



|  |  |
| --- | --- |
| **题目：** | **Mini SQL详细设计报告** |
| **姓名：** | **郑无恙 3180101821** |
| **负责模块：** | **Interpreter, API, Catalog Manager, Record Manager** |
| **指导老师：** | **庄越挺** |
| **课程名称：** | **数据库系统** |
| **专业：** | **计算机科学与技术** |

# 一、本人工作

## 1.1 Interpreter

istream& operator>>(istream& is, interpreter& in)

ostream& operator<<(ostream& os, const interpreter& in)

bool querydecode()

void Insert(API &api)

void exec\_file()

void saveblock()

bool If\_int(string word, string& valid)

bool If\_char(string word, int char\_size, string& valid)

bool If\_float(string word, string& valid)

## 1.2 API

所有函数

## 1.3 Catalog Manager

所有函数

## 1.4 Record Manager

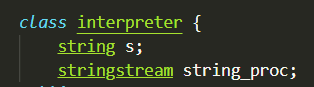
void CreateIndex(string table\_name, int attri\_no, string index\_name)

bool Search\_with\_Index()

insert、select以及delete函数涉及Index Manager的部分

# Interpreter模块

## 2.1 类型声明及实现思路



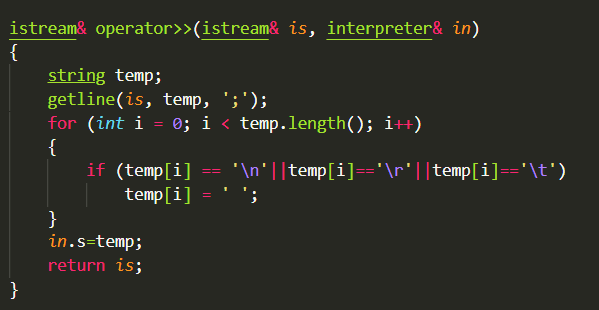
图表 1：interpreter成员变量

interpreter中的成员变量分别是字符串变量s以及字符串流变量string\_proc。s的使用目的是存储处理完成后的sql语句字符串。而string\_proc的功能是处理s中每一个“关键字”的变量。其具体工作方式可通过成员函数进行展示。

interpreter的首要目的是将用户输入的sql语句转换成可供API、datatype以及各manaager进行操作的数据结构（如raw\_data、current\_table等）。接下来将介绍本人在interpreter中实现的函数。

## 2.2成员函数

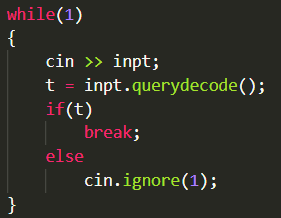
### 2.2.1 operator>>函数



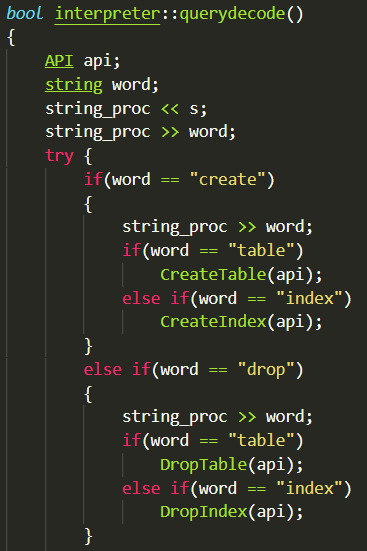
图表 2：operator>>操作符重载函数代码

函数参数共两个变量，其中is变量为当前输入的SQL语句，in变量为interpreter类的对象。首先使用getline函数读取is变量的输入流并存放到temp的字符串里，第三个参数为输入流读取到该字符便停止继续读取（和mysql中的delimiter功能相同）。执行了getline函数后，temp便存放了当前的sql字符串。接下来进入循环，将字符串中回车、换行、以及缩进的字符替换为空格的字符。处理完成后，将temp存放在in变量的s成员变量中。最后返回is变量。

### 2.2.2 querydecode函数

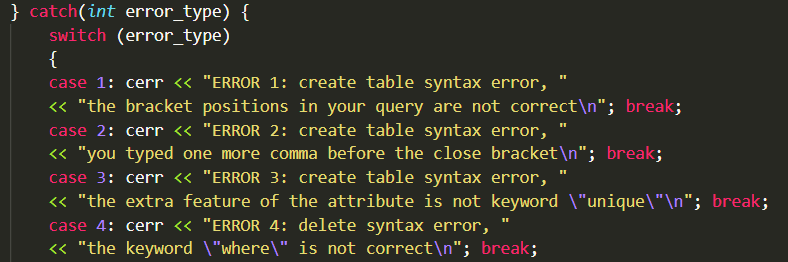


图表 3：main函数中调用querydecode



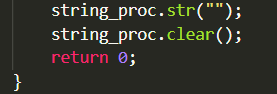
图表 4：querydecode函数部分代码

从图表3可以看到，当用户输入了sql语句后，下一步便执行interpreter类的querydecode函数（图表4）。首先将s读取到string\_proc中，再从string\_proc读出一个“关键字”输出到word中，此时word便等于sql语句的第一个关键字。接下来将执行一系列的字符串比较，检查该关键字是否合法，若合法将进入相应的函数中。



图表 5：querydecode异常接管代码

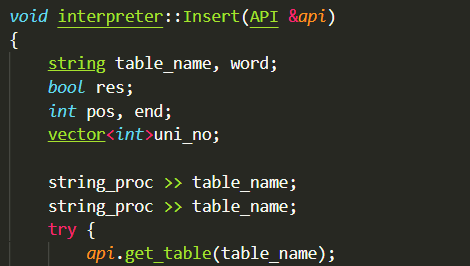
由于在进入字符串判断前使用了try关键字，因此如果调用的函数出现异常，则图表5的代码部分会接管这些异常并根据异常返回的数值输出对应的错误信息。



图表 6：清空string\_proc

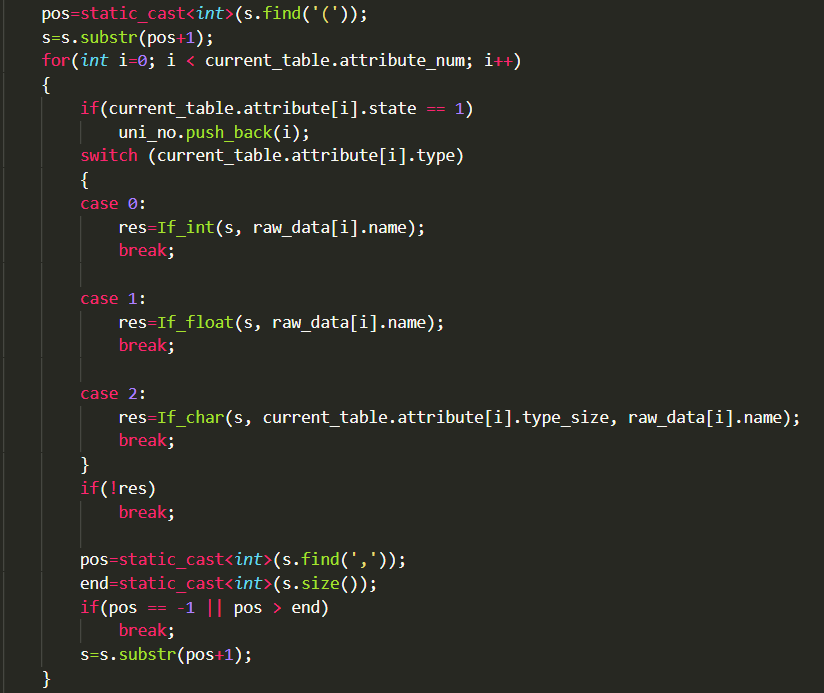
异常处理完成后，对string\_proc内容清空，准备用户的下一轮的输入。

### 2.2.3 Insert函数及If\_char函数



图表 7：Insert函数部分代码1

假如当前进入的是Insert函数，首先可以看到string\_proc输出了两次到table\_name中。这是因为insert语句是按照”insert into xxx”格式输入的，因此这里就是将第二个关键字”into”跳过，这样table\_name变量就能读取表名。下一步调用api的get\_table以及Table\_Tuple\_size函数。其作用为从Buffer Manager中读取该表信息，比如该表的属性数量、每个属性的名称、类型及状态（是否为unique、是否有index）等，接着存放到current\_table变量（于datatype.cpp定义的全局变量，可以被所有Manager调用）中。



图表 8：Insert函数部分代码2

接下来在s中寻找开括号的位置，找到该位置后将s移动到开括号的下一个字符。比如像

”insert into City values ('Kabul','AFG','Kabol',1780000);”这个sql语句。

那么s此时便等于”'Kabul','AFG','Kabol',1780000)”。

下一步进入循环，对每一个即将insert的变量检查其属性是否合法，即根据current\_table的第i个属性进入对应的If\_int、 If\_char及If\_float函数。接下来仅介绍If\_char函数，其他两个函数只是数据类型不同，但实现的效果相同。



图表 9：If\_char函数代码

可以看到word等于s当前的字符串，即

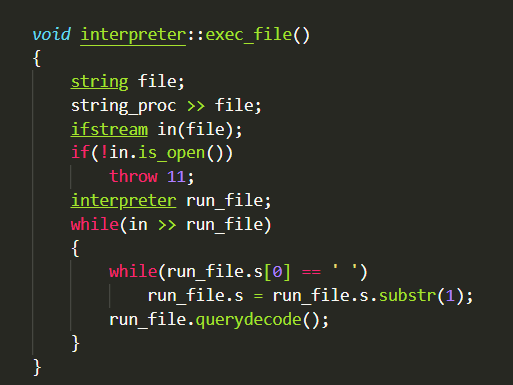
word=”'Kabul','AFG','Kabol',1780000)”

接下来这个函数将word转换为”Kabul”，把原来的单引号消除后，存放到第i个raw\_data的name字符串变量中，并返回true。

但假如word=”1111,'Kabul','AFG','Kabol',1780000)”，那函数会认为这不是一个合法的char数值（第一个字符不是单引号/双引号），于是便返回false。

经过这一轮循环后，interpreter便能检查用户输入的values是否合法，若有不合法的values，interpreter将弹出输入数据不合法（即ERROR 13）的错误信息。若输入数据合法，则调用api将用户输入的数值插入到表中。

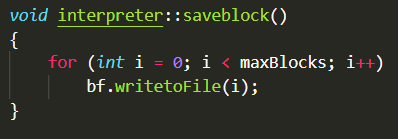
### 2.2.4 exec\_file函数



图表 10：exec\_file函数代码

假如用户输入了”execfile xxx.sql”，那string\_proc会将”xxx.sql”输出到file字符串变量中，接着利用file字符串对in文件流变量进行初始化。下一步判断该文件是否存在，若不存在，则显示文件打开失败的错误信息。然后再定义一个run\_file的interpreter变量，利用上面提及的操作符重载函数读入文件内的sql语句并执行，直至读取完文件所有内容。

### 2.2.5 saveblock函数



图表 11：saveblock函数代码

由于用户当前输入的数据都是存在buffer缓冲区上的，暂时还没有保存到文件上，因此这个函数的作用便是将当前缓冲区的数据写回文件，防止信息丢失。用户可使用save;进入这个函数。

# 三、API模块

## 3.1 类型声明及实现思路

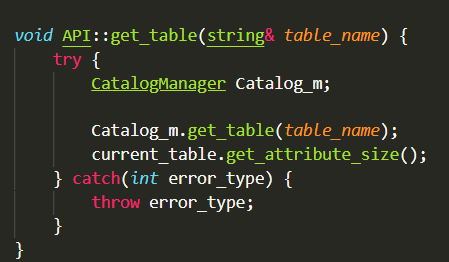


图表 12：API类型声明

API的作用为提供一个interpreter与各manager间的接口，并将manager的异常信息丢回interpreter模块进行处理。接下来对各个成员函数进行介绍。

## 3.2 成员函数

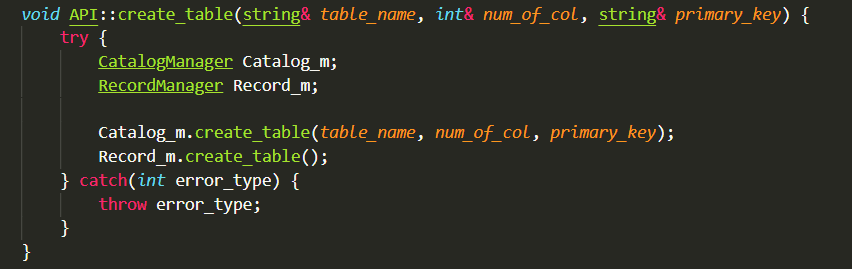
### 3.2.1 get\_table函数



图表 13：get\_table函数代码

负责调用Catalog Manager的get\_table函数，并计算当前表属性所占据的内存空间大小。

### 3.2.2 create\_table函数



图表 14：create\_table函数代码

负责调用Catalog Manager以及Record Manager的create\_table函数。

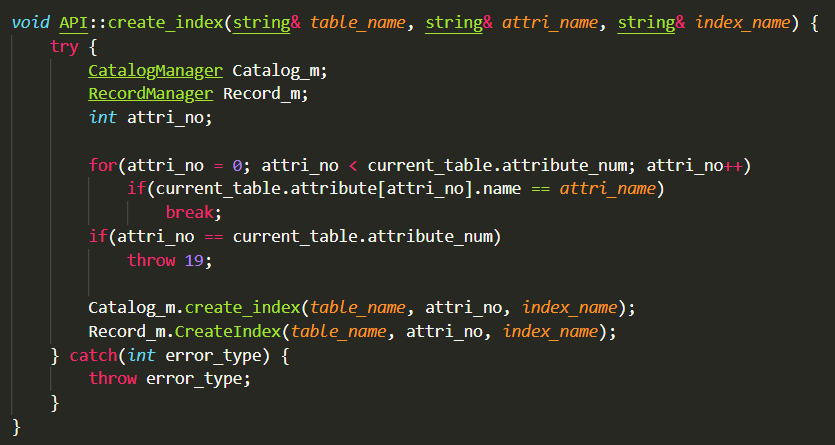
### 3.2.3 drop\_table函数



图表 15：drop\_table函数代码

drop\_table函数首先调用另一成员函数get\_table以获取当前表信息，然后通过遍历当前表的index成员调用Index Manager的Drop函数，删除对应的index及bptree文件。接着调用Catalog Manager以及Record Manager的drop\_table函数删除schema以及tuple文件。最后输出丢弃表成功的信息。

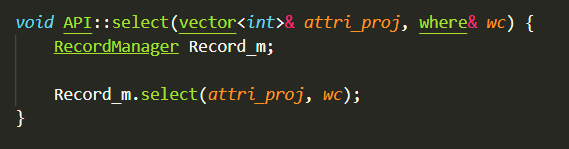
### 3.2.4 create\_index函数



图表 16：create\_index函数代码

首先遍历当前表的属性，查找该表是否存在用户输入的属性名。若不存在则丢出数值为19的异常。如果在当前表查找到该属性名，便调用Catalog Manager以及Record Manager的create\_index函数。

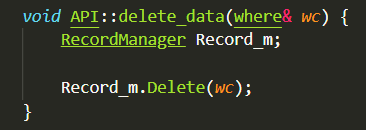
### 3.2.5 select函数



图表 17：select函数代码

负责调用Record Manager的select函数。参数attri\_proj为当前表符合where判断条件的记录中需输出的属性（即投影操作），参数wc为储存where判断条件语句的数据类型。

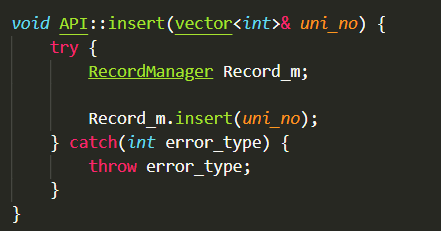
### 3.2.6 delete\_data函数



图表 18：delete\_data函数代码

负责调用Record Manager的Delete函数。

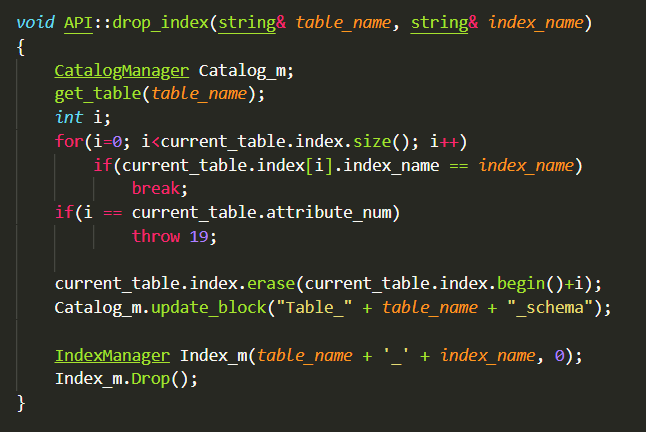
### 3.2.7 insert函数



图表 19：insert函数代码

负责调用Record Manager的insert函数。

### 3.2.8 drop\_index函数

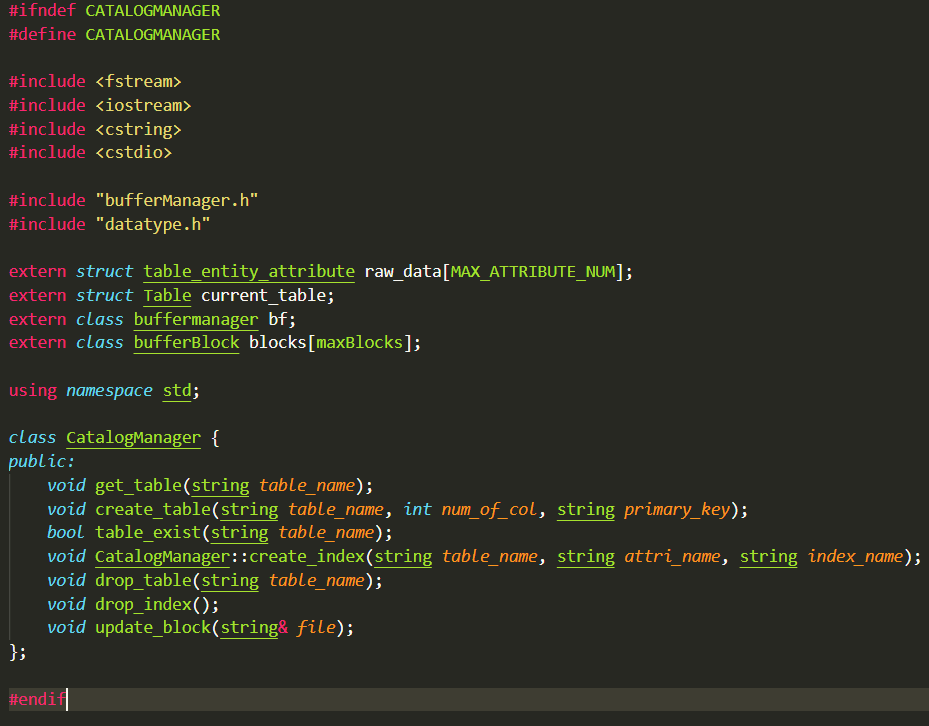


图表 20：drop\_index函数代码

drop\_index函数首先调用另一成员函数get\_table以获取当前表信息，然后寻找该表中是否存在用户输入的index名。若不存在则丢出数值为19的异常；存在则在当前表中移除该index并调用Catalog Manager的update\_block函数更新块，最后再调用Index Manager的Drop函数移除该表对应的index及bptree文件。

# 四、Catalog Manager模块

## 4.1类型声明及实现思路

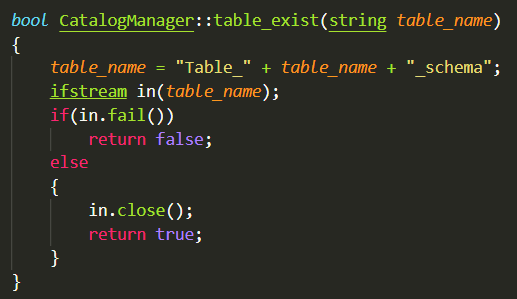


图表 21：CatalogManager类型声明

Catalog Manager函数的主要目的是将当前所使用的表中所有信息输入到名称为Table\_xxx\_schema的文件上，并在下一次使用时从文件读出上一次存储的信息。接下来将对这些函数进行具体介绍。

## 4.2成员函数

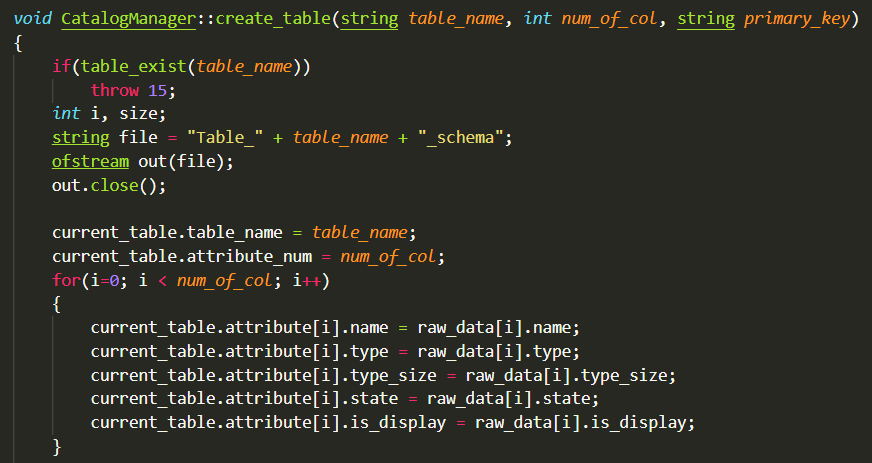
### 4.2.1 table\_exist函数



图表 22：table\_exist函数代码

从图表22可以看到，table\_exist函数通过ifstream测试存储当前所使用表的文件是否存在。若存在，则返回true；不存在则返回false。

### 4.2.2 create\_table函数



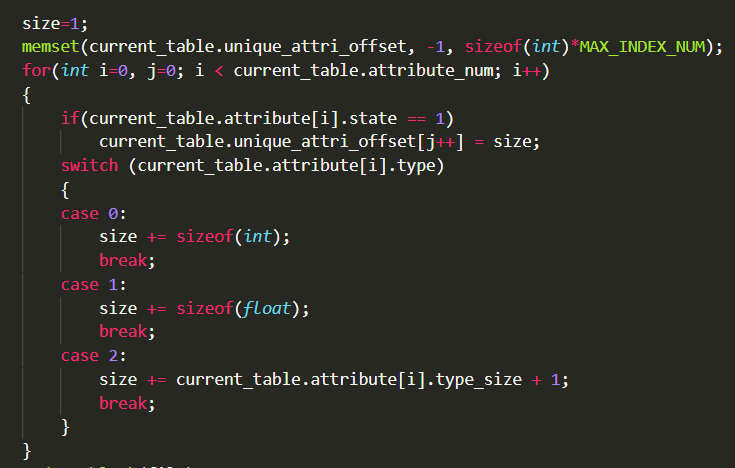
图表 23：create\_table函数部分代码1

函数首先判断该表是否已经存在其对应的schema文件。如果存在，则丢出数值为15的异常；如果不存在，则创建一个格式为”Table\_xxx\_schema”的文件，其中xxx为参数table\_name的字符串。接着将每一个raw\_data（即create table中的每一个属性）复制到current\_table全局变量（当前所使用的表）的attribute成员中。



图表 24：create\_table函数部分代码2

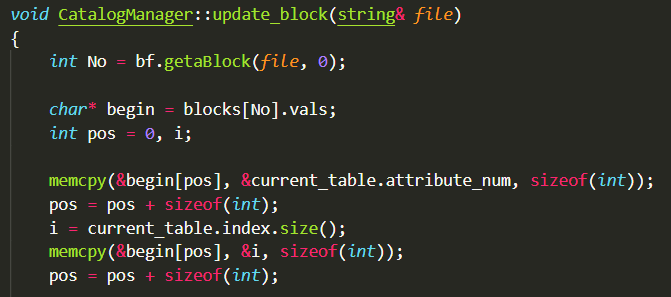
然后判断是否存在primary key。若primary key存在（即其字符串大小为非零），则进入图表24的语句中。首先size被赋值为1，并初始化一个Index类型的变量pri\_index。接着进入循环，遍历当前表的每一个属性。若当前属性名称为primary key，则进入第二个if的语句中，并使用table\_name、primary\_key以及当前属性的类型对Index Manager进行初始化。初始化完成后，再调用CreateIndex函数创建格式为”Table\_xxx\_pri\_yyy\_index”及”Table\_xxx\_pri\_yyy\_bptree”两个文件，其中xxx为当前表名，yyy则是成为primary key的属性名。接着将当前primary key的信息（为当前表的第几个属性、index名称以及该属性的偏移量）保存到当前表中，保存成功后跳出循环。若当前属性名称并非primary key，则计算当前属性实际占用的内存空间并把计算结果加到size中。



图表 25：create\_table函数部分代码3

接下来遍历当前表的属性并找出被定义为unique的属性，计算其内存空间偏移量并保存到当前表的unique\_attri\_offset数组中。接下来进入update\_block函数。

### 4.2.3 update\_block函数



图表 26：update\_block函数代码

首先，调用Buffer Manager的getaBlock函数从buffer中找到一个属于当前表schema文件的块，然后将当前表的数据逐一保存在这个块中，因此使用了大量memcpy函数。最后，调用Buffer Manager的write函数表明当前表的所有数据已写入buffer中。

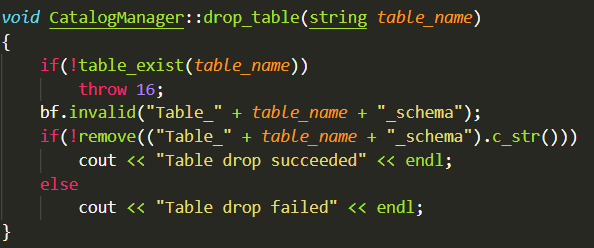
### 4.2.4 get\_table函数



图表 27：get\_table函数代码

可以理解为与update\_block功能相反的函数，即从buffer中寻找某个属于当前表schema文件的块。找到后，将该块的数据读到当前表中。

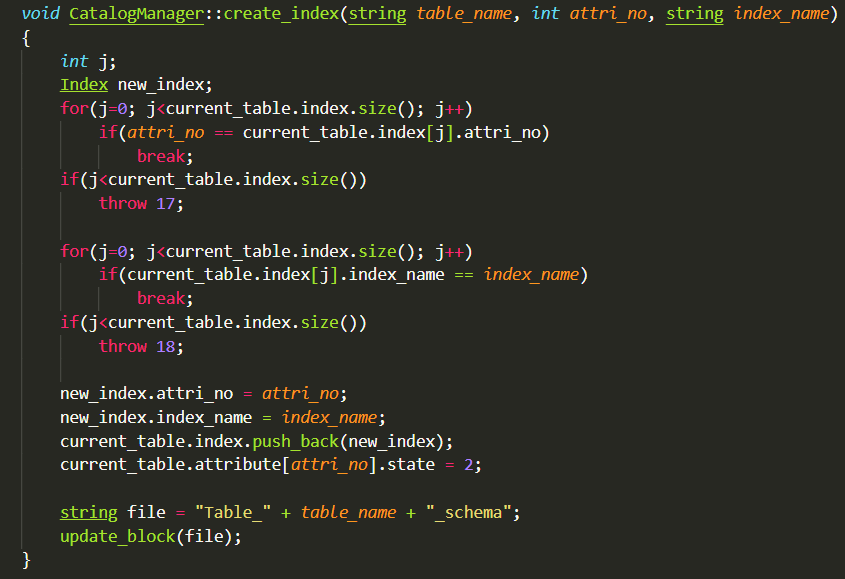
### 4.2.5 drop\_table函数



图表 28：drop\_table函数代码

首先判断待丢弃表的schema文件是否存在，若不存在则丢出数值为16的异常。接下来调用Buffer Manager的invalid函数表明属于这个表schema文件的块已经不会再使用了，可以被重写。然后调用remove函数把当前表的schema文件删除。若删除成功则弹出成功信息；删除失败则弹出失败信息。

### 4.2.6 create\_index函数



图表 29：create\_index函数代码

首先第一个循环判断当前指定创建index的属性是否已经存在一个index，若存在则丢出数值为17的异常。第二个循环判断index名是否已经被使用，若已被使用则丢出数值为18的异常。接下来将index写入当前表中，并更新属于其schema文件的块。

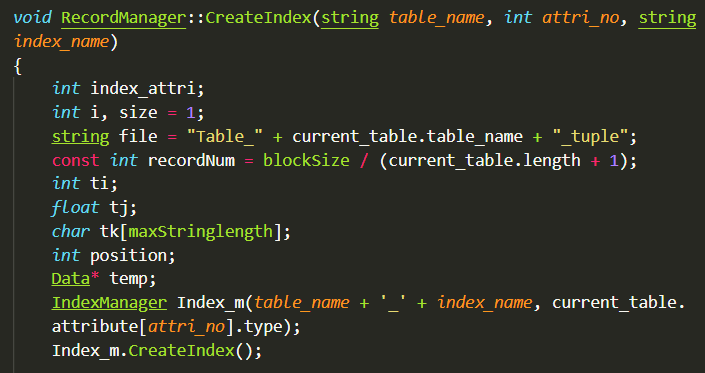
# 五、Record Manager模块

## 5.1实现思路

由于在实际操作上Record Manager与Index Manager有较高的关联性，因此需在Record Manager原有操作的基础上实现index的查找、插入以及删除操作。接下来将介绍本人在Record Manager中实现index操作的函数。

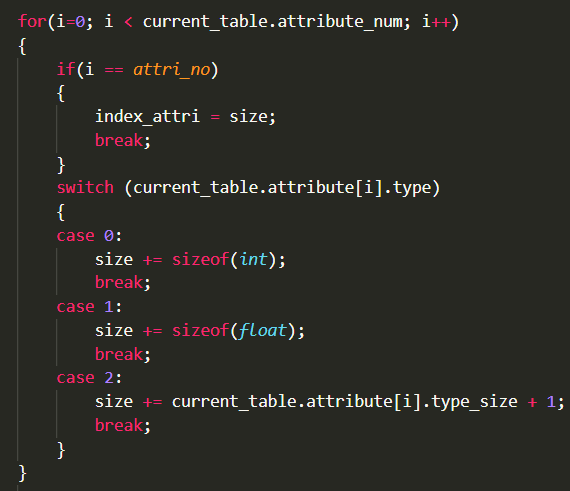
## 5.2 成员函数

### 5.2.1 CreateIndex函数



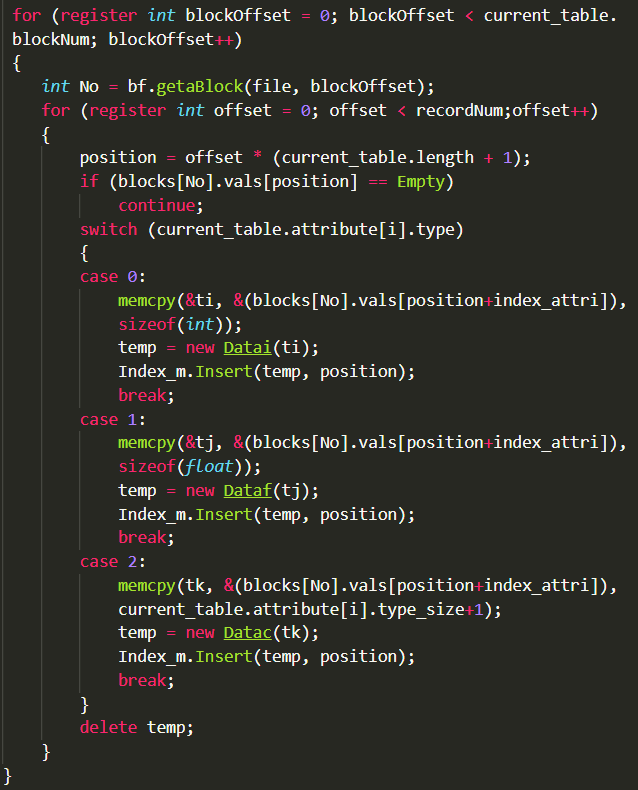
图表 30：CreateIndex函数部分代码1

首先初始化Index Manager，然后调用CreateIndex函数创建对应的index及bptree文件。



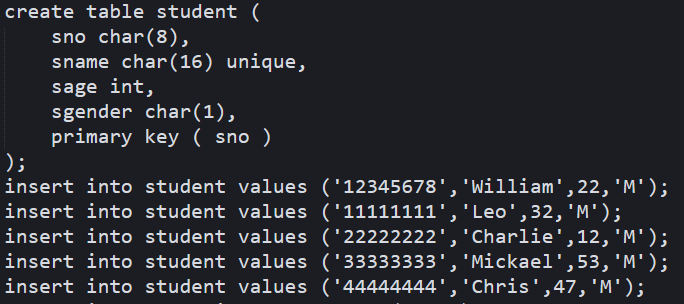
图表 31：CreateIndex函数部分代码2

接下来进入循环计算该index于当前表属性的内存空间偏移量。



图表 32：CreateIndex函数部分代码3

然后进入图表32的双重循环。第一个循环将遍历当前表正在使用的所有块号，接着计算每个记录在块中的偏移量，即position的数值（比如当offset=0时，当前position指向第一个记录；当offset=1时，当前position指向第二个记录）。计算完position后，加上刚才计算出的index偏移量，即可取出当前记录为index属性的数值。

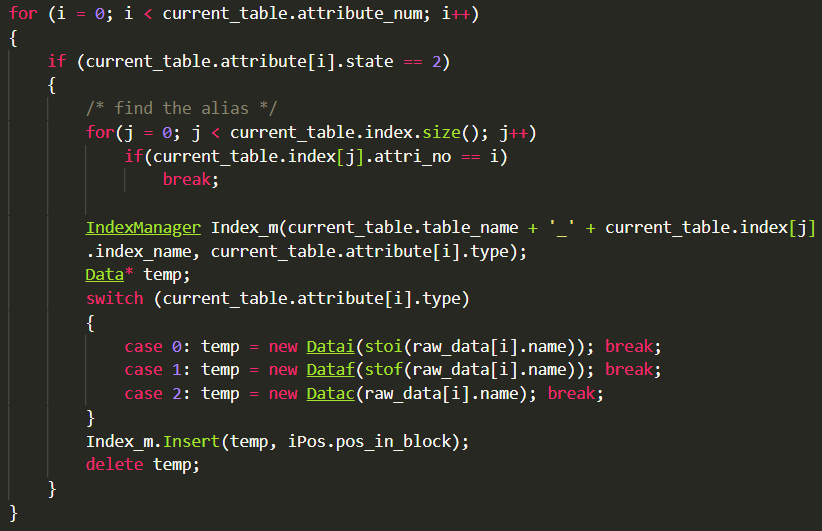


图表 33：用户输入数据示例

假设当前用户已经输入了这些数据，并且程序正在执行create index stunameidx on student ( sname )这个语句，那上面的循环便会从每一个块中读取记录，并计算出每个sname数据所在的位置。即分别读出”William”、 ”Leo”、 ”Charlie”、 ”Mickael”以及”Chris”，然后将这些字符串插入到当前表创建的index中，完成index的创建。

### 5.2.2 Insert函数（Index部分）

Insert函数首先会使用Search\_with\_Index函数以及Search\_without\_Index函数查找当前表插入过的所有记录，以判断当前插入的数据是否合法（使用了index、状态为unique或primary key的属性不允许出现重复的数值）。其中使用了index以及状态为primary key的属性，将会使用Search\_with\_Index函数进行查重。后续部分将详细介绍Search\_with\_Index函数。本部分仅介绍本人于该函数实现index相关操作的部分，其余部分的介绍由负责Record Manager的同学进行讲解。



图表 34：Insert函数部分代码

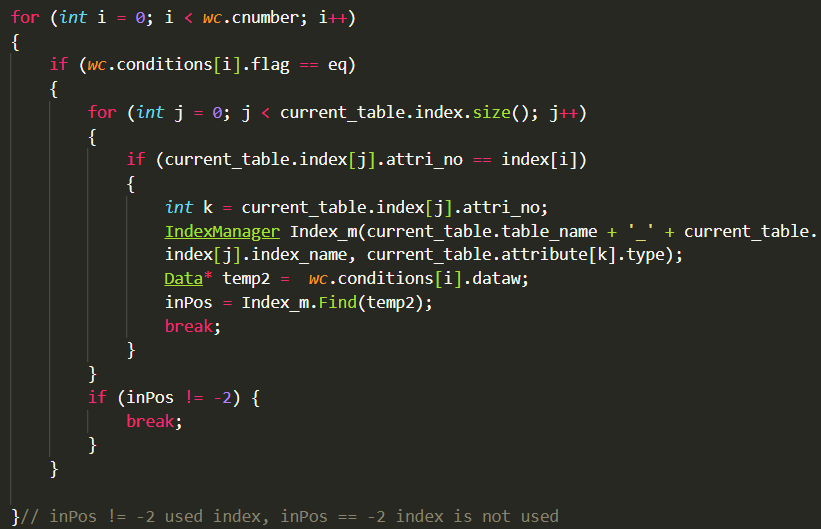
当调用了Buffer Manager的find\_Insertable\_pos函数后。此时iPos变量便指向了当前可插入块的偏移量。第一个循环遍历当前表的每个属性，寻找是否存在使用了index/primary key的属性。若该属性存在，则进入第二个循环，寻找当前表中使用了index的属性。接着将Index Manager以及temp变量根据该属性的数据类型进行初始化操作。最后调用Insert函数将temp的数值以及该记录在块中的偏移量保存到index中。完成index的插入操作。

### 5.2.3 Select函数（Index部分）



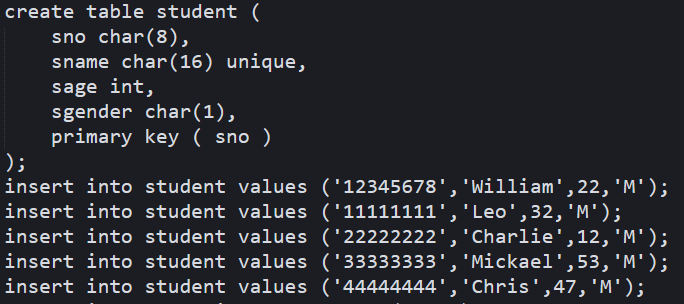
图表 35：Select函数使用index涉及的变量

函数首先定义了inPos以及No变量，其中inPos被初始化为-2。接着函数将计算where语句中待查询属性在当前表的位置。



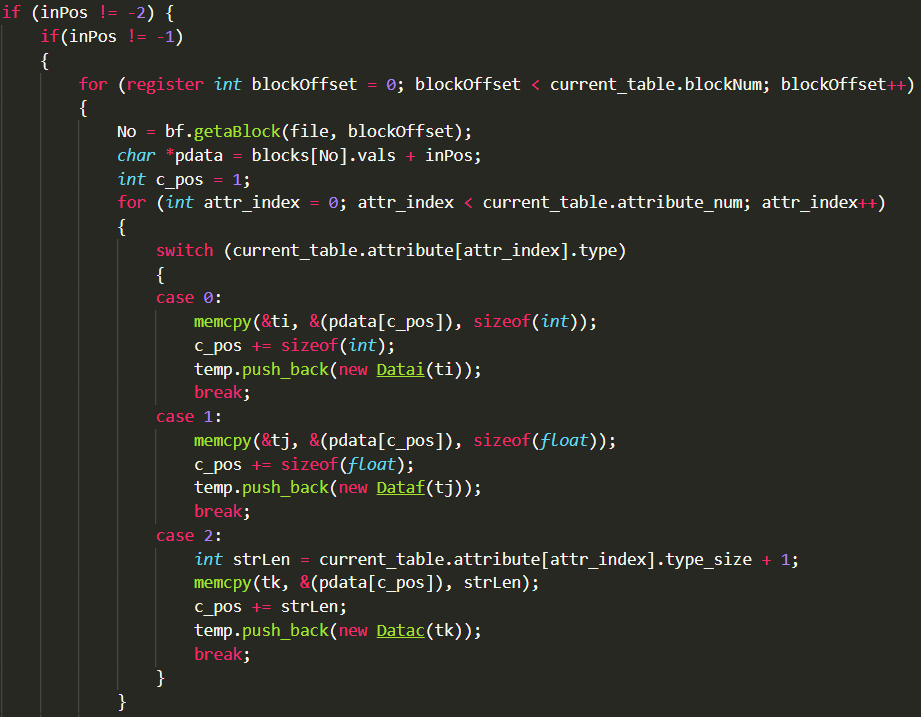
图表 36：Select函数部分代码1

当计算完查询属性的位置。函数将进入该循环，循环中将判断语句是否存在等号以及等号左边的属性是否使用了index。若两个条件都满足，则对Index Manager进行初始化，并以等号右边的数值作为参数调用Index Manager的Find函数。Find函数将返回-1或大于0的数值，其中-1表示index不存在该数值，大于0表示index存在该数值。假如返回值大于零，则该返回值为存放了此数值的记录在块中的偏移量。

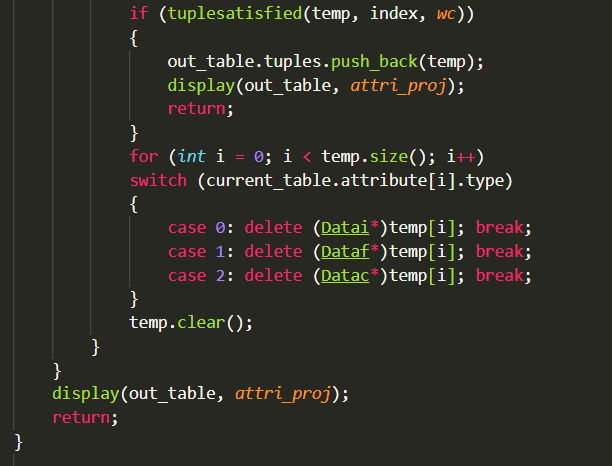


图表 37：用户输入数据示例

继续以刚才这个输入为例子，假设程序现正处理select \* from student where sno = '22222222'这个语句，那么Find函数将返回存放着数值'22222222'记录（即图表37中插入的第3个数据）在块中的偏移量。



图表 38：Select函数部分代码2



图表 39：Select函数部分代码3

接下来判断inPos是否不等于2。若等于2则表明该变量数值没有被修改过，不进入if语句；不等于2代表调用过了Index Manager的Find函数，此时应进入if语句中。假如inPos等于-1，则说明符合条件的记录不存在，此时调用图表39下方的display函数并返回。若inPos不等于 -1，则进入第二个if语句中的双重循环。第一个循环遍历当前表相关文件占据的块，使用pdata变量读取当前块的偏移地址。第二个循环取出记录的每个属性的数值，全部取出后进入tuplesatisfied函数判断该记录是否满足where的条件。若不满足则清除该记录所有属性的数值并清空temp变量（存放当前读出记录的属性数值），重新读取下个块；假如满足条件则该记录放入out\_table变量中，接着调用display显示结果，最后返回。

### 5.2.4 Delete函数（Index部分）

一开始使用index进行查找的操作与select部分完全相同，但取出记录后的操作与其略有不同。



图表 40：Delete函数部分代码

若取出的记录满足where条件，则将pdata指向的首个元素赋值为”Deleted”，该标记能被Buffer Manager识别。接着遍历当前表的index，调用Index Manager的Delete函数将index中对应属性的数值删除。最后将读出记录所有属性的数值删除并返回。而当记录不满足where条件的时候，则直接删除所有属性的数值并清空temp变量。Delect函数后半部分使用非index方式的查找满足where条件后，所执行的也同样是图表40的语句。

### 5.2.5 CreateIndex函数



图表 41：CreateIndex函数部分代码1

size变量用于计算当前成为index属性的内存空间偏移量，计算完成后保存到index\_attri变量中。



图表 42：CreateIndex函数部分代码2

接着进入双重循环。第一个循环遍历当前表相关文件占据的块，No变量记录当前使用的块号。第二个循环取出当前表记录中index属性的数值，利用刚才算出的index\_attri得到该属性在块中对应的内存地址，这样就不必读出整个记录，直接读出index属性的数值并插入到index中。

# 六、小结

以上就是本人在本次project中负责的主要工作。