DataBase MiniSQL 设计报告

秋冬学期数据库系统Project

作者：范源颢 3180103574 课程：秋冬学期数据库系统（周一一、二节）

宋天泽 学号？？ 指导老师：庄越挺 慕宗燊

李国耀 学号？？

江雨辰 学号？？

用于回应课题：

“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。”

“报告应该包括1.整体框架 2.各模块实现功能 3.分工说明 4.各模块提供接口和内部实现 5.界面说明 6.系统测试这几部分”

# 概论

本实验工程是秋冬学期数据库系统的期末作业，功能是实现一个精简单用户SQL引擎MiniSQL。实验是单用户的，因此并不涉及用户登录的问题，在运行程序之后即可开始对于数据库的操作，支持的操作包括建立/删除表，索引的建立/删除以及表记录的插入/删除/查找。

## 1.1开发环境和使用方法

本工程通过系统兼容性较强的python语言实现，使用的语言标准可以在python3.7-3.8的版本中正常运行（尚未测试更加古老的版本）。实验过程中的开发环境，操作系统通常是win10，工程软件包括pycharm、vscode、visual studio等等，均可以在装有上述软件的win10电脑中右键点击文件夹-点击“通过pycharm打开”/“通过vscode打开”等打开文件夹查看代码，实验的主程序是Interpreter.py，在终端中输入python Interpreter.py即可开始运行程序。运行过程中可以使用quit命令退出程序。

## 1.2支持的数据类型

按照本实验的期末作业要求，我们支持三种基本数据类型：int【整型变量】，char(n)【带长度限定的字符型变量，1<= n <= 255】，float【浮点型小数变量】。

## 1.3支持的表结构

实验中创建的表最多包含32个属性，各个属性可以指定是否为unique，支持单属性主键定义，不过要求表在创建时必须包含主键的声明（否则程序报错primary\_key 'None' does not exist. 【主键不存在】，拒绝创建表）。

创建表的语句格式如下：

create table 表名 (

列名 类型 ,

列名 类型 ,

列名 类型 ,

primary key ( 列名 )

);

删除表使用drop命令：

drop table 表名 ;

## 1.4支持的索引结构

实验中涉及到的索引都是单属性单值的，对于表的主属性，程序将自动建立B+树的索引， 对于用户声明为unique的属性，用户可以通过SQL语句制定建立/删除B+树索引。

相关命令如下：

创建索引：

create index 索引名 on 表名 ( 列名 );

删除索引：

drop index 索引名;

## 1.5支持的查询语言

实验中设计的数据库访问语言，支持的包括查找（select）、删除（delete）、插入（insert）。Delete和insert查询语句都支持使用where条件句，条件允许使用and、or等逻辑词并列，条件中也可以出现 = <> < > <= >=等比较运算符（但暂时不支持算数运算）。

相关语句的格式如下：

选择：（示例语句， \* 表示全部）

select \* from student;

select \* from student where sno = ‘88888888’;

select \* from student where sage > 20 and sgender = ‘F’;

插入：

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

删除：（示例语句）

delete from student;

delete from student where sno = ‘88888888’;

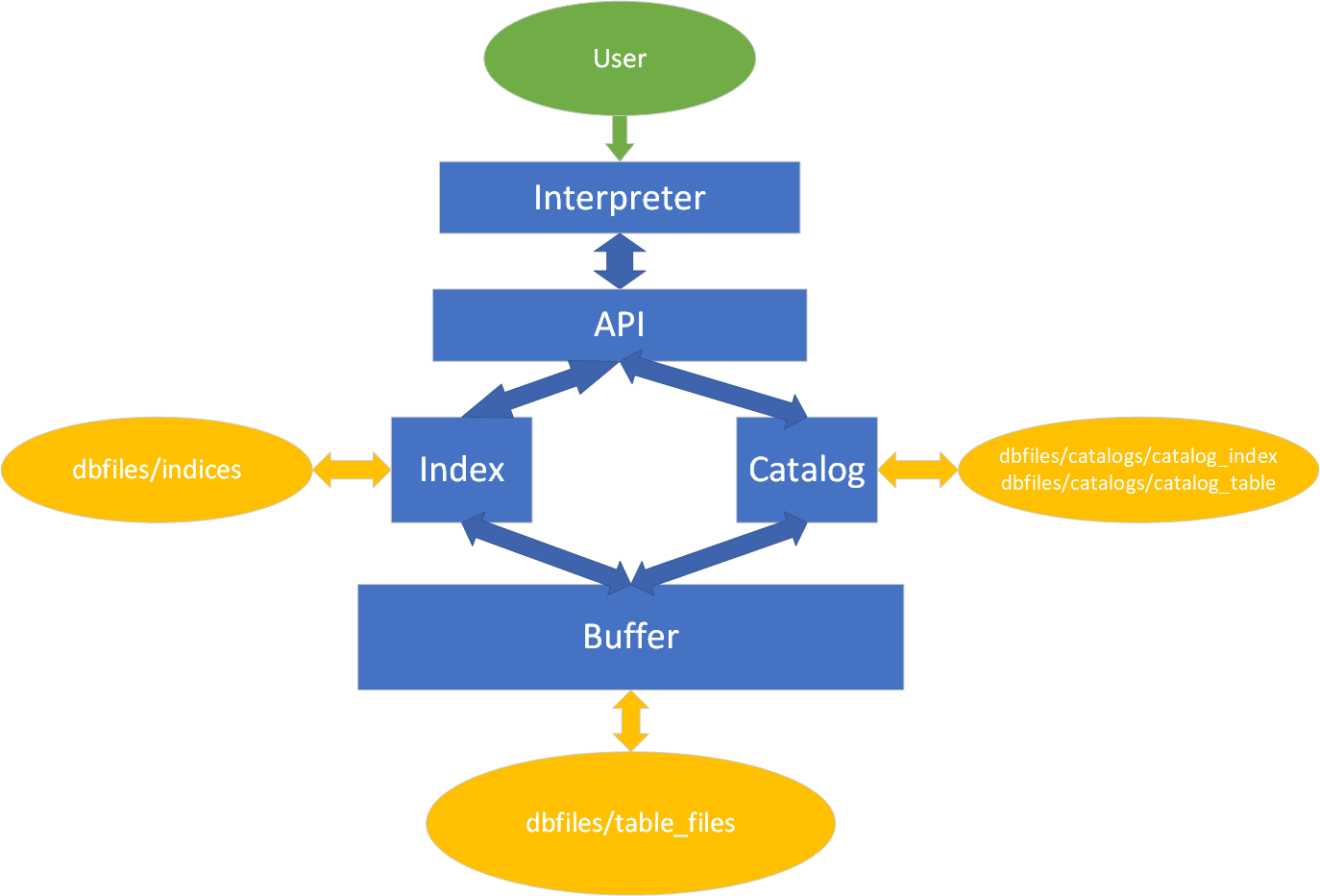
相关的查询语言可以编写为execfile脚本，其中可以包含任意多条上述的SQL语言，程序读入之后按照顺序执行脚本中的指令。

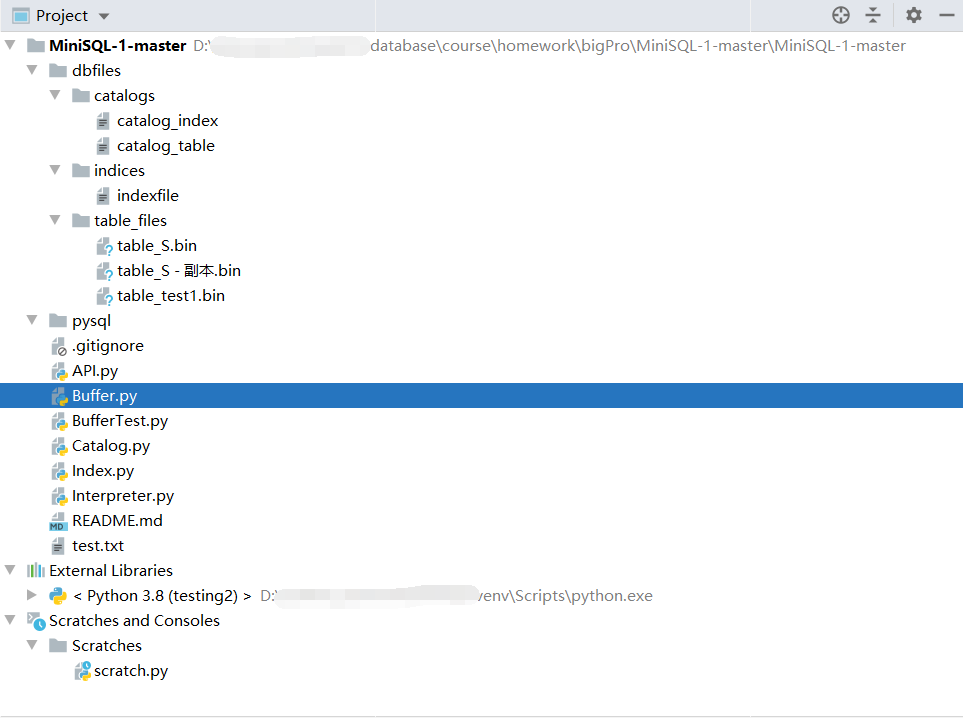
# 整体框架

实验按照题目给出的建议，我们设计了Interpreter.py和API.py作为交互接口，之后，我们插入的所有record都在Buffer.py中暂存（这里注意和实验课题的建议有所不同，我们没有recordManager或者与之相似的模块，实验建议中，应该由recordManager实现的内容都被Buffer.py实现了，亦即，在本程序中。RecordManager 和BufferManager 合并了，合并后的名字在这里是Buffer.py）之后，我们的Index.py和Catalog.py分别处理用户输入的索引信息和表头信息，然后，表头信息、索引信息，连同Buffer处理过的记录信息全部被存储到了dbfiles文件夹下。表头信息，也就是各个表的名称和行、列属性储存在catalogs文件夹下，由catalog\_index和catalog\_table分别储存。索引信息由indices下的indexfile储存，各个表格的记录则由table\_files下的各个table\_***tableName***储存。

我们总体的模块构成可以用下图表示：

下面这张图也可以帮我改改



 存储目录如下：

由上图可知，已经创建的表格包括‘S’和‘test1’。

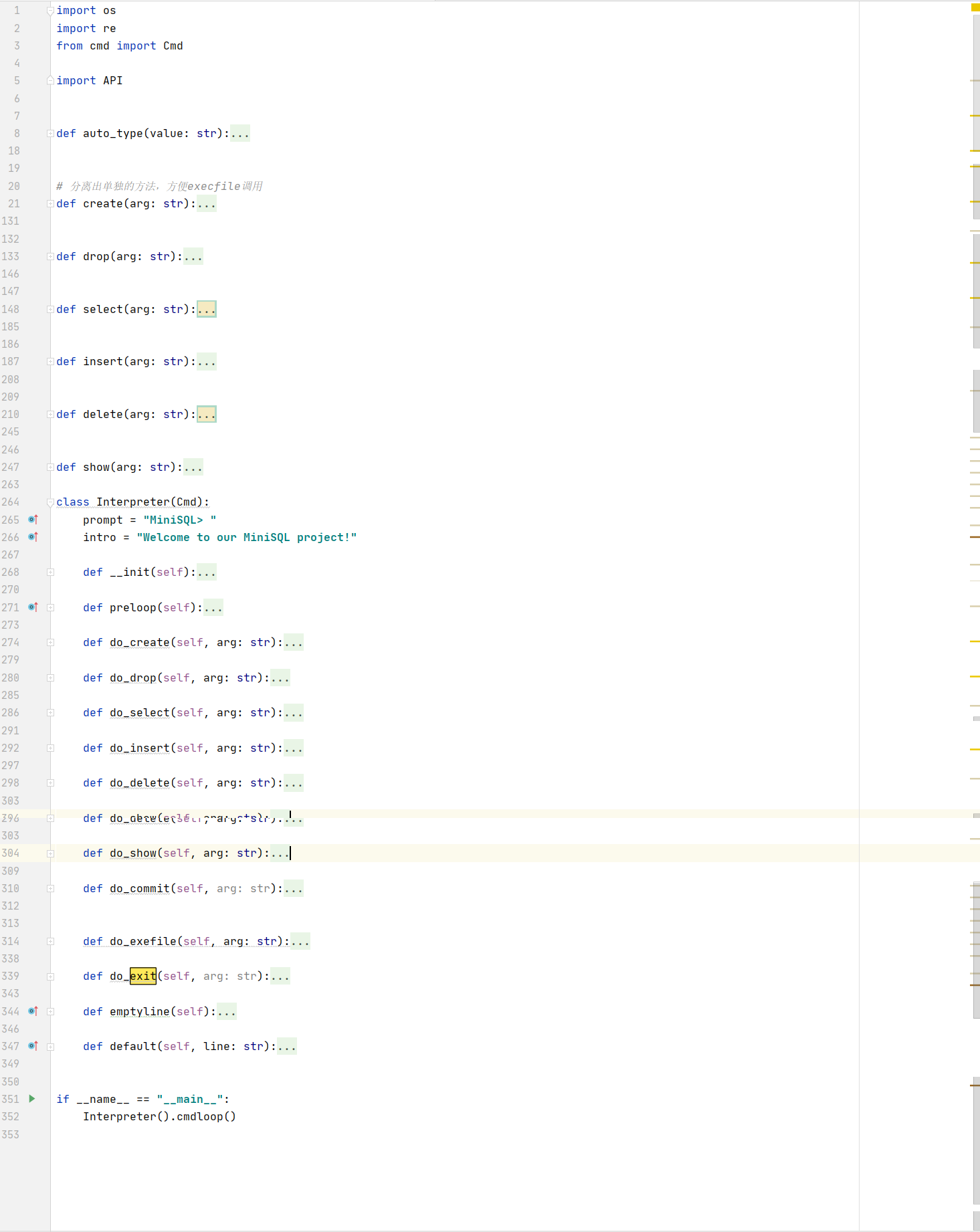
# 各模块实现功能

## 3.1 Interpreter

Interpreter （义即，解释器）模块直接与用户交互，是程序的前段，主要实现以下功能:

1. 程序流程控制，即启动并且初始化、接受命令、处理命令、显示命令结果、循环、退出等等流程
2. 处理用户输入的命令，主要通过python的re库中的正则表达式方法，将用户在终端输入的指令转化为可释读的结构，将语句做剖析（parse），同时先进行简单的语法检查，确认是支持的语句之后，通过调用API来实现相关的命令，如果不是合法的语句，则会通过错误捕捉机制打印相关的错误。
3. SQL脚本文件执行的部分也在这个模块之内完成。

文本结构：



## 3.2 API

API（application program interface义即应用程序编程接口）是程序前段和后端的接口，是系统的核心部件，主要功能就是将Interpreter层解析出的函数作为模块输入，之后根据Catalog提供的信息作为执行规则，调用Index、 Catalog、Buffer提供的相应接口执行，最后返回的执行结果给Interpreter。

主要实现的函数包括:

create\_table, 创建表

create\_index,创建目录

drop\_table,删除表

drop\_index,删除目录

select,选择记录

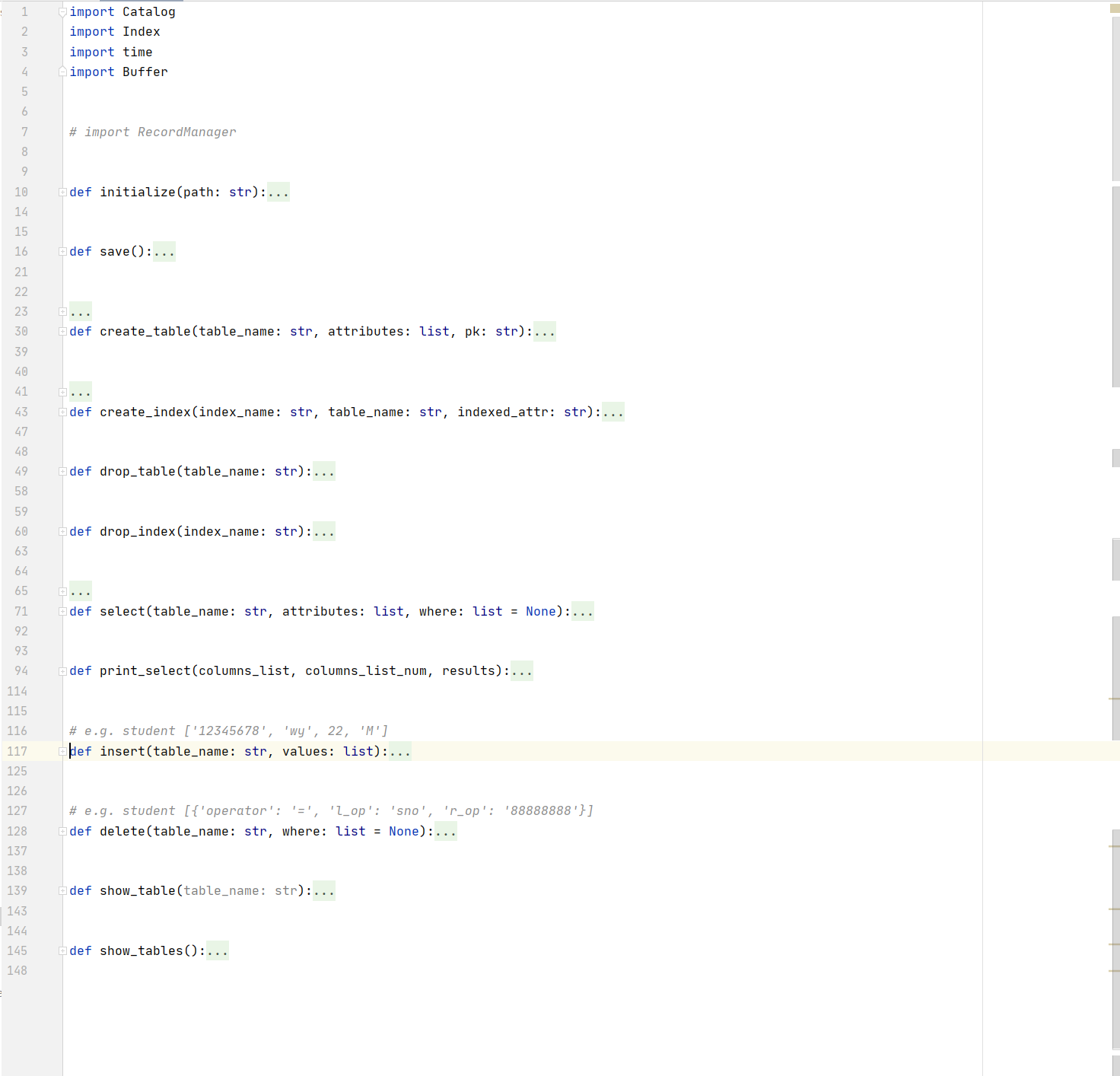
print\_select (用于进行选择结果的打印)

insert，插入记录

delete，删除记录

show\_table

show\_tables,(这两个似乎没有实现？)

文件总览：

## 3.3 Catalog

这里的catalog（编目）是catalog Manager的意思，他主要负责实现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还提供了访问及操作的接口，供Interpreter和API模块使用。

Catalog定义了表头文件tables，目录文件indices，同时也指定了相关的文件路径，即catalogpath、tablecatalog、indexcatalog。

之后Catalog设定了两个比较重要的类，Table()：记录表的名字，他的主键，以及每列的名字；Column()：组成Table类的一个结构，表示的是Table的各个attribute，记录了这一列的数据类型，以及名称、是否限定不可重复，总长度等等信息。

Catalog随后实现了以下功能：

create table:将API反馈的table写入到tables字典储存（这一步似乎没有通过buffer，直接写入）

drop\_table：将相关的表从tables 字典对象中删除

check\_types\_of\_table: 根据tables字典对象，检查table中各个属性的类型和用户插入的是否一致的函数，被API 模块调用，如果不满足则抛出一个异常。

exists\_table：根据tables字典对象，检查table是否已经存在的函数，若有则抛出异常，也是被API调用的函数；

not\_exist\_table：根据tables字典对象，检查table是否不存在的函数，若确实不存在则抛出异常，也是被API调用的函数，用于对于用户的行为作出检查。

exists\_index: 根据indices字典对象，检查index是否存在，若已经存在则抛出异常，被API调用；

exists\_index：根据indices字典对象，检查index是否不存在的函数，功用同上；

drop\_index：实现将索引从indices字典对象中删除的功能；

create\_index：实现创建索引并且储存到indices字典对象中的功能；

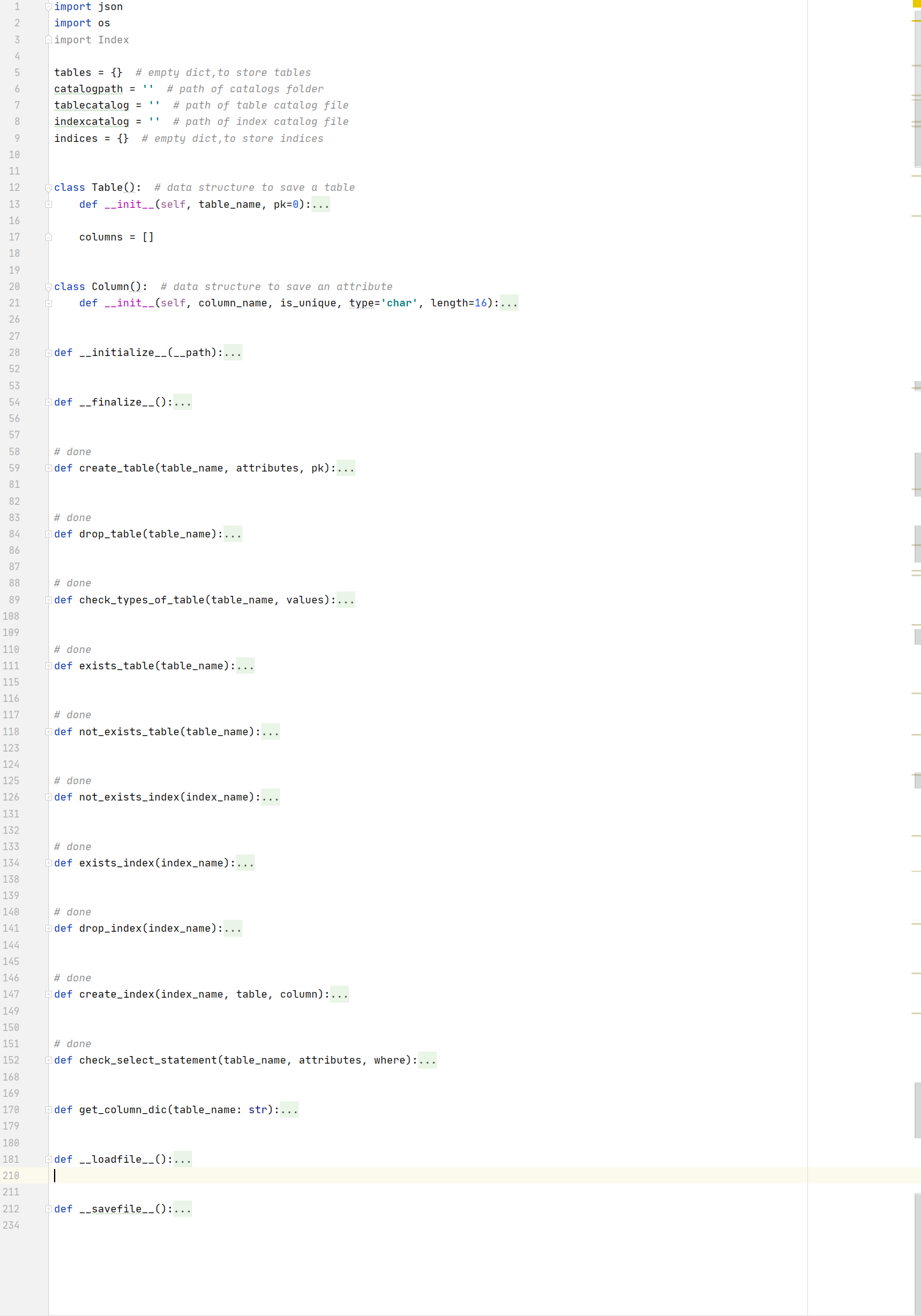
check\_select\_statement：根据tables字典检查select语法的函数。

get\_column\_dic：返回相关表的属性的函数，用在API中，方便select、delete的实现

\_\_loadfile\_\_：将table\_fies文件中读出，然后储存到tables字典中，同时将indices文件读出，将相关变量储存到indices字典中；

\_\_savefile\_\_: 将对于indices和tables字典的修改保存到相关的磁盘文件中。

文件总览：



## Index

Index Manager负责B+树索引的实现，实现B+树的创建和删除（由索引的定义与删除引起）、等值查找、插入键值、删除键值等操作，并对外提供相应的接口。

B+树中节点大小应与缓冲区的块大小相同，B+树的叉数由节点大小与索引键大小计算得到。

Index文件中定义了B+树的节点：node类型。

之后我们实现了以下函数：

\_\_load\_\_: 从indices文件夹下的indexfile文件中读入B+树的信息；

load\_nodes: 将indexfile中的B+树解析成一个个树节点，还原成可操作的B+树的pointer\_list和node\_list。

\_\_store\_\_:将操作之后的B+树存储回indexfile文件中, 调用recursive\_store\_node实现；

recursive\_store\_node: 递归法将各个node中的信息存储到相应文件中。

insert\_into\_table: 根据API解析后的insert命令修改B+树内容；

create\_table：根据API解析后的create命令增加一颗B+树；

delete\_from\_table：根据API解析后的delete命令删除B+树的记录；

check\_conditions：检查select语句中的操作符（==， >=等等）便于搜索；

maintain\_B\_plus\_tree\_after\_delete：在delete之后修正B+树的结构；

create\_index: 根据用户创建的索引重新生成一颗B+树

print\_select: 将搜索的结果打印出来，被API的同名函数调用。

select\_from\_table: 实现API中的select函数

check\_unique: 检查用户的操作是否破坏了unique约束

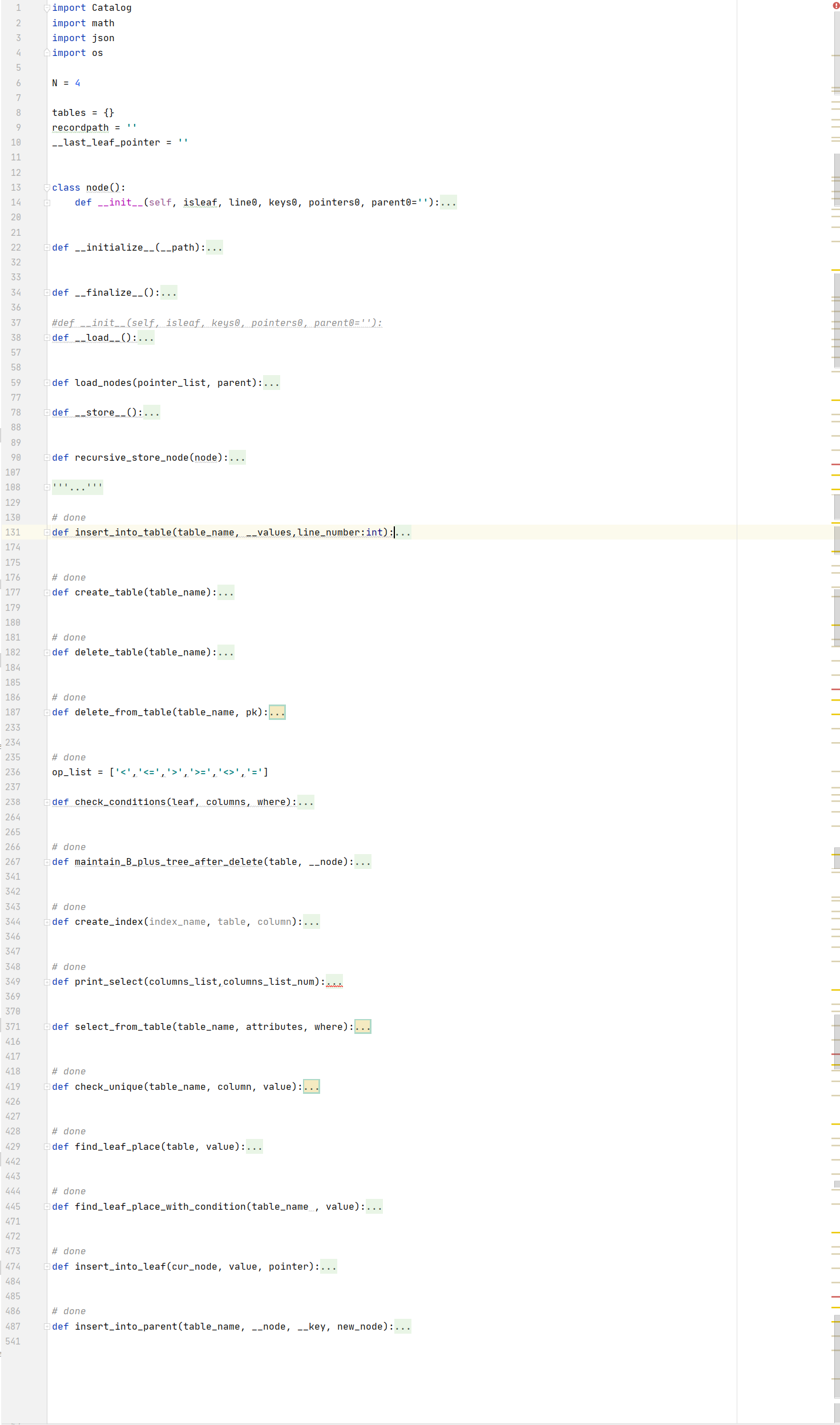
find\_leaf\_place：B+树操作，用于快速寻找特定的B+树节点；

find\_leaf\_place\_with\_condition：B+树操作，用于快速寻找B+树中一个用户指定的区间；

insert\_into\_leaf：B+树操作，用于在B+树的叶结点中插入一条记录；

insert\_into\_parent: B+树操作，用于在B+树递归的插入需要的parent节点。

文件概览：



## Buffer（这部分我真没怎么看懂，想知道line是怎么回事）

Buffer Manager负责缓冲区的管理，主要功能有：

1. 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
2. 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
3. 记录缓冲区中各页的状态，如是否被修改过等
4. 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。

Buffer中我们指定的缓冲大小比较小，用两个宏来定义，分别是BUFFER\_SIZE【每个表可以包含的记录数】，以及BUFFER\_NUM【缓冲区可以包含的表数目】

Buffer文件中定义了Buffer类，用来描述buffer的大小信息，并且定义了判定是否满，是否可写入，以及保存等方法。

Buffer中实现的函数如下：

check: 检查API解析出的语句是否合法；

decode: 将API解析出的各个元素解码以便分析其属性；

find\_attr\_pos: 定位到需要的位置上

find\_line;

find\_record:

delete\_line:

delete\_record

check\_unique;

insert\_record

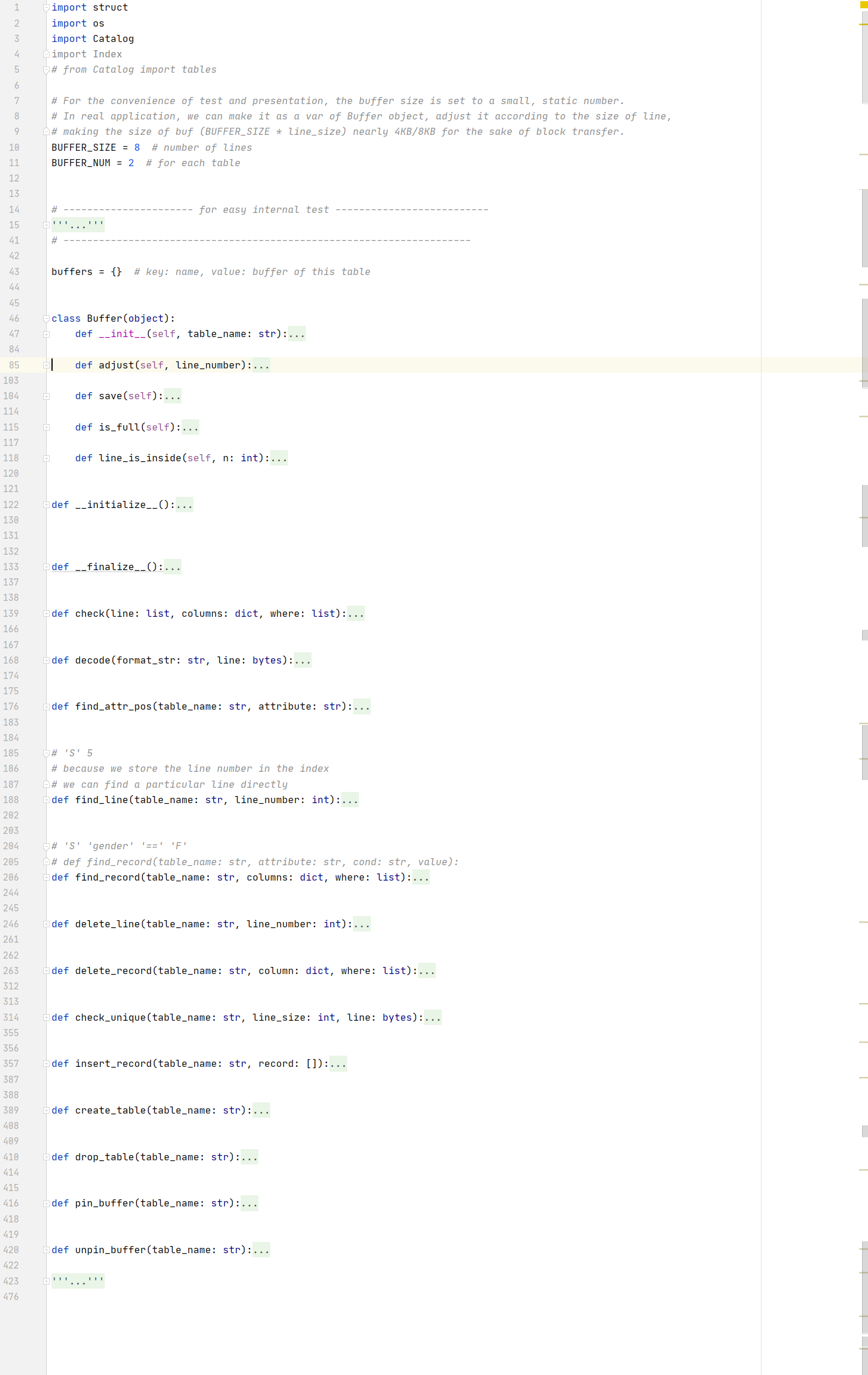
create\_table：

drop\_table

pin\_buffer: 锁定缓冲区，不允许替换

unpin\_buffer：解除缓冲区锁定

文件概览：



# 分工说明

各自领取积分好了，可以重复

# 各模块接口和内部实现

## 5.1 Interpreter

Interpreter的实现借助了python的包OS（operation system操作系统包），re（regular expression正则表达式包）以及cmd（command，命令行包）。

其实现的主要逻辑如下：

### 5.1.1 类型转换

我们首先根据正则表达式，设定auto\_type函数将语句中的整形数字、浮点型数字、以及引号内的字符串转化为相关的变量类型，语句的这些部分不再理解为关键字。

**def** auto\_type**(**value**:** **str):**

**if** value**[**0**]** **==** "'" **and** value**[-**1**]** **==** "'"**:**

value **=** value**[**1**:-**1**]**

**elif** re**.**match**(**r'^-?[0-9]+\.+[0-9]+$'**,** value**):**

value **=** **float(**value**)**

**elif** re**.**match**(**r'^-?[0-9]+'**,** value**):**

value **=** **int(**value**)**

**else:**

**raise** **Exception(**f"Unsupported format: {value}"**)**

**return** value

### 5.1.2 句法分析

之后我们进行句法分析，根据cmd中读入的字符，我们将用户输入的合法字符串分类为create、drop、select、insert、delete、show这几类，每一种按照相关的语法拆分这些字符串，然后将相关的关键字下的内容传递给API，使得API完成用户根据完整句子给出的需求，

这里谨以最复杂的create为例，展示句法分析如何完成：

create**(**arg**:** **str):**

arg **=** arg**.**strip**()**

**if** arg**[-**1**]** **==** '('**:** # 支持create的多行输入

**while** 1**:**

line **=** **input().**rstrip**()**

arg **=** arg **+** line

**if** line**[-**1**]** **!=** ',' **and** line**[-**1**]** **!=** ')'**:**

**break**

arg **=** arg**.**rstrip**(**';'**).**strip**()** # 去尾部分号

arg **=** re**.**sub**(**' +'**,** ' '**,** arg**)** # 将多空格换为单空格（两边是没有空格的）

**if** arg**[:**5**]** **==** 'table'**:**

arg **=** arg**[**5**:]**

arg **=** arg**.**lstrip**()** # 去table后空格

table\_name **=** arg**[:**arg**.**find**(**'('**)].**strip**()** # 通过定位'('获取表名

**if** table\_name **==** ''**:**

**raise** **Exception(**"No table name found."**)**

# 去除定义表的括号

arg **=** arg**[**arg**.**find**(**'('**):]**

arg **=** arg**.**lstrip**(**'('**).**strip**()**

**if** arg**[-**1**]** **==** ')'**:**

arg **=** arg**[:-**1**]**

arg **=** arg**.**strip**()**

**if** arg **==** ''**:**

**raise** **Exception(**"No table specification found."**)**

# 获取属性定义、pk等

attribute\_specifications **=** arg**.**split**(**','**)**

attribute\_specifications **=** **list(map(str.**strip**,** attribute\_specifications**))**

**if** attribute\_specifications **==** **[]:**

**raise** **Exception(**"No table attribute found."**)**

# 先处理pk

pk **=** **None**

**if** attribute\_specifications**[-**1**].**startswith**(**'primary key'**):**

pk **=** attribute\_specifications**[-**1**]**

**if** ',' **in** pk**:**

**raise** **Exception(**"Only single primary key is supported."**)**

pk **=** pk**[**11**:].**strip**().**lstrip**(**'('**).**rstrip**(**')'**).**strip**()**

attribute\_specifications **=** attribute\_specifications**[:-**1**]**

# 依序处理属性定义

attributes **=** **[]**

attribute\_names **=** **[]**

**for** attribute\_specification **in** attribute\_specifications**:**

# item: attribute name, type, and optional unique

unique **=** **False**

type\_len **=** 0

item **=** attribute\_specification**.**split**(**' '**)**

**if** item**[**1**]** **not** **in** **[**'int'**,** 'float'**]:**

**if** item**[**1**].**startswith**(**'char'**):**

type\_len **=** **int(**item**[**1**][**4**:].**strip**().**lstrip**(**'('**).**rstrip**(**')'**))**

**if** type\_len **<=** 0**:**

**raise** **Exception(**f"The size of the type is negative."**)**

item**[**1**]** **=** 'char'

**else:**

**raise** **Exception(**f"The type of attribute {item**[**0**]**} is {item**[**1**]**}, which is not supported."**)**

**if** **len(**item**)** **==** 3**:**

**if** item**[**2**]** **==** "unique"**:**

unique **=** **True**

**else:**

**raise** **Exception(**f"The command behind {item**[**0**]**} {item**[**1**]**} is not supported."**)**

attribute\_names**.**append**(**item**[**0**])**

attributes**.**append**({**

'attribute\_name'**:** item**[**0**],**

'type'**:** item**[**1**],**

'type\_len'**:** type\_len**,**

'unique'**:** unique

**})**

**if** pk**:**

**if** pk **not** **in** attribute\_names**:**

**raise** **Exception(**f"The primary key {pk} you want is not in the attribute list."**)**

**else:**

attributes**[**attribute\_names**.**index**(**pk**)][**'unique'**]** **=** **True**

**print(**table\_name**,** attributes**,** pk**)**

API**.**create\_table**(**table\_name**,** attributes**,** pk**)**

**elif** arg**[:**5**]** **==** 'index'**:**

arg **=** arg**[**5**:]**

arg **=** arg**.**lstrip**()** # 去index后空格

location\_on **=** arg**.**find**(**'on'**)**

**if** location\_on **==** **-**1**:**

**raise** **Exception(**f"'on' is missing when creating index."**)**

index\_name **=** arg**[:**location\_on**].**strip**()**

location\_lbracket **=** arg**.**find**(**'('**)**

**if** location\_lbracket **==** **-**1**:**

**raise** **Exception(**f"Indexed attribute format is wrong."**)**

table\_name **=** arg**[**location\_on **+** **len(**'on'**):** location\_lbracket**].**strip**()**

location\_rbracket **=** arg**.**find**(**')'**)**

**if** location\_rbracket **==** **-**1**:**

**raise** **Exception(**f"Indexed attribute format is wrong."**)**

indexed\_attr **=** arg**[**location\_lbracket**+**1**:** location\_rbracket**].**strip**()**

**if** ',' **in** indexed\_attr**:**

**raise** **Exception(**'Only single attribute index is supported.'**)**

**print(**index\_name**,** table\_name**,** indexed\_attr**)**

API**.**create\_index**(**index\_name**,** table\_name**,** indexed\_attr**)**

**else:**

**raise** **Exception(**"The item you want to create is not supported."**)**

可以看到，在处理完了所有报错之后，我们将创建表或者创建索引的任务交给了API

### 5.1.3 用户的命令行交互

在实现了用户的句法分析之后，我们还需要通过终端捕捉用户的行为，为此我们使用了python含有的cmd类来实现程序和用户的交互

Cmd类主要给出了对用户的提示和导引，对于用户的操作解析全部放到了5.1.2的那些句法分析函数之中了，此类的主要责任是使用try：except语法给出必要的报错

**class** **Interpreter(**Cmd**):**

prompt **=** "MiniSQL> "

intro **=** "Welcome to our MiniSQL project!"

**def** \_\_init**(**self**):**

Cmd**.**\_\_init\_\_**(**self**)**

**def** preloop**(**self**):**

API**.**initialize**(**os**.**getcwd**())**

**def** do\_create**(**self**,** arg**:** **str):**

**try:**

create**(**arg**)**

**except** **Exception** **as** e**:**

**print(**e**)**

**def** do\_drop**(**self**,** arg**:** **str):**

**try:**

drop**(**arg**)**

**except** **Exception** **as** e**:**

**print(**e**)**

**def** do\_select**(**self**,** arg**:** **str):**

**try:**

select**(**arg**)**

**except** **Exception** **as** e**:**

**print(**e**)**

**def** do\_insert**(**self**,** arg**:** **str):**

**try:**

insert**(**arg**)**

**except** **Exception** **as** e**:**

**print(**e**)**

**def** do\_delete**(**self**,** arg**:** **str):**

**try:**

delete**(**arg**)**

**except** **Exception** **as** e**:**

**print(**e**)**

**def** do\_show**(**self**,** arg**:** **str):**

**try:**

show**(**arg**)**

**except** **Exception** **as** e**:**

**print(**e**)**

**def** do\_commit**(**self**,** arg**:** **str):**

API**.**save**()**

**def** do\_exefile**(**self**,** arg**:** **str):**

switch **=** **{**

'create'**:** create**,**

'drop'**:** drop**,**

'select'**:** select**,**

'insert'**:** insert**,**

'delete'**:** delete**,**

'show'**:** show

**}**

i **=** 1

**try:**

f **=** **open(**arg**.**strip**(**';'**).**strip**(),** 'r'**)**

**while** 1**:**

line **=** f**.**readline**().**strip**()**

**if** line **==** ''**:**

**break**

command **=** line**[:**line**.**find**(**' '**)]**

arg **=** line**[**line**.**find**(**' '**):]**

switch**[**command**](**arg**)**

i **+=** 1

**except** **Exception** **as** e**:**

**print(**f"An exception occurred at line {i}:"**)**

**print(**e**)**

**pass**

**def** do\_exit**(**self**,** arg**:** **str):**

API**.**save**()**

**print(**'Bye~'**)**

**return** **True**

**def** emptyline**(**self**):**

**pass**

**def** default**(**self**,** line**:** **str):**

**print(**f"Unknown command: {line**.**split**(**' '**)[**0**]**}"**)**

### 5.1.4 主函数

在定义cmd类之后，我们所需要的的就是一个轮询的命令提示符的函数，就可以完成设计：

**if** \_\_name\_\_ **==** "\_\_main\_\_"**:**

Interpreter**().**cmdloop**()**

## 5.2 API

API主要是程序的接口文件，负责将句法分析的结果转交给Catalog和Index来实现，然后根据这两个模块的返回结果反馈用户信息。然而，其实，真正用户需要即时反馈信息的命令只有select一个，其他的函数我们仅仅根据Catalog和Index的结果告诉用户是否有报错，如若成功，再依靠time()函数给出操作总共的用时。API作为一个接口，主要的任务是转交，因此它本身的代码量反倒不多，下文以期最复杂的select命令为例展示其实现过程。

**def** select**(**table\_name**:** **str,** attributes**:** **list,** where**:** **list** **=** **None):**

time\_start **=** time**.**time**()**

Catalog**.**not\_exists\_table**(**table\_name**)**

Catalog**.**check\_select\_statement**(**table\_name**,** attributes**,** where**)**

#Index.select\_from\_table(table\_name, attributes, where)

col\_dic **=** Catalog**.**get\_column\_dic**(**table\_name**)**

**print(**col\_dic**)**

results **=** Buffer**.**find\_record**(**table\_name**,** col\_dic**,** where**)**

**print(**results**)**

numlist **=** **[]**

**if** attributes **==** **[**'\*'**]:**

attributes **=** **list(**col\_dic**.**keys**())**

numlist **=** **list(**col\_dic**.**values**())**

**else:**

**for** att **in** attributes**:**

**print(**att**)**

numlist**.**append**(**col\_dic**[**att**])**

print\_select**(**attributes**,** numlist**,** results**)**

time\_end **=** time**.**time**()**

**print(**" time elapsed : %fs." **%** **(**time\_end **-** time\_start**))**

**def** print\_select**(**columns\_list**,** columns\_list\_num**,** results**):**

**print(**'-' **\*** **(**17 **\*** **len(**columns\_list\_num**)** **+** 1**))**

**for** i **in** columns\_list**:**

**if** **len(str(**i**))** **>** 14**:**

output **=** **str(**i**)[**0**:**14**]**

**else:**

output **=** **str(**i**)**

**print(**'|'**,** output**.**center**(**15**),** end**=**''**)**

**print(**'|'**)**

**print(**'-' **\*** **(**17 **\*** **len(**columns\_list\_num**)** **+** 1**))**

**for** i **in** results**:**

**for** j **in** columns\_list\_num**:**

**if** **len(str(**i**[**j**]))** **>** 14**:**

output **=** **str(**i**[**j**])[**0**:**14**]**

**else:**

output **=** **str(**i**[**j**])**

**print(**'|'**,** output**.**center**(**15**),** end**=**''**)**

**print(**'|'**)**

**print(**'-' **\*** **(**17 **\*** **len(**columns\_list\_num**)** **+** 1**))**

**print(**"Returned %d entries," **%** **len(**results**),** end**=**''**)**

可以看到，在正式插入之前，我们首先用Catalog模块的几个函数检查了语法是否和table文件夹下的表头定义是否相合，然后我们有调用Buffer模块的内容将相应的记录真正搜寻出来，之后我们用numlist暂存需要打印的属性（attribute，attr），最后通过print\_select函数将这些内容全部打印出来。

其他的功能也类似，在具体的某个函数中，如何和Catalog以及 Index联系是这个函数的功能所在，例如，相对简单的一个create\_table 的函数实现过程如下：

**def** create\_table**(**table\_name**:** **str,** attributes**:** **list,** pk**:** **str):**

time\_start **=** time**.**time**()**

Catalog**.**exists\_table**(**table\_name**)**

Index**.**create\_table**(**table\_name**)**

Catalog**.**create\_table**(**table\_name**,** attributes**,** pk**)**

Buffer**.**create\_table**(**table\_name**)**

time\_end **=** time**.**time**()**

**print(**"Successfully create table '%s', time elapsed : %fs." **%**

**(**table\_name**,** time\_end **-** time\_start**))**

## 5.3 Catalog

## 5.4 Index

## 5.5 Buffer

# 界面说明

# 七、系统测试