



数据库系统实验报告

作业名称: MiniSQL

姓 名: 曾 充 张海川 刘隽良

学 号:

电子邮箱:

联系电话:

指导教师: 孙建伶

2020年6月21日

一、实验目的

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

二、系统需求

2.1 需求概述

数据类型

只要求支持三种基本数据类型：int、char(n)、float。

其中char(n)满足 $1 \leq n \leq 255$ 。

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 unique 属性的主键定义。

索引的建立和删除

对于表的主键自动建立B+树索引，对于声明为unique 的属性可以通过SQL 语句由用户指定建立、删除B+树索引。

因此，所有的B+树索引都是单属性单值的。

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。（where 条件是范围时删除多条）

2.2 语法说明

MinisQL 支持标准的SQL 语句格式，每一条SQL 语句以分号结尾，一条SQL 语句可写在一行或多行。为简化编程，要求所有的关键字都为小写。在以下语句的语法说明中，用黑体显示的部分表示语句中的原始字符串，如create 就严格的表示字符串“create”，其他非黑体显示的有其他的含义，如 表名 并不是表示字符串 “表名”，而是表示表的名称。

创建表语句

该语句的语法如下：

```
create table 表名 (
    列名 类型 ,
    列名 类型 ,
    列名 类型 ,
    primary key ( 列名 )
);
```

其中类型的说明见第二节“功能需求”。

若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

删除表语句

该语句的语法如下：

```
drop table 表名 ;
```

若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

创建索引语句

该语句的语法如下：

```
create index 索引名 on 表名 ( 列名 );
```

若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

删除索引语句

该语句的语法如下：

```
drop index 索引名 ;
```

若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

选择语句

该语句的语法如下：

```
select * from 表名 ;
```

或

```
select * from 表名 where 条件 ;
```

其中“条件”具有以下格式：列 op 值 and 列 op 值 ... and 列 op 值

op 是算术比较符：`= < > < > <= >=`

若该语句执行成功且查询结果不为空，则按行输出查询结果，第一行为属性名，其余每一行表示一条记录；若查询结果为空，则输出信息告诉用户查询结果为空；若失败，必须告诉用户失败的原因。

插入记录语句

该语句的语法如下：

```
insert into 表名 values ( 值1 , 值2 , ... , 值n );
```

若该语句执行成功，则输出执行成功信息；若失败，必须告诉用户失败的原因。

删除记录语句

该语句的语法如下：

```
delete from 表名 ;
```

或

```
delete from 表名 where 条件 ;
```

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

退出MiniSQL 系统语句

该语句的语法如下：

```
quit;
```

执行SQL 脚本文件语句

该语句的语法如下：

```
execfile 文件名 ;
```

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

关于返回信息

返回信息的内容参考mysql，语言限定为英文，主要包括执行sql 返回的行数(n row in set), 或者影响的行数(n row affects), 必须给出执行SQL 所花费的时间Duration。

执行出错需要指出错误的类型，具体位置不做要求。错误类型包括(SYNTAX ERROR, INVALID IDENTIFIER, INVALID VALUE , INVALID ATTR FOR INDEX), 即语法错误(语法参考mysql, miniSQL 不支持的数据类型和SQL 等也报语法错误), 标识符错误(不存在的表名, 字段名等), 值错误(变量长度不足以容纳实际数据), 索引定义在非unique 属性上。可以参考mysql workbench 和mysql terminal client 的返回信息。

三、实验环境

- 开发语言: C++
- 交互界面: 基于Web的GUI前端界面
- 系统要求: 跨平台

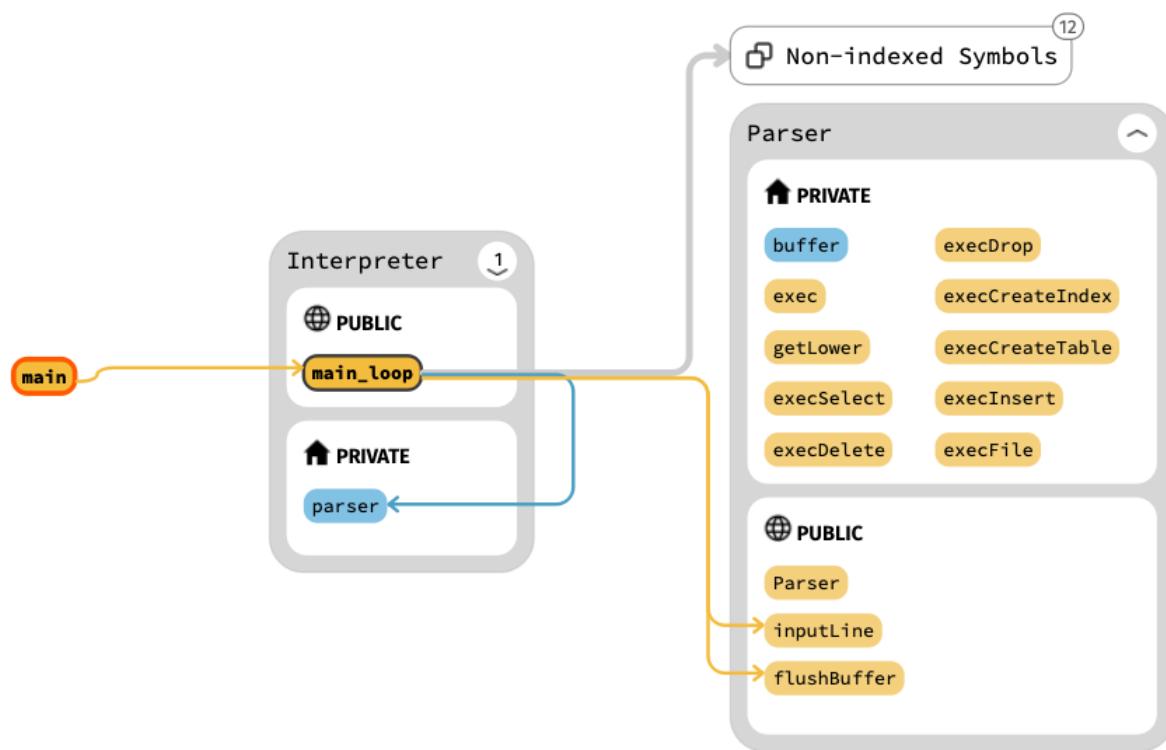
四、系统设计

4.1 分工情况

1. Interpreter, API, Catalog Manager 曾充
2. Index Manager, B+ Tree 张海川
3. Record Manager, Buffer Manager 刘隽良

4.2 Interpreter

4.2.1 程序执行流程图



与所有C++程序一样，程序入口是 `main()`，`main()` 中唯一的函数就是Interpreter中的 `main_loop()` 主循环。图中可以看出 `main_loop()` 中含有一个 `parser`，`parser` 对输入的语句进行解析之后，调用相关的接口进行命令的执行。

4.2.2 主要数据结构

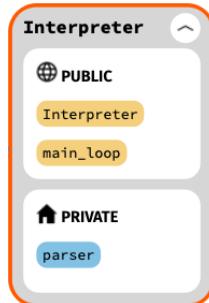
Interpreter类中最为重要的数据结构就是一个buffer缓冲队列。考虑到有多行输入，一行多条语句的需求，我设计了一个buffer队列，将输入的每一行指令分词后，把所有的词和标点插入队列中，并在检测到 ; 后将分号前的分词全部pop掉，放入一个vector给后续的执行器进行执行。

其中，buffer队列直接使用STL中的双端队列 `deque` 进行实现，因为存在检索 `;` 的需求，使用双端队列能提高遍历的效率。

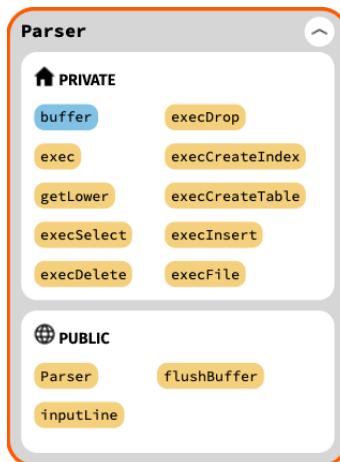
而把分词传给exec系列执行函数的时候，是通过 `vector` 进行实现的，因为在函数内经常需要乱序访问，此时使用向量更为合适。

4.2.3 类图及类内关系

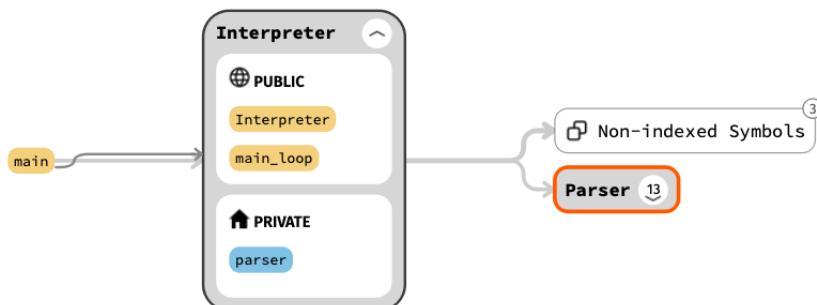
(1) Interpreter 类



(2) Parser类



(3) 类内关系



Interpreter类内有一个Parser对象成员，用于解析指令。

4.2.4 SQL错误检测

错误检测主要在两个地方进行，一个是在 `Parser::exec()` 函数内，对输入的指令进行预处理，匹配到更加具体的执行器中，如果匹配失败，则会报错提示 `SYNTAX ERROR`，例如：

```
MinisQL> sle * form xxx;
[Error] SYNTAX ERROR: You have an error in your SQL syntax, command sle not found!
```

第二处错误检测位于每一个具体的执行函数之内，针对每一种命令进行单独的解析之后，调用API的接口进行执行。当解析失败时，便会检测出错误，例如：

```
MinisQL> select * from xyz where ;
[Error] Error! Table xyz not found!
MinisQL> select * form b;
[Error] SYNTAX ERROR: You have an error in your SQL syntax, select command needs a table name!
```

其他地方的错误检测与这两处完全一致，故不再赘述。

4.3 API

4.3.1 功能描述

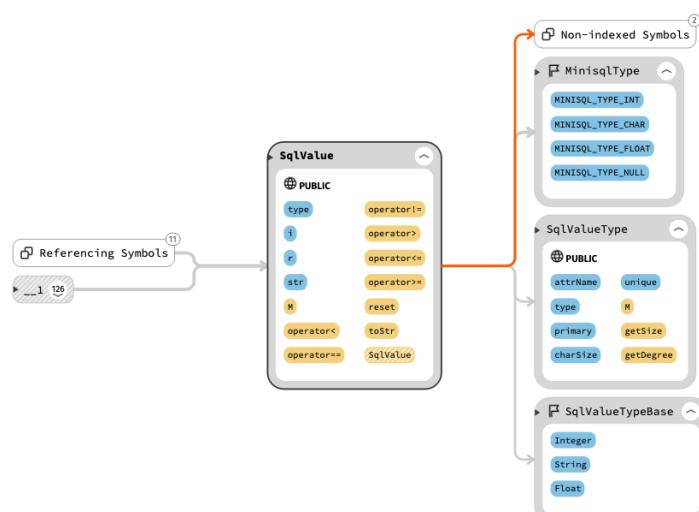
功能上，API模块是对底层的Manager提供的各种接口进行调用，来完成相应的操作，同时将操作封装为统一的接口，供Interpreter调用。该接口以 Interpreter 层解释生成的命令内部表示为输入，根据 Catalog Manager 提供的信息确定执行规则，并调用Record Manager、Index Manager和Catalog Manager 提供的相应接口进行执行，最后返回执行结果给 Interpreter 模块。

其中每一个接口的具体作用在前面的接口设计中以及陈述，此处不再赘述。

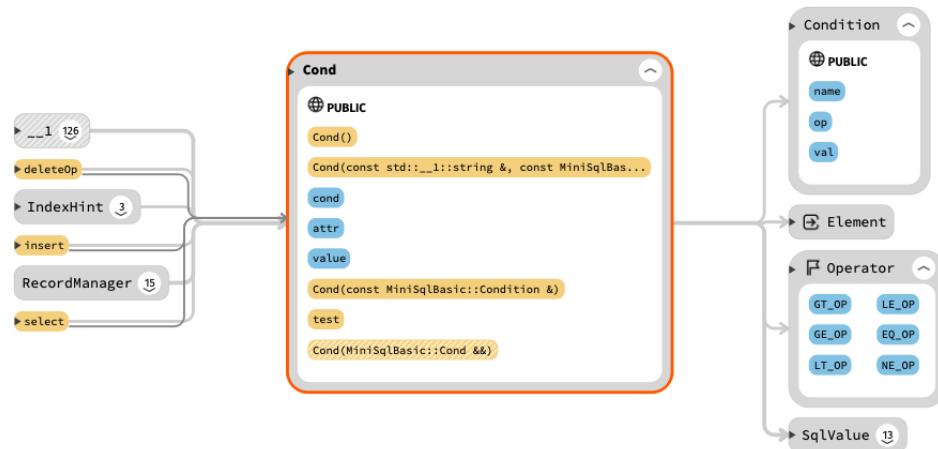
4.3.2 主要数据结构

API模块中，主要的数据结构为SqlValue和Cond。

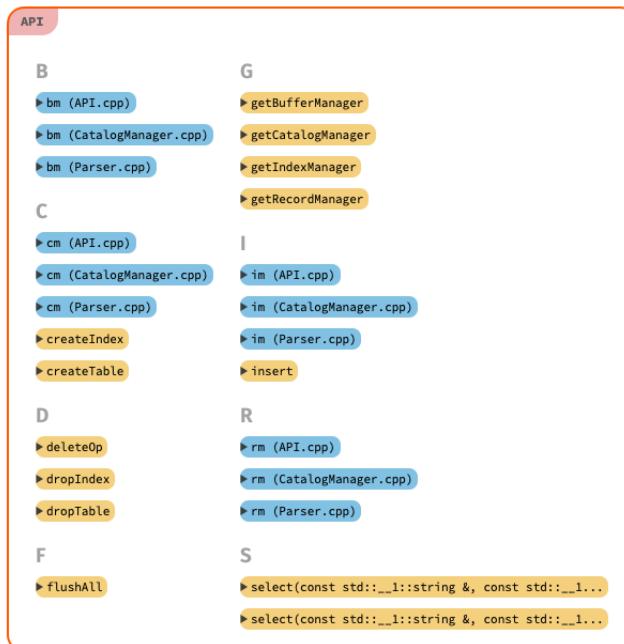
其中SqlValue主要用于储存所有的值，通过在struct中定义一些枚举类型，来标记每一个Value的数据类型，并储存其值，具体的设计在下图中呈现。



另一个重要的数据结构是Cond，它是每一个查询条件的内部描述。内部定义了属性名，条件，以及参考值，比如 `a <> 123` 会被储存为属性a、不等于、123，将序列化的信息结构化，便于后期的处理。



4.3.3 类图及类内关系



事实上，API的接口全部是放在命名空间中的，而不是放进一个类，因为API本身应该设计为一个无状态的存在，引用透明，所以最终并没有设计一个类。但是为了防止命名冲突，也为了编码上的统一，最终使用了命名空间来解决了这个问题。

4.4 Catalog Manager

4.4.1 功能描述

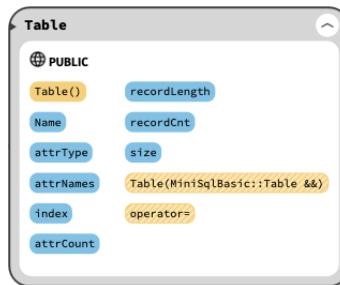
Catalog Manager 负责管理数据库的所有模式信息，包括：

1. 数据库中所有表的定义信息，包括表的名称、表中字段（列）数、主键、定义在该表上的索引。
2. 表中每个字段的定义信息，包括字段类型、是否唯一等。
3. 数据库中所有索引的定义，包括所属表、索引建立在那个字段上等。

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。

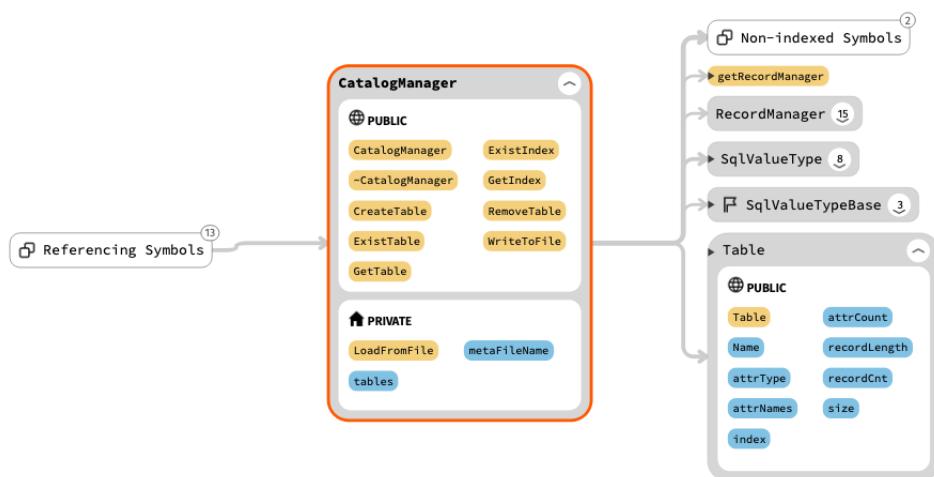
4.4.2 主要数据结构

Catalog Manager主要是储存表的定义信息，因此此处最重要的数据结构就是Table的定义。



其中的 `attrType`、`attrName`、`index` 均为向量，用来描述每一个属性的类型和名称，以及每一个索引的信息。`attrCount`、`recordLength`、`recordCnt`、`size` 等用来记录属性个数、记录长度、记录个数、大小等信息，便于写文件时用到。

4.4.3 类图及类内关系



Catalog Manager类对外提供了创建表、检测表存在、获得表、创建索引、检测索引、删除表、写文件等接口，内部维护了读文件的功能，并且保存了表定义的元数据。

4.5 Index Manager

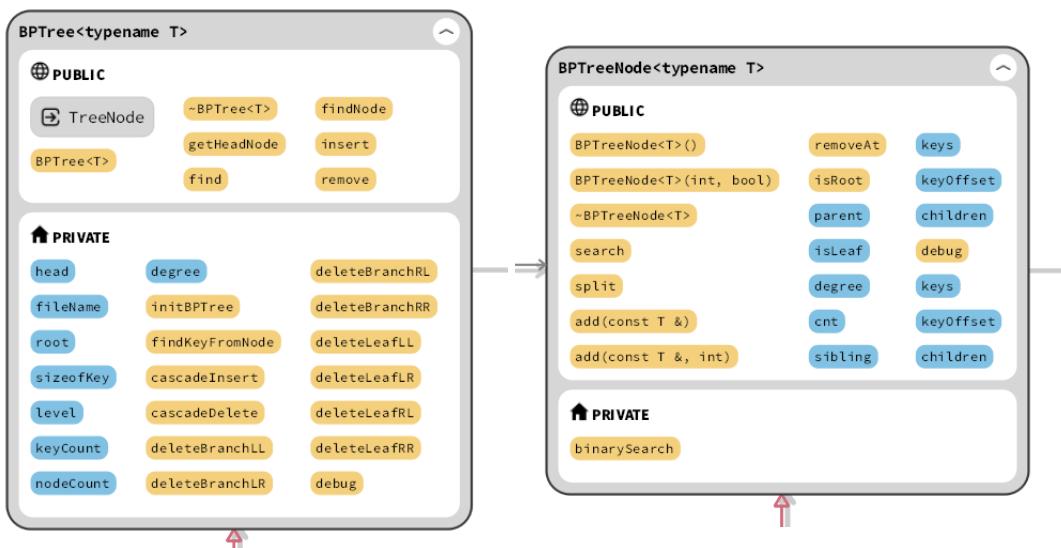
4.5.1 功能描述

Index Manager 负责管理数据库中的索引，也即索引的创建、删除、以及对索引内容进行插入、查找、删除。

Index Manager 同时提供接口，供 API、Catalog Manager、Parser 使用。

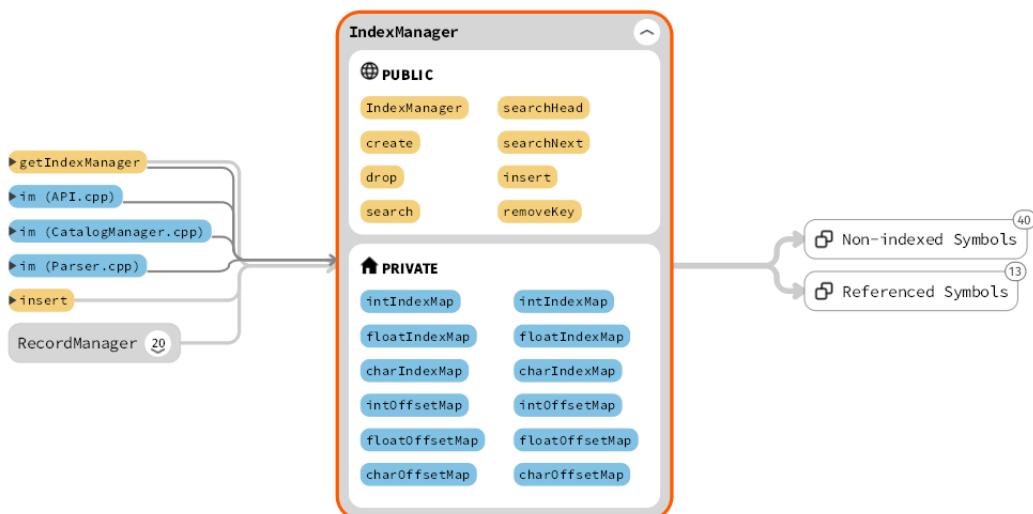
4.5.2 主要数据结构

Index Manager 的核心是B+树，即代码中 `BPTree` 和 `BPTreeNode` 类：



这两个类中的字段基本按照 B+ 树的概念和原理进行实现，在此不多赘述。

4.5.3 类图及类内关系



Index Manager 内部使用 `std::map` 保存名称到索引的对应关系，各个方法实现上主要是调用 `BPTree` 的各个方法。

4.6 Record Manager

4.6.1 功能描述

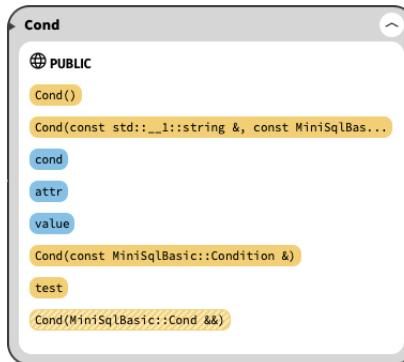
Record Manager最核心的功能在于直接执行对records/tuples的操作。记录层面的操作在这模块内被实现，对Catalog Manager提供必要的接口。具体功能包括：

- select：实现对给定表、条件下对记录的查询并做相应显示。并且当选择条件中包括含索引属性时应能够通过Index Manager对于查询过程进行优化。
- delete：与select类似，对给定表名和条件进行删除操作。
- insert：给定表名、属性值列表后将记录插入至Buffer Manager。

4.6.2 主要数据结构

Record Manager中使用到的主要数据结构包括条件**Cond**、记录**Tuple**的定义。

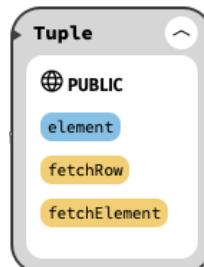
- **Cond**



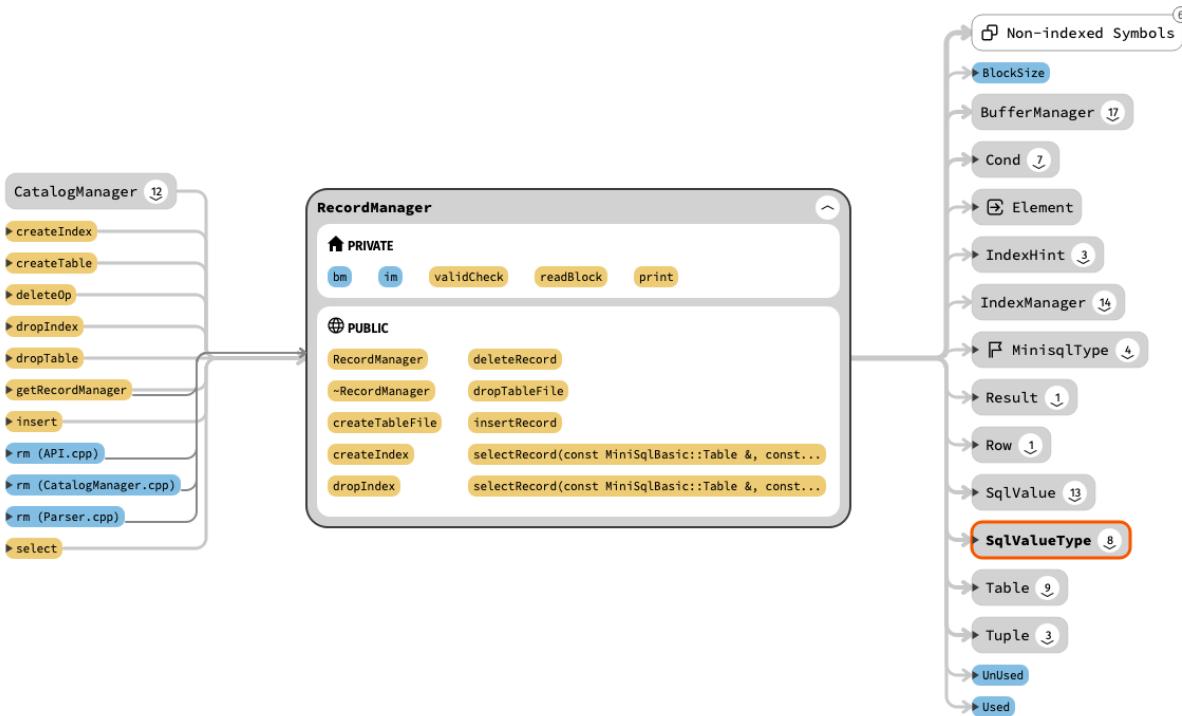
其中cond表示运算符类型，attr表示作用属性，value表示值。test接受一个`MiniSqlBasic::Element`实例并返回该实例与value的逻辑运算结果。

- **Tuple**

其中Element是一个`MiniSqlBasic::Element`列表，用于存放一条记录。fetchRow返回一个适合输出的数据结构，fetchElement返回记录中特定属性名的值。



4.6.3 类图及类内关系



Record Manager类对外提供了对记录进行插入、删除、查找的接口。并因为其作用于API和Buffer Manager之间，在整个工程中处于较为中间的层次，也因为本身功能相近，所以还集成了一些功能简单的接口如创建表、index相关文件的接口供更高层模块调用。

4.7 Buffer Manager

4.7.1 功能描述

Buffer Manager维护一个in memory buffer(size: 128, 可调节)，缓存表文件中的相关记录，实现按块读取(size: 4096B, 可调节)。Buffer Manager位于整个工程的最底层，直接与文件系统进行交互。对 API、Record Manager提供创建文件、访问文件、访问相关记录、操作Buffer Block的相关接口。

4.7.2 主要数据结构

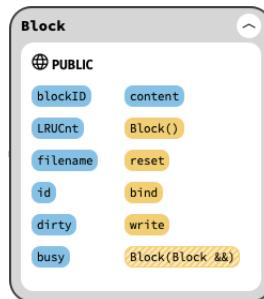
Buffer Manager通过blockBuffer数组在内存中维护一个 $\text{BlockSize} \times \text{MaxBlocks}$ 的空间用于进行缓存。同时，还维护一个Block信息到blockBuffer数组中index的映射方便进行高效访问。Buffer Manager所处理的主要请求与相关思路包括：

- 给定表名和记录id，返回相关Block内容。该功能可通过id与RecordSize计算出BlockID即目标Block在文件中的偏移量。再根据表名和BlockID在blockMap中进行查找。Buffer的替换采用LRU策略：如有则hit一次，使用计数增加，返回Block中内容；如未找到则将LRU写回到邮件并根据Block信息在文件中读取相关内容。
- 返回空Block。当有insert、create等请求发生时，则先寻找是否有空闲位置，如有则返回，如果没有则在相关文件后写入新的Block。

- 对通过Block的busy、dirty位的维护实现内容读写控制。可避免脏读或事冲突的发生。
- 对已有文件的维护，如对文件权限、大小的控制等。可通过`stat()`等接口对权限、大小进行控制。
- 其余与文件系统相关的简单功能如创建表、索引文件等。将这些功能加入该模块以使层级更明确，不相关功能隔离度更高。

4.7.3 类图及类内关系

最主要的数据结构即对Block的定义。



其中，filename和blockID可以唯一确定一个Block的位置，blockID表示了该Block在文件\$(filename)中的偏移量(以Block记)。id录了该block在blockBuffer中的序号，LRUCnt维护了buffer中该block的hit次数。busy、dirty位分别记录该块是否正在被访问、该块数据是否被修改。

4.8 DB Files 存储文件格式说明

4.8.1 Catalog Manager DB Files

Catalog Manager在整个过程中主要会产生`tables.meta`、`tablename.catalog`等DB Files，它们全部是文本文件。

(1) Meta 文件

`tables.meta`文件中储存了全部的表名称。其中第一行保存了数据库中所有表的数量总和，下面的每一行是一个表名，并为了规避不同系统下换行产生的问题，使用了0作为不同表名称的分隔符。例如，在创建了abc和233两张表后，`tables.meta`的文件内容如下：

```

3
abc
0
233
0
student
0

```

(2) Catalog 文件

Catalog文件记录了每一张表的定义，他的构成是 $1 + 7N$ 行，其中第一行储存了表的属性个数 N 。

后面跟着的 $7N$ 行，储存了 N 个属性的定义，每个属性占7行。第一行储存了属性的名称，第二行给出属性的类型，第三行给出属性的大小，其中 `char` 类型标记最大大小，而 `int` 和 `float` 类型直接记 0，后面跟着3行0、1，分别代表该属性是否为主键、是否唯一、以及是否包含索引。最后一行给出该属性上的索引名称，如果没有就写 `-`。

具体的例子可以见下，当使用如下的SQL语句建立了 student 表，并在 `sname` 属性上创建索引后，得到的 `student.catalog` 文件。

- SQL

```
create table student (
    sno int,
    sname char(233) unique,
    score float,
    primary key (sno));

create index idx on student (sname);
```

- `student.catalog`

```
3
sno
int
0
1
1
1
pri_student_sno
sname
char
233
0
1
1
idx
score
float
0
0
0
0
-
```

五、系统实现分析及运行截图

5.1 指令运行结果与分析

5.1.1 创建表语句

示例语句

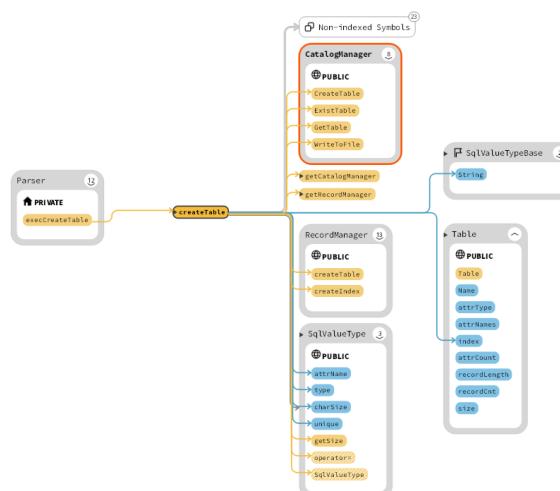
```
create table student2(
    id int,
    name char(12) unique,
    score float,
    primary key(id)
);
```

执行结果

```
MiniSQL> create table student2(
    >         id int,
    >         name char(12) unique,
    >         score float,
    >         primary key(id)
    > );
Create table student2 success.
(0.002081505 s)
MiniSQL> █
```

流程分析

整体流程如图所示：



输入`create`语句后，首先由Interpreter进行最初的解析，分解得到表名、表定义、主键等信息，并转化为内部定义的数据结构。之后，调用API中的 `createTable()` 方法进行执行。

在API中，Insert操作的第一步就是检查所要插入的那张表是否存在，为此需要通过`getCatalogManager()`获得一个Catalog Manager的实例，然后查找、获取这张表。之后，对照表定义依次检查各个value是否符合表定义，此外还需要检测每一个被标记为唯一的属性是否发生了冲突，包括主键和声明了unique的属性。最后，调用Index Manager和Record Manager中的接口进行最终的插入。

其中Record manager会进一步调用Buffer manager创建表记录的文件，而Catalog Manager则是直接修改内存中的表定义，等待下一次flush的时候写入硬盘。

5.1.2 删除表语句

示例语句

```
drop table student2;
```

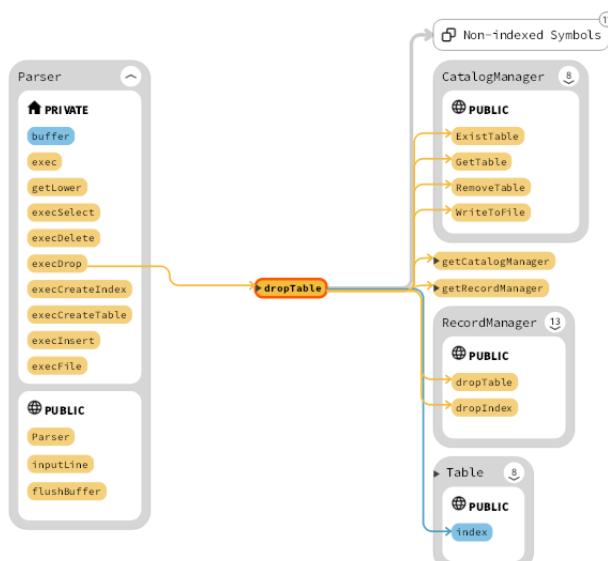
执行结果

```
MinisQL> drop table student2;
Table student2 dropped.
(0.002824240 s)
MinisQL> █
```

流程分析

删除表语句在输入后，首先由解析器分析出表名称，然后调用API中的`dropTable()`方法执行命令。

`dropTable()`会首先检查表是否存在，如果存在则调用Catalog Manager中的`RemoveTable()`删除这张表在内存中的定义，并删除相关的文件。Record Manager则会通过删除文件的方式删掉表中的所有记录，并且将相关的索引也一并删除。整个流程大致如下：



5.1.3 创建索引语句

示例语句

```
create index sidx on student (name);
```

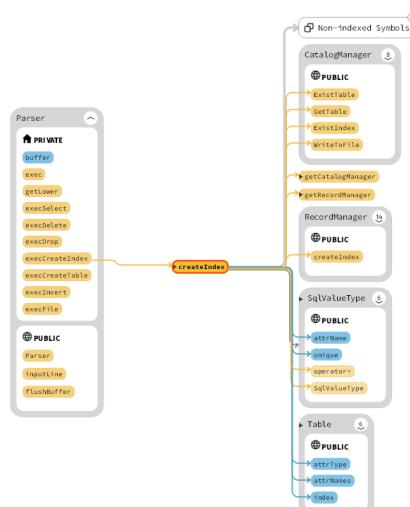
执行结果

```
MiniSQL> create index sidx on student2 (name);
Create index success
(0.001779220 s)
MiniSQL> █
```

流程分析

创建索引的语句输入后，解析器查找出表名和目标属性名，然后调用API中的接口进行执行。API会首先检查表的存在性，以及同名的索引是否已经被创建过。之后，会检查这一属性是否为唯一的(unique)，所有检查结束后，调用Catalog Manager登记索引信息、写入文件，并调用Record Manager创建索引，此时Record Manager会进一步调用Index Manager创建B树。

整体流程如图所示：



5.1.4 删除索引语句

示例语句

```
drop index sidx;
```

执行结果

```
MiniSQL> drop index sidx;
Index sidx dropped.
(0.001497089 s)
MiniSQL> █
```

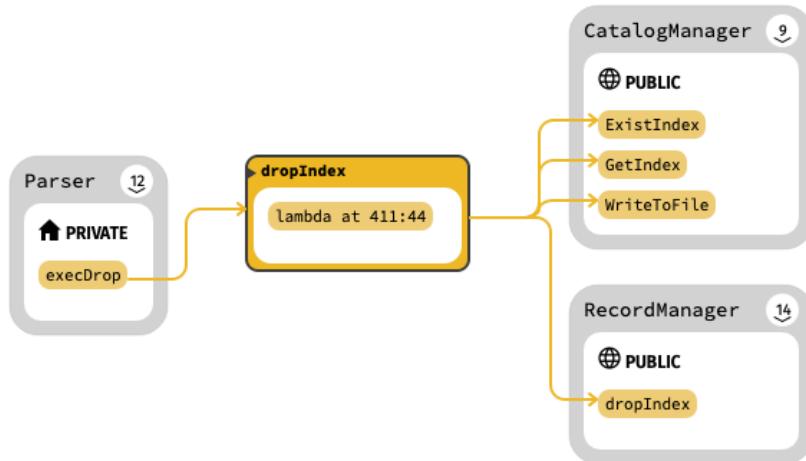
流程分析

当输入drop index指令时，由Parser解析并调用API::dropIndex()。

```
// find and get index
bool index = _cm->ExistIndex(indexName);
if (!index) {
    std::cerr << "Index not found!" << std::endl;
    return false;
}
auto &table = _cm->GetIndex(indexName);

// remove index
for (auto &idx: table.index) {
    if (idx.second == indexName) {
        _rm->dropIndex(table, idx.first);
        table.index.erase(std::find_if(table.index.begin(), table.index.end(),
            [&indexName](const std::pair<std::string, std::string> &item) {return
item.second == indexName;}));
        std::cout << "Index " << indexName << " dropped." << std::endl;
        _cm->WriteToFile();
        return true;
    }
}
```

检查index名不存在时，报错 `Index not found!`；若存在，则通过调用RecordManager掉用 `IndexManager::drop()`，之后再在CatalogManager中删除相关记录。`IndexManager::drop()`则将相应BPTree销毁。整个示意图如下



5.1.5 选择语句

示例语句

```
select * from student2;
```

```
select * from student2 where name <> 'name123' and score > 99.0;
```

```
select * from student2 where id = 1080100978;
```

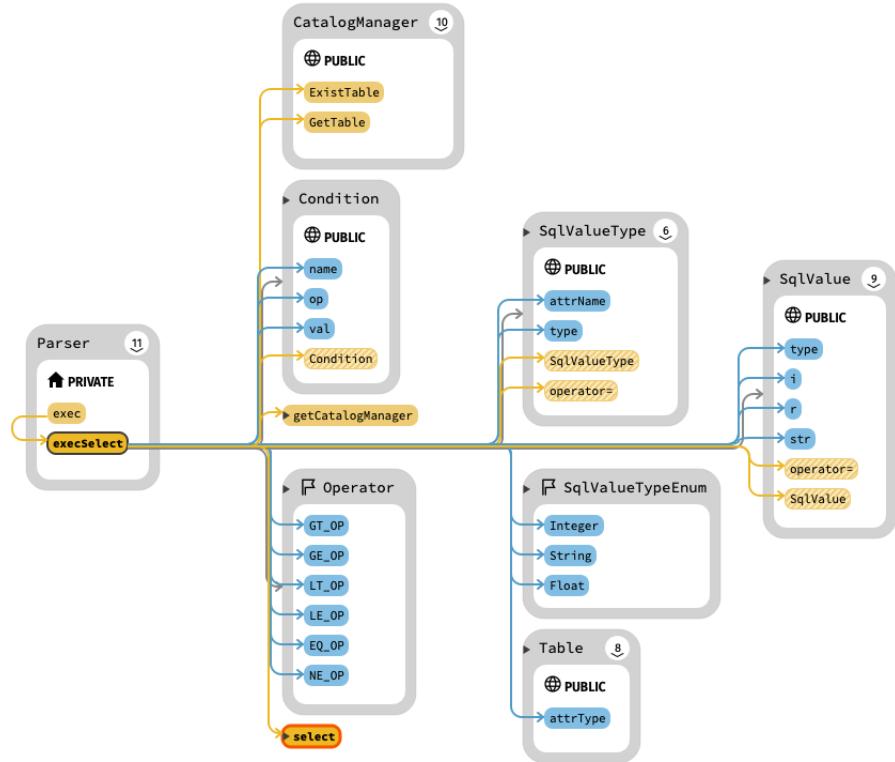
执行结果

| | | | | | | | |
|------------|-----------|-----------|--|------------|----------|-----------|--|
| 1080109970 | name9970 | 55.000000 | | 1080107629 | name7629 | 99.500000 | |
| 1080109971 | name9971 | 66.500000 | | 1080107666 | name7666 | 99.500000 | |
| 1080109972 | name9972 | 85.500000 | | 1080107923 | name7923 | 99.500000 | |
| 1080109973 | name9973 | 76.500000 | | 1080108102 | name8102 | 99.500000 | |
| 1080109974 | name9974 | 52.500000 | | 1080108107 | name8107 | 99.500000 | |
| 1080109975 | name9975 | 97.000000 | | 1080108307 | name8307 | 99.500000 | |
| 1080109976 | name9976 | 50.000000 | | 1080108394 | name8394 | 99.500000 | |
| 1080109977 | name9977 | 56.500000 | | 1080108401 | name8401 | 99.500000 | |
| 1080109978 | name9978 | 84.500000 | | 1080108403 | name8403 | 99.500000 | |
| 1080109979 | name9979 | 99.500000 | | 1080108407 | name8407 | 99.500000 | |
| 1080109980 | name9980 | 88.000000 | | 1080108532 | name8532 | 99.500000 | |
| 1080109981 | name9981 | 78.500000 | | 1080108545 | name8545 | 99.500000 | |
| 1080109982 | name9982 | 53.500000 | | 1080108643 | name8643 | 99.500000 | |
| 1080109983 | name9983 | 58.500000 | | 1080108659 | name8659 | 99.500000 | |
| 1080109984 | name9984 | 64.500000 | | 1080108753 | name8753 | 99.500000 | |
| 1080109985 | name9985 | 71.000000 | | 1080108797 | name8797 | 99.500000 | |
| 1080109986 | name9986 | 61.000000 | | 1080108848 | name8848 | 99.500000 | |
| 1080109987 | name9987 | 91.500000 | | 1080109037 | name9037 | 99.500000 | |
| 1080109988 | name9988 | 75.000000 | | 1080109095 | name9095 | 99.500000 | |
| 1080109989 | name9989 | 58.000000 | | 1080109200 | name9200 | 99.500000 | |
| 1080109990 | name9990 | 75.000000 | | 1080109277 | name9277 | 99.500000 | |
| 1080109991 | name9991 | 69.000000 | | 1080109289 | name9289 | 99.500000 | |
| 1080109992 | name9992 | 50.500000 | | 1080109298 | name9298 | 99.500000 | |
| 1080109993 | name9993 | 67.500000 | | 1080109345 | name9345 | 99.500000 | |
| 1080109994 | name9994 | 97.500000 | | 1080109348 | name9348 | 99.500000 | |
| 1080109995 | name9995 | 59.500000 | | 1080109398 | name9398 | 99.500000 | |
| 1080109996 | name9996 | 65.500000 | | 1080109574 | name9574 | 99.500000 | |
| 1080109997 | name9997 | 61.000000 | | 1080109629 | name9629 | 99.500000 | |
| 1080109998 | name9998 | 84.500000 | | 1080109634 | name9634 | 99.500000 | |
| 1080109999 | name9999 | 69.500000 | | 1080109680 | name9680 | 99.500000 | |
| 1080110000 | name10000 | 80.500000 | | 1080109979 | name9979 | 99.500000 | |

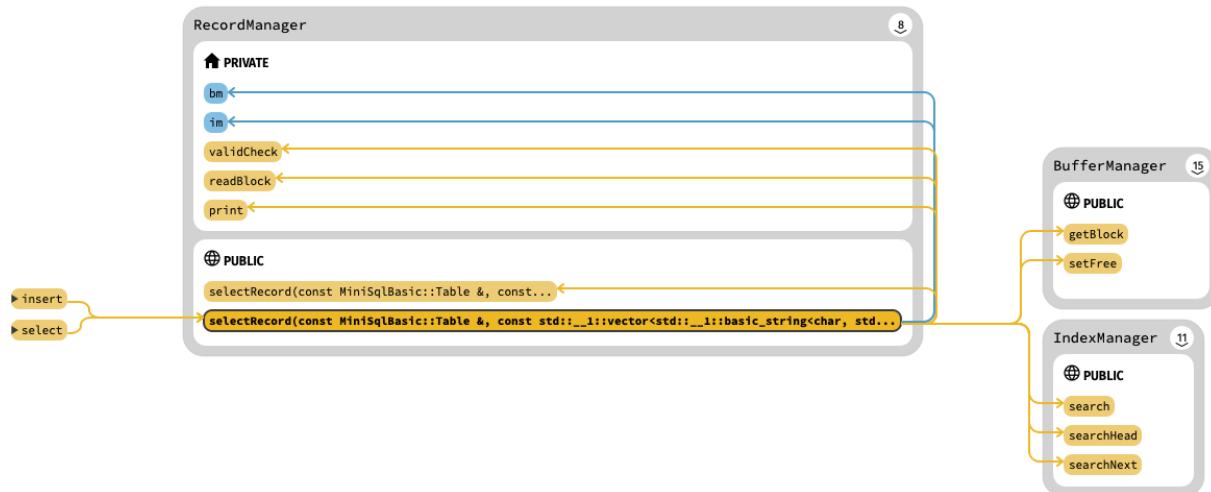
```
MinisQL> select from student2 where id = 1080100978;
| 1080100978 | name978 | 51.500000 |
1 selected.
(0.000186009 s)
MinisQL>
```

流程分析

Parse解析语句后，execSelect被调用，首先对表名、条件等进行检查，如果表名不存在、语法有错误、条件不合法则报错并返回。如果检查通过，则构建相关数据结构并调用API::select()。execSelect调用情况如下



之后API::select根据Catalog中的信息对请求中条件的合法性进一步检查。通过后，RecordManager::selectRecord()根据条件中是否存在带索引属性而被调用。之后，满足条件的记录将被输出。示意图如下，具体流程见4.6.1节。



5.1.6 插入记录语句

示例语句

```

insert into student2 values (1008612315,'name999',59.9);
insert into student2 values (1008612315,'nameABC',59.9);

```

执行结果

```

MiniSQL> insert into student2 values (1008612315, 'name999', 59.9);
Insert finished. 1 row affected.
(0.000458553 s)
MiniSQL> select from student2 where name = 'name999';
| 1080100999 | name999 | 64.500000 |
1 selected.
(0.016420853 s)

```

流程分析

`insert` 命令被解析后，`execInsert` 被调用，首先将检查表名、条件、参数的合法性。如不通过将报出如下错误

```

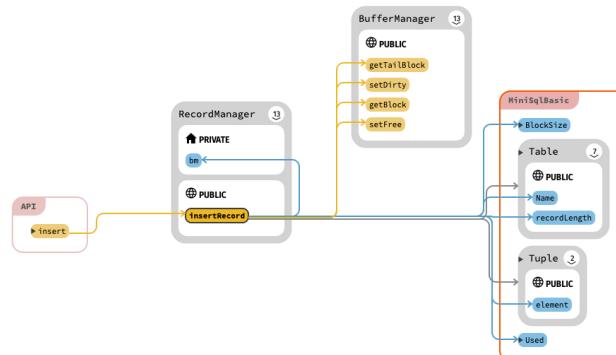
throw std::runtime_error("Table " + tableName + " not found!");
throw std::runtime_error("SYNTAX ERROR: Value type unknown! We only support
char, int, float!");
throw std::runtime_error("SYNTAX ERROR: You have an error in your SQL syntax!
Insert command needs more arguments!");

```

通过检查后，将从命令中解析记录值，构建相关数据结构，包括 `Tuple`、`AttrList` 等数据结构，传给 `API::insert()` 进行处理。API 对 `unique` 属性进行检查通过后，调用 `Record::insert()`，通过 `BufferManager` 中的接口，将值写入表中存在空位的 block，`insert` 操作就完成了。列表如下：

1. 调用 `CatalogManager::ExistTable`，检查表格是否存在，不存在则报错
2. 调用 `API::insert` 进行插入
 1. 检查插入的值是否匹配表中字段的数量和类型，不匹配则报错
 2. 检查表中的每个索引，检查新记录是否满足 `unique` 条件，不满足则报错
 3. 调用 `RecordManager::insertRecord` 插入记录
 4. 对表中的每个索引，调用 `IndexManager::insert` 更新索引

相关调用关系如下



5.1.7 删除记录语句

示例语句

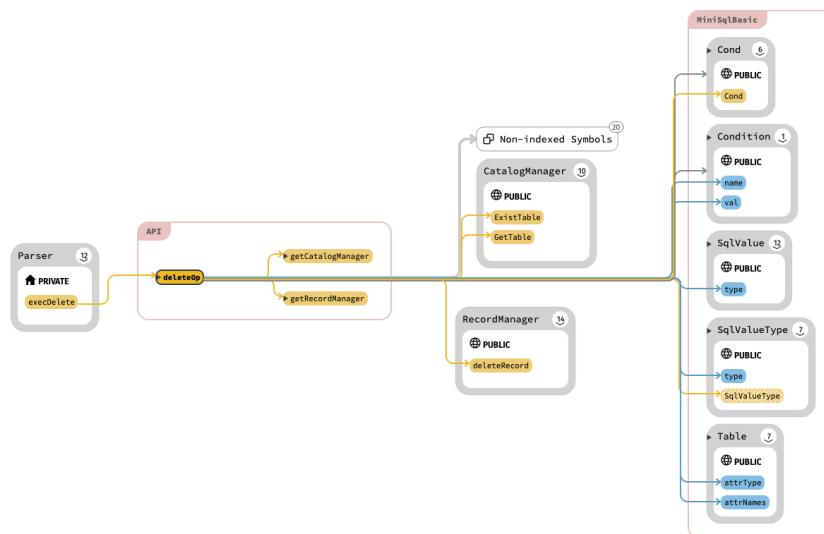
```
delete from student2 where id = 1080100978;
```

执行结果

```
MiniSQL> delete from student2 where id = 1080100978;
Delete success
(0.015484943 s)
MiniSQL> select from student2 where id = 1080100978;
0 selected.
(0.000102827 s)
```

流程分析

1. 调用 `CatalogManager::ExistTable`，检查表格是否存在，不存在则报错
2. 分析查询语句结构，生成 `Condition` 对象用于执行
3. 调用 `API::deleteOp` 进行删除
 1. 将 `Condition` 对象转换为 `Cond` 对象
 2. 调用 `RecordManager::deleteRecord` 删除记录



5.1.8 退出MiniSQL系统语句

示例语句

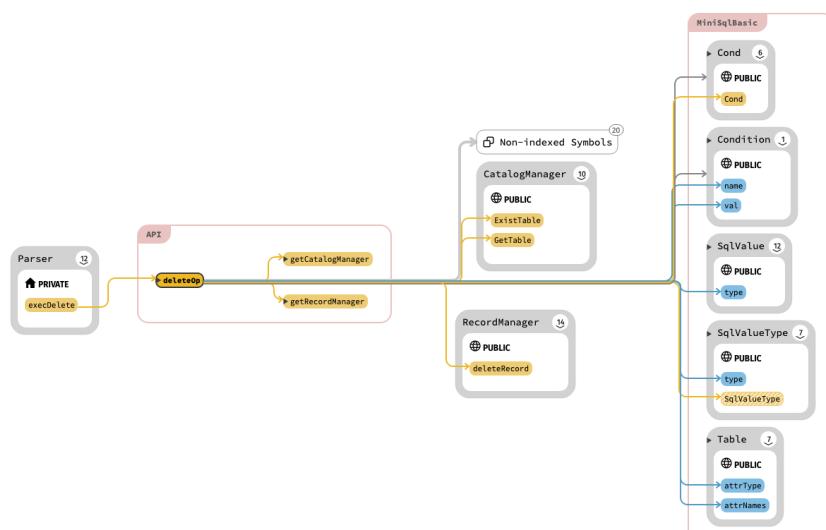
```
quit;
```

执行结果

```
| 1080109992 | name9992 | 50.500000 |
| 1080109993 | name9993 | 67.500000 |
| 1080109994 | name9994 | 97.500000 |
| 1080109995 | name9995 | 59.500000 |
| 1080109996 | name9996 | 65.500000 |
| 1080109997 | name9997 | 61.000000 |
| 1080109998 | name9998 | 84.500000 |
| 1080109999 | name9999 | 69.500000 |
| 1080110000 | name10000 | 80.500000 |
10000 selected.
(0.086039983 s)
MiniSQL> insert into student2 values (1008612315,'name999',59.9);
Insert finished. 1 row affected.
(0.000458553 s)
MiniSQL> select from student2 where name = 'name999';
| 108010999 | name999 | 64.500000 |
1 selected.
(0.016420853 s)
MiniSQL> insert into student2 values (1008612315,'name999',59.9);
Insert failed. Duplicate key!
(0.000142531 s)
MiniSQL> delete from student2 where id = 1080100978;
Delete success
(0.015484943 s)
MiniSQL> select from student2 where id = 1080100978;
0 selected.
(0.000102827 s)
MiniSQL> quit;
Bye!
```

流程分析

退出语句通过调用 `API::flushAll` 实现，该函数则调用 `BufferManager::writeAll`，将所有 dirty block 写入磁盘并退出。



5.1.9 执行SQL脚本文件语句

示例语句

脚本内容

```
insert into student2 values (1190102323, '蒋中文', 88.4);
select from student2 where name = '蒋中文';
create index nameIdx on student2(name);
select from student2 where name = '蒋中文';
```

执行语句

```
execfile example.sql;
```

执行结果

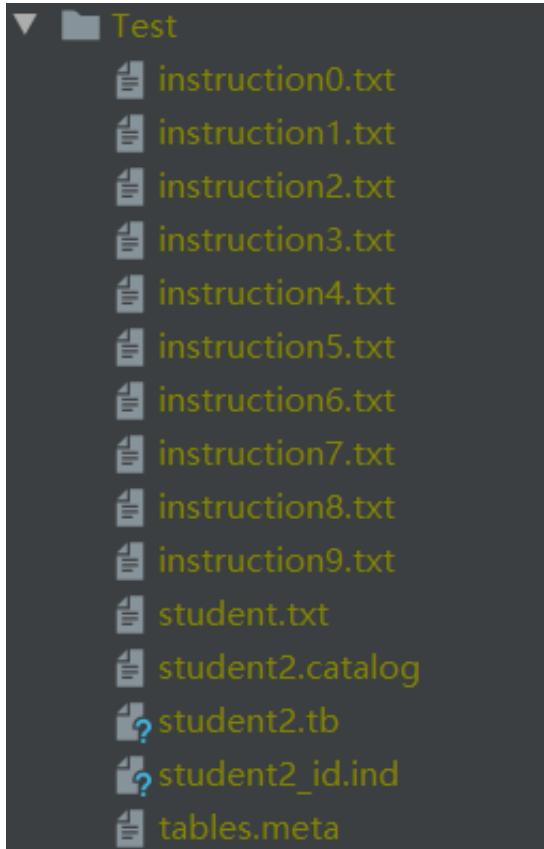
```
MiniSQL> execfile example.sql;
Insert finished. 1 row affected.
(0.000132315 s)
| 1190102323 | 蒋中文 | 88.400002 |
1 selected.
(0.002906134 s)
Not a unique attribute !!
(0.000008022 s)
| 1190102323 | 蒋中文 | 88.400002 |
1 selected.
(0.001794271 s)
MiniSQL> █
```

流程分析

1. 创建一个临时的 `Parser` 对象
2. 不断调用 `Parser::inputLine`, 对文件中的每一行内容进行处理。
3. 输出结果

5.2 文件内容变化

执行验收数据的指令文件，观察文件夹内容的变化：



可以看到，新增的文件有：

- student2.catalog：表 `student2` 的元数据，即字段数量、名称、类型、是否是主键、是否唯一、是否有索引等，格式为文本文件。
- student2.tb：表 `student2` 的数据内容，格式为二进制文件，按照 4K block 进行组织。
- student2_id.ind：表 `student2` 中，索引 `id` 的数据，暂时没有写入内容。
- tables.meta：包含表格数量、表格名称和记录数量，在 Catalog Manager 初始化读入，格式为文本文件。

继续执行指令，创建新表格 `test` 之后，观察 `tables.meta` 文件内容，变为：

```
2
student2
0
test
0
```

可以看到，`test` 表格附加在 `student2` 表之后，说明程序正确读取了 meta 文件。

同时，文件夹中也相应地新增了 `test.catalog`、`test.tb` 和 `test.catalog` 文件，这也符合我们对程序的工作预期。