

数据库系统 project 报告

2022-2023 学年第 2 学期 (CST31106)

数据库系统 project 任务书	
名称	数据库 SQL 引擎设计与模拟实现
类型	<input type="checkbox"/> 验证性 <input type="checkbox"/> 设计性 <input checked="" type="checkbox"/> 综合性
内容	<p>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、SQL 执行，进行需求分析，并设计一个简单的数据库系统 SQL 引擎模块，根据设计模拟实现 SQL 基本操作、数据存储等，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 SQL 引擎所进行的各方面的管理和操作（包含硬盘数据块）；设计索引，并比较有索引和无索引的区别。</p> <p>模拟实现采用：python 或者 Java 实现具体功能，设计模型所需求存储的数据可采用数据库管理系统、excel 或者文本文件，可用文件模拟硬盘数据块，SQL 语句采用函数实现。</p>
要求	<p>(1) 设计方案要合理；</p> <p>(2) 能基于该 SQL 引擎模块完成 SQL 的模拟实现；</p> <p>(3) 设计方案有一定的效率分析。</p>
任务时间	2023 年 4 月 17 日至 2023 年 5 月 20 日

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、项目背景

随着信息技术的广泛应用，对于有效管理和存取大量数据资源的需求变得尤为重要。数据库技术的不断发展和进步对各行各业的发展起到了巨大的推动作用，解决了数据冗余不一致、查询困难、关联度低以及无法并发操作等问题。目前，关系型数据库和非关系型数据库是常见的数据库类型。本项目参考了关系型数据库的设计思想，实现一个数据库存储系统。在该系统中，数据采用关系模型进行存储，被组织在一组拥有正式描述性的表格中。这些表格充当特殊的数据集合，可以通过不同的方式进行存取或重新组合，而无需对数据库表格进行重新组织。

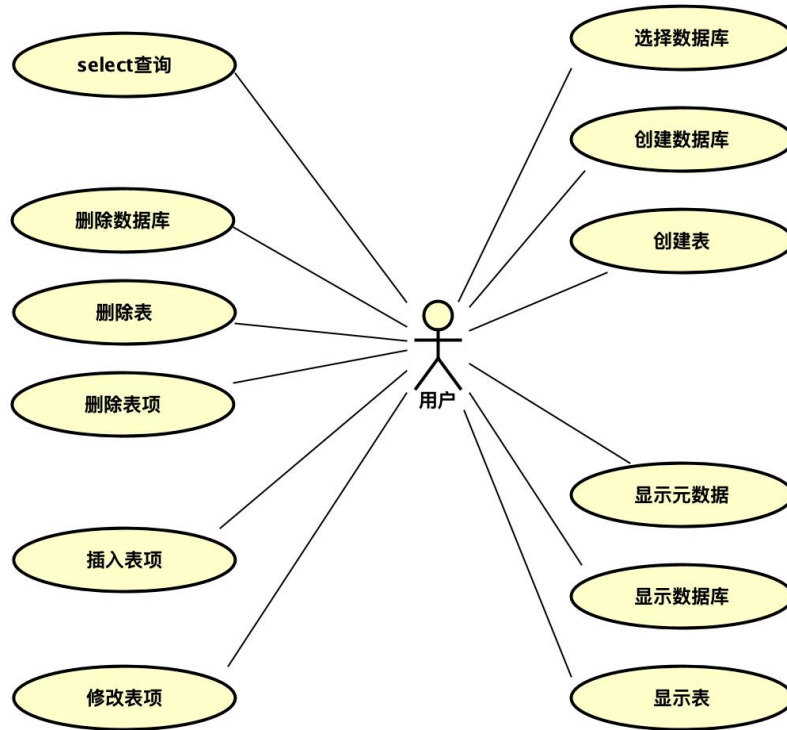
通过这种设计，我们能够更高效地管理和存取数据，提供更灵活的数据检索和操作方式。关系型数据库的设计和实现使得数据存储和访问更加便捷，为各行业的业务需求提供了支持。本项目的目标是设计和实现一个可靠的数据库存储系统，通过采用关系模型和表格的形式，提供高效的数据管理和检索能力。通过这个项目，我们可以深入了解关系型数据库的特点和优势，并将其应用于实际情境中，为各行业的数据管理问题提供解决方案。

二、设计思路

本次项目使用 `python` 实现具体功能，设计模型所需求存储的数据采用 `excel`。小组经过讨论，先给出了需求分析，然后根据需求设计数据库的功能，具体实现时使用 `python` 的 `pandas` 第三方库对 `csv` 文件进行增删改查以模拟数据存储管理器操作。查询语句语法规则与 `MySQL` 相同，见 `README.md`。

三、需求分析

项目需求主要就是明确数据存储管理器的**功能需求**，便于后续的综合设计。小组成员经过充分讨论，绘制了用例图来描述参与者和用例之间的关系：



本数据存储管理器中的主体就是当前使用本数据存储管理器的用户；用例设计有 12 个，每个用例涉及的操作都需要被数据存储管理器的元数据进行组织、记录或修改。每个用例分析如下：

- **创建数据库：**用户可以通过 `create database db_name` 语句创建名为 `db_name` 的数据库；
- **选择数据表：**用户可以通过 `use db_name` 语句选择名为 `db_name` 的数据库；
- **创建数据表：**用户选择数据库后，可以使用 `create table tb_name(type value.....)` 在 `db_name` 数据库下创建表；
- **查看数据库：**用户可以通过 `show databases` 语句查看现有的数据库；
- **查看数据表：**用户可以通过 `show tables` 语句查看某个数据库下面现有的数据表；
- **查看元数据：**用户可以通过 `show meta data` 语句查看数据存储管理器中的元数据；
- **查询记录：**用户可以通过 `select` 语句查询数据表中指定条件的记录，包括 `select * from table_name` 或 `select column_name, ... from table_name`，以及有条件查询 `select column_name, ... from table_name where condition`；
- **删除数据库：**用户可以通过 `drop database db_name` 删除名为 `db_name` 的数据库；
- **删除数据表：**用户可以通过 `drop table tb_name` 删除名为 `tb_name` 的数据表；
- **删除记录：**用户可以通过 `delete from table_name where condition` 语句在数据表中删除指定条件的记录；

- **添加记录：**用户可以通过 `insert into table_name values(value1, value2, ...)` 语句向选定数据库的表中添加记录；
- **修改记录：**用户可以通过 `update table_name set column_name=value where condition` 语句更新数据表中指定条件的记录；

四、数据存储设计

在设计数据存储前需要考虑数据库的元数据，即定义数据库各类对象结构的数据，主要有数据库和表。因此，我们设计了两个 csv 文件作为元数据表来存储本数据存储管理器的元数据：database_meta_data.csv 存储数据库元数据，table_meta_data.csv 存储数据表元数据。通过将 db_id 设置为 table_meta_data 表的外键来将两张元数据表关联起来，因此可以有多个数据库实例，并且其具体条目都存储在 table_meta_data 表中。table_meta_data 表和 database_meta_data 表的具体字段说明如下：

1. database_meta_data 表

字段名称	字段类型	约束	说明
db_id	INTEGER	PRIMARY KEY	数据库id
db_name	VARCHAR(100)	UNIQUE NOT NULL	数据库名称
create_time	VARCHAR(100)	NOT NULL	数据库创建时间
is_del	BOOLEAN	NOT NULL	数据库是否被删除

2. table_meta_data 表

字段名称	字段类型	约束	说明
db_id	INTEGER	FOREIGN KEY	数据库id，引用 database_meta_data表的db_id 字段
table_id	VARCHAR(100)	PRIMARY KEY	数据表id
table_name	VARCHAR(100)	UNIQUE NOT NULL	数据表名称

column_list	VARCHAR	NOT NULL	数据表属性
type_list	VARCHAR	NOT NULL	数据表属性类型
primary_key	VARCHAR(100)	NOT NULL	主键
foreign_key	VARCHAR(100)		外键
row_num	INTEGER	NOT NULL	数据表记录个数
size_in_byte	INTEGER	NOT NULL	数据表大小
modify_time	VARCHAR(100)	NOT NULL	修改时间
create_time	VARCHAR(100)	NOT NULL	创建shij
uid	INTEGER	NOT NULL	用户id
gid	INTEGER		组id
is_del	BOOLEAN	NOT NULL	数据表是否被删除

五、具体代码实现

在具体实现中，我们设计了功能函数模块 DBManager.py 和主函数模块 main.py，主函数是用来调用功能函数，以命令行的方式呈现给用户：

```
import re
from DBManager import DBManager

def main():
    print('=====欢迎使用数据库系统=====')
    # 生成数据库管理器对象
    dbm = DBManager()
    # 初始化数据库表
    dbm.init()
    # 获取用户输入的指令
    instr = input(" - 请输入指令：")
    # 默认数据库名称
```

```

db_name = dbm.DEFAULT_DB_NAME
while instr != "quit":
    # 解析用户输入的指令
    if re.fullmatch(dbm.use_dbs_instr, instr):
        db_name = dbm.analyse_use_db(instr)
    elif instr == "compare":
        dbm.analyse_instr(instr, db_name, True)
    else:
        dbm.analyse_instr(instr, db_name)
    instr =
input("\n===== \n - 请输入指令: ")
    print("\n\n===== 退出成功! =====")
if __name__ == '__main__':
    main()

```

下面详细介绍功能模块，定义了 DBManager 类：

1. 构造函数

在构造函数中定义了一些属性，包含基本路径以及各种 SQL 语句规范：

```

def __init__(self) -> None:
    # 基本路径
    self.BASE_PATH = os.getcwd()
    self.DB_META_DATA_PATH = os.path.join(self.BASE_PATH,
r"META_DATA/database_meta_data.csv")
    self.TB_META_DATA_PATH = os.path.join(self.BASE_PATH,
r"META_DATA/table_meta_data.csv")
    self.DEFAULT_DB_NAME = "default"

    # 显示所有列
    pd.set_option('display.max_columns', None)
    # 显示所有行
    pd.set_option('display.max_rows', None)
    # 选择数据库
    self.use_dbs_instr = r"use\b\s[a-zA-Z0-9_]+"
    # insert 语句
    self.insert_instr="insert\\b\\sinto\\b\\s[a-zA-Z0-9_]+\\b\\svalues\\(((\\0-9*)|([a-zA-Z0-9_]*)?)?(\\,((\\0-9*)|([a-zA-Z0-9_]*)*)*)\\)"
    # update 语句
    self.update_instr="update\\b\\s[a-zA-Z0-9_]+\\b\\sset\\b\\s[a-zA-Z0-9_]+[>=<][a-zA-Z0-9_]+\\b\\swhere\\b\\s[a-zA-Z0-9_]+[>=<][a-zA-Z0-9_]+"
    # 删除数据库
    self.drop_database_instr = r"drop\b\sdatabase\b\s[a-zA-Z0-9_]+"
    # 删除表
    self.drop_table_instr = r"drop\b\sstable\b\s[a-zA-Z0-9_]+"
    # 删除表项

```

```

        self.delete_where_table_instr = r"delete\b\sfrom\b\s[a-zA-Z0-9_]+\b\swhere\b\s[a-zA-Z0-9_]+[>=<][a-zA-Z0-9_]+"
        # 查询
        self.select_table_instr = r"(select\b\s([a-zA-Z0-9_]+,)*[a-zA-Z0-9_]+\b\sfrom\b\s[a-zA-Z0-9_]+)"
        self.select_where_table_instr = r"select\b\s([a-zA-Z0-9_]+,)*[a-zA-Z0-9_]+\b\sfrom\b\s[a-zA-Z0-9_]+\b\swhere\b\s[a-zA-Z0-9_]+[>=<][0-9_]+"
        self.type_list = ["varchar", "int", "null"]

```

2. 初始化函数

除了构造函数中的属性需要初始化，元数据表和元数据库表也需要初始化：

```

# 初始化元数据库表
def init_databases_table(self):
    try:
        db_meta_data_pd = pd.read_csv(self.DB_META_DATA_PATH)

    except FileNotFoundError:
        db_meta_data_columns = ["db_id", "db_name", "create_time", "is_del"]
        db_meta_data_pd = pd.DataFrame(columns=db_meta_data_columns,
index=[0])
        db_meta_data_pd.to_csv(self.DB_META_DATA_PATH, index=False, sep=",")

# 初始化元数据表
def init_meta_data(self):
    try:
        tb_meta_data_pd = pd.read_csv(self.TB_META_DATA_PATH)
    except FileNotFoundError:
        tb_meta_data_columns = ["db_id", "table_id", "table_name",
"column_list",
                                "type_list", "primary_key", "foreign_key",
                                "row_num", "size_in_byte", "modify_time",
                                "create_time", "uid", "gid", "is_del"]
        tb_meta_data_pd = pd.DataFrame(columns=tb_meta_data_columns,
index=[0])
        tb_meta_data_pd.to_csv(self.TB_META_DATA_PATH, index=False, sep=",")

```

3. 解析指令函数

在初始化完成后，允许用户使用 SQL 语句进行查询。当用户输入 SQL 语句时，程序对语句进行词法单元分析与语法分析，按照空格、左右括号、等号进行语句分割，得到每个 SQL 语句的词法单元，存储到 tokens 列表中，对词法单元进行语法分析，判断具体执行哪种操作：


```
def analyse_instr(self, instr, use_db_name, if_compare=False)->list:
    # 拆出单词
    USE_DATABASE_PATH = os.path.join(self.BASE_PATH, use_db_name)
    tokens = re.split(r"([ ,();=])", instr.lower().strip())
    tokens = [t for t in tokens if t not in [' ', ',', '\n']]
```

我们设计的数据存储管理器分为四个模块，分别是添加模块、删除模块、修改模块与查询模块，语法分析后便可以调用对应模块的代码进行操作。添加模块包括创建数据库、创建数据表、添加记录；删除模块包括删除数据库、删除数据表、删除记录；修改模块包括修改记录；查询模块包括：查看数据库、查看数据表、查看元数据、查询记录。下面分别对上述模块的具体实现进行介绍。

3.1 添加模块

3.1.1 创建数据库

当用户输入 `create database database_name` 指令时，程序首先会进入创建数据库分支，然后根据用户输入的数据库名称，判断新建的数据库是否存在。若数据库不存在则程序更新元数据表中的信息，然后再新建数据库，输出创建成功信息；若数据库已经存在，则输出数据库已经存在信息。

```
if tokens[0] == "create" and tokens[1] == "database":
    db_meta_data_df = pd.read_csv(self.DB_META_DATA_PATH)
    if pd.isnull(db_meta_data_df.loc[0, "db_id"]):
        db_id = 0
    else:
        db_id = db_meta_data_df.shape[0]
    new_db_name = tokens[2]
    if not os.path.exists(new_db_name):
        os.mkdir(new_db_name)
        print("create database", new_db_name, "successfully!")
    else:
        print("Error: database already exist! ")
        exit()
    db_stat = os.stat(new_db_name)
    db_meta_data_df.loc[db_id, "db_id"] = db_id
    db_meta_data_df.loc[db_id, "db_name"] = new_db_name
    db_meta_data_df.loc[db_id, "create_time"] =
time.ctime(db_stat.st_ctime)
    db_meta_data_df.loc[db_id, "is_del"] = False
    db_meta_data_df.to_csv(self.DB_META_DATA_PATH, index=False, sep=",")
```

当用户输入 `use database_name` 指令指定使用某已存在的数据库时，程序就会把路径设置为相应数据库路径。本项目的设计逻辑是一个文件夹代表一个数据库，一个 csv 文件代表一张关系表，所以在用户想要切换数据库时，只需要输入 `use [db_name]` 指令，解析指令函数如下：

```
def analyse_use_db(self, instr)->str:
    _, db_name = instr.split(" ")
```

```

        db_meta_data_df = pd.read_csv(self.DB_META_DATA_PATH)
        is_del = db_meta_data_df[db_meta_data_df["db_name"] ==
db_name]["is_del"].values.tolist()[0]
        if db_name not in db_meta_data_df["db_name"].values or is_del == True:
            print("Error: database`", db_name, "`not exist! ")
            exit()
        print("当前数据库:", db_name)

        return db_name

```

3.1.2 创建数据表

当用户使用 `create table tb_name(type1 col1, type2 col2, primary key(col1), foreign key(cid) references course(cid))` 型语句创建名为 `tb_name` 的数据表时，分析词法单元的第一个与第二个是否为 `create` 和 `table`，然后读取 `database_meta_data.csv` 得到数据库元数据。根据用户选择的数据库或者默认数据库名称获取对应数据库 `id`；再读取 `table_meta_data.csv` 得到数据表元数据，为即将要新创建的数据表分配 `table_id`。

```

        elif tokens[0] == "create" and tokens[1] == "table":
            # 读取元库数据并获得所属数据库 id
            db_meta_data_df = pd.read_csv(self.DB_META_DATA_PATH)
            db_id = int(db_meta_data_df[db_meta_data_df["db_name"] ==
use_db_name]["db_id"].values[0])
            # 读取元表数据并为新表分配 id
            tb_meta_data_df = pd.read_csv(self.TB_META_DATA_PATH)
            if pd.isnull(tb_meta_data_df.loc[0, "table_id"]):
                table_id = 0
            else:
                table_id = tb_meta_data_df.shape[0]

```

接下来，从词法单元中获取新表的名称并判断对应数据库中是否有未被删除且重名的表，若有则异常退出：

```

        # 获取新表 name
        new_tb_name = tokens[2]
        temp_df = tb_meta_data_df[tb_meta_data_df["is_del"] == False]
        tb_name_list = temp_df[temp_df["db_id"] ==
db_id]["table_name"].values
        if new_tb_name in tb_name_list:
            print("Error: table", new_tb_name, "already exist! ")
            exit()

```

然后从 `token` 中获取新表的属性和对应的属性类型，并且判断属性数量与类型数量是否相同、类型是否合法：

```

        # 获取新表属性以及类型
        col_tokens = [] # 属性和类型 list
        if "primary" in tokens:
            col_tokens = tokens[4:tokens.index("primary")-1]
        else:
            col_tokens = tokens[4:-1]
        # 属性 list

```

```

col_name_list = col_tokens[::3]
# 类型 list
col_type_list = col_tokens[1::3]
# 判断属性和类型数目是否相同
if len(col_name_list) != len(col_type_list):
    print("Error: type error.")
    exit()
# 判断类型必须是已有的类型
for i in col_type_list:
    if i not in self.type_list:
        print("Error: type error.")
        exit()

```

接下来需要设置主键，注意判断主键属性是否存在：

```

# 设置主键
pk = ""
if "primary" in tokens:
    pk = tokens[tokens.index("primary")+3]
    if pk not in col_name_list:
        print("Error: primary key error.")
        exit()

```

如果有外键，同样需要设置，注意异常处理：

```

# 设置外键
fk = "null"
fk_r = ""
if "foreign" in tokens:
    fk = tokens[tokens.index("foreign")+3] # 设置的外键值
    if "references" in tokens:
        fk_r = tokens[tokens.index("references")+1] # 外键所在的表
        fk_r_v = tokens[tokens.index("references")+3] # 外键值
        # 设置的外键矛盾，报错
        if fk != fk_r_v:
            print("Error: foreign key error.")
            exit()
    # 没有 references，报错
    else:
        print("Error: foreign key error.")
        exit()
    temp_df = tb_meta_data_df[tb_meta_data_df["is_del"] == False]
    fk_r_cols = temp_df[temp_df["table_name"] == fk_r].values
    print(fk_r_cols)
    # 没有 references 的表，报错
    if fk_r_v not in fk_r_cols:
        print("Error: foreign key error.")
        exit()
    else:
        fk_type = temp_df[temp_df["table_name"] ==
fk_r]["type_list"].values.tolist()[0]
        col_name_list.append(fk)

```

```
print(fk_type)
col_type_list.append(fk_type)
```

最后需要创建 table_name.csv 文件，并将该表相关的数据更新到 table_meta_data 元数据表中

```
# 新增表
new_table_df = pd.DataFrame(columns=col_name_list)
new_table_path = os.path.join(USE_DATABASE_PATH, new_tb_name + ".csv")
new_table_df.to_csv(new_table_path, index=False, sep=',')
print("create table", new_tb_name, "successfully!")
table_stat = os.stat(new_table_path)
# 更新表元数据
join_str = ' '
col_name_str = join_str.join(col_name_list)
col_type_str = join_str.join(col_type_list)
tb_meta_data_df.loc[table_id, 'db_id'] = db_id
tb_meta_data_df.loc[table_id, 'table_id'] = table_id
tb_meta_data_df.loc[table_id, 'table_name'] = new_tb_name
tb_meta_data_df.loc[table_id, 'column_list'] = col_name_str
tb_meta_data_df.loc[table_id, 'type_list'] = col_type_str
tb_meta_data_df.loc[table_id, 'primary_key'] = pk
tb_meta_data_df.loc[table_id, 'foreign_key'] = fk
tb_meta_data_df.loc[table_id, 'row_num'] = 0
tb_meta_data_df.loc[table_id, 'size_in_byte'] = table_stat.st_size
tb_meta_data_df.loc[table_id, 'modify_time'] =
time.ctime(table_stat.st_mtime)
tb_meta_data_df.loc[table_id, 'create_time'] =
time.ctime(table_stat.st_ctime)
tb_meta_data_df.loc[table_id, 'uid'] = table_stat.st_uid
tb_meta_data_df.loc[table_id, 'gid'] = table_stat.st_gid
tb_meta_data_df.loc[table_id, 'is_del'] = False
tb_meta_data_df.to_csv(self.TB_META_DATA_PATH, index=False, sep=',')
```

创建数据表完毕。

3.1.2 添加记录

添加记录时需要先对 insert 的 SQL 语句进行正则匹配，判断语句是否是 insert 语句，匹配成功后再进行解析：

```
elif re.fullmatch(self.insert_instr, instr):
    # INSERT INTO table_name values(a,b,c,d) 命令格式
    tokens=[i for i in re.split(r"([ ,();=])", instr.lower().strip()) if
i not in[' ',',','(',')','(',')']]
    table_name=tokens[2]
    idx=tokens.index("values")
```

读取 csv 数据文件：

```
table_path = os.path.join(USE_DATABASE_PATH, table_name + ".csv")
try:
    table_df = pd.read_csv(table_path)
```

```

        meta_data_df = pd.read_csv(self.TB_META_DATA_PATH)
        index = meta_data_df[meta_data_df["table_name"] ==
table_name].index.tolist()[0]
    except FileNotFoundError:
        print("Error: file not found.")
        exit()

```

插入前需要检查插入数据是否过多、外键和主键是否合法等问题：

```

    for column in table_df.columns:
        idx+=1
        new[column] = tokens[idx]
    table_df.loc[table_df.shape[0]] = list(new.values())

    # 外键约束
    fk = str(meta_data_df.loc[index, "foreign_key"])
    if fk == "null":
        pass
    elif fk.count(" ") == 1:
        rtable, fk_v = fk.split(" ")
        try:
            rtable_path = os.path.join(USE_DATABASE_PATH, rtable + ".csv")
            rdf = pd.read_csv(rtable_path)
        except FileNotFoundError:
            print("Error: file not found.")
            exit()
        if new[fk_v] not in [str(i) for i in list(rdf.loc[:, fk_v])]:
            print("Error: fk constraint error.")
            exit()

    # 主键约束
    pk = str(meta_data_df.loc[index, "primary_key"])
    if new[pk] in [str(i) for i in list(table_df.loc[:, pk][: -1])]:
        print("Error: pk constraint error.")
        exit()

```

最后保存数据并更新修改时间、数据大小等信息即可：

```

table_df.to_csv(table_path, index=False, sep=',')
stat = os.stat(table_path)

meta_data_df.loc[index, "row_num"] += 1
meta_data_df.loc[index, "modify_time"] = time.ctime(stat.st_mtime)
meta_data_df.loc[index, "size_in_byte"] = stat.st_size
meta_data_df.to_csv(self.TB_META_DATA_PATH, index=False, sep=',')
print("insert successfully!")

```

3.2 删除模块

3.2.1 删除数据库

删除数据库前需要解析 SQL 语句，如果符合删除数据库的正则表达式，就从语句中得到要删除的数据库的名称：

```
elif re.fullmatch(self.drop_database_instr, instr):
    _, _, db = instr.split(" ")
    db_table_temp = pd.read_csv(self.DB_META_DATA_PATH)
```

然后从数据库表文件中查看是否有这个名称对应的并且其删除标记为 FALSE 的元组：

```
# 从数据库表文件中查看是否有这个名称对应的并且其删除标记为 FALSE 的元组
try:
    index = db_table_temp[(db_table_temp["db_name"] == db) \
        & (db_table_temp["is_del"] == False)].index.tolist()[0]
except:
    # 如果没有，要删除的数据库不存在
    print("Error: 没有" + db + "数据库!")
    exit()
```

查找存放所有表数据的表，选择所有属于该数据库且删除标记为 FALSE 的元组：

```
# 如果存在，那么去存放所有表数据的表，将所有属于该数据库且删除标记为 FALSE
# 的表的删除标记改为 TRUE
# 表示已经删除
db_table_temp.loc[db_table_temp["db_name"] == db, 'is_del'] = True
# 获得数据库 id
db_id_temp = db_table_temp.at[index, 'db_id']
db_table_temp.to_csv(self.DB_META_DATA_PATH, index=False, sep=",")
# 查找存放所有表数据的表，选择所有属于该数据库且删除标记为 FALSE 的元组（也
# 就是属于该数据库的表）
# 如果有元组被选择，将上述元组的删除标记改为 TRUE，表示已经删除
tables = pd.read_csv(self.TB_META_DATA_PATH)
tables.loc[tables["db_id"] == db_id_temp, 'is_del'] = True
tables.to_csv(self.TB_META_DATA_PATH, index=False, sep=",")
```

3.2.2 删除数据表

删除数据表思路同上，不再赘述：

```
# 删除表
elif re.fullmatch(self.drop_table_instr, instr):
    # 解析 sql 语句，如果符合删除数据库的正则表达式，那么从语句中得到要删除的表
    # 的名称
    _, _, tb = instr.split(" ")
    db_table_temp = pd.read_csv(self.DB_META_DATA_PATH)
    try:
        # 如果数据库表文件中有这个名称对应的并且其删除标记为 FALSE 的元组，继续
        index = \
            db_table_temp[
```

```

        (db_table_temp["db_name"] == use_db_name) &\
        (db_table_temp["is_del"] == False)].index.tolist()[0]
    except:
        # 提示选择数据库
        print("Error: 没有" + use_db_name + "数据库! ")
        exit()
    # 获得数据库 id
    db_id_temp = db_table_temp.at[index, 'db_id']
    tables = pd.read_csv(self.TB_META_DATA_PATH)
    try:
        # 查找存放所有表数据的表, 选择所有属于该数据库且表名为需要删除的表名,
        # 删除标记为 FALSE 的元组
        temp = tables[(tables["table_name"] == tb) & (tables["db_id"] ==
db_id_temp) \
        & (tables["is_del"] == False)].index.tolist()[0]
    except:
        # 如果没有元组被选择, 提示要删除的表不存在
        print("Error: 没有" + tb + "表! ")
        exit()
    # 如果有元组被选择, 将上述元组的删除标记改为 TRUE, 表示已经删除
    tables.loc[(tables["table_name"] == tb) & (tables["db_id"] ==
db_id_temp), "is_del"] = True
    tables.to_csv(self.TB_META_DATA_PATH, index=False, sep=",")

```

3.2.3 删除记录

思路同上, 不再赘述:

```

# 删除表项
elif re.fullmatch(self.delete_where_table_instr, instr):
    # 解析 sql 语句, 如果符合删除数据库的正则表达式
    # 那么从语句中得到要删除的表项所在的表的名称以及筛选条件
    _, _, tb, _, condition = instr.split(" ")
    db_table_temp = pd.read_csv(self.DB_META_DATA_PATH)
    try:
        # 从全局变量获得现在选择的数据库名称
        # 如果数据库表文件中有这个名称对应的并且其删除标记为 FALSE 的元组, 继续
        index = \
            db_table_temp[
                (db_table_temp["db_name"] == use_db_name) &\
                (db_table_temp["is_del"] == False)].index.tolist()[0]
    except:
        # 否则, 提示选择数据库
        print("Error: 没有" + use_db_name + "数据库! ")
        exit()
    # 获得数据库 id
    db_id_temp = db_table_temp.at[index, 'db_id']
    try:

```

```

        # 查找存放所有表数据的表，选择所有属于该数据库且表名为需要删除的表名，
        删除标记为 FALSE 的元组
        # 如果有元组被选择，说明要删除的表项所在的表存在，继续
        tables = pd.read_csv(use_db_name + "\\\" + tb + ".csv")
        tables1 = pd.read_csv(self.TB_META_DATA_PATH)
        temp = tables1[(tables1["table_name"] == tb) & (tables1["db_id"]
== db_id_temp) & (
            tables1["is_del"] == False)].index.tolist()[0]
    except:
        # 如果没有元组被选择，提示要删除的表项所在的表不存在
        print("Error: 没有" + tb + "表!")
        exit()
    # 读取要删除的表项所在的表，然后删除所有满足筛选条件的行
    if condition.find("<") != -1:
        condition_list = condition.split("<")
        tables = tables.drop(tables[tables[condition_list[0]] <
int(condition_list[1])].index)
    elif condition.find(">") != -1:
        condition_list = condition.split(">")
        tables = tables.drop(tables[tables[condition_list[0]] >
int(condition_list[1])].index)
    elif condition.find("=") != -1:
        condition_list = condition.split("=")
        tables = tables.drop(tables[tables[condition_list[0]] ==
int(condition_list[1])].index)
    tables.to_csv(use_db_name + "\\\" + tb + ".csv", index=False, sep=",")

```

3.3 修改模块

修改模块的功能只有修改记录，先对 update 的 SQL 语句进行正则匹配，然后解析获得表名和待插入的数据：

```

    elif re.fullmatch(self.update_instr, instr):
        # update Person set FirstName=a where LastName=b
        tokens=[i for i in re.split(r"([ ,();=])", instr.lower().strip()) if
i not in[' ', ',', ')', '(', '=', '']]
        table_name=tokens[1]

```

读取 csv 数据文件：

```

        table_path = os.path.join(USE_DATABASE_PATH, table_name + ".csv")
        try:
            table_df = pd.read_csv(table_path)
            meta_data_df = pd.read_csv(self.TB_META_DATA_PATH)
            index_m = meta_data_df[meta_data_df["table_name"] ==
table_name].index.tolist()[0]
        except FileNotFoundError:
            print("Error: file not found.")
            exit()

```


判断键值合法性后保存数据并更新修改时间、数据大小等信息：

```
attr=tokens[-2]          # where 后的索引 name
attrv=tokens[-1]         # attr 的值
change_item=tokens[3]    # 修改表项 name
newv=tokens[4]           # 修改值
try:
    table_df[change_item]=table_df[change_item].astype('str')
    table_df[attr]=table_df[attr].astype('str')
    if(len(table_df.loc[table_df[attr]==attrv,change_item].index)==0):
        exit()
    table_df.loc[table_df[attr]==attrv,change_item]=newv
except :
    print("Error: wrong key.")
    exit()
table_df.to_csv(table_path, index=False, sep=',')
stat = os.stat(table_path)
meta_data_df.loc[index_m, "row_num"] += 1
meta_data_df.loc[index_m, "modify_time"] = time.ctime(stat.st_mtime)
meta_data_df.loc[index_m, "size_in_byte"] = stat.st_size
meta_data_df.to_csv(self.TB_META_DATA_PATH, index=False, sep=',')
print("update successfully!")
```

3.4 查询模块

3.4.1 查看数据库

当用户使用 `show databases` 语句时，读取 `database_meta_data.csv` 得到元数据库数据，查找未被标记删除的记录的 `db_name` 字段并打印显示：

```
elif tokens[0] == "show" and tokens[1] == "databases":
    db_meta_data_df = pd.read_csv(self.DB_META_DATA_PATH)
    db_name_list = db_meta_data_df[db_meta_data_df["is_del"] ==
False]["db_name"].values
    for db_name in db_name_list:
        print(db_name)
```

3.4.2 查看数据表

当用户使用 `show tables` 语句时，思路同上：

```
elif tokens[0] == "show" and tokens[1] == "tables":
    db_meta_data_df = pd.read_csv(self.DB_META_DATA_PATH)
    db_id = int(db_meta_data_df[db_meta_data_df["db_name"] ==
use_db_name]["db_id"].values[0])
    tb_meta_data_df = pd.read_csv(self.TB_META_DATA_PATH)
    temp_df = tb_meta_data_df[tb_meta_data_df["is_del"] == False]
    tb_name_list = temp_df[temp_df["db_id"] ==
db_id]["table_name"].values
    for tb_name in tb_name_list:
        print(tb_name)
```

3.4.3 查看元数据

当用户使用 `show meta data` 语句时，读取 `database_meta_data.csv` 得到元数据库数据，读取 `table_meta_data.csv` 得到元数据表数据，分别进行打印显示：

```
elif tokens[0] == "show" and tokens[1] == "meta" and tokens[2] == "data":
    tb_meta_data_df = pd.read_csv(self.TB_META_DATA_PATH)
    db_meta_data_df = pd.read_csv(self.DB_META_DATA_PATH)
    print('table_meta_data:', tb_meta_data_df.columns.values)
    print('database_meta_data:', db_meta_data_df.columns.values)
```

3.4.4 查询记录

当用户使用 `select col_1, ... , col_n from tb_name` 语句时，程序先解析指令得到表名和列名，然后显示相应的属性，分别进行打印显示：

```
elif re.fullmatch(self.select_table_instr, instr):
    _, attr, _, table_name = instr.split(" ")
    attr_list = attr.split(",")
    try:
        table_path = os.path.join(USE_DATABASE_PATH, table_name + ".csv")
        table_df = pd.read_csv(table_path)
    except FileNotFoundError:
        print("Error: file not found.")
        exit()
    print(table_df.loc[:, attr_list])
```

当用户使用 `select * from table_name` 语句时，输出所有属性：

```
elif tokens[0] == "select" and tokens[1] == "*" and len(tokens) <= 4:
    table_name = tokens[3]
    try:
        table_path = os.path.join(USE_DATABASE_PATH, table_name + ".csv")
        table_df = pd.read_csv(table_path)
    except FileNotFoundError:
        print("Error: file not found.")
        exit()
    print(table_df)
```

当用户使用 `select column_name, ... from table_name where condition` 查询时：

```
elif re.fullmatch(self.select_where_table_instr, instr):
    t1 = int(round(time.time() * 1000))
    _, attr, _, table_name, _, condition = instr.split(" ")
    attr_list = attr.split(",")
    if condition.find("<") != -1:
        condition_list = condition.split("<")
        condition_list.append("<")
    elif condition.find(">") != -1:
        condition_list = condition.split(">")
        condition_list.append(">")
    elif condition.find("=") != -1:
        condition_list = condition.split("=")
        condition_list.append("=")
```

```

try:
    table_df = pd.read_csv(os.path.join(USE_DATABASE_PATH, table_name
+ ".csv"))
except FileNotFoundError:
    print("Error: file not found.")
    exit()
t2 = int(round(time.time() * 1000))
if condition_list[-1] == "=":
    ans = table_df[table_df[condition_list[0]] ==
int(condition_list[1])]
elif condition_list[-1] == ">":
    ans = table_df[table_df[condition_list[0]] >
int(condition_list[1])]
elif condition_list[-1] == "<":
    ans = table_df[table_df[condition_list[0]] <
int(condition_list[1])]
t3 = int(round(time.time() * 1000))
if attr == "all":
    print(ans)
else:
    print(ans.loc[:, attr_list])
t4 = int(round(time.time() * 1000))
print("读取文件时间(ms): ", t2 - t1)
print("查找时间(ms): ", t3 - t2)
print("打印结果时间(ms): ", t4 - t3)
print("总时间(ms): ", t4 - t1)

```

六、功能测试

➤ 创建数据库:

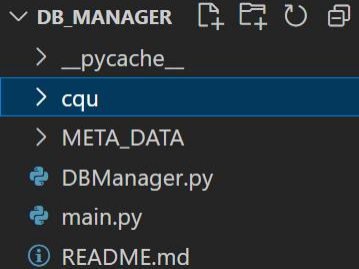
通过 `create database cqu` 语句创建名为 `cqu` 的数据库:

```

=====欢迎使用数据库系统=====
- 请输入指令: create database cqu
create database cqu successfully!

```

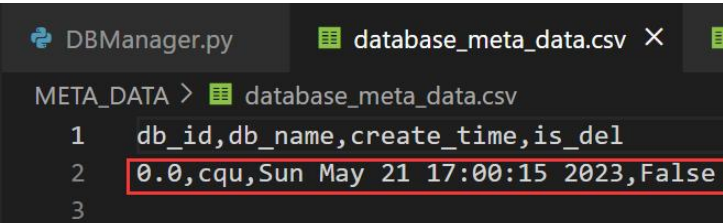
创建后, 会在根目录下生成相应文件夹, 在 `database_meta_data.csv` 中也可看到该数据库的相关信息:



```

DB_MANAGER
├── __pycache__
├── cqu
├── META_DATA
├── DBManager.py
├── main.py
└── README.md

```



```

META_DATA > database_meta_data.csv
1 db_id,db_name,create_time,is_del
2 0.0,cqu,Sun May 21 17:00:15 2023,False
3

```

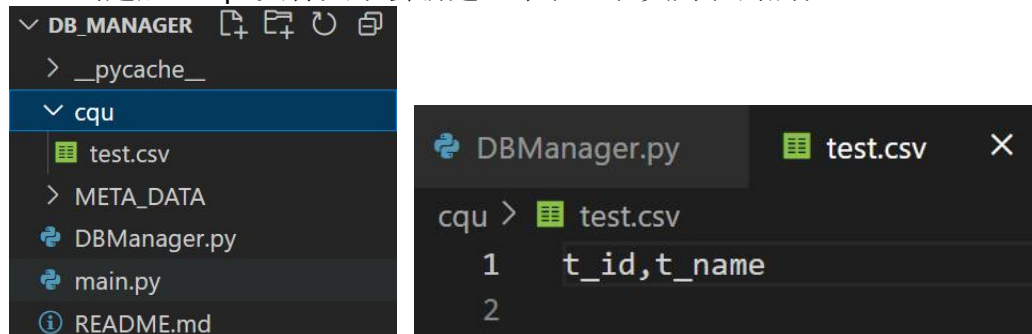
➤ 创建数据表:

通过 `use cqu` 语句选择数据库，随后使用 `create table test(t_id int, t_name varchar, primary key(t_id))` 在数据库下创建表:

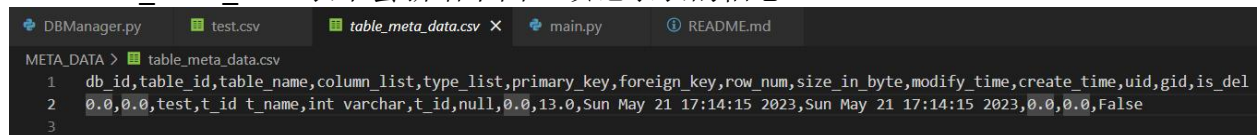
```
=====
- 请输入指令: use cqu
当前数据库: cqu

=====
- 请输入指令: create table test(t_id int, t_name varchar, primary key(t_id))
d:\LKH\重大\Curricular\大三下\数据库系统\project\project2\DB_Manager\DBManager.py:123: FutureWarning: elementwise comparison failed; returning scalar instead, but in the future will perform elementwise comparison
  if new_tb_name in tb_name_list:
create table test successfully!
```

创建后，cqu 文件夹下会新建一个表，表头为表的属性:



table_meta_data 表中会新增下面一项记录表的信息:



➤ 添加记录:

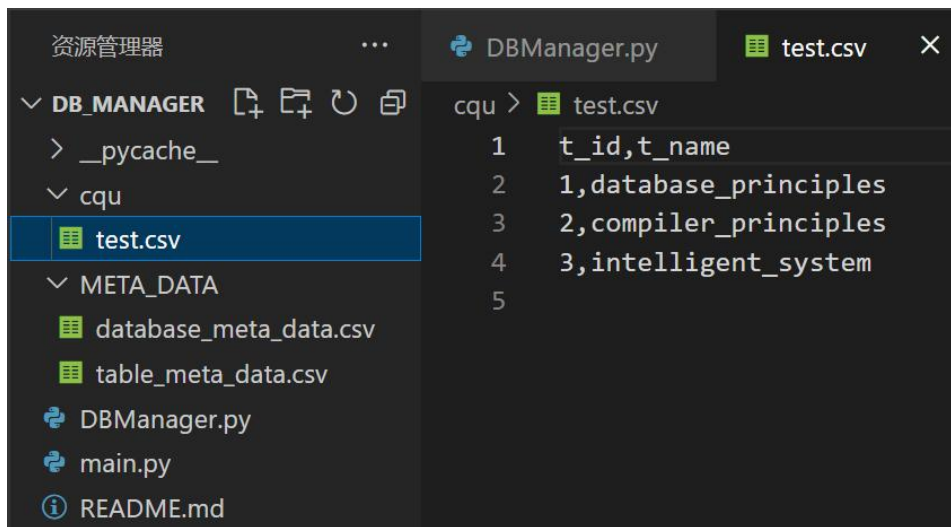
通过 `use cqu` 选定数据库，然后使用 `insert into test values(1,database_principles)` 语句添加记录:

```
=====
- 请输入指令: insert into test values(1,database_principles)
insert successfully!

=====
- 请输入指令: insert into test values(2,compiler_principles)
insert successfully!

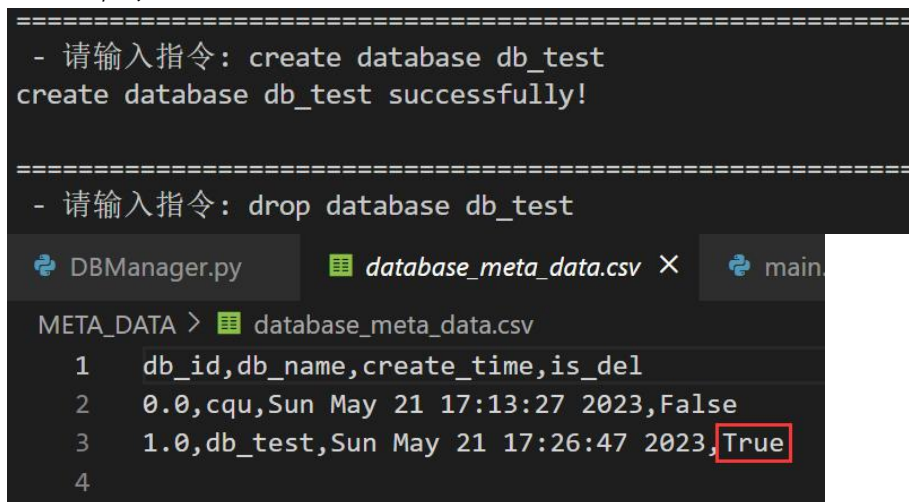
=====
- 请输入指令: insert into test values(3,intelligent_system)
insert successfully!
```

数据被写入 csv 文件:



➤ 删除数据库:

先创建一个数据库 `create database db_test` 以供删除, 通过 `drop database db_test` 删除名为 `db_test` 的数据库; 本操作为软删除, 在 `database_meta_data` 中将 `is_del` 标记为 `TRUE` 即可:



➤ 删除数据表:

先创建一个数据库 `use cqu;create table test2` 以供删除, 通过 `drop table test2` 删除名为 `test2` 的数据表; 本操作为软删除, 在 `table_meta_data` 中将 `is_del` 标记为 `TRUE`:


```
=====
- 请输入指令: use cqu
当前数据库: cqu

=====
- 请输入指令: create table test2
create table test2 successfully!

=====
- 请输入指令: drop table test2

=====
DBManager.py  table_meta_data.csv  main.py  README.md
META_DATA > table_meta_data.csv
1 db_id,table_id,table_name,column_list,type_list,primary_key,foreign_key,row_num,size_in_byte,modify_time,create_time,uid,gid,is_del
2 0.0,0.0,test,t_id,t_name,int varchar,t_id,3.0,81.0,Sun May 21 17:22:32 2023,Sun May 21 17:14:15 2023,0.0,0.0,False
3 0.0,1.0,test2,,,,,0.0,2.0,Sun May 21 17:30:23 2023,Sun May 21 17:30:23 2023,0.0,0.0,True
4
```

➤ 删除记录:

通过 `delete from test where t_id=0` 语句在数据表 `test` 中删除 `t_id=0` 的记录;

```
=====
- 请输入指令: delete from test where t_id=1
```

删除前后的截图如下所示:

```
DBManager.py  test.csv  X
cqu > test.csv
1 t_id,t_name
2 1,database_principles
3 2,compiler_principles
4 3,intelligent_system
```

```
DBManager.py  test.csv  X
cqu > test.csv
1 t_id,t_name
2 2,compiler_principles
3 3,intelligent_system
```

➤ 修改记录:

使用 `update test set t_name=new_name where t_id=2` 语句更新数据表中指定条件的记录:

```
=====
- 请输入指令: update test set t_name=new_name where t_id=2
update successfully!
```

将 `t_id` 为 0 的记录, `t_name` 改为 `test_name_0`, 左图为改之前, 右图为改之后:

```
DBManager.py  test.csv  X
cqu > test.csv
1 t_id,t_name
2 2,compiler_principles
3 3,intelligent_system
```

```
DBManager.py  test.csv  X
cqu > test.csv
1 t_id,t_name
2 2,new_name
3 3,intelligent_system
4
```

➤ 查询记录:

通过 `select * from test`、`select t_id from test`、`select t_name from test where t_id=2` 语句查询数据表中的全部记录、某个属性的记录、按条件查询记录;

```
- 请输入指令: use cqu
当前数据库: cqu

=====
- 请输入指令: select * from test
  t_id          t_name
0      2          new_name
1      3 intelligent_system

=====
- 请输入指令: select t_id from test
  t_id
0      2
1      3

=====
- 请输入指令: select t_name from test where t_id=2
  t_name
0 new_name
读取文件时间(ms): 5
查找时间(ms): 0
打印结果时间(ms): 2
总时间(ms): 7
```

➤ 查看数据库:

通过 `show databases` 语句查看现有的数据库;

```
=====
- 请输入指令: show databases
cqu
```

由于 `db_test` 已经被删除, 因此只剩下 `cqu` 这一个数据库。

➤ 查看数据表:

进入 `cqu` 数据库通过 `show tables` 语句查看现有的数据表;

```
=====
- 请输入指令: use cqu
当前数据库: cqu

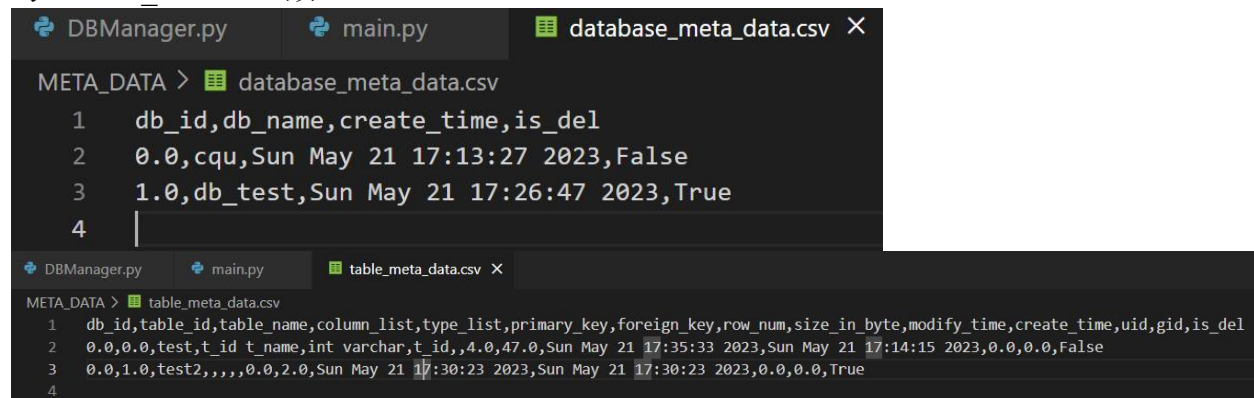
=====
- 请输入指令: show tables
test
```

➤ 查看元数据:

通过 `show meta data` 语句查看数据存储管理器中的元数据;

```
=====
- 请输入指令: show meta data
table_meta_data: ['db_id' 'table_id' 'table_name' 'column_list' 'type_list' 'primary_key'
'foreign_key' 'row_num' 'size_in_byte' 'modify_time' 'create_time' 'uid'
'gid' 'is_del']
database_meta_data: ['db_id' 'db_name' 'create_time' 'is_del']
```

与 META_DATA 一致:



```
DBManager.py  main.py  database_meta_data.csv X
META_DATA > database_meta_data.csv
1 db_id,db_name,create_time,is_del
2 0.0,cqu,Sun May 21 17:13:27 2023,False
3 1.0,db_test,Sun May 21 17:26:47 2023,True
4

DBManager.py  main.py  table_meta_data.csv X
META_DATA > table_meta_data.csv
1 db_id,table_id,table_name,column_list,type_list,primary_key,foreign_key,row_num,size_in_byte,modify_time,create_time,uid,gid,is_del
2 0.0,0.0,test,t_id t_name,int varchar,t_id,,4.0,47.0,Sun May 21 17:35:33 2023,Sun May 21 17:14:15 2023,0.0,0.0,False
3 0.0,1.0,test2,,,,,0.0,2.0,Sun May 21 17:30:23 2023,Sun May 21 17:30:23 2023,0.0,0.0,True
4
```

七、索引性能分析

1. 索引的使用原因

我们在学习或者实验使用中的数据库的表，表项一般都不会超过 10 条，因此我们并未对数据库的查询性能有很直观的感受，而当我们查询的表有十万百万级别的表项的时候，普通查询的效率就会极低。例如如下的 student 表：

sid 学号	sname 名字	sage 年龄
--------	----------	---------

如果这张表里有 50 万条数据，并且主键 sid 是无序的，那么此时要找学号为 20206666 的学生，需要遍历 50 万条记录，这样的查询代价是无法接受的。而如更新、插入、删除这类操作都是建立在查询之上，因此，我们必须探索一种高效的查询机制，而这种机制就是索引。

2. 索引简介

索引是一种与(数据库)文件相关联的附加结构，额外增加的一个辅助文件，包括顺序索引和散列索引，索引由很多索引项构成，索引项由一个搜索码值和指向具有该搜索码值的一条/多条记录的指针构成(索引项/索引记录是构成索引结构/索引文件中的基本要素)，搜索码是用于在文件中查找记录的属性/属性组。例如，sid- (sid,sname,sage) 就是一个索

引项，sid 是搜索码。

3. 索引实现原理

3.1 顺序索引

顺序索引就是把搜索码顺序排序，这样我们就可以实现查找速度的提升，从最小的搜索码开始遍历直到找到记录，或者当前的搜索码大于要搜索的值，此时的查找时间复杂度为 $O(n/2)$ 左右。我们还可以对有序的搜索码使用二分查找，这样的查找时间复杂度为 $O(\log n)$ 左右，极大的提高了查找效率，50 万条记录平均只查 19 次。

3.2 散列索引

散列索引采用散列函数将搜索码映射到散列桶，通过散列索引，支持基于搜索码的记录快速查找。本质其实就是一个哈希表，key 是搜索码，value 是记录。当我们寻找记录的时候，只需要将要搜索的值通过散列函数映射查看是否存在即可，这样的查找时间复杂度为 $O(1)$ 左右，查找效率非常高。

在实现了高效率的查找之后，不仅仅是查找的性能，基于查询的其他三个操作，更新、插入、删除，性能也可以得到提升。

4. 系统中的实现

我们在本数据存储管理系统中使用的是 pandas+Excel 的形式来模拟数据库，因此我们可以使用 pandas 的索引来模拟数据库的索引机制，注意当我们不使用索引的时候，pandas 的查找基本是无序遍历，也就是 $O(n)$ 复杂度。如果使用索引的话：

- 如果搜索码是唯一的，Pandas 会使用哈希表优化，查询性能为 $O(1)$ ；
- 如果搜索码不是唯一的，但是有序，Pandas 会使用二分查找算法，查询性能为 $O(\log N)$ ；
- 如果搜索码是完全随机的，那么每次查询都要扫描全表，查询性能为 $O(N)$ ；


```

test_name = "易海涛"
test_name_list = table_df["纳税人名称"].tolist()
t1 = int(round(time.time() * 1000))
for i in range(len(test_name_list)):
    if test_name_list[i] == test_name:
        print(table_df.iloc[i])
t2 = int(round(time.time() * 1000))
print("=*10,无索引用时: ", t2-t1, "ms", "=*10)

test_record = 42270
t3 = int(round(time.time() * 1000))
ans_df = table_df.iloc[test_record]
t4 = int(round(time.time() * 1000))
print(ans_df)
print("=*10,有索引用时: ", t4-t3, "ms", "=*10)

```

对比查询结果如下，无索引用时为 5ms，有索引用时为 0ms，几乎是实时的查询，可见为数据库建立索引的重要性。

Name: 42270, dtype: object

===== 无索引用时: 5 ms =====

Name: 42270, dtype: object

===== 有索引用时: 0 ms =====

当然，我们也可以使用 index 更多更强大的索引数据结构支持：

- CategoricalIndex，基于分类数据的索引
- MultiIndex，多维索引，用于 groupby 多维聚合后结果等
- DatetimeIndex，时间类型索引，强大的日期和时间的方法支持

八、总结

关系型数据库是一种采用自描述方式进行数据库系统管理的系统，它能够存储与关系相关的数据。本次项目我们针对关系型数据库的存储和 SQL 执行进行需求分析，设计了一个简单的数据库系统 SQL 引擎模块。该模块将根据设计原则模拟实现 SQL 的基本操作和数据存储等功能，实现了数据库创建、表格创建、数据的添加、删除和更新等操作过程中，以及数据库系统 SQL 引擎需要进行的各种管理和操作。我们设计了数据库元数据表与数据表元数据表来分别管理数据库和数据表，通过对这两张表中元数据的操作来实现数据库创建、数据表创建、数据添加、删除等操作。此外，我们还设计了索引，并比较有索引和无索引情况下的差异。

通过实现这个数据库系统 SQL 引擎模块，我们深入了解了关系型数据库的内部工作原理和基本操作的实现方式。此项目提供了一个简单而有效的数据库管理工具，为用户提供数据存储和检索的功能。同时，通过对有索引和无索引情况进行比较，我们进一步了解了索引在数据库性能中的重要性。该项目也增强了我们对关系型数据库的理解，并提高对 SQL 执行和数据库系统管理的实践能力。让我们从更底层的原理上理解数据库，同时分析建立索引带来的性能优化，这也许才是我们真正计算机科学与技术人才做的工作。