行环境 5](#_Toc518233669)

[2.3 具体分析 5](#_Toc518233670)

[3、minisql系统结构设计 5](#_Toc518233671)

[3.1总体设计 5](#_Toc518233672)

[3.2 Interpreter模块 6](#_Toc518233673)

[3.3 API模块 6](#_Toc518233674)

[3.4 Catalog Manager模块 7](#_Toc518233675)

[3.5 Record Manager模块 7](#_Toc518233676)

[3.6 Index Manager模块 7](#_Toc518233677)

[3.7 Buffer Manager模块 7](#_Toc518233678)

[3.8 DB Files 8](#_Toc518233679)

[4、设计分工 8](#_Toc518233680)

[二、Interpreter模块设计报告 9](#_Toc518233681)

[1、模块功能 9](#_Toc518233682)

[2、具体实现 9](#_Toc518233683)

[2.1设计思路 9](#_Toc518233684)

[2.2 函数 9](#_Toc518233685)

[2.3 逐个实现 9](#_Toc518233686)

[三、API模块设计说明 13](#_Toc518233687)

[1、API主要功能 13](#_Toc518233688)

[2、成员函数 13](#_Toc518233689)

[3、设计细节 15](#_Toc518233690)

[四、Catalog Manager模块设计说明 16](#_Toc518233691)

[1、Catalog Manager主要功能 16](#_Toc518233692)

[2、字典文件 16](#_Toc518233693)

[3、数据结构 17](#_Toc518233694)

[4、成员函数 20](#_Toc518233695)

[5、设计细节 24](#_Toc518233696)

[五、Buffer Manager 模块设计说明 26](#_Toc518233697)

[1、功能说明 26](#_Toc518233698)

[2、总体设计思路 26](#_Toc518233699)

[3、具体实现 26](#_Toc518233700)

[3.1 数据结构 26](#_Toc518233701)

[3.2 函数/接口说明 28](#_Toc518233702)

[4、设计优化 30](#_Toc518233703)

[六、Index Manager设计说明文档 31](#_Toc518233704)

[1、模块概述 31](#_Toc518233705)

[2、模块功能 31](#_Toc518233706)

[3、设计思路 31](#_Toc518233707)

[4、数据结构 32](#_Toc518233708)

[5、函数说明 33](#_Toc518233709)

[七、Record Manager模块设计说明 36](#_Toc518233710)

[1、模块说明 36](#_Toc518233711)

[2、设计思路 36](#_Toc518233712)

[3、函数说明 37](#_Toc518233713)

[4、功能描述 38](#_Toc518233714)

[八、测试部分 39](#_Toc518233715)

[1、测试用例设计 39](#_Toc518233716)

[2、测试环境与配置 39](#_Toc518233717)

[3、功能测试 40](#_Toc518233718)

[3.1控制 40](#_Toc518233719)

[3.2 创建表语句 40](#_Toc518233720)

[3.3删除表语句 40](#_Toc518233721)

[3.4 创建索引语句 41](#_Toc518233722)

[3.5 删除索引语句 41](#_Toc518233723)

[3.6插入记录语句 42](#_Toc518233724)

[3.7 选择语句 42](#_Toc518233725)

[3.8 删除记录语句 43](#_Toc518233726)

[3.9退出系统语句 43](#_Toc518233727)

[3.10 执行SQL脚本语句 43](#_Toc518233728)

[3.11 测试脚本运行结果 44](#_Toc518233729)

[4、边界测试 48](#_Toc518233730)

[4.1数据类型测试 48](#_Toc518233731)

[4.2 表属性定义 49](#_Toc518233732)

[5、测试结论与建议 49](#_Toc518233733)

[5.1 测试结论 49](#_Toc518233734)

[5.2 改进建议 49](#_Toc518233735)

# 一、Minisql总体设计报告

## 1、引言

### 1.1 编写目的

为了更好地介绍我们的minisql数据库管理系统，这里对系统进行一个总体的分析，说明各个层次的设计，使结构明朗，使读者对minisql的总体规划有一个很好的了解。

### 1.2 项目背景

指导教师：陈岭

组员：李明光，周益聪，汤烨春，成菲尔，徐雅涵

知识：运用了数据库系统的知识，结合了c++编程技术，通过合作，由五位同学共同完成。

### 1.3定义

minisql：微型数据库管理系统。

C++：面向对象编程语言。

SQL（structured query language 结构化查询讯语言）：数据库管理系统访问管理语言，和标准定义的SQL语言略有不同。

### 1.4 参考资料

《数据库系统概念》 Abraham Silberschatz

Henry F.Korth

S. Sudarshan

## 2、任务概述

### 2.1功能概述

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。

通过对MiniSQL的设计与实现，提高学生的系统编程能力，加深对数据库系统原理的理解。

### 2.2 运行环境

Windows 10 操作系统上面完成测试。

### 2.3 具体分析

数据类型

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

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。

## 3、minisql系统结构设计

### 3.1总体设计

**总体设计结构图如下所示：**

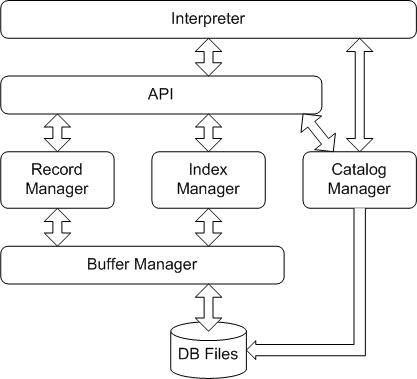


Figure 1 minisql总体设计图

### 3.2 Interpreter模块

Interpreter模块直接与用户交互，主要实现以下功能：

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。

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。

### 3.3 API模块

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。

### 3.4 Catalog Manager模块

Catalog Manager负责管理数据库的所有模式信息，包括：

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。

表中每个字段的定义信息，包括字段类型、是否唯一等。

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。

为减小模块之间的耦合，Catalog模块采用直接访问磁盘文件的形式，不通过Buffer Manager。

### 3.5 Record Manager模块

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。

### 3.6 Index Manager模块

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。

### 3.7 Buffer Manager模块

Buffer Manager负责缓冲区的管理，主要功能有：

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

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

记录缓冲区中各页的状态，如是否被修改过等

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。

### 3.8 DB Files

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

## 4、设计分工

**本系统分工如下：**

Interpreter模块：李明光

Buffer Manager模块：周益聪

Record Manager模块：徐雅涵

Index Manager模块：汤烨春

Catalog Manager模块：成菲尔

API模块：成菲尔

# 二、Interpreter模块设计报告

## 1、模块功能

Interpreter模块直接与用户交互，主要实现以下功能：

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。

接收并解释用户输入的命令，生成命令的内部数据结构表示，同时检查命令的语法正确性和语义正确性，对正确的命令调用API层提供的函数执行并显示执行结果，对不正确的命令显示错误信息。

## 2、具体实现

### 2.1设计思路

在实现本模块时，考虑的对语句解析后的各成分输出，将整个模块作为一个类来定义，即Interpreter类。各成分则作为该类的成员。于是就可以实现通过一条SQL语句对一个Interpreter实例进行初始化，然后返回整个实例以进行下面对应的操作。而对语句的具体解析过程则放在了此类的构造函数中。

### 2.2 函数

选择将各个语句分一类，然后分别处理和实现。

成员函数1）int Interpreter::interpreter(string s) 核心的构造函数

2）word = getWord(s, &tmp); 实现构造函数所要调用的子函数

### 2.3 逐个实现

按照大纲要求：

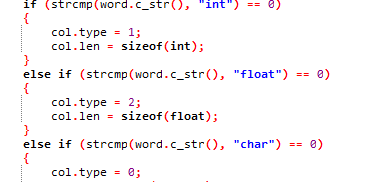
1. **Create**

**Create table ：**

**定义主键和表名：**



**处理数据类型**



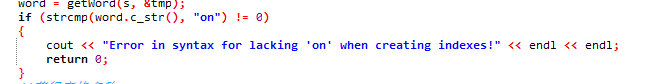
**处理主键：**



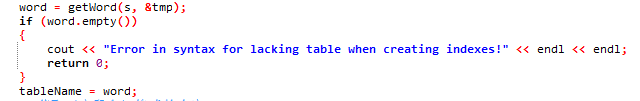
**Create index**

**获取索引名称之后**

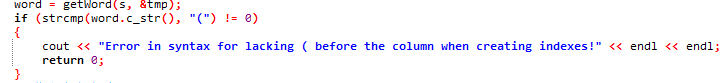
**获得on**



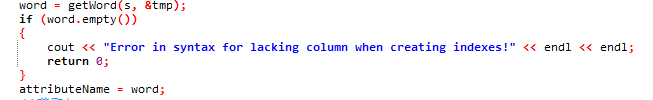
**获得表格名称：**



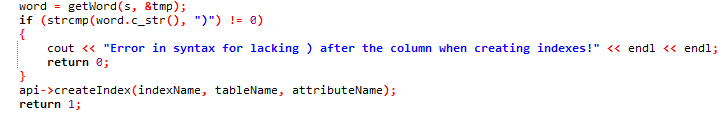
**获得格式内容**



**获得字段名称**

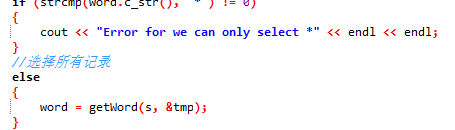


**获得“）**”



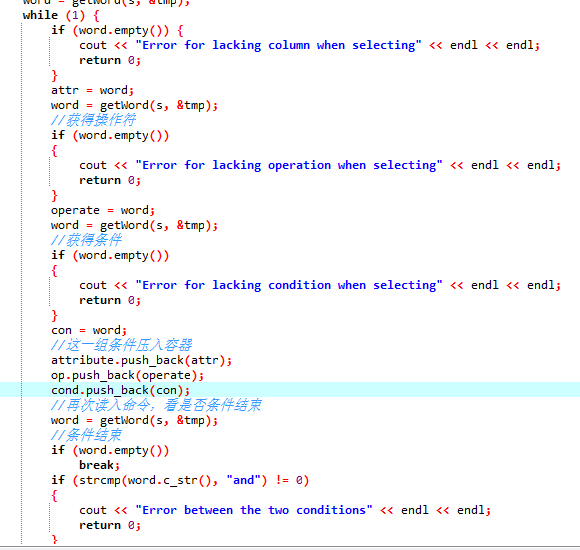
1. **选择语句select**

**分为两种情况：select from：**



选择所有记录或者并非选择所有记录。

**和select from where：**



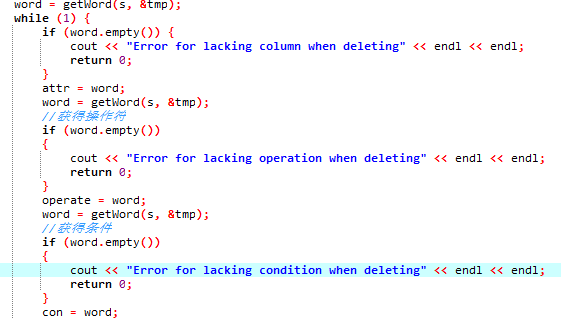
1. **删除索引语句drop**

这个相对简洁：获取表格名称后按步书写。

1. **delete**

**获取表格名称之后判断是否是无条件删除：**

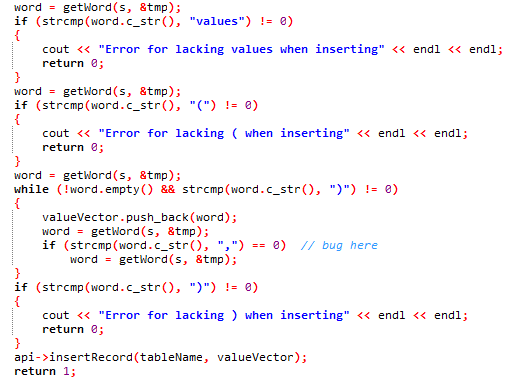
看一下**delete from where：**



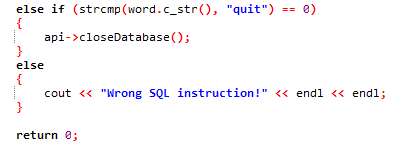
1. **insert**

获取表格名称之后

**获取后面的values**



1. **quit**



# 三、API模块设计说明

## 1、API主要功能

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。

在我设计的API模块中，程序根据Interpreter模块给出的命令，调用Catalog Manager、Index Manager、Record Manager三个模块中的接口，来执行输入的各个语句。API模块是把其他五个模块结合到一起的枢纽，实现各个函数的调度使用。

## 2、成员函数

由于API中的函数大多只是执行一个调用其他模块已存在的函数的功能，且需要与数据库将要实现的功能一一对应，所以API中函数较少，目的性强。

1. 根据interpreter传入的表格名称和字段容器，调用Catalog Manager模块中的函数，创建一个表格。

//根据列、主键等信息创建表

void createTable**(**string name**,**vector**<**Catalog\_manager**::**Column**>&**cols**);**

1. 根据interpreter传入的表格名称，调用Catalog Manager模块中删除表格函数，Record Manager模块中删除一个表格中所有记录的函数，删除包括字典和内存中所有与该表格有关的信息。

//删除一个表，包括删除目录中表格和内存中所有记录

void dropTable**(**string tableName**);**

1. 根据interpreter传入的索引名称、所建立索引的表格和属性名称，调用Catalog Manager模块中创建索引的函数，创建一个新索引。

//根据给定的表、属性，创建索引

void createIndex**(**string name**,** string tableName**,** string colName**);**

1. 根据interpreter传入的索引名称，调用Catalog Manager模块中删除索引的函数，删除指定索引。

//删除指定索引

void dropIndex**(**string name**);**

1. 根据interpreter传入的表格名称和所插入记录按照字段顺序的string类型信息，调用Record Manager模块中的插入记录函数，传递相同的参数，插入该记录。

//向表中插入入记录,传入的参数是表格名称和一个按照顺序存储各个字段的数据的容器

void insertRecord**(**string tableName**,** vector**<**string**>** v**);**

1. 根据interpreter提供的表格名称，调用Record Manager模块中的查询记录函数，按照第一行为各个属性名称，下面对齐显示相应的属性信息的格式，输出该表格中的所有记录。

//打印一个表中所有的记录和打印的记录数

void printRecord**(**string tableName**);**

1. 根据interpreter提供的表格名称，“属性+操作符+条件”一一对应的三个string类型的容器，调用Record Manager中可以多条件查询的函数，传入相同的参数，输出表格中符合条件的记录。

//根据where条件打印表中的记录和打印的记录数，可以有多组and条件

void printRecord**(**string TableName**,** vector**<**string**>** **&**columns**,** vector**<**string**>** **&**op**,** vector**<**string**>** **&**conditions**);**

1. 根据interpreter提供的表格名称，调用Record Manager模块中删除表格所有记录的函数，这个函数在Record Manager模块中会自行调用Catalog Manager模块中删除表格中所有记录的函数，来更新表格中记录的信息，从而删除指定表格中内存中的所有记录并更新字典信息。

//将表中的记录全部删除,同时输出删除的记录数目

void deleteValue**(**string tableName**);**

1. 根据interpreter提供的表格名称，“属性+操作符+条件”一一对应的三个string类型的容器，调用Record Manager中可以多条件删除的函数，传入相同的参数，删除表格中符合条件的记录。

//根据where条件删除表中的记录，同时输出删除记录的数目

void deleteValue**(**string tableName**,** vector**<**string**>** **&**columns**,** vector**<**string**>** **&**op**,** vector**<**string**>** **&**conditions**);**

1. 在用户输入quit命令时，为interpreter提供关闭数据库的函数，调用Catalog Manager模块中写入文件函数，更新并保存字典信息。

//关闭数据库

void closeDatabase**();**

## 3、设计细节

1. 删除一个表格时，不仅要删除字典信息中的表格、索引，还要删除内存中的索引，与该表格中所有记录占的内存。
2. 在有条件的查询和删除中，传入的参数在表达条件时使用了三个数组，分别是一一对应的属性、运算符和条件。
3. 由于有的功能需要调用不同模块的函数，而这些函数在某些数据信息上是有依赖性的，所以要注意调用顺序。

# 四、Catalog Manager模块设计说明

## 1、Catalog Manager主要功能

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、表格上的各个字段、主键、定义在该表上的索引。
2. 表中每个字段的定义信息，包括字段名称、字段类型、是否唯一、是否是主键、该字段数据类型在内存中的长度以及根据其他成员的要求所添加的一些信息。
3. 数据库中所有索引的定义，包括索引名称、所属表、索引建立在哪个字段上、以及根据其他成员要求所添加的一些信息。
4. 数据库中表格和索引的数目。
5. 每个表格中记录的条数。

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。

为减小模块之间的耦合，Catalog模块采用直接访问磁盘文件的形式，不通过Buffer Manager。

## 2、字典文件

为了存储上面提到的Catalog Manager需要管理的各个信息，我会创建一个相应的字典文件“db.info”进行记录。

在Minisql开始执行时，程序会从这个文件中读入该数据库中已经存在的信息，并存储到程序运行时内部的数据结构中。在执行一系列sql语句时，这些信息会随之改变。在关闭Minisql时，这些可能已经被改变了的信息就会写入“db.info”文件，以储存本次使用Minisql的结果，供下一次再次使用。

所以，“db.info”中内容存储的格式应该是固定且易懂的，在这里，我设置为如下格式：

文件开头放置表格和索引数量。

用“@”来表示开始记录表格信息，且每张表的信息开始前都会出现“@”。

“@”后面紧接着表格的名称，第二行是表格属性的数量，下面每一行里面放置一个字段的信息，包括名称、类型、长度、是否主键和唯一。

另起一行记录该字段上的索引名称。主键字段上自动会生成索引，该索引的名称是“表格名+属性名”。如果该字段上不存在索引，则记录为“NULL”。

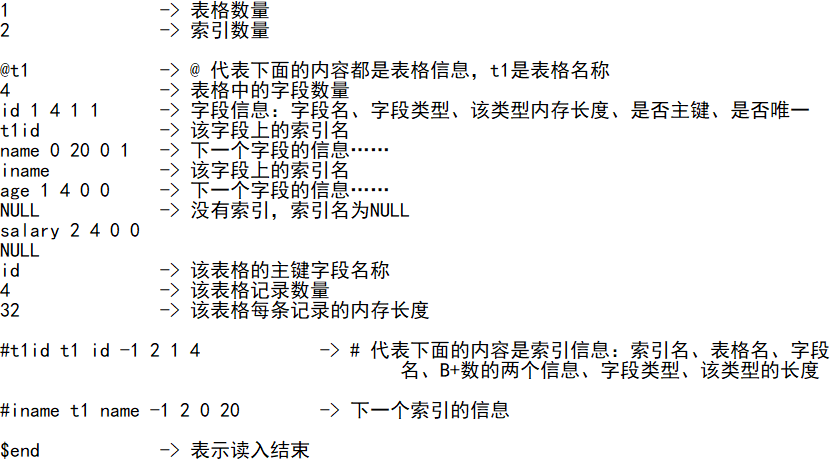
下面依次记录下一个字段的各个信息。

字段信息全部记录完成后，下面三行分别记录三个信息：该表格主键的字段名称（没有则为“NULL”），该表格中记录数量，每条记录的内存长度。

用“#”来表示开始记录索引信息，且每行记录一个索引信息，包括：索引名称、索引所在的表格名称、索引所在的属性名称、按照其他成员要求记录的该索引的B+数的两个信息、索引在的字段的类型、这种类型在内存中的长度。

用“$end”来表示字典内容结束。

下面是运行过Minisql后，用记事本打开“db.info”的结果图，其中“->”和后面的内容都是为了可读性添加的注释，在真正的文件中并不会存在。



## 3、数据结构

为了储存数据库中表格、表格中的属性、索引这三个对象的信息，我建立了三个结构Table、Column和Index。

Column数据结构如下图：

struct Column

**{**

string column\_name**;** //属性名称

int type**;** //属性类型，仅三种：char 0, int 1, float 2

int len**;** //若为char类型，则为char(n)中的n+1（放\0）；

//若为其他类型，则为其他类型的大小

bool is\_pri**;** //是否主键

bool is\_unique**;** //是否unique

**};**

在我设计的Column数据结构中，储存了每个属性的属性名称、属性类型、对应类型的内存长度、是否主键、是否唯一的信息。其中，属性类型我是用int类型数据表示，char时为0，int时为1，float时为2，而属性长度在属性是char（n）时记录为n\*sizeof(char)，在属性为int型和float型时则是这两个类型数据在内存中所占用的空间。是否主键、是否唯一的信息我使用bool类型数据表示。

Table数据结构如下图：

struct Table

**{**

string table\_name**;** //表格名称

vector**<**Column**>** clolums**;** //表格的各个属性

int colu\_num**;** //表格属性的数目

int record\_num**;** //表格中记录的条数

int record\_len**;** //表格中存放记录的长度

string pri\_key**;** //表格的主键

vector**<**string**>** index**;** //对应字段上的索引信息

**};**

在我设计的Table数据结构中，储存了每个表格的信息，包括表格名称、表格的各个属性，表格属性的数目，表格中记录的条数，表格中存放的每条记录的内存长度，表格的主键和对应字段上的索引名称。

为了存储表格中各个属性的信息，我使用了容器vector来存储Column数据结构，同时，对应的索引名称我也是用string的容器来存储，这样能够实现字段与字段上的索引名一一对应。

Index数据结构如下图：

struct Index

**{**

string index\_name**;** //索引名称(table-column)

string index\_table**;** //索引所在的表格名称

string index\_col**;** //索引所在的字段名称

int freeNum**;** //空链表

int root**;** //根节点

int type**;** //所建字段的类型：char 0, int 1, float 2

int len**;** //若为char类型，则为char(n)中的n+1（放\0）；

//若为其他类型，则为其他类型的大小

**};**

在我设计的Index数据结构中，存储了索引的各个信息，包括索引名称、索引所在的表格名称，索引所在的字段名称。又由于一个索引对应一个B+树，所以我的索引结构中还存储了指示空链表、根节点的两个整型数据。又为了给其他成员储存数据到内存中时方便，我还记录了所在字段的类型和内存长度。

除了上面的三个结构，我还定义了四个成员变量：

vector **<** Table **>** tables**;** //设定一个存放表格的容器

vector **<** Index **>** indexs**;** //设定一个存放索引的容器

int table\_num**;** //表格数量

int index\_num**;** //索引数量

两个整形变量分别用来存储这个数据库中表格和索引的数量；两个容器数据，分别用来存储数据库中的各个表格和索引。

## 4、成员函数

首先，既然Catalog Manager需要执行的功能包括：创建表格、删除表格、插入记录、删除记录、创建索引和插入索引，那么针对这些功能，我设计了一系列函数。

**创建表格：**

大纲中要求实现的创建表格的语句格式如下：

create table 表名 (

列名 类型 ,

列名 类型 ,

列名 类型 ,

primary key ( 列名 )

);

这样格式的语句经过interpreter后，会把表格属性各个信息处理好并把所有字段存入一个字段容器中，与表格名称、主键名称一起传递给API。为了更新数据库字典中的信息，我在Catalog Manager也提供了一个创建表格的函数：

//使用给定的名称和字段创建表格

int Create\_table**(**string name**,** vector**<**Column**>&**cols**);**

这个函数会接受要创建的表格名称和各个字段的信息。首先，函数内部会判断该名称的表格是否已存在，若存在，则输出错误提示，否则开始创建，并在函数内部对传入的字段容器分析，获得字段数量、每条记录的内存长度、每个字段对应的索引（如果没有则为“NULL”）、表格的主键（如果没有则为“NULL”）。另外，如果表格存在主键的话，则会自动生成一个索引，索引名就是“表格名”+“字段名”，并调用创建索引的函数。最终把信息装填成功的Table结构压入tables容器，更新表格总数目，并输出成功提示。

**删除表格：**

大纲中要求实现的删除表格的语句格式如下：

drop table 表名 ;

经过interpreter的处理后，要删除的表格名称会传递到API模块中，并调用Catalog Manager中的函数：

//删除给定名称的表格

int Delete\_table**(**string name**);**

在这个函数里，我会首先判断该名称的表格是否存在，若不存在，则输出错误提示，否则开始删除。删除一张表，不仅要吧tables容器中的那个表格删除，还要更新整个数据库表格的数量，并且要把建立该表格上的所有索引删除。这里我会调用另一个函数：

//删除指定表格上的所有索引

int Delete\_index\_ontable**(**string table**);**

这个函数会通过表格名找到该表格对应的Table结构数据，然后找到Table结构中的index容器，从而找到这张表格上的所有索引，并调用删除索引的函数来全部删除。

这些全部完成之后，输出成功提示。

**创建索引：**

大纲中要求实现的创建索引的语句格式如下：

create index 索引名 on 表名 ( 列名 );

经过interpreter处理后，会传递索引名、表格名和属性名到API模块，在API模块里调用我的Catalog Manager函数：

//对给定表格的字段创建索引

int Create\_index**(**string name**,** string column**,** string table**);**

在这个函数里，我会首先判断该表格该属性是否存在，属性上是否已经有了索引，如果是，那么会输出错误提示，否则，我会把传入的参数信息存入一个Index结构中，并根据传入的属性信息，把相应表格中与index有关的容器更新，结构中的根节点、空节点和索引总数目更新，并调用Index Manager创建索引的函数来创建一个相应的B+树。最终把这个更新好的Index结构压入indexs容器中，并输出成功提示。

**删除索引：**

大纲中要求实现的删除索引的语句格式如下：

drop index stunameidx;

经过interpreter把要删除的索引名称传入API模块后，调用Catalog Manager模块的函数：

//删除指定索引

int Delete\_index**(**string name**);**

首先这个函数会判断该索引是否存在，若并不存在，则输出错误提示，否则开始删除。删除一个索引，不仅需要从indexs容器中删除这个结构，还需要找到索引所在表的Table结构，将里面的index数据更新。完成这些后，把索引总数更新，并输出成功提示。

**插入/删除记录：**

由于我设计的Table数据结构中还存储了还表格中的记录的条数，所以每次插入或删除记录，虽然不需要对字典文件做什么操作，但是都需要更新记录数，这里我设计了三个函数：

//插入记录

int Insert\_rec**(**string table**,** int num**);**

//删除表格中所有记录

int Delete\_all**(**string name**);**

//删除表格中指定条的记录

int Delect\_recs**(**string name**,** int num**);**

第一个函数传入的参数分别是插入的表格名称和插入记录的条数，第二个是删除指定表格的所有记录，第三个是删除表格中的特定条记录。

**其他函数：**

除了上面的函数，Catalog Manager还设计了其他函数，用来提供辅助功能或者是为其他模块服务，其中包括：

//构造函数

Catalog\_manager**();**

功能：本模块的构造函数，在数据库启动时执行，从字典文件“db.info”（若不存在则新创建）中读入相应格式的数据并存入程序内部的数据结构中，以供后面的数据库操作使用。

//返回指定表格字段上的索引

Index**&** Get\_index**(**string table**,** string column**);**

功能：供Index Manager模块调用，返回指定表格指定索引上面的索引结构，让Index Manager模块能够从外部改变Catalog Manager内部变量的信息。

//根据表名获得所有字段

vector**<**Column**>** Get\_table\_col**(**string table**);**

功能：返回指定名称表格的字段容器。

//查找文件判断该表格是否存在

bool Table\_exist**(**string name**);**

功能：通过表格名称查找tables容器，判断这个名称的表格是否存在。

//查找文件判断该索引是否存在

bool Index\_exist**(**string name**);**

功能：通过索引名称查找indexs容器，判断这个名称的索引是否存在。

//判断指定表格字段上是否存在索引

bool has\_index**(**string table**,** string colName**);**

功能：通过指定的表格名称和属性名称，判断这个表格这个属性上是否建立了索引。

//计算一个字段类型的长度

int Calculate\_clo**(**Column C**);**

功能：通过输入的属性数据结构，计算存储在该属性下的数据需要的内存长度。

//计算指定多字段的记录长度

int Calculate\_clos**(**vector**<**Column**>&**cols**);**

功能：通过输入的属性数据结构容器，计算储存在这些属性下的数据所需要的内存长度。

//计算指定表格中每条记录的长度

int Calculate\_rec**(**string name**);**

功能：通过输入的表格名称，计算在这个表格下储存一条记录所需要的内存长度。

//计算指定表格中所有记录的长度

int Calculate\_recs**(**string name**);**

功能：通过输入的表格名称，计算储存在该表格下的所有记录的内存长度。

//返回指定表格在容器中的位置

int Table\_pos**(**string name**);**

功能：通过输入的表格名称，返回该表格在本tables数组中的位置。

//把信息写入文件

int Write\_in**();**

功能：在用户选择关闭数据库时，讲程序内部储存的数据结构中的信息按照一定的格式写入字典文件保存起来，用于下一次打开数据库时读入使用。

## 5、设计细节

1. 设计本模块的构造函数时，要注意字典文件不存在的情况。
2. 设计本模块的数据结构时，使用容器可以更加方便快捷地储存数据。
3. 设计创建表格函数时要注意，若用户设置了主键，那么要自动为主键创立索引。
4. 设计删除表格函数时要注意，有可能会有一些未删除地索引存在于表格上，所以，在删除表格前，需要先删除这些索引。

# 五、Buffer Manager 模块设计说明

## 1、功能说明

Record manager负责缓冲区的管理，主要功能有：

1. 根据需要，读取指定的数据到系统缓冲区或将缓冲区中的数据写出到文件；
2. 实现缓冲区的替换算法，当缓冲区满时选择合适的页进行替换；
3. 记录缓冲区中各页的状态，如是否被修改过等；
4. 提供缓冲区页的pin功能，及锁定缓冲区的页，不允许替换出去。

为提高磁盘I/O操作的效率，缓冲区与文件系统交互的单位是块，大小为4KB。

## 2、总体设计思路

记录管理模块（Record Manager）和索引管理模块（Index Manager）向缓冲区管理申请所要的块，缓冲区管理器首先在缓冲区中查看块是否存在，若存在，直接返回，否则，从磁盘中将数据读入缓冲区，然后返回。

当块满了之后，读取新块时需要替换老的块，用到的替换算法为LRU（最少使用）算法：使用链表存储块，每次替换块时将链尾的块删除，在链头加上新块。

改进：对链表中的所有块加上索引，加快判断块是否在链表中的判断。

## 3、具体实现

### 3.1 数据结构

总体数据结构分为块的定义和Buffer Manager的定义

**3.1.1 块**

struct Block\_t{

string filename;

// specify which file this block belong to

bool busy;

// indicate whether the block is in use.

bool dirty;

// indicate whether the block is changed

size\_t ID;

// indicate the location. For example:

// ID=5 means the entry of the block is 5\*Block\_Size offsets to

// the begin of the file.

BYTE content[Block\_Size];

// content

Block\_t(const string& f, size\_t id)

:filename(f), busy(false), dirty(false), ID(id)

{

memset(content, 0, Block\_Size);

}

~Block\_t()=default;

void clear();

Block\_t &write\_back();

Block\_t& operator=(const Block\_t& b);

Block\_t& reset(const string& filename, size\_t ID);

};

每个块包含的信息有：

* 所属文件名。
* 是否可替换。
* 是否是脏块。
* 当前块的序号。
* 数据区。

**3.1.2 Buffer Manager**

class BufferManager{

public:

BufferManager()=default;

~BufferManager();

bool NewFile(const string&);

// create new file

bool DelFile(const string&);

// delete a file

BYTE\* GetBlock(const string &filename, size\_t ID, bool newBlock=false);

// get a block by filename and ID

size\_t TotalBlocks(const string& filename);

// get total number of blocks from a file

bool set\_busy(const string& filename, size\_t ID);

// set a block busy, and buffermanager will not

// replace this block until you set it free.

// return true if operation succeeded

bool set\_free(const string& filename, size\_t ID);

// set a block free

// return true if operations succeeded

bool set\_dirty(const string& filename, size\_t ID);

// if you write something into the block, remember

// to set it dirty

// return true if operations succeeded

void write\_all\_back();

// force all dirty blocks to write contents back to disk

#ifdef DEBUG\_MODE

void BlockList\_layout();

#endif

private:

using BlockMap\_t = unordered\_map<string, list<Block\_t\*>::iterator>;

list<Block\_t\*> BlockList;

// linked-list for storing blocks in memory

BlockMap\_t block\_index;

// hash map for accessing block in constant time

};

其中Buffer Manger需要维护以下几个数据结构：

* BlockList: 存储指向块的指针的链表。
* block\_index: 存储链表中指针的索引。

### 3.2 函数/接口说明

**3.2.1块**

Block\_t &write\_back();

将当前块写回磁盘。

Block\_t& reset(const string& filename, size\_t ID);

将当前块重置。

**3.2.2 Buffer Manager**

bool NewFile(const string&);

// create new file

bool DelFile(const string&);

// delete a file

创建一个新的文件与删除一个现存的文件。两个函数都会在执行成功后返回true，在执行失败后返回false。

size\_t TotalBlocks(const string& filename);

查询一个文件总共有多少块。当文件不存在时，返回0.

BYTE\* GetBlock(const string &filename, size\_t ID, bool newBlock=false);

获取某一个文件的某一块。当获取的块大于文件总的块数的时候，需要显式将newBlock设置为true。需要注意，申请新块只能申请当前所有块数+1的块。

bool set\_busy(const string& filename, size\_t ID);

// set a block busy, and buffermanager will not

// replace this block until you set it free.

// return true if operation succeeded

bool set\_free(const string& filename, size\_t ID);

// set a block free

// return true if operations succeeded

bool set\_dirty(const string& filename, size\_t ID);

// if you write something into the block, remember

// to set it dirty

// return true if operations succeeded

以上三个函数分别实现了：

* 将某个文件的块固定。
* 就某一个文件的块解除固定。
* 将某一文件的块标记为脏块。

像C/C++中动态申请内存一样，固定一个块后，不需要时需要将块手动释放，否则该块会一直留在块链表中。

而对某一个块进行修改后，需要将其标记为脏块，否则Buffer Manager会认为该块没有进行改动，而不将内容写回磁盘。

void write\_all\_back();

该函数是强制将所有的脏块写回磁盘。

## 4、设计优化

传统实现LRU算法时，仅仅使用了链表来提高替换的效率。但是只使用链表会带来一个问题：在判断所需要的块是否在链表中，需要遍历搜索，这个时间效率是O(N)的。

为了解决这个问题，我额外使用了hash map来将判断过程的时间复杂度降低到常数级别。

# 六、Index Manager设计说明文档

## 1、模块概述

Index Manager负责B+树索引的实现，实现B+树的创建和删除（由索引的定义与删除引起）、等值查找、插入键值、删除键值等操作，并对外提供相应的接口。

B+树中节点大小应与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。

## 2、模块功能

1. 创建索引：现在由Record Manager调用InsertOne函数实现，而当创建空索引文件时不需要做什么，只需要把索引文件的第一块第一个字节置为＃。
2. 删除索引值：根据需要，当一个记录被删除时，其相应的索引值也要删除，调用DeleteOne函数。
3. 插入索引值：当一条记录插入时，在索引文件中也应插入其值，调用InsertOne函数。
4. 查找某一记录的位置：当要查找某一记录的记录偏移量时，调用SearchOne函数实现。
5. 查找某一批记录的位置（<,<=,>,>=）：当要查找某一批记录的起始记录偏移量时，调用SearchMany函数实现。

## 3、设计思路

**块的设计**

把4kb的block作为B＋树的节点。在每个节点的最开始分别以“!”和“?”来代表叶子节点和中间节点。

**中间节点**

以“?”开头用以表示中间节点。接着紧跟一个指针指向所对应的block，然后存储key来表示block中的最小值。以此重复，形成4个block的节点。

? key key key

block

block

block

block

中间节点

**叶子节点**

以“!”开头用以表示叶子节点。接着紧跟一个指针指向所对应的record，其中record包括block和offset两个变量。然后存储key。依此重复。若该叶子节点为最后的叶子节点，则最后一个指针指向“end”字符；若不是，则指向下一个block。

！ key key key

block

record

record

record

叶子节点（非最后）

！ key key key

end

block

block

block

叶子节点（最后）

## 4、数据结构

1. SearchKey

*struct* SearchKey {

*int* type;

*int* length;

*char*\* value;

*int* block;

*int* offset;

    };

1. MiddleSearchKey

*struct* MiddleSearchKey {

*int* type;

*int* length;

*char*\* value;

*int* leftBlock;

*int* rightBlock;

    };

## 5、函数说明

1. 创建一个索引

//创建一个索引

*void* CreateIndex(string tablename, string column, string filename);

1. 删除索引

//删除一个索引

*void* DeleteIndex(string filename);

    //删除一个索引内所有信息

*void* DeleteIndex(string tablename, string column);

1. 查找一个值等于value的记录

//查找一个值等于value的记录, 没有则返回0

*int* SearchOne(string tablename, string column, string value, *int*& block, *int*& offset);

1. 查找一批>, >=, <, <=记录的地址

//查找一批>, >=, <, <=记录的地址

*void* SearchMany(string tablename, string column, string value, *int* type, string op, vector<*int*>& block, vector<*int*>& offset);

1. 查找一个搜索码可能存在的叶子号

//查找一个搜索码可能存在的叶子号

*int* SearchLeaf(string tablename, string column, string value);

1. 插入一条记录

//插入一条记录

*void* InsertRecord(string tablename, string column, *char* \* value, *int* block, *int* offset);

1. 删除一条记录

//删除一条记录

*void* DeleteRecord(string tablename, string column, *char*\* value);

1. 判断叶子或中间节点是否已满

//判断叶子或中间节点是否已满

*bool* IsFull(string filename, *int* block);

1. 判断块中的value值是否过少

//判断块中的value值是否过少

*bool* IsLess(string filename, *int* block);

1. 判断是否有富余的value值

//判断是否有富余的value值

*bool* IsAmple(string filename, *int* block);

1. 在叶子中删除一个节点

//在叶子中删除一个节点

*void* DeleteOne(string filename, *int* leaf, *char* \*value, *int* type, *int* len);

1. 向左借一条记录

//向左借一条记录

*void* BorrowFromLeft(string filename, *int* parent, *int* left, *int* right, *int* len);

1. 向右借一条记录

//向右借一条记录

*void* BorrowFromRight(string filename, *int* parent, *int* left, *int* right, *int* len);

1. 合并两个叶子

//合并两个叶子

*void* MergeLeaves(string filename, *int* parent, *int* left, *int* right, Catalog\_manager::Index & index);

1. 合并两个中间节

//合并两个中间节点

*void* MergeNodes(string filename, *int* parent, *int* left, *int* right, Catalog\_manager::Index & index);

1. 查找左节点

//查找左节点（没有则返回-1）

*int* FindLeft(string filename, *int* parent, *int* child, *int* len);

1. 查找右节点

//查找右节点（没有则返回-1）

*int* FindRight(string filename, *int* parent, *int* child, *int* len);

1. 块满时调用函数

//到叶子块满时调用函数

*void* InsertDivide(string filename, *int* oldLeaf, *int* newLeaf, SearchKey& key, MiddleSearchKey& mKey);

    //当中间块满时调用函数

*void* InsertDivide(string filename, *int* oldblock, *int* newblock, MiddleSearchKey& mKey);

1. 找到可能存在记录的叶子块的块号, 并存储路径

//找到可能存在记录的叶子块的块号, 并存储路径

*int* FindLeaf(string filename, *int* root, string value, *int* type, *int* len, vector<*int*> &path);

1. 插入节点

//在叶子节点中插入一个节点,成功则返回1

*int* InsertOne(string filename, *int* block, SearchKey &key);

    //在非叶子节点中插入一个节点,成功则返回1

*int* InsertOne(string filename, *int* leaf, MiddleSearchKey &key);

1. 找到一个可用的块

//找到一个可用的块

*int* FindEmptyBlock(string tablename, string column);

1. 创建一个新的节点

//创建一个新的叶子节点

*char*\* CreateLeaf(string filename, *int* block);

//创建一个新的中间节点

*char*\* CreateNode(string filename, *int* block);

1. 比较大小

//比较大小, 小于返回负值， 等于返回0， 大于返回正值

*int* Compare(*char*\* x, *char*\* y, *int* type);

*int* Compare(*char*\* x, string y, *int* type);

1. 输出b+树的结构（测试用）

//输出b+树的结构（测试用）

*void* printBTree(string tablename, string column);

# 七、Record Manager模块设计说明

## 1、模块说明

Record Manager负责管理记录表中数据的数据文件。主要功能为实现数据文件的创建与删除（由表的定义与删除引起）、记录的插入、删除与查找操作，并对外提供相应的接口。其中记录的查找操作要求能够支持不带条件的查找和带一个条件的查找（包括等值查找、不等值查找和区间查找）。

数据文件由一个或多个数据块组成，块大小应与缓冲区块大小相同。一个块中包含一条至多条记录，为简单起见，只要求支持定长记录的存储，且不要求支持记录的跨块存储。

## 2、设计思路

Record Manager的操作主要分三部分：表的创建与删除、记录的插入与删除、记录的查找。

**选择语句**

select \* from 表名;

select \* from 表名 where 条件;

若无可用的索引，则须对表文件进行遍历，并对每条记录，进行where条件匹配，若符合，则输出相应属性的值。

若有可用的索引，则可调用Index Manager模块的功能，查找符合where条件的记录，并输出相应属性的值。在最后输出被选择的记录的条数。

**插入记录语句**

insert into 表名 values (值)

根据Catalog Manager处理生成的初步内部数据形式，提取表名及记录，根据信息计算插入记录的块号，并调用Buffer Manager 的功能，获取指定的内存块，并将记录插入到内存中，同时修改脏位及表信息。

**删除记录语句**

delete from 表名;

delete from 表名 where 条件;

根据Catalog Manager处理生成的内部数据形式，提取表名及可用的索引。

若有可用的索引，则须根据索引来查找符合条件的记录，删除该记录并更新该表在数据字典中的信息，循环往复，直至所有符合条件的记录都被删除为止。

若无可用的索引，则须对表文件中的所有记录进行一次遍历，并对每一条记录都需要判别是否符合条件。若符合条件，则删除该记录并更新表在数据字典中的信息。在最后输出被已被删除的记录的条数。

## 3、函数说明

* 1. 向表中插入元组,函数内部会判断传进参数的数量是否满足条件，插入成功则返回1

int InsertRecord(string TableName, vector<string> &Record);

* 1. 判断是否有重复值, 没有返回-1

int IsExist(string TableName, vector<int>& index, vector<string>& value);

* 1. 向一个块中插入记录

void InsertAtBlock(string TableName, int block, vector<string>& Record);

* 1. 无条件的select语句,返回所有记录，无记录时只输出属性

int SelectRecord(string TableName);

* 1. 有条件的select语句

int SelectRecord(string TableName, vector<string> &columns, vector<string> &op, vector<string> &conditions);

* 1. 判断是否符合一个where条件

bool ConfirmToWhere(char\* value, int type, string op, string condition);

* 1. 判断一条记录是否符合判断条件

bool ConfirmToWhere(string TableName, char\* record, vector<string> columns, vector<string> ops, vector<string> conditions);

* 1. 删除表中所有记录，返回删除记录的数量

int DeleteRecord(string TableName);

* 1. 有条件的delete语句,返回删除记录的数量

int DeletetRecord(string TableName, vector<string> &columns, vector<string> &op, vector<string> &conditions);

* 1. 根据块号和偏移量得到数据

char\* GetData(string TableName, int block, int offset);

* 1. 根据块号和偏移量删除记录及其索引

void SetDelete(string TableName, int block, int offset);

* 1. 打印一条记录

void PrintRecord(string TableName, char\* record);

* 1. 遍历所有记录，返回某一列的所有value及其入口

void SelectValue(string TableName, string Column, vector<char \*> &Values, vector<int> &block, vector<int> &offset);

* 1. 判断一个字符串是否是整数

bool IsInt(string s);

* 1. 判断一个字符串是否是浮点数

bool IsFloat(string s);

## 4、功能描述

* 1. 数据文件的创建与删除
  2. 记录的插入和删除
  3. 查找操作（包括不带条件的查找和带条件的查找）
  4. 可支持的数据类型包括int,float,char(n)

# 八、测试部分

## 1、测试用例设计

|  |  |  |
| --- | --- | --- |
| 功能模块 | 输入数据/操作 | 可能的输出结果 |
| 创建表 | ·创建不含主键的表  ·创建含主键的表  ·重复创建已存在的表 | 创建成功/失败 |
| 删除表 | ·删除创建的表  ·重复删除已删除的表  ·删除不存在的表 | 删除成功/失败 |
| 创建索引 | ·正确创建索引  ·在非unique列上创建索引  ·在不存在的表上创建索引  ·在不存在的列上创建索引  ·重复创建已存在的索引 | 创建成功/失败 |
| 删除索引 | ·正确删除索引  ·重复删除已删除的索引  ·删除不存在的索引 | 删除成功/失败 |
| 选择 | ·批量测试不同的op值(=、<>、<、>、<=、>=)的正确性  ·测试and条件的正确性  ·在不存在的表中选择  ·在不存在的列中选择 | 选择的结果正确/错误 |
| 插入记录 | ·正确插入记录  ·插入记录主键冲突  ·在不存在的表中插入记录  ·插入数据的数据类型错误 | 插入成功/失败 |
| 删除记录 | ·批量测试不同的op值(=、<>、<、>、<=、>=)的正确性  ·测试and条件的正确性  ·在不存在的表中删除记录  ·在不存在的列中删除记录 | 删除成功/失败 |
| 执行脚本 | ·执行测试用例7个脚本文件  ·执行10000+条记录脚本文件 | 执行成功/失败  结果正确/错误 |
| 退出程序 | ·quit | 退出程序成功/失败 |

## 2、测试环境与配置

操作系统：Windows 10

编译环境：Visual Studio 2015

## 3、功能测试

### 3.1控制

通过手动输入和脚本输入，观察MiniSQL提示反应。

### 3.2 创建表语句

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| create table test(  id int,  name char(20),  salary float  );  (初次创建表) | 创建成功 | Successfully create table test . | Pass |
| create table test(  id int,  name char(20),  salary float  );  (重复创建表) | 创建失败 | There is table test already. | Pass |
| create table t1(  id int ,  name char(20) unique,  age int ,  salary float,  primary key(id)  );  (含主键) | 创建成功  创建默认索引 | Create index of id in tablet1 successfully  Successfully create table t1 . | Pass |

### 3.3删除表语句

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| drop table test;  (删除创建的表) | 删除成功 | Remove file test failed!  Drop table test successfully | Pass |
| drop table test;  (重复删除表) | 删除失败 | There is no table test | Pass |
| drop table p;  (删除不存在的表) | 删除失败 | There is no table p | Pass |

### 3.4 创建索引语句

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| create index iname on t1(name);  (初次创建索引) | 创建成功 | Create index of name in tablet1 successfully | Pass |
| create index iname on t1(name);  (重复创建索引) | 创建失败 | There is index of name in tablet1 already | Pass |
| create index testidx on t1(name);  (在同一列上创建不同索引) | 创建失败 | Fail to create index, for the index already exists. | Pass |
| create index testidx on p(name);  (表不存在) | 创建失败 | Fail to create index, for table p does not exist. | Pass |
| create index testidx on t1(noname);  (列不存在) | 创建失败 | Fail to create index, for coll noname does not exist. | Pass |
| create index testidx on test(name);  (非unique的列上创建索引) | 创建失败 | Error for the column is not unique! | Pass |

### 3.5 删除索引语句

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| drop index iname;  (删除创建的索引) | 删除成功 | Drop index iname successfully. | Pass |
| drop index iname;  (重复删除索引) | 删除失败 | There is no index iname | Pass |
| drop index testidx;  (删除不存在的索引) | 删除失败 | There is no index testidx | Pass |

### 3.6插入记录语句

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| insert into t1 values(1,'Jim',20,2000.00);  (向表t1中插入) | 插入成功 | Insert 1 record. | Pass |
| insert into t1 values(1,'Eleanor',16,2300.00)  (主键冲突) | 插入失败 | Do not allow duplicate values of id in Table t1 | Pass |
| insert into t1 values(10,'Jim',20,2000.00);  (索引列冲突) | 插入失败 | Do not allow duplicate values of name in Table t1 | Pass |
| insert into p values(1,'Jim',20,2000.00);  (不存在的表中插入语句) | 插入失败 | There is no table p | Pass |
| insert into t1 values('w','Jim',20,2000.00);  (数据类型错误) | 插入失败 | Worng data type!  Do not allow duplicate values of name in Table t1 | Pass |

### 3.7 选择语句

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| select \* from t1;  (选择全部) | 选择表t1中的所有记录 | 选择表t1中的所有记录 | Pass |
| select \* from t1 where id > 6  (测试op>) | 选择表t1中所有符合条件的记录 | 选择id = 7、8的记录 | Pass |
| select \* from t1 where id < 3;  (测试op<) | 选择表t1中所有符合条件的记录 | 选择id = 1、2的记录 | Pass |
| select \* from t1 where id >= 6;  (测试op>=) | 选择表t1中所有符合条件的记录 | 选择id = 6、7、8的记录 | Pass |
| select \* from t1 where id <= 3;  (测试op<=) | 选择表t1中所有符合条件的记录 | 选择id = 1、2、3的记录 | Pass |
| select \* from t1 where id = 6;  (测试op=) | 选择表t1中所有符合条件的记录 | 选择id = 6的记录 | Pass |
| select \* from t1 where id <> 6;  (测试op<>) | 选择表t1中所有符合条件的记录 | 选择id != 6的记录 | Pass |
| select \* from t1 where id > 4 and salary >= 2000.00;  (测试and) | 选择表t1中所有符合条件的记录 | 选择id = 6的记录 | Pass |
| select \* from p;  (不存在的表中查询) | 查询失败 | There is no table p | Pass |
| select \* from t1 where s>0;  (不存在的列) | 查询失败 | Error for lacking operation when selecting | Pass |

### 3.8 删除记录语句

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| delete from t1 where id > 7;  (测试op) | 删除成功 | Delete 1 records in t1 | Pass |
| delete from t1 where id < 4 and salary > 3000.00;  (测试and) | 删除成功 | Delete 1 records in t1 | Pass |
| delete from t1;  (删除全部) | 删除成功 | Delete 4 records in t1 | Pass |
| delete from p;  (不存在表) | 删除失败 | There is no table p | Pass |
| delete from t1 where s>10;  (不存在列名) | 删除失败 | Error for lacking operation when deleting | Pass |

### 3.9退出系统语句

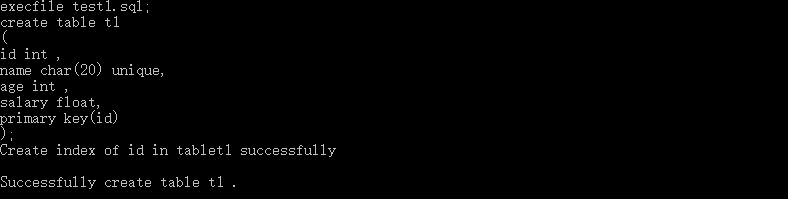
|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| quit; | 退出程序 | 退出程序 | Pass |

### 3.10 执行SQL脚本语句

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| execfile t1; | 文件脚本执行成功 | 文件脚本执行成功  (参见截图) | Pass |
| execfile employee.sql  select \* from employee  (10000+条记录) | 文件脚本执行成功 | 文件脚本执行成功  (参见截图) | Pass |
| execfile p;  (不存在的文件) | 导入文件失败 | fail to open file | Pass |

### 3.11 测试脚本运行结果

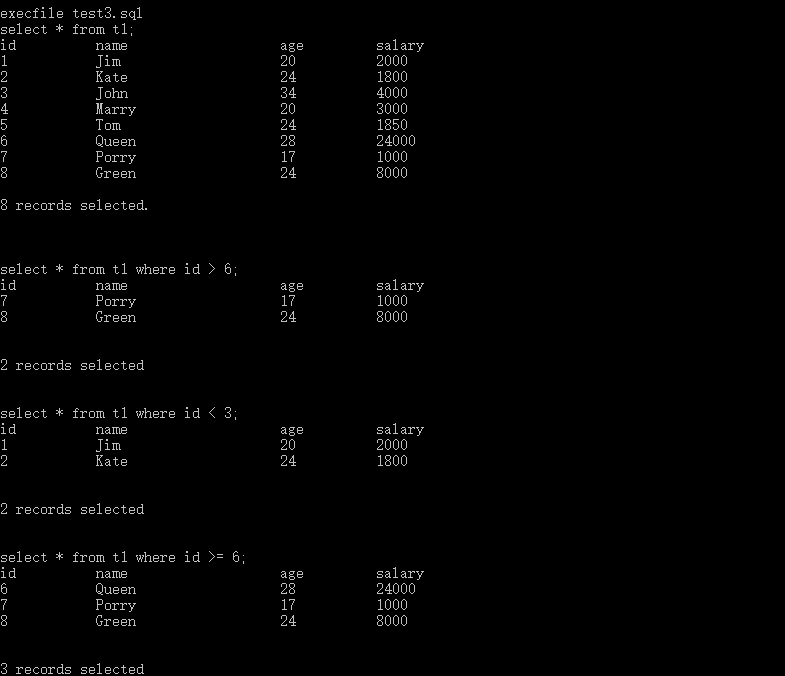
1. test1.sql

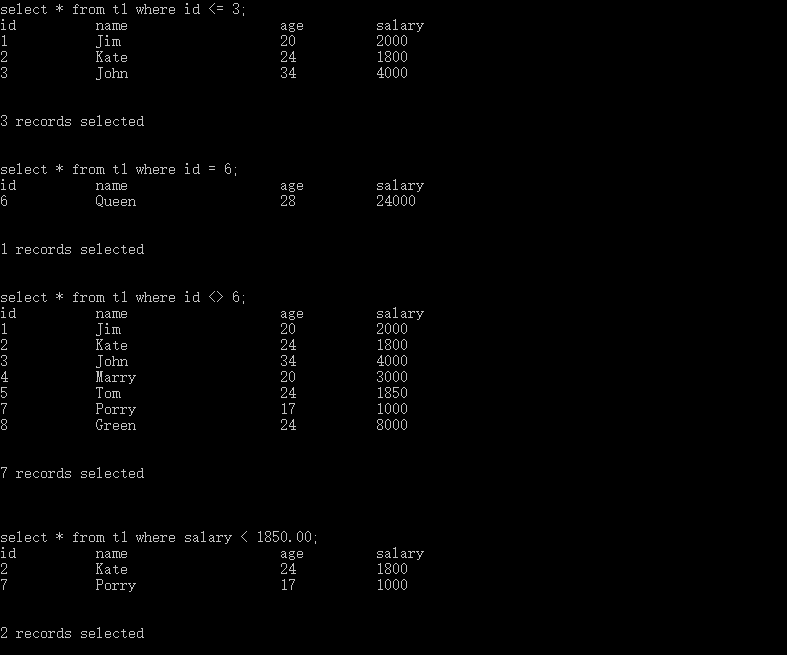


1. test2.sql

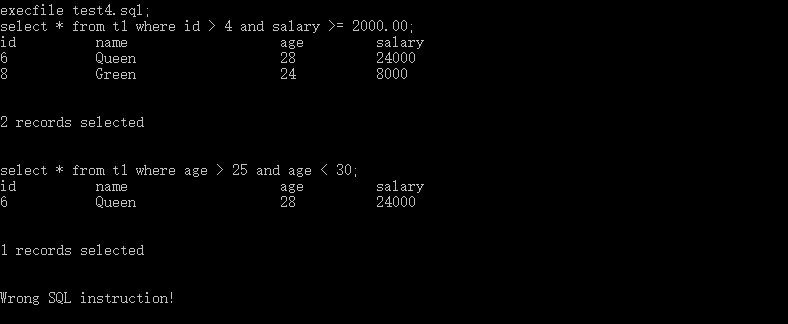


1. test3.sql

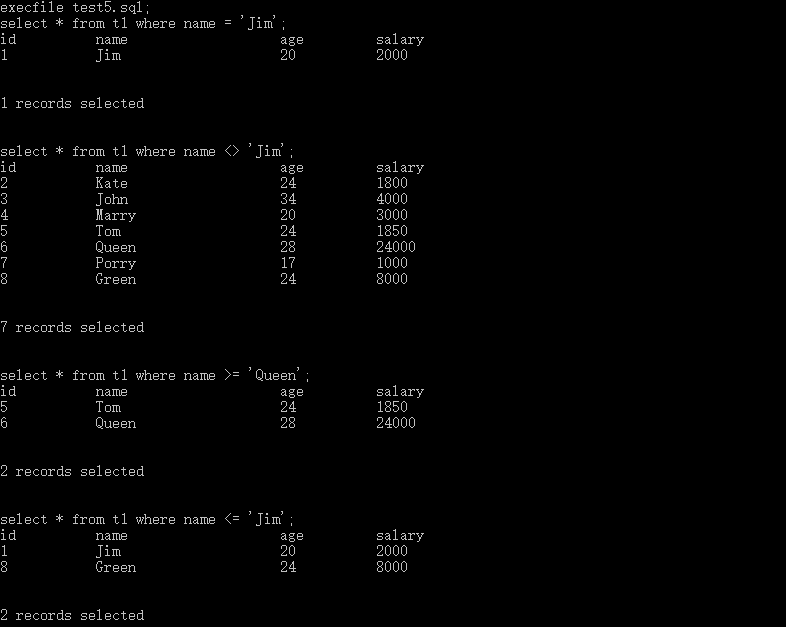


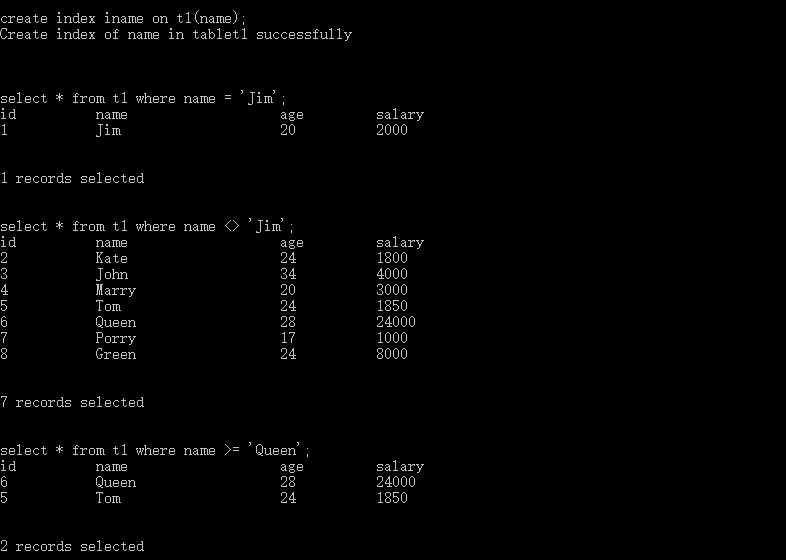


1. test4.sql



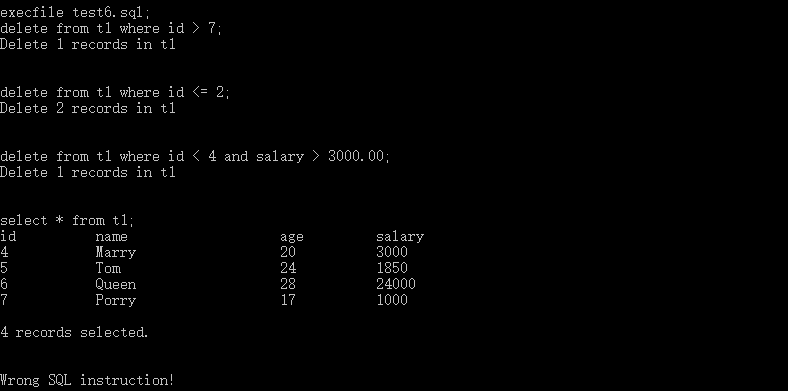
1. test5.sql



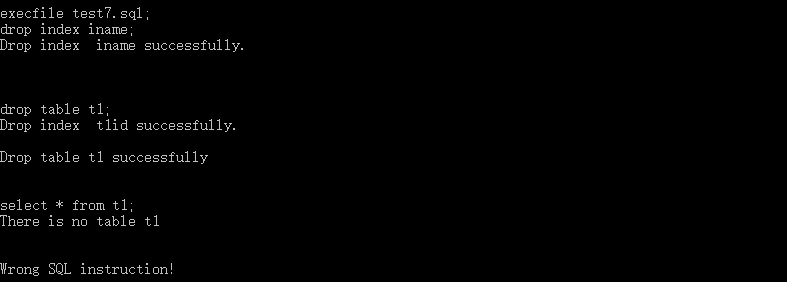




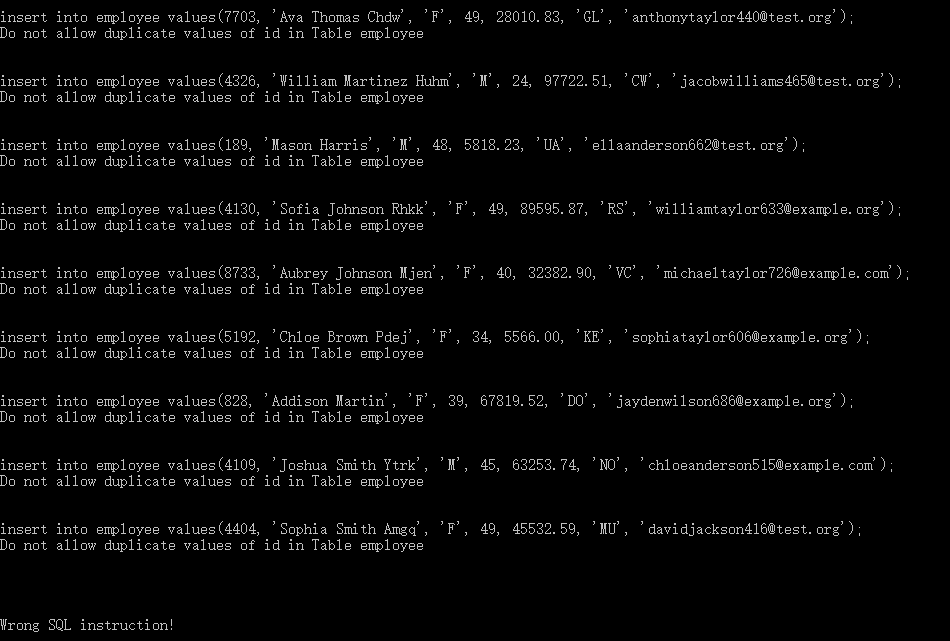
1. test6.sql



1. test7.sql



1. employee.sql（部分）



## 4、边界测试

### 4.1数据类型测试

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| create table test(  name char(0)  );  (char(n)满足 1 <= n <= 255) | 创建失败 | Successfully create table test . | Fail |
| create table test(  name char(1)  );  (char(n)满足 1 <= n <= 255) | 创建成功 | Successfully create table test . | Pass |
| create table test(  name char(255)  );  (char(n)满足 1 <= n <= 255) | 创建成功 | Successfully create table test . | Pass |
| create table test(  name char(256)  );  (char(n)满足 1 <= n <= 255) | 创建失败 | Successfully create table test . | Fail |

### 4.2 表属性定义

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 预期输出 | 实际输出 | Pass/Fail |
| 在一个表中定义33个属性（一个表最多可以定义32个属性） | 创建失败 | Successfully create table test | Fail |

## 5、测试结论与建议

### 5.1 测试结论

该MiniSQL能完成MiniSQL大纲上要求的所有内容。包括：创建/删除表，创建/删除索引，选择记录，插入/删除记录，同时能完成脚本文件的运行和程序的正常退出。同时运行效率高，在10000+条记录的情况下也能良好运行。

### 5.2 改进建议

在大纲要求的边界值上还可以有改进。