数据库系统课程Project: miniSQL

详细设计报告

小组编号：B

张溢弛 **3180103772**

张琦 **3180103162**

聂俊哲 **3180103501**

〇、小组成员及分工

组员	任务
张溢弛 3180103772	interpreter层，API层，Catalog Manager层，bufferManager层，stringProcessor和Global层，报告撰写
张琦 3180103162	Record Manager层，buffer Manager层，项目的集成与debug，报告撰写
聂俊哲 3180103501	Index Manager层，项目的集成与debug，报告撰写

一、需求分析与功能介绍

1.1 实现功能分析

1.1.0 开发与运行环境

- 集成开发环境(IDE): JetBrains Clion 2020.01版
- 编程语言与版本：C++ 17标准
- 支持平台：Windows 10，macOS等操作系统

1.1.1 总体目标

- 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。

1.1.2 需求概述

- **数据类型**：支持的数据类型是`int,float,char(n)`等三种类型，并且char类型的长度在1到255之间
- **表的定义**：一张表最多定义32个属性，属性可以设置为**unique**和**primary key**
- **索引的建立和删除**：对于一张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语句来建立B+树的索引，所有的索引都是单属性单值的
- **查找记录**：查找记录的过程中可以通过用and进行多个条件的连接
    - 在此基础上我们小组实现了**or**的条件查询

- **插入和删除记录**：插入只支持单条记录的插入，删除操作支持一条和多条记录的删除
- 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
- 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
- 本系统主要通过输入一系列SQL语句执行来完成相应的操作和功能，SQL语句支持单行和多行的输入，最后必须用分号结尾作为SQL语句结束的标志
- 所有的关键字都采用小写的形式，如果输入的是大写字母会被自动转换为小写

1.1.3 支持的SQL语句与其他指令

- 启动系统：需要将源代码进行编译或者打开可执行文件进行启动
- 退出miniSQL系统的语句：`quit;`
- 创建表的语句：关键字为`create table`具体的语法格式如下

```
create table table_name(
    attribution1 date_type1,
    attribution2 data_type2 (unique),
    ……
    primary key(attribution_name)
);
```

- 创建索引的语句：关键字为`create index`，具体的语法格式如下

```
create index index_name on table_name (attribution_name);
```

- 删除表的语句

```
drop table table_name;
```

- 删除索引的语句

```
drop index index_name;
```

- 选择语句，关键字为`select`，只支持`select *`即显示全部属性

```
select * from table_name;
select * from table_name where conditions;
```

- 插入记录语句，关键字为`insert into`，支持where的条件表达式，语法格式如下

```
insert into table_name values (value1, value2, value3……);
```

- 删除记录语句，关键字为`delete from`，语法格式如下

```
delete from table_name;
delete from table_name where conditions;
```

- 执行SQL脚本文件

```
exefile file_name
```

二、各模块功能详解

2.1 Interpreter

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。

2.2 API

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

2.3 Catalog Manager

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。

2.4 Record Manager

Record Manager 负责管理记录表中数据的数据文件。主要功能为:

- 实现数据文件的创建与删除（由表的定义与删除引起）、记录的插入、删除与查找操作，并对外提供相应的接口。其中记录的查找操作要求能够支持不带条件的查找和
- 带一个条件的查找（包括等值查找、不等值查找和区间查找）

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

2.5 Index Manager

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

2.6 Buffer Manager

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，本系统中定为4KB

2.7 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。

三、各模块的具体实现与核心数据结构介绍

3.1 异常处理模块

- 在miniSQL处理SQL语句过程中，遇到各种各样的异常，比如SQL语句的**语法错误(Syntax Error)**，**运行时产生的错误(Run Time Error)** 等，本系统会在分析SQL语句语法和各个已经存在的数据表，索引之后，抛出异常提示来显示当前执行的SQL语句出现的问题

3.1.1 语法错误 Syntax Error

- 本系统会检测的语法错误如下，包含了SQL语句各种可能出现的问题

错误状态码	错误信息	具体含义
100	Syntax Error! Please try again!	quit语句语法错误
101	Syntax Error! Please check your brackets in the SQL!	SQL语句的括号匹配失败
102	Syntax Error! Extra parameters in create table	create table语句中有多余参数
103	Syntax Error! Extra parameters in create index!	create index语句中有多余参数
104	Syntax Error! Lack of parameters in create index!	create index 缺少参数
105	Syntax Error! No such create index operation!	create index语句关键词错误
106	Syntax Error! Insert Key Word Error!	insert语句关键词错误
107	Syntax Error! Extra parameters in insert!	insert语句中有多余参数
108	Syntax Error! Lack of parameters in insert.	insert语句缺少参数
109	Syntax Error! No executable SQL command!	输入为空，没有可执行的SQL语句
110	Syntax Error! Extra parameters in drop table!	drop table 参数多余
111	Syntax Error! Extra parameters in drop index!	drop index 参数多余
112	Syntax Error! No such drop operation!	drop语句关键词错误
113	Syntax Error! No such delete operation!	delete语句关键词错误，没有from
114	Syntax Error! Lack of parameters.	SQL语句缺少参数
115	Syntax Error! Key word Error!	delete语句关键词错误，缺少where
116	Syntax Error! Lack of parameters.	select语句缺少参数
117	Syntax Error! No such select operation!	select的对象错误，只能是*
118	Syntax Error! Key Word Error!	select语句关键词错误，没有from或条件查询中缺少where
119	Error! Can't find the file on the path! Please try again!	打开文件执行SQL语句的命令失败
120	Syntax error! Can't create table! Illegal create command!	create table在**语法分解时**发现语句的语法错误
121	Syntax error! Can't create table! No defined attribution!	创建表的时候没有定义属性

错误状态码	错误信息	具体含义
122	Syntax error! Can't create table! More than 1 primary keys!	创建表的时候定义了超过一个主键
123	Syntax error! Can't create table! No primary keys!	创建表的时候没有定义主键
124	Syntax error! Can't create table! No matching primary key!	创建表的时候定义的主键和属性不匹配
125	Syntax error! Illegal data type!	数据类型非法，不是int，flaot和char中的一种

3.1.2 运行时错误 **Run time error**

- 主要包含可以通过语法检查，但是在调用具体的模块时导致的各类错误，具体内容如下表所示

错误状态码	错误信息	具体含义
200	Run time error! Illegal logic type!	不存在的逻辑关系类型
201	Run time error! No logic operator!	语句中不存在逻辑运算符
202	Run time error! Not a(n) xxx value in the SQL	对应的属性值类型不匹配
203	Run time error! Can't open catalog!	无法打开catalog日志文件
204	Run time error! Can't open index table!	不能打开index table文件
205	Run time error! The table has already exist.	要创建的表名早已存在
206	Run time error! Too many attributions!	属性超过32条
207	Run time error! Illegal data type!	数据类型错误
208	Run time error! Char length is too large!	char*类型字符串长度超过255
209	Run time error! Duplicate attribution names!	有重复的属性名
210	Run time error! Can't create non-primary index!	不能在非主键上建立索引
211	Run time error! No primary key when create table!	创建表时没有主键
212	Run time error! Too many primary keys when create table!	创建表时发现主键超过一个
213	Run time error! Fail to open table file!	无法打开数据表信息文件
214	Run time error! Index already exists!	要创建的索引已经存在
215	Run time error! Table doesn't exist!	找不到带创建索引所在的表
216	Run time error! Attribution doesn't exist!	找不到待创建索引对应的属性
217	Run time error! Attribution is not unique!	待创建索引的属性不是unique的
218	Run time error! Redundant index!	同名索引已经存在
219	Run time error! No index with such index name!	找不到对应name的index
220	Run time error! No index to drop!	需要drop的索引不存在
221	Run time error! No index in the table column!	该表没有创建索引
222	Run time error! Fail to delete table file!	删除数据表信息失败
223	Run time error! Fail to open the table file!	打开数据信息表失败

3.2 字符串处理库`stringProcessor`实现

- 设计思想
  - 本系统通过字符界面输入字符串形式的SQL指令来完成对应的操作，因此对于字符串的处理非常重要，我们实现了一个字符串处理的类stringProcessor来帮助我们进行字符串的相关处理

    - 该类主要在Interpreter层的语义分解中调用，在其它模块也会有少量的调用
- 该类对外提供如下字符串处理函数，包含如下功能
    - 字符串按条件分割
    - 关系符提取
    - 去首尾空格
    - 判断是否为int，float和char类型

```
class stringProcessor {
public:
    //去空格的处理函数，其实应该还需要一个把多个空格替换成一个空格的函数，不过这里先考虑最简单的情况先不写了
    static void preTrim(string &s);
    //把字符串按某个符号进行切分的处理函数
    //s 是要处理的长字符串，unit是切分的标志
    static vector<string> Split(const string &s, const string &unit);
    // 对cmd命令的优化，包括去首尾空格和全部转换成小写，卸载这个类里提高了代码的复用性
    static string cmdOptimum(string& cmd);
    static void showOperation(vector<string> op);
    //用来删除命令中的括号
    //返回值说明：0表示语法错误，1表示有括号，是create或者insert语句，-1表示没有括号，是其他类型的SQL语句
    static int bracketProcessor(string& cmd);
    static int getCompareType(const string& x);
    static logicCompare* getLogic(string x, int compareType, vector<dataType*>* attribution);
    static pair<bool, int> intCheckAndChange(const string& val);
    static pair<bool, float> floatCheckAndChange(const string& val);
    static bool charCheckAndChange(string& val);
    static bool charCheck(string& val);
};
```

3.3 核心数据结构实现

3.3.1 tableValue 数据结构

- 该结构主要使用一个struct包含了miniSQL所需的三种数据类型，即`int`，`float`和`char*`使得该结构体可以存储数据表中任意一格的值，在取用过程中，先获取该处的数据类型，再从结构体里读出对应的数据即可使用，这样的设计使得tableValue这个结构成为了一个较为通用的数据结构

```
struct tableValue{
public:
    int INT;
    float FLOAT;
    char* CHAR;
};
```

3.3.2 dataType 数据结构

- dataType定义了miniSQL中数据表的一个属性的具体信息，包括类型，属性名，属性的长度，是否唯一，是否为主键，是否有索引等信息，作为接下来的table数据结构的一个子模块

```
class dataType{
public:
    int type, n;
    string typeName;
    bool isUnique, isPrimaryKey, hasIndex;
    dataType(int input_type, int input_n, string input_typeName, bool
input_isUnique, bool input_isPrimaryKey, bool input_hasIndex);
    //获取数据长度的方法
    int getDataLength();
    // 这是一个可以看data type内部情况的成员函数，用于在测试的时候使用
    void showDataType();
};
```

3.3.3 Table 数据结构

- 定义了miniSQL中一张数据表的结构，包括表名，属性个数和种类，属性对应的变量类型，主键对应的属性名，表的长度，是否有索引等信息，并且用STL模板存储了该表的每一个属性名和索引的信息

```
class Table{
public:
    string tableName;
    //这里primaryKey表示对应的主键的属性的下标
    int columnCount, primaryKey, numPerBlock, size;
    vector<dataType*>* tableAttribution;
    vector<string>* indexAttribution;
    Table(string name, vector<dataType*>* attribute, vector<string>*
index);
    //这里好像也可以改成引用
    dataType* searchAttribution(const string& name);
    dataType* searchAttribution(const char* name);
    int searchPosition(const string& name);
    int searchPosition(const char* name);
    void dropIndex(const string& indexName);
};
```

3.3.4 Block 数据结构

- 用来表示一个缓冲区区块的数据结构，包含了一个区块的基本信息，包括文件名，区块ID，保存的数据和是否被上锁和进行修改

```
class Block{
public:
    string fileName;
    int blockID;
    //pin表示一个block被锁了，不能被删除
    //is changed代表这个block是否被进行了修改
    bool pin, isChanged;
    char data[blockSize];
    Block(const string& name, const int& id){
        fileName = name;
        blockID = id;
```

```
        pin = false;
        isChanged = false;
    }
};
```

- 若干个block之间采用链表的方式进行链接，链表的定义如下所示

```
struct bufferNode{
   Block* block;
    bufferNode *last, *next;
    explicit bufferNode(Block* temp){
        block = temp;
        last = next = nullptr;
    }
    ~bufferNode(){
        remove();
    }
    void remove() const{
        if(next != nullptr)
            last->next = next;
        if(last != nullptr)
            next->last = last;
    }
};
```

3.3.5 logicCompare 数据结构

- 用于存储select语句和delete语句中的where后面的信息的数据结构，存储了需要比较的属性名，比较符号的种类和待比较的值等信息，并定义了三个进行大小比较的成员函数

```
#define EQUAL 0
#define NOTEQUAL 1
#define LESS 2
#define GREATER 3
#define LESSEQUAL 4
#define GREATEREQUAL 5

class logicCompare{
private:
    string valName;
    tableValue immediate;
public:
    string getValueName();
    tableValue getImmediate();
    bool checkCondition(int result);
    logicCompare(string& name, int op, tableValue imm);
    static int compareInt(int a, int b);
    static int compareFloat(float a, float b);
    static int compareChar(const char* a, const char*b, int length);

    int operation;
};
```

3.4 `Interpreter` 与 `API` 实现

- Interpreter层的主要功能是负责与用户之间的交互，包括输入的SQL语句的处理和处理结果的呈现，Interpreter支持的SQL语句的种类前面已经讲到过了，其总体工作流程如下
  - 语句读入，以分号为标志读入一句完整的SQL指令
  - 一级语法分解，提取SQL语句中的**操作关键字**，对输入的SQL语句进行初步的分类
  - 二级语法分解，根据分类后的SQL语句提取SQL语句中的表名，属性名，数据值等关键值，并按照规定的数据结构存储，将其作为参数传递给callAPI模块，来调用对应的API
  - 语法检查和报错，在全过程中对于不符合规范的SQL语句进行语法检查，并在遇到错误时进行对应的报错
- 下面着重介绍一下**语法分解**的部分
  - 首先寻找SQL语句的标志词，比如insert，create，delete等，在找到对应的关键词后所进行的处理如下
    - `quit`：直接退出miniSQL系统
    - `execfile`：再读取文件名，调用interpreter中的`exeFile`函数打开文件进行读取
    - `create`：读取需要create的内容，如果是table则读取表名和后面定义的各个属性名，如果是index则读取索引名，表名和列明
      - `create table` 先去掉括号，对读取到的属性名定义按照逗号进行分解和去空格，并整合成一个dataType的vector
      - `create index` 比较简单，不需要进行复杂的二次分解，直接读取索引名，表名和列名即可
    - `insert`：读取表名和values后面的属性值，将属性值按照逗号进行分割，拆分成一个tableValue的vector
    - `delete`：读取表名和where后面的条件，将条件以and为分割依据，分解为一系列的logicCompare
    - `drop`：读取需要drop的内容，如果是table就读取表名，是index则读取索引名
    - `select`：先读取表名，然后读取where后面的查询条件，以and为标志划分成一系列的logicCompare
  - 以上各部分均包含了对于SQL语句的关键词的检查，如果关键词错误，或者SQL语句中的参数缺少或者过多，都会引发报错并退出当此循环。开始读取下一句SQL语句
  - 第一层语法分解主要在interpreter中进行，而对于SQL语句，第二层的语法分解在callAPI中进行，callAPI的类定义了如下若干静态成员函数

```
class callAPI {
public:
    // 这两个分别调用drop表和索引的API，比较容易写
    static bool callDropTableAPI(const string& tableName);
    static bool callDropIndexAPI(const string& indexName);
    static bool callCreateIndexAPI(const string& index, const string&
table, const string& attribute);
    static bool callInsertAPI(const string& table, string& value);
    static bool callCreateTableAPI(const string& table, string&
element);
    static vector<vector<tableValue>*>* callSelectAPI(string& table,
string& condition);
    // 调用delete API的静态成员函数，condition是where后面的全部内容
    // 1表示成功，0表示失败，-1表示异常
    static int callDeleteAPI(string& tableName, string& condition);
};
```

- 在两层的语法分割结束后，将得到的内容作为参数来调用对应的API，并将结果通过`showResult`类进行呈现

```
class showResult {
public:
    // 一些用来显示结果的static成员函数
    static void showDropTable(bool flag, const string& table);
    static void showDropIndex(bool flag, const string& index);
    static void showCreateTable(bool flag, const string& table);
    static void showCreateIndex(bool flag, const string& index);
    static void showInsert(bool flag, const string& table);
    static void showDelete(int flag, const string& table);
    static void showSelect(vector<vector<tableValue>*>* result, string
tableName);
};
```

- API层提供如下接口，**根据不同的操作**来调用catalog，index和buffer模块中的不同成员函数完成对应的操作，具体的会在各个Manager的模块中详细介绍

```
class API {
public:
    API();
    ~API();
    bool createTable(const string& tableName, vector<dataType*>*
attribution);
    bool createIndex(string indexName, string tableName, string
attribution);
    bool dropTable(const string& tableName);
    bool dropIndex(const string& indexName);

    int deleteValue(const string& tableName, vector<logicCompare>*
conditions);
    vector<vector<tableValue>*>* select(const string& tableName,
vector<logicCompare>* condtions);
    Table* getTable(const string& tableName);
    int remove(string tableName, vector<logicCompare>* conditions);
    bool insertValue(const string& tableName, vector<string> valueList);

private:
    bufferManager* buffer;
    indexManager* Index;
    catalogManager* catalog;
    recordManager* record;

    int findRecord(const string& tableName, vector<logicCompare>*
conditions, vector<vector<tableValue>*>* results, vector<int>* ids);
    void writeKey(dataType* attribution, char* key, tableValue v);
};
```

3.5 `CatalogManager` 接口

- 设计思想

- `CatalogManager` 负责管理数据库中各个表和索引的信息，用**若干文本文件**来记录数据库中的所有表的表名，所有索引的信息，以及每一张表的结构
- 对于增删查改等操作，API可以调用catalogManager接口来对日志文件进行相应的修改

- `Catalog Manager`支持如下的功能

```
class catalogManager {
private:
    static set<string> tableNameList;
    static map<string, index*> indexMap;
public:
    //每种操作对应的对于数据表和索引信息的文件的更新操作
    //用两个文件分别存储所有表和索引的名称等信息，用若干个文件每个分别存储一个表中的所有信息和一个索引的信息
    catalogManager();
    //~catalogManager();
    Table* getTable(const string& tableName);
    bool catalogCreateTable(const string& tableName, vector<dataType*>* attributions);
    bool catalogDropTable(const string& tableName);
    bool catalogUpdateTable(Table* table);
    bool catalogDropIndex(const string& indexName);
    bool catalogDropIndex(const string& tableName, const string& columnName);
    bool catalogCreateIndex(string& indexName, string& tableName, string& columnName);
    index* getIndex(const string& indexName);
    index* getIndex(const string& tableName, const string& columnName);
    void getIndexbyTable(const string& tableName, vector<index*>* indexVector);
};
```

3.6 `Record Manager` 接口

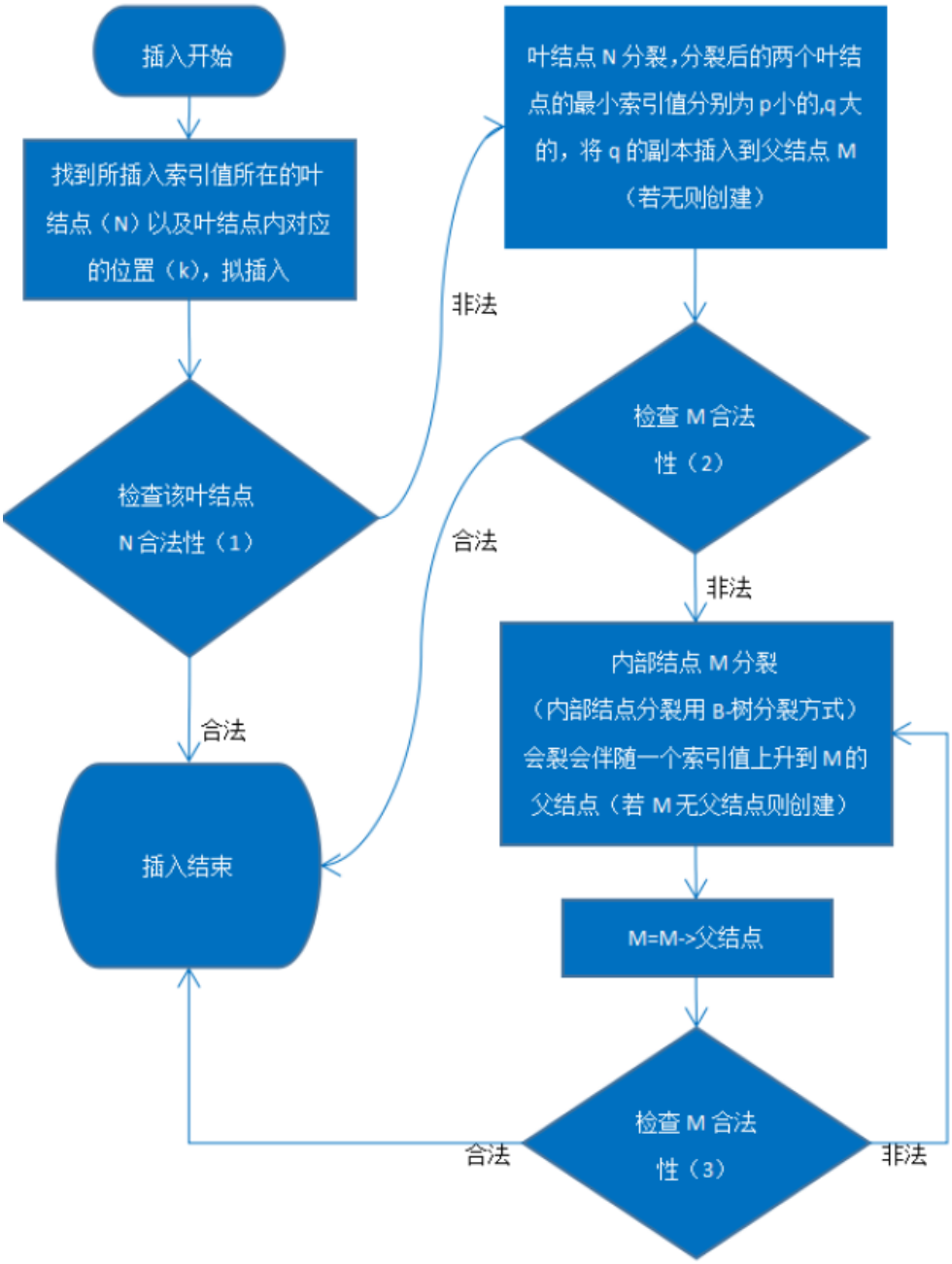
- 设计思想与具体实现
  - `Record Manager` 负责管理记录表中数据的数据文件，实现最终对文件内记录的增删查改四种操作，其总体设计思想如下：
    - 单表存储：一张表上的所有记录被存放在一个文件中
    - 顺序存储：文件中的数据内容按照线性顺序进行存储，方便增删查改
    - 文件读写：使用BufferMananger对文件块进行读写，用缓存的方式减小文件I/O所需要的时间

`RecordManager` 对外提供如下接口：

```
bool recordCreateTable(const string& tableName);
bool recordDropTable(const string& tableName);
bool recordDeleteTable(const string& tableName, vector<int>* list);
int recordInsertTable(const string& tableName, vector<tableValue>* value);
bool recordCheck(Table* table, vector<tableValue>* record,
vector<logicCompare>* conditions);
bool recordCheckDuplicate(const string& tableName, vector<tableValue>*
record);
vector<int> * recordSelectTable(const string& tableName,
vector<logicCompare>* conditions);
vector<tableValue>* recordGetByID(const string& tableName, int id);
```

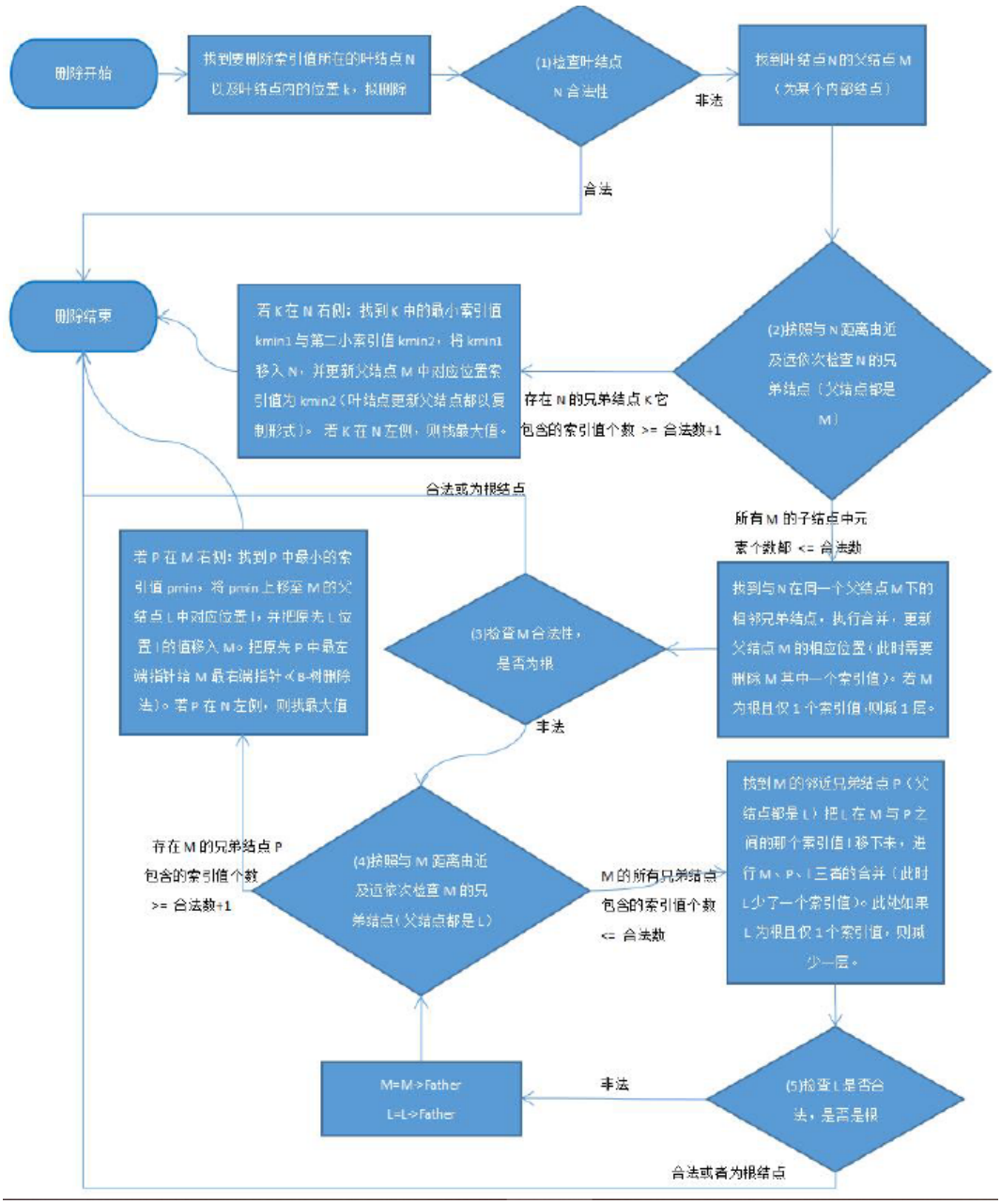
3.7 `IndexManager` 接口

- 核心数据结构：`B+树`
  - B+树的定义：对于一个M阶的B+树，每个B+树节点有最多M个键值和M+1个指针，分为两种情况
    - 叶节点：最底层的节点，若干叶节点将待索引的值按递增的顺序存放，一个叶节点中至少有M/2个待索引的值
    - 内部节点：通过键值进行索引，通过指针指向B+树的下一层
    - 这样的结构，对于规模为N的问题，可以实现时间复杂度为$\log N$的插入和查询
  - B+树的查找操作
    - 从根节点开始，先从节点内部查找，根据待查找的值和键之间的大小关系进行选择，如果待查找的值小于当前的键值，则顺着键值左边的指针向下查找，否则顺着右边的指针向下查找，知道找到叶节点为止
    - 如果叶节点中存在待查找的值，就说明查找成功，否则查找失败
  - B+树的插入操作
    - B+树插入操作可以用如下流程图来表示



- 总结起来就是满了就要分裂成两个，插入之后需要向上修改键值

- B+树的删除操作
  - B+树的删除比插入更加复杂，可以用如下流程图来表示



- 设计思想
  - IndexManager 负责管理数据库的索引，利用B+树的log N级别的查询速度来优化`select`和`delete`语句的执行效率，采用了哈希存储，地址映射，二次筛选和文件读写等设计思想，具体的由`index`类，`index manager`类，`BPTree`类和`BPTreeNode`类构成
  - index类定义了一个索引的基本结构供各模块调用，其定义如下

```
class index {
public:
  index(string& self, string& table, string& column);
    string getName() const;
  string getTableName() const;
    string getColumnName() const;

private:
    string name, tableName, columnName;
};
```

- Index Manager提供建立，更新，删除索引等相关操作，其核心是B+树的插入与删除，该接口暴露出`IndexManager`作为接口，通过`BPTree.h`来构建B+树索引，而BPTree的基本组成单位是`BPTreeNode.h`
- IndexManager的类定义如下

```
//索引管理，包括基本的查找、插入、删除、创建等操作
class indexManager {
public:
    indexManager();
    int find(basic_string<char> indexName, const char* key);
    bool insert(const char* indexName, const char* key, int value);
    bool remove(const char* indexName, const char* key);
    bool createIndex(const char* indexName);
    bool dropIndex(const char* indexName);
private:
    bufferManager *buffer;
};
```

- 而其中最关键的BPTreeNode和BPTree的定义如下

```
class BPTree {
public:

    static void createFile(const char* _filename, int _keyLength, int
_dataType, int _fan_out = -1);

    BPTree(const char* _filename);

    ~BPTree();

    int find(const char* _key);

    bool add(const char* _key, int _value);

    bool remove(const char* _key);

private:
    int fan_out;
    int keyLength;
    int dataType;
    int nodeCount;
    int root;
    int firstEmpty;
    string filename;

    // Key-value to deal with
    char* key;
    int value;

    int find(int id);
    int add(int id);
    int remove(int id, int sibId, bool leftSib, const char* parentKey);
    int getFirstEmpty();
```

```
    void removeBlock(int id);
    void updateHeader();
};
```

```
class BPTreeNode {
public:
    // from file
    BPTreeNode(const char* _filename, int _id, int _keyLength, int
_dataType);
    // empty
    BPTreeNode(const char* _filename, int _id, int _keyLength, int
_dataType, bool _leaf, int firstPtr);

    ~BPTreeNode();

    int getSize() const;
    int getKeyLength() const;
    bool isLeaf() const;
    const char* getKey(int pos) const;
    int getPtr(int pos) const;
    int findPos(const char* key);
    void placeKey(int pos, const char* key);
    void placePtr(int pos, int ptr);
    void setRemoved();
    void insert(int pos, const char* key, int ptr);
    void remove(int pos);
    BPTreeNode* split(int newId, char* newKey);
    const char* borrowFromSib(BPTreeNode* sib, bool isLeft, const char*
parentKey);
    void mergeRight(BPTreeNode* sib, const char* parentKey);

private:
    string filename;
    int id;
    int size;
    int keyLength;
    int dataType;
    bool leaf;
    bool dirty;
    bool removed;
    vector<char*> keys;
    vector<int> ptrs;
    int nodecmp(const char* a,const  char* b)
};
```

3.8 `Buffer Manager`模块实现

- 设计思想：Buffer Manager通过与磁盘文件进行数据交换来实现缓存，在本系统中为了提高文件系统的I/O效率，我们规定了缓冲区的一个block的大小为4KB，同时缓冲区中最多可以容纳50个block，其核心设计包括
    - 从磁盘中读写文件，通过File Mananger实现文件的读写
    - 通过链表来维护各个block，按照访问的最新时间排序，即最近访问的block越靠前，缓冲区替换采用LRU算法，每次将最不常访问的block删除，插入新的区块
- 其核心的类定义和结构设计如下

- block的设计和其重要参数

```
class Block{
public:
    string fileName;
    int blockID;
    //pin表示一个block被锁了，不能被删除
    //is changed代表这个block是否被进行了修改
    bool pin, isChanged;
    char data[blockSize];
    Block(const string& name, const int& id){
        fileName = name;
        blockID = id;
        pin = false;
        isChanged = false;
    }
};

struct bufferNode{
   Block* block;
    bufferNode *last, *next;
    explicit bufferNode(Block* temp){
        block = temp;
        last = next = nullptr;
    }
    ~bufferNode(){
        remove();
    }
    void remove() const;
};
```

- Buffer Manager的设计与具体实现

```
class bufferManager {
private:
    static int blockCount;
    static bufferNode *head, *tail;
    static map<string, bufferNode*> nodeMap;
    Block* loadBlock(const string& fileName, int id);
    bool deleteNode(bufferNode *node);
public:
    static const int maxBlockNumber;
    bufferManager();
    Block* getBlock(const string& fileName, int id);
    bool writeBlock(const string& fileName, int id);
    bool writeBlock(Block* block);
    bool accessNode(bufferNode* node);
    bool appendNode(const string& fileName);
    bool clearBuffer();
};
```

四、系统测试

- 本系统的集成与测试耗费了组员们大量时间，目前本系统已经彻底完成，以下是运行的结果

4.1 语法错误测试

1. create table没有属性

```
miniSQL>create table minisql(
       >);
Syntax error! Can't create table! No defined attribution!
Failed when creating table minisql. Please try again!
Finish the operation in 0.003 seconds
```

2. create table 缺少主键

```
miniSQL>create table minisql(
       >name int,
       >id float
       >);
Syntax error! Can't create table! No primary keys!
Failed when creating table minisql. Please try again!
Finish the operation in 0.003 seconds
```

3. create index 参数错误

```
miniSQL>create index xxx on table test(id);
Syntax Error! Extra parameters in create index!
miniSQL>create index xxx test(id);
Syntax Error! Lack of parameters in create index!
```

4. 在不存在的表上建立索引

```
miniSQL>create index xxx on yyy(zzz);
1
Run time error! Table doesn't exist!
Failed when creating table yyy. Please try again!
Finish the operation in 0.002 seconds
```

5. select语句参数错误和待查的表不存在

```
miniSQL>select id from test;
Syntax Error! No such select operation!
miniSQL>select * from table test;
Syntax Error! Key Word Error!
miniSQL>select * from test where id=='100';
Run time error! Table doesn't exist!
Run time error! Don't find such table!
No selected result at all!
Finish the operation in 0.003 seconds
```

6. delete语句参数错误

```
miniSQL>delete from table test where num==1;
Syntax Error! Key word Error!
miniSQL>delete table test num==1;
Syntax Error! No such delete operation!
miniSQL>delete from table test;
Syntax Error! Lack of parameters.
```

7. delete不存在的数据或表

```
miniSQL>delete from test where num==1;
test
num==1
Run time error! Table doesn't exist!
Run time error! Table doesn't exist!
Finish the operation in 0.002 seconds
```

8. drop table和index时对应的内容不存在

```
miniSQL>drop table ads;
Run time error! Table doesn't exist!
```

```
miniSQL>drop index fsesb;
Run time error! No index with such index name!
Failed in dropping index fsesb. Please try again!
Finish the operation in 0.002 seconds
```

4.2 SQL语句运行测试

1. 退出系统

```
miniSQL>quit;
Thank you for using such a sb system, see you next time!
----------------------------------------
请按任意键继续. . .


Process finished with exit code 0
```

2. 成功创建一张表

```
miniSQL>create table test(
       >num int,
       >id char(10),
       >primary key(id)
       >);
----------show data type-----------
type: 3772
n: 0
typename: num
unique: 0
primary: 0
index: 0
Create Table test Successfully!
Finish the operation in 0.043 seconds
```

3. 向表中插入数据

```
miniSQL>insert into test values(1,'100');
Insert into table test Successfully!
Finish the operation in 0.032 seconds
miniSQL>insert into test values(2,'123');
Insert into table test Successfully!
Finish the operation in 0.034 seconds
```

4. 使用**select**语句查询

```
miniSQL>select * from test;
--------------------------
|num          |id         |
--------------------------
|1            |100        |
--------------------------
|2            |123        |
--------------------------
|6            |343        |
--------------------------
|7            |213        |
--------------------------
```

5. select 条件查询

```
miniSQL>select * from test where num > 5;
--------------------------
|num          |id         |
--------------------------
|6            |343        |
--------------------------
|7            |213        |
--------------------------

Finish the operation in 0.007 seconds
```

6. select多重条件查询

```
miniSQL>select * from test where num > 5 and num < 7;
--------------------------
|num          |id         |
--------------------------
|6            |343        |
--------------------------

Finish the operation in 0.005 seconds
```

7. 删除表中的值

```
miniSQL>delete from test where id='100';
test
id='100'
Delete from table test Successfully!
Finish the operation in 0.033 seconds
miniSQL>select * from test;
--------------------------
|num          |id         |
--------------------------
|2            |123        |
--------------------------
|6            |343        |
--------------------------
|7            |213        |
--------------------------

Finish the operation in 0.007 seconds
```

8. 删除表

```
miniSQL>drop table test;
Drop Table test Successfully!
Finish the operation in 0.003 seconds
```

9. 对于**unique**类型建立索引

- 需要先创建一张新的表并插入一些值

```
miniSQL>create table test2(
       >num int unique,
       >price float,
       >id char(10),
       >primary key(id)
       >);
----------show data type-----------
type: 3772
n: 0
typename: num
unique: 1
primary: 0
index: 0
Create Table test2 Successfully!
Finish the operation in 0.032 seconds
```

```
miniSQL>insert into test2 values(1, 2.3, '100');
Insert into table test2 Successfully!
Finish the operation in 0.035 seconds
miniSQL>insert into test2 values(2, 3.4, '2234');
Insert into table test2 Successfully!
Finish the operation in 0.036 seconds
miniSQL>insert into test2 values(3, 5.6, '43546');
Insert into table test2 Successfully!
Finish the operation in 0.03 seconds
miniSQL>insert into test2 values(7, 4.5, '90909090');
Insert into table test2 Successfully!
Finish the operation in 0.025 seconds
```

- 在`unique`类型的`num`上建立索引

```
miniSQL>create index zyc on test2(num);
1
Create Index on test2 Successfully!
Finish the operation in 0.065 seconds
```

10. 删除索引

```
miniSQL>drop index zyc;
Drop Index zyc Successfully!
Finish the operation in 0.004 seconds
```

11. or的条件查询

```
miniSQL>select * from student2;
-----------------------------------------
|id          |name        |score          |
-----------------------------------------
|1080100001  |name1       |99             |
-----------------------------------------
|1080100002  |name2       |52.5           |
-----------------------------------------
|1080100006  |name6       |89.5           |
-----------------------------------------
|1080106012  |name6012    |89.5           |
-----------------------------------------
```

```
miniSQL>select * from student2 where score=89.5 or name="name2";
-----------------------------------------
|id          |name        |score          |
-----------------------------------------
|1080100002  |name2       |52.5           |
-----------------------------------------
|1080100006  |name6       |89.5           |
-----------------------------------------
|1080106012  |name6012    |89.5           |
-----------------------------------------
```

五、其他说明

5.1 项目GitHub地址

- 本项目采用GitHub作为代码托管工具，并使用git进行代码版本的控制，待验收结束之后将把项目开源，目前暂时不开放

5.2 项目日志

- 本项目采用了开发日志进行项目管理，每个人每天修改和添加代码之后都需要在开发日志中进行对应的说明以方便其他小组成员了解项目的开发进展，开发日志在GitHub的项目上存放，待开源之后就会公布