

# **MINISQL**

## **设计总报告**

组员： 邓墨琳 3150103457  
王立东 3160102170  
彭昊 3160104383  
陈逸雪 3160102516  
课程：数据库系统  
指导老师：黄忠东  
时间：2018-06

# 目录

## 第 1 章           MINISQL 总体框架

---

- 第 1.1 节       MiniSQL 实现功能分析
- 第 1.2 节       MiniSQL 系统体系结构
- 第 1.3 节       设计语言与运行环境
- 第 1.3 节       文件组织架构

## 第 2 章           MINISQL 各模块设计

---

- 第 2.1 节       Interpreter 设计
- 第 2.2 节       API 设计
- 第 2.3 节       Catalog Manager 设计
- 第 2.4 节       Record Manager 设计
- 第 2.5 节       Index Manager 设计
- 第 2.6 节       Buffer Manager 设计

## 第 3 章           数据结构及各模块接口

---

- 第 3.1 节       数据结构
- 第 3.2 节       主窗口及主函数设计
- 第 3.3 节       Catalog Manager 接口
- 第 3.4 节       Record Manager 接口
- 第 3.5 节       Index Manager 接口
- 第 3.6 节       Buffer Manager 接口

## 第 4 章           MINISQL 系统测试

---

## 第 5 章           分工说明

---

## 第 6 章           附录

---

# 第 1 章 MINISQL 总体框架

## 第 1.1 节 MiniSQL 实现功能分析

我们设计并实现了一个精简型单用户 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。

### 1. 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  $1 \leq n \leq 255$ 。

### 2. 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。

### 3. 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）。

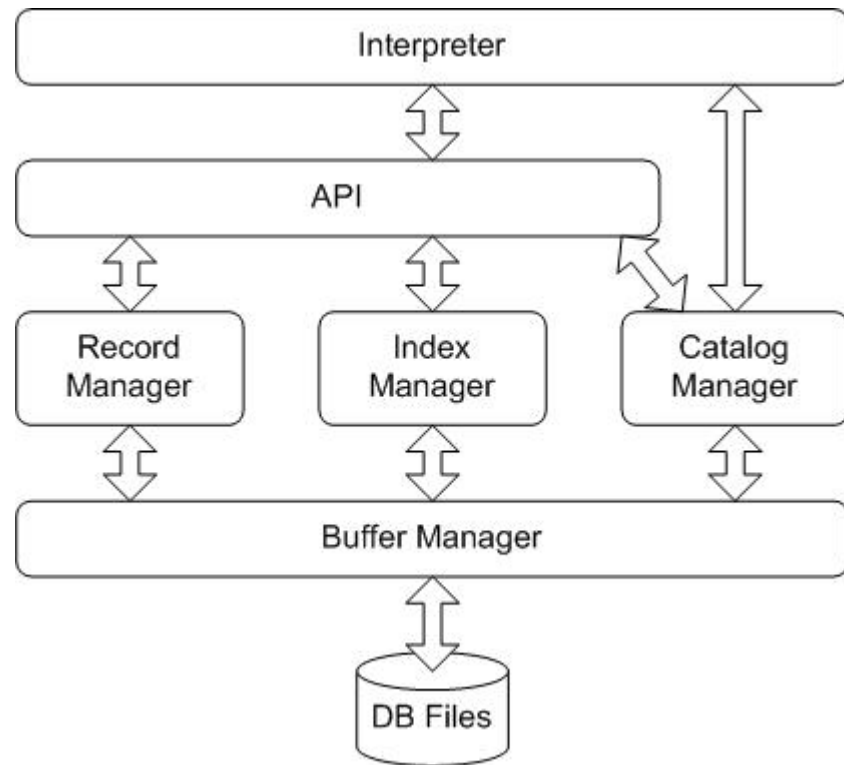
### 4. 查找记录

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。

### 5. 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。

## 第 1.2 节 MiniSQL 系统体系结构



MiniSQL 体系结构

## 第 1.3 节 设计语言与运行环境

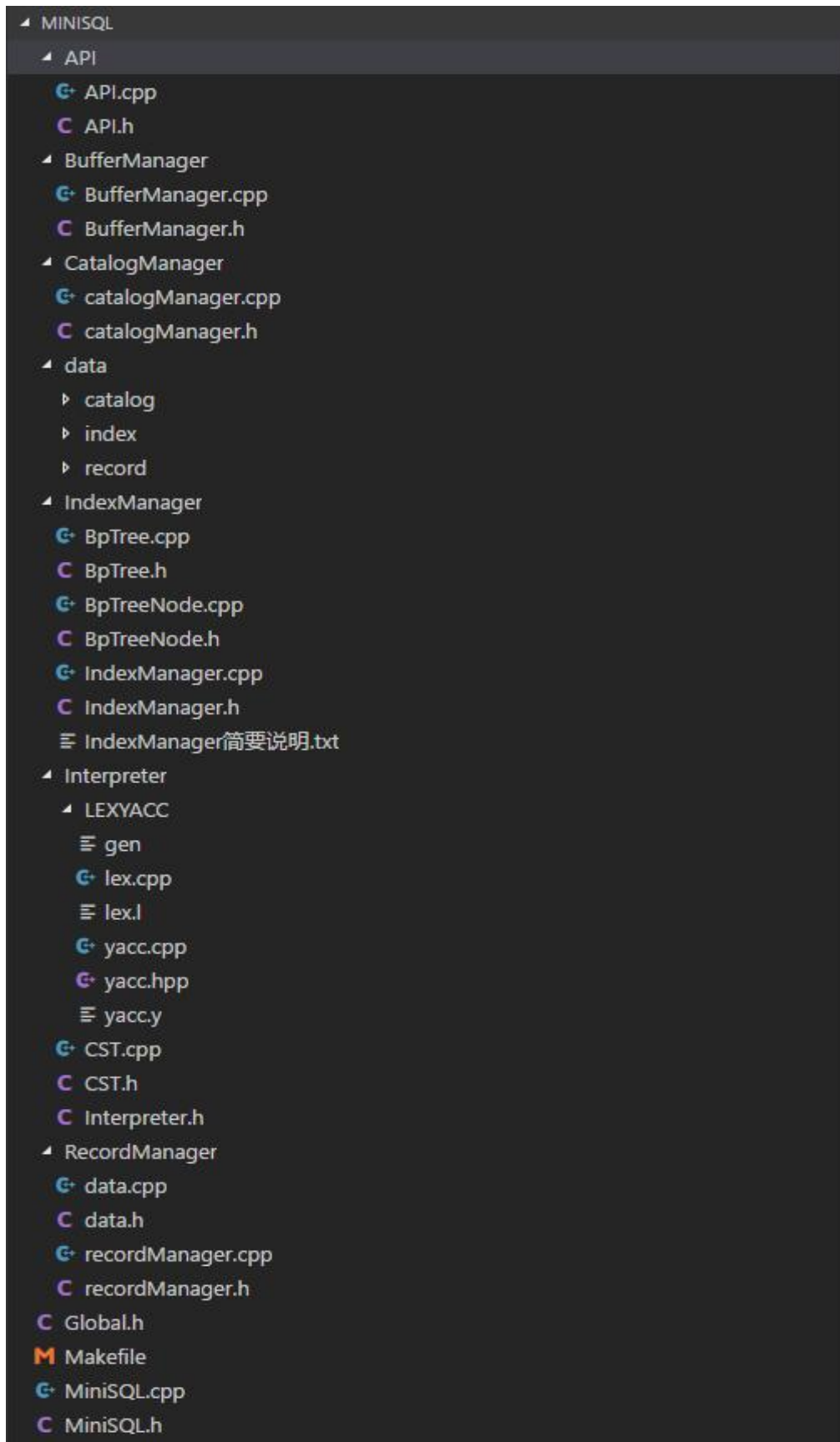
工具: C++

环境: g++ 4.9.2 or higher

bison 3.0.4

flex 2.6.4

## 第 1.4 节 文件组织架构



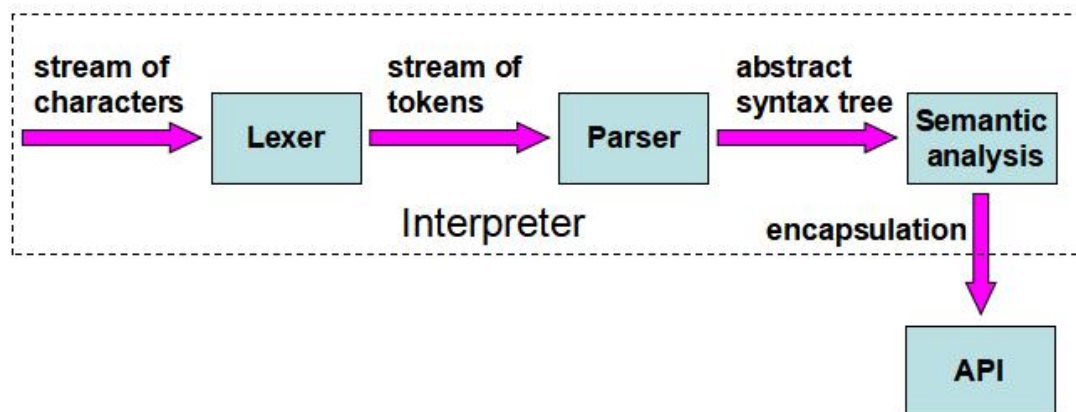
## 第 2 章 MINISQL 各模块设计

### 第 2.1 节 Interpreter 设计

#### 2.1.1 概述

本组 miniSQL 解释器通过 LEX、YACC 实现，LEX 进行词法分析，将字符流转化为 token 流。YACC 进行语法分析，构建抽象语法树，并通过语义分析得到语义值封装成 SQL schema 数据结构，传给 API 相关接口，来实现语义动作。通过在 Interpreter 类中调用 YYPARSE 函数可完成这一功能。

简易流程图如下所示



本节内容与数据库设计关联不大，属于编译原理课程内容，故不详细说明。

#### 2.1.2 编译环境

Ubuntu 16.04 LTS

g++ 4.9.2

bison 3.0.4

flex 2.6.4

### 2.1.3 词法分析

#### (1) 关键字

CREATE、SELECT、WHERE、DROP、FROM、PRIMARY 等。

#### (2) 正则表达式

以 float 为例，正则表达式为

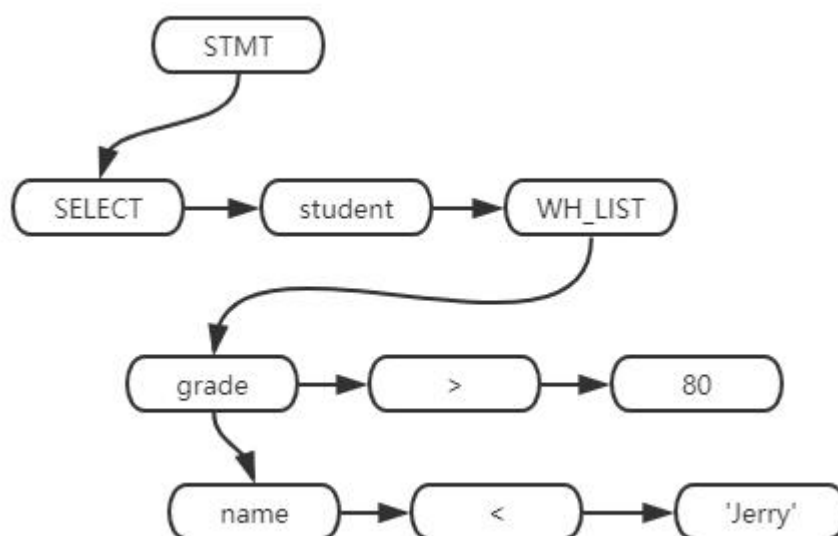
```
-?[0-9]+"."[0-9]*  
-?"[0-9]+  
-?[0-9]+E[-+]?[0-9]+  
-?[0-9]+"."[0-9]*E[-+]?[0-9]+  
-?"[0-9]+E[-+]?[0-9]+
```

### 2.1.4 语法分析

miniSQL 的文法生成规则见附录。

以 `select * from student where grade > 80 and name < 'Jerry';`

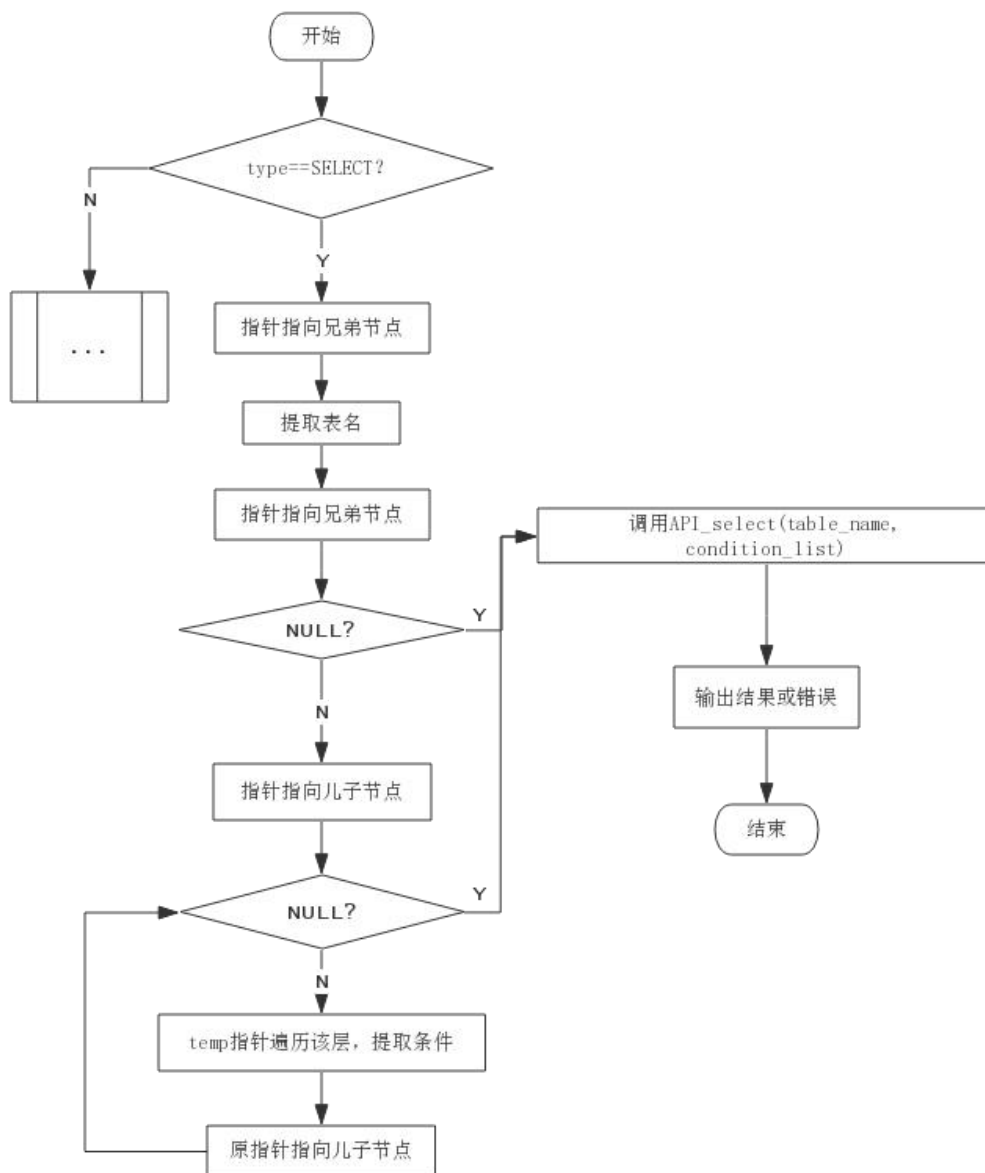
为例展示抽象语法树



### 2.1.5 语义提取

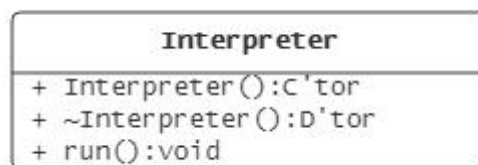
同样以 `select * from student where grade > 80 and name < 'Jerry';`

为例，以下流程图展示了 CST.cpp 中的 `_Select` 函数如何遍历抽象语法树，从中提取语义信息，封装成 `schema` 结构，最后调用 API 执行相关语义动作。



其他操作类似。

### 2.1.6 Interpreter 类



`run():void` 调用 `yyparse()` 函数，完成解释器功能。



### 2.1.7 错误处理

由 yyerror()函数处理

```
void yyerror(const char *s, ...) {
    va_list ap;
    va_start(ap, s);
    fprintf(stderr, "line %d: error near '%s': ", yylineno, yytext);
    vfprintf(stderr, s, ap);
    fprintf(stderr, "\n");
    va_end(ap);
}
```

当 yywrap()函数检测到语法错误时, 调用 yyerror 函数输出错误信息, 同时停止解析, 释放语法树空间。

```
stmt_list: error ';'          {yyerror("grammar error."); nm_clear();}
| stmt_list error ';'        {yyerror("grammar error."); nm_clear();}
```

## 第 2.2 节 API 设计

### 2.2.1 概述

根据 Interpreter 提供的数据库信息和 CatlongManager、RecordManager 和 IndexManager 提供的接口函数进行拼接, 实现数据库的功能。

### 2.2.2 API 函数

- + API\_Create\_Table(Table& table):bool 创建表格
- + API\_Drop\_Table(string table\_name):bool 删除表格
- + API\_Create\_Index(Index& index):bool 创建索引
- + API\_Drop\_Index(string index\_name):bool 删除索引
- + API\_Insert(Record& record):bool 插入键值
- + API\_Select(string table\_name, Condition\_list clist, bool if\_where = false):int 查找数据, 返回影响的行数
- + API\_Delete(string table\_name, Condition\_list clist, bool if\_where = false):int 删除记录, 返回影响的行数

## 第 2.3 节 Catalog Manager 设计

2.3.1 概述

Catalog Manager 负责管理数据库的所有模式信息，并提供访问以及操作相应信息的函数接口。

管理的模式信息有：

- 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。

提供接口的功能有：

- 1.判断字段类型是否唯一、是否是主键。
- 2.添加或删除表的信息。
- 3.通过表名获取表的信息。
- 4.添加或者删除索引。
- 5.通过索引名或者通过表名以及字段名获取索引信息。
- 6.通过表名获取所有表上的索引名。
- 7.通过表名获取表上字段数。

另外，本数据库中 Catalog Manager 通过析构时将所有模式信息存入 buffer 区，构造时将所有模式信息取出 buffer 区，实现信息文件的磁盘存取与读写。

2.3.2 CatalogManager 类

CatalogManager
<div>- tableMap: unordered_map&lt;string, Table*&gt;</div> <div>- indexMap: unordered_map&lt;string, Index*&gt;</div>
<div>+ CatalogManager(): C'tor</div> <div>+ ~CatalogManager(): D'tor</div> <div>+ tableExists(const string tableName): bool</div> <div>+ indexExists(const string indexName): bool</div> <div>+ isUnique(const string attrName, const string tableName): bool</div> <div>+ isPK(const Attribute attrName): bool</div>

```

+ addtable(const string tableName, const Attribute*
    attrName ,const int attrCount): bool
+ deletetable(const string tableName): bool
+ gettable(const string tableName): Table*
+ addIndex(const string indexName, const string tableName, const
    string attrName): bool
+ deleteIndex(const string indexName): bool
+ getIndex(const string indexName): Index*
+ getattr_count(const string tableName): int
+ GetIndexName(const string tableName): const IndexName*
+ getIndexByTableCol(const string tableName, const string
    attrName): Index*

```

```

- tableMap: unordered_map<string, Table*> 存储 table 信息的 map
- indexMap: unordered_map<string, Index*> 存储 index 信息的 map
+ CatalogManager(): C'tor 构造函数，将磁盘文件存储的信息读入内存
+ ~CatalogManager(): D'tor 析构函数，将内存中的信息存入 buffer 区
+ tableExists(const string tableName): bool 判断表是否存在
+ indexExists(const string indexName): bool 判断索引是否存在
+ isUnique(const string attrName, const string tableName): bool 判
断字段是否唯一
+ isPK(const Attribute attrName): bool 判断字段是否为主键
+ addtable(const string tableName, const Attribute*
    attrName ,const int attrCount): bool 添加新表
+ deletetable(const string tableName): bool 删除已有表
+ gettable(const string tableName): Table* 获取表信息
+ addIndex(const string indexName, const string tableName, const
    string attrName): bool 添加新索引
+ deleteIndex(const string indexName): bool 删除已有索引
+ getIndex(const string indexName): Index* 获取索引信息（通过索引

```

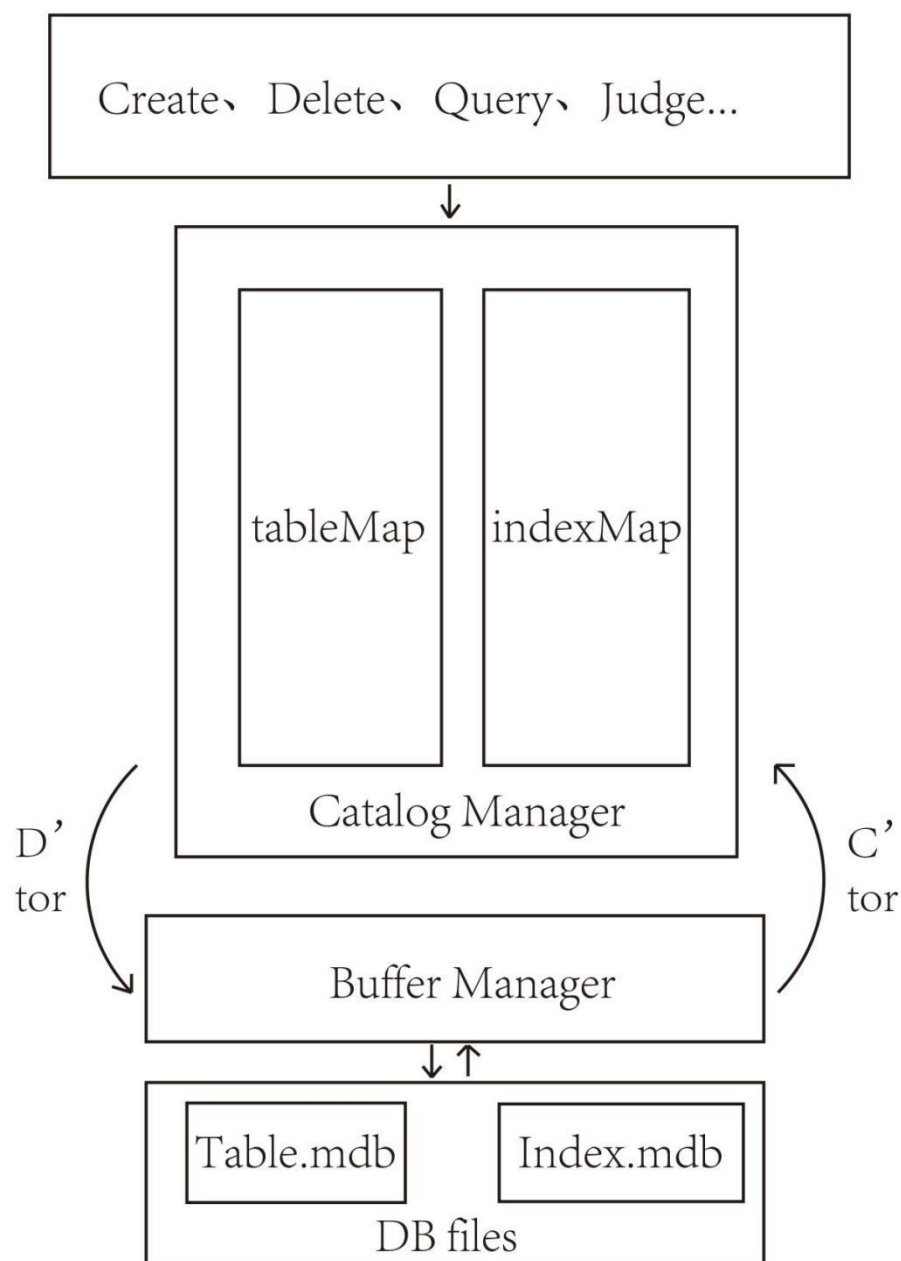
名)

+ `getattr_count(const string tableName): int` 获取表上字段数

+ `GetIndexName(const string tableName): const IndexName*` 获取表上所有索引名

+ `getIndexByTableCol(const string tableName, const string attrName): Index*` 获取索引信息 (通过表名和字段名)

### 2.3.3 CatalogManager 设计思路



主内存中通过 tableMap 与 indexMap 分别储存表与索引的信息，并提供接口给上层程序。Catalog Manager 析构时将内存中的两张 Map 存入 buffer 区，由 BufferManager 将信息存入磁盘上的 Table.mdb 以及 Index.mdb 两个文件中。而 Catalog Manager 构造时通过 BufferManager 将磁盘文件中储存的信息重新读入内存的两张 Map 中。

## 第 2.4 节 Record Manager 设计

### 2.4.1 概述

Record Manager 是与 Catalog Manager 和 Index Manager 并列的管理数据文件的模块。它为 API 提供与数据记录操作相关的函数接口，并调用 Buffer Manager 的函数实现对内存和磁盘文件的读写。Record Manager 负责管理记录表中数据的数据文件。

根据实验要求，数据文件暂时只支持定长记录的存储。Record Manager 可支持的数据类型有 int, float 和 char。数据文件由一个或多个数据块组成，块大小与缓冲区块大小相同。

实现功能有：

#### 1. 数据文件的创建（由表的定义与删除引起）

如果语句执行成功，则返回 true；否则，返回 false。

#### 2. 数据文件的删除

如果语句执行成功，则返回 true；否则，返回 false。

#### 3. 记录的插入

能够根据所给的记录去插入到相应的表中，并返回插入记录的位置。

#### 4. 记录的删除

该操作能够支持不带条件的删除和带一个条件的删除（包括等值查找、不等值查找和区间查找）。

#### 5. 记录的查询

该操作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）。

#### 6. 提供相应接口

提供相应的接口供 API 和其它函数调用。

### 2.4.2 record Manager 类

record Manager
<b>+ createTable(Table* table): bool</b> <b>+ dropTable(Table* table): bool</b> <b>+ select(Table* table, Condition_list clist): Data*</b> <b>+ select(Table* table): Data*</b> <b>+ insert(Record&amp; record): int</b> <b>+ deleteValue(Table* table, Condition_list clist): bool</b> <b>+ deleteValue(Table* table): bool</b>

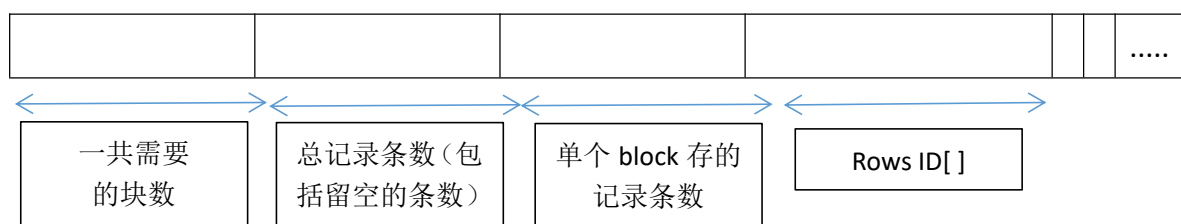
**+ createTable(Table\* table): bool** 创建表  
**+ dropTable(Table\* table): bool** 删除表  
**+ select(Table\* table, Condition\_list clist): Data\*** 按照条件查找记录  
**+ select(Table\* table): Data\*** 返回整张表  
**+ insert(Record& record): int** 插入记录  
**+ deleteValue(Table\* table, Condition\_list clist): bool** 按照条件删除表记录  
**+ deleteValue(Table\* table): bool** 删除整张表中的记录

### 2.4.3 record Manager 设计思路

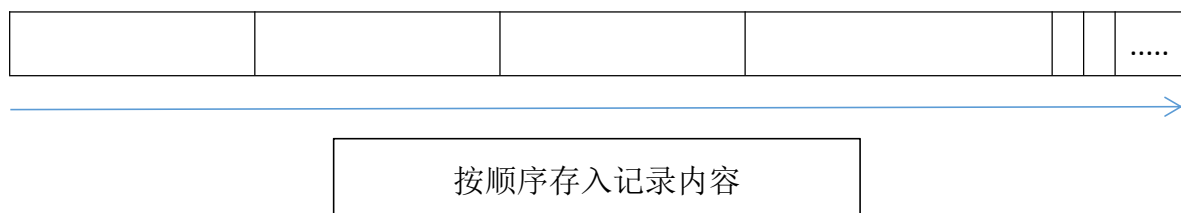
Record Manager 模块包含了两个 cpp 文件,recordManager.cpp 主要是对操作行为的管理,而 data.cpp 是对对象进行的实际操作。

首先是对 data 数据结构的设计,它包含了表名 (tableName),以记录元组 (tuples) 为元素的容器,还有记录了每条记录位置的数组 rowId (rowId [0]表示所储存的总记录条数,rowId [i]表示第 i 个位置所储存的记录序号,若该位置要留空,则值为 -1)。

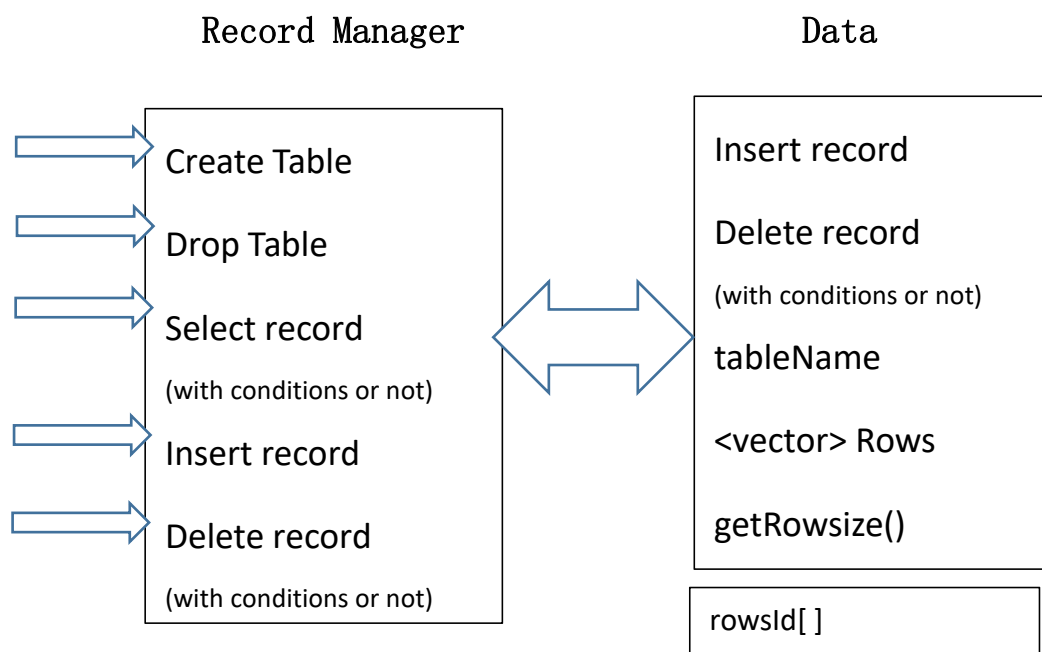
块头信息 Block[0]:



子块:



然后对于 recordManager 来说，只是对 data 数据结构进行相应函数的调用。



## 第 2.5 节 Index Manager 设计

### 2.5.1 概述

Index Manager 负责 B+树索引的实现，具有的功能有：

1. 根据索引名、键长创建 B+树索引
2. 根据索引名删除 B+树索引
3. 根据索引名和键值查找特定记录的地址信息
4. 根据索引名、键值和地址信息插入键值
5. 根据索引名、键值移除键值

### 2.5.2 IndexManager 类

IndexManager
<b>+ CreateIndex(const char* filename, int KeyLength):bool</b> <b>+ DropIndex(const char* filename):bool</b> <b>+ Find(const char* filename, const char* key):int</b> <b>+ Insert(const char* filename, const char* key, const int value):bool</b> <b>+ Remove(const char* filename, const char* key):bool</b>

1. + CreateIndex(const char\* filename, int KeyLength):bool 创建索引
2. + DropIndex(const char\* filename):bool 删除索引
3. + Find(const char\* filename, const char\* key):int 等值查找
4. + Insert(const char\* filename, const char\* key, const int value):bool 插入键值
5. + Remove(const char\* filename, const char\* key):bool 删除键值

BpTree
<b>- order:int</b> <b>- root:int</b> <b>- KeyLength:int</b> <b>- NodeCount:int</b>



- FirstEmpty:int
- _filename:string
+ BpTree(const char* filename):C'tor
+ ~BpTree():D'tor
+ Insert(const char* key, const int value, const int type = -1):bool
+ Remove(const char* key, const int type = -1):bool
+ Find(const char* key):int
+ GetFirstEmpty():int
+ GetKeyLength():int

1. - NodeCount:int 节点个数
2. - FirstEmpty:int 第一个空块号
3. - \_filename:string 索引名
4. + BpTree(const char\* filename):C'tor 构造函数, 读取 block0 数据
5. + ~BpTree():D'tor 析构函数, 写入 block0 函数
6. + Insert(const char\* key, const int value, const int type = -1):bool 插入键值
7. + Remove(const char\* key, const int type = -1):bool 移除键值
8. + Find(const char\* key):int 等值查找
9. + GetFirstEmpty():int 获得第一个空块号
10. + GetKeyLength():int 获得键长

BpTreeNode
- ptr:vector<int>
- keys:vector<char*>
- size:int
- id:int
- isleaf:bool
- length:int

```

- _filename:string
+ BpTreeNode(const char* filename, int value, int KeyLength):C'tor
+ ~BpTreeNode():D'tor
+ Insert(const char* key, const int order, const int value, const
int type = -1):int
+ Remove(const char* key, const int order, int num = -1, const int
type = -1):int
+ Find(const char* key):int
+ Getisleaf():bool
+ Split(const int value):BpTreeNode*
+ GetId():int
+ GetKey(int num):const char*
+ GetPtr(int num):const int
+ Add(BpTreeNode* a, const int order):int
+ GetSize():int
+ Drop(const int FirstEmpty):bool

```

1. - ptr:vector<int>存储整数地址信息
2. - keys:vector<char\*>存储键值信息
3. - size:int 键值个数
4. - id:int 节点对应的块号
5. - isleaf:bool 判断是否为叶节点
6. - length:int 键长
7. - \_filename:string 索引名
8. + BpTreeNode(const char\* filename, int value, int KeyLength):C'tor 构造函数
9. + ~BpTreeNode():D'tor 析构函数
10. + Insert(const char\* key, const int order, const int value, const int type = -1):int 插入键值
11. + Remove(const char\* key, const int order, int num = -1,

const int type = -1):int 移除键值

12. + Find(const char\* key):int 等值查询

13. + Getisleaf():bool 获取叶节点的状态判断变量

14. + Split(const int value):BpTreeNode\* 节点分裂

15. + GetId():int 获取对应块号

16. + GetKey(int num):const char\* 根据序号获取键值

17. + GetPtr(int num):const int 根据序号获取整数地址

18. + Add(BpTreeNode\* a, const int order):int 节点合并

19. + GetSize():int 获得键值个数

20. + Drop(const int FirstEmpty):bool 删除节点信息，留下指针信息

### 2.5.3 IndexManager 设计思路

IndexManager模块总共分为三个结构来实现，分别为IndexManager、BpTree、BpTreeNode。IndexManager结构作为IndexManager模块的管理器，提供对外的接口函数，这些接口函数通过对BpTreeNode和BpTree里的成员变量进行操作实现对B+树的操作。

**数据结构：**一个节点一个块，一个B+树索引一个文件。

**Block 0:**

阶数	键长	根节点块号	节点个数	第一个空块号	...
----	----	-------	------	--------	-----

**Block 1 ~ N:**

键数	地址	键值	指针	...
----	----	----	----	-----

### 函数思路:

当创建B+树索引的接口函数被调用时，IndexManager会创建一个以索引名命名的mdb文件，然后通过BufferManager模块读取该mdb文件的第一个块，将文件头信息写入到该块中。文件头信息包含B+树阶数（int，由键长、块的大小以及其他一些数据计算）、键长（int）、根节点的块号（int）、节点个数（int）、第一个空块号（int）。

当删除B+树索引的接口函数被调用时，IndexManager会删除对应索引名的mdb文件。

当等值查询的接口函数被调用时，IndexManager会以索引名构造BpTree结构。BpTree构造函数调用时，通过BufferManager模块读取对应mdb文件的第一个块（文件头信息所在的块），将该块里的信息加载到成员变量中。而后BpTree结构会调用本结构下的等值查询函数，在这里，我们命名它为等值查询k。等值查询k会以本结构下的索引名、键长、块号（第一次为根节点的块号，后面为等值查询k返回的块号，直到结束该查询）构造BpTreeNode结构，BpTreeNode构造函数会通过BufferManager模块读取对应块号的块，然后将块中的数据加载到成员变量中。而后BpTreeNode结构会调用本结构下的等值查询函数，在这里，我们命名它为等值查询m。等值查询m会遍历本结构下的数据，返回结果。

当插入键值的接口函数被调用时，与上述等值查询相似。IndexManager会构造BpTree读取文件头，BpTree根据需求获得对应块的BpTreeNode结构，直到将键值插入，若出现需要分裂节点的情况，BpTree会进行一系列操作完成。

删除键值的思路与插入相同。

## 第 2.6 节 Buffer Manager 设计

### 2.6.1 概述

Buffer Manager 负责内存中 block 管理，实现功能有

1. 从 disk 中读取数据到内存中；将内存中的数据写入 disk

2. 实现 LRU 调度算法,当缓冲区满时选择 least recent used block swap out

3. 记录 block 是否为脏页

4. 提供 pin 功能

本项目中定义最大 block 大小为 4KB, 最大 block 数(缓冲区)为 100.

### 2.6.2 BufferManager 类

BufferManager
- block_cnt:int - lru_head:Block_node* - lru_tail:Block_node* - node_map:unordered_map<string, Block_node*>
+ BufferManager():C'tor + ~BufferManager():D'tor + get_block(const char* filename, int id):Block* + remove_block(const char* filename):void - delete_from_mem(Block_node* node, bool write = true):void - load_from_file(const char* filename, int id):Block* - write_to_file(const char* filename, int id):void

-block\_cnt:int 当前内存中 block 数量

-lru\_head:Block\_node\* 双向链表伪头部

-lru\_tail:Block\_node\* 双向链表伪尾部

-node\_map:unordered\_map<string, Block\_node\*> 索引信息

+BufferManager():C'tor 构造函数

+~BufferManager():D'tor 析构函数

+get\_block(const char\* filename, int id):Block\* 获取内存中第 id 个 block

+remove\_block(const char\* filename):void 删除内存上所有以 filename 命名的 block

-delete\_from\_mem(Block\_node\* node, bool write = true):void 删除双向链表中该节点以及内存中的 block

-load\_from\_file(const char\* filename, int id):Block\* 从 disk 中读取第 id 个 block

`-write_to_file(const char* filename, int id):void` 将 block 写入 disk

### 2.6.3 LRU 调度算法实现

数据结构为双向链表，头尾均添加 dummy pointers 方便管理。swap out 规则类似于 FIFO。

新加入的 block 插入链表头；

新被访问的数据插入链表头；

swap out 从尾部直接淘汰



在链表头插入 node 的方法由结构体 Block\_node 内部实现。

```
void attach(Block_node* node_) {  
    pre = node_; next = node_->next;  
    node_->next->pre = this; node_->next = this;  
}
```

## 第 3 章 数据结构及各模块接口

### 第 3.1 节 数据结构

数据库 schema:

1. 字段属性:

```
struct Attribute {  
    string attr_name;  
    int attr_type;  
    int attr_key_type;  
    int attr_len;  
    int attr_id;  
};
```

attr\_type:int 字段数据类型, 分别为 CHAR,FLOAT,INT

attr\_key\_tpe:int 完整性约束信息, 分别为 PRIMARY,UNIQUE,NULL

attr\_len:int 字段长度, CHAR 为 1, 其余为 4

attr\_id:int 字段 id, 记录位于表中位置

2. Table 描述 relation schema:

```
struct Table {  
    string table_name;  
    int attr_count;  
    Attribute attrs[32];  
    ...//some functions  
}
```

table\_name:string 表名

attr\_count:int 表中字段个数

attrs:Attribute[32] 表中字段列表, 最多 32 个属性

3. 查询条件信息:

```
struct Condition {  
    string attr_name;  
    int attr_type;  
    string op_type;  
    string cmp_value;  
};
```

attr\_name:string 字段名

attr\_type:int 字段对应的数据类型

op\_type:string 运算符类型, 分别为<>,<=<,>,>=

cmp\_value:string 操作数, 进行比较的值

#### 4. 查询条件信息表:

```
typedef list<Condition> Condition_list;
```

包含多条查询信息。

#### 5. 索引信息

```
struct Index {  
    string index_name;  
    string table_name;  
    string attr_name;  
};
```

index\_name:string 索引名

table\_name:string 表名

attr\_name:string 索引对应的字段名

#### 6. 索引名

```
struct IndexName {  
    string name[100];  
    int len;  
};
```

单独储存索引名

#### 7. 记录

```
struct Record {  
    string table_name;  
    vector<string> attr_values;  
    int num_values;  
};
```

table\_name:string 表名

attr\_values:vector<string> 一条记录的值

num\_values:int 插入记录的字段个数, 应与 table 中字段数相匹配

### Interpreter:

SQL 语句抽象语法树(实际为 concrete syntax tree)

```
struct CST {  
    int type;  
    char text[256];  
    CST *lpNext;  
    CST *lpSub;  
};
```



`type:int` 抽象语义动作，对应 SQL 查询语句类型，分别为  
 CREATE,DROP,INSERT,DELETE,SELECT,EXECFILE(创建删除包括表和 index)  
`text:char[256]` 语义值  
`lpNext:CST*` 指向兄弟节点  
`lpSub:CST*` 指向子节点

## B+树中间块结构设计：

键数↴	地址↴	键值↴	指针↴	...
-----	-----	-----	-----	-----

通过第一个整数大小的值判断是否为叶节点，若整数地址为-1 则为叶节点，若大于 0 则为非叶节点。

## Catalog 数据：

内存中，表与索引分别存在两个 `unordered_map` 中：

```

class CatalogManager
{
private:
    // table map
    unordered_map<string, Table*> tableMap;

    // Index name map
    unordered_map<string, Index*> indexMap;
  
```

`tableMap: unordered_map<string, Table*>` 存储 table 信息的 map

`indexMap: unordered_map<string, Index*>` 存储 index 信息的 map

## Buffer Block 数据结构：

数据块 Block：

```

struct Block {
    string file_name;
    int id;
    bool dirty;
    bool pin;
    char data[MAX_BLOCK_SIZE];
    // C'tor
    Block(const char* _file_name, int _id):
        file_name(_file_name), id(_id) {
  
```

```

        dirty = pin = false;
    }
};

```

`id:int` block 的 index

`dirty:bool` 标记是否为脏页

`pin:bool` 标记是否该 block 被锁定

`data:char[]` block 中的数据，以二进制读入和存放

`Block():C'tor` 构造函数

数据块双向链表:

```

struct Block_node {
    Block* block;
    Block_node* pre;
    Block_node* next;
    // C'tor
    Block_node(Block* _block): block(_block) {}
    // D'tor
    ~Block_node() { remove_node(); }
    // Add to position after node_
    void attach(Block_node* node_) {
        pre = node_; next = node_>next;
        node_>next->pre = this; node_>next = this;
    }
    // Remove from linked list
    void remove_node() { pre->next = next; next->pre = pre;}
};

```

`block:Block*` 指向 block 的指针

`pre:Block_node*` 指向链表前一个结点

`next:Block_node*` 指向链表下一个结点

`Block_node():C'tor` 构造函数

`~Block_node():D'tor` 析构函数

`remove_node():void` 删除当前结点

## 第 3.2 节 主窗口及主函数设计

运行系统时界面如下：

```
=====Welcome to minisql!=====
minisql> _
```

## 第 3.3 节 Catalog Manager 接口

**isUnique(const string attrName, const string tableName): bool**

判断字段是否唯一，若该字段能在表中的类型为 UNIQUE 或者 PRIMAY，则是唯一属性，返回 true，否则返回 false。

**isPK(const Attribute attrName): bool**

判断字段是否为主键，若该字段在表中类型为 PRIMARY，则是主键，返回 true，否则返回 false。

**addtable(const string tableName, const Attribute\* attrName ,const int attrCount): bool**

添加新表，先检查新表名是否已经在 tableMap 中存在、新表名是否为空、传入字段是否有主键、传入字段是否重名等错误，相应地返回错误信息。若无错误，则新建一个 table 结构将信息一一储存，再将 table 放入 tableMap，返回 true。

**deletetable(const string tableName): bool**

删除已有表，先检查应删除表是否存在于 tableMap，若不存在则返回错误信息。若无错误，则将相应 table 从 tableMap 中删去，返回 true。

**gettable(const string tableName): Table\***

获取表信息，先检查该查找表是否存在于 tableMap，若不存在则返回错误信息。若无错误，则将相应 table 从 tableMap 中找出，返回该表结构。

**addIndex(const string indexName, const string tableName, const string attrName): bool**

添加新索引，先检查对应表名是否不存在于 tableMap、对应表名是否为空、新索引名是否为空、新索引名是否已经在 indexMap 中存在、对应字段是否属性唯一、对应表上的对应字段是否已经存在索引等错误，相应地返回错误信息。若无错误，则新建一个 index 结构将信息一一存入，再将 index 放入 indexMap，返回 true。

**deleteIndex(const string indexName): bool**

删除已有索引，先检查应删除索引是否存在于 indexMap，若不存在则返回错误信息。若无错误，则将相应 index 从 indexMap 中删去，返回 true。

**getIndex(const string indexName): Index\***

获取索引信息（通过索引名），先检查该查找索引是否存在于 indexMap，若不存在则返回错误信息。若无错误，则将相应 index 从 indexMap 中找出，返回该索引结构。

**getattr\_count(const string tableName): int**

获取表上字段数，先检查该查找表是否存在于 tableMap，若不存在则返回错误信息。若无错误，则将相应 table 从 tableMap 中找出，得到 attr\_count 值，返回该值。

**GetIndexName(const string tableName): const IndexName\***

获取表上所有索引名，遍历 indexMap 寻找出所有符合表名的 index，将索引名记录进数组结构中，最后返回数组指针。

**getIndexByTableCol(const string tableName, const string attrName): Index\***

获取索引信息（通过表名和字段名），先检查该查找表是否存在于 tableMap，若不存在则返回错误信息。再遍历表上字段，寻找索引，若能找到则返回该索引，若无法找到，则返回错误信息。

### 第 3.4 节 Record Manager 接口

**+ createTable(Table\* table): bool** 按照传入的 table 指针建表，创建 mdb 文件。若表已存在或者创建失败，返回 false 并输出错误信息。

**+ dropTable(Table\* table): bool** 删除对应的 mdb 文件，若表不存在或者删除失败，返回 false 并输出错误信息。

**+ select(Table\* table, Condition\_list clist): Data\*** 按照条件查找对应记录，返回存有记录的 data 数据结构。

**+ select(Table\* table): Data\*** 首先判断表是否存在，存在的话返回整个 data 数据结构。

**+ insert(Record& record): int** 首先判断表是否存在，然后对记录进行检验，是否存在主键相同的问题，然后插入记录并返回记录所存储位置的 id。

**+ deleteValue(Table\* table, Condition\_list clist): bool** 按照条件删除表中的对应记录，成功返回 true，失败返回 false 并输出错误信息。

**+ deleteValue(Table\* table): bool** 删除整张表中的记录。

### 第 3.5 节 Index Manager 接口

**bool CreateIndex(const char\* filename, int KeyLength);**

根据索引名、键长创建索引

**bool DropIndex(const char\* filename);**

根据索引名删除索引

**int Find(const char\* filename, const char\* key);**

根据索引名、键值获取整数地址信息

**bool Insert(const char\* filename, const char\* key, const int value);**

根据索引名、键值和整数地址信息插入键值

**bool Remove(const char\* filename, const char\* key);**

根据索引名、键值移除键值

## 第 3.6 节 Buffer Manager 接口

`Block* get_block(const char* filename, int id)`

获取内存中第 id 个 block, 同时将该 block 置换至链表头; 如果缓存区已满(链表长度达到最大长度), 从链表尾部将 block 写入 disk。

```
Block* BufferManager::get_block(const char* filename, int id) {
    string block_name = string(filename) + "." + to_string(id);
    if (node_map.find(block_name) != node_map.end()) {
        // Set block as most recently used
        Block_node* node = node_map[block_name];
        node->remove_node();
        node->attach(lru_head);
        return node->block;
    }
    if (block_cnt == MAX_BLOCK_COUNT) {
        // Current block number full
        // Find the least recently used block
        Block_node* node = lru_tail->pre;
        while (node->block->pin)
            node = node->pre;
        delete_from_mem(node);
    }
    return load_from_file(filename, id);
}
```

`void remove_block(const char* filename)`

在内存中删除以 filename 命名的 block, 同时将它们写入 disk

```
void BufferManager::remove_block(const char* filename) {
    Block_node* next_node;
    for (Block_node* node = lru_head->next; node != lru_tail; node =
next_node)
    {
        next_node = node->next;
        if (node->block->file_name == filename)
            delete_from_mem(node, false);
    }
}
```

## 第 4 章 MINISQL 系统测试

### (1) 创建表

```
=====Welcome to minisql!=====
minisql> create table stud (
    -> sno char(8),
    -> sname char(16) unique,
    -> sage int,
    -> sgender char (1),
    -> primary key ( sno )
    -> );
Success to create 1530431666 indxe.
Success to create stud table.
1 table created. Query done in 0.003s.
```

### (2) 插入数据

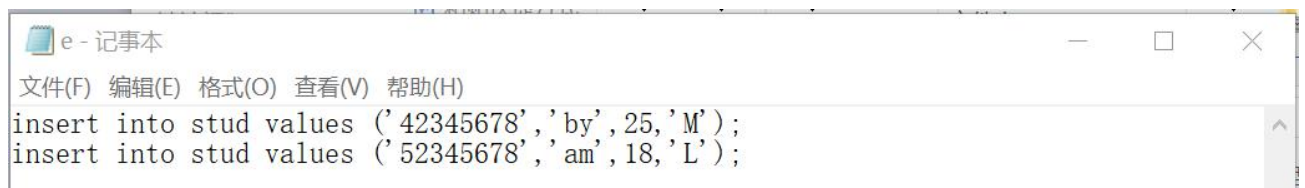
```
minisql> insert into stud values ('12345678','wy',22,'M');
Success to insert.
1 record inserted. Query done in 0.002s.

minisql> insert into stud values ('22345678','cc',20,'L');
Success to insert.
1 record inserted. Query done in 0s.

minisql> insert into stud values ('32345678','xc',19,'L');
Success to insert.
1 record inserted. Query done in 0.001s.
```

### (3) 执行文件

文件内容:



```
e - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
insert into stud values ('42345678','by',25,'M');
insert into stud values ('52345678','am',18,'L');
```

执行结果:

```
minisql> execfile e;
Success to insert.
1 record inserted. Query done in 0.001s.
Success to insert.
1 record inserted. Query done in 0.001s.
```

#### (4) 无索引查找

不带条件:

```
minisql> select * from stud;
sno      sname    sage    sgender
12345678      wy      22      M
22345678      cc      20      L
32345678      xc      19      L
42345678      by      25      M
52345678      am      18      L
5 record(s) selected. Query done in 0.004s.
```

带条件:

```
minisql> select * from stud where sno = '12345678';
sno      sname    sage    sgender
12345678      wy      22      M
1 record(s) selected. Query done in 0.002s.
```

#### (5) 创建索引

```
minisql> create index stunameid on stud ( sname );
Success to create stunameid indxe.
1 index created. Query done in 0.004s.
```

#### (6) 删除索引

```
minisql> drop index stunameid;
Success to drop stunameid index.
1 index dropped. Query done in 0.001s.
```

#### (7) 删除记录

条件删除

```
minisql> delete from stud where sno = '12345678';
Success to delete.
1 record(s) deleted. Query done in 0.001s.
```

全部删除



```
minisql> delete from stud;
Success to delete.
4 record(s) deleted. Query done in 0.001s.
```

#### (8) 删除表

```
minisql> drop table stud;
Success to drop 1530432768 index.
Success to drop stud table.
1 table dropped. Query done in 0.003s.
```

#### (9) 退出

```
minisql> quit

Bye~ ^ ^
请按任意键继续. . .
```

#### (10) 新建索引查询

```
minisql> insert into stud values ('12345678','wy',22,'M');
Success to insert.
1 record inserted. Query done in 0.003s.

minisql> insert into stud values ('22345678','cc',20,'L');
Success to insert.
1 record inserted. Query done in 0.005s.

minisql> insert into stud values ('32345678','xc',19,'L');
Success to insert.
1 record inserted. Query done in 0.002s.

minisql> create index studindex on stud(sname);
Success to create studindex indxe.
1 index created. Query done in 0.002s.

minisql> select * from stud where sname = 'cc';
sno      sname    sage    sgender
22345678      cc      20      L
1 record(s) selected. Query done in 0.004s.

minisql> _
```

## 第 5 章 分工说明

本系统的分工如下：

Interpreter 模块.....	邓墨琳
API 模块.....	王立冬
Catalog Manager 模块 .....	彭昊
Record Manager 模块.....	陈逸雪
Index Manager 模块.....	王立冬
Buffer Manager 模块.....	邓墨琳
数据库 schema 数据结构设计 .....	邓墨琳
总体设计报告.....	ALL
模块汇总.....	彭昊
测试.....	陈逸雪

## 第 6 章 附录

### miniSQL 文法生成规则

```
stmt_list: error ';'
|      stmt_list error ';'

stmt_list: stmt ';'
|      stmt_list stmt ';'
;

stmt: CREATE TABLE NAME '(' attr_info ')'
|      DROP TABLE NAME
|      CREATE INDEX NAME ON NAME '(' NAME ')'
|      INSERT INTO NAME VALUES '(' attr_value_list ')'
|      DELETE FROM NAME
|      DELETE FROM NAME WHERE wh_list
|      SELECT '*' FROM NAME
|      SELECT '*' FROM NAME WHERE wh_list
|      EXECFILE NAME
|      QUIT
;

attr_info: attr_list
|      attr_list ',' PRIMARY KEY '(' NAME ')'
;

attr_list: attr
|      attr_list ',' attr
;

attr: NAME data_type
|      NAME data_type UNIQUE
;

data_type: INT
|      FLOAT
|      CHAR '(' INTNUM ')'
;

attr_value_list: real_value
|      attr_value_list ',' real_value
;

wh_list: wh_name
```

```

|   wh_list AND wh_name
wh_name: NAME COMPARISON real_value
real_value: STRING
|   INTNUM
|   FLOATNUM
|   EMPTY
;

```

## Makefile

```

CC = g++ -std=c++11
IT_SRC = Interpreter/
LY_SRC = Interpreter/LEXYACC/
API_SRC = API/
BM_SRC = BufferManager/
IM_SRC = IndexManager/
CM_SRC = CatalogManager/
RM_SRC = RecordManager/

minisql : MiniSQL.cpp MiniSQL.h Global.h\
    $(LY_SRC)yacc.cpp $(LY_SRC)yacc.hpp $(LY_SRC)lex.cpp \
    $(IT_SRC)CST.cpp $(IT_SRC)CST.h $(IT_SRC)interpreter.h \
    $(API_SRC)API.cpp $(API_SRC)API.h \
    $(BM_SRC)BufferManager.cpp $(BM_SRC)BufferManager.h \
    $(IM_SRC)BpTree.cpp $(IM_SRC)BpTree.h $(IM_SRC)BpTreeNode.cpp
$(IM_SRC)BpTreeNode.h \
    $(IM_SRC)IndexManager.cpp $(IM_SRC)IndexManager.h\
    $(CM_SRC)catalogManager.cpp $(CM_SRC)catalogManager.h\
    $(RM_SRC)recordManager.cpp $(RM_SRC)recordManager.h
$(RM_SRC)data.cpp $(RM_SRC)data.h
    $(CC) -o minisql MiniSQL.cpp $(LY_SRC)yacc.cpp $(LY_SRC)lex.cpp \
    $(IT_SRC)CST.cpp $(API_SRC)API.cpp \
    $(BM_SRC)BufferManager.cpp $(IM_SRC)BpTree.cpp
$(IM_SRC)BpTreeNode.cpp \
    $(IM_SRC)IndexManager.cpp\
    $(CM_SRC)catalogManager.cpp\
    $(RM_SRC)recordManager.cpp $(RM_SRC)data.cpp

```