浙江大学

数据库系统实验报告

作业名称：	Minisql
姓　　名：	上官越 陈婉仪 李鸿屹
学　　号：	3170104168 3170104968 3170106234
电子邮箱：	sgyersula@163.com
联系电话：	18988836322
指导老师：	孙建伶

2019 年　6 月　23 日

实验名称:　　MiniSQL

一、　实验目的

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

二、　系统需求

2.1 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 <= n <= 255 。

2.2 表定义

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

2.3 索引的建立和删除

对于表的主键自动建立 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）。

2.4 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。

2.5 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(where 条件是范围时删除多条)。

三、　实验环境

Visual Studio 2017

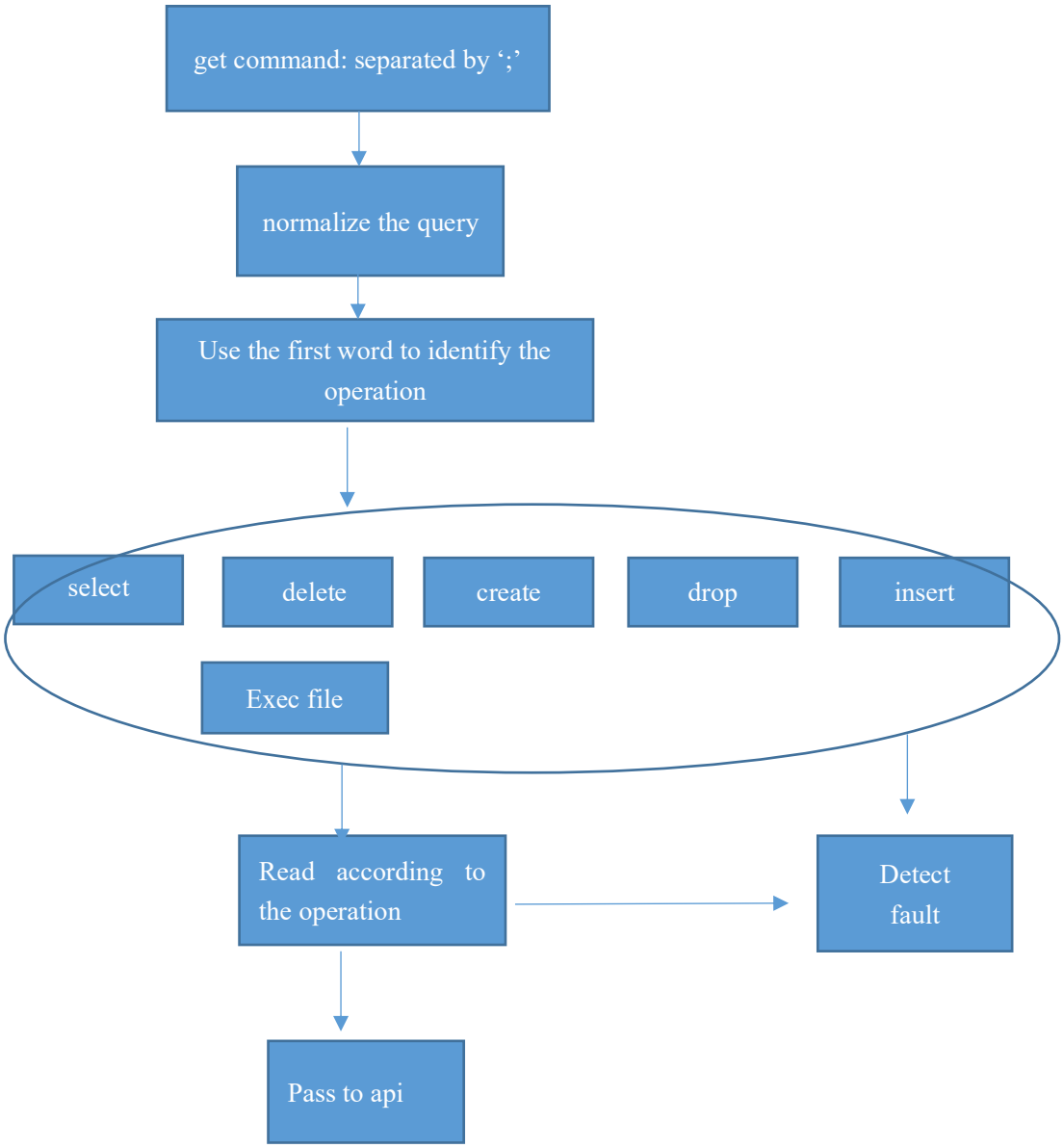
四、　系统设计

4.1 分工情况

上官越 负责 Interpreter，api，catalog manager 模块
陈婉仪 负责 buffermanager，recordmanagermokuai1
李鸿屹 负责 indexmanager 模块

4.2 Interpreter

程序执行流程图：



主要数据结构 ：Api 类，catalog 类，stringstream 类。

除了数据结构中的类，还有几个主要的类，分别是：

exception 类：定义了一个 exception 类，通过继承派生 exception 类，获得了多个表示 minisql 的错误类型的类。其中包括：

（1）枚举变量 type，用于在 interpreter 中通过 switch 捕捉错误类型

```
typedef enum error {
    e_table_exist, e_table_not_exist, e_attribute_not_exist,
    e_index_exist, e_index_not_exist, e_primary_key_conflict,
    e_not_unique,  e_data_type_conflict, e_unique_conflict,
    e_syntax_error,e_format_error, e_invalid_indentifier
}type;
```

（2）EXCEPTION: 继承自 exception，增加了 type 和 string 类型的变量。

```
class EXCEPTION:public exception {
public:
    type name;
    string target;
};
```

（3）与 type 类对应的类（如 class table_not_exist），在初始化时将内部的 type 初始化为其对应的类型，并读入一个 string 类，用于错误输出。

Limit/requirement 类：表示输入 select 和 delete 的限定条件。
class limit:用于表示 select 和 delete 中的一个条件。

```
struct limit {
public:
    string attrname;
    int operid;
    element data;
    limit(string attrname, int operid, element data) {
        this->attrname = attrname;
        this->operid = operid;
        this->data = data;
    }
};
```

class requirement: select, delete 中往往有多个条件，requirement 包括了其定义在哪个表上，以及其限制（一个或多个）。

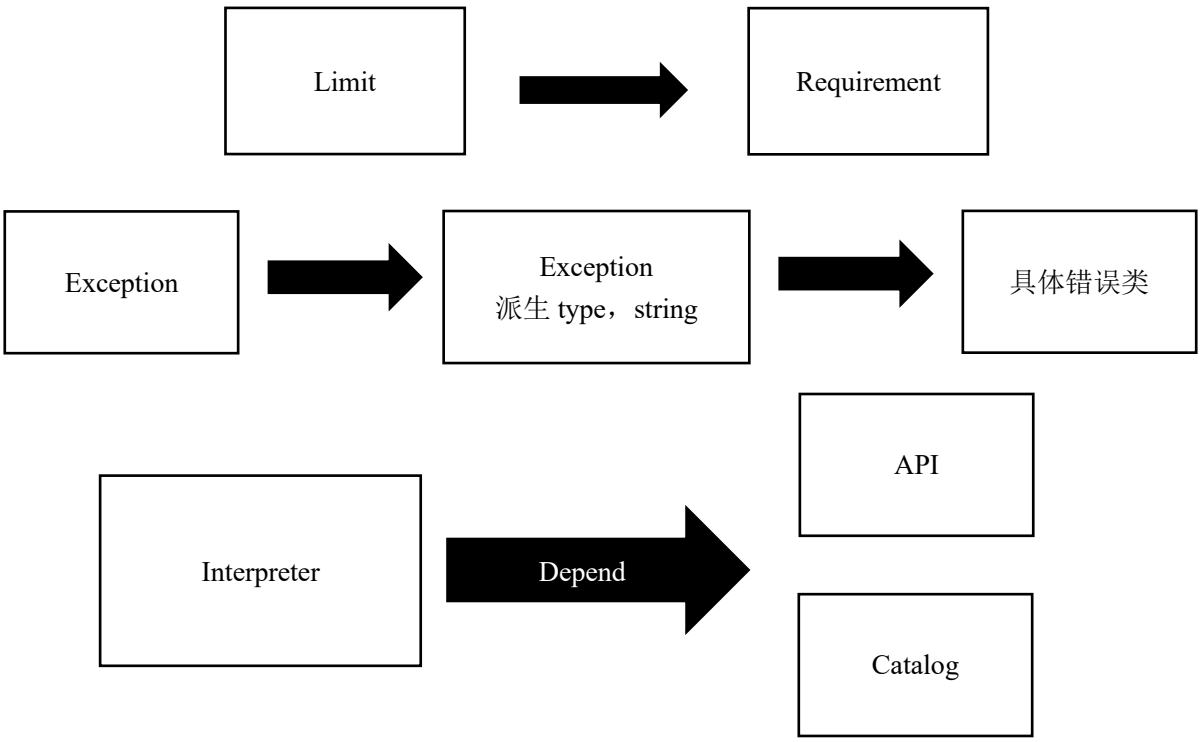
Interpreter 类：用于顶层处理，调用下层模块。

```
class Interpreter {
public:
    Interpreter();
```

```
4.      void getcmd();
5.      /*
6.      1. split it into several query if ';' exists;
7.      2. check syntax error: whether it ends with ';'
8.          error type:
9.      3. analyse each query, delete the white space.
10.     */
11.
12.     void normalize();
13.     /*
14.     in this function , we need to make the input query easy to split
15.     */
16.
17.     int process();
18.     /*
19.     1. analyse which type the query belongs to
20.         if not exists throw "command not exists"
21.     2. go into each function to process this query
22.     */
23.
24.     void sql_insert(stringstream &ss);
25.     /*
26.     format: insert into 表名 values(v1,v2...);
27.     check: table exists? attribute match?
28.     */
29.
30.     void sql_delete(stringstream &ss);
31.     /*
32.     format: delete from 条件 where 表名;
33.     check: table exists? attribute match?
34.     */
35.
36.     void sql_exit();
37.     /*
38.     exit
39.     */
40.
41.     void sql_execfile(string file_path);
42.     /*
43.     exec the file
44.     */
45.
46.     void sql_select(stringstream &ss);
47.     /*
```

```
48.     format: select 列名 from 表名 where 条件;
49.     check: table exists? attribute match?
50.     */
51.
52.     void create_table(stringstream &ss);
53.     /*
54.     create table 表名
55. (
56.     列名 类型() ,
57.     列名 类型() ,
58.     列名 类型() ,
59.     primary key ( 列名 )
60.     );
61.     */
62.
63.     void drop_table(stringstream &ss);
64.     /*
65.     drop table 表名;
66.     */
67.
68.     void create_index(stringstream &ss);
69.     /*
70.     create index 索引名 on 表名 ( 列名 );
71.     check: table exist? attribute exist? have index or not?
72.     */
73.
74.     void drop_index(stringstream &ss);
75.     /*
76.     drop index 索引 on 表名;
77.     check: attribute exist? table exist? index exist?
78.     */
79.
80.     element changeformat(string tablename, string target,int count);
81.
82. private:
83.     string query;
84.     API api;
85. };
86.
87.             element Interpreter::changeformat(string tablename,string target
    ,int id);
88.                 //change the string into an element
```

类间关系：



如何检测 SQL 是否有错误

Interpreter 通过 getcmd 获得由';'分割的初始语句，在合适的地方插入空格，将语义分开。将原先 string 类型的 query 变成一个 stringstream 类型的对象，该对象可以自动根据空格分开单词。在 interpreter 中根据 minisql 实验指导书中的语法规则对单词进行读入和判断，如果不符合预期，则抛出语法错误。由 exception 的类型定义可知，抛出的类有一个 string 类型的成员函数，可以帮助我们定位到错误的单词。以及 e_primary_key_conflict，用来判断是否有单一的主键。

逻辑上的判断大部分由 api 或更底部的模块进行。如：

1. `e_table_exist,`：`table` 已存在，不可再创建。
2. `e_table_not_exist,`：`table` 不存在，不可删除。
3. `e_attribute_not_exist,`：该属性不存在。
4. `e_index_exist,`：`index` 已存在，不可再创建。
5. `e_index_not_exist,`：`index` 不存在，不可删除。
6. `e_data_type_conflict`：在条件中的比较对象与定义对象不符，如将 `char` 类和浮点数比较。
7. `e_format_error`：输入的数值不符合定义要求。
8. `e_unique_conflict`：在 `unique` 属性中已有相同值。
9. `e_primary_key_conflict`：没有主键

```
10. e_not_unique：不能在非 unique 的属性上定义索引
11. e_invalid_indentifier：未识别该运算符
```

4.3 API

功能描述：

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。

从 interpreter 获得一个指令的基本信息后，通过 api 模块调用 catalogmanager，recordmanager 和 indexmanager 的接口，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，后返回执行结果给 Interpreter 模块。

功能与 interpreter 中的指令一一匹配，有 select，delete，create table，create index， drop table， drop index， insert 等几个基本操作。

主要数据结构：

Table, element, attribute, recordmanager, catalogmanager, indexmanager：均已在后文定义。

class sqlreponse:用于 api 的返回值,表示是否成功，影响的行数和耗时。

```
1.  class sqlresponse {
2.  public:
3.      string op; //操作类型
4.      bool success; //是否成功
5.      int rowaffected; //影响的行数
6.      double time//消耗的时间
7.      sqlresponse();
8.      sqlresponse(string ope, bool suss, int row,double duration);
9.  //构造函数
10.     void output();//写出
11. };
```

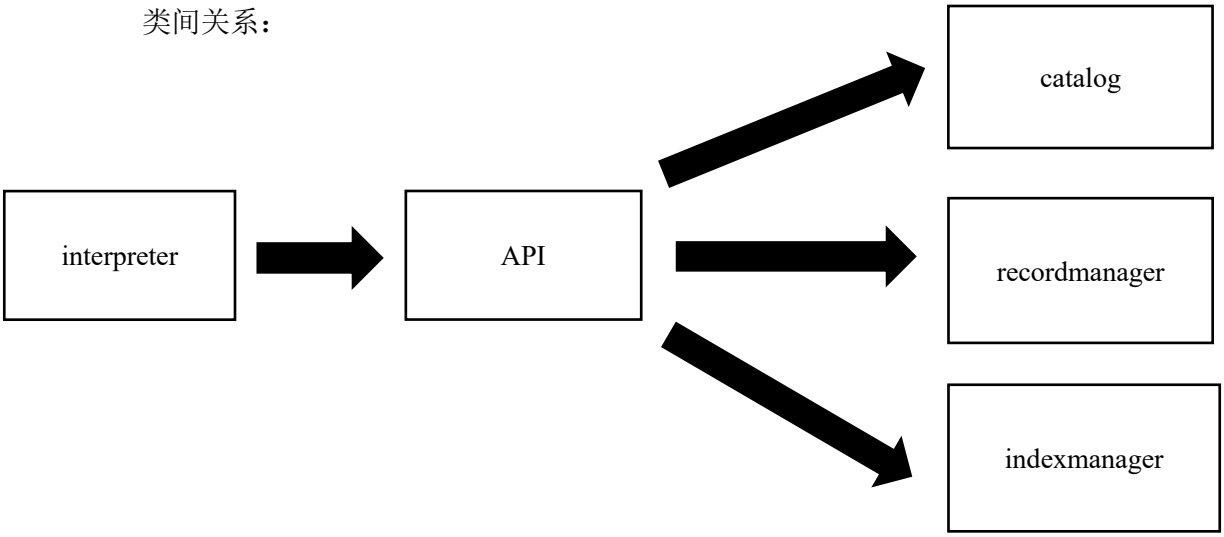
类与类间关系
除了数据结构中用到的类，还有 api 类，定义如下：

```
1.  class API {
2.  public:
3.      API() {};
4.      sqlresponse Createtable(string tablename, vector<attribute> attr);
5.      sqlresponse Createindex(string tablename, string indexname,string attr);


6.
```

```
7.      sqlresponse Droptable(string tablename);
8.      sqlresponse Dropindex(string indexname);
9.
10.     sqlresponse Select(requirement require,int print);
11.     sqlresponse Insert(string tablename, vector<element> data);
12.     sqlresponse Delete(requirement require);
13.
14.
15.     void printout(table mytable, vector<vector<element>> rows);
16.
17.     RecordManage rm;
18.     CatalogManager cm;
19.     IndexManager im;
20. };
```

类间关系：



4.4 Catalog Manager

功能描述

Catalog Manager 负责管理数据库的所有模式信息，用于为 api 提供接口。

其模式信息包括：

1. 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引，表的 buffer 储存长度等等。

2. 表中每个字段的定义信息，包括字段类型、是否唯一、长度、是否是主键等。

3. 数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。 Catalog Manager 还必需提供访问及操作上述信息的接口，供 Interpreter 和 API 模块使用。

其功能包括：

1. 关于 table 的 create，drop 操作，判定一个特定 tablename 的 table 是否存在，根据 table 的名字获取 table 信息。

2. 关于 index 的 create，drop 操作，判定一个特定 indexname 的 index 是否存在，根据 index

的名字获取 index 信息。
3.根据一个 tablename 和 attributename 判定是否存在一个满足条件的属性；根据一个 tablename 和 attributename 获得该属性。

主要数据结构：
Catalog 需要建立表的信息（table），表的信息中包含属性的信息（attribute）table 和 attribute 在 record 模块都已有定义，就不再在此赘述。

cmIndex: 与 table 定义相似，包含定位一个索引所需要的所有信息。

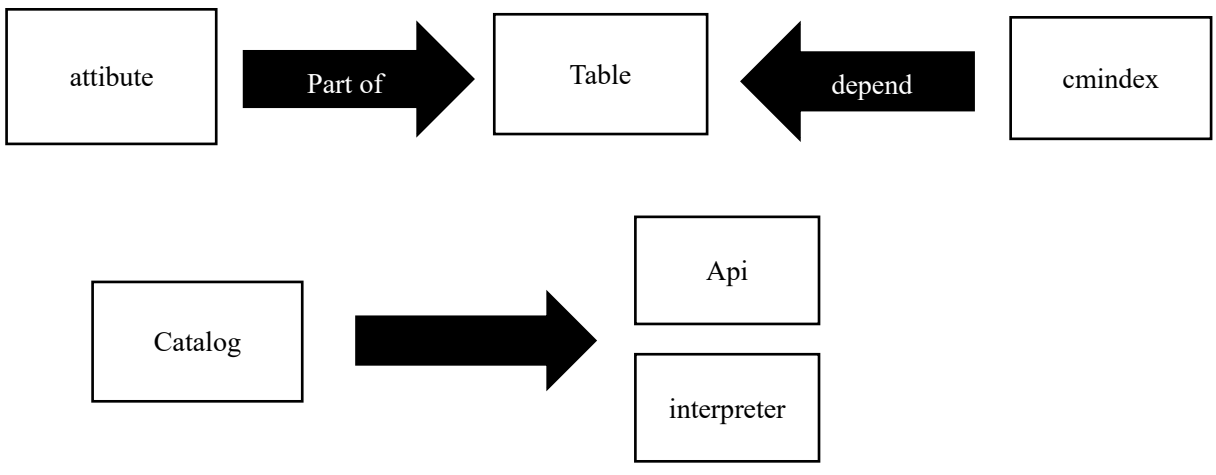
```
1.  class cmIndex {
2.  public:
3.      string tablename;
4.      string indexname;
5.      string attrname;
6.
7.      cmIndex() {};
8.      cmIndex(string tablename, string indexname, string attrname);
9.  //initialize the index
10.     void writeout();
11. //write out the index information
12.     void getin(string indexname);
13. //get the index information
14. };
```

包含定位一个索引所需要的所有信息，writeout 和 getin 分别用来写出和获得索引信息。

类与类间关系：

除了数据结构的类外，还有 catalogmanager 类，类定义如下：

类间关系：



4.5 Record Manager

功能描述

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。

1) 创建与删除表：每次 create table 或 drop table 时都要调用 Record Manager，进行表的数据文件的创建与删除

2) 记录的插入：在插入数据时，调用 Record Manager 将数据插入内存中并写到文件中进行保存，并返回给 API 数据所插入的在内存中的位置

3) 记录的删除：在删除数据时，根据 API 所提供的记录在内存中的位置找到记录并标记，进行懒惰删除，并将此变化写进文件进行保存

4)记录的查找：在查找数据时，API 会根据所输入指令判断条件找出所有符合条件的记录所在的内存中的位置，然后调用 Record Manager 将内存中的数据转化为 element 格式返回给 API 进行处理

主要数据结构:

表中的数据，一个属性下的值

```
1.  struct element
2.  {
3.      int i ;
4.      float f;
5.      string s;
6.      int type; // 0-int; 1-double; 2-string; -1:invalid
7.  };
```

其中 i, f, s 的初始值都为-1，如果 type 为 0，则将数据储存到 i 中；如果 type 为 1，则将数据储存到 f 中；如果 type 为 2，则将数据储存到 s 中。

表的属性

```
1.  struct attribute {
2.      string name; //属性的名字
3.      int type; //0: int, 1: double, 2: string, -1:invalid
4.      int charnum; //如 varchar(20)的 20,如果为 int 则是 int 的最大数长度为 10，float
    亦然
5.      int Isprimary;//是否为主键
6.      int id; //是表里的第几个属性
```

```
    bool IsUnique; //属性是否 unique
    long StartPosition; //属性在存储时开始的位置
    bool HasIndex; //是否有索引
    string Index; //索引名
};
```

这是建表时就确定好的属性的信息，成员函数未写出。

表：

```
class table
{
public:
    string tablename; //表的名字
    string primarykeys; //表的主键
    int primaryid; //主键是第几个属性
    int EntrySize; //一条 Record 长度
    int AttrCount; //属性数量
    int BlockCount; //表在内存中所占的块数
    vector<attribute>attr;
};
```

记录的选择的结果：

```
class Data {
public:
    vector<vector<element>> rows;
};
```

vector<element>可以表示一条记录的值，Data 是记录的集合。
记录在内存中存储的位置

```
typedef pair<int, int> tag;
```

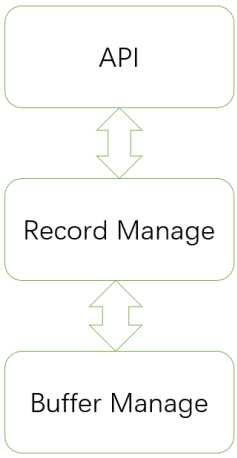
其中第一个是在内存中的块号，第二给是在一个块信息 data 中开始的位置。

类与类间关系

```
class RecordManage {
public:
    BufferManage bm;
    /*引入 buffer*/
    void CreateTable(table &t);
```

```
    /*创建一个信息为 RecordTable 的表文件*/
    void DropTable(table &t);
    /*删除信息为 RecordTable 的表文件*/
    tag InsertRecord(table &t, vector<element>colomns);
    /*返回记录所在的 bufferIndex 和 dataPosition*/
    vector<element>select(table &t, tag T);
    /*根据 tag 找到记录并返回这条记录具体的值*/
    void Delete(table &t, tag T);
    /*根据 tag 找到记录进行惰性删除*/
    void Delete(table &t);
    /*删除整个表的记录*/
    vector<tag> FindAllTag(table &t);
    /*返回给 API 整张表记录在内存中的储存位置*/
private:
    string transtostring(table &t, vector<element>colomns);
    /*将 vector<element>转化为字符串方便放入内存中*/
    vector <element> transtoelement(table &t, string str);
    /*从内存中获取字符串然后转化为 vector<element>给 api*/
    int mallocBlockInTableFile(table &t);
    /*给数据文件开辟缓存*/
};
```

其中函数为 public 的为提供给 API 的接口，为 private 即内部自己的调用的函数。



图：与 Record Manage 有关的类间关系

RecordManage 要调用 BufferManage 模块，进行数据文件的创建、写入、修改与删除，而 RecordManage 将提供接口给 API 进行记录的插入、选择与删除操作。

4.6 Index Manager

功能描述

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、范围查找等操作，并对外提供相应的接口。

B+树的创建：B+树的创建即在已有的大量数据上根据 API 传递过来的需求，调用 Record 管理的记录在内存中的位置信息实现 B+树的建立。

等值查找：等值查找根据 API 传入的查询要求，在 B+树内查找相应的叶子节点，并返回记录的位置信息。依赖的是 B+树的搜索树的性质，在 B+树上自顶向下进行查询。

插入键值：接收 API 传入的键值，先找到其所在位置，再将其插入叶子节点上，如果插入导致 B+树的性质收到破坏，将自底向上地对 B+树的结构进行调整。

删除键值：找到对应的位置，然后将对应节点上的可访问标记进行修改：惰性删除。

范围查找：采用剪枝的思想，对 B+树进行便利，不符合要求的节点不进行递归，从而获取所有符合要去的节点的索引值。

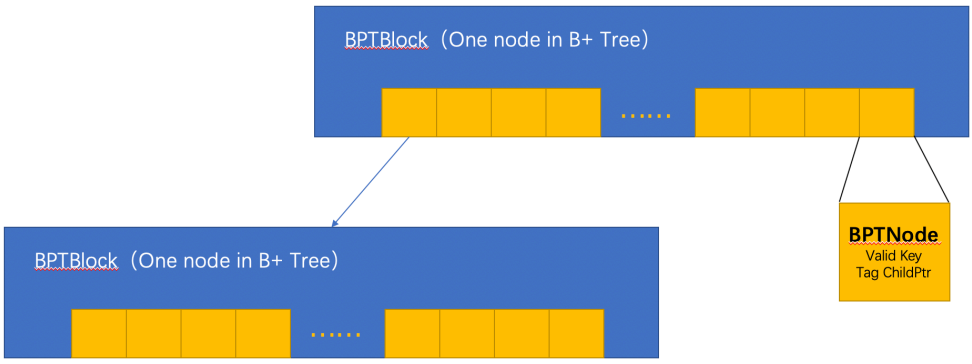
B+树中节点大小与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。在该工程中，B+树节点占用内存同缓冲块大小相同，均为 4 kB，B+树的叉数为 55。

本次工程中，由于采用的索引为内存位置信息索引，因此 B+树也是建立在内存中，建立索引后如果关机，索引也随之消失，再次开机需重新建立索引。

主要数据结构

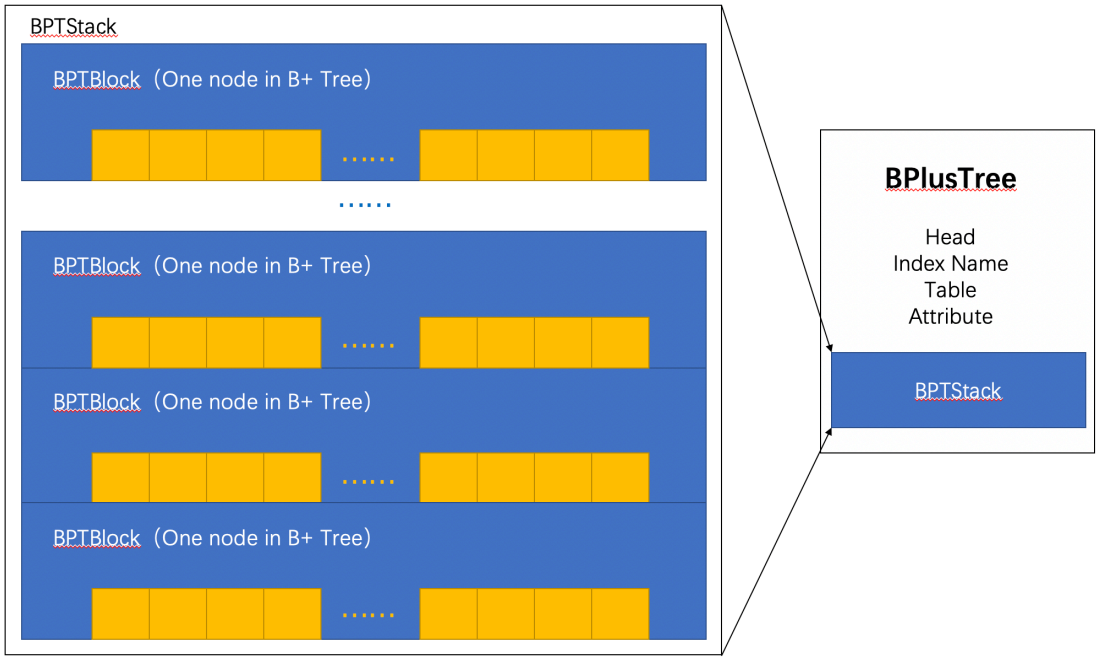
Index Manager 所采用的数据结构就是 B+树，B+树的基本结构与原理理较为复杂这里不再赘述，主要描述本次工程中 B+树的设计。

最初为实现存储的方便，B+树采用了数组存储（vector）而不是指针连接的方式进行构建，单个节点的结构设计如下图所示：



所有的节点连续存放在如下所示的 BPTStack 中，这个 BPTStack 也就代表了整个 B+树的结构。值得注意的是，每一个 BPTBlock 中第一个 BPTNode 的 Key 都是-1，以表示 B+树上小于父节点中最小值的子树这一信息。

而除了 BPTStack 以外，为了方便索引的管理，实际 BPlussTree 的内容还包括根节点的序号，对应的表的名字等等信息，如下图所示。



他们具体的定义如下所示：

BPTNode：

```
//B+树节点的一个小块
class BPTNode {
public:
    element Key;                // 关键值
    long ChildPtr;              // 非叶子节点的指针，指向 B+树起孩子节点
    bool Valid;
    tag IndexNumber;
};
```

BPTBlock：

```
struct BPTBlock
{
    int Layer;                  // 层数
    int Number;                 // 孩子的个数
    long Parent;                // 祖先节点
    BPTNode BPTList[BranchNum + 1];
};
```

BPlusTree

```
//一棵 B+树
struct BPlusTree
{
```

```
    int Head;                    //根结点的位置
    string IndexName;            //索引树的名字，用于唯一标识特定的 B+树
    string Table;                //表的名字
    string Attribute;            //索引数索引的属性
    vector<BPTBlock> BPTStack;   //B+树的上层
};
```

类与类间关系

Index Manager 的类定义如下：

```
class IndexManager {
public:
    IndexManager() {};

    //RM
    RecordManage RM;

    //打印 B+树
    void Visit(BPTBlock Tree, BPlusTree TESTINGTREE, int Layer);

    //在 Buffer 的基础上创建一棵 B+树
    BPlusTree CreateTree(const string& IndexName, const string& Table, const string& Attribute, table& T);

    //删除某个指定的 B+树
    bool DropTree(const string& IndexName);

    //插入数据，B+树的架构做相应的变化
    bool InsertRecord(const string& IndexName, element& e, tag offset);

    //删除数据，B+树的架构做相应的变化
    bool DeleteRecord(const string& IndexName, element& e);

    //寻找查找某一条记录所有相关的记录所在的位置（返回的是偏移量）
    vector<tag> FindRecord(const string& IndexName, element& e);

    //返回所有大于某个值的记录的索引(开区间)
    vector<tag> Greater(const std::string& indexName, element& e);

    //返回所有小于某个值的记录的索引(开区间)
    vector<tag> Less(const std::string& indexName, element& e);

```

```
    //将某棵 B+树存回文件
    void BPT_To_File(const string& IndexName, BufferManage& BFM);

    //查找树：存在返回下标，不存在返回-1
    long FindTree(const string& IndexName);

    IndexManager(RecordManage& NRM) {
        RM = NRM;
    }
private:
    //范围访问
    vector<tag> GreaterVisit(BPTBlock Tree, vector<tag> List, element& e, BPlusTre
e TESTINGTREE);

    vector<tag> LessVisit(BPTBlock Tree, vector<tag> List, element& e, BPlusTree T
ESTINGTREE);

    vector<tag> EQVisit(BPTBlock Tree, vector<tag> List, element& e, BPlusTree T
ESTINGTREE);

    //管理的所有 B+树
    vector<BPlusTree> BPlusTreeLibrary;
};
```

从与其他类的关系上，与 Index Manager 关联的模块包括 API，Buffer Manager，Record Manager 三个部分具体的关系为：API 负责向 Index Manager 提出查询需求与插入删除指令；索引文件的存储依赖 Buffer Manager 进行，以实现以 4KB 为单元进行磁盘的交互；而如何获得记录对应的索引值就主要依赖 Record Manager 进行。

4.7 Buffer Manager

功能描述

Buffer Manager 负责缓冲区的管理，主要功能有：

1.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中

2. 实现缓冲区的替换算法 LRU，当缓冲区满时选择最近最少用到的页进行替换

3. 记录缓冲区中各页的状态，如是否被修改过等； 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

主要数据结构

宏定义：

```
#define OneBlockSize 4096 //一个内存块大小为 4KB
#define BlockNum 300//缓冲区内存块的最大数量
#define EMPTY '\0'
```

内存块，其中需要包括存放信息的字符型数组、磁盘文件信息、脏数据位、是否上锁、被使用次数等信息。其中成员函数被略去：

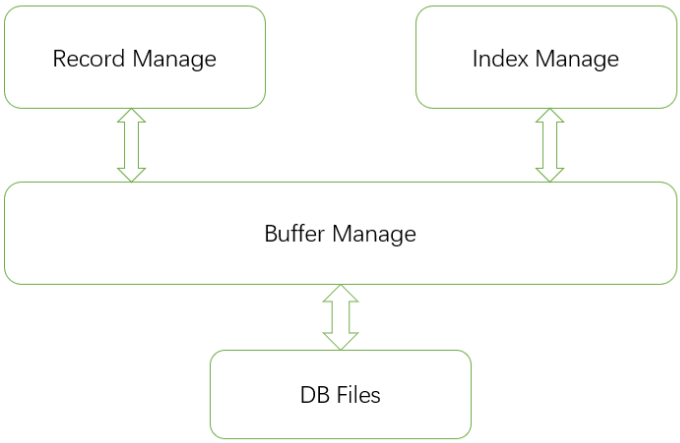
```
class Block
{
public:
    string fileName;/*表示 fileName 文件的内存块*/
    bool Dirty;/*是否被修改*/
    bool isEmpty;/*是否为空*/
    bool Pin;/*是否被锁*/
    int  blockIndex;/*表示对应的 fileName 文件的块号*/
    int  LRU;/*表示未使用次数*/
    char data[OneBlockSize + 1]; /*内存块中储存的信息*/
};
```

类以及类间关系

```
class BufferManage
{
public:
    Block bufferBlock[BlockNum];
    /*内存块*/
    BufferManage();
    /*构造函数*/
    ~BufferManage();
    /*析构函数*/
    void BufferFlush(int bufferIndex);
    /*将下标为 bufferIndex 的这一内存块写到文件中*/
    int GetBufferIndex(string fileName, int blockIndex);
    /*根据文件的名字和块号得到内存中的下标，返回缓存索引 bufferIndex*/
    void BufferRead(string fileName, int blockIndex, int bufferIndex);
    /*从磁盘中读取信息存到内存钟*/
    void RecordBuffer(int bufferIndex);
    /*记录 buffer 调用次数，用于 LRU 算法*/
    int GetEmptyBuffer();
    /*找到空闲的缓存索引*/
    int checkInBuffer(string fileName, int blockIndex);
    /*查找磁盘中该块是否已经写入了内存*/
    void BufferPin(int bufferIndex);
```

```
23.     /*将缓冲区锁住，使其不可被替换*/
24.     void deleteFile(string fileName);
25.     /*删除磁盘文件*/
26.     void writeToFile(int bufferIndex);
27.     /*将内存块中的信息写到文件中但是不清空内存*/
28. };
```

Buffer Manage 中有个子类是 block，即内存中的块，大小为 4096bit，内存区最大的块数为 300。所有的接口都是对外开放提供给 Record Manage 和 Index Manage 的。



图：与 Buffer Manage 有关的类间关系

4.8 DB Files

存储文件格式说明

表的文件名为 tablename+".table"
索引的文件名为 indexname+".index"
目录的文件名为 catalogname+".cm"

五、 系统实现分析及运行截图

1. 创建表语句

1. 若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。
2. 分析执行该 SQL 所需的模块以及程序执行流程

Interpreter.getcmd->
Interpreter.createtable->
Api.createtabl->
Recordmanager.createtable->
Catalagmanager.createtable->

3.运行结果截图

示例语句：

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

```
Hello, minisql.
 >> create table student2(
 >> id int,
 >> name char(12) unique,
 >> score float,
 >> primary key(id)
 >> );
create table succeed.    0 rows affected.        (0.004s)
```

错误案例：

```
 >> create tabl on( );
you should only create type'table' and type 'index'
syntax error: error may exist around tabl
```

```
 >> create table student2(
 >> id int,
 >> name char(12) unique,
 >> score float,
 >> primary key(id)
 >> );
cm create table:
table student2 has existed!
```

```
 >> create table student1 (cno char(1));
primarykey does not exists!
```

2. 删除表语句

1. 若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。
2. 分析执行该 SQL 所需的模块以及程序执行流程

Interpreter .getcmd->
Interpreter.droptable->
Api.droptable->
对表上所有索引调用 indexmanager.drop->
Recordmanager.droptable->
Catalog.dropindex, catalog.droptable

3. 运行结果截图

```
>> drop table student2;
drop table succeed.      0 rows affected.          (0.005s)
```

```
>> drop table student1;
table student1 has not existed!
```

3. 创建索引语句

1. 若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。
2. 分析执行该 SQL 所需的模块以及程序执行流程

Interpreter .getcmd()->
Interpreter.createindex()->
Api.createindex()->
catalog.createindex()->
indexmanager.createindex(在此将先前记录插入)

3. 运行结果截图

```
>> create index stuid on student2(id);
Tagsize 1046
Find: 0
Arrtibute: 0 id
Maplist 1046
create index succeed.    0 rows affected.          (0.015s)
```

索引有效提高了查找速度：

```
>>  select * from student2 where id < 1080100500;
```

建立之前：

```
select succeed.          499 rows affected.       (2.1s)
```

建立之后：

```
select succeed.          499 rows affected.       (1.709s)
```

4. 删除索引语句

1. 若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。
2. 分析执行该 SQL 所需的模块以及程序执行流程

Interpreter .getcmd()->
Interpreter.dropindex()->
Api.dropindex()->
catalog.dropindex, indexmanager.drop->

3. 运行结果截图

示例语句：

```
drop index succeed.      0 rows affected.      (0.005s)
```

5. 选择语句

1. 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；若查询结果为空，则输出信息告诉用户查询结果为空；若失败，必须告诉用户失败的原因。
2. 分析执行该 SQL 所需的模块以及程序执行流程

   Interpreter.getcmd()->

   Interpreter.select()->

   Api.select()->

   Recordmanager.getalltag, recordmanager.select, indexmanager.select()->

   Api.printout()
3. 运行结果截图

Select*

```
>> select * from student2;

id                  name                score
1080109000          name9000            71.5
1080109001          name9001            67
1080109002          name9002            70
1080109003          name9003            68
1080109004          name9004            57.5
1080109005          name9005            76.5
1080109006          name9006            80.5
1080109007          name9007            62.5
1080109008          name9008            52.5
1080109009          name9009            95.5
1080109010          name9010            93
1080109011          name9011            62.5
1080109012          name9012            52.5
1080109013          name9013            51
```

略

```
1080109043          name9043            92.5
1080109044          name9044            81.5
1080109045          name9045            58.5
1080109046          name9046            59.5

select succeed.     47 rows affected.   (0.141s)
```

Select * from student2 单条件

```
>> select * from student2 where id == 1080109046;

id                  name                score
1080109046          name9046            59.5

select succeed.     1 rows affected.    (0.026s)
```

Select * from student2 双条件

找到

```
>> select * from student2 where id == 1080109046 and name == 'name9046';

id                    name                  score
1080109046            name9046              59.5

select succeed.       1 rows affected.      (0.026s)
```

没找到

```
select succeed.       1 rows affected.      (0.026s)
 >>  select * from student2 where id == 1080109046 and name == 'name9047';

id                    name                  score

select succeed.       0 rows affected.      (0.021s)
```

报错情况与 delete 类似

```
 >> select * from student2 where heelo == 1;
attribute heelo has not existed!
```

6. 插入记录语句

1. 若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。
2. 分析执行该 SQL 所需的模块以及程序执行流程
   Interpreter.getcmd()->
   Api.insert->(此过程需判定是否有 unique 冲突，会用到 select)
   Recordmanager.insert->
   If(exist) indexmanager.insert->
3. 运行结果截图

```
 >> insert into student2 values(108019995,'name1234',58.5);
insert succeed.          1 rows affected.          (0.002s)
 >> insert into student2 values(108019995,'name1234',58.5);
the attribute id is unique!
 >> insert into student3 values(108019995,'name1234',58.5);
table student3 has not existed!
 >> insert into student2 values(108019995,'name1234','hello');
the input may have format error around score definition! please justify it.
```

第一条为正确插入。

第二条为重复插入，不符合 unique 和 primary，被 primary 拦截，失败。

第三条插入了不存在的表，失败。

第四条插入了不符合表定义的属性，失败。

7. 删除记录语句

1. 若该语句执行成功，则输出执行成功信息，其中包括删除的记录数；若失败，必须告诉用户失败的原因。

2. 分析执行该 SQL 所需的模块以及程序执行流程
   Interpreter.getcmd()->
   Api.delete->（此过程内部与 select 相似）
   Recordmanager.delete->
   If(exist) indexmanager.delete->
3. 运行结果截图

成功：

```
>> select * from student2 where name == 'name9033';

-------------------------------------------------------------------------------------------------------------------
id                      name                    score
1080109033              name9033                67.5
-------------------------------------------------------------------------------------------------------------------

select succeed.         1 rows affected.        (0.009s)
>> delete from student2 where name == 'name9033';
delete succeed.         1 rows affected.        (0.002s)
>> select * from student2 where name == 'name9033';

-------------------------------------------------------------------------------------------------------------------
id                      name                    score
-------------------------------------------------------------------------------------------------------------------

select succeed.         0 rows affected.        (0.011s)
```

不存在这样的属性：（类似的有不存在这样的表等等）

```
>> delete from student2 where hello == 1;
attribute hello has not existed!
```

数据类型不匹配：

```
>> delete from student2 where name == 9.33;
datatype  name has conflict!
```

8. 退出 MiniSQL 系统语句

该语句的语法如下：

quit;

9. 执行 SQL 脚本文件语句

该语句的语法如下：

execfile 文件名 **;**

SQL 脚本文件中可以包含任意多条上述 8 种 SQL 语句，MiniSQL 系统读入该文件，然后按序依次逐条执行脚本中的 SQL 语句。

1. 若该语句执行成功，则输出执行成功的命令数；若失败，必须告诉用户失败的原因。
2. 分析执行该命令所需的模块以及程序执行流程
   Interpreter.getcmd()->获得执行文件的指令：打开文件
   Interpreter.execfile()->运行文件
   Interpreter.process()-> 获取每个语句，根据需要调用 api，并调用底层模块

   运行结果截图

```
>> execfile test.txt;
create table student2(  id int, name char(12) unique,    score float,    primary key(id) )
create table succeed.    0 rows affected.        (0.001s)
insert into student2 values(1080109000,'name9000',71.5)
insert succeed.          1 rows affected.        (0.002s)
insert into student2 values(1080109001,'name9001',67)
insert succeed.          1 rows affected.        (0.002s)
insert into student2 values(1080109002,'name9002',70)
insert succeed.          1 rows affected.        (0.003s)
insert into student2 values(1080109003,'name9003',68)
insert succeed.          1 rows affected.        (0.003s)
insert into student2 values(1080109004,'name9004',57.5)
insert succeed.          1 rows affected.        (0.002s)
insert into student2 values(1080109005,'name9005',76.5)
insert succeed.          1 rows affected.        (0.002s)
insert into student2 values(1080109006,'name9006',80.5)
```

略

```
insert succeed.          1 rows affected.        (0.006s)
insert into student2 values(1080109045,'name9045',58.5)
insert succeed.          1 rows affected.        (0.003s)
insert into student2 values(1080109046,'name9046',59.5)
insert succeed.          1 rows affected.        (0.003s)
execfile  succeed.       48 operations affected.          (s)
```

错误返回可参考每条指令。