DataBase MiniSQL 设计报告

秋冬学期数据库系统 Project

作者：范源颢 3180103574
宋天泽 3180105221
李国耀 3180103687
江雨辰 3180106317

课程：秋冬学期数据库系统（周一一、二节）
指导老师：庄越挺 慕宗燊

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”

一、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）

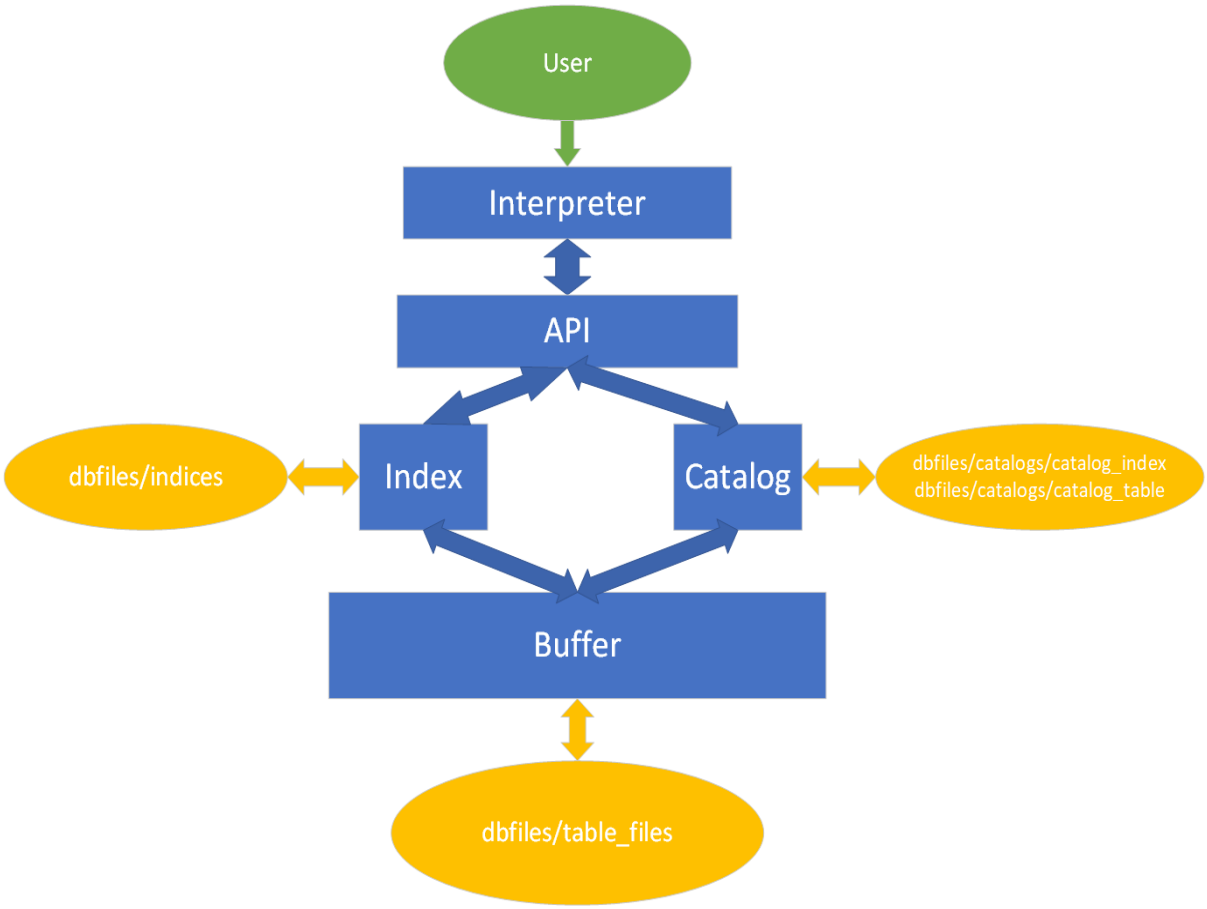
```
delete from student;
delete from student where sno = '88888888';
```

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。

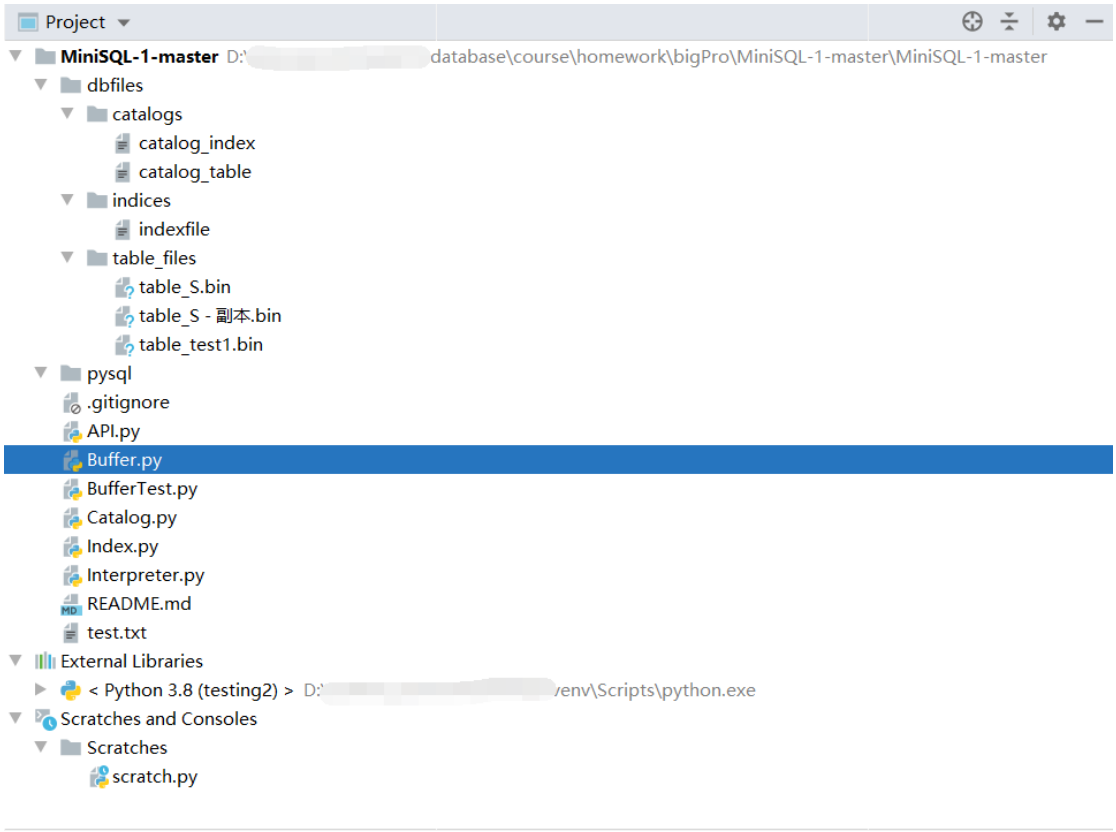
二、整体框架

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：



存储目录如下：



由上图可知，已经创建的表格包括‘S’和‘test1’。

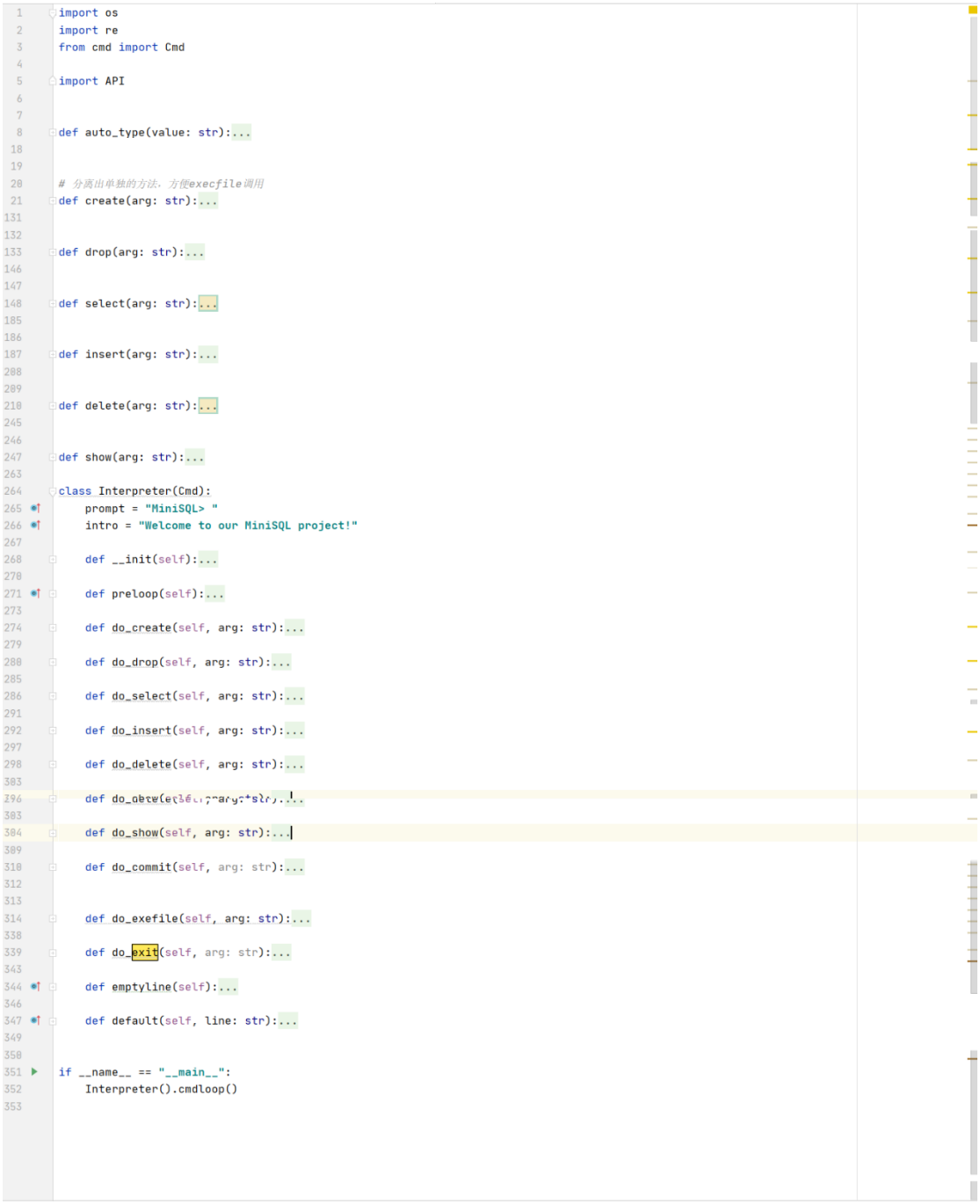
三、各模块实现功能

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：



```
import os
import re
from cmd import Cmd

import API


def auto_type(value: str):...


# 分离出单独的方法，方便execfile调用
def create(arg: str):...


def drop(arg: str):...


def select(arg: str):...


def insert(arg: str):...


def delete(arg: str):...


def show(arg: str):...

class Interpreter(Cmd):
    prompt = "MiniSQL> "
    intro = "Welcome to our MiniSQL project!"

    def __init(self):...

    def preloop(self):...

    def do_create(self, arg: str):...

    def do_drop(self, arg: str):...

    def do_select(self, arg: str):...

    def do_insert(self, arg: str):...

    def do_delete(self, arg: str):...

    def do_show(self, arg: str):...

    def do_commit(self, arg: str):...


    def do_exefile(self, arg: str):...

    def do_exit(self, arg: str):...

    def emptyline(self):...

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


if __name__ == "__main__":
    Interpreter().cmdloop()
```

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，展示数据库某个特定表的表头信息
show_tables，展示数据库中所有的表

文件总览:

```python
import Catalog
import Index
import time
import Buffer


# import RecordManager


def initialize(path: str):...


def save():...


...
def create_table(table_name: str, attributes: list, pk: str):...


...
def create_index(index_name: str, table_name: str, indexed_attr: str):...


def drop_table(table_name: str):...


def drop_index(index_name: str):...


...
def select(table_name: str, attributes: list, where: list = None):...


def print_select(columns_list, columns_list_num, results):...


# e.g. student ['12345678', 'wy', 22, 'M']
def insert(table_name: str, values: list):...


# e.g. student [{'operator': '=', 'l_op': 'sno', 'r_op': '88888888'}]
def delete(table_name: str, where: list = None):...


def show_table(table_name: str):...


def show_tables():...
```

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：

```python
import json
import os
import Index

tables = {}  # empty dict,to store tables
catalogpath = ''  # path of catalogs folder
tablecatalog = ''  # path of table catalog file
indexcatalog = ''  # path of index catalog file
indices = {}  # empty dict,to store indices


class Table():  # data structure to save a table
    def __init__(self, table_name, pk=0):...

    columns = []


class Column():  # data structure to save an attribute
    def __init__(self, column_name, is_unique, type='char', length=16):...


def __initialize__(__path):...


def __finalize__():...


# done
def create_table(table_name, attributes, pk):...


# done
def drop_table(table_name):...


# done
def check_types_of_table(table_name, values):...


# done
def exists_table(table_name):...


# done
def not_exists_table(table_name):...


# done
def not_exists_index(index_name):...


# done
def exists_index(index_name):...


# done
def drop_index(index_name):...


# done
def create_index(index_name, table, column):...


# done
def check_select_statement(table_name, attributes, where):...


def get_column_dic(table_name: str):...


def __loadfile__():...


def __savefile__():...
```

3.4 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: 创建索引以及相关内容

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，递归的插入需要的 parent 节点，用于维护。

文件概览：

```python
import Catalog
import math
import json
import os

N = 4

tables = {}
recordpath = ''
__last_leaf_pointer = ''


class node():
    def __init__(self, isleaf, line0, keys0, pointers0, parent0=''):...


def __initialize__(__path):...


def __finalize__():...

#def __init__(self, isleaf, keys0, pointers0, parent0='');
def __load__():...


def load_nodes(pointer_list, parent):...

def __store__():...


def recursive_store_node(node):...

'''...'''

# done
def insert_into_table(table_name, __values,line_number:int):...


# done
def create_table(table_name):...


# done
def delete_table(table_name):...


# done
def delete_from_table(table_name, pk):...


# done
op_list = ['<','<=','>','>=','<>','=']

def check_conditions(leaf, columns, where):...


# done
def maintain_B_plus_tree_after_delete(table, __node):...


# done
def create_index(index_name, table, column):...


# done
def print_select(columns_list,columns_list_num):...


def select_from_table(table_name, attributes, where):...


# done
def check_unique(table_name, column, value):...


# done
def find_leaf_place(table, value):...


# done
def find_leaf_place_with_condition(table_name , value):...


# done
def insert_into_leaf(cur_node, value, pointer):...


# done
def insert_into_parent(table_name, __node, __key, new_node):...
```

3.5 Buffer

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 BUFFER_SIZE 来设置缓冲区可以包含的记录数。

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: 检查某条记录是否符合 where 条件；

decode: 将字节串按照格式解码为实际的记录；

find_attr_pos: 找到当前的属性是名是第几列；

find_line: 从缓冲区或文件取得对应行数的记录

find_record: 从缓冲区或文件取得符合条件的记录

delete_line: 从缓冲区或文件删除对应行数的记录

delete_record: 从缓冲区或文件删除符合条件的记录

check_unique: 检查待插入记录中 unique 属性的唯一性是否被破坏

insert_record: 插入记录

create_table: 创建新的表文件

drop_table: 删除表文件

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：

```python
import struct
import os
import Catalog
import Index
# from Catalog import tables

# For the convenience of test and presentation, the buffer size is set to a small, static number.
# In real application, we can make it as a var of Buffer object, adjust it according to the size of line,
# making the size of buf (BUFFER_SIZE * line_size) nearly 4KB/8KB for the sake of block transfer.
BUFFER_SIZE = 8  # number of lines
BUFFER_NUM = 2  # for each table


# ---------------------- for easy internal test --------------------------
'''...'''
# ------------------------------------------------------------------------

buffers = {}  # key: name, value: buffer of this table


class Buffer(object):
    def __init__(self, table_name: str):...

    def adjust(self, line_number):...

    def save(self):...

    def is_full(self):...

    def line_is_inside(self, n: int):...


def __initialize__():...



def __finalize__():...


def check(line: list, columns: dict, where: list):...


def decode(format_str: str, line: bytes):...


def find_attr_pos(table_name: str, attribute: str):...


# 'S' 5
# because we store the line number in the index
# we can find a particular line directly
def find_line(table_name: str, line_number: int):...


# 'S' 'gender' '==' 'F'
# def find_record(table_name: str, attribute: str, cond: str, value):
def find_record(table_name: str, columns: dict, where: list):...


def delete_line(table_name: str, line_number: int):...


def delete_record(table_name: str, column: dict, where: list):...


def check_unique(table_name: str, line_size: int, line: bytes):...


def insert_record(table_name: str, record: []):...


def create_table(table_name: str):...


def drop_table(table_name: str):...


def pin_buffer(table_name: str):...


def unpin_buffer(table_name: str):...

'''...'''
```

四、分工说明

宋天泽：interpreter、API、Buffer
李国耀：Index、Catalog、Buffer
江雨辰：Index、Catalog、Buffer
范源颢：interpreter、API、Index

五、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

Catalog 是程序和硬盘的接口，是将表头信息存储到相应的文件中的模块，和硬盘的交互主要是借助了 python 系统中的 os 模块（用于读取和存储硬盘的文件），以及 json 模块

（用于高效率的将存储的表头信息读取出来）。

5.3.1 数据类型

在处理表的时候，我们定义了表的类型（class Table），表中的各个属性也有自己的信息，储存在 class Column（）中。

```
class Table():  # data structure to save a table
    def __init__(self, table_name, pk=0):
        self.table_name = table_name
        self.primary_key = pk

    columns = []


class Column():  # data structure to save an attribute
    def __init__(self, column_name, is_unique, type='char',
length=16):
        self.column_name = column_name
        self.is_unique = is_unique
        self.type = type
        self.length = length
```

5.3.2 文件存储和读取

我们通过__loadfile__(),以及__savefile__()两个函数和将硬盘中的表文件 catalog_index，catalog_table 读入，解析其语法结构，然后写入本地的 Table 和 Column 的变量之中，便于操作。

catalog_table 的文件样例如下：

```
    {"S": {"columns": {"ID": [true, "int", 0], "name": [true, "char",
12], "age": [false, "int", 0], "gender": [false, "char", 1]}, "pk":
0}, "test1": {"columns": {"ID": [true, "int", 0], "name": [true,
"char", 12], "gender": [false, "char", 1]}, "pk": 0}}
```

读取这样的文件我们借助了 python 库中的 json.loads 函数，可以将上述的字符串 Table 类型的字典变量 tables[]之中，相关的函数时__loadfile__()，实现如下：

```
    def __loadfile__():  # from file to memory
    f = open(tablecatalog)
    json_tables = json.loads(f.read())
    for table in json_tables.items():
        temp_name = table[0]
        temp_pk = table[1]['pk']
        temp_columns = []
```

```
        __table = Table(temp_name, temp_pk)  # table_name&primary key
        for __column in table[1]['columns'].items():
            temp_attname = __column[0]
            temp_isunique = __column[1][0]
            temp_type = __column[1][1]
            temp_len = __column[1][2]

            temp_columns.append(Column(temp_attname, temp_isunique,
temp_type, temp_len))
        __table.columns = temp_columns

        tables[temp_name] = __table  # add into the tables dict in
memory
    f.close()

    f = open(indexcatalog)
    json_indices = f.read()
    json_indices = json.loads(json_indices)
    for index in json_indices.items():
        temp_indexname = index[0]  # name of this index
        temp_index = index[1]  # the actual component of this index
        indices[temp_indexname] = temp_index
    f.close()
```

相关操作完成之后，将 tables 字典中的文件转存到文件的函数也可以类似实现，如下：

```
def __savefile__():  # from memory to file
    __tables = {}
    for items in tables.items():
        definition = {}
        temp_name = items[0]
        __columns = {}
        for i in items[1].columns:
            __columns[i.column_name] = [i.is_unique, i.type, i.length]

        definition['columns'] = __columns
        definition['pk'] = items[1].primary_key
        __tables[temp_name] = definition

    j_tables = json.dumps(__tables)
    j_indices = json.dumps(indices)

    f = open(tablecatalog, 'w')
    f.write(j_tables)
    f.close()
```

```
    f = open(indexcatalog, 'w')
    f.write(j_indices)
    f.close()
```

5.3.3 与硬盘的交互

catalog 处理两个文件，catalog_index 和 catalog_table，即表头信息和索引信息，这些内容存放在电脑的硬盘之中，5.3.2 讲述了如何将文件读入为可用的信息，然后这里我们将实现从硬盘中找到文件的功能。

其原理是使用 python 的库 os，根据 os.path 设定文件路径，然后用 open、close 函数打开文件读写。这些功能主要在__initialize__()和__finalize__()函数中完成。

内容如下：

```
    def __initialize__(__path):  # initialize the file of catalog
    global catalogpath
    global tablecatalog
    global indexcatalog
    catalogpath = os.path.join(__path, 'dbfiles/catalogs')
    tablecatalog = os.path.join(catalogpath, 'catalog_table')
    indexcatalog = os.path.join(catalogpath, 'catalog_index')

    if not os.path.exists(catalogpath):
        os.makedirs(catalogpath)

        f1 = open(tablecatalog, 'w')
        f2 = open(indexcatalog, 'w')
        f1.close()
        f2.close()

        __savefile__()
    __loadfile__()


def __finalize__():
    __savefile__()
```

5.3.4 数据操作

如前所述，这里主要处理和表格相关的信息，最基础的是 catalog 创建表的操作，如下：

```
    def create_table(table_name, attributes, pk):
    global tables
```

```
    cur_table = Table(table_name, pk)
    columns = []
    for attr in attributes:
        columns.append(Column(attr['attribute_name'],
                              attr['unique'],
                              attr['type'],
                              attr['type_len']))

    cur_table.columns = columns
    seed = False
    for index, __column in enumerate(cur_table.columns):
        if __column.column_name == cur_table.primary_key:
            cur_table.primary_key = index
            seed = True
            break
    if seed == False:
        raise Exception("primary_key '%s' does not exist."
                        % cur_table.primary_key)

    tables[table_name] = cur_table
```

按照 API 的实现功能，Catalog 还需要检查用户输入的 table 的名称是否存在或者不存在，以便后续操作，函数实现和上文类似，也是通过遍历 tables[]字典实现的。

这一部分模块还负责和检查 select 的听到的属性是否和数据库的表头相合，相关的检查函数是 check_select_statement，实现如下：

```
    def check_select_statement(table_name, attributes, where):
    # raise an exception if something is wrong
    columns = []
    for i in tables[table_name].columns:
        columns.append(i.column_name)
    if where is not None:
        for i in where:
            if i['l_op'] not in columns:
                raise Exception("No column"
                                " name '%s'." % i['l_op'])
    if attributes == ['*']:
        return

    for i in attributes:
        if i not in columns:
            raise Exception("No column name ("No column name '%s'." %
i)
```

我们还设计了 get_column_dic 函数快速获取一个表的属性列表

```
    def get_column_dic(table_name: str):
    result = {}
    cnt = 0
    global tables
    for fullcol in tables[table_name].columns:
        colname = fullcol.column_name
        result[colname] = cnt
        cnt += 1
    return result
```

5.4 Index

Index 是通过 B+树结构存储索引的模块，通过 node 变量存储建立 B+树，B+树首先从硬盘中的文件 indexfile 读取，然后再根据 API 解析出来的命令，进行 B+树的搜寻、查找、删除操作:

5.4.1 数据类型

主要数据类型是 node，存储 B+树的结点，包含键值、指针、是否为叶结点的判定。

```
    class node():
    def __init__(self, isleaf, line0, keys0, pointers0, parent0=''):
        self.is_leaf = isleaf
        self.line = line0
        self.keys = keys0
        self.pointers = pointers0
        self.parent = parent0
```

5.4.2 文件存储和读取

B+树在文件中的存储和上文中 catalog 的存储很类似，也是一段长条形的字符串，一层层顺序排列，如下所示。

```
    {"S": {"is_leaf": true, "line": 0, "keys": []}, "test1":
{"is_leaf": true, "line": [], "keys": []}, "test2": {"is_leaf": true,
"line": [], "keys": []}}
```

读取时也按照上文所述的 json 库实现，但是这里根据文件的特性将函数分拆为两部分，

首先调用__load__,之后在__load__中调用 load_node，具体如下：

```
def __load__():
    global __last_leaf_pointer
    print(recordpath)
    f = open(os.path.join(recordpath, 'indexfile'))
    json_tables = json.loads(f.read())
    f.close()
    for table in json_tables.items():
        temp_name = table[0]
        temp_content = table[1]
        if len(temp_content['keys']) == 0:
            tables[temp_name] = node(True, 0, [], [])
            continue
        tables[temp_name] = \

node(temp_content['is_leaf'],temp_content['line'],temp_content['keys'
], temp_content['pointers'], '')
        if tables[temp_name].is_leaf:
            continue

        tables[temp_name].pointers = \
            load_nodes(temp_content['pointers'], tables[temp_name])
def load_nodes(pointer_list, parent):
    global __last_leaf_pointer
    nodelist = []
    for pointer in pointer_list:

        if pointer['is_leaf']:
            new_node = node(pointer['is_leaf'],
pointer['line'],pointer['keys'], pointer['pointers'], parent)
            nodelist.append(new_node)
            if __last_leaf_pointer == '':
                __last_leaf_pointer = new_node
            else:
                __last_leaf_pointer.pointers.append(new_node)
                __last_leaf_pointer = new_node
        else:
            new_node = node(pointer['is_leaf'],
pointer['line'],pointer['keys'], pointer['pointers'], parent)
            nodelist.append(new_node)
            new_node.pointers = load_nodes(pointer['pointers'],
nodelist[-1])
    return nodelist
```

至于存储，则和 catalog 基本一样：

```
def __store__():
    global recordpath
    __tables = {}
    for table in tables.items():
        __tables[table[0]] = recursive_store_node(table[1])

    f = open(os.path.join(recordpath, 'indexfile'), 'w')
    json_tables = json.dumps(__tables)
    f.write(json_tables)
    f.close()
```

5.4.3 和硬盘的交互

和 catalog 中的思路一样，我们首先通过 os 库确定文件在硬盘中的存储路径（path），之后调用 load，或者__store__

```
def __initialize__(__path):
    global recordpath
    recordpath = os.path.join(__path, 'dbfiles/indices')

    if not os.path.exists(recordpath):
        os.makedirs(recordpath)
        tables['sys'] = node(True, [], ['key0'], [['key0', 'key1'],
''])
        __store__()

    __load__()


def __finalize__():
    __store__()
```

5.4.4 B+树操作

B+树的操作大体和课本的描述一致，这里分别展示一下对于 B+树 insert、find、和 delete 的代码，注意这里我们构造的 B+树是 N=4 的类型，也就是一颗 2-3-4 树

find 方法

通过 B+树内部的索引查找相关数值，首先实现 find_leaf_node，寻找单个叶结点，之后实现 find_leaf_node_with_condition，实现在查找区间，这样就可以实现 select 函数，根据

select 的条件获得返回值。

find_leaf_node:

```
def find_leaf_place(table, value):
    # search on primary key
    cur_node = tables[table]
    while not cur_node.is_leaf:
        seed = False
        for index, key in enumerate(cur_node.keys):
            if key > value:
                cur_node = cur_node.pointers[index]
                seed = True
                break
        if seed == False:
            cur_node = cur_node.pointers[-1]
    return cur_node
```

find_leaf_place_with_condition:

```
def find_leaf_place_with_condition(table_name , value):
    # __primary_key =
CatalogManager.catalog.tables[table].primary_key
    __primary_key = 0
    head_node = tables[table_name]
    first_leaf_node = head_node
    while first_leaf_node.is_leaf != True:
        first_leaf_node = first_leaf_node.pointers[0]
    lists = []
    #op_list = ['<','<=','>','>=','<>','=']
    #if __primary_key == column and condition != op_list[4]:
    while not head_node.is_leaf:
        seed = False
        for index, key in enumerate(head_node.keys):
            if key > value:
                head_node = head_node.pointers[index]
                seed = True
                break
        if seed == False:
            head_node = head_node.pointers[-1]


    for pointer in head_node.pointers[0:-1]:
         if pointer[0] == value:
            lists.append(head_node)

    return lists
```

select_from_table

```
def select_from_table(table_name, attributes, where):
    results = []
    columns = {}
    for i, col in enumerate(Catalog.tables[table_name].columns):
        columns[col.column_name] = i
    __primary_key = Catalog.tables[table_name].primary_key
    # __primary_key = 0
    # columns = {'num': 0, 'val': 1}

    if len(tables[table_name].keys) == 0:
        pass
    else:
        if where is not None:
            nodes = find_leaf_place_with_condition(table_name,
columns[where[0]['l_op']], where[0]['operator'], where[0]['r_op'])
            for cond in where:
                if columns[cond['l_op']] == __primary_key:
                    nodes = find_leaf_place_with_condition(table_name,
columns[cond['l_op']], cond['operator'], cond['r_op'])
                    break
            for __node in nodes:
                for pointer in __node.pointers[0:-1]:
                    if check_conditions(pointer, columns, where):
                        results.append(pointer)
        else:
            first_leaf_node = tables[table_name]
            while first_leaf_node.is_leaf != True:
                first_leaf_node = first_leaf_node.pointers[0]
            while True:
                for i in first_leaf_node.pointers[0:-1]:
                    results.append(i)
                if first_leaf_node.pointers[-1] != '':
                    first_leaf_node = first_leaf_node.pointers[-1]
                else:
                    break

    if attributes[0] == '*':
        __columns_list = list(columns.keys())
        __columns_list_num = list(columns.values())
    else:
        __columns_list = []
        __columns_list_num = []
        for i in range(0,len(attributes)):
```

```
            __columns_list.append(attributes[i])
            __columns_list_num.append(columns(attributes[i]))

    print_select(__columns_list,__columns_list_num)
```

print_select 的实现和 API 中的类似，这里就不赘述了。

insert 方法

主函数是 insert_into_tables，按照 B+树插入法，调用 find_leaf_node，以及 find_leaf_place_with_condition 实现。

主函数 insert_into_tables:

```
def insert_into_table(table_name, __values,line_number:int):

    cur_node = tables[table_name]
    __primary_key = Catalog.tables[table_name].primary_key
    # __primary_key = 0
    if len(cur_node.keys) == 0:
        # new tree
        cur_node.keys.append(__values[__primary_key])

cur_node.pointers.append([__values[__primary_key]]+[line_number])
        cur_node.pointers.append('')
        print('Successfully insert into table %s,' % table_name,
end='')
        return

    cur_node = find_leaf_place(table_name, __values[__primary_key])
    if len(cur_node.keys) < N - 1:
        insert_into_leaf(cur_node, __values[__primary_key],
[__values[__primary_key]]+[line_number])

    else:
        insert_into_leaf(cur_node, __values[__primary_key],
[__values[__primary_key]]+[line_number])
        new_node = node(True, [],[], [])
        tmp_keys = cur_node.keys
        tmp_pointers = cur_node.pointers
        cur_node.keys = []
        cur_node.pointers = []
        for i in range(math.ceil(N / 2)):
            cur_node.keys.append(tmp_keys.pop(0))
```

```
            cur_node.pointers.append(tmp_pointers.pop(0))
        for i in range(N - math.ceil(N / 2)):
            new_node.keys.append(tmp_keys.pop(0))
            new_node.pointers.append(tmp_pointers.pop(0))
        cur_node.pointers.append(new_node)
        new_node.pointers.append(tmp_pointers.pop(0))
        insert_into_parent(table_name, cur_node, new_node.keys[0],
new_node)

    print('Successfully insert into table %s,' % table_name, end='')
```

insert_into_leaf，用于在叶结点上插入值

```
def insert_into_leaf(cur_node, value, pointer):
    for index, key in enumerate(cur_node.keys):
        if key == value:
            raise Exception("Index Module : primary_key already
exists.")
        if key > value:
            cur_node.pointers.insert(index, pointer)
            cur_node.keys.insert(index, value)
            return
    cur_node.pointers.insert(len(cur_node.keys), pointer)
    cur_node.keys.insert(len(cur_node.keys), value)
```

insert_into_parent,用于在父节点中增加必要的新的值，递归实现

```
def insert_into_parent(table_name, __node, __key, new_node):
    if __node.parent == '':
        cur_node = node(False, [], [],[], '')
        cur_node.pointers.append(__node)
        cur_node.pointers.append(new_node)
        cur_node.keys.append(__key)
        __node.parent = cur_node
        new_node.parent = cur_node
        tables[table_name] = cur_node
    else:
        p = __node.parent
        if len(p.pointers) < N:
            seed = False
            for index, key in enumerate(p.keys):
                if __key < key:
                    p.keys.insert(index, __key)
                    p.pointers.insert(index + 1, new_node)
                    seed = True
```

```
                break
        if seed == False:
            p.keys.append(__key)
            p.pointers.append(new_node)
        new_node.parent = p
    else:
        seed = False
        for index, key in enumerate(p.keys):
            if __key < key:
                p.keys.insert(index, __key)
                p.pointers.insert(index + 1, new_node)
                seed = True
                break
        if seed == False:
            p.keys.append(__key)
            p.pointers.append(new_node)
        __new_node = node(False, [], [],[])
        tmp_keys = p.keys
        tmp_pointers = p.pointers
        p.keys = []
        p.pointers = []
        for i in range(math.ceil(N / 2)):
            p.keys.append(tmp_keys.pop(0))
            p.pointers.append(tmp_pointers.pop(0))
        p.pointers.append(tmp_pointers.pop(0))
        k__ = tmp_keys.pop(0)
        for i in range(N - math.ceil(N / 2) - 1):
            __new_node.keys.append(tmp_keys.pop(0))
            __tmp = tmp_pointers.pop(0)
            __tmp.parent = __new_node
            __new_node.pointers.append(__tmp)
        __tmp = tmp_pointers.pop(0)
        __tmp.parent = __new_node
        __new_node.pointers.append(__tmp)
        new_node.parent = __new_node
        insert_into_parent(table_name, p, k__, __new_node)
```

delete 方法

主要分为两步，首先将用户要求 delete 的值从树中减去，然后通过一个专门的函数 maintain_B_plus_tree_after_delete 来维持删除后 B+树的性质。

删除的主函数:

```
def delete_from_table(table_name, pk):
    # delete rows from table according to the statement's condition
    # usage : find_leaf_place_with_condition(table, column,
condition, value)
    #[{'operator': '=', 'l_op': 'sno', 'r_op': '88888888'}]

    print(pk)
    print("pk printed")
    times = 0
    for eachpk in pk:
        #nodes = find_leaf_place_with_condition(table_name,
columns[where[0]['l_op']], where[0]['operator'], where[0]['r_op'])
        nodes = find_leaf_place(table_name,eachpk)



        #if len(nodes) == 0:
         #   break
        seed = False
        #for __node in nodes:

          # if seed == True:
              # break
        __node=nodes;
        for index, leaf in enumerate(__node.pointers[0:-1]):
           #if check_conditions(leaf, columns, where):
           if(leaf[0]==eachpk):
               __node.pointers.pop(index)
               __node.keys.pop(index)
               maintain_B_plus_tree_after_delete(table_name, __node)
               times = times + 1
              # seed = True
              # break
        #if seed == False:
          # break
    print("Index:Successfully deleted")
```

删除之后维持 B+树的性质

```
def maintain_B_plus_tree_after_delete(table, __node):
    global N
    if __node.parent == '' and len(__node.pointers) == 1:
        __node.pointers = []
    elif ((len(__node.pointers) < math.ceil(N / 2) and __node.is_leaf
== False) or
```

```
        (len(__node.keys) < math.ceil((N - 1) / 2) and
__node.is_leaf == True)) \
          and __node.parent != '':
       previous = False
       other_node = node(True,[], [], [])
       K = ''
       __index = 0
       for index, i in enumerate(__node.parent.pointers):
           if i == __node:
               if index == len(__node.parent.pointers) - 1:
                   other_node = __node.parent.pointers[-2]
                   previous = True
                   K = __node.parent.keys[index - 1]
               else:
                   K = __node.parent.keys[index]
                   other_node = __node.parent.pointers[index + 1]
                   __index = index + 1

       if (other_node.is_leaf == True and len(other_node.keys) +
len(__node.keys) < N) or \
               (other_node.is_leaf == False and
len(other_node.pointers) +
                len(__node.pointers) <= N):
           if previous == True:
               if other_node.is_leaf == False:
                   other_node.pointers = other_node.pointers +
__node.pointers
                   other_node.keys = other_node.keys + [K] +
__node.keys
                   for __node__ in __node.pointers:
                       __node__.parent = other_node
               else:
                   other_node.pointers = other_node.pointers[0:-1]
                   other_node.pointers = other_node.pointers +
__node.pointers
                   other_node.keys = other_node.keys + __node.keys
               __node.parent.pointers = __node.parent.pointers[0:-1]
               __node.parent.keys = __node.parent.keys[0:-1]
               maintain_B_plus_tree_after_delete(table, __node.parent)
           else:
               if other_node.is_leaf == False:
                   __node.pointers = __node.pointers +
other_node.pointers
                   __node.keys = __node.keys + [K] + other_node.keys
```

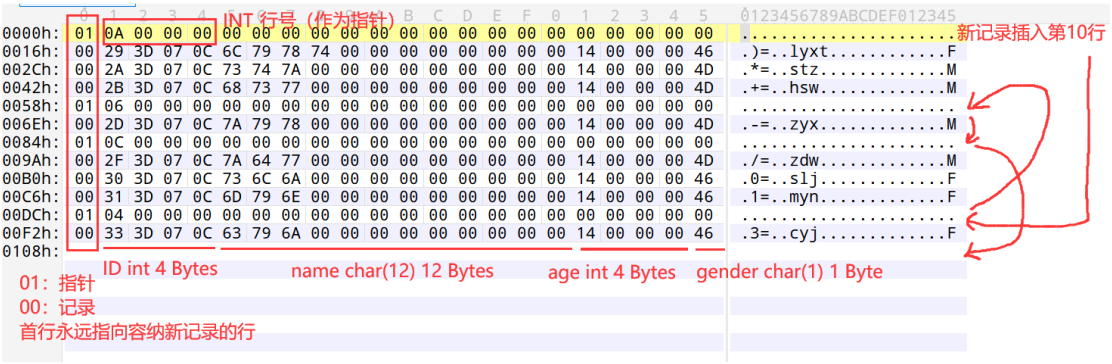
```
                for __node__ in other_node.pointers:
                    __node__.parent = __node
            else:
                __node.pointers = __node.pointers[0:-1]
                __node.pointers = __node.pointers +
other_node.pointers
                __node.keys = __node.keys + other_node.keys
            __node.parent.pointers.pop(__index)
            __node.parent.keys.pop(__index - 1)
            maintain_B_plus_tree_after_delete(table, __node.parent)
        else:
            if previous == True:
                if other_node.is_leaf == True:
                    __node.keys.insert(0, other_node.keys.pop(-1))
                    __node.pointers.insert(0, other_node.pointers.pop(-
2))
                    __node.parent.keys[-1] = __node.keys[0]
                else:
                    __tmp = other_node.pointers.pop(-1)
                    __tmp.parent = __node
                    __node.pointers.insert(0, __tmp)
                    __node.keys.insert(0, __node.parent.keys[-1])
                    __node.parent.keys[-1] = other_node.keys.pop(-1)
            else:
                if other_node.is_leaf == True:
                    __node.keys.insert(-1, other_node.keys.pop(0))
                    __node.pointers.insert(-2,
other_node.pointers.pop(0))
                    __node.parent.keys[__index - 1] = other_node.keys[0]
                else:
                    __tmp = other_node.pointers.pop(0)
                    __tmp.parent = __node
                    __node.pointers.insert(-1, __tmp)
                    __node.keys.insert(-1, __node.parent.keys[__index -
1])
                    __node.parent.keys[__index - 1] =
other_node.keys.pop(0)
```

5.5 Buffer

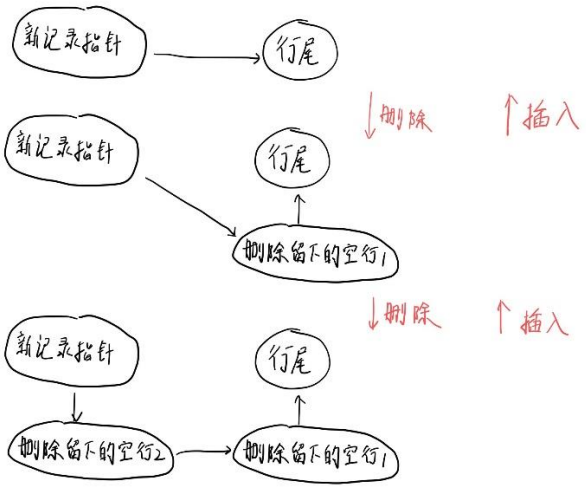
buffer 模块通过 python 的 struct 包，将记录转换为二进制字节串，直接储存在二进制文件中。这种方法储存的文件密度高，且由于我们储存的是定长记录，通过行数*每行的字节数可以很方便地找到某条记录的偏移位置，方便按照行数直接建立索引，快速查询记录。

5.5.1 记录储存方式

接下来以 S(ID int, name char(12) unique, age int, gender char(1), primary key (ID)) 为例展示记录在文件中的储存形式。



首先，在创建表时，我们会根据表中各个属性的大小决定二进制文件中“一行”的字节数。（注意，由于采用二进制文件进行储存，这里的“一行”不是文本文件意义上的一行（如以 CRLF 作为行的结尾），而是人为划定字节个数的、储存一条记录的“一行”，后文若无特别说明，均为这种含义）。int、float 为 4 字节（且使用小端存储），char 按照创建表时规定的字节大小，再加上用来指示本行是记录还是指针的一个字节，即为一行的字节数。若属性所占字节数小于 4，也会补成 4，以在不储存记录的时候给指针提供 4 个字节的空间。图中一行的字节数为 22。之后，文件的首行固定为指针行，储存容纳新记录的行的行号（为方便，以后称作新记录指针）。

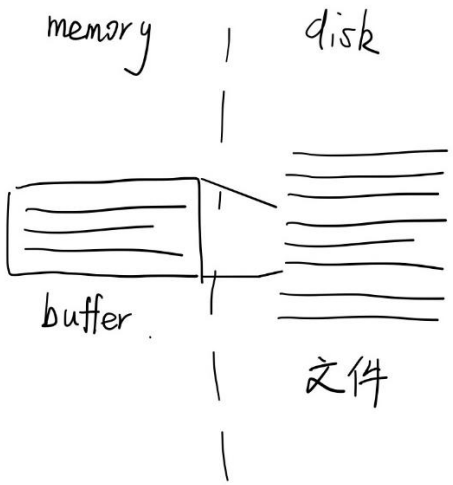


由于我们要按照行数建立索引，因此记录所在的行号是不方便改动的，因此我们采用指针将所有删除的行串起来，并且越晚删除的行越靠近链头。当我们删除记录时，我们将该行清空，首字节变 01 表示指针行，在 2~5 个字节中指向当前新记录指针指向的位置，再将新记录指针指向该行。这样当我们插入记录时，可以首先填补空行，而不是一味地在行尾插入，中间留下许多“空洞”。以上的删除也可以连续进行多次，新删除的记录总会变成指针指向老的空行，形成一个链表。当我们插入新记录时，首先按照新记录指针跳到该行，并检查其中内容。如果是文件末尾，那么直接插入即可，同时新记录指针+1。如果不是文件末尾，而是一个指针行，那么我们先将储存的行号转存到新记录指针里（这样下次插入就可以填充链表里的第下个空行了），在存入相应的记录。

简单来说，我们利用删除的记录留下的空间，维护一个所有空行的链表，使得对一些记录的操作不会影响其他记录的行号，同时又能填补穿插在记录之间的空行。而且只通过链头的指针进行操作，保证找到空行只需要常数的时间。

5.5.2 buffer 与文件的关系

在程序运行的过程中，文件的一部分内容是放在程序内部的结构里的。这些内容存在于内存之中，因而存取较快。下面就将展示 buffer 是如何设计的。



buffer 容纳了文件中连续的几行，因此我们有必要记录 buffer 从文件的哪行开始，记录了几行，其内容是什么。为了正确地解释从文件中读取的字节，我们还需要记录一行的大小以及一个格式码，用来编码、解码。同时，由于 buffer 不一定会容纳文件的首行，我们还要单独储存新记录指针，这样在插入新行时不需要将文件指针倒回文件开头查询。再加上指示 buffer 是否 dirty、是否被锁的两个布尔变量，以及 buffer 所属的表名，就构成了 buffer 的所有成员。

除了成员，buffer 也需要一些操作。构造函数负责按文件最开头几行的内容初始化 buffer，按照 catalog 模块的信息构建格式码、行大小。save 函数负责将 buffer 的改动保存到文件中。adjust 负责调整 buffer 储存的内容（从文件的第几行开始）。同时还要 is_full, line_is_inside 两个辅助函数，用来返回 buffer 是否已满、以及某行号对应的行是否储存在 buffer 之中。

以下是 buffer 的结构体

```
class Buffer(object):
    def __init__(self, table_name: str):
        self.table_name = table_name
        self.file_line = 0  # the position of buffer's first line in
actual file
        self.is_dirty = False
        self.pin = False
        self.format_list = ['<c']
        self.line_size = 1  # we have a flag of 1 byte indicating
record (0x00) or pointer (0x01)
```

```
        # get the format char to support pack and unpack
        for column in Catalog.tables[table_name].columns:
            if column.type == 'int':
                self.format_list += ['i']
                self.line_size += 4
            elif column.type == 'float':
                self.format_list += ['f']
                self.line_size += 4
            elif column.type == 'char':
                self.format_list += [f'{column.length}s']
                self.line_size += column.length
        if self.line_size < 5:
            self.line_size = 5  # we need 4 extra bytes at least to
store the empty line pointer

        f = open(f'dbfiles/table_files/table_{self.table_name}.bin',
'rb')
        self.buf_size = BUFFER_SIZE  # as I mentioned before, it can
be adjusted according to the real application
        # self.buf_size = 4096 // self_size  # such as this
        self.cur_size = BUFFER_SIZE
        self.content = []

        # fill the buffer
        for i in range(self.buf_size):
            line = f.read(self.line_size)
            if line == b'':
                self.cur_size = i
                break
            self.content.append(line)
        f.close()
        self.ins_pos = struct.unpack('<I', self.content[0][1:5])[0]

    def adjust(self, line_number):
        if self.pin:
            raise Exception("Locked buffer is not allowed to replace!")
        if self.is_dirty:
            self.save()
        f = open(f'dbfiles/table_files/table_{self.table_name}.bin',
'rb')
        # set a different start point
        f.seek(line_number * self.line_size)
        self.cur_size = BUFFER_SIZE
        self.content = []
```

```
        for i in range(self.buf_size):
            line = f.read(self.line_size)
            if line == b'':
                self.cur_size = i
                break
            self.content.append(line)
        self.file_line = line_number
        f.close()

    def save(self):
        f = open(f'dbfiles/table_files/table_{self.table_name}.bin',
'rb+')
        f.seek(self.file_line * self.line_size)
        for line in self.content:
            f.write(line)
        # save the insert position as well
        f.seek(0)
        f.write(struct.pack(f'<cI{self.line_size - 5}s', b'\x01',
self.ins_pos, b'\x00' * (self.line_size - 5)))
        f.close()
        self.is_dirty = False

    def is_full(self):
        return self.cur_size == self.buf_size

    def line_is_inside(self, n: int):
        return self.file_line <= n < self.file_line + self.cur_size
```

5.5.3 数据处理操作（insert、delete、find）

有了前面的基础操作，数据处理也就可以方便地实现。例如 find_line 函数返回文件的某一行，可以先通过 buffer 的 line_is_inside 方法检查这一行是否在 buffer 中，如果在可以直接快速返回。如果不在，就将 buffer 通过 adjust 调整到这一行，再进行读取。insert_record, delete_line 的方法也类似，只不过加入了对 buffer 记录指针的修改，插入的时候还要检查 unique 性质是否保持。find_line, delete_line 功能是按照条件搜索，所以 buffer 和文件中的记录都要搜索，注意在搜索文件时到 buffer 已经存储的行号时，就进入 buffer 搜索，以免读了已被改动的数据，同时保持记录在文件里的存放顺序。其实数据处理操作的细节还有很多，比如 dirty 置位、编码解码、空 buffer 插入等等，这里不再赘述。以下是各种操作的代码。

我们先给出 find_line 的代码

```
def find_line(table_name: str, line_number: int):
```

```
    global buffers
    buffer = buffers[table_name]
    # if the line is not in buffer, we need to fetch the page first
    if not buffer.line_is_inside(line_number):
        buffer.adjust(line_number)
        if not buffer.line_is_inside(line_number):
            raise Exception("The line you want to retrieve exceeds the
file.")

    line = buffer.content[line_number - buffer.file_line]
    if line[0] == 1:
        raise Exception("The line you want to retrieve is not
existed.")
    line = decode(''.join(buffer.format_list), line)
    return line
```

注意其中对于 line_is_inside，adjust 的使用。

相关的 delete_line 如下：

```
def delete_line(table_name: str, line_number: int):
    global buffers
    buffer = buffers[table_name]
    remain = struct.pack(f'<cI{buffer.line_size - 5}s', b'\x01',
buffer.ins_pos, b'\x00' * (buffer.line_size - 5))
    # if the line is not in buffer, we need to fetch the page first
    if not buffer.line_is_inside(line_number):
        buffer.adjust(line_number)
        if not buffer.line_is_inside(line_number):
            raise Exception("The line you want to retrieve exceeds the
file.")

    if buffer.content[line_number - buffer.file_line][0] == 1:
        raise Exception("The line you want to retrieve is not
existed.")
    buffer.content[line_number - buffer.file_line] = remain
    buffer.is_dirty = True
    buffer.ins_pos = line_number
```

对于 record 的处理也就比较简单了，首先在 Buffer 中查找是否有这样的二进制串，如若没有，进入 disk 查找。

比如如下的 find

```
def find_record(table_name: str, columns: dict, where: list):
    # first find them in buffer
    global buffers
    buffer = buffers[table_name]
```

```
    results = []
    buffer_range = range(buffer.file_line, buffer.file_line +
buffer.cur_size)
    # then find them in file (no fetch)
    f = open(f'dbfiles/table_files/table_{table_name}.bin', 'rb')
    f.seek(buffer.line_size)
    i = 0
    while 1:
        i += 1
        # skip the line already scanned in the buffer
        if i in buffer_range:
            # buffer 里面的查找放在这里 保证记录整体顺序
            for line in buffer.content:
                if line[0] == 1:
                    continue
                line = decode(''.join(buffer.format_list), line)
                if check(line, columns, where):
                    results += [line]

            i = buffer_range[-1]
            f.seek(buffer.line_size * (i + 1))
            continue
        line = f.read(buffer.line_size)
        # reach EOF
        if line == b'':
            f.close()
            break
        # a pointer line
        if line[0] == 1:
            continue
        else:
            line = decode(''.join(buffer.format_list), line)
            if check(line, columns, where):
                results += [line]
    return results
```

还有 delete 同理:

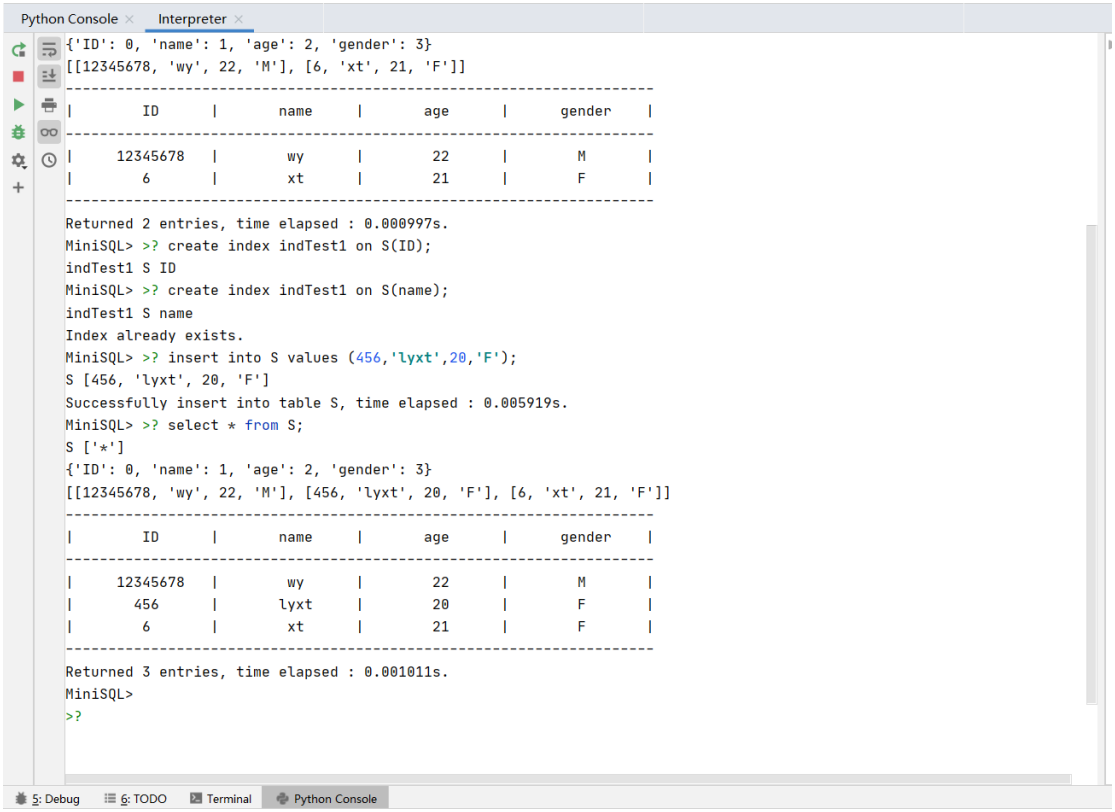
```
def delete_record(table_name: str, column: dict, where: list):
    pk_pos = Catalog.tables[table_name].primary_key
    # first search in the buffer
    global buffers
    buffer = buffers[table_name]
    buffer_range = range(buffer.file_line, buffer.file_line +
buffer.cur_size)
    deleted_pks = []
```

```
    for i, line in enumerate(buffer.content):
        if line[0] == 1:
            continue
        line = decode(''.join(buffer.format_list), line)
        if check(line, column, where):
            deleted_pks += [line[pk_pos]]
            buffer.content[i] = \
                struct.pack(f'<cI{buffer.line_size - 5}s',
                            b'\x01', buffer.ins_pos,
                            b'\x00' * (buffer.line_size - 5))
            buffer.ins_pos = buffer.file_line + i
            buffer.is_dirty = True
    # then search in the file (no fetch)
    f = open(f'dbfiles/table_files/table_{table_name}.bin', 'rb+')
    f.seek(buffer.line_size)
    i = 0
    while 1:
        i += 1
        if i in buffer_range:
            i = buffer_range[-1]
            f.seek(buffer.line_size * (i + 1))
            continue
        line = f.read(buffer.line_size)
        if line == b'':
            f.close()
            break
        if line[0] == 1:
            continue
        else:
            line = decode(''.join(buffer.format_list), line)
            if check(line, column, where):
                deleted_pks += [line[pk_pos]]
                f.seek(buffer.line_size * i)
                f.write(
                    struct.pack(f'<cI{buffer.line_size - 5}s',
                                b'\x01',
                                buffer.ins_pos,
                                b'\x00' * (buffer.line_size - 5)))
                buffer.ins_pos = i
    return deleted_pks
    # when you doing conditional delete, first call
Buffer.delete_record,
    # it will tell you pks of the records which are deleted
```

六、界面说明

界面使用的是 win10 内置的 Linux 终端界面，Apple 可以直接在 vim 中运行，使用 vim 的界面

如果是使用 pycharm，则可以调用 pycharm 内部的 python Console 终端，显示效果如下：

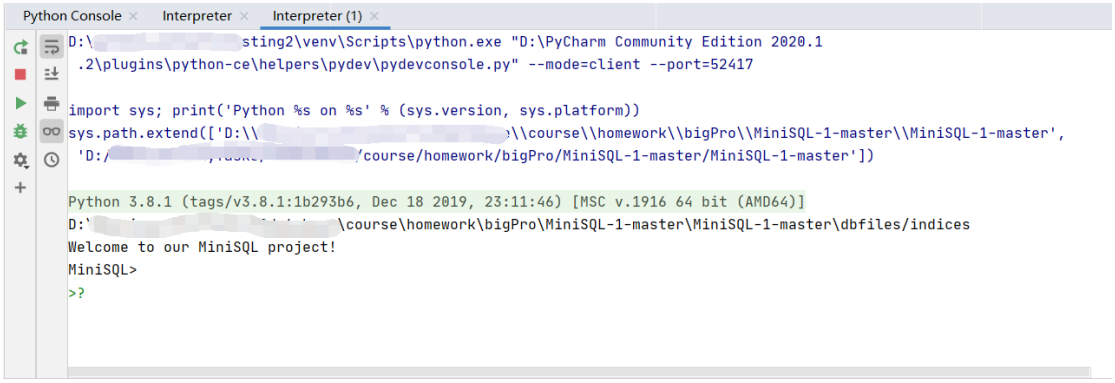


七、系统测试

首先进行完备性测试，我们以一个 test1 的表作为实验体。

开始运行

在 pycharm 中的 interpreter 文件中点击运行即可开始



表格创建 create

```
Welcome to our MiniSQL project!
MiniSQL> >? create table test(ID int,name char(12) unique, age int, primary key(ID));
test [{'attribute_name': 'ID', 'type': 'int', 'type_len': 0, 'unique': True}, {'attribute_name': 'name', 'type':
 'char', 'type_len': 12, 'unique': True}, {'attribute_name': 'age', 'type': 'int', 'type_len': 0, 'unique': False}] ID
Successfully create table 'test', time elapsed : 0.000996s.
MiniSQL>
>?
```

创建成功，注意我们将其 catalog_table 的内容和创建用时反馈了出来。

插入记录 insert

```
MiniSQL> >? insert into test values(1,'stz',18);
test [1, 'stz', 18]
Successfully insert into table test, time elapsed : 0.000950s.
MiniSQL> >? insert into test values(2,'jyc',19);
test [2, 'jyc', 19]
Successfully insert into table test, time elapsed : 0.001049s.
MiniSQL> >? insert into test values(3,'lgy',20);
test [3, 'lgy', 20]
Successfully insert into table test, time elapsed : 0.001920s.
MiniSQL> >? insert into test values(4,'fyh',18);
test [4, 'fyh', 18]
Successfully insert into table test, time elapsed : 0.002102s.
MiniSQL> >? insert into test values(4,'motoka',18);
test [4, 'motoka', 18]
Unique constraint is not conserved.
MiniSQL> >? insert into test values(5,'lgy',100);
test [5, 'lgy', 100]
Unique constraint is not conserved.
MiniSQL> >? insert into test values('homura',6,100);
test ['homura', 6, 100]
invalid literal for int() with base 10: 'homura'
MiniSQL>
>?
```

如上所述，这里总共成功插入了四条记录，最后三条是错误示范，分别违背了 unique 唯一性原则，和插入数值和表头对应的原则。

搜索记录 select

```
MiniSQL> >? insert into test values('homura',6,100);
test ['homura', 6, 100]
invalid literal for int() with base 10: 'homura'
MiniSQL> >? select * from test;
test ['*']
{'ID': 0, 'name': 1, 'age': 2}
[[1, 'stz', 18], [2, 'jyc', 19], [3, 'lgy', 20], [4, 'fyh', 18]]
-----------------------------------------------------
|        ID      |       name     |       age      |
-----------------------------------------------------
|        1       |       stz      |        18      |
|        2       |       jyc      |        19      |
|        3       |       lgy      |        20      |
|        4       |       fyh      |        18      |
-----------------------------------------------------
Returned 4 entries, time elapsed : 0.001991s.
MiniSQL> >? select name from test where age < 19;
test ['name'] [{'operator': '<', 'l_op': 'age', 'r_op': 19}]
{'ID': 0, 'name': 1, 'age': 2}
[[1, 'stz', 18], [4, 'fyh', 18]]
name
------------------
|       name     |
------------------
|       stz      |
|       fyh      |
------------------
Returned 2 entries, time elapsed : 0.000996s.
MiniSQL>
>?
```

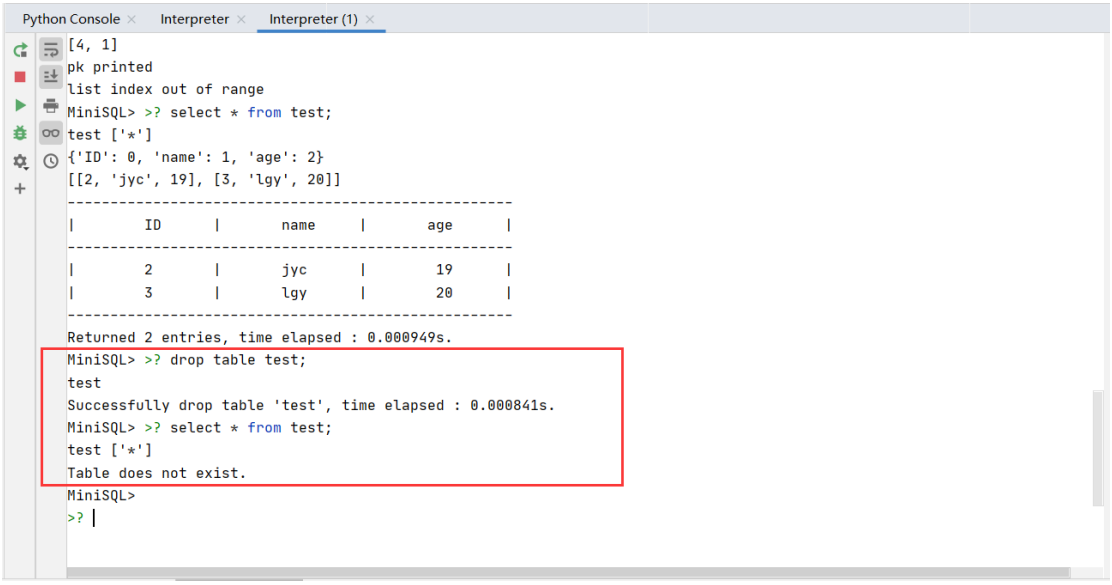
如图测试了 select * ， select 。。。where。。。的语法，均可以正常返回。

删除记录 delete

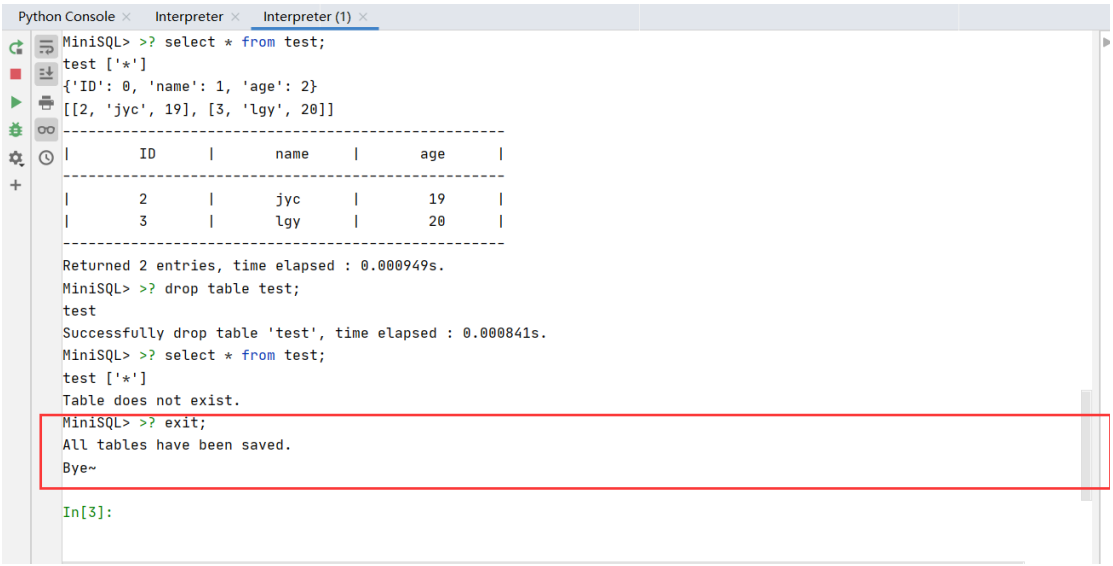
```
|       stz      |
|       fyh      |
------------------
Returned 2 entries, time elapsed : 0.000996s.
MiniSQL> >? delete from test where age = 18;
test [{'operator': '=', 'l_op': 'age', 'r_op': 18}]
[4, 1]
pk printed
list index out of range
MiniSQL> >? select * from test;
test ['*']
{'ID': 0, 'name': 1, 'age': 2}
[[2, 'jyc', 19], [3, 'lgy', 20]]
-----------------------------------------------------
|        ID      |       name     |       age      |
-----------------------------------------------------
|        2       |       jyc      |        19      |
|        3       |       lgy      |        20      |
-----------------------------------------------------
Returned 2 entries, time elapsed : 0.000949s.
MiniSQL>
>?
```

测试了 delete 和 select 的结果，可以返回正常的结果。

删除表格和退出程序



关于 drop 的命令正常。



我们用 exit 或 quit 实现退出。

脚本文件

```
drop table S;
create table S(ID int, name char(12) unique, age int, gender
char(1),primary key (ID));
# skip this line
```

```
insert into S values(1,'stz',20,'M');
insert into S values(2,'jyc',19,'M');
insert into S values(3,'lgy',20,'M');
insert into S values(4,'fyh',19,'M');
insert into S values(5,'homura',500,'F');
insert into S values(6,'Motoka',600,'F');
insert into S values(7,'AAB',19,'F');
insert into S values(8,'AAC',17,'F');
insert into S values(9,'AAD',23,'F');
insert into S values(10,'AAE',13,'F');
insert into S values(11,'AAF',10,'M');
insert into S values(12,'AAG',10,'F');
insert into S values(13,'AAH',12,'M');
insert into S values(14,'AAI',13,'M');
insert into S values(15,'AAJ',25,'F');
insert into S values(16,'AAK',22,'M');
insert into S values(17,'AAL',18,'M');
insert into S values(18,'AAM',20,'M');
insert into S values(19,'AAN',24,'F');
insert into S values(20,'AAO',30,'M');
insert into S values(21,'AAP',25,'F');
insert into S values(22,'AAQ',13,'F');
insert into S values(23,'AAR',18,'M');
insert into S values(24,'AAS',30,'M');
insert into S values(25,'AAT',10,'M');
insert into S values(26,'AAU',15,'F');
insert into S values(27,'AAV',23,'M');
insert into S values(28,'AAW',10,'F');
insert into S values(29,'AAX',17,'M');
insert into S values(30,'AAY',11,'M');
insert into S values(31,'AAZ',12,'F');
insert into S values(32,'ABA',11,'F');
insert into S values(33,'ABB',24,'F');
insert into S values(34,'ABC',29,'M');
insert into S values(35,'ABD',25,'M');
insert into S values(36,'ABE',13,'F');
insert into S values(37,'ABF',12,'M');
insert into S values(38,'ABG',30,'F');
insert into S values(39,'ABH',10,'M');
insert into S values(40,'ABI',27,'M');
insert into S values(41,'ABJ',23,'F');
insert into S values(42,'ABK',14,'M');
insert into S values(43,'ABL',20,'F');
insert into S values(44,'ABM',23,'M');
```

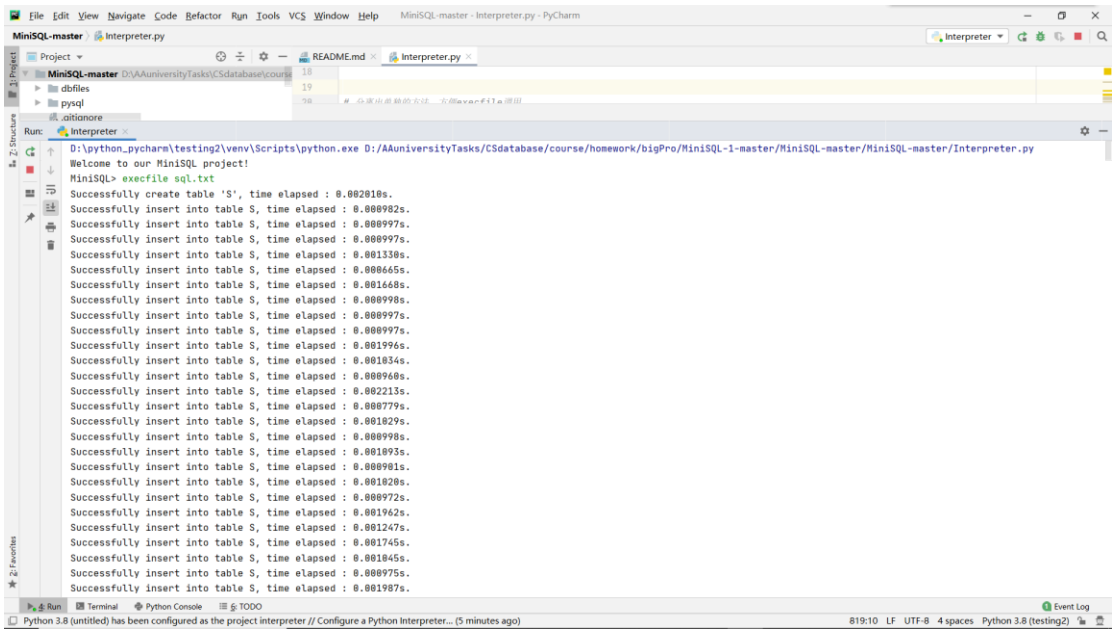
```
insert into S values(45,'ABN',17,'M');
insert into S values(46,'ABO',12,'M');
insert into S values(47,'ABP',13,'M');
insert into S values(48,'ABQ',23,'M');
insert into S values(49,'ABR',10,'F');
insert into S values(50,'ABS',28,'F');
insert into S values(51,'ABT',16,'F');
insert into S values(52,'ABU',12,'M');
insert into S values(53,'ABV',29,'M');
insert into S values(54,'ABW',15,'M');
insert into S values(55,'ABX',25,'M');
insert into S values(56,'ABY',11,'M');
insert into S values(57,'ABZ',12,'F');
insert into S values(58,'ACA',28,'F');
insert into S values(59,'ACB',23,'M');
insert into S values(60,'ACC',17,'M');
insert into S values(61,'ACD',12,'F');
insert into S values(62,'ACE',20,'F');
insert into S values(63,'ACF',21,'F');
insert into S values(64,'ACG',24,'F');
insert into S values(65,'ACH',24,'M');
insert into S values(66,'ACI',15,'M');
insert into S values(67,'ACJ',13,'F');
insert into S values(68,'ACK',15,'M');
insert into S values(69,'ACL',15,'M');
insert into S values(70,'ACM',14,'M');
insert into S values(71,'ACN',25,'M');
insert into S values(72,'ACO',27,'M');
insert into S values(73,'ACP',15,'F');
insert into S values(74,'ACQ',23,'F');
insert into S values(75,'ACR',19,'M');
insert into S values(76,'ACS',17,'M');
insert into S values(77,'ACT',27,'M');
insert into S values(78,'ACU',29,'F');
insert into S values(79,'ACV',22,'F');
insert into S values(80,'ACW',11,'F');
insert into S values(81,'ACX',21,'F');
insert into S values(82,'ACY',20,'F');
insert into S values(83,'ACZ',18,'F');
insert into S values(84,'ADA',12,'F');
insert into S values(85,'ADB',18,'M');
insert into S values(86,'ADC',26,'F');
insert into S values(87,'ADD',15,'F');
insert into S values(88,'ADE',25,'F');
```

```
insert into S values(89,'ADF',17,'F');
insert into S values(90,'ADG',22,'M');
insert into S values(91,'ADH',16,'F');
insert into S values(92,'ADI',14,'M');
insert into S values(93,'ADJ',29,'M');
insert into S values(94,'ADK',27,'M');
insert into S values(95,'ADL',26,'F');
insert into S values(96,'ADM',15,'M');
insert into S values(97,'ADN',14,'F');
insert into S values(98,'ADO',13,'M');
insert into S values(99,'ADP',20,'M');
insert into S values(100,'ADQ',28,'M');
insert into S values(101,'ADR',11,'F');
insert into S values(102,'ADS',16,'F');
insert into S values(103,'ADT',24,'F');
insert into S values(104,'ADU',11,'M');
insert into S values(105,'ADV',16,'M');
insert into S values(106,'ADW',28,'F');
insert into S values(107,'ADX',13,'M');
insert into S values(108,'ADY',23,'F');
insert into S values(109,'ADZ',22,'F');
insert into S values(110,'AEA',24,'F');
insert into S values(111,'AEB',23,'M');
insert into S values(112,'AEC',28,'M');
insert into S values(113,'AED',25,'M');
insert into S values(114,'AEE',17,'M');
insert into S values(115,'AEF',12,'F');
insert into S values(116,'AEG',11,'F');
insert into S values(117,'AEH',13,'F');
insert into S values(118,'AEI',12,'M');
insert into S values(119,'AEJ',18,'F');
insert into S values(120,'AEK',19,'F');
insert into S values(121,'AEL',19,'F');
insert into S values(122,'AEM',14,'M');
insert into S values(123,'AEN',23,'F');
insert into S values(124,'AEO',22,'F');
insert into S values(125,'AEP',27,'F');
insert into S values(126,'AEQ',27,'F');
insert into S values(127,'AER',12,'M');
insert into S values(128,'AES',27,'F');
insert into S values(129,'AET',24,'M');
insert into S values(130,'AEU',30,'F');
insert into S values(131,'AEV',29,'F');
insert into S values(132,'AEW',18,'M');
```

```
insert into S values(133,'AEX',13,'F');
insert into S values(134,'AEY',16,'M');
insert into S values(135,'AEZ',18,'M');
insert into S values(136,'AFA',24,'M');
insert into S values(137,'AFB',10,'F');
insert into S values(138,'AFC',23,'F');
insert into S values(139,'AFD',28,'M');
insert into S values(140,'AFE',13,'M');
insert into S values(141,'AFF',20,'M');
insert into S values(142,'AFG',16,'M');
insert into S values(143,'AFH',19,'M');
insert into S values(144,'AFI',25,'F');
insert into S values(145,'AFJ',11,'M');
insert into S values(146,'AFK',20,'M');
insert into S values(147,'AFL',16,'M');
insert into S values(148,'AFM',26,'M');
insert into S values(149,'AFN',13,'F');
insert into S values(150,'AFO',25,'M');
insert into S values(151,'AFP',24,'M');
insert into S values(152,'AFQ',13,'M');
insert into S values(153,'AFR',19,'F');
insert into S values(154,'AFS',23,'M');
insert into S values(155,'AFT',29,'M');
insert into S values(156,'AFU',19,'F');
insert into S values(157,'AFV',15,'M');
insert into S values(158,'AFW',12,'F');
insert into S values(159,'AFX',15,'F');
insert into S values(160,'AFY',25,'M');
insert into S values(161,'AFZ',18,'M');
insert into S values(162,'AGA',25,'M');
insert into S values(163,'AGB',20,'F');
insert into S values(164,'AGC',24,'M');
insert into S values(165,'AGD',21,'M');
insert into S values(166,'AGE',20,'M');
insert into S values(167,'AGF',18,'M');
insert into S values(168,'AGG',24,'M');
insert into S values(169,'AGH',10,'M');
insert into S values(170,'AGI',23,'F');
insert into S values(171,'AGJ',14,'M');
insert into S values(172,'AGK',23,'M');
insert into S values(173,'AGL',18,'M');
insert into S values(174,'AGM',16,'M');
insert into S values(175,'AGN',22,'M');
insert into S values(176,'AGO',21,'F');
```

```
insert into S values(177,'AGP',15,'F');
insert into S values(178,'AGQ',21,'F');
insert into S values(179,'AGR',11,'M');
insert into S values(180,'AGS',21,'M');
insert into S values(181,'AGT',28,'F');
insert into S values(182,'AGU',14,'M');
insert into S values(183,'AGV',30,'M');
insert into S values(184,'AGW',21,'M');
insert into S values(185,'AGX',20,'F');
insert into S values(186,'AGY',28,'F');
insert into S values(187,'AGZ',22,'M');
insert into S values(188,'AHA',30,'M');
insert into S values(189,'AHB',30,'M');
insert into S values(190,'AHC',15,'F');
insert into S values(191,'AHD',18,'M');
insert into S values(192,'AHE',11,'M');
insert into S values(193,'AHF',11,'F');
insert into S values(194,'AHG',25,'M');
insert into S values(195,'AHH',29,'F');
insert into S values(196,'AHI',14,'F');
insert into S values(197,'AHJ',21,'F');
insert into S values(198,'AHK',19,'M');
insert into S values(199,'AHL',28,'F');
insert into S values(200,'AHM',23,'F');
#
create index i_name on S (name);
#
select * from S;
select name from S where age < 20 and age > 10;
select ID from S where gender = 'F';
#
delete from S where age < 18;
delete from S where ID = 13;
#
select * from S;
select name from S where age < 20 and age > 10;
select ID from S where gender = 'F';
#
drop table S;
```

测试结果



如图所示，所有的记录都可以在千分之秒内完成
最后的完成用时不超过 1 秒钟。
对于学生的信息查找是正确的。
详见我们的视频。