

# MiniSQL 设计总报告

## 人 员

夏 涵	3140105252
胡 凡	3140104011
谢 昂	3150102207
孟林昊	3140104948

## 目录

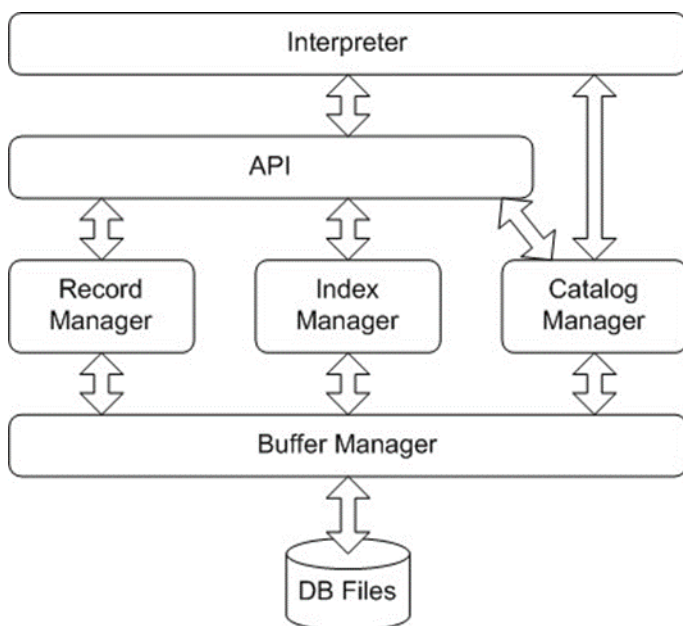
MINISQL 设计总报告 .....	1
第 1 章 MiniSQL 总体框架 .....	2
第 2 章 MiniSQL 各模块实现功能.....	3
第 3 章 各模块外部接口及实现技术 .....	4
第 4 章 MINISQL 系统测试 .....	19
第 5 章 分工说明 .....	24

## 第 1 章 MiniSQL 总体框架

### 第 1.1 节 MiniSQL 实现功能分析

- 1) 总功能：允许用户通过命令行界面输入 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  $1 \leq N \leq 255$ ；
- 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）；
- 5) 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；
- 6) 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



### 第 1.3 节 设计语言与运行环境

工具：C++      环境：Windows/Mac OS

## 第 2 章 MiniSQL 各模块实现功能

### 第 2.1 节 Interpreter 实现功能

Interpreter 负责将外界 SQL 语句转换为相应的数据信息，传递给 API 中的相应函数。并对一些错误语法进行分析，提示报错。

### 第 2.2 节 API 实现功能

API 模块是整个系统的核心，其主要功能是根据 Interpreter 层解释生成的命令内部形式，调用 Catalog Manager 提供的信息进行进一步的验证及确定执行规则，并调用 Record Manager、Index Manager 和 Catalog Manager 提供的相应接口执行各 SQL 语句及命令语句。概而言之，API 是 Interpreter 模块，Record Manager 模块及 Catalog Manager 模块进行沟通的枢纽，三者之间的交互一般都由 API 模块来进行转接才能进行相互之间的功能调用。因此，虽然 API 模块所实现的功能十分有限，但是该模块却是整个系统的核心。

### 第 2.3 节 Catalog Manager 实现功能

Catalog Manager 负责管理数据库各种模式信息

1. 数据库中所有表的信息：表名（文件名），主键，记录数，属性数，属性名。每个表维护一个“表名.catalog”文件。
2. 数据库中所有索引的信息：索引名，所在表名，所在属性名，属性类型。所有索引信息维护在一个“index.catalog”的文件中。
3. 同时，在我们的程序中，每一个 attribute 是一个类对象，其中包含 indexname 的变量，用来记录建立在该属性上的 indexname，Catalog Manager 也负责维护。

### 第 2.4 节 Record Manager 实现功能

Record Manager 负责处理表的实际文件创建、实际文件删除、数据的插入、查找以及删除操作。Record Manager 模块从 API 模块、Catalog 模块处接收信息，必要时通过 API 调用 Index 模块用于实现数据的更新、查找，并对外提供相应的接口。

为简单起见，这里只实现了定长记录的存储，且不要求记录的跨块存储。一个块中包含一条至多条记录，每条记录前面有一位的标记位来标记该记录是否被删除。

### 第 2.5 节 Index Manager 实现功能

MiniSQL 的整体设计要求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）。

Index Manager 负责 B+树索引的实现，实现 B+树的创建和删除（由索引的创建与删除引起）、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+树中节点大小与缓冲区的块大小相同，B+树的度

数由块大小与索引键大小计算得到。)

## 第 2.6 节 Buffer Manager 实现功能

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

## 第 2.7 节 DB Files 实现功能

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。

# 第 3 章 各模块外部接口及实现技术

## 第 3.1 节 Interpreter 接口

### 一、外部接口

//main 函数调用此函数，传递 SQL 语句进行语法分析

```
int mainInterpreter(string s);
```

### 二、实现技术

#### 1. 字符串提取：

相关函数：string Interpreter::getNextString(string s, int \*pos)

采用 C 风格字符串分割方式。每次提取首先都是将下一个字符串前面的所有空格通过一个 while 循环忽略：

```
/* 初始位置移到没有空格为止 */
while ((s[*pos]==' ' || s[*pos]=='\n' || s[*pos]=='\t' || s[*pos]=='\r' || s[*pos]==0) && s[*pos]!='\0'){
    (*pos)++;
}
beginPos=*pos;
```

之后扫描是否下一个字符串是符号，如果是则只提取这一个符号并返回，这样可以分析语法错误。

最后再是从第一个不是符号，且不是空格的位置开始，一直向后读取不是空格，不是字符且不为空的位置，返回。

#### 2. 语义分析：

相关函数：int Interpreter::mainInterpreter(string s)

按照每次返回的单词，类似树形结构进行判断使用哪一个功能模块，并且使用相应的变量存储信息传递给 API。

## 第 3.2 节 API 接口

### 一、外部接口

1. 建表, 传入属性及主键名,成功返回 true

```
bool CreateTable(string tablename,vector<Attribute> *tableattribute,string primarykeyname);
```

2. 删除表

```
bool DropTable(string tablename);
```

3. 创建索引(传入索引名、表名和列名)

```
void CreateIndex(string indexname,string tablename,string attributename);
```

4. 删除索引

```
void DropIndex(string indexname);
```

5. 插入一条记录,传入表名和属性值

```
void InsertRecord(string tablename,vector<string> *attributevalue);
```

6. 查找记录,若查找属性为\*, 传入 attributename 为 NULL,若无 where 选择条件,可不传入 conditionlist 参数;

```
void SelectRecord(string tablename,vector<string> *attributename=NULL, vector<Condition> *conditionlist=NULL);
```

7. 删除记录,若无 where 选择条件, 无需传入 conditionlist 参数

```
void DeleteRecord(string tablename,vector<Condition> *conditionlist=NULL);
```

8. 插入索引(每条记录,传入记录起始位置, 记录大小, 表所有属性, 记录所在块偏移量)

```
void InsertIndexInRecord(char *recordbegin,int recordsize,vector<Attribute> *allattribute, int blockoffset);
```

9. 插入索引(针对每个属性, 传入这个属性的索引名, 属性指针, 属性类型, 插入的单条记录所在块偏移量)

```
void InsertIndex(string indexname,char* attributebegin,int type,int blockoffset);
```

10. 删除索引

```
void deleteIndex(string indexFileName,char* attributebegin,int type);
```

11. 获取所有的 index

```
void getAllIndex(vector<IndexInfo>* indexlist);
```

12. 获取 table 属性保存在 attribute 中并返回一条记录的大小 (不包括标记位)

```
int GetAttribute(string tablename,vector<Attribute> *attribute);
```

13. 获取实际类型大小, type 对应 Attribute 中所给数值

```
int TypeSizeGet(int type);
```

## 二、实现技术

### 头文件说明：

```
class API{
public:
    CatalogManager *cm;
    RecordManager *rm;
    IndexManager *im;
public:
    API();
    ~API();
    ... (对外接口)
private:
    //获取 tableAttribute 中属性名为 attrname 的类型
    int getAttrType(vector<Attribute>* tableAttribute, string attrName);
    //将属性值从 vector<string>类型转化为 char*
    char* recordStringGet(string tableName, vector<string>* recordContent);
};
```

### 接口示例说明：

插入记录，首先对单值或索引属性是否重复进行判断是否可将记录插入，然后通过 RecordManager 进行记录的实际插入，通过 IndexManager 进行索引的插入，并调用 catalog 对记录数目进行更改。

```

void API::InsertRecord(string tablename, vector<string> *attributevalue){
    if (cm->findTable(tablename) != TABLE_FILE){
        cout << "Error: Table " << tablename << " doesnot exist!" << endl;
        return;
    }
    vector<Attribute> atmp;
    cm->getAttribute(tablename, &atmp);
    vector<string>::iterator iter2 = attributevalue->begin();
    for (vector<Attribute>::iterator iter1 = atmp.begin(); iter1 != atmp.end(); iter1++, iter2++){
        if (iter1->index != ""){//判断索引属性数据是否会重复
            int offset = im->searchIndex(iter1->index + ".index", *iter2, iter1->type);
            if (offset != -1){//找不到会返回-1
                cout << "Error: data duplication in unique attribute!" << endl;
                return;
            }
        }
        if (iter1->ifUnique == true){//判断单值属性是否会重复
            Condition condition(iter1->name, OP_EQUAL, *iter2);
            vector<Condition> conditionlist;
            conditionlist.insert(conditionlist.end(), condition);
            if (rm->FindRecord(tablename, &conditionlist)){
                cout << "Insert fails because unique value exists!" << endl;
                return;
            }
        }
    }
}

char* recordstring = recordStringGet(tablename, attributevalue);
if (recordstring == NULL) return;
int recordsize = cm->calculateSize(tablename);
int blockoffset = rm->InsertRecord(tablename, recordstring, recordsize);
if (blockoffset != -1){
    InsertIndexInRecord(recordstring, recordsize, &atmp, blockoffset);
    cm->insertRecord(tablename, 1);//更改catalog记录数目
}
delete[] recordstring;
}

```

## 第 3.3 节 Catalog Manager 接口

### 一、外部接口

#### 1. 创建一个新的表信息并填入信息

```
int addTable(string tableName, vector<Attribute> * tableAttribute, string primaryKeyName);
```

#### 2. 创建一个新的索引信息文件（若 index.catalog 文件没有建立），填入信息

```
int addIndex(string indexName, string tableName, string attributeName, int type);
```

#### 3. 判断该表信息文件是否存在

```
int findTable(string tableName);
```

#### 4. 判断该索引信息文件是否存在 返回值待定

```
int findIndex(string indexName);
```

#### 5. 删除一个表信息文件

```
int dropTable(string tableName);
```

#### 6. 删除一个索引信息

```
int dropIndex(string indexName);
```

#### 7. 有记录删除时，修改表信息文件中的记录总数

```
int removeRecord(string tableName,int removeNum);
```

#### 8. 有记录插入时，修改表信息文件中的记录总数

```
int insertRecord(string tableName,int insertNum);
```

#### 9. 获取表中的记录总数

```
int getRecordNum(string tableName);
```

#### 10. 获取索引数据类型信息

```
int getIndexType(string indexName);
```

#### 11. 获取表中所有属性信息

```
int getAttribute(string tableName,vector<Attribute>* tableAttribute);
```

#### 12. 获取一张表中所有索引信息

```
void getAllIndexInTable(string tableName, vector<string>& indexName);
```

#### 13. 计算返回表中一条记录占用的字节数

```
int calculateSize(string tableName);
```

#### 14. 计算返回不同数据类型所占字节数

```
int calculateSize(int type);
```

#### 15. 获取所有索引信息

```
void getAllIndexInfo(vector<IndexInfo>* indexList);
```

## 二、实现技术

### 1.文件的存储协议

```
/* 重新定义catalog文件中的数据存放结构:
   对于 “表名.catalog” 文件  int-insert的记录数  char[100]-主键名  int-属性的数量  char[100]-属性对应的index
                           char[100]-属性名  int-type  bool-ifUnique
   对于 “index.catalog” 文件  bool-isDelete  char[100]-indexName  char[100]-tableName  char[100]-Attribute  int-type
*/
```

规定了以上存储协议，每一条记录都是定长记录。

### 2.新建文件



```

string FileName = tablename + ".catalog"; // 定义表名称
ofstream f;
f.open(FileName.c_str(), ios::out); // 以输出方式打开表，如果没有则新建一个，如果有则清空
if (!f){
    return 0; // 操作失败，返回0
}
else{
    f.close(); // 直接关闭，接下来通过buffer写入数据
}

```

### 3. 读入文件的 block 通过指针写数据

```

FileNode *fn = bm.getFile(FileName.c_str()); // 调用getFile函数获得文件指针
BlockNode *bn = bm.getBlockHead(fn); // 调用getBlockHead函数获得该文件的首个块地址
char *strpos; // 用来指向拷贝字符串的位置

if (bn){
    char *beginAddress = bm.getContent(bn); // 首地址指针
    int *recordNum; // 指向 总记录数对应位置 指针
    int *attributeNum; // 指向 记录属性数量位置 指针
    int *type;
    bool *ifUnique;

    recordNum = (int*)beginAddress;
    *recordNum = 0; // 总记录数初始化为0
    beginAddress += sizeof(int); // beginAddress后移对应大小

    strpos = beginAddress;
    strcpy_s(strpos, STRING_SIZE, primaryKeyName.c_str());
    beginAddress += STRING_SIZE;

    attributeNum = (int*)beginAddress; // 属性数量
    *attributeNum = tableAttribute->size();
    beginAddress += sizeof(int);
}

```

## 第 3.4 节 Record Manager 接口

### 一、外部接口

#### 1. 建立表文件

bool Createtable(string tablename);

#### 2. 删除表文件

bool DropTable(string tablename);

#### 3. 插入一条记录，成功返回插入位置块偏移量，不成功返回-1

int InsertRecord(string tablename, char \*record, int recordsize);

#### 4. 查找具有特定值的记录，找到返回 1，没有返回 0

int FindRecord(string tablename, vector<Condition> \*condition);

#### 5. 查找满足条件的记录并输出到控制台，若没有 where 条件 conditionlist 为 null, offset 为块偏移量，若未使用索引，可不输入, 返回查找到的记录数量

```
int SelectRecord(string tablename,vector<string> *attributename,vector<Condition> *conditionlist,int
offset=-1);
```

## 6. 删除满足条件的记录，返回删除记录数量

```
int DeleteRecord(string tablename,vector<Condition> *conditionlist,int offset=-1);
```

## 7. 创建索引(只是为了把参数传给 indexmanager)

```
int IndexCreateFromRecord(string tablename,string indexname);
```

## 二、实现技术

### 1. 插入一条记录：

先查找是否块中有之前被删除的记录空位可以放，因为只支持定长记录的存储，所以若有空位一定能放下，若没有空位，则在所有记录末尾插入该记录，并返回插入位置的块偏移量。

```
int RecordManager::InsertRecord(string tablename,char *record,int recordsize){
    string tablefilename=tablename+".table";
    FileNode *ftmp=bm.getFile(tablefilename.c_str()); //得到文件结点
    BlockNode *btmp=bm.getBlockHead(ftmp); //得到文件结点内第一个块结点
    while(true){
        if(btmp == NULL){
            cout<<"insert "<<tablename<<" record fails"<<endl;
            return -1;
        }
        //如果有删除记录可以被替代，就将插入记录写入被删除记录的位置
        if(InsertRecordInBlock(btmp,record,recordsize)){
            return btmp->offset;
        }
        if(recordsize+1<=bm.getBlockSize()-bm.getUsingSize(btmp)){ //判断该块是否足够插入一条记录
            char *addr;
            addr=bm.getContent(btmp)+bm.getUsingSize(btmp); //块结点开始(不算第一个size_t)+已经使用的空间长度
            *addr=0; //标记位表示是否被删除
            addr+=1;
            memcpy(addr,record,recordsize); //拷贝记录内容
            bm.setUsingSize(btmp,bm.getUsingSize(btmp)+recordsize+1);
            bm.set_dirty(btmp); //标记为脏节点，要重新写入文件
            return btmp->offset;
        }
        else{
            btmp=bm.getNextBlock(ftmp,btmp); //跳到下一个块结点
        }
    }
    return -1;
}
```

### 2. 查找满足指定条件的记录：

若 offset>=0，说明通过索引查找已经定位到指定块，若未有索引，则只能从第一个块结点不断向后找直到最后一个块，通过 PrintRecord 函数从控制台输出满足条件的记录并记录数。

```

int RecordManager::SelectRecord(string tablename,vector<string> *attributename,vector<Condition> *conditionlist,int offset){
    string tablefilename=tablename+".table";
    FileNode *ftmp=bm.getFile(tablefilename.c_str());
    BlockNode *btmp=bm.getBlockHead(ftmp);

    int recordsize;
    vector<Attribute> allattribute;
    recordsize=api->GetAttribute(tablename,&allattribute);//获取一条记录大小以及全部属性

    if(offset>=0){
        btmp=bm.getBlockByOffset(ftmp,offset);//获取指定偏移量的块结点
        return SelectRecordInBlock(btmp,recordsize,&allattribute,attributename,conditionlist);
    }

    int count=0;//记录数
    while(true){
        if(btmp==NULL){
            cout<<"The records don't exist!"<<endl;
            return -1;
        }
        int cnttmp=SelectRecordInBlock(btmp,recordsize,&allattribute,attributename,conditionlist);
        count+=cnttmp;
        if(btmp->isBottom){
            return count;
        }
        btmp=bm.getNextBlock(ftmp,btmp);
    }
    return -1;
}

```

### 3. 删除记录：

若有索引，可直接定位到指定块，否则对每个块进行遍历，找到满足条件的要删除的指定记录，将其标记位置为 1，并通过 API 删除相应该记录的索引。

```

int RecordManager::DeleteRecord(string tablename,vector<Condition> *conditionlist,int offset){
    string tablefilename=tablename+".table";
    FileNode *ftmp=bm.getFile(tablefilename.c_str());//得到文件结点
    BlockNode *btmp=bm.getBlockHead(ftmp);//得到文件结点内第一个块结点
    int recordsize;
    int count=0; //记录满足删除条件的记录个数
    vector<Attribute> allattribute;
    recordsize=api->GetAttribute(tablename,&allattribute);//获取一条记录大小以及全部属性

    if(offset>=0){
        btmp=bm.getBlockByOffset(ftmp,offset);//获取指定偏移量的块结点
        count=DeleteRecordInBlock(btmp,recordsize,&allattribute,conditionlist);
        return count;
    }

    while(true){
        if(btmp==NULL){
            return -1;
        }
        count+=DeleteRecordInBlock(btmp,recordsize,&allattribute,conditionlist);
        if(btmp->isBottom){
            return count;
        }
        btmp=bm.getNextBlock(ftmp,btmp);
    }
    return -1;
}

```

```

int RecordManager::DeleteRecordInBlock(BlockNode *btmp, int recordsize, vector<Attribute> *allattribute, vector<Condition> *conditionlist){
    if(btmp==NULL){
        return 0;
    }
    char *recordbegin=btmp.getContent(btmp);
    char *blockend=recordbegin+btmp.getUsingSize(btmp);
    int count=0;
    while(recordbegin<blockend){
        char *attributebegin=recordbegin+1;
        if (*recordbegin == 0){
            if (ConditionFit(recordbegin + 1, recordsize, allattribute, conditionlist)){
                *recordbegin = 1; //1代表已被删除
                for (int i = 0; i < allattribute->size(); i++){
                    int type = (*allattribute)[i].type; //获得每个属性的类型
                    int typesize = api->TypeSizeGet(type); //获取该属性类型的实际大小
                    if ((*allattribute)[i].index != ""){
                        api->deleteIndex((*allattribute)[i].index + ".index", attributebegin, type); //若该属性值上有索引则删除
                    }
                    attributebegin += typesize;
                }
                btmp.set_dirty(btmp); //标记为脏节点，要重新写入文件
                count++;
            }
            recordbegin+=recordsize+1;
        }
    }
    return count;
}

```

## 第 3.5 节 Index Manager 接口

### 一、外部接口

#### 1. 创建索引

void createIndex(string filePath, int type);

根据索引文件名和键值类型，创建新的索引文件，并建立相应 B+ 树以及二者间的映射。

#### 2. 删除索引

void dropIndex(string filePath, int type);

根据索引文件名和键值类型，删除索引文件和对应 B+ 树。

#### 3. 等值查找

offsetNumber searchIndex(string filePath, string key, int type);

根据索引文件名、键值和类型，查找相应记录在块中的偏移量。

#### 4. 插入新索引值

void insertIndex(string filePath, string key, offsetNumber blockOffset, int type);

根据索引文件名、键值、块中偏移量和类型，在 B+ 树中插入新索引值。

#### 5. 删除索引值

void deleteIndex(string filePath, string key, int type);

根据索引文件名、键值和类型，在 B+ 树中删除对应索引值。

#### 6. 读取索引文件

void readIndex(string filePath, int type);

根据索引文件名和类型，读入索引文件，并建立相应 B+树以及二者间的映射。

## 二、实现技术

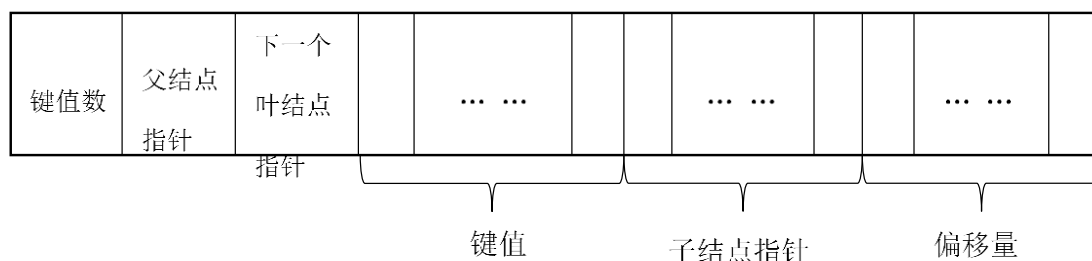
B+树并不建立实际的文件，而是在数据库运行后，从索引文件中读入索引信息，并建立对应的 B+树，当退出数据库时，B+树销毁，索引信息又从 B+树中写回文件中（实际是文件在内存中对应的块）。这样在文件块中只用存储键值和对于记录偏移量，而不用存储其他信息，节省了空间，而 B+树的度数也就由块大小（4KB）除以键值长度+偏移量长度（实际为整型，占 4B）算出。

索引文件块结构如下：



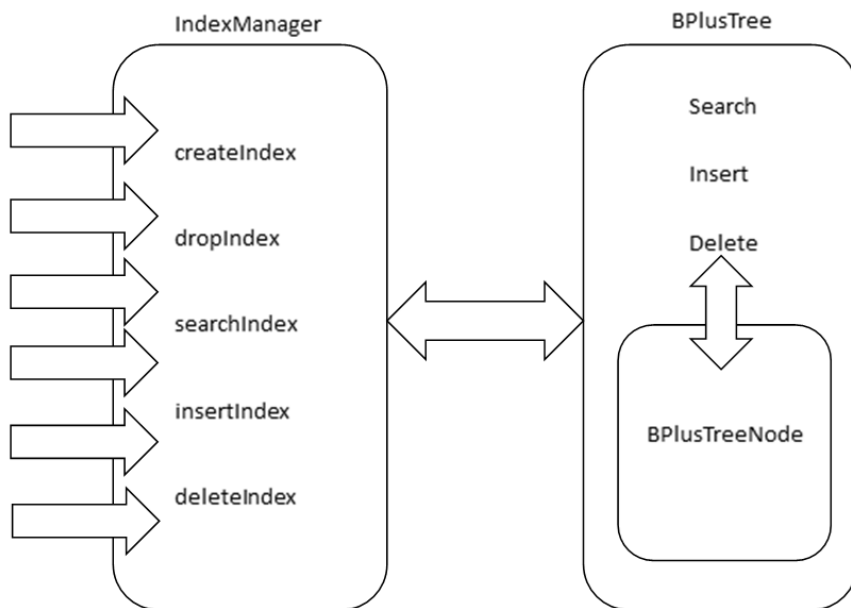
需要建立一棵 B+树，即 BPlusTree 类，它由 B+树结点构成。结点由 BPlusTreeNode 类实现，其中存储键值信息和一些指针。B+树中除了结点外，还需其他的必要信息（如树的根节点、结点数等）。Index Manager 的实现与 BPlusTree 是息息相关的，由 Index Manager 提出操作需求，BPlusTree 进行具体实施。

B+树结点结构如下：



设计类 IndexManager，由其提供对外接口。每次程序运行，IndexManager 从 API 中获得索引文件列表，并生成对应 B+树及建立二者映射。每次更新操作都在生成的 B+树上进行，当程序退出时，更新写回索引文件。

在 B+树内，设计一个 BPlusTreeNode 类，每个 Node 保存一个索引文件块中的信息，并通过一定的函数方法操作，新置、修改块内信息。B+树 中主要有构造、插入、等值查找和删除函数，以及插入和删除后的调整函数，而 BPlusTreeNode 类承担插入、查找、删除的具体操作。



BPlusTree 的结构：

```

template <class ElementType>
struct NodeInfo { // 结点信息类，用于记录结点信息，比结点类所占空间小
    TreeNode<ElementType>* pNode; // 指向对应结点的指针
    int index; // 需要的key在该节点中的位置
    bool exist; // 该节点中是否存在需要的key
};

template <class ElementType>
class BPlusTree { // B+树类
private:
    BufferManager buffer; // 缓冲区
    typedef TreeNode<ElementType>* Node; // 树节点指针定义为Node，方便书写

    string filename; // 对应的索引文件名
    Node root; // 树的根节点
    Node leafhead; // 第一个叶结点
    FileNode* file; // 树的文件结点
    int degree; // 树的度数，也是一块中能保存key的个数
    int keycount; // 树中key的个数
    int level; // 树的层数
    int nodecount; // 树的结点数
    int keysize; // key的大小

    ElementType FindMin(Node pNode); // 返回pNode所对应子树中最小的key
    bool AfterInsert(Node pNode); // 在pNode上执行完插入后的调整
    bool AfterDelete(Node pNode); // 在pNode上执行完删除后的调整
    void FindLeaf(Node pNode, ElementType key, NodeInfo<ElementType> &info); // 在pNode对应子树中查找key所在叶结点，并将其信息保存在info中

public:
    BPlusTree(string filename, int keysize, int degree);
    ~BPlusTree();
    void ReadTree(); // 从文件中读入key信息并生成对应的树
    void DropTree(Node pNode); // 删除pNode所对应子树
    void WriteBack(); // 将树中的key信息写回缓冲区

    offsetNumber Search(ElementType key); // 查找一个key，并返回其偏移量
    bool Insert(ElementType key, offsetNumber offset); // 在offset位置插入一个key
    bool Delete(ElementType key); // 删除一个key
};

```

BPlusTree 的插入函数：

```

template <class ElementType>
bool BPlusTree<ElementType>::Insert(ElementType key, offsetNumber offset) { //根据偏移量插入key
    NodeInfo<ElementType> newnode;
    if(!root) { //树为空，还没有根节点
        root = new TreeNode<ElementType>(degree, true);
        leafhead = root;
    }
    FindLeaf(root, key, newnode);
    if(newnode.exist) { //树中已存在该key
        cout << "Error: Cannot insert key to index: the duplicated key!" << endl;
        return false;
    }
    else {
        newnode.pNode->insertKey(key, offset);
        if(newnode.pNode->keycount == degree) //插入后结点的key数量等于度数（即大于度数-1），需要调整
            AfterInsert(newnode.pNode);
        keycount++;
        return true;
    }
}

```

```

template <class ElementType>
bool BPlusTree<ElementType>::AfterInsert(Node pNode) { //插入key后的调整
    ElementType key;
    offsetNumber offset;
    Node newnode = pNode->divide(key, offset); //当前结点分裂
    nodecount++;
    if(pNode->isRoot()) { //原结点为根结点，分裂后应创建新的根结点
        Node root = new TreeNode<ElementType>(degree, false);
        if(root == NULL) { //创建新结点失败
            cout << "Error: can not allocate memory for the new root after insert" << endl;
            return false;
        }
        else { //结点信息调整
            this->root = root;
            root->insertKey(key, -1);
            root->childs[0] = pNode;
            root->childs[1] = newnode;
            pNode->parent = root;
            newnode->parent = root;
            level++; //树的层数增加
            nodecount++;
            return true;
        }
    }
    else { //不是根结点
        Node parent = pNode->parent;
        int index = parent->insertKey(key); //父结点中插入新结点最小的key
        parent->childs[index + 1] = newnode; //父结点增加一个子结点
        newnode->parent = parent;
        if(parent->keycount == degree) { //父结点的key数也超过限制，需要调整
            return AfterInsert(parent);
        }
        return true;
    }
}

```

IndexManager 的插入函数（插入新索引值，已有索引则报错）：

```
void IndexManager::insertIndex(string filePath, string key, offsetNumber blockOffset, int type) { //在指定位置插入key
    setKey(type, key); //按照类型设置key
    //在索引对应B+树中，根据偏移量插入key
    if(type == TYPE_INT) { //key为整型的情况
        intMap::iterator itInt = indexIntMap.find(filePath);
        if(itInt == indexIntMap.end()) {
            cout << "Error:in insert index, index not exists" << endl;
            return;
        }
        else {
            itInt->second->Insert(intTmp, blockOffset);
            return;
        }
    }
    else if(type == TYPE_FLOAT) {
        floatMap::iterator itFloat = indexFloatMap.find(filePath); //查找对应索引文件
        if(itFloat == indexFloatMap.end()) {
            cout << "Error:in insert index, index not exists" << endl; //索引文件不存在
            return;
        }
        else {
            itFloat->second->Insert(floatTmp, blockOffset); //在索引文件对应B+树中插入key
            return;
        }
    }
    else if(type > 0) {
        stringMap::iterator itString = indexStringMap.find(filePath);
        if(itString == indexStringMap.end()) {
            cout << "Error:in insert index, index not exists" << endl;
            return;
        }
        else {
            itString->second->Insert(stringTmp, blockOffset);
            return;
        }
    }
    else {
        cout << "ERROR: in insert index: Invalid type" << endl; //无效类型
        return;
    }
}
```

## 第 3.6 节 Buffer Manager 接口

### 一、外部接口

1. 取得一个文件，根据文件的名称，如果当前没有这个文件就生成这个文件。  
 FileNode\* getFile(const char\* fileName);
2. 取得得一个文件的第一个块。  
 BlockNode\* getBlockHead(FileNode\* file);
3. 取得一个文件的某个块的后一个块。  
 BlockNode\* getNextBlock(FileNode\* file, BlockNode\* block);
4. 根据在文件中的偏移值（位置）取得块。  
 BlockNode\* getBlockByOffset(FileNode\* file, int offsetNum);



5. 为块或者文件的 dirty、pin 赋值。  
`void set_dirty(BlockNode* block);`  
`void set_pin(FileNode* file, bool pin);`  
`void set_pin(BlockNode* block, bool pin);`
6. 取得或更改某个块已被使用的空间大小。  
`size_t getUsingSize(BlockNode* block);`  
`void setUsingSize(BlockNode* block, size_t usingSize);`
7. 取得某个块的储存空间指针。  
`char* getContent(BlockNode* block);`
8. 在内存中删除某个文件，不会回写到硬盘。  
`void delete_fileNode(const char* fileName);`
9. 取得某个块的可用空间字节数。  
`int getBlockSize();`

## 二、实现技术

在缓存的管理的相关操作中，最关键的是申请文件和申请块的操作。在申请的块或者文件在内存中存在时要返回相应目标，否则需要申请新的空间，并考虑文件和块空间不足时替换已存在的文件或者块的相关维护操作。每个文件都存在一个块链表，其成员为在内存中存在的这个文件的相关块。对块的申请提供了 3 种方式：申请一个文件的头块、申请某一个块的下一个块、根据偏移量申请块。块的替换原则遵照了 Least Recently Used (LRU) 原则，而且对块的操作需要考虑对所属文件的块链表的影响。同时为了提高写的效率，给每个块赋予了 dirty 属性，所有对块的修改都会讲 dirty 置为真，将块写回磁盘时只有 dirty 为真的块才会真正被写入。

缓存管理与磁盘的交互采取了 write through 原则，即并不在数据更改时马上写回磁盘，读写主要发生在以下情形：

- 析构函数时，即数据库关闭时，将内存中所有文件的所有在内存中的块写回磁盘。
- 块池溢出时，需要将 LRU 原则选择的块写回磁盘。
- 文件池溢出时，将该文件在内存中的所有块写回磁盘。
- 申请块时，尝试读取磁盘中对应位置的数据。

还要注意的一点是，所有被锁定的块或者文件都不会被替换而提前写回磁盘。

核心函数为私有函数块获取函数 `BlockNode* BufferManager::getBlock(FileNode *curFile, BlockNode *curBlock)`，这个函数主要分为 3 部分，首先是块空间的获取：

```

if(BlockNum==0){
    btmp=&BlockPool[BlockNum++];
}
else if(BlockNum<MAX_BLOCK_NUM){
    int i;
    for(i=0;BlockPool[i].offset!=-1;i++);
    btmp=&BlockPool[i];
    BlockNum++;
}
else{
    int i;
    for(i=(LastReplaced==MAX_BLOCK_NUM-1?0:LastReplaced+1);!(BlockPool[i].LRU==false&&BlockPool[i].pin==false);i=(i==MAX_BLOCK_NUM-1)?0:i+1)
        if(BlockPool[i].LRU==true)
            BlockPool[i].LRU=false;
    LastReplaced=i;
    btmp=&BlockPool[i];
    if(btmp->preBlock)
        btmp->preBlock->nextBlock=btmp->nextBlock;
    if(btmp->nextBlock)
        btmp->nextBlock->preBlock=btmp->preBlock;
    if(!btmp->preBlock){
        if(btmp->block_file->HeadBlock!=btmp){
            cout << "program bug" << endl;
            exit(2);
        }
        btmp->block_file->HeadBlock=btmp->nextBlock;
    }
    writeBackToDisk(btmp->block_file->FileName, btmp);
    refresh_block(*btmp);
}

```

之后是对所属文件的块链表的修饰：

```

btmp->LRU=true;
btmp->block_file=curFile;

if(!curBlock){
    if(curFile->HeadBlock){
        curFile->HeadBlock->preBlock=btmp;
        btmp->nextBlock=curFile->HeadBlock;
    }
    curFile->HeadBlock=btmp;
    btmp->offset=0;
}
else if(!curBlock->nextBlock){
    btmp->preBlock=curBlock;
    curBlock->nextBlock=btmp;
    btmp->offset=curBlock->offset+1;
}
else{
    btmp->preBlock=curBlock;
    btmp->nextBlock=curBlock->nextBlock;
    curBlock->nextBlock->preBlock=btmp;
    curBlock->nextBlock=btmp;
    btmp->offset=curBlock->offset+1;
}

```

最后尝试从硬盘中读取相应的数据到内存：

```

fstream Tfile(curFile->FileName, ios::in|ios::binary);

Tfile.seekg(BLOCK_SIZE*btmp->offset, ios::beg);
Tfile.read(btmp->address, BLOCK_SIZE);

btmp->usingSize=*(size_t*)btmp->address;

Tfile.seekg((btmp->offset + 1)*BLOCK_SIZE, ios::beg);
char tempchar;
Tfile.read(&tempchar, 1);
if(Tfile.gcount() == 0)
    btmp->isBottom = true;

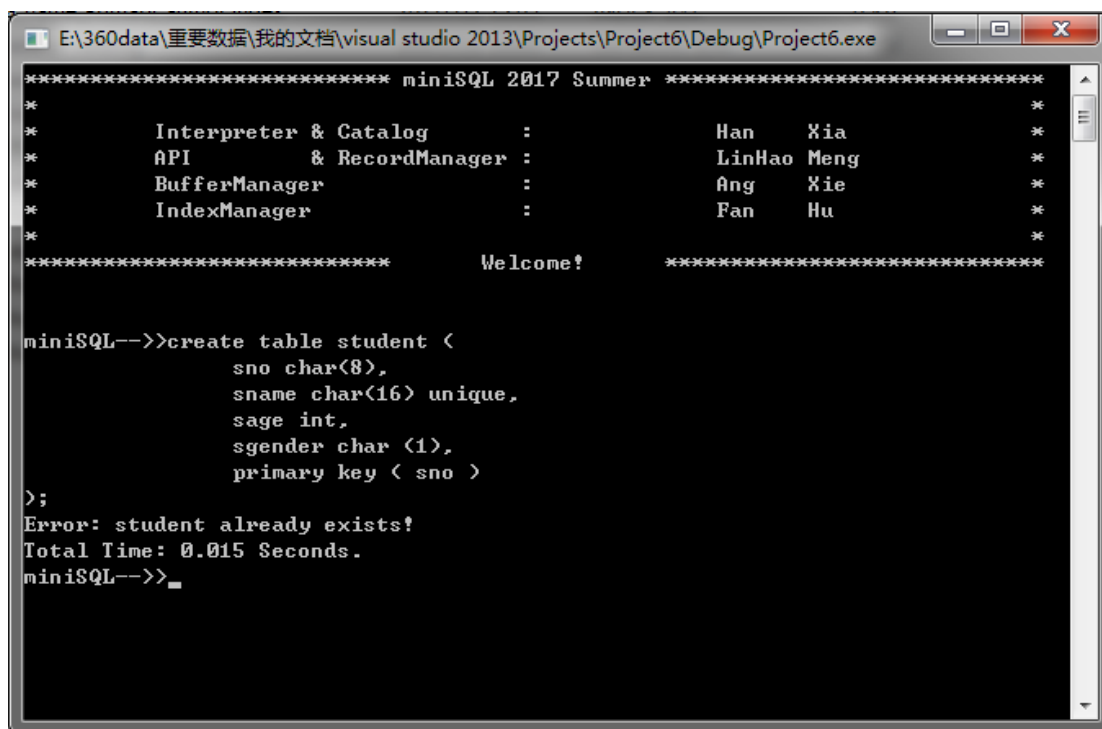
return btmp;

```

## 第 4 章 MINISQL 系统测试

### 正常测试

#### 1.建表



```

E:\360data\重要数据\我的文档\visual studio 2013\Projects\Project6\Debug\Project6.exe
***** miniSQL 2017 Summer *****
*
* Interpreter & Catalog      : Han Xia
* API & RecordManager      : LinHao Meng
* BufferManager             : Ang Xie
* IndexManager              : Fan Hu
*
***** Welcome! *****

miniSQL-->>create table student (
    sno char(8),
    sname char(16) unique,
    sage int,
    sgender char(1),
    primary key (sno)
);
Error: student already exists!
Total Time: 0.015 Seconds.
miniSQL-->>_

```

#### 2.建立索引

```

miniSQL-->>create index stunameidx on student (sname);
Total Time: 0.031 Seconds.

```

#### 3.插入值

```

miniSQL-->>insert into student values (<'12345678','wy',22,'M'>);
Total Time:      0 Seconds.
miniSQL-->>
insert into student values (<'1','wy1',22,'M'>);
Total Time:      0 Seconds.
miniSQL-->>
insert into student values (<'2','wy2',22,'M'>);
Total Time:      0 Seconds.
miniSQL-->>
insert into student values (<'7','wy55',23,'M'>);
Total Time: 0.016 Seconds.
miniSQL-->>
insert into student values (<'8','wy66',24,'M'>);
Total Time:      0 Seconds.

```

#### 4.包含“\*”选择语句

```

miniSQL-->>select * from student;
!12345678      !!wy      !!22      !!M      !
!1             !!wy1     !!22      !!M      !
!2             !!wy2     !!22      !!M      !
!7             !!wy55     !!23      !!M      !
!8             !!wy66     !!24      !!M      !
Total Time: 0.047 Seconds.

```

#### 5.包含选择条件的语句

```

miniSQL-->>select * from student where sage > 22 and sage <= 23;
!7             !!wy55     !!23      !!M      !
Total Time: 0      Seconds.

```






#### 6.使用条件删除记录并选择“\*”

```

miniSQL-->>delete from student where sno = '2';
1 records are deleted!
Total Time: 0.016 Seconds.
miniSQL-->>select * from student;
!12345678      !!wy      !!22      !!M      !
!1             !!wy1     !!22      !!M      !
!7             !!wy55     !!23      !!M      !
!8             !!wy66     !!24      !!M      !
Total Time: 0.063 Seconds.

```

#### 7.执行以上步骤所产生文件

 sno_Primary_student.index	2017/7/6 21:14	INDEX 文件
 student.catalog	2017/7/6 21:14	CATALOG 文件
 student.table	2017/7/6 21:14	TABLE 文件
 index.catalog	2017/7/6 21:14	CATALOG 文件
 stunameidx.index	2017/7/6 21:14	INDEX 文件





#### 8.执行 300 行左右语句的脚本文件

```

miniSQL-->>execfile test1.txt;
Openingtest1.txt...
Total Time: 0      Seconds.
Total Time: 1.95   Seconds.

```

产生的文件：

 a_name_Primary_author.index	2017/7/6 21:19	INDEX 文件	0 KB
 author.catalog	2017/7/6 21:19	CATALOG 文件	0 KB
 author.table	2017/7/6 21:19	TABLE 文件	0 KB
 index.catalog	2017/7/6 21:19	CATALOG 文件	0 KB

脚本部分语句：

```

test1.txt - 记事本
文件(F)  编辑(E)  格式(O)  查看(V)  帮助(H)

create table author
(
  a_id int,
  a_name char(20),
  primary key(a_name)
);

insert into author values(1, 'naaa');
insert into author values(2, 'naab');
insert into author values(3, 'naac');
insert into author values(4, 'naad');
insert into author values(5, 'naae');
insert into author values(6, 'naaf');
insert into author values(7, 'naag');
insert into author values(8, 'naah');
insert into author values(9, 'naai');
insert into author values(10, 'naaj');
insert into author values(11, 'naak');
insert into author values(12, 'naal');
insert into author values(13, 'naam');
insert into author values(14, 'naan');
insert into author values(15, 'naao');
insert into author values(16, 'naap');
insert into author values(17, 'naaq');
insert into author values(18, 'naar');
insert into author values(19, 'naas');
insert into author values(20, 'naat');
insert into author values(21, 'naau');
insert into author values(22, 'naav');
insert into author values(23, 'naaw');
insert into author values(24, 'naax');
insert into author values(25, 'naay');
insert into author values(26, 'naaz');
insert into author values(27, 'naba');
insert into author values(28, 'nabb');
insert into author values(29, 'nabc');
insert into author values(30, 'nabd');
insert into author values(31, 'nabe');
insert into author values(32, 'nabf');
insert into author values(33, 'nabg');
insert into author values(34, 'nabh');
insert into author values(35, 'nabi');
insert into author values(36, 'nabj');
insert into author values(37, 'nabk');
insert into author values(38, 'nabl');

insert into author values(1, 'nbaa');
insert into author values(2, 'nbab');
insert into author values(3, 'nbac');
insert into author values(4, 'nbad');
insert into author values(5, 'nbae');

```

选择“\*”：

```

!16: !nmap
!17: !nmaq
!18: !nmar
!19: !nmas
!20: !nmat
!21: !nmau
!22: !nmav
!23: !nmau
!24: !nmax
!25: !nmay
!26: !nmaz
!27: !nmba
!28: !nmbb
!29: !nmbc
!30: !nmbd
!31: !nmbc
!32: !nmbf
!33: !nmbg
!34: !nmbh
!35: !nmbi
!36: !nmbj
!37: !nmbk
!38: !nmb1
Total Time: 0.811 Seconds.
miniSQL-->>

```

9.quit 后 buffermanager 析构函数把数据写入磁盘

author.catalog	2017/7/6 21:21	CATALOG 文件	4 KB
author.table	2017/7/6 21:21	TABLE 文件	16 KB
index.catalog	2017/7/6 21:21	CATALOG 文件	4 KB
sno_Primary_student.index	2017/7/6 21:21	INDEX 文件	4 KB
student.catalog	2017/7/6 21:21	CATALOG 文件	4 KB
student.table	2017/7/6 21:21	TABLE 文件	4 KB
stunameidx.index	2017/7/6 21:21	INDEX 文件	4 KB

10.删除索引

```

miniSQL-->>
drop index stunameidx;
Total Time: 0.006 Seconds.

```

11.删除表

```

Total Time: 0.006 Seconds.
miniSQL-->>
drop table student;
Total Time: 0.006 Seconds.

```

执行以上两步后, student.catalog 和 stunameidx.index 都被删除了

test.txt	2017/6/20 21:02	文本文档	1 KB
test1.txt	2017/6/20 23:18	文本文档	20 KB
10w.txt	2017/6/20 23:34	文本文档	9,770 KB
例子代码.txt	2017/7/2 16:32	文本文档	2 KB
Project6.exe	2017/7/6 10:40	应用程序	582 KB
Project6.ilk	2017/7/6 10:40	Incremental Link...	2,560 KB
Project6.pdb	2017/7/6 10:40	Program Debug...	5,132 KB
index.catalog	2017/7/6 21:22	CATALOG 文件	0 KB

## 异常语句

### 1. 插入 unique 属性值相同的记录

```
miniSQL-->>create table student (
    sno char(8),
    sname char(16) unique,
    sage int,
    sgender char (1),
    primary key ( sno )
);
Total Time: 0.031 Seconds.
miniSQL-->>insert into student values ('2','wy2',22,'M');
Total Time: 0 Seconds.
miniSQL-->>
insert into student values ('2','wy333',22,'M');
Error: data duplication in unique attribute!
Total Time: 0.016 Seconds.
```

### 2. 重复建立相同属性上的索引

```
miniSQL-->>create index stunameidx on student ( sname );
Total Time: 0.011 Seconds.
miniSQL-->>create index stunameidx on student ( sname );
There is an index stunameidx already exists!
Total Time: 0.003 Seconds.
```

### 3. 在 int 属性值上输入非 int 型数据

```
miniSQL-->>insert into student values ('5','wy5',aa,'M')
;
Error:aa is not int type!
Total Time: 0.003 Seconds.
```

### 4. 删除不存在的数据

空表情况：

```
miniSQL-->>delete from student where sno = '2';
Error: The table is empty!
0 records are deleted!
Total Time: 0.004 Seconds.
```

非空情况：

```
miniSQL-->>insert into student values ('1','wy1',22,'M');
Total Time: 0.002 Seconds.
miniSQL-->>delete from student where sno = '2';
0 records are deleted!
Total Time: 0.002 Seconds.
```

#### 5. 输入无法识别的命令

```
miniSQL-->>alalal;
Command: alalal ERROR
```

#### 6. 删除不存在的 index

```
Command: alalal ERROR
miniSQL-->>drop index stunameidx;
Index stunameidx doesnot exist!
Total Time: 0.002 Seconds.
```

#### 7. 删除不存在的表

```
miniSQL-->>drop table student;
Error: Table student does not exist!
Total Time: 0.003 Seconds.
```

#### 8. 插入不存在的表的数据

```
miniSQL-->>insert into student values ('2','wy2',22,'M');
Error: Table student doesnot exist!
Total Time: 0.002 Seconds.
```

#### 9. SQL 语句错误

```
miniSQL-->>
insert into student v ('2','wy2',22,'M');
Syntax Error: expect "values" after tablename
```

其余语法错误绝大部分可以检查，不一一列举了

## 第 5 章 分工说明

本系统的分工如下：

Interpreter 模块	夏涵
API 模块	孟林昊
Catalog Manager 模块	夏涵
Record Manager 模块	孟林昊
Index Manager 模块	胡凡
Buffer Manager 模块	谢昂
测试	ALL