



**题 目 Mini SQL总体设计报告**

**组 员**

**曹闵丞、郑无恙、麦蓝、曾致语、崔翔源**

**指导教师 庄越挺**

**课程名称 数据库系统原理**

**专 业 计算机科学与技术**

**MiniSQL 总体报告**

**1.Minisql总体设计概述**

**1.1 Minisql实验平台**

设计语言：C++

编译器：Visual C++

**1.2 Minisql实现功能概述**

**1. 功能概述**：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。

**2. 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 。

**3. 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。

**4. 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）。

create index exi on example(ex3);

drop index exi;

**5. 查找记录：**可通过指定用and连接的多个条件进行查询，支持等值查询和区间查询。

**6. 插入和删除记录：**

插入：支持每次一条记录的插入操作

insert into test values ('test1', 1, 2.2, 'test2');

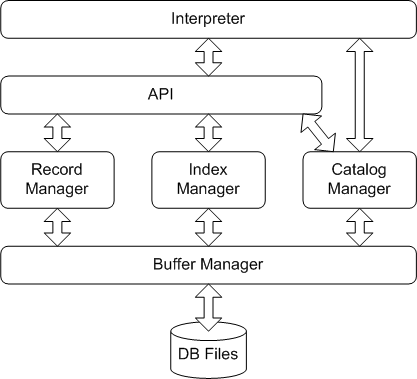
删除：支持每次一条或多条记录的删除操作

delete from test;

delete from test where t1='test1' and t2='test2';

7 退出系统：

**1.2 Minisql体系结构**



**2.模块介绍**

**BufferManager**

**2.1模块概述**

Buffermanger模块是用于管理缓冲区的。它的主要功能是负责内存与硬盘上文件及数据的交互。程序所有的数据读取与写入皆是直接在内存的缓冲区进行的，而这些操作完成后，再以buffermanager进行检查，然后决定是否要将相对应的缓冲区块写入硬盘上的文件。

Buffermanager管理了多个block，也就是缓存区。为了使其与硬盘的交互效率最大化，块的大小应为文件系统与磁盘交互单位的整数倍，因此我将缓存区的大小设为4KB,每一次读取文件或是向缓存区写入数据时，皆是以这4KB的文件块进行操作。

而进行写入或读取时都必须向管理这些操作的Buffermanager提出申请。如果文件已经在内存中，就返回内存块的编号，如果被要求的的文件当时不在内存中，则需要将文件加载至内存中，并将内存块的编号返回给请求者。

而其他的模块接受到Buffermanager返回的编号后，可以藉由这个编号对对应的缓存区进行访问。总而言之，就是模拟物理寻址的方法，buffermanager会创建及管理特定的缓存区位置，函数要对其中的数据进行访问前，必须要使用buffermanager返回的内存模块号码才能对此区域进行数据的存区。并藉此实现函数的功能，并且为了方便缓冲区与其他模块间的交互，block为全局变量。

在所有的block都是满情况下，会实现缓存区的替换算法（LRU），替换最长时间没访问的块。Block内的属性可以记录上次的访问时间（last\_vis\_time），以便于该功能的实现。

**2.2数据结构、接口**

**BufferBlock**

#define blockSize 4096 //内存块空间大小

#define maxBlocks 1000 //内存块最大数量

extern class **bufferBlock** blocks[maxBlocks]; //内存块

extern struct **Table** current\_table;

class **bufferBlock**

{

public:

    unsigned int blockpos;*//n-th block*

    unsigned int pos\_in\_block;*//position in the block*

**std**::string fname;

    char vals[blockSize + 1];

    bool dirty;

*//是否修改过*

    bool valid;

*//该block的有效性*

    unsigned int last\_vis\_time;

*//记录上次访问时间（用于替换算法）*

    unsigned int offset;

*//偏移量*

    void **init**();

*//初始化函数*

**string** **getvals**(int start, int end);

*//获得数值*

**bufferBlock**()*//初始化bufferBlock*

    {

**init**();

    }

};

**外部交互（用于外部函数调用）**

void **write**(int num);

*//在block被改写时对其进行标记*

void **used**(int num);

*//Block被读取时对其进行标记，记录其上次访问时间*

unsigned int **getaBlock**(**string** fname, int offset);

*//输入文件名称与偏移量，返回对应内存块编号，以便其他函数进行访问*

**bufferBlock** **find\_Insertable\_pos**(**table\_entity\_attribute** tabIn);

*//返回可插入数据的位置*

**内部交互**

void **writetoFile**(int num);

*//将block的内容立即写入文件*

void **read**(**string** fname, int offset, int num);

*//将文件读入block*

unsigned int **find\_Empty\_Buffer**();

*//寻找空block*

unsigned int **find\_Empty\_BufferExp**(**std**::**string** fname);

*//与上者类似，但不能替换参数指定的文件*

int **find\_assign\_buffer**(**string** fname, int offset);

*//判断指定文件是否在内存中，并返回编号*

unsigned int **inflate**(**table\_entity\_attribute**& tabIn);

*//在文件之后插入新的block,并返回新block的编号*

void **readinBuffer**(**table\_entity\_attribute** tabIn);

*//将整个表读入文件中*

unsigned int **blockNum**(**std**::**string** fname, int offset);

*//获得指定block在内存的编号*

void **invalid**(**string** fname);

*//将与参数所代表的文件相关的全部block标示为无效（invalid）*

void **changeblock**(**std**::**string** tname, int bn)

*//对block的属性进行修改*

**2.3 主要功能概述**

1. 作为程序与硬盘上数据间的接口，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. 记录缓冲区中各页的状态，如是否被修改过等

3. 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

4. 管理缓存区，在缓存块进行替换时，依据记录到的状态，判断内容是否需要被写入文件， 并加载新的模块

5. 实现缓冲区的替换算法（LRU），当缓冲区满时，选择距离上一次访问最久的页进行替换

6. 程序结束时，将所有脏数据（更动后的数据）写入文件中

**Catalog Manager**

**2.1模块描述与功能概述**

Catalog Manager函数的主要目的是将当前所使用的表中所有信息输入到名称为 Table\_xxx\_schema的文件上，并在下一次使用时从文件读出上一次存储的信息。接下来将对 这些函数进行具体介绍。

**2.2数据结构、接口**

class CatalogManager {

public:

void get\_table(string table\_name);

void create\_table(string table\_name, int num\_of\_col, string primary\_key);

bool table\_exist(string table\_name);

void create\_index(string table\_name, int attri\_no, string index\_name);

void drop\_table(string table\_name);

void drop\_index();

void update\_block(string& file);

};

**2.3函数介绍**

### 1. table\_exist函数

table\_exist函数通过ifstream测试存储当前所使用表的文件是否存在。若存在，则返回true； 不存在则返回false。

**2. create\_table函数**

函数首先判断该表是否已经存在其对应的schema文件。如果存在，则丢出数值为15的 异常；如果不存在，则创建一个格式为”Table\_xxx\_schema”的文件，其中xxx为参数 table\_name的字符串。接着将每一个raw\_data（即create table中的每一个属性）复制 到 current\_table全局变量（当前所使用的表）的attribute成员中。

然后判断是否存在primary key。若primary key存在（即其字符串大小为非零），则进 入对 应语句中。首先size被赋值为1，并初始化一个Index类型的变量pri\_index。 接着进入 循环，遍历当前表的每一个属性。若当前属性名称为primary key，则进入第二个 if的 语句 中，并使用table\_name、primary\_key以及当前属性的类型对Index Manager进行初始化。初 始化完成后，再调用CreateIndex函数创建格式为”Table\_xxx\_pri\_yyy\_index” 及”Table\_xxx\_pri\_yyy\_bptree”两个文件，其中xxx为当前表名，yyy则是成为primary key的属 性名。接着将当前primary key的信息（为当前表的第几个属性、index名称以及该属性的偏 移量）保存到当前表中，保存成功后跳出循环。若当前属性名称并非primary key，则计算当 前属性实际占用的内存空间并把计算结果加到size中。

接下来遍历当前表的属性并找出被定义为unique的属性，计算其内存空间偏移量并保存到当 前表的unique\_attri\_offset数组中。接下来进入update\_block函数。

### 3. update\_block函数

首先，调用Buffer Manager的getaBlock函数从buffer中找到一个属于当前表schema文件 的块，然后将当前表的数据逐一保存在这个块中，因此使用了大量memcpy函数。最后，调 用Buffer Manager的write函数表明当前表的所有数据已写入buffer中。

### 4.get\_table函数

可以理解为与update\_block功能相反的函数，即从buffer中寻找某个属于当前表schema文 件的块。找到后，将该块的数据读到当前表中。

### 5.drop\_table函数

首先判断待丢弃表的schema文件是否存在，若不存在则丢出数值为16的异常。接下来调用 Buffer Manager的invalid函数表明属于这个表schema文件的块已经不会再使用了，可以被 重写。然后调用remove函数把当前表的schema文件删除。若删除成功则弹出成功信息；删 除失败则弹出失败信息。

### 6. create\_index函数

首先第一个循环判断当前指定创建index的属性是否已经存在一个index，若存在则丢出数值 为17的异常。第二个循环判断index名是否已经被使用，若已被使用则丢出数值为18的异 常。接下来将index写入当前表中，并更新属于其schema文件的块。

**Interpreter**

**2.1模块概述**

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，对正确的命令进行执行

**2.2数据结构、接口**

class interpreter {

string s;

stringstream string\_proc;

public:

interpreter();

~interpreter();

void querydecode();

void Insert();

bool If\_int(string word, string& valid);

bool If\_char(string word, int char\_size, string& valid);

bool If\_float(string word, string& valid);

void CreateTable();

void DropTable();

void DeleteData();

void CreateIndex();

void DropIndex();

void Select();

friend istream&

operator>>(istream& is, interpreter& in);

friend ostream&

operator<<(ostream& os, const interpreter& in);

};

## **2.3 类型声明及实现思路**

class interpreter {

string s;

stringstream string\_proc;

interpreter中的成员变量分别是字符串变量s以及字符串流变量string\_proc。s的使用目的是 存储处理完成后的sql语句字符串。而string\_proc的功能是处理s中每一个“关键字”的变量。 其具体工作方式可通过成员函数进行展示。

interpreter的首要目的是将用户输入的sql语句转换成可供API、datatype以及各 manaager进行操作的数据结构（如raw\_data、current\_table等）。

## **2.4成员函数**

### operator>>函数

函数参数共两个变量，其中is变量为当前输入的SQL语句，in变量为interpreter类的对象。 首先使用getline函数读取is变量的输入流并存放到temp的字符串里，第三个参数为输入流 读取到该字符便停止继续读取（和mysql中的delimiter功能相同）。执行了getline函数后， temp便存放了当前的sql字符串。接下来进入循环，将字符串中回车、换行、以及缩进的字 符替换为空格的字符。处理完成后，将temp存放在in变量的s成员变量中。最后返回is变 量。

### querydecode函数

当用户输入了sql语句后，下一步便执行interpreter类的querydecode函数。首先将s读 取到string\_proc中，再从string\_proc读出一个“关键字”输出到word中，此时word便等于sql语句的第一个关键字。接下来将执行一系列的字符串比较，检查该关键字是否合法，若合法将进入相应的函数中。由于在进入字符串判断前使用了try关键字，因此如果调用的函数出现异常，则图表5的代码部分会接管这些异常并根据异常返回的数值输出对应的错误信息。异常处理完成后，对string\_proc内容清空，准备用户的下一轮的输入。

### Insert函数及If\_char函数

假如当前进入的是Insert函数，首先可以看到string\_proc输出了两次到table\_name中。这是 因为insert语句是按照”insert into xxx”格式输入的，因此这里就是将第二个关键字”into”跳过， 这样table\_name变量就能读取表名。下一步调用api的get\_table以及Table\_Tuple\_size函数。 其作用为从Buffer Manager中读取该表信息，比如该表的属性数量、每个属性的名称、类型 及状态（是否为unique、是否有index）等，接着存放到current\_table变量（于datatype.cpp 定义的全局变量，可以被所有Manager调用）中。

接下来在s中寻找开括号的位置，找到该位置后将s移动到开括号的下一个字符。下一步进 入循环，对每一个即将insert的变量检查其属性是否合法，即根据current\_table的第i个属 性进入对应的If\_int、 If\_char及If\_float函数。接下来仅介绍If\_char函数，其他两个函数只是 数据类型不同，但实现的效果相同。经过这一轮循环后，interpreter便能检查用户输入的values 是否合法，若有不合法的values，interpreter将弹出输入数据不合法（即ERROR 13）的错误 信息。若输入数据合法，则调用api将用户输入的数值插入到表中。

### 4. exec\_file函数

假如用户输入了”execfile xxx.sql”，那string\_proc会将”xxx.sql”输出到file字符串变量中，接着 利用file字符串对in文件流变量进行初始化。下一步判断该文件是否存在，若不存在，则显 示文件打开失败的错误信息。然后再定义一个run\_file的interpreter变量，利用上面提及的 操作符重载函数读入文件内的sql语句并执行，直至读取完文件所有内容。

### 5. saveblock函数

由于用户当前输入的数据都是存在buffer缓冲区上的，暂时还没有保存到文件上，因此这个 函数的作用便是将当前缓冲区的数据写回文件，防止信息丢失。用户可使用save;进入这个函 数。

**DB Files**

**2.1模块描述与功能概述**

DBFiles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。

进行通用数据结构设计

1.table\_entity\_attribute

2.Index

3.Data

4.Datai

5.Dataf

6.Datac

7.Tuple

8.Table

9.condition

实现Buffer与文件系统的交互

1.将Buffer里的内容写到文件中

2.将文件里的内容写到Buffer中

比较函数，根据datatype进行相应的类型转换，并按照比较结果返回相对应的数据

**2.2数据结构、接口**

**通用数据结构设计**

**属性结构**

struct table\_entity\_attribute

{

string name;

short type; // int = 0 , float = 1, char = 2

unsigned char type\_size;

short state; // normal attribute = 0 unique attribute = 1 normal attribute+index = 2 unique attribute+index = 3

bool is\_display;

};

name表示属性名称

type表示属性的数据类型，0为int，1为float，2为char

state表示属性的状态，普通属性为0，unique属性为1，index属性为2，同时为 unique和 index的属性为3

**索引结构**

struct Index

{

int num;

short location[MAX\_INDEX\_NUM];

string indexname[MAX\_INDEX\_NUM];

};

num表示索引数目

location表示索引在属性列中的位置

indexname表示索引名称

**数据项结构**

class Data

{

public:

short type;

virtual ~Data() {};

};

class Datai : public Data

{

public:

int x;

Datai(int i):x(i){

type = 0;

};

Datai() :x(0) {

type = 0;

};

virtual ~Datai() {};

};

class Dataf : public Data

{

public:

float x;

Dataf(float f):x(f){

type = 1;

};

Dataf() :x(0) {

type = 1;

};

virtual ~Dataf() {};

};

class Datac : public Data

{

public:

string x;

unsigned char type\_size;

Datac(string c):x(c){

type\_size = c.length();

type = 2;

};

Datac() :x("") {

type\_size = 0;

type = 2;

};

virtual ~Datac() {};

};

首先定义父类数据项Data，其中type用于标示数据类型，然后设计三个子类，存储不同类 型（int float string）数据。

Datai构造函数，type置为0，int x赋值

Dataf构造函数，type置为1，float x赋值

Datac构造函数，type置为2，string x赋值，typesize记录大小

**元组结构**

typedef vector<Data\*> Tuple;

typedef vector<Tuple> Tuples;

**buffer到文件**

//主要使用seekp与write

file.seekp(m\_blocks->offset\* BLOCK\_SIZE);

file.write(m\_blocks->address, BLOCK\_SIZE);

**文件到buffer**

//主要使用seekg与read

file.seekg(offset\* BLOCK\_SIZE);

file.read(m\_blocks[number].address, BLOCK\_SIZE);

**表结构**

struct Table

{

friend class CatalogManager;

int blockNum;

string table\_name;

short attribute\_num;

table\_entity\_attribute attribute[MAX\_ATTRIBUTE\_NUM];

Tuples tuples;

Index index;

short primary;

int unique\_attri\_offset[MAX\_INDEX\_NUM]; //not including indexed unique

Table();

Table(const Table& ori);

int get\_attribute\_size();

};

void Table::get\_attribute\_size()

{

length=0;

for(int i=0; i < attribute\_num; i++)

{

switch (attribute[i].type)

{

case 0:

length += sizeof(int);

break;

case 1:

length += sizeof(float);

break;

case 2:

length += attribute[i].type\_size + 1;

break;

}

}

}

Table::Table() {}

Table::Table(const Table& ori)

{

attribute\_num = ori.attribute\_num;

for(int i=0; i < attribute\_num; i++)

{

attribute[i].name = ori.attribute[i].name;

attribute[i].type = ori.attribute[i].type;

attribute[i].type\_size = ori.attribute[i].type\_size;

attribute[i].state = ori.attribute[i].state;

attribute[i].is\_display = ori.attribute[i].is\_display;

}

}

blockNum标示该表占据的数据块数

table\_name记录表名称

attribute\_num记录属性数目

attribute数组存储属性

primary记录主键的位置

Index为索引信息

**condition结构**

struct condition

{

string entity;

string op;

string restrict;

};

处理类似于“where salary > 5000”的where语句时，entity记录“city”属性，op 记录“>”运算， restrict记录“5000”限制条件

**compare函数**

int compare(void\* a, void\* b, int datatype);

int compare(Data\* a, void\* b, int datatype, int i);

int compare(void\* a, void\* b, int datatype, int i);

根据datatype进行类型的转换

如果datatype=0，转换为int\*（Datai\*）

如果datatype=1，转换为float\*（Dataf\*）

否则，转换为string\*（Datac\*）

进行数据的比较，根据比较结果返回

**API**

**2.1 实现思路**

API的作用为提供一个interpreter与各manager间的接口，并将manager的异常信息丢回 interpreter模块进行处理。接下来对各个成员函数进行介绍。

**2.2 数据结构、接口**

class API {

public:

void get\_table(string& table\_name);

void create\_table(string& table\_name, int& num\_of\_col, string& primary\_key);

void drop\_table(string& table\_name);

void delete\_data(where& wc);

void create\_index(string& table\_name, string& attri\_name, string& index\_name);

void select(vector<int>& attri\_proj, where& wc);

void insert(vector<int>& uni\_no);

void drop\_index(string& table\_name, string& index\_name);

};

## **2.3成员函数**

### 1.get\_table函数

负责调用Catalog Manager的get\_table函数，并计算当前表属性所占据的内存空间大小。

### 2.create\_table函数

负责调用Catalog Manager以及Record Manager的create\_table函数。

### 3.drop\_table函数

drop\_table函数首先调用另一成员函数get\_table以获取当前表信息，然后通过遍历当前表的 index成员调用Index Manager的Drop函数，删除对应的index及bptree文件。接着调用 Catalog Manager以及Record Manager的drop\_table函数删除schema以及tuple文件。最 后输出丢弃表成功的信息。

### create\_index函数

首先遍历当前表的属性，查找该表是否存在用户输入的属性名。若不存在则丢出数值为19 的异常。如果在当前表查找到该属性名，便调用Catalog Manager以及Record Manager的 create\_index函数。

### select函数

负责调用Record Manager的select函数。参数attri\_proj为当前表符合where判断条件的记 录中需输出的属性（即投影操作），参数wc为储存where判断条件语句的数据类型。

### delete\_data函数

负责调用Record Manager的Delete函数。

### insert函数

负责调用Record Manager的insert函数。

### drop\_index函数

drop\_index函数首先调用另一成员函数get\_table以获取当前表信息，然后寻找该表中是否存 在用户输入的index名。若不存在则丢出数值为19的异常；存在则在当前表中移除该index 并调用Catalog Manager的update\_block函数更新块，最后再调用Index Manager的Drop 函数移除该表对应的index及bptree文件。

**IndexManager**

**2.1 数据结构、接口**

class IndexManager

{

int col\_type;

string table\_index\_name;

public:

IndexManager(string name, int col\_type\_in) : table\_index\_name(name), col\_type(col\_type\_in) {}

void CreateIndex();

void Drop();

void Insert(Data\*& key,int offset);

void Delete(Data\*& key);

int Find(Data\*& key);

vector<int> FindRange();

// vector<int> FindRange(Data\*& keystart, Data\*& keyend);

int index\_exist();

};、

**2.2 函数介绍**

### 1.构造函数

储存对操作的index\_table的名及其索引类型，为之后的操作所用

### 2.CreateIndex

CreateIndex的实现代码。生成两个文件，Table\_ tablename\_ bptree和Table\_ tablename\_ index 这两个文件，前者为储存B+树的一些基本资料，如root的位置，index类型，B+树叉数，节 点个数等。另一个为储存B+树的结构和数据。创建两个文件后对B+树做一个初始化的动作， 即创建一个空的B+树。

**3.Drop**

这个函数的主要操作是通过buffer manager删除两个储存index table的文件，以达到删除索 引的目的

**4.Find**

这个函数的主要操作是在B+树中找出input key所对应的Address，并返回其值，若未建立 Index则在index\_exist中被识别出，然后返回-1

**5.Insert**

这个函数的主要操作是键值的插入，将输入的键值key和其数据插入到B+树中，插入过程可 在BpTree这个class中的Insert函数中体现，在插入之前先作两个判断，一是昰否存在该index table，二是该值是否己经在B+树中存在。如上述两个条件有其中一个成立，则不会执行Insert 操作，而是返回错误讯息。

**6.Delete**

这个函数的主要操作是键值的删除，和Insert一样，在插入之前先作两个判断，一是昰否存 在该index table，二是该值是否不存在于B+树中。如上述两个条件有其中一个成立，则不会 执行Delete操作，而是返回错误讯息。反而将其在B+树中删除。

**7.B+树的实现函数**

其中设计了每一个节点的结构为：

每当读取节点前会先生成一个Bpnode并初始化﹙申请空间﹚，然后从buffer中读取内存资 料到Bpnode的结构中，再作其他操作，完成操作后，都会把Bpnode结构中的资料通过函 数Bpsave储存回buffer中，若之后不会用到Bpnode的话，则调用freenode释放已申请的 内存

* + 1. 构造函数

这个函数的主要操作是把存在Table\_ tablename\_ bptree文件中的B+树的基本资料读出﹙通 过调用buffer manager﹚并放到class的内部变量中

* + 1. Bpsave

这个函数的主要操作是把存在class的内部变量的B+树的基本资料存回Table\_ tablename\_ bptree文件中﹙通过调用buffer manager﹚，为构造函数的逆操作

* + 1. Create

这个函数的主要操作是把初始化Root以及B+树，并保存

* + 1. Insert

这个函数的主要操作是输入的key以及Addr insert 到B+树中，先找到要插入的node，插入 后由叶结点开始判断当前结点是否满了，若否则保存后return，若是则执行splitnode并向上 递归。其中调用了大量的中间函数。

* + 1. Init

这个函数的主要操作是申请内存给Bpnode的结构data和pointer，用作储存数据及指针

* + 1. Find

这个函数的主要操作是在B+树中找到key所对应的Addr，若找到返回Addr，找不到则返回 -1。大致流程是先找到目标所在的node，然后在node中顺序查找。其中调用了大量的中间 函数。

* + 1. Deletenode

这个函数的主要操作是删除B+树中给定的Key及其Addr。首先也是找到key所在的node， 然后把它删掉，最后递归看看返回的路上有没有结点不符合要求。其中调用了大量的中间函 数。

* + 1. 中间函数

void init(int col\_type, int maxchildren);

int insert(Data\* key, int offset, int col\_type); //leaf

int insert(void\* key, int offset, int col\_type); // internal

void deleteRecord(Data\* key, int col\_type);//leaf

void deleteRecord(void\* key, int col\_type);//internal

void updateInternalKey(Data\* originKey, Data\*updateKey, int col\_type);

void cleannode(int numofnode,int&rootpos,int col\_type,int maxchildren, string table\_name);

//============================================================================================

void mergeLeaf(Bpnode& node, int col\_type, int maxchildren, string table\_name); // merge two leaf

void mergeInternal(Bpnode& node, int col\_type, string table\_name, int maxchildren); // merge two internal

int rightBorrowOK(int col\_type, string table\_name, int maxchildren);

int leftBorrowOK(int col\_type, string table\_name, int maxchildren);

int rightMergeOK(int col\_type, string table\_name, int maxchildren);

int leftMergeOK(int col\_type, string table\_name, int maxchildren);

void rightBorrowInternal (int col\_type, string table\_name, int maxchildren);

void rightBorrowLeaf(int col\_type, string table\_name, int maxchildren);

void leftBorrowInternal(int col\_type, string table\_name, int maxchildren);

void leftBorrowLeaf(int col\_type, string table\_name, int maxchildren);

//=============================================================================================

void splitLeafnode(Bpnode& splitnode, int col\_type, int maxchildren);

void splitInternalnode(Bpnode& splitnode, int col\_type, int maxchildren, string table\_name);

//==========================================================================================

void getNode(int nodeAddr\_in, string table\_name, int col\_type, int maxchildren);

void saveNode(int nodeAddr\_in, int col\_type, int maxchildren);

void freeNode(int col\_type);

以上函数主要是作一些对node的操作

int deleteBack(int curAddr, Data\* key, int col\_type, int& maxchildren, int& numofnode, int& rootpos, string table\_name);

int insertBack(int curAddr, Data\* key, int col\_type, int offset, int& maxchildren, int& numofnode, int& rootpos, string table\_name);

以上函数主要为由叶结点开始往上递归来维护不合符B+树特点的结点

**RecordManager**

## 2.1模块概述

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

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

在进行数据的文件处理与解析时，每次会根据用户的指令来执行相应的操作。例如当用户执 行 select 操作时，该模块会把所有满足select条件的数据转换成相应的元组，放到用于输出 的表格out\_table，用于显示select的输出。

## 2.2主要功能

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

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

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

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

## 2.3数据结构与接口设计

extern struct Table current\_table;

extern struct table\_entity\_attribute raw\_data[MAX\_ATTRIBUTE\_NUM];

extern class buffermanager bf;

extern class bufferBlock blocks[maxBlocks];

typedef enum {

eq, leq, l, geq, g, neq

} iswhere; // 判定where后面运算符的枚举集合

struct cond {

Data\* dataw;

iswhere flag;

string entity;

}; // 将where后面的条件分为标签，运算符和数据三个模块

struct where {

vector<cond>conditions;

int cnumber;

}; // 将where后面的condition存入动态数组并计算个数

### 外部接口

void create\_table(); // 创建表

void insert(); //插入数据

void select(vector<int>& attri\_proj, where& wc); // 执行select语句

void Delete(where& wc); //删除数据

void DropTable(string table\_name); //删除表

**内部接口**

where interwhere(vector<condition>& conditons); // 将where后面的条件数据化

void Table\_Tuple\_size(); // 得到Tuple的长度

void display(Table& out\_table, vector<int>& attri\_proj); // 显示select的输出

bool tuplesatisfied(Tuple& temp, int(&index)[MAX\_ATTRIBUTE\_NUM], where& wc);

// 在select 中判断单条元组是否满足输入的条件

void SelectProject(Table& out\_table, vector<int>& attri\_proj);

//没有where 语句的select

## 2.4设计思路

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

对于第一个问题，我们采用的方法是在数据存取转换时使用stoi,stof转化成字符串保存，把 其他所有的类型的数据都直接拷贝到长度相对应的字符串中，这种方法极大的降低了数据丢 失与损坏的可能性，且方便我们用比memcpy函数把相应的值拷贝到对应的数据区块。其中 对于字符串数据，不论字符串的原有长度是多少，我们都按照其最大的长度进行转换，这样 可以使得数据块的数据对齐，方便我们在读取数据时进行的操作。以下是一个数据转字符串 的实例：

Void insert()；

ti = stoi(raw\_data[i].name);

memcpy(begin + pos, &ti, sizeof(int));

我们按照这种方法就可以保证数据保存到文件中时一定是按照原来的格式保存的，保证了 数据不会丢失。

对于第二个问题，我们采用 Lazy deIete 的方法。由于这种方法需要一个位做标记，我们 需要额外在每一条数据前加一位来标识这条记录是否有效。插入数据是通过 getvals函数 来找到数据块中可用的空位置，然后我们就可以直接把数据插入到这些位置。

stringRow = blocks[No].getvals(position, position + length);

if (stringRow[0] == Empty) continue;

## 2.5实现方法

### Select 语句

函数原型：void select(vector<int>& attri\_proj, where& wc)

实现方法：

函数的第一个输入是记录select所选择的标签的attri\_proj，第二个输入是选择的条 件，也就是where子句后面的条件。对于原先的vector<condition>因为储存的三个变量 都为字符串，不便于操作，因此通过interwhere() 函数将where后面的条件分别转化为 数据，枚举集合，和标签字符串，作为vector<where>保存。

在实现 Select 时，要先判断 where 后面的条件是否为空，如果是空的，那就直接调用 SelectProject 函数。如果后面的属性非空，那就需要每次执行时先找到相应的表在硬盘 上的文件，然后通过 Buffer Manager将这些文件读取到内存中，然后再把数据放置到等 待输出的表格out\_table，通过display(Table& out\_table, vector<int>& attri\_proj) 显示 select语句的结果。如果where 条件不为空，那么就需要调用 tupleSatisified 函 数逐 条判断该条元组是否满足所给的条件，如果满足，就把该条记录插入到结果的表中。 一直这样循环，直到遍历玩所有的元组为止，然后将最后形成的表输出就可以得到结果。

### Insert 语句

函数原型：void insert();

实现方法:

表格Current\_table储存了需要插入的信息，插入时需要先获取可以插入数据的位置，然 后根据数据的不同格式，分别调用stoi,stof 函数，把要插入的元组的信息用之前提到的方 法转换成字符串，然后利用memcpy()将字符串直接写入缓存区中，并且调用 bf.write() 和 bf. writetoFile() 函数，表示已经对 buffer进行了写操作。

### Delete 语句

函数原型:void Delete(where& wc); //删除数据

实现方法:

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

### CreateTable 语句

函数原型：void create\_table(); // 创建表

实现方法:

CreateTabIe 所需要做的事情十分简单，就只需要新建立一个以表名为文件名的数据文件, 初始化blocknum为1，在调用bf.changeblock()函数改变使用的内存空间。

### DropTable 语句

函数原型：void DropTable(string table\_name); //删除表

实现方法:

要想删除一个表，需要把缓存区中所有有关这个表的数据块全都设置为无效，这个可以调 用 Buffer Manager 的 invalid ( ) 函数来实现，这样就可以直接删除这些还没有被写入到 文件中的函数。

**Record Manager (Index 部分)**

## 2.1实现思路

由于在实际操作上Record Manager与Index Manager有较高的关联性，因此需在Record Manager原有操作的基础上实现index的查找、插入以及删除操作。接下来将介绍本人在 Record Manager中实现index操作的函数。

## 2.2成员函数

### 1.CreateIndex函数

### 首先初始化Index Manager，然后调用CreateIndex函数创建对应的index及bptree文件。 接下来进入循环计算该index于当前表属性的内存空间偏移量。

然后进入图表32的双重循环。第一个循环将遍历当前表正在使用的所有块号，接着计算每个 记录在块中的偏移量，即position的数值（比如当offset=0时，当前position指向第一个记 录；当offset=1时，当前position指向第二个记录）。计算完position后，加上刚才计算出 的index偏移量，即可取出当前记录为index属性的数值。

### 2.Insert函数（Index部分）

Insert函数首先会使用Search\_with\_Index函数以及Search\_without\_Index函数查找当前表插 入过的所有记录，以判断当前插入的数据是否合法（使用了index、状态为unique或primary key的属性不允许出现重复的数值）。其中使用了index以及状态为primary key的属性，将 会使用Search\_with\_Index函数进行查重。后续部分将详细介绍Search\_with\_Index函数。

当调用了Buffer Manager的find\_Insertable\_pos函数后。此时iPos变量便指向了当前可插入 块的偏移量。第一个循环遍历当前表的每个属性，寻找是否存在使用了index/primary key的 属性。若该属性存在，则进入第二个循环，寻找当前表中使用了index的属性。接着将Index Manager以及temp变量根据该属性的数据类型进行初始化操作。最后调用Insert函数将temp 的数值以及该记录在块中的偏移量保存到index中。完成index的插入操作。

### 3.select函数（Index部分）

函数首先定义了inPos以及No变量，其中inPos被初始化为-2。接着函数将计算where语句 中待查询属性在当前表的位置。当计算完查询属性的位置。函数将进入该循环，循环中将判 断语句是否存在等号以及等号左边的属性是否使用了index。若两个条件都满足，则对Index Manager进行初始化，并以等号右边的数值作为参数调用Index Manager的Find函数。Find 函数将返回-1或大于0的数值，其中-1表示index不存在该数值，大于0表示index存在该 数值。假如返回值大于零，则该返回值为存放了此数值的记录在块中的偏移量。

继续以刚才这个输入为例子，假设程序现正处理select \* from student where sno = '22222222' 这个语句，那么Find函数将返回存放着数值'22222222'记录（即图表37中插入的第3个数据） 在块中的偏移量。接下来判断inPos是否不等于2。若等于2则表明该变量数值没有被修改 过，不进入if语句；不等于2代表调用过了Index Manager的Find函数，此时应进入if语句 中。假如inPos等于-1，则说明符合条件的记录不存在，此时调用图表39下方的display函 数并返回。若inPos不等于 -1，则进入第二个if语句中的双重循环。第一个循环遍历当前 表相关文件占据的块，使用pdata变量读取当前块的偏移地址。第二个循环取出记录的每个 属性的数值，全部取出后进入tuplesatisfied函数判断该记录是否满足where的条件。若不满 足则清除该记录所有属性的数值并清空temp变量（存放当前读出记录的属性数值），重新 读取下个块；假如满足条件则该记录放入out\_table变量中，接着调用display显示结果，最 后返回。

### 4.Delete函数（Index部分）

一开始使用index进行查找的操作与select部分完全相同，但取出记录后的操作与其略有不 同。若取出的记录满足where条件，则将pdata指向的首个元素赋值为”Deleted”，该标记能 被Buffer Manager识别。接着遍历当前表的index，调用Index Manager的Delete函数将index 中对应属性的数值删除。最后将读出记录所有属性的数值删除并返回。而当记录不满足where 条件的时候，则直接删除所有属性的数值并清空temp变量。

### 5.CreateIndex函数

size变量用于计算当前成为index属性的内存空间偏移量，计算完成后保存到index\_attri变量 中。接着进入双重循环。第一个循环遍历当前表相关文件占据的块，No变量记录当前使用的 块号。第二个循环取出当前表记录中index属性的数值，利用刚才算出的index\_attri得到该 属性在块中对应的内存地址，这样就不必读出整个记录，直接读出index属性的数值并插入 到index中。

**3.Minisql功能测试**

## 3.1表的建立

**3.1.1测试目标**

创建表create语句的执行，若创建失败告诉用户失败原因

数据类型支持int，char(n)，float，其中char(n)满足1<=n<=255

一个表属性可以指定是否为unique

单属性的主键定义

create table student (

sno char(8),

sname char(16) unique,

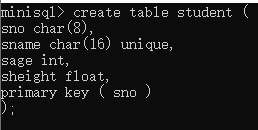
sage int,

sheight float,

primary key ( sno )

);

**3.1.2测试结果**



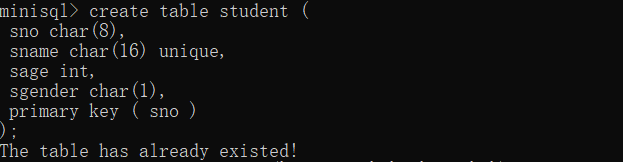


**3.1.3测试结果分析**

经过测试，create语句执行，相关文件生成，对于primary key的属性自动创建了索引，表创建语句成功执行

**3.1.4创建表若失败，告诉用户失败的原因**

创建表时该表已存在，弹出the table has already existed

****

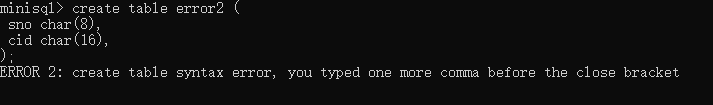
创建表时多了一个逗号，弹出error 2

create table error2 (

sno char(8),

cid char(16),

);



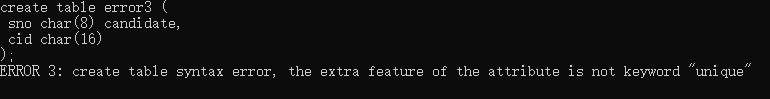
创建语句为非unique约束，弹出error 3

create table error3 (

sno char(8) candidate,

cid char(16)

);



## 3.2数据的插入、查找

**3.2.1测试目标**

每次一条记录的插入操作，若插入失败则告诉用户失败原因

unique和primary key约束的有效性

\*查询和选定标签查询，并通过and连接多个条件进行查询，进行等值查询与区间查询

//插入数据

insert into student values ('12345678','wy',22,1.87);

insert into student values ('11111111','leo',32,1.69);

insert into student values ('22222222','caomincheng',12,1.72);

insert into student values ('33333333','mailan',53,1.88);

insert into student values ('44444444','cuixiangyuan',47,1.81);

select \* from student;

//测试 \* 功能和没有where条件的情况

insert into student values ('55555555','leo',54,1.90);

//测试unique约束是否有效

select \* from student;

insert into student values ('12345678','wy',22,1.87);

//测试primary key是否有效

select \* from student;

select sno, sname from student where sheight = 1.72;

select sno, sname from student where sheight <= 1.72;

select sno, sname from student where sheight < 1.72;

select sno, sname from student where sheight >= 1.72;

select sno, sname from student where sheight > 1.72;

select sno, sname from student where sheight <> 1.72;

//测试选定标签的select和存在where条件的情况，

//并测试算术比较符：= <> < > <= >=

select sno, sname from student where sage > 30 and sage <= 47;

//测试选定标签的select，存在多条件where语句的范围查询

select \* from student where sage < 50;

//测试 \* 功能，存在where条件

select \* from student where sage > 32 and sage <> 47;

//测试 \* 功能，存在多条件的where语句查询

select \* from student where sage = 0;

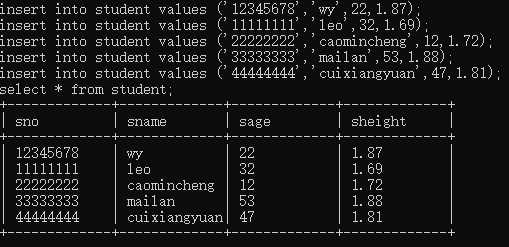
//查询结果为空，输出信息告诉用户结果为空

save;

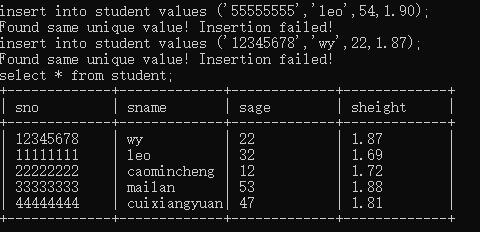
//将数据保存到文件中

**3.2.2测试结果**

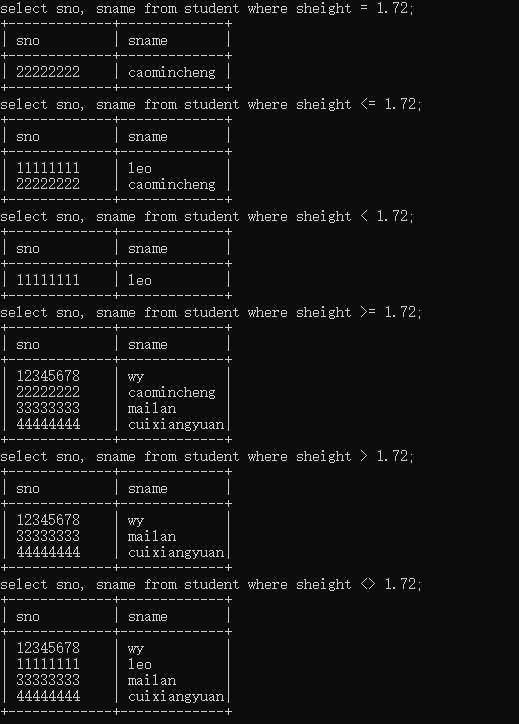
插入数据后，使用\*查询



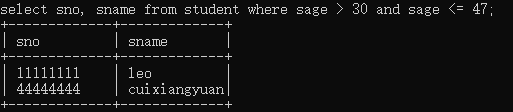
插入违反unique和primary key约束的重复数据后，使用\*查询



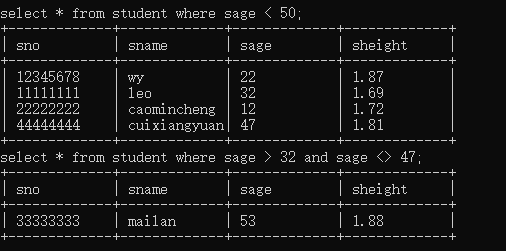
进行等值与区间的选定标签查询，包含算术运算符> = < >= <= <>



and连接多条件where语句的选定标签查询



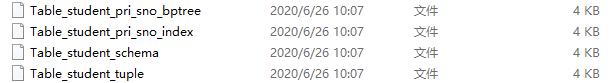
where条件语句\*查询



查询结果为空，则输出信息告诉用户查询结果为空



save命令将插入的记录保存到对应的文件中



**3.2.3测试结果分析**

经过测试，程序支持记录插入的insert语句，数据成功插入，并可以通过save命令将数据保存到文件中

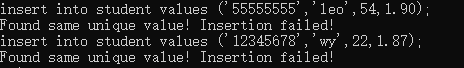
unique和primary key约束有效，成功阻止对应属性上重复数据的插入，并弹出提示

数据查询支持\*查询和选定标签查询，可以通过and连接多个条件进行查询，支持等值查询和区间查询，算数运算符< >= >= <= <>可以成功执行

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，若查询时若查询结果为空，则输出Empty set信息告诉用户查询结果为空

**3.2.4插入或查询失败告诉用户失败原因**

如果插入违反了unique和primary key约束，则输出失败原因，Found same unique value! Insertion failed!



查询时关键词from where and错误，分别输出error 8、9、10

select \* frrm student where sage = 12;

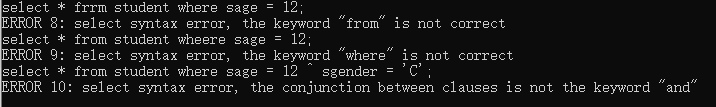
//from关键词不正确

select \* from student wheere sage = 12;

//where关键词不正确

select \* from student where sage = 12 ^ sgender = 'C';

//and关键词不正确



查询时表名不存在，输出error 14

select \* from error;



**3.3 数据的删除**

**3.3.1测试目标**

数据删除delete语句的执行，并进行一条或多条记录的删除

若删除失败，输出失败原因

delete from student where sage = 22;

//delete语句测试

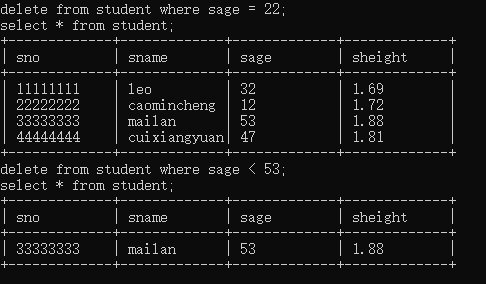
select \* from student; // 再次调用select 语句测试delete的情况

delete from student where sage < 53;

//一次多条记录的删除测试

select \* from student; // 再次调用select 语句测试delete的情况

**3.3.2测试结果**



**3.3.3测试结果分析**

通过测试，该程序成功实现了delete语句的执行，并可以通过where语句进行单条记录的删除或多条记录的删除操作

**3.3.4 若删除失败，告诉用户失败的原因**

关键词where错误，输出error 4

delete from student wher sage = 12;

//关键词“where”不正确



关键词and错误，输出error 5

delete from student where sage = 12 andi sgener = 'C';

//关键词“and”不正确



**3.4 索引的创建和删除**

**3.4.1测试目标**

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，若失败则告诉用户失败原因，并且测试索引的有效性

create index snameindex on student ( sname );

//创建拥有unique约束的属性sname的索引

drop index snameindex;

//删除索引

**3.4.2测试结果**

对于表的主属性，程序在创建表时自动建立索引，并将信息保存在了文件中



用户对于声明为unique的属性通过create index语句创建索引



通过drop index语句，相关index文件被删除

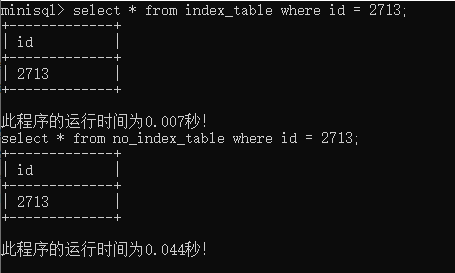
**3.4.3测试结果分析**

经过测试，该系统可以根据表的主属性自动建立B+树索引，并且用户可以通过create index和drop index语句创建和删除索引

**3.4.4索引有效性测试**

由于系统对于primary key的属性自动建立索引，因此建立两个测试表，一个表拥有primary key属性，存在索引，另一个表没有primary key属性，不存在索引

通过insert语句分别向两个表中插入4000条相同的数据，再进行where语句查询，观察程序执行时间。创建表和插入数据的语句保存在index\_check.txt文件中



可以看到，不带有索引的表查询时间大约为带有索引表查询时间的六倍

经测试，索引的有效性得到实现

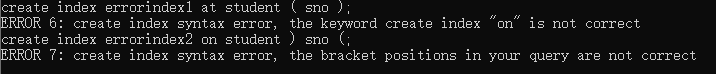
**3.4.5若操作失败，告诉用户失败原因**

create index errorindex1 at student ( sno );

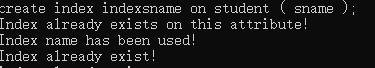
//on关键词错误

create index errorindex2 on student ) sno (;

//括号位置错误



索引名称已存在和被使用



表名不存在



属性名不存在



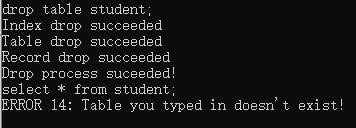
**3.5 删除表**

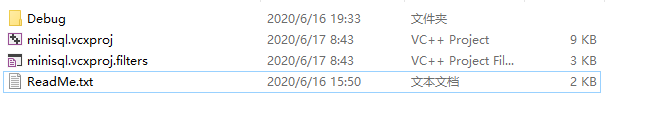
通过drop语句删除表

drop table student;

select \* from student;

//输出 The table does not exist

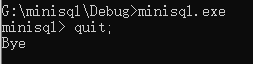




表相关文件被删除，再次进行该表查询则反馈表名不存在，表删除语句成功

**3.6 退出MiniSQL系统语句**

quit；



在执行quit命令时，自动执行save命令，即将数据自动保存到文件中，之后退出minisql

**3.7执行SQL脚本文件**

execfile test.txt

文件test.txt包含

创建表的create语句，unique约束创建，单属性的primary key定义

插入数据的insert语句，包括插入重复数据检查unique和primary key是否有效

select查询语句，select\*和select 选定属性，where条件语句（多条件用关键词and连接），条件语句中<>,<,>,<=,>=,=运算符的执行

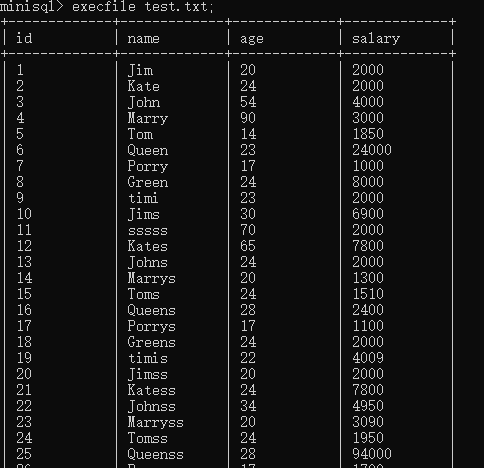
delete删除数据语句，delete from 表名 where条件

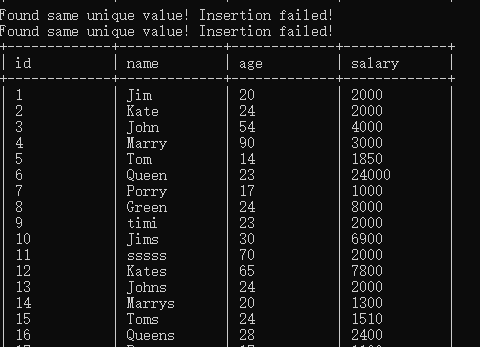
创建索引删除索引

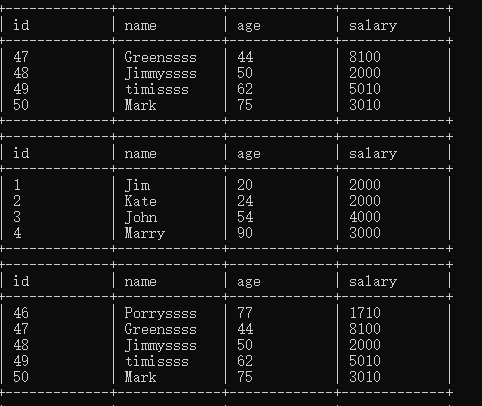
drop table

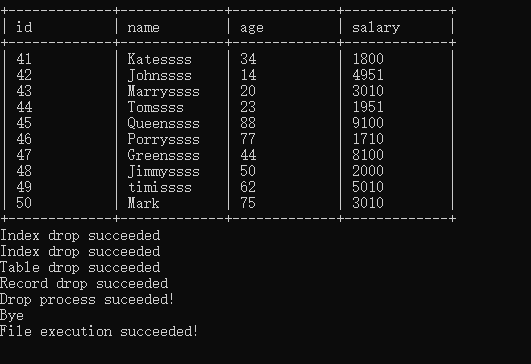
quit

程序执行部分截图









**4.分工说明**

郑无恙：Interpreter,API,Catalog Manager,Record Manager (Index 相关部分）

麦蓝：Interpreter,Index Manager,部分datatype，

曹闵丞：buffer manager，验收视频录制

崔翔源：DB files，功能测试，总体报告整合

曾致语：Record Manager，总体报告整合，功能测试

备注：测试全员参与，报告每个人写各自负责的模块