**数据库系统project报告**

2021-2022学年第2学期（CST31106）

|  |  |
| --- | --- |
| 数据库系统project任务书 | |
| 名称 | 数据库存储管理设计与模拟实现 |
| 类型 | □验证性 □设计性 综合性 |
| 内容 | 关系型数据库采用自描述的方式进行数据库系统管理，关于关系的相关数据也存储在数据库管理系统里。  针对关系型数据库存储模块，进行需求分析，并设计一个简单的数据库系统存储模块，根据设计模拟实现数据存储管理器，主要实现：数据库创建、表格创建、数据添加、删除、更新等操作过程中，数据库系统所进行的存储方面的管理；设计索引，并比较有索引和无索引的区别。  模拟实现采用：python或者Java实现具体功能，数据存储采用excel或者文本文件，可用文本文件模拟硬盘数据块，不需要实现SQL语句的执行编译，SQL语句采用函数实现。 |
| 要求 | （1）设计方案要合理；  （2）能基于该存储管理模块实现数据的存储和读取；  （3）设计方案有一定的效率分析。 |
| 任务时间 | 2022年4月28日至2022年5月20日 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 小组成员 | | | | | | |
| 20191675 | | 20191584 | | 20192242 | | 20194305 |
| 高明 | | 谭星 | | 魏永森 | | 周圣杰 |
| 项目评分表 | | | | | | |
| 序号 | 评分项 | | 分值 | | 得分 | |
| 1 | 需求分析 | | 3分 | |  | |
| 2 | 综合设计与实现 | | 4分 | |  | |
| 3 | 团队协作 | | 3分 | |  | |
| 项目总得分： | | | | | | |

**课程项目评分标准（总分10分）**

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 评分项目 | 完成情况 | 得分 |
| 1 | 需求分析 | 分析合理 | 3分 |
| 分析较合理 | 2分 |
| 分析不合理 | 1分 |
| 分析完全错误 | 0分 |
| 2 | 综合设计与实现 | 设计完整，设计合理，工具使用熟练 | 4分 |
| 设计较完整，设计合理，工具使用较熟练 | 3分 |
| 设计较完整，设计较合理，工具使用较熟练 | 2分 |
| 设计较完整，设计不合理，工具使用不熟练 | 1分 |
| 抄袭、被抄袭 | 0分 |
| 3 | 团队协作 | 有团队，分工合理，密切协作 | 3分 |
| 有团队，分工合理，有一定协作 | 2分 |
| 有团队，分工不合理，无协作 | 1分 |
| 无团队，无协作 | 0分 |

# 一、 小组分工

|  |  |
| --- | --- |
| 需求分析 | 高明，谭星，魏永森，周圣杰 |
| 项目结构 | 谭星，高明 |
| 语法解析实现 | 周圣杰 |
| SQL操作实现 | 高明，谭星，魏永森， 周圣杰 |
| 无所引数据库实现 | 高明，周圣杰 |
| 有无索引对比，分析 | 谭星，魏永森 |
| 报告 | 高明，谭星，魏永森，周圣杰 |

代码地址：<https://github.com/neurotics77/database_pro2>

# 二、 需求分析

设计一个数据库系统存储模块，需要能够模拟实现数据库管理系统的操作。

## 基本操作

* 数据库的创建
* 表格的创建
* 数据表的添加
* 数据表的删除
* 数据表的更新

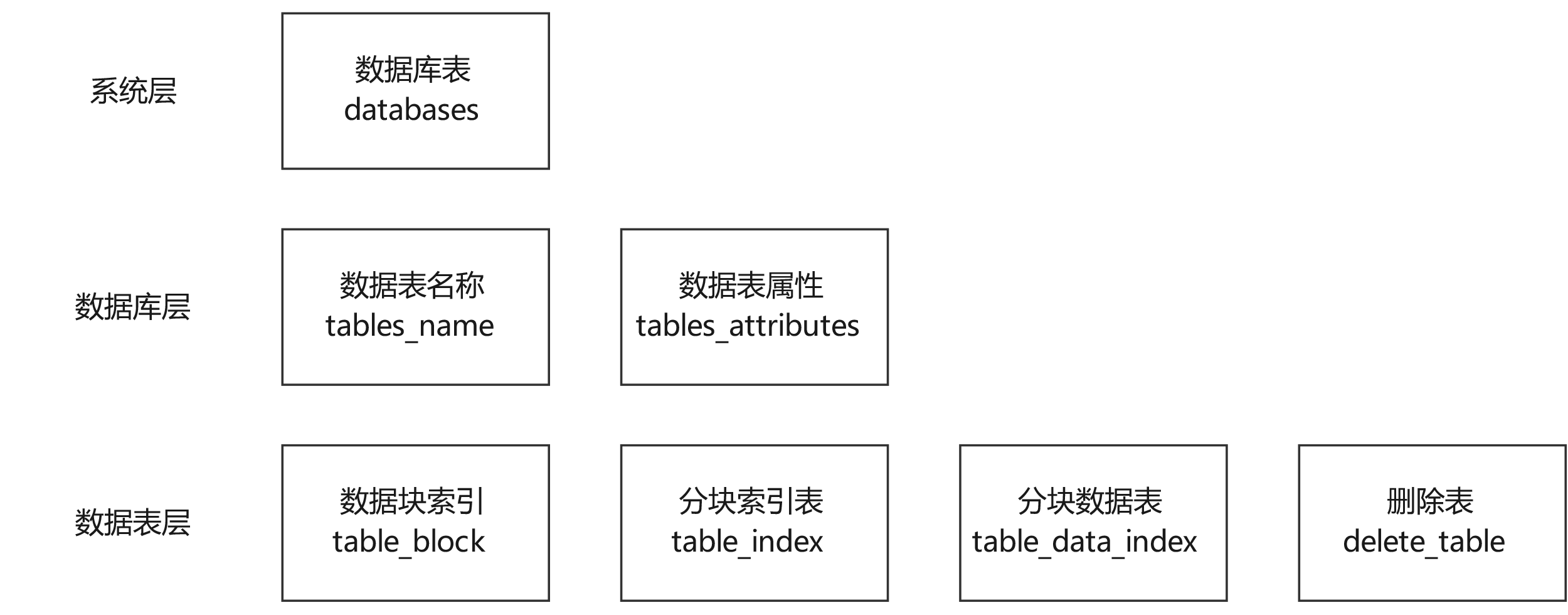
## 存储管理

* 需要能够存储数据库信息
* 存储数据库中表的信息
* 存取表中数据的信息
* 分块进行数据存储
* 块定位信息的存储
* 数据库，表，数据，块的索引信息

比较有索引与无所引的版本，查看性能差异，分析原因。

# 三、 综合设计与实现

## 1. 整体架构



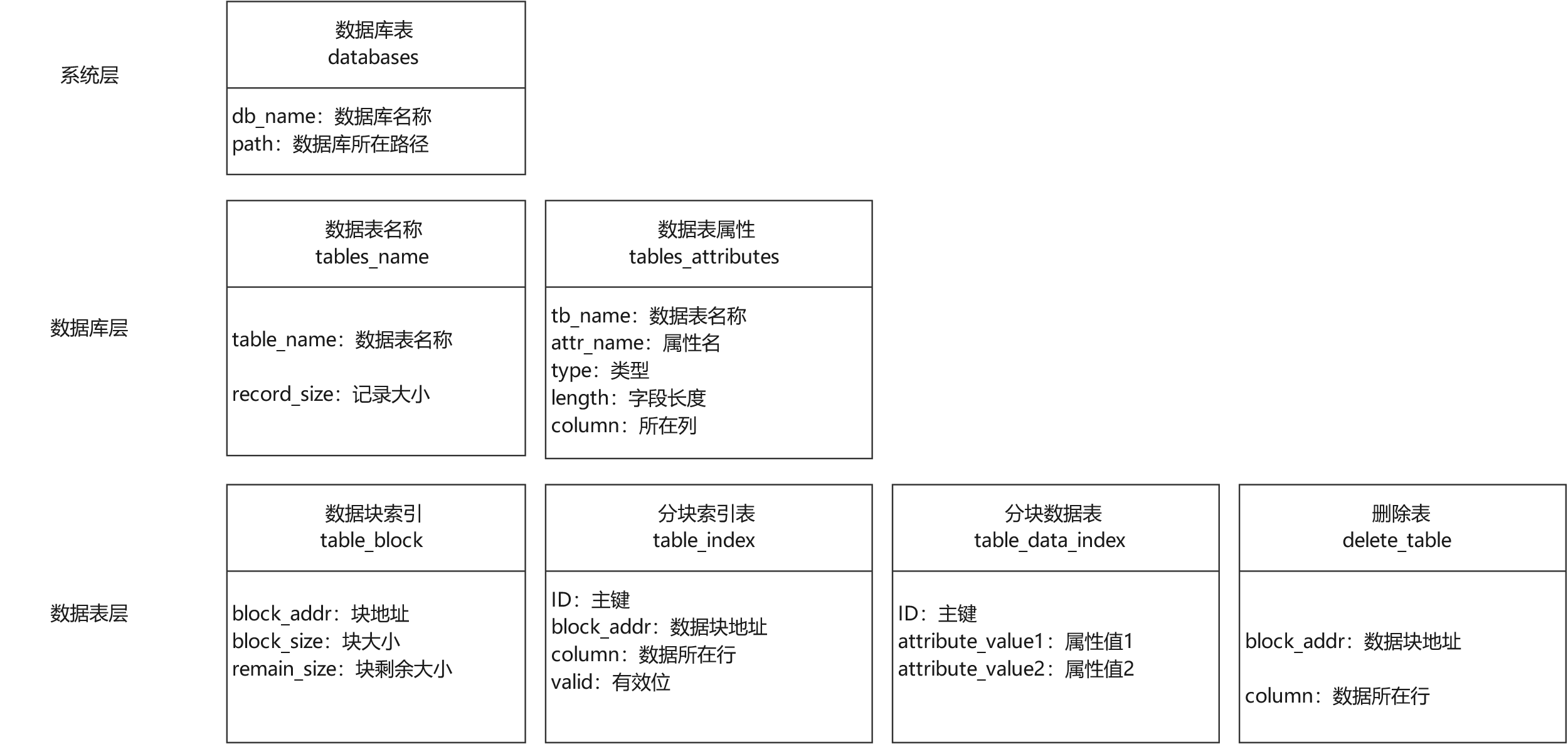
RDBMS结构

RDBMS中首先应该含有一个数据库表databases用来记录目前拥有多少数据库以及各个数据库的名字。

每个数据库中又需要有一个记录该数据库含有的所有数据表信息的table\_names, 其中会记录数据表的名称以及该数据表数据单元(一条数据)占用块存储空间的大小，可以根据数据表的名称来索引该表数据单元大小。除此之外，还设置有一个记录每个数据表中属性对应的名称，类型与所在列数的tables\_attributes表，可以根据数据表的名称来索引属性的名称，类型与所处列数。

对于单个数据表使用‘块’来存储，我们在实现过程中，使用csv文件来代表块，也就是分块数据表table\_data\_index，其中存有数据表真正的数据单元。并且每个数据表都使用了分块索引表table\_index，这样就可以根据数据表主键的值来索引该条数据具体存在于哪一个块，以及块中的哪一行，其中还设有一位有效位来标记该位置的数据是否被删除。删除表delete\_table用来维护块中数据的添加与删除操作，其中记录了块中被删除数据的具体位置，如果进行插入操作，那么会优先选取delete\_table中存储的位置进行数据的插入。在单个数据库所创建的表格中，每个表格又需要对应一个数据块索引表table\_block,其中包含有块的地址，块的容量，以及块中目前还剩余的容量，可以根据块地址来索引当前块的容量和目前剩余容量。

上述提到的各表只有table\_data\_index，table\_index使用了csv文件实现，其他提到的表格均采用python中字典数据类型来实现。上文提到的所有表格数据字典设计如下：



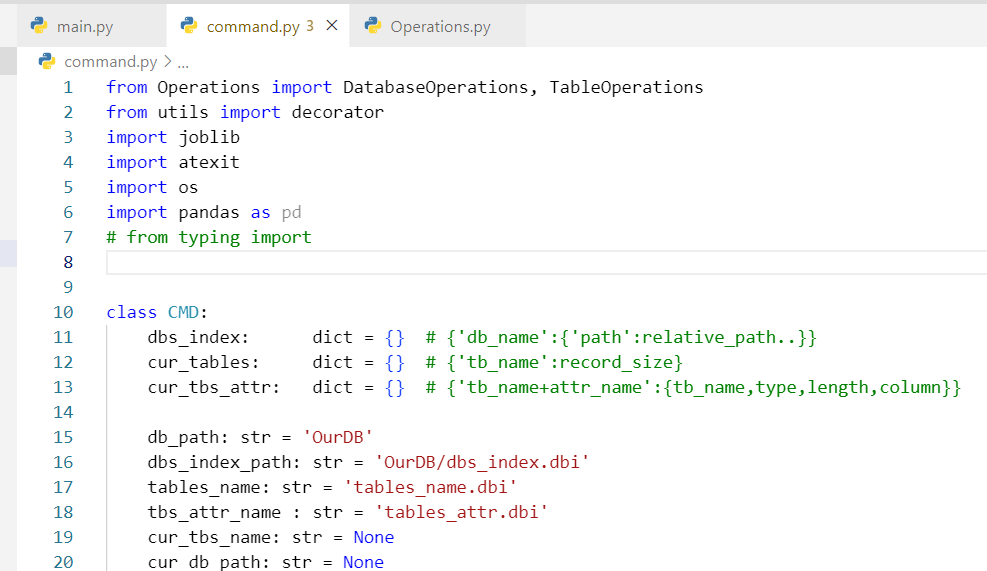
数据字典

系统整体代码实现：

command中主要实现控制台命令的读取与sql语法解析。

Operations中实现数据库的逻辑操作，创建以及增删查改。

上述两个文件均使用面向对象变成，使用类与静态函数的方法来在main函数中调用。



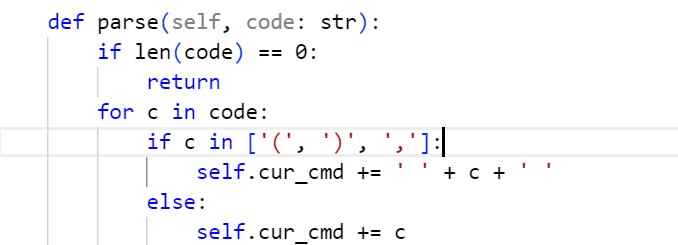
## 2. SQL语句解析

使用python语言对用户在终端输入的字符串进行解析，如果语法正确那么会进行相应的sql语法解析，然后进行对应的操作；如果语法错误则会报错。

目前设计支持的语法有：

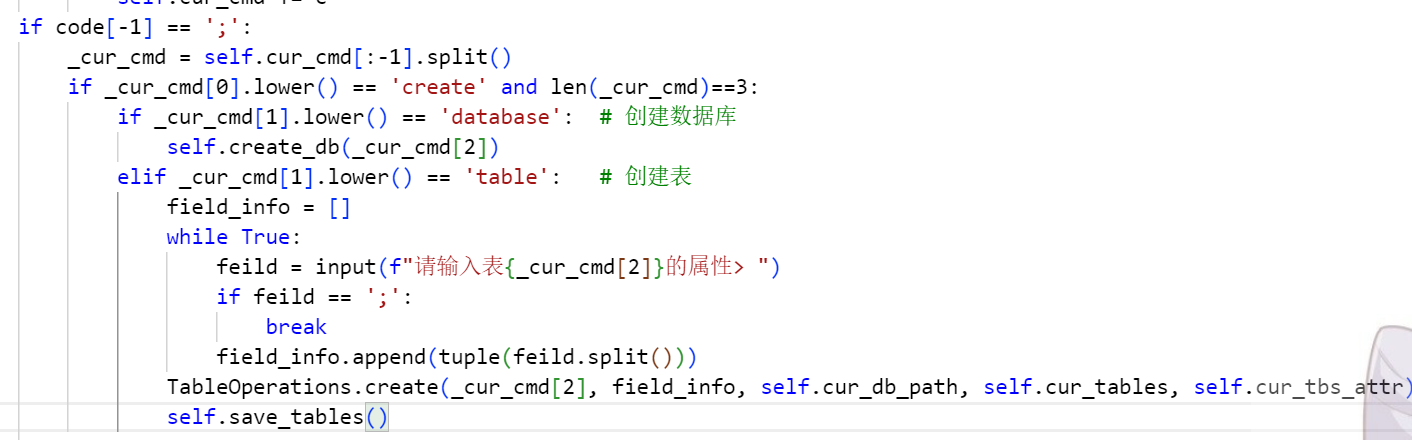
* create database 数据库名； 创建数据库
* show(); 展示所有的数据库
* use 数据库名； 进入对应的数据库
* create table 表名； 创建表格，进而会提示输入表格对应的字段
* insert 表名 values (主码，属性值2，属性值3)； 向表格中插入元组
* delete from 表名 where (主码== value); 删除表格中主码为value的那一行数据
* update 表名 set (属性=value1) where (主码=value2)； 将表格中主码为value2的那一行数据的对应属性值改为value1。
* select \* from 表名 where (主码=value); 将表格中主码为value的数据元组展示出来。
* exit； 退出表格/数据库

首先parse()函数实现命令行的读取



这里将输入的字符串中的‘（’，‘）’，‘，’，都转换为空格。后续方便split变为数组操作。

如果输入的字符串的最后一个字符为‘；’，那么就是识别到了命令的结束，之后对命令开始分析。先将上面一步处理过的字符串以空格分割为字符串数组。然后判断字符串数组的长度和字符串数组的第一个字符串是不是符合要求的命令。

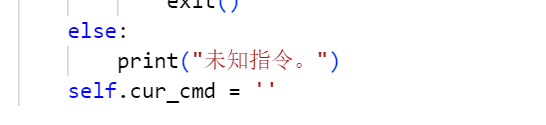


如果识别到了create关键字，就进行第二个字符串的检验，如果是database，那么就调用创建数据库的命令；如果是table，就调用创建表格的命令。创建表格的过程使用了控制台提示输入的方法。



剩下的命令如上图片一样进行处理。包括delete，select，insert，show，update，user，exit命令。

如果不合上述所有状况，那么就会在终端提示“未知指令。”



## 3. 语句实现

### 3.1 创建数据库

#### 实现原理

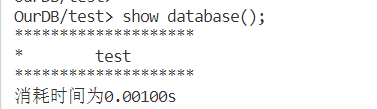
执行该操作会创建新的数据库，在系统下新建数据库同名文件夹，然后在讲数据库的信息添加到数据库表databases中。

### 3.2 展示数据库

#### 实现原理

show操作，访问数据库表databases，展示所有的数据库名称。

#### 实现效果



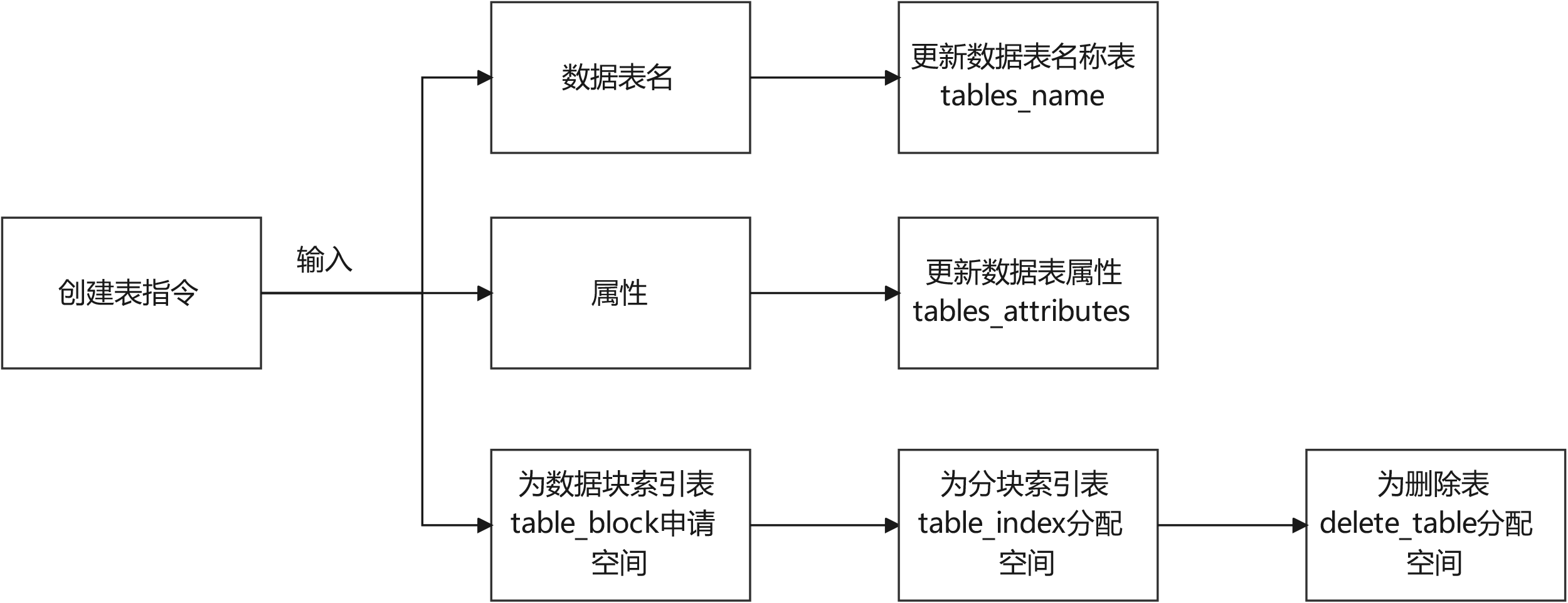
### 3.3 进入数据库

#### 实现原理

识别到use关键字，进入数据库建立表格。

### 3.4 创建数据表

#### 实现原理



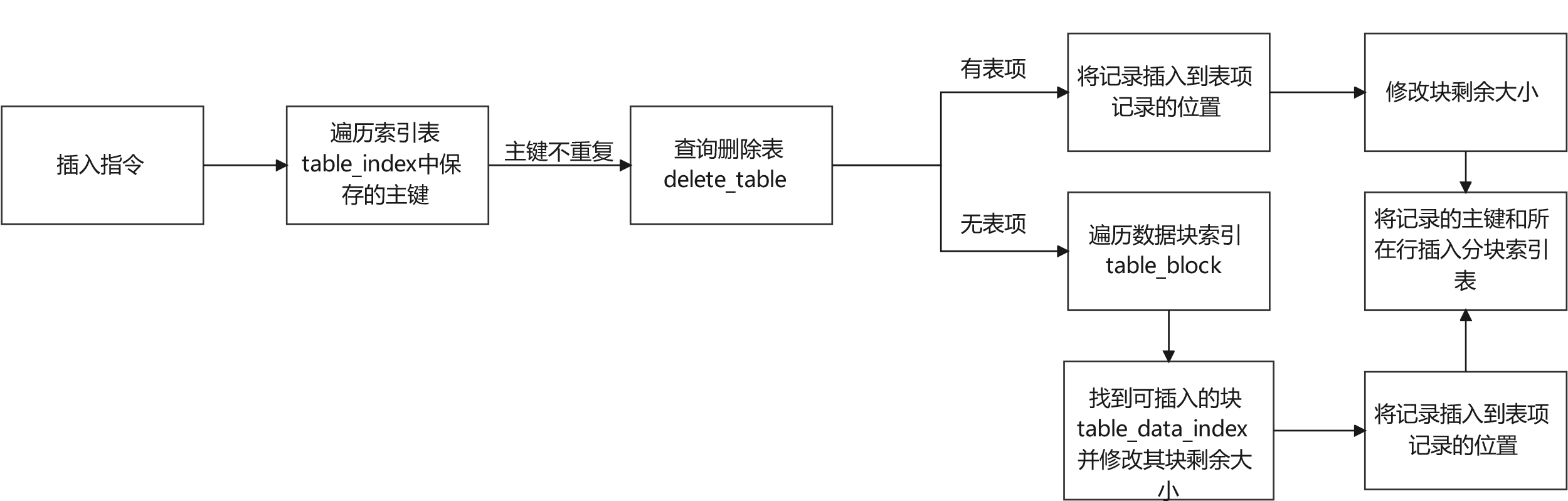
执行创建数据表指令时，需要读入用户输入的数据表名称，以此来更新数据表名称表table\_names，需要插入对应数据表的名称和一个记录(数据单元)占有块存储空间的大小。然后按照用户依次输入的属性名和属性类型来更新数据表属性表格tables\_attributes表，为数据表的属性字段建立索引。然后上文结构中提到每个表格都有对应的table\_block，table\_index，delete\_table，table\_data\_index表格，其中前三者都是在创建表格时初始化，此刻需要为他们分配存储空间。最后的table\_data\_index在插入操作进行初始化。

#### 实现效果



### 3.5 插入语句

#### 实现原理



当执行插入记录(数据单元)操作时，首先访问该数据表对应的分块索引表table\_index，查询主键是否有重复值。如果有重复，那么就会返回查询失败，然后终端命令；如果主键无重复，那么接下来继续访问删除表delete\_table，查询当前表已经被删除掉的数据的位置，优先插入到该位置中。如果delete\_table中有表项，那么就将记录插入到该表项对应的数据块table\_data\_index，然后修改数据块大小并返回table\_block，再将插入的新记录的主键以及在块中所处的位置添加到分块索引表table\_index中方便之后的索引；如果当前delete\_table中无值，那么就遍历table\_block寻找目前还有剩余容量的块，然后寻找到对应存储数据单元的块table\_data\_index，将记录插入到当前块，然后在table\_block中更新块剩余大小，table\_index中添加主键 。

#### 实现代码

def insert(table\_name: str, field\_info: List,

            cur\_db\_path: str, cur\_tbs\_index: dict, cur\_tbs\_attr: dict) -> None:

        # 判断主键是否重复 默认第一个属性为主键

        if None not in TableOperations.traversal\_key(f'{cur\_db\_path}/{table\_name}', int(field\_info[0])):

            print("ERROR主键值重复")

        # 查找表的记录大小：

        # tables\_name = TableOperations.load\_tables\_name(cur\_db\_path)

        # print("tbs\_name ", tables\_name)

        record\_size = cur\_tbs\_index[table\_name]

        # 找出可以用的块：

        table\_blocks = TableOperations.load\_table\_blocks(f'{cur\_db\_path}/{table\_name}')

        avail\_block\_addr = ""

        isFind = False

        block\_rest\_size = 0

        avail\_block\_addr = f'{table\_name}\_data\_0.csv' # 默认值

        # print('table\_blocks', table\_blocks)

        for addr, value in table\_blocks.items():

            avail\_block\_addr = addr

            if value[1] > record\_size:

                # 更新块剩余大小

                table\_blocks[addr][1] -= record\_size

                block\_rest\_size = table\_blocks[addr][1]

                isFind = True

                break

        # 申请新的块

        if not isFind:

            cur\_block\_num = int(avail\_block\_addr.split('\_')[-1].split('.')[0])+1

            pd.DataFrame(columns = list(range(len(field\_info)))).to\_csv(f'{cur\_db\_path}/{table\_name}/data/{table\_name}\_data\_{cur\_block\_num}.csv', index=False)

            avail\_block\_addr = f'{table\_name}\_data\_{cur\_block\_num}.csv'

            table\_blocks[avail\_block\_addr] = [4000, 4000-record\_size]

            block\_test\_size = 4000-record\_size

        # 保存table\_blocks

        TableOperations.save\_table\_blocks(f'{cur\_db\_path}/{table\_name}', table\_blocks)

        # 向 data/可用csv路径中插入值

        data = pd.read\_csv(f'{cur\_db\_path}/{table\_name}/data/{avail\_block\_addr}')

        data.loc[len(data)] = field\_info

        data.to\_csv(f'{cur\_db\_path}/{table\_name}/data/{avail\_block\_addr}', index=False)

        # 为记录建立索引

        # 寻找可用的索引块

        isFind = False

        table\_indexs = os.listdir(f'{cur\_db\_path}/{table\_name}/table\_index')

        avail\_index\_addr = ""

        cur\_index\_count = 0

        for table\_index in table\_indexs:

            block = pd.read\_csv(f'{cur\_db\_path}/{table\_name}/table\_index/{table\_index}')

            index\_rest\_size = block.iloc[0].values[0]

            if index\_rest\_size >= 8:

                avail\_index\_addr = table\_index

                isFind = True

                break

            cur\_index\_count+=1

        # 没有找到可用索引块的话,新建索引块文件

        if not isFind:

            cur\_index\_count += 1

            df = pd.DataFrame([[4000, None, None, None]], columns=list(range(4)))

            df.to\_csv(f'{cur\_db\_path}/{table\_name}/table\_index/{table\_name}\_{cur\_index\_count}.csv', index=False)

            avail\_index\_addr = f'{table\_name}\_{cur\_index\_count}.csv'

        # 向可用的索引块中插入索引

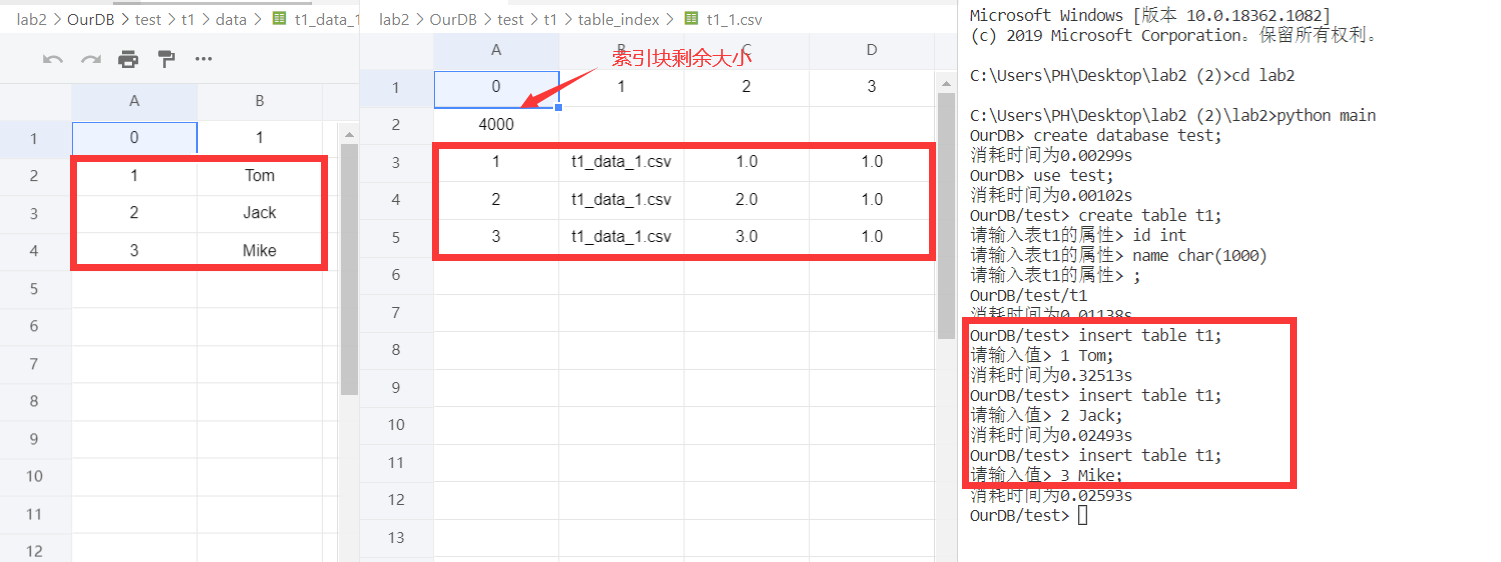
        index\_block = pd.read\_csv(f'{cur\_db\_path}/{table\_name}/table\_index/{avail\_index\_addr}')

        index\_block.iloc[0].values[0] -= 8

        index\_block.loc[len(index\_block)] = [field\_info[0], avail\_block\_addr, data.shape[0], 1]

        index\_block.to\_csv(f'{cur\_db\_path}/{table\_name}/table\_index/{avail\_index\_addr}', index=False)

#### 实现效果



### 3.6 删除语句

#### 实现原理



当执行删除语句操作时，需要根据输入的主键值删去数据表中对应的记录。首先要遍历分块索引表table\_index，找到对应主键标识的有效位更改为无效，再将当前记录所在的块table\_data\_index以及具体存储在块中的位置添加到删除表delete\_table中。最后更新块table\_data\_index对应的块剩余容量到table\_block中。

#### 实现代码：

def delete(table\_name: str,cur\_db\_path: str, field\_info: List, cur\_tables: dict) -> None:

        cur\_tb\_path = f'{cur\_db\_path}/{table\_name}'

        table\_indexs = os.listdir(f'{cur\_tb\_path}/table\_index')

        for table\_index in table\_indexs:

            block = pd.read\_csv(f'{cur\_tb\_path}/table\_index/{table\_index}')

            row = 1

            while(row < block.shape[0]):

                # 键相等且信息有效

                if block.iloc[row,0] == field\_info[0] and block.iloc[row,3] == 1:

                    # 有效位置0

                    block.iloc[row,3] = 0

                    # 保存

                    block.to\_csv(f'{cur\_tb\_path}/table\_index/{table\_index}', index=False)

                    # 加载table\_block

                    table\_blocks = TableOperations.load\_table\_blocks(cur\_tb\_path)

                    # 获取表名以索引table\_blocks

                    table\_name = cur\_tb\_path.split('/')[-1]

                    # 更改table\_blocks中的块剩余大小

                    table\_blocks[str(block.iloc[row,1])][1] += cur\_tables[table\_name]

                    TableOperations.save\_table\_blocks(cur\_tb\_path, table\_blocks)

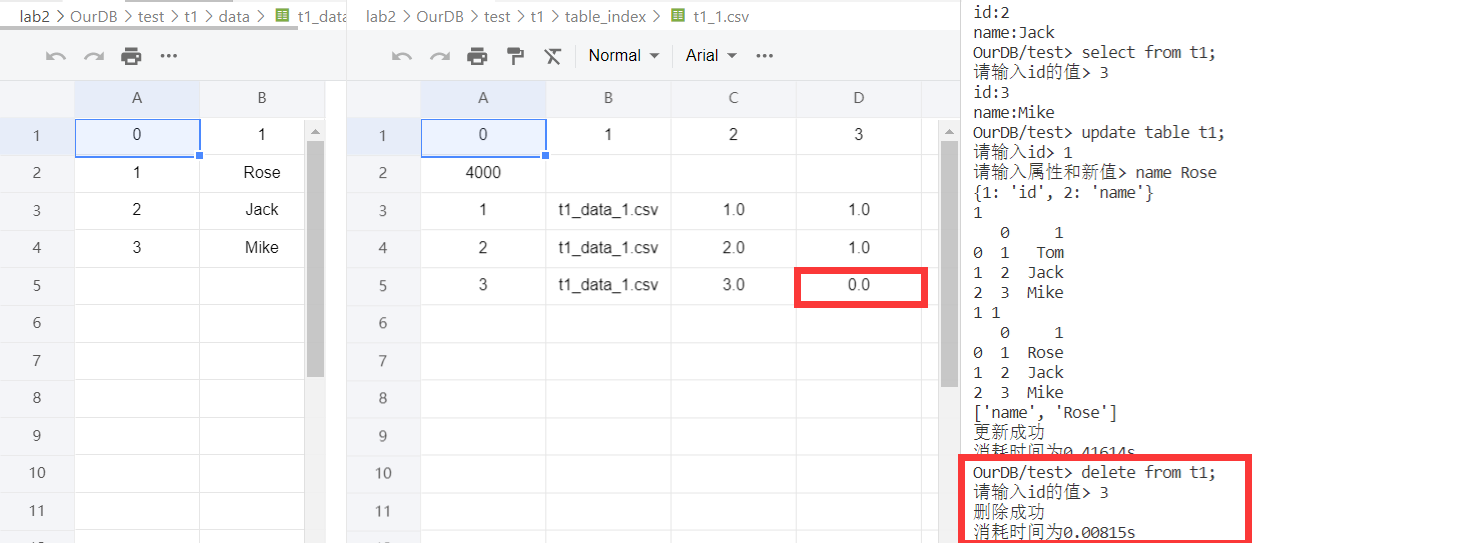
                    print("删除成功")

                    return

                row += 1

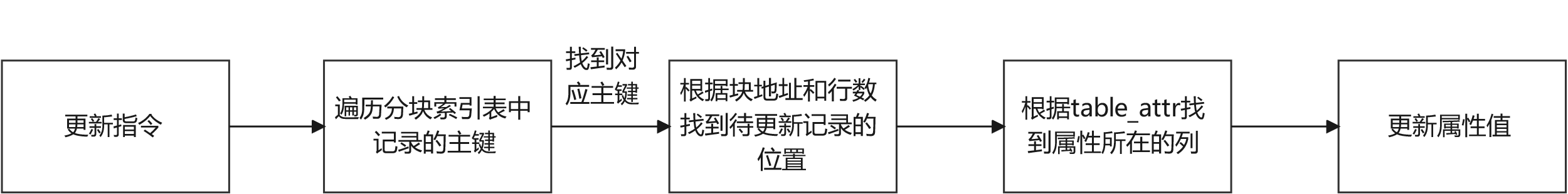
        print("删除失败")

#### 实现效果：



### 3.7 更新语句

#### 实现原理



执行更新指令，需要根据输入的主键值以及想要更改的属性名称和属性值来进行数据块中记录的更改。首先遍历分块索引表table\_index来查找该主键对应记录存储的数据块table\_data\_index的位置以及该记录在块中的具体位置。然后还需要访问表格的属性表table\_attributes来寻找我们想要更改的属性在数据表中存储的列数column，然后访问该记录的第column个属性进行属性值的更新。

#### 实现代码：

def update(table\_name: str, field\_info: List, key: int, cur\_db\_path: str, cur\_tbs\_attr: dict) -> None:

        attr = {value[3]: key.split("\_")[-1] for key, value in cur\_tbs\_attr.items() if value[0] == table\_name}

        print(attr)

        block\_addr, row = TableOperations.traversal\_key(f'{cur\_db\_path}/{table\_name}', key)

        print(key)

        if block\_addr is None or row is None:

            print("更新出错")

        else:

            # 数据所在的块

            select\_data\_block = pd.read\_csv(f'{cur\_db\_path}/{table\_name}/data/{block\_addr}')

            # field\_info[0]为属性，field\_info[1]为新值

            idx = [k for (k,v) in attr.items() if v == "id"]

            print(select\_data\_block)

            print(int(row),idx[0])

            select\_data\_block.iloc[int(row)-1,idx[0]] = field\_info[1]

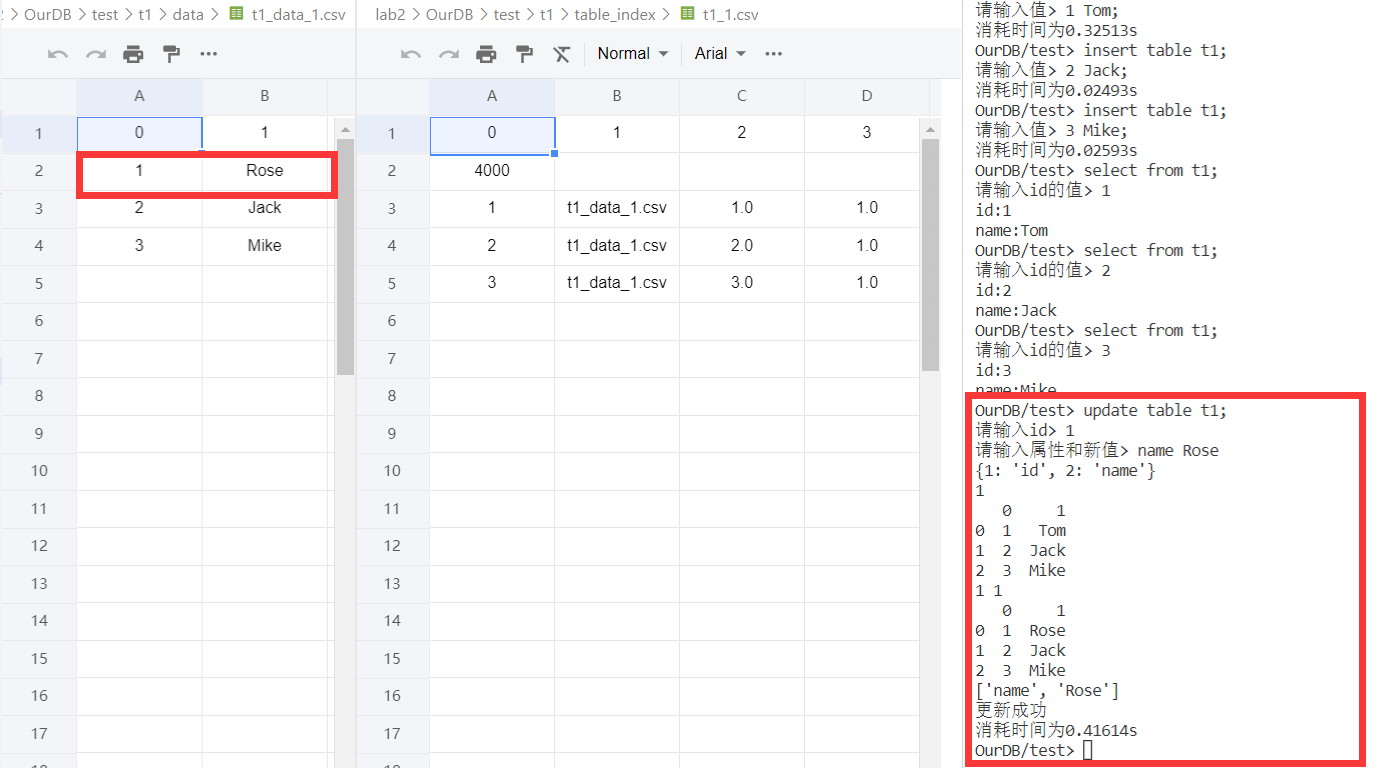
            print(select\_data\_block)

            print(field\_info)

            select\_data\_block.to\_csv(f'{cur\_db\_path}/{table\_name}/data/{block\_addr}', index=False)

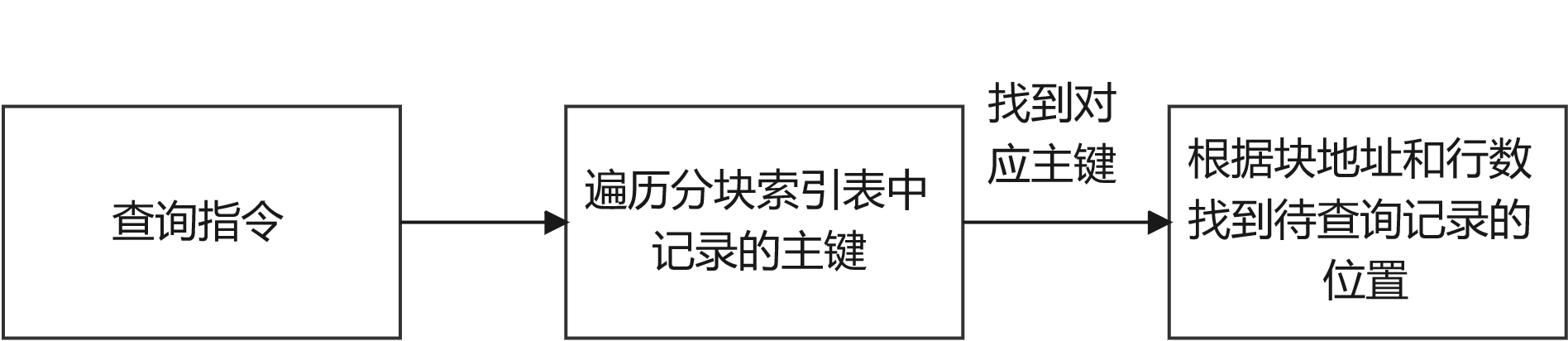
            print("更新成功")

#### 实现效果



### 3.8 查询语句

#### 实现原理



执行查询操作时，需要根据输入的主键值查询对应的记录的内容。直接对分块索引表table\_index进行遍历，找到对应主键存储的数据块table\_data\_index的位置以及记录在块中的位置。

#### 实现代码：

def retrieve(table\_name: str, field\_info: List,cur\_db\_path: str, cur\_tbs\_attr: dict) -> None:

        # attr = {}

        # for key,value in cur\_tbs\_attr.items():

        #     if value[0] == table\_name:

        #         attr[value[3]] = key.split("\_")[-1]

        attr = {value[3]: key.split("\_")[-1] for key, value in cur\_tbs\_attr.items() if value[0] == table\_name}

        block\_addr, row = TableOperations.traversal\_key(f'{cur\_db\_path}/{table\_name}', field\_info[0])

        if block\_addr is None or row is None:

            print("查询出错")

        else:

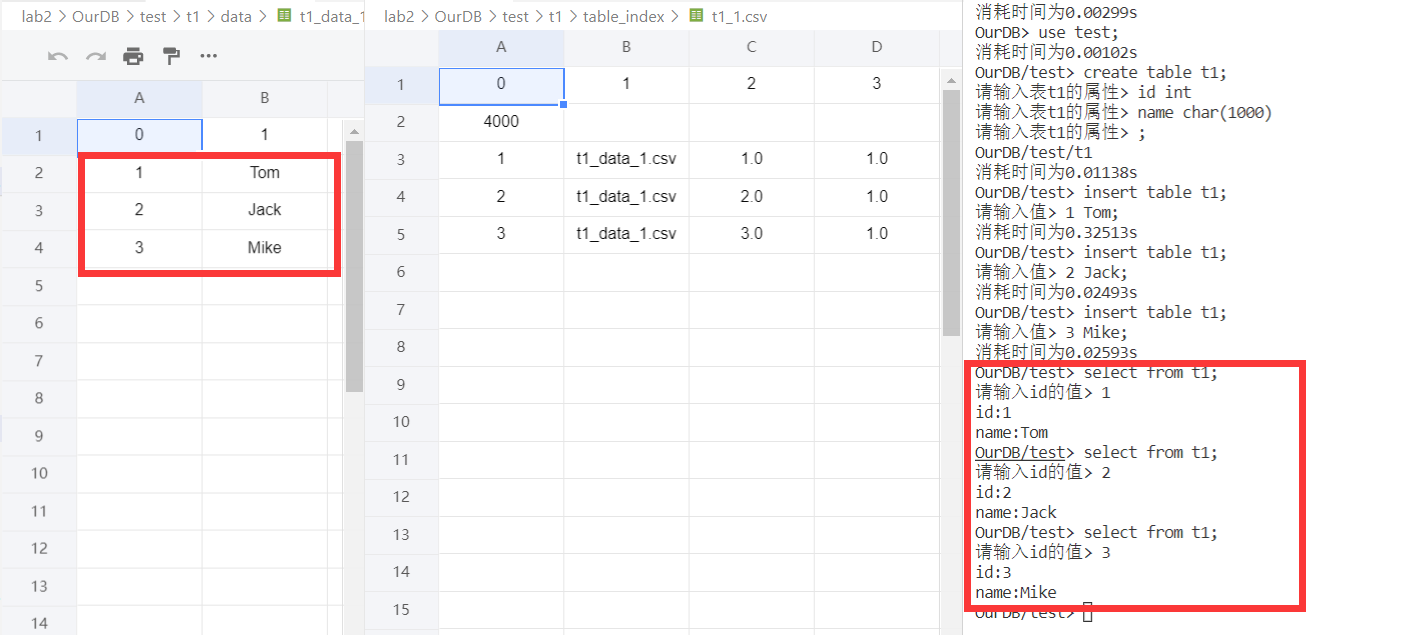
            # 数据所在的块

            select\_data\_block = pd.read\_csv(f'{cur\_db\_path}/{table\_name}/data/{block\_addr}')

            for i in range(select\_data\_block.shape[1]):

                print(attr[int(i+1)] + ":" + str(select\_data\_block.iloc[int(row)-1,i]))

#### 实现过程：



### 3.9 退出语句

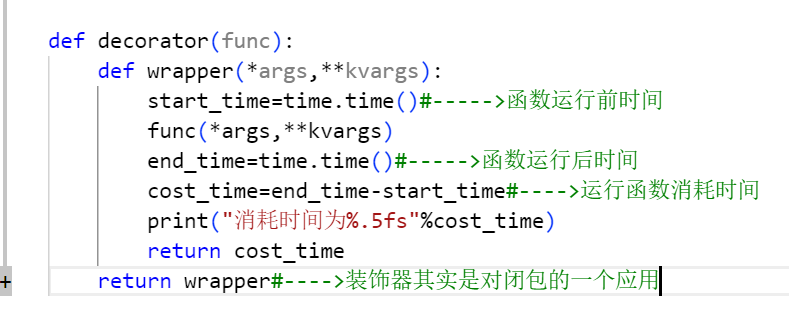
#### 实现原理

识别到exit返回上一级，直至退出系统。

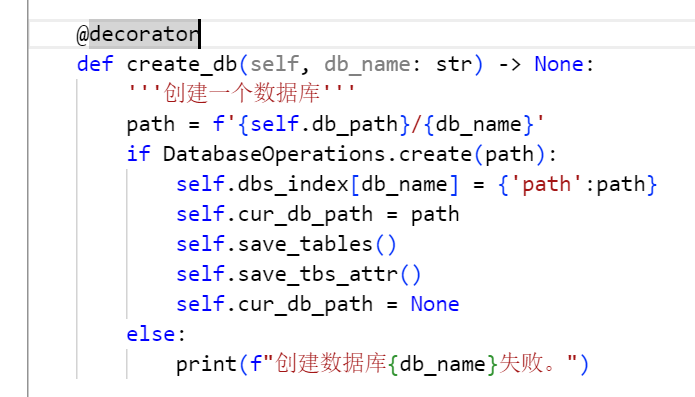
## 4. 性能测试

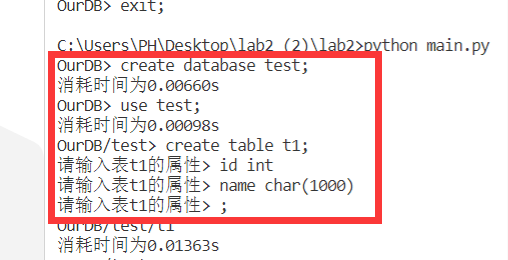
性能测试使用了python中的内置的装饰器decorator进行了处理。

Python的装饰器本质上是一个嵌套函数，它接受被装饰的函数(func)作为参数，并返回一个包装过的函数。这样我们可以在不改变被装饰函数的代码的情况下给被装饰函数或程序添加新的功能。



只需要在我们需要的函数顶以前前加 @decorator就可以进行对该函数的计时分析，之后调用该函数会自动输出函数运行消耗时间。效果如下图所示

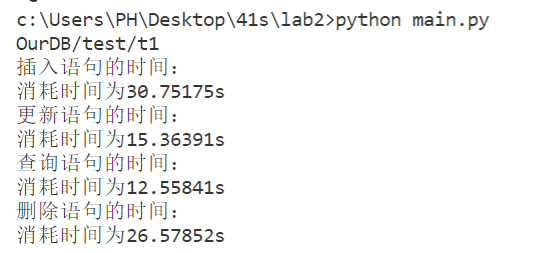




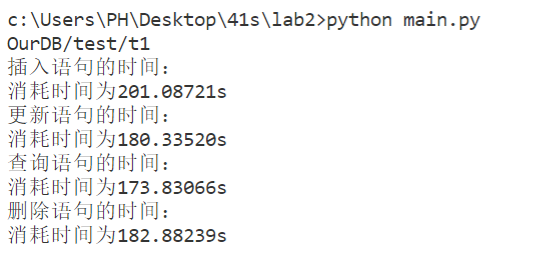
块大小设置为4KB，创建数据表test设置属性为id和name，类型分别为int和char(1000), 设置较大的记录占用空间大小是为了能够产生更多的块来存储相同数量的记录，以此来体现使用索引的优势。

插入操作使用1000条insert table id,name语句进行测试；在插入1000条数据之后，更新、查询、删除操作均使用随机产生的1000条sql语句进行测试。

有索引数据库：



无索引数据库：



插入1000条数据后，共产生了334个csv文件，即334个数据块，而索引块共产生了2个。因此在更新、查询、删除测试时，有索引情况下最多仅需遍历两个文件，而无索引情况下最多需遍历334个文件，在磁盘读取和存储上花费了大量的时间。

