MiniSQL设计说明书

浙江大学2019~2020学年春夏学期《数据库系统》课程大程序报告

1 总体框架

1.1实现功能分析

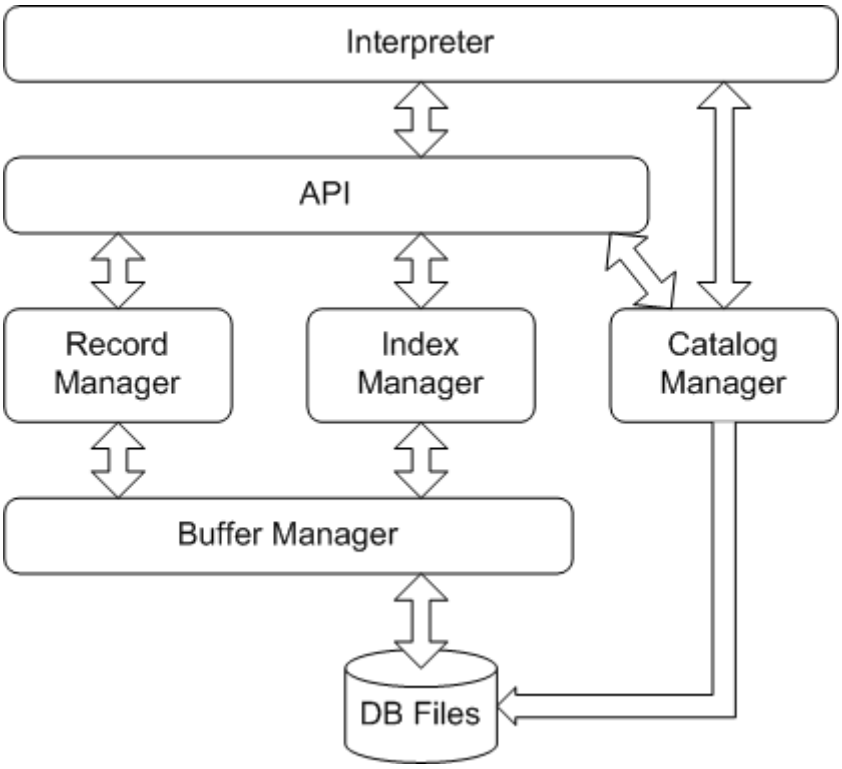
1.1.1实验目标

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实验基本需求

- **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$1 <= n <= 255$
- **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。
- **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）。
- **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。
- **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。

1.2 系统体系结构



1.3 运行环境

语言：C++ 14

开发环境：CLion 2020

2 各模块功能分析

2.1 *Interpreter*

`Interpreter`模块直接与用户交互，负责接收用户输入的命令（Sql语句/help/exit..）,并解析语句的格式、语法和分析其正确性，同时调用`API`模块和`Attribut`相应的方法执行语句，并通过`Condition`模块返回正确执行或错误的结果信息，实现程序总体流程控制

2.2 *API*

API模块是整个系统的核心，其主要功能是提供执行SQL语句的中间层，供*Interpreter*层调用，并根据语句解释后生成的命令内部形式，调用*Catalog Manager*同时根据其返回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底层功能，完成操作后将执行结果返回*Interpreter*层，完成各SQL语句及命令语句的实现与反馈。

2.3 *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。

2.4 *Index Manager*

- *Index Manager*是程序的索引部分,直接对*Buffer Manager*提供的内存索引块操作。它负责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+树中结点大小与缓冲区的块大小相同，B+树的度degree（阶数）由块大小和索引键（关键字）大小计算得到。

2.5 *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，一般可定为4KB或8KB。

2.6 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。
- 数据表中的记录条数及空记录串的头记录号。
- 数据库内已建的表的数目。

*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。

为减小模块之间的耦合，*Catalog*模块采用直接访问磁盘文件的形式，不通过*Buffer Manager*，*Catalog*中的数据也不要求分块存储。

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。

3 各模块具体实现

3.1 *Interpreter*

`Interpreter`成员函数组织如下：

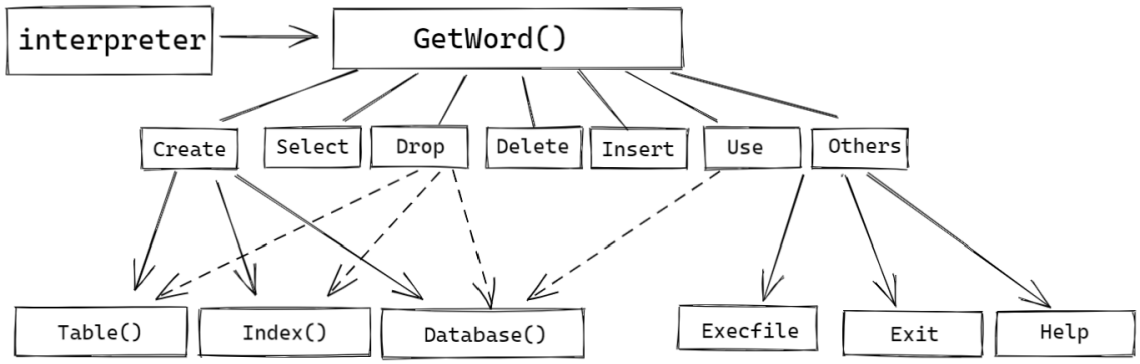


Diagram 1 **Interpreter**

具体执行上，`Interpreter`分为两级解析，其核心思想是正则表达式的运用：

- 第一级解析为语句分类，分为Sql语句、其他语句并调用相应的解析函数。
- 第二季解析为属性解析，将Sql语句中涉及的变量名、属性名进一步解析，并调用对象`API`和`Attribute`相应的成员函数执行，再通过`Condition`返回执行结果的信息。

3.1.1 一级解析

1. **语句输入：**该过程在main函数中执行。首先循环采用`getline(cin, str, ';')`的形式从输入流中读入一行字符，表明一条语句的结束，并将`;`以前的字符串传入`Interpreter`对象中，处理完成后，返回等待下一次命令输入。
2. **正则替换：**类`Interpreter`拥有成员变量`index`，表明当前语句解析的起始索引，成员函数`GetWord()`对语句从`Index`开始的部分执行正则替换。当第一次取得语句后，将头尾的空白字符去除，将语句内的多个空白字符替换成一个空格，得到正则替换后的命令语句。
3. **初步解析：**对命令语句再次使用`GetWord()`进行分割，这一次得到第一个`token`，对关键字进行判断，如`create`，`delete`，`drop`,并调用对应解析函数，进入二级解析。若没有对应关键字，则输出错误信息，返回错误`status`至外部循环。
4. **二级解析：**初步解析后，根据关键字跳转至相应的解析函数中，内部进行最终解析，根据相应的命令语句调用
   API 模块函数完成交互，显示结果，或根据语法错误输出错误信息。
5. **循环控制：**将执行之后的`status`返回至外部循环，正确执行返回1，错误执行返回0，退出返回2。

3.1.2 二级解析

- *CreateTable:*首先调用`GetWord()`进行正则替换首先进行正则替换，将`()`和`,`的分割转化为统一形式。接着跳过`create table`关键字，读取表名，然后将属性定义两边的() 去掉，获得中间属性。得到整个的属性定义字符串后，再次调用`GetWord()`函数对`,`进行分割，分别得到每个属性的定义。对于每个属性，再次调用`GetWord`函数对空格进行分割，得到属性名、类型、长度、唯

一约束，主键定义等信息。将所有读取的属性信息全部放入`Attribute`对象中，再根据之前读取的表名，调用API 中的`tableCreate`函数完成表的创建。此外，流程会逐步对语法错误进行判断，并返回相应的错误信息。

- *CreateIndex:*首先进行正则替换，将() 的分割转化为统一形式。接着跳过*create index* 关键字，读取索
  引名、然后跳过*on* 关键字，读取表名，将属性名两边的() 去掉，读取属性名。根据读入的索引名、表名、属
  性名信息，构造一个*Index* 类变量，然后调用API 中的*create_index* 函数完成索引的创建。此外，流程会对
  语法错误进行判断，并输出相应的错误信息。
- *drop table ：* 首先跳过drop table 关键字，读取表名，然后根据表名调用API 中的drop_table 函数完成对
  表的删除。此外，流程会对语法错误进行判断，并抛出相应的异常。
- *drop index ：* 首先跳过drop index 关键字，读取索引名，然后根据索引名调用API 中的drop_index 函数完
  成对索引的删除。此外，流程会对语法错误进行判断，并抛出相应的异常。
- *show ：* 首先跳过show 关键字，然后根据下一个词是indexes 还是tables 来决定显示索引还是数据表，如果
  关键词不是这两者则报错。
- *insert ：* 首先进行正则替换，将() 和, 的分割转化为统一形式。接着跳过insert into 关键字，读取表名，
  然后跳过values 关键字，将属性值两边的() 去掉，提取中间的属性值。得到整个属性值定义字符串后，调用
  String 类中split 函数对, 进行分割，得到每个属性的值。对于每个属性值，判断其两端是否有'' 或"" ，
  若有则将其去除，将全部得到属性值构造成一个TableRow 类型的变量，调用API 中insert_row 函数对记录进
  行插入。此外，流程会对语法错误进行判断，并抛出相应的异常。
- *select ：* select 语句主要有以下几种形式
  - *select * from [表名] ;*
  - *select * from [表名] where [条件];*
  - *select [属性名] from [表名];*
  - *select [属性名] from [表名] where [条件]; 。*

我们通过自定义函数String substring(String stmt, String start, String end) 来得到位于两个字符串之间的子串，以最复杂的select [属性名] from [表名] where [条件]; 为例，先通过substring(stmt,"select ", "from") 来得到属性名或* ，如果是属性名则通过split 函数根据, 进行分割。同理通过substring (stmt,"from ", " where") 可以得到表名，通过substring (stmt,"wuere ", "") 可以得到条件。然后只需根据具体情况判断即可。

- *delete ：* delete 的处理方式与select 类似。
- *exefile :* 跳过关键字，读取文件名。递归调用解析流程，直到达到文件末尾或异常退出

3.2.3 函数和接口介绍

```
/*
 * Member variables
*/
API *ap;//the pointer to api interface
string fileName;//execfile_name
int idx = 0;//the index of the sentence
```

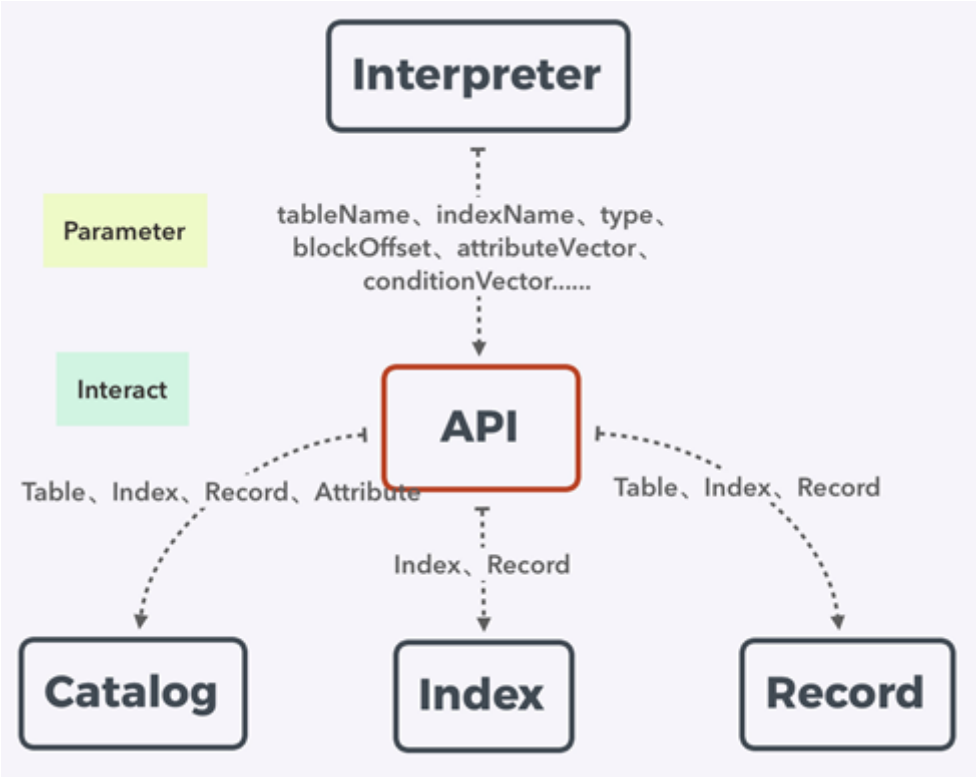
```
    /**
     * This function classifies and analyzes the sentences entered by the user
and
     * call the corresponding member functions.
     * @param s The string entered by the user.
     * @return The status if the sentence done correctly.
     */
    int interpreter(string &s);

    /**
     * This function splits sentences into need words
     * @param s The whole sentence
     * @return the first word we need started from idx in the sentence
     */
    string getWord(string &s);

    /**
     * These functions call the corresponding API model as the function name
suggests
     * if the sentence is classified correctly or print ERROR::INFO.
     * @param word The word split from s
     * @param s The whole sentence
     * @return The status if the sentence done correctly
     */
    int CreateTable(string &word, string &s);

    int CreateIndex(string &word, string &s);

    int Select(string &word, string &s);

    int Drop(string &word, string &s);

    int Delete(string &word, string &s);

    int Insert(string &word, string &s);
};
```

3.2 *API*

3.2.1 工作原理

API模块的交互主要通过协调不同函数的操作级别与反馈路由实现，内部实现主要可以分为四个组件，以下为API模块与其他模块数据流交互图，以及API模块内各组件工作原理与实现细则。

- **API与各模块交互架构图**



- **数据表(Table)操作**

  据表的操作分为创建新表和删除已有表，其中创建新表操作首先根据Interpreter层传入的表名、表字段属性以及主键的信息，调用Record Manager和Catalog Manager查询数据库当前状态，并执行创建新表操作；

  删除已有表的操作则首先检测表是否存在，存在则调用IndexDrop函数删除其索引后，调用Record Manager和Catalog Manager将底层记录删除。

  在处理过程中，若某一环节或底层返回值在查询过程中异常，则输出相关错误提示，并终止当前操作；若查询正确并执行成功，同样返回正确提示，完成表操作的执行。

- **数据索引(Index)操作**

  数据索引操作分为索引的修改与查询，其中索引修改包含了索引的创建、删除、插入索引操作，索引的查询用以获取索引的存放地址。

  索引操作的调用发生在Interpreter层，由其中的CreateIndex与Drop函数调用，负责执行Create、Drop与Insert操作，其实现则通过查询Catalog和Record层获取相关状态，之后调用Index中的函数进行索引的更新。

  索引的查询则是首先调用Catalog中的Get_Indexes进行索引名获取，之后通过循环更新传参指针的索引名。

  本部分处理涉及子部分较广，因此提供了多种报错信息，若发生错误可以清楚地得到错误位置，从而做出修改。

- **记录(Record)操作**

  记录的操作分为记录的更新、查询与回显操作，其中更新操作分为插入、删除、记录索引的修改等操作。

  记录的更新同样是由Interpreter调用，在解析到Drop、Delete、和Insert操作后，Interpreter将读入的参数传递给API中的RecordDelete、和RecordInsert函数，并在实现中进一步调用RecordIndexDelete、RecordIndexInsert函数用以执行记录索引的操作。在工作过程中，除了使用到Index模块、Catalog模块与Record模块的接口外，还需要用到Attribute功能函数，这部分将在下一节中讲解。

记录的查询操作分为获取记录数量GetRecordNum与记录大小GetRecordSize，这两个函数都只需要表名这一个参数传入，具体的实现则是调用了Catalog中的底层函数，本处仅作为上下层次间的路由功能。

记录的回显函数RecordShow可以将记录输出，此处应用于用户输入Select语句的情况，返回搜索表中的查询数据。

在工作过程中，如遇到表名不存在或查询区块位移失败等情况，系统会输出相应错误情况，用户可相应改变查询操作。

- **属性(Attribute)操作**

  本部分在API与Record Manager模块中作为接口，其中GetTypeSize函数获得属性类型的大小，仅作为中转，进一步调用Catalog里的Get_Length函数；GetAttribute为本部分主要函数，通过在Catalog中查询表中属性集合并返回到Record层或本层中，作为表属性的连接接口。

  由于本部分主要作为路由进行中转，因此主要功能的实现还需依赖于Catalog层，具体实现方式可查看该部分报告内容。

3.2.2 主要函数功能描述

以下为API模块实现的重要函数及其功能清单，除去上一章中提到的四类操作中主要的函数，还补充了内部的私有函数，主要作为内部接口使用。

1、void TableCreate(string tableName, vector *attributes, string primaryKeyName, int primaryKeyLocation)

目的：创建一个新的表，若之前已经存在同名表，则报错。

参数： string tableName　新创建表的名称

vector *attributes　该新建表的属性向量

string primaryKeyName　命名为主键的字段名称

int primaryKeyLocation　主键定位字段位置

思路：检测是否已有重复表，若无则调用Catalog与Record中的创建函数建立新表。

2、void TableDrop(string tableName)

目的：删除名称为tableName的表。

参数： string tableName　即将删除的表的名称

思路：错误检测后通过私有函数获取索引值，并调用Catalog与Record中的删除表函数。

3、void IndexCreate(string indexName, string tableName, string attributeName)

目的：创建一个新的索引，若之前已经存在同名索引，则报错。

参数： string indexName　新索引的名称

string tableName　所在表的名称

string attributeName 索引所在的属性名

思路：属性是否重复与表状态检测后，若无问题则调用Catalog与Record中的创建索引函数。

4、void IndexDrop(string indexName)

目的：删除名称为indexName的索引。

参数： string indexName 删除的索引名称

思路：检测到存在该索引值后调用Catalog与Index中的删除索引函数执行删除。

5、void IndexInsert(string indexName, char * contentBegin, int type, int blockOffset)

目的：向索引树中插入索引值。

参数： string indexName 插入索引的名称

char* contentBegin 插入字段所在内存指针

int type 字段类型

int blockOffset 偏移位

思路：根据type值检测类型，对于int、float等类型执行不同操作，并通过强制转换与流操作进行转换后，调用Index模块的InsertIndex函数。

6、void RecordInsert(string tableName, vector *recordContent)

目的：向表中插入一条记录。

参数： string tableName 插入表的名称

vector *recordContent 插入记录内容存放的向量地址

思路：首先获取当前所有属性信息，并检测插入值是否与数据库状态冲突，若冲突则报错，否则执行记录的插入操作。

7、void RecordDelete(string tableName, vector *conditionVector)

目的：在表中删除一条/多条记录。

参数： string tableName 删除表的名称

vector *conditionVector 提供插入记录的状态信息

思路：本函数为重载函数，当只提供第一个参数时，默认删除所有记录，在给定conditionVector后，首先获取当前全部属性，并调用Index模块的SearchIndex函数检测存在性之后，调用Record Manager与Catalog Manager中的函数执行记录所在块的删除。

8、void RecordShow(string tableName, vector *attributeNameVector, vector *conditionVector)

目的：展示表中的一条/多条数据。

参数： string tableName 所在表的名称

vector *attributeNameVector 选定的属性名称

vector *conditionVector 提供输出记录的状态信息

思路：本函数为重载函数，当只提供前两个参数时，默认展示所有符合属性的记录，在给定conditionVector后，首先获取当前全部属性并打印，之后调用Index模块的SearchIndex函数检测存在性，再通过Record Manager中的函数执行记录信息的输出。

9、int GetAttribute(string tableName, vector *attributeVector)

目的：获取表中的属性。

参数： string tableName　查询表的名称

vector *attributeVector　保存查询信息

思路：首先检测表的存在性，无误后调用Catalog中的Get_Attribute执行查询。

10、int TableExist(string tableName)

目的：检查表的存在性。

参数： string tableName　需检查的表名称

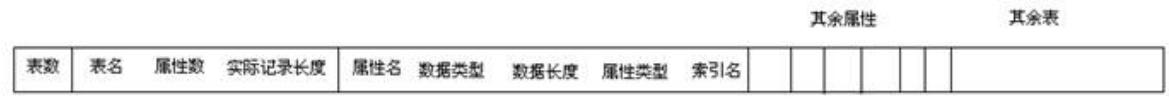
思路：本函数为私有函数，作为内部接口用于检验正确性。首先调用Catalog中的Find_Table执行查询，将查询结果与预先设计的宏定义TABLE_FILE比较后进行输出与返回，若正确则返回1，不正确则输出表不存在，并返回0。

API作为整个数据库系统的中间件，起到了整合与通信的作用，将Catalog Manager模块，Record Manager模块和Index Manager模块的各个功能函数形成统一的接口，将结果反馈回最前端的Interpreter模块和Main入口函数，为整个系统的顺利运转和各模块之间的协调提供了有效的接口。

---

3. 3 *Catalog Manager*

数据字典格式



其具体实现也是按照语句类型进行分别处理：

创建数据库语句

CREATE DATABASE 数据库名；

根据Interpreter处理生成的初步内部数据形式，提取数据库名，验证是否有重名现象。若无则建立数据库文件夹及数据字典文件，否则不能创建该数据库并打印出错信息供用户参考。

创建表语句

CREATE TABLE 表名 （

列名 类型

列名 类型

……

PRIMARY KEY(列名)

）；

根据Interpreter处理生成的初步内部数据形式，提取表名，验证是否有重名现象。若无则建立表文件，同时将表的各种信息记录在数据字典内，若有主键定义，应对主键生成索引，并将索引信息记录在数据字典内。若有重名表，则创建表命令失败，并打印出错信息。

创建索引语句

CREATE INDEX 索引名 ON 表名（列名）；

根据Interpreter处理生成的初步内部数据形式，提取表名及列名，在数据字典中查找有无该表及属性的定义，且验证是否可对该属性建立索引。若可建，则创建索引文件，同时将索引信息记录在数据字典内。若不可建索引，则根据不同类型打印出错信息供用户参考。

选择语句

SELECT * FROM 表名； SELECT 列1，列2，....列n FROM 表名；

SELECT * FROM 表名 WHERE 条件； SELECT 列1，列2，...列n FROM 表名 WHERE 条件

根据Interpreter处理生成的初步内部数据形式，提取表名及各列名，在数据字典中查找有无该表及各属性的定义，同时须查找条件中属性名是否在表中有定义，且常量值的数据类型与属性的数据类型是否可比。若有条件，须先根据条件及条件中已建索引的属性名来选择一个合适的索引供Record Manager使用。任何的不符都将打印出相应的出错信息供用户参考。

插入记录语句

INSERT INTO 表名 VALUES (值1，值2，.....,值n)；

根据Interpreter处理生成的初步内部数据形式，提取表名及各属性值，在数据字典中查找该表的定义，并验证各属性值是否满足各属性的数据类型定义。同时将各属性值转化为记录形式。任何的不符都将打印出相应的出错信息供用户参考。

删除记录语句

DELETE FORM 表名；

根据Interpreter处理生成的初步内部数据形式，提取表名，在数据字典中查找该表的定义。若有，删除该表文件内的所有记录并更新该表在数据字典中的信息,同时须查找建立在该表之上的所有索引，并将所有索引文件内的节点删除且须更新索引文件在数据字典中的信息。若在数据字典中无该表的定义，则须打印出必要的出错信息供用户参考。

DELETE FORM 表名 WHERE 条件；

根据Interpreter处理生成的初步内部数据形式，提取表名及WHERE条件，，在数据字典中查找有无该表的定义，同时须查找条件中属性名是否在表中有定义，且常量值的数据类型与属性的数据类型是否可比。若有条件，须先根据条件及条件中已建索引的属性名来选择一个合适的索引供Record Manager 使用。任何的不符都视为语句的语义错误，并打印出相应的错误信息供用户参考。

退出MiniSQL系统语句

EXECFILE 脚本文件名；

根据Interpreter处理生成的初步内部数据形式，提取脚本文件名，并查找有无该文件。若有，则打开文件，依次读取每条命令并执行。若不存在该文件，则打印出相应的出错信息供用户参考。

删除索引语句

DROP INDEX 索引名；

根据Interpreter处理生成的初步内部数据形式，提取索引名，在数据字典中查找有无该索引的定义。若有，则删除该索引在数据字典中的信息，并删除索引文件。若无，则打印出相应的出错信息供用户参考。

删除表语句

DROP TALBE 表名；

根据Interpreter处理生成的初步内部数据形式，提取表名，在数据字典中查找有无该表的定义。若有，则先查找该表上的所有索引并删除，然后删除该表在数据字典中的信息，最后删除该表文件。若在数据字典中无该表的定义，则打印出相应的出错信息供用户参考。

删除数据库语句

DROP DATABASE 数据库名；

根据Interpreter处理生成的初步内部数据形式，提取数据库名，查找有无以数据库为名的文件夹。若有，删除文件夹内所有的文件并删除文件夹。若无此文件夹，则打印出相应的出错信息供用户参考。

使用数据库语句

USE 数据库名；

根据Interpreter处理生成的初步内部数据形式，提取数据库名，并查找有无以数据库为名的文件夹。若有，则先判断是否与当前使用的数据库一致，若不一致，才须写回当前数据库的缓冲区内的内存块，并将当前数据库更新为所指定的数据库名。若不存在所查找的文件夹，则表明不存在用户指定的数据库，则须打印出相应的出错信息供用户参考。

其主要函数的功能描述如下：

1. **Table相关构建函数**

- 创建table：
  `int Create_Table(string tableName, vector<Attribute> *attributeVector, int primaryKeyLocation);`
- 查找对应的table文件：
  `int Find_Table(string tableName);`
- 查找对应的table文件：
  `int Drop_Table(string tableName);`

2. **index相关构建函数**

- 创建index：
  `int Create_Index(string indexName, string tableName, string attributeName, int type);`
- 查找对应的Index文件：
  `int Find_Index(string indexName);`
- 删除对应的Index文件：
  `int Drop_Index(string index);`

3. **Record相关构建函数**

- 插入记录：
  `int Insert_Record(string tableName, int recordNum);`
- 删除记录：
  `int Delete_Record(string tableName, int deleteNum);`

4. **找特定Table的所有Index，存在vector数组中:**
   `int Get_Indexes(string tableName, vector<string> *indexNameVector);`
   Overload function:找所有的Index，存在vector数组中（overload）：
   `int Get_Indexes(vector<IndexInfo> *indexs);`
5. **返回Index类型:**
   `int Get_Indextype(string indexName);`
6. **返回特定table的所有attribute:**
   `int Get_Attribute(string tableName, vector<Attribute> *attributeVector);`

7. **返回长度:**
   `int Get_Length(string tableName);`
   返回type对应长度(overload):
   `int Get_Length(int type);`
8. **返回特定table的record，通过修改参数实现返回:**
   `void Get_Record(string tableName, vector<string> *recordContent, char *recordResult);`
9. **返回所有record数目:**
   `int Get_RecordNum(string tableName);`

通过Catalog Manager 模块的处理，已完成了部分语句的功能，对于未完成的语句，也对语句的内部表示形式进行了扩充，通过API接口的功能，即可将信息传递给Record Manager模块和Index Manager模块，为Record Manager模块及Index Manager模块的实现提供了充足的信息。

3.4 *Record Manager*

其具体实现如下

选择语句

SELECT * FROM 表名； SELECT 列1，列2，....列n FROM 表名；

无WHERE条件，则须对表文件进行遍历，并对每条记录，根据选择属性名组，打印出对应的属性值。

SELECT * FROM 表名 WHERE 条件； SELECT 列1，列2，...列n FROM 表名 WHERE 条件

若无可用的索引，则须对表文件进行遍历，并对每条记录，进行WHERE 条件匹配，若符合，则打印相应属性的值。若有可用的索引，则可调用Index Manater模块的功能，查找符合WHERE 条件的记录，并打印相应属性的值。在最后打印出被选择的记录的条数。

插入记录语句

INSERT INTO 表名 VALUES (值1，值2，.....,值n);

根据Catalog Manager处理生成的初步内部数据形式，提取表名及记录，根据信息计算插入记录的块号，并调用Buffer Manager 的功能，获取指定的内存块，并将记录插入到内存中，同时修改脏位及表信息即可。

删除记录语句

DELETE FORM 表名 WHERE 条件;

根据Catalog Manager处理生成的内部数据形式，提取表名及可用的索引。若有可用的索引，则须根据索引来查找符合条件的记录，删除该记录并更新该表在数据字典中的信息，循环往复，直至所有符合条件的记录都被删除为止。若无可用的索引，则须对表文件中的所有记录进行一次遍历，并对每一条记录都需要判别是否符合条件。若符合条件，则删除该记录并更新表在数据字典中的信息。在最后打印出被已被删除的记录的条数。

其主要函数的功能描述如下:

```
创建表:
int RecordManager::tableCreate(string table_name)

删除表:
int RecordManager::tableDrop(string table_name)

创建索引:
int RecordManager::indexCreate(string index_name)
```

删除索引：
```
int RecordManager::indexDrop(string index_name)
```

把一条记录插入到表中：
```
int RecordManager::recordInsert(string table_name, char *record, int
size_of_recourd)
```

显示表中所有符合条件的记录：
```
int RecordManager::recordAllShow(string table_name, vector<string>
*attribute_name_vector, vector<Condition> *condition_vector)
```

显示一个block中的一张表的记录：
```
int RecordManager::recordBlockShow(string table_name, vector<string>
*attribute_name_vector, vector<Condition> *condition_vector, int block_offset)
```

显示一个block中的一张表的记录（overload）：
```
int RecordManager::recordBlockShow(string table_name, vector<string>
*attribute_name_vector, vector<Condition> *condition_vector, blockNode *block)
```

一张表中满足条件的记录条数：
```
int RecordManager::recordAllFind(string table_name, vector<Condition>
*condition_vector)
```

一个block中一张表的记录数量：
```
int RecordManager::recordBlockFind(string table_name, vector<Condition>
*condition_vector, blockNode *block)
```

删除所有符合条件的记录：
```
int RecordManager::recordAllDelete(string table_name, vector<Condition>
*condition_vector)
```

删除一个block中一张表中的记录：
```
int RecordManager::recordBlockDelete(string table_name, vector<Condition>
*condition_vector, int block_offset)
```

删除一个block中一张表中的记录（overload）：
```
int RecordManager::recordBlockDelete(string table_name, vector<Condition>
*condition_vector, blockNode *block)
```

插入表中所有记录的索引：
```
int RecordManager::indexRecordAllAlreadyInsert(string table_name, string
index_name)
```

插入表中所有记录的索引（overload）：
```
int RecordManager::indexRecordBlockAlreadyInsert(string table_name, string
index_name, blockNode *block)
```

判断一条记录是否满足条件：
```
bool RecordManager::recordConditionFit(char *recordBegin, int size_of_recourd,
vector<Attribute> *attributeVector,  vector<Condition> *condition_vector)
```

显示记录的内容：
```
void RecordManager::recordPrint(char *recordBegin, int size_of_recourd,
vector<Attribute> *attributeVector, vector<string> *attribute_name_vector)
```

显示记录内容的值：

```
void RecordManager::contentPrint(char *content, int type)

判断记录内容是否满足条件:
bool RecordManager::contentConditionFit(char *content, int type, Condition
*condition)

获取索引文件名:
string RecordManager::indexFileNameGet(string index_name)

获取表的文件名:
string RecordManager::tableFileNameGet(string table_name)
```

通过Record Manager模块的功能，最终实现了对记录的插入，删除和选择的SQL语句。但实现该功能，则是在Interpreter模块，API模块和Catalog Manager模块的功能之上，在调用Buffer Manager模块和Index Manager模块的功能，才得以实现。由此可见，这三条SQL语句通贯了所有模块的功能，是最能体现模块之间协调工作的测试用例。

3.5 *Index Manager*

3.5.1 B+树

1、B+树定义及其性质

B+树是B树的一种变形，它的叶子结点将存储所有的数据以及相应记录的地址，而叶子结点以上各层则作为索引，只存放一些关键字和孩子指针，因此最大化了内部结点的分支因子。此外B+树还将所有的叶子结点串成链表，因此对于遍历也更加地高效。下图是一棵典型的B+树结构。

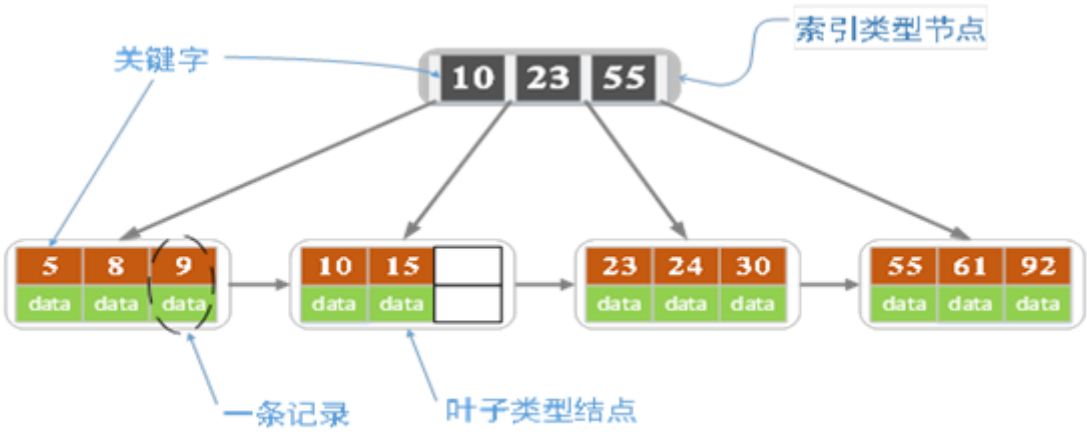


图2.1 B+树的结构

具体地，对于一棵n阶B+树(或称B+树的度degree为n)而言，任意一个结点所能存放最大的关键字数量为n-1，而关键字数量最少不能少于(n–1)/2。例如若n = 5，一个结点所能包含的关键字数量为2-4个。而所有的关键字都将以非降序的顺序存放，使得keys[0] < keys[1] < ... < keys[n-1]。另外，对于每个结点而言，将有一个指向其父结点的指针以及一个布尔变量leaf来判断该结点是否为叶结点。如果不是叶子结点（即为内部结点），将有n个指向其孩子结点的指针，孩子结点的数量恰好为该结点中存储的关键字数量加1，而叶子结点没有指向孩子结点的指针。

对于叶子结点而言，首先所有叶子结点具有相同的深度，即树的高度h；还有指向下一结点的指针next，方便遍历整棵B+树。由于存储的不多于n-1个关键字都是对数据表中各记录的索引值，因此需要同等数量的值来存储各索引值在索引文件中的地址信息。

而对于内部结点而言，其中的关键字对存储在其所有子树中的关键字范围加以分割。举个例子，假设keys[i]代表一个内部结点中存储的第i个关键字，而childkey[i]代表其第i个孩子结点所在子树中的任意一个关键字，那么有：

$$childkey[0] < keys[0] < childKey[1] < \ldots < childKey[i] < keys[i] < childKey[i+1] < \ldots < keys[n-1] < childkey[n].$$

另外，内部结点中并不存储真正的信息，而是保存每个叶子结点中（除了第一个叶子结点）的最小值作为索引。那么这样，我们就可以有效实现对B+树中某个关键字的搜索。

值得注意的是，在本次大程中，为了方便B+树的实现，我们将B+树的度设置为奇数。

2、B+树的搜索

对于B+树的搜索操作，首先，我们从根结点出发，寻找目标值key在当前结点所有关键字中的大小位置，一旦确定keys[i] ≤ key < keys[i+1]，那么我们就进入到第(i+1)个子结点childs[i+1]。重复上述步骤，直到最终进入到叶子结点，在叶子结点中搜索目标值，如果找到则返回对应的记录地址，否则该关键字不存在。

3、B+树的插入

（1）对于空树的情况，直接创建一个叶子结点，将记录插入其中，此时这个叶子结点也是根结点。而一般情况下，插入的第一步类似于B+树的搜索过程，从根结点出发，一直来到对应的叶子结点。此时，我们在叶子结点中寻找新值将插入的位置i，即i满足keys[i-1]< NewKey <keys[i]，需要注意的是一旦该叶子结点中已经包含相同的关键字值，则插入失败。

（2）在叶子结点中完成插入后，接下来可能需要对B+树的结构进行进一步的调整。一旦插入后当前叶子结点关键字数量大于等于m，我们需要将这个叶子结点分裂成左右两个子结点，左子结点包含前(m+1)/2个关键字，右结点包含剩余(m-1)/2个关键字，将第(m+1)/2个关键字进位到父结点中。进位到父结点的关键字左孩子指针指向左结点，右孩子指针指向右结点。并将分裂得到的两个子结点的父指针指向该父结点。接下来将当前指针指向父结点。

（3）首先，当前结点由于是叶子结点的父结点，因此必然是内部结点。那么如果当前关键字个数小于等于m-1，则不需要调整，插入结束。否则，将其同样分为左右两个内部结点，左结点包含前(m-1)/2个关键字，右结点包含后(m-1)/2个关键字，将第(m+1)/2个关键字进位到当前结点对应的父结点中，并将进位到父结点的关键字的左孩子指向左结点，右孩子指向右结点。重复上述步骤，直至不需再调整。

（4）如果最后出现了根结点中关键字数量等于m的情况，需要新建一个结点作为新的根结点，同时也是原根结点的父结点，这样再进行步骤（3）即可。

下图展示了一棵B+树的插入过程（包括结点分裂）。

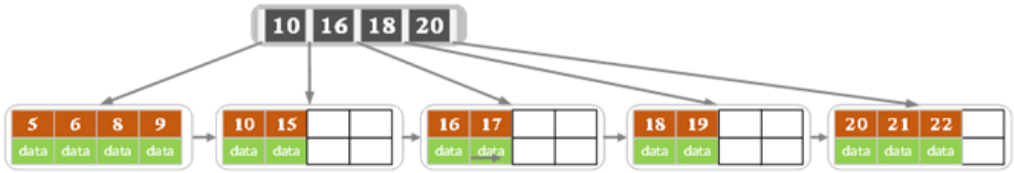


图 2.2 插入前的 B+树

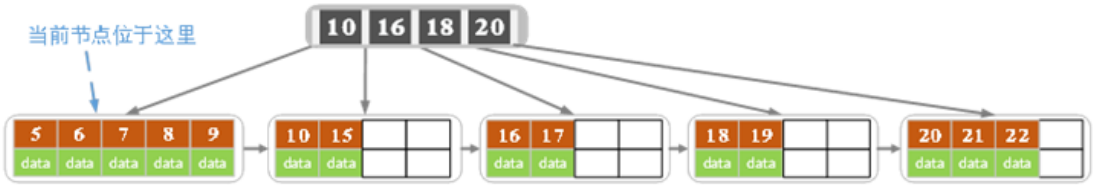


图 2.3 插入后未调整的 B+树

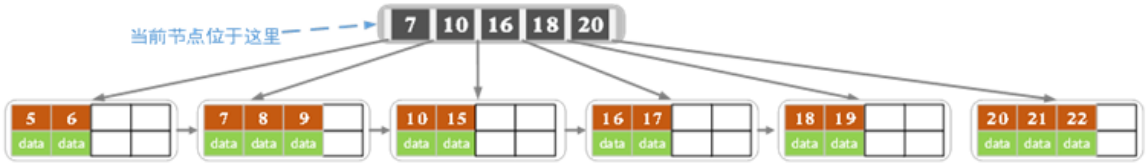


图 2.4 第一次调整后的 B+树

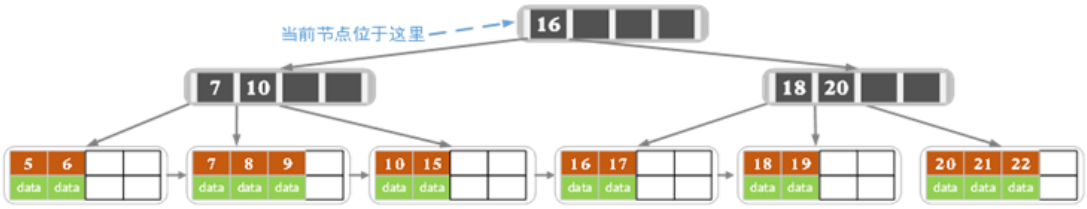


图 2.5 第二次调整后的 B+树

4、B+树的删除

（1）类似于B+树的搜索过程，从根结点出发，一直来到对应的叶子结点。在叶子结点中寻找被删除的key值，如果不存在，则删除失败。否则删除叶子结点中对应的关键字。完成删除之后，判断删除后结点的关键字数量是否大于等于(m-1)/2，若小于，则需要对B+树的结构进行进一步的调整。

（2）若存在兄弟结点有富余的关键字，也就是说关键字数量大于(m-1)/2，那么只需向其借一个关键字过来即可。同时在父结点中对关键字进行一定的替换（若跟左兄弟借，在父结点中用左兄弟的最大关键字来替换自身的最小关键字；若跟右兄弟借，在父结点中用右兄弟的第二个关键字来替换原第一个关键字）。完成调整后结束删除操作。

（3）若兄弟结点中没有富余的关键字，也就是说关键字数量恰好等于(m-1)/2，则将当前结点和兄弟结点合并成一个新的叶子结点，并删除父亲结点中相应的关键字。将当前结点转向该父结点，如果其关键字数量大于等于(m-1)/2，则删除操作结束，否则执行下一步（4）。

（4）此时当前结点必然是内部结点（内部结点或叶子结点的父亲结点）。若兄弟结点有富余的关键字，也就是说关键字数量大于(m-1)/2，只需将兄弟结点的最大或最小关键字上移至父结点，而父结点中对应的关键字下移至当前结点，删除结束。

（5）若兄弟结点都没有富余的关键字，也就是说关键字数量恰好等于(m-1)/2，将当前结点和兄弟结点以及父亲结点下移的一个关键字合并成一个新的结点，并将该新结点的父指针指向父亲结点。将当前结点转向父亲结点，如果其关键字数量大于等于(m-1)/2，则删除操作结束，否则重复（4）、（5）直至结束。

下图展示了一棵B+树的删除过程（包括结点合并）。

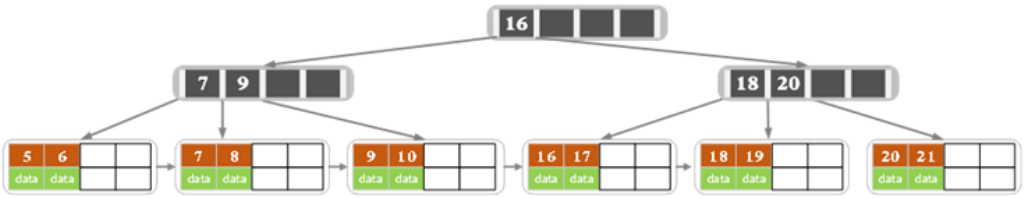


图 2.6　删除前的 B+树

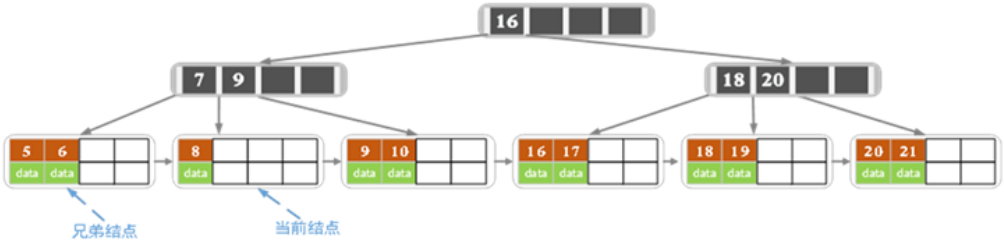


图 2.7　删除后未调整的 B+树

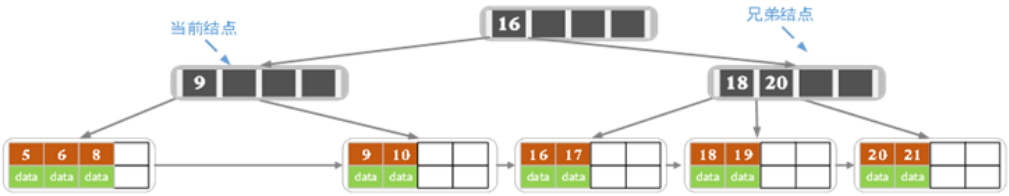


图 2.8　第一次调整后的 B+树

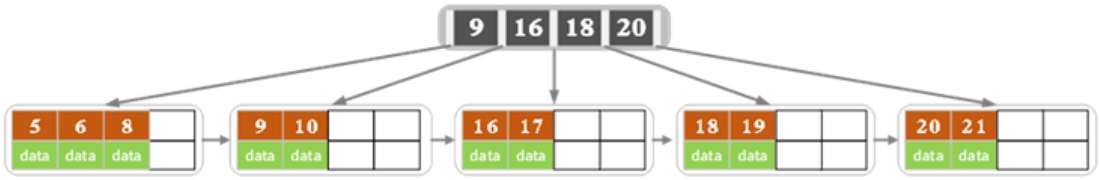


图 2.9　第二次调整后的 B+树

3.5.2 设计思路

首先考虑到miniSQL只需要支持int , float和char(n)三种数据类型，因此我们可以借助map容器构造三种不同的数据类型来实现对索引的分类存储：

map<string, BPlusTree* >，map<string, BPlusTree* >，map<string, BPlusTree* >；此外，由于我们借助B+树实现对索引的操作和存储，因此在B+树的实现上，我们运用了模板类的方法：使用 template 声明一个模板后，我们只需将B+树中关键字的数据类型定义为Type即可，这也就实现了B+树操作针对不同数据类型的通用性。

另外，在B+树层面，我们将以索引值作为B+树各结点的关键字，而在叶子结点中，将存储每个索引值在索引文件中的地址信息(用int变量存储)。而一个block将对应一个叶子结点，因此在设计B+树的degree时，我们将一个block的size除以一个关键字的size与一个地址信息的size之和。但当计算结果为偶数时，为了更方便地进行B+树操作，我们将其减一后再作为最终的degree值。

另外，还有一个问题在于，在Index Manager中进行索引创建，索引删除，索引值插入，索引值删除等操作时，如何针对索引值的数据类型而选择正确的存储索引的map容器进行操作？对此，在索引创建删除等成员函数中，我们引入一个int类型的参数dataType来记录索引类型，dataType为-1时表示数据类型为float，dataType为0时表示数据类型为int，dataType>0时代表数据类型为string。（或者为char(n), n的值记为dataType的值）

3.5.3 函数介绍

以下是本模块实现中最重要的一些函数，包括Index Manager中的一些基本函数，B+树中的一些操作函数，以及对B+树结点的一些操作函数。

1、void CreateIndex(string fileName, int dataType);

目的：创建一个新的索引，若之前已经存在相同的索引，则重新构建。

参数： string fileName　索引对应的文件名（包括文件路径）

int dataType　　索引类型

思路：根据dataType判别索引类型后，为该索引创建一棵新的B+树。

2、void DropIndex(string fileName, int dataType)

目的：删除一个已经存在的索引。

参数： string fileName　　索引对应的文件名（包括文件路径）

int dataType　　索引类型

思路：根据dataType判别索引类型后，删除该索引所对应的B+树，并在对应的map中利用erase()函数删除该索引。

3、int SearchIndex(string fileName, string value, int dataType);

目的：在索引中查找某索引值。

参数： string fileName　　索引对应的文件名（包括文件路径）

string value　　想要查找的索引值（字符串形式）

int dataType　　索引类型

返回值：某索引在索引文件中的地址信息，未找到返回-1。

思路：根据dataType判别索引类型后，将value转化成对应类型的数据，并借助B+树的查找操作寻找索引值。

4、void InsertIndex(string fileName, string value, int blockOffset, int dataType);

目的：在索引中插入某索引值。

参数： string fileName　　索引对应的文件名（包括文件路径）

string value　　想要插入的索引值（字符串形式）

int blockOffset　　索引值对应的记录地址信息

int dataType　　索引类型

思路：根据dataType判别索引类型后，将value转化成对应类型的数据，并借助B+树的插入操作来插入索引值。

5、void DeleteIndex(string fileName, string value, int dataType);

目的：在索引中删除某索引值。

参数： string fileName　索引对应的文件名（包括文件路径）

string value　想要删除的索引值（字符串形式）

int dataType　索引类型

思路：根据dataType判别索引类型后，将value转化成对应类型的数据，并借助B+树的插入操作来插入索引值。

6、IndexManager();

目的：构造函数，创建索引。

参数：无

思路：通过调用CreateIndex函数来为给定的索引信息创建索引。

7、~IndexManager()

目的：析构函数，析构的同时将所有修改过的索引写回索引文件。

参数：无

思路：分别对map<string, BPlusTree* >，map<string, BPlusTree* >，map<string, BPlusTree* >类型的三种容器析构索引。同时调用WritteBackToDiskAll将各索引写回对应的索引文件。

8、void BuildTree()

目的：创建一棵B+树（只有根结点）

参数：无

思路：利用new实现对根结点的创建，同时初始化层数，结点数，关键字数等信息。

9、void DropTree(BNode* curNode)

目的：删除一棵B+树，包括内存释放

参数：BNode* curNode　记录当前要删除的B+树结点

思路：利用递归的方法，自上而下实现对B+树的删除操作。

10、void FindNode(Type keyvalue, NodeInfo &nodeSearch)

目的：在B+树中寻找某一关键字值的位置信息，并将相关信息存储在参数nodeSearch中。

参数： type keyvalue　想要寻找的关键字的值

NodeInfo& nodeSearch　存储该关键字在B+树中的位置信息，包括最终进入的叶子结点，在叶子结点中的位置，能否找到该关键字。

思路：从根结点出发，通过不断比较目标值与当前结点各关键字值，确定下一步要进入的（孩子）结点。由此不断循环，直到最终进入到叶子结点确定关键字的最终位置。

11、int SearchKey(Type keyvalue);

目的：在B+树中寻找某一索引值所对应的在索引文件中的地址信息。

参数： type KeyValue 目标索引值

返回值： -1 未找到目标索引值

>0 对应的地址信息

思路：借助FindNode函数找到新关键字在叶子结点中应存在的位置，若不存在返回-1，若存在进一步返回相应地址信息即可。

12、bool InsertKey(Type& keyvalue, int vals)

目的：在B+树中插入一个新关键字。

参数： type KeyValue 想要插入的关键字的值

int vals 新插入的索引值在索引文件中的地址信息

返回值： true 插入成功

false 插入失败

思路：借助FindNode函数找到新关键字在叶子结点中应插入的位置，若该关键字已经存在，则返回false，否则在该位置插入新关键字。若当前结点的关键字数量等于degree，则调用InsertAdjust()函数对B+树进行调整。

13、void InsertAdjust(BNode* curNode)

目的：对完成插入后的B+树结构进行调整

参数：BNode* curNode 第一个调整的结点

思路：根据之前介绍的B+树插入后调整方法自下而上调整B+树的结构。

14、bool DeleteKey(Type& keyvalue)

目的：在B+树中删除一个关键字

参数： type KeyValue 想要删除的关键字的值

返回值： true 删除成功

false 删除失败

思路：借助FindNode函数找到要删除关键字在叶子结点中的位置，若该关键字不存在，则返回false，否则在该位置删除关键字。值得注意的是，若该关键字恰好是当前叶子结点的第一个关键字且当前叶子结点不是第一个叶子结点，需要在父辈结点中进行相应的关键字调整。此外，若当前结点关键字数量小于(degree-1)/2，则调用DeleteAdjust()函数对B+树进行调整。

15、void DeleteAdjust(BNode* curNode)

目的：对完成删除后的B+树结构进行调整。

参数：BNode* curNode　第一个调整的结点

思路：根据之前介绍的B+树删除后调整方法自下而上调整B+树的结构。

16、void ReadFromDiskAll(blockNode* btmp);

目的：读取索引信息。

参数： blockNode* btmp　　bufferManager定义的内存块

思路：借助BufferManager，按block块读取索引文件中的索引值，借助InsertKey()函数将其存储至B+树中。

17、void WriteBackToDiskAll();

目的：将更改过的索引值重新写回索引文件。

参数：无

思路：借助BufferManager，将存储于B+树中全部索引值回写到索引文件中。

18、void FindKey(Type keyvalue, KeyInfo &keySearch);

目的：在当前结点中寻找某一关键字值应存放的位置。

参数： type KeyValue　　想要寻找的关键字的值

KeyInfo& keySearch　存放关键字的查询信息，包括是否存在以及　　应存在的位置

思路：依次遍历当前结点各个关键字的值，KeyValue应存在的位置i满足keys[i]≤keyvalue<keys[i+1]。

19、bool AddKey(Type &keyvalue, Node* child);

bool AddKey(Type &keyvalue, int val);

思路：在当前结点中添加一个新的关键字

参数： type &KeyValue　想要添加的关键字的值

Node* child　　对应要添加的孩子结点

int val　　添加索引值在索引文件中对应的地址信息

返回值：true　　添加成功

false　　添加失败

思路：首先，对于两个函数而言，前者适用于在内部结点中添加新关键字，后者适用于在叶子结点中添加新关键字。添加位置的寻找通过调用FindKey函数来实现，若发现要添加的关键字已存在则插入失败。另外，对于内部结点的添加而言，孩子结点的添加位置位于新关键字的右孩子处。

20、bool RemoveKey(Type &keyvalue);

思路：在当前结点中删除一个关键字

参数： type& KeyValue　想要删除的关键字的值

返回值： true　删除成功

false　删除失败

思路：首先，借助FindNode函数找到想要删除的关键字的存在位置，若不存在则返回false，删除失败。其次对于叶子结点而言，还要删除相应的索引值地址信息，而对于内部结点而言，删除该关键字的右孩子。

21、Node* Split();

思路：将当前结点分裂成两个结点

参数：无

返回值：新分裂产生的结点

思路：当一个结点所含的关键字数量等于degree时，需要对结点进行分裂。分裂分两种情况，对于叶子结点的分裂，将前(degree+1)/2个关键字分配给分裂出来的左结点，将剩余(degree-1)/2个关键字分配给右结点。而对于内部结点而言，将前(degree-1)/2个关键字分配给分裂出来的左结点，将后(degree-1)/2个关键字分配给右结点。需要注意的是，在执行分裂前，需记录原结点中间第(degree+1)/2个关键字的值，方便其向上传递给父亲结点。

3.5.4 结构体说明

```
/*
1、存储索引信息
*/
class IndexInfo{
string indexName;        //索引名称
string tableName;        //数据表名称
string attribute;
int type;                //索引类型
}
/*
2、Index Manager中对所有索引按照变量类型进行分类管理
*/
typedef map<string, BPlusTree<int>* > intBTreeMap;
typedef map<string, BPlusTree<float>* > floatBTreeMap;
typedef map<string, BPlusTree<string>* > stringBTreeMap;
class IndexManager{
    intBTreeMap indexInt;            //int类型索引
    floatBTreeMap indexFloat;        //float类型索引
    stringBTreeMap indexString;      //string类型索引
    int static _INT = Attribute::TYPE_INT;
    float static _FLOAT = Attribute::TYPE_FLOAT;
    API *api;
}
/*
3、B+树中的结点，存放结点中的关键字，孩子结点等信息
*/
template <class Type>
class Node {
    int keycount;                    //键值数量
    int degree;                  //度(或阶数)
    bool leaf;                   //是否为叶结点
```

```
    Node* parent;                //父结点
    Node* next;                  //下一个叶子结点（叶结点特有）
    vector <Node*> childs;       //孩子结点（内部结点特有）
    vector <Type> keys;          //关键字
    vector <int> vals;           //索引值在索引文件地址信息(叶结点特有)
}
/*
4、在当前结点中寻找某关键字时，存储有无找到和应存在位置等信息
*/
struct KeyInfo{
    int position;       //目标关键字应存在的位置
    bool ifExist;       //目标关键字是否存在
};
/*
5、存储索引所用的B+树类
*/
template <class Type>
class BPlusTree {
    typedef Node<Type> BNode;
    struct NodeInfo{
        BNode* n;
        int position;
        bool ifExist;
    };
    BNode* root;             //根结点
    BNode* firstLeaf;        //第一个叶子结点
    int degree;              //B+树的度（阶数）
    int level;               //层数
    int keycount;                //键值总数
    int nodecount;           //结点总数
    int keysize;                 //键值大小
    string fileName;
    fileNode* file;
}
/*
其中，在B+树类中包含这样一个结构
*/
typedef Node<Type> BNode;
struct NodeInfo{
    BNode* n;            //目标关键字应出现的叶子结点
    int position;            //目标关键字应出现的位置
        bool ifExist;            //目标关键字是否已经存在
};
```

其中，struct NodeInfo适用于在B+树中寻找某一关键字的值时，存储应出现的叶子结点，在叶子结点中应存在的位置以及该关键字是否存在等信息。

3.5.5 主要功能

1、首先，Index Manager负责对所有索引的分类管理，按照索引类型(int、float或char(n))构建三个map容器来存储所有索引。在其构造函数中，将执行创建索引的操作。

2、创建索引：判别将要创建的索引类型后，在相应的map容器中创建索引。创建索引将通过构造B+树的形式来实现。

3、删除索引：判别将要删除的索引类型后，在相应的map容器中找到相应的索引，先释放构建的B+树内存，再用erase()函数删除即可。

4、插入索引值：判别要插入的索引值类型，将其转化为相应类型的变量后，在相应的map容器中找到相应的索引，在其对应的B+树中执行InsertKey操作即可。

5、删除索引值：判别要删除的索引值类型，将其转化为相应类型的变量后，在相应的map容器中找到相应的索引，在其对应的B+树中执行DeleteKey操作即可。

6、搜索索引值：判别要搜索的索引值类型，将其转化为相应类型的变量后，在相应的map容器中找到相应的索引，在其对应的B+树中执行SearchKey操作即可。

7、Index Manager的析构：在对Index Manager进行析构时，在释放Index Manager中三个map容器的同时，还要将每个索引对应的B+树的索引值回写到相应的索引文件中。

3.6 *Buffer Manager*

3.6.1 总体设计思路

记录管理模块（Record Manager）和索引管理模块（Index Manager）向缓冲区管理申请所要的数据，缓冲区管理器首先在缓冲区中查看数据是否存在，若存在，直接返回，否则，从磁盘中将数据读入缓冲区，然后返回。

最近最少使用(LRU)算法：用一个链表记录所有的缓冲块，每次访问到一个缓冲块就将它插入到链表的尾部，如果访问过已经插入的缓冲块就让这个缓冲块有一个标记reference,这样从链表头开始遍历，第一个没有被reference并且没有被锁住的缓冲块就是最近最少使用的块，在需要的时候就可以替换出去。

注意，在这一次替换后，下一次开始遍历的位置就不是链表头，而是改成了当前位置，用一个全局的变量replaced_block来标记，以这样的方式就可以实现LRU的替换。

3.6.2 具体实现

1. **宏**

```
#define MAX_FILE_NUM 40
#define MAX_BLOCK_NUM 300
#define MAX_FILE_NAME 100
#define BLOCK_LEN 4096
#define BLOCK_LEN_ENABLE 4088
//The size of the block that other module can use. Others cannot use the block head.
```

2. **2个结构体，1个BufferManager类**

- **文件头结构体(struct fileInfo)：**

记录文件每个节点（Node）几个关键信息（名称，是否被锁定，文件的head block，前一文件，后一文件），作为基础单元，在BufferManager类中以数组形式存在成为一个file_pool

```
struct fileNode
{
    char *fileName;
    bool pin; // the flag that this file is locked
    blockNode *blockHead;
    fileNode * nextFile;
    fileNode * preFile;
};
```

- **块信息结构体(struct blockInfo)：**

记录每个块节点的基础信息，作为BufferManager的友类(friend class)，在buffermanager中以数组形式的block_pool形式存在，提供节点的基础信息，如block在block list中的偏移量，是否上锁，是否位于文件末端，所处的文件名称，内容地址，前后的block指针，用于替换算法LRU的标志reference, 是否被修改过(dirty)，已经使用的大小using_size。

```
struct blockNode
{
    int offsetNum; // the offset number in the block list
    bool pin;  // the flag that this block is locked
    bool ifbottom; // flag that this is the end of the file node
    char* fileName; // the file which the block node belongs to
    friend class BufferManager;

private:
    char *address; // the content address
    blockNode * preBlock;
    blockNode * nextBlock;
    bool reference; // the LRU replacement flag
    bool dirty; // the flag that this block is dirty, which needs to written
back to the disk later
size_t using_size; // the byte size that the block have used. The total size of
the block is BLOCK_LEN . This value is stored in the block head.
};
```

- **BufferManager类(struct blockInfo):**

该类中包含了BufferManager的全部操作，其中，所有与RecordManager和IndexManager的接口全部作为该类的public成员函数。此外，该类中另定义了几类数据结构，一下为该类该类中包含了BufferManager的全部操作，其中，所有与RecordManager和IndexManager的接口全部作为该类的public成员函数。此外，该类中另定义了几类数据结构，一下为该类的部分内容。定义了文件的头结点，文件池，块池，目前块的总数，目前文件的总数等。

```
class BufferManager
{
    private:
        fileNode *fileHead;
        fileNode file_pool[MAX_FILE_NUM];
        blockNode block_pool[MAX_BLOCK_NUM];
        int current_block_num; // the number of block thIat have been used,
which means the block is in the list.
        int current_file_num; // the number of file that have been used, which
means the file is in the list.
}
```

3.6.3 函数功能及其描述

所有函数均在BufferManager类中，分private成员函数与public成员函数（接口）两部分

Private成员函数

1. **void InitBlock(blockNode & block);**

输入：需要初始化的block

返回：空值。

作用：初始化该block，用memset函数给对应的内存赋值，初始化对应的标记如pin,dirty等。

2. **void InitFile(fileNode & file);**

输入：需要初始化的file

输出：空值

作用：初始化file, 用memset函数给对应的内存赋值，初始化各参数。

3. **void SetPin(blockNode & block, bool pin);**

输入：操作的block，对应的pin指令

输出：空值

作用：设置block的pin值将该函数封装在类内，防止外部直接调用，形成一定的独立和保护。

4. **void SetPin(fileNode & file, bool pin);**

输入：操作的file，对应的pin指令

输出：空值

作用：overload函数，与3类似。

Public成员函数

1. **BufferManager();**

作用：构造函数，构造BufferManager对象。在构造函数中给file和block结构体中的成员分配内存空间，调用了初始化函数做相应的初始化操作。

2. **~BufferManager();**

作用：析构函数，释放对应的内存空间

3. **void SetDirty(blockNode & block);**

输入：block

输出：空值

作用：将一个block设置为dirty，每次修改的blockNode的时候必须调用这个函数。

4. **void SetUsingSize(blockNode & block,size_t usage);**

输入：需要修改的block以及对应使用的空间usage

输出：空值

作用：设置当前block占用的空间，无返回值。

5. **size_t GetUsingSize(blockNode & block);**

输入：当前的block

输出：size_t类型的变量。

作用：返回当前block已经使用的大小

6. **fileNode* GetFile(const char* fileName, bool if_pin = false);**

输入：file的名称，是否选择将file锁定，缺省参数默认为否。

输出：对应file的指针。

作用：首先判断输入的file是否在file的list中，用一个指针从头开始遍历fileNode通过指针连成的链表，如果找到了则说明需要的file在list中，直接停止遍历，返回对应的指针即可。若没有找到file，则说明fileNode不在List中，考虑当前list中的file数目，如果filelist为空，则创建filelist, 并且加入当前文件；如果fileNode的数目小于MAX_FILE_NUM，则将当前的文件放置于list的末尾；如果fileNode数目大于最大值，则需要找到一个file作为替换，这个file是没有被锁定（pin）的第一个file，替换时调用WriteToDisk函数将该file的所有block全部写回磁盘。

7. **char* GetContent(blockNode& block);**
8. **blockNode* GetBlock(fileNode * file, blockNode* position, bool if_pin = false);**

输入：希望加入block的file，对应block加入的位置，是否锁定block，缺省参数默认为否。

输出：对应block的指针

作用：如果block已经在list中，则直接返回这个block节点。如果block不在list中，则采用LRU替换算法，从链表头开始遍历，找到第一个没有reference标记并且没有被pin的缓冲块，该缓冲块就是被替换的缓冲块，并且在遍历的过程中把碰到的所有reference的标记清除，用于考虑下一次替换。把输入的block替换进去，并对替换出的block做标记，只有替换出的block是dirty的，才会调用WriteToDisk函数将其写回磁盘。

9. **blockNode* GetNextBlock(fileNode * file,blockNode* block);**

输入：当前的file和希望加入的block

输出：blockNode的指针

作用：插入当前block如果block已经在list中存在，并且nextBlock的指针非空则返回下一个block，如果block不存在则调用GetBlock函数，插入block并且返回其返回值。

10. **blockNode* GetBlockHead(fileNode* file)**

输入：当前的file

输出：blockNode的指针

作用：如果该文件存在blockhead，则返回，否则调用GetBlock(file,NULL)并且返回对应的blockNode指针。

11. **blockNode* GetBlockByOffset(fileNode* file, int offset);**

输入：当前的file, 偏离量offset

输出：blockNode指针

作用：根据偏移量找block，如果offset为0直接调用GetBlockHead(file)，否则用循环不断调用GetNextBlock(file, btmp)并且让offset递减。直到最终offset为0

12. **void DeleteFileNode(const char * fileName);**

输入：文件名

输出：空值

作用：删除文件以及对应的block值，循环调用GetNextBlock函数，将所有的block压入一个stack，并依次删除，更新current_block_num，最终删除文件，更新current_file_num。

13. **void WriteToDiskAll();**

输入：空值

输出：空值

作用：从FileHead开始，遍历file list中的所有file以及内部的block，将其全部写回磁盘中（调用WriteToDisk函数）。

14. **void WriteToDisk(const char* fileName,blockNode* block);**

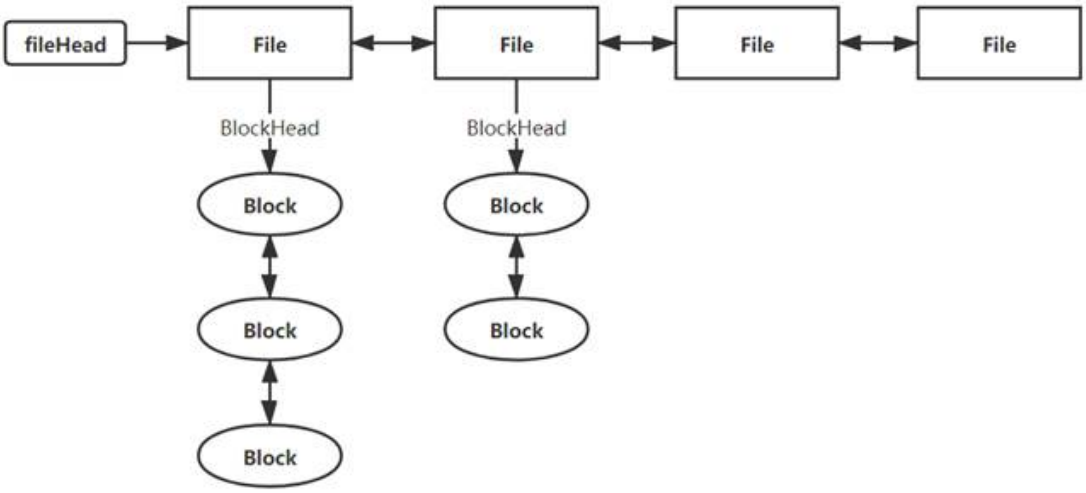
输入：文件名，对应写回磁盘的block

输出：空值

作用：当且仅当block是dirty时才会被写回，打开fileName对应的file,用rb+的方式二进制写入文件。

3.6.4 内存中维护的结构

fileHead与BlockHead的维护的结构



4.错误信息处理

错误信息主要分为两类，一是由*interpreter*模块进行语法分析产生的错误，二是底层模块遇到异常时抛出的错误。

底层错误，使用*Catalog Manager*抛出，由*Interpreter*捕捉，并输出到屏幕，同时返回到主程序进行相应的处理

语法错误，由*Interpreter*对用户输入语句进行分析时产生

- Error, command xx not found
- Syntax Error for xx
- Syntax Error for no table name
- Syntax Error: unknown data type
- Syntax error : illegal number in char()
- Syntax Error: unknown data type
- Syntax Error: unknown or missing data type!
- Syntax Error for ,!
- Syntax Error: primaryKey does not exist in attributes
- Syntax Error: ')' absent!"
- Incomplete definition in attribute xx
- Error definition of primary key in table xx
- Attribute xx Not Found in table xxx
- Index xx Not Found in table xxx
- Error attribute type xx in xx
- Extra parameters in create index

- Not specify index name
- Syntax error: Not specify the insert value
- Empty attribute value in insert value
- Can't find the file
- Other Errors in Syntax

5.系统测试

1. 初始界面

```
    _/      _/      _/      _/      _/      _/        _/_/_/        _/_/          _/
   _/_/  _/_/              _/_/    _/              _/            _/    _/        _/
  _/  _/  _/      _/      _/  _/  _/      _/        _/_/        _/  _/_/        _/
 _/      _/      _/      _/    _/_/      _/            _/      _/    _/        _/
_/      _/      _/      _/      _/      _/      _/_/_/          _/_/  _/      _/_/_/_/_/

************************************************************
Version:1.2
Date: 2020/06/27
By: LI Xiang, WANG ZITeng, PAN KAIHang, QIU HAOZe, YANG Rui
************************************************************
minisql>>
```

2. 创建表

创建成功：

```
minisql>>create table Book(name char(35), price float, stock int, rank int unique, primary key(name));
Create Table Book Successfully
Create Index PRIMARY_Book SuccessfullyThe duration is 0.5 milliseconds
```

错误1-没有用逗号隔开：

```
minisql>>create table Book(name char(35) price float stock int rank int unique primary key(name));
Syntax Error for ,!
The duration is 0.1 milliseconds
```

错误2-使用不被支持的数据类型：

```
minisql>>create table Book(name char(35), price boolean, stock int, rank int unique, primary key(name));
Syntax Error: unknown or missing data type!
The duration is 0.0 milliseconds
```

错误3-重复创建同名表：

```
minisql>>create table Book(storage int, name char(20), rate float, primary key(name));
Table Book Alredy Exists
The duration is 0.1 milliseconds
```

3. 创建、删除索引

创建成功：

```
minisql>>create index POPULARITY on Book(rank);
Create Index POPULARITY SuccessfullyThe duration is 0.5 milliseconds
```

错误-创建on的属性不存在：

```
minisql>>create index NOVEL on Book(rate);
Attribute rate Not Found in table Book
```

删除成功

```
minisql>>drop index POPULARITY on Book;
Drop Index POPULARITY SuccessfullyThe duration is 0.2 milliseconds
```

错误-删除的索引不存在：

```
minisql>>drop index NOVEL;
error
The duration is 0.3 milliseconds
```

4. 向表中插入记录

插入成功（主键相同的记录无法插入）：

```
minisql>>insert into Book values('Little Prince', 15.5, 100, 1);
Insert Into Book Successfully
The duration is 0.2 milliseconds
minisql>>insert into Book values('Little Prince', 15.5, 100, 1);
Fail to Insert : Index Value Exists
The duration is 0.1 milliseconds
minisql>>insert into Book values('C Primer Plus', 45.5, 66, 3);
Insert Into Book Successfully
The duration is 0.4 milliseconds
minisql>>insert into Book values('Dark Forest', 50.5, 35, 4);
Insert Into Book Successfully
The duration is 0.2 milliseconds
minisql>>insert into Book values('Death End', 25.5, 16, 2);
Insert Into Book Successfully
The duration is 0.2 milliseconds
minisql>>insert into Book values('Three Body', 40.0, 18, 5);
Insert Into Book Successfully
The duration is 0.2 milliseconds
minisql>>insert into Book values('San guo', 16.0, 20, 6);
Insert Into Book Successfully
The duration is 0.2 milliseconds
minisql>>insert into Book values('shui hu', 16.0, 24, 7);
Insert Into Book Successfully
The duration is 0.2 milliseconds
```

5. 查询记录

查询表中所有记录：

```
minisql>>select * from Book;
name
price
stock
rank

Little Prince 15.500000 100 1
C Primer Plus 45.500000 66 3
Dark Forest 50.500000 35 4
Death End 25.500000 16 2
Three Body 40.000000 18 5
San guo 16.000000 20 6
shui hu 16.000000 24 7
7 Records Selected
The duration is 0.2 milliseconds
```

单值查询：

```
minisql>>select name from Book where price = 16;
name

San guo
shui hu
2 Records Selected
The duration is 0.3 milliseconds
```

复合查询：

```
minisql>>select name from Book where price = 16 and rank = 7;
name

shui hu
1 Records Selected
The duration is 0.3 milliseconds
```

区间查询：

```
minisql>>select name from Book where rank >= 2 and rank <= 5;
name

C Primer Plus
Dark Forest
Death End
Three Body
4 Records Selected
The duration is 0.2 milliseconds
```

6. 删除记录

```
minisql>>delete from Book where stock <> 66;
Delete 6 Records in Table Book
The duration is 0.2 milliseconds
minisql>>select * from Book;
name
price
stock
rank

C Primer Plus 45.500000 66 3
1 Records Selected
The duration is 0.2 milliseconds
```

7. 删除表

```
minisql>>drop table Book;
PRIMARY_Book
Drop Index PRIMARY_Book SuccessfullyDrop Table Book Successfully
The duration is 0.4 milliseconds
```

6.分工

- **李想** Buffermanager模块设计及报告，Catalog模块设计，系统整合部分工作
- **求昊泽** RecordManager模块设计及报告，测试演示视频录制
- **王子腾** API模块设计及报告，系统整合部分工作
- **潘凯航** IndexManager模块设计及报告
- **杨锐** Interpreter模块设计及报告，报告整合