课程作业

课程名称 ： **数据库系统原理**
作业次数 ： **作业#7**
学号 ： **21281280**
姓名 ： **柯劲帆**
班级 ： **物联网2101班**
指导老师 ： **郝爽**
修改日期 ： **2024年6月3日**

1. 题目1

> **一、 存储过程和触发器实验**
>
> 1. 请在你选用的数据库平台上，针对你的应用场景，对如下操作至少各实现一个存储过程：
>   1. 单表或多表查询
>   2. 数据插入
>   3. 数据删除
>   4. 数据修改
> 2. 通过ODBC、OLEDB、JDBC或任意其他的途径，在前端程序（C/S或B/S模式）中调用所实现的后台存储过程。
> 3. 在你的案例场景中，分别设计并实现一个由数据插入、数据更新、数据删除所引发的触发器（前触发或后触发都可以），测试触发器执行效果。

1.1. 创建存储过程

1. 单表或多表查询

   用于确认用户登录的密码。

   查询指定用户ID的密码，与用户输入的密码匹配。

```
CREATE PROCEDURE VerifyUser(
    IN p_id BIGINT,
    IN p_password VARCHAR(255),
    OUT verify_status VARCHAR(255)
)
BEGIN
    DECLARE id_exist INT DEFAULT 0;
```

```
    DECLARE record_password VARCHAR(255);

    SELECT COUNT(*) INTO id_exist
    FROM passengers
    WHERE ID = p_id;

    IF id_exist = 0 THEN
        SET verify_status = 'NO_USER';
    ELSE
        SELECT `Password` INTO record_password
        FROM passengers
        WHERE ID = p_id;

        IF record_password != p_password THEN
            SET verify_status = 'WRONG_PASSWORD';
        ELSE
            SET verify_status = 'USER_VERIFIED';
        END IF;
    END IF;
END;
```

2. 数据插入

用户注册信息插入。

```
CREATE PROCEDURE RegisterPassenger(
    IN p_id BIGINT,
    IN p_name VARCHAR(255),
    IN p_phone_number BIGINT,
    IN p_password VARCHAR(255),
    OUT result_message VARCHAR(255)
)
BEGIN
    INSERT INTO passengers (ID, `Name`, Phone_number, `Password`)
    VALUES (p_id, p_name, p_phone_number, p_password);
    SET result_message = '注册成功';
END;
```

3. 数据删除和修改

```
CREATE PROCEDURE ModifyPassengerInfo(
    IN p_id BIGINT,
    IN p_modify_type VARCHAR(255),
    IN p_new_password VARCHAR(255),
    IN p_phone_number BIGINT,
    OUT result_message VARCHAR(255)
)
BEGIN
    IF p_modify_type = 'delete account' THEN
        DELETE FROM passengers WHERE ID = p_id;
        SET result_message = '删除账户成功';
    ELSEIF p_modify_type = 'modify Password' THEN
        UPDATE passengers SET `Password` = p_new_password WHERE ID =
p_id;
        SET result_message = '修改密码成功';
    ELSEIF p_modify_type = 'modify Phone_Number' THEN
```

```
        UPDATE passengers SET Phone_number = p_phone_number WHERE ID =
p_id;
        SET result_message = '修改手机号成功';
    ELSE
        SET result_message = '无效的修改类型';
    END IF;
END;
```

1.2.调用存储过程

与实验五的逻辑一致，将后台python执行sql语句，改为调用存储过程。

1. 数据插入

   将用户注册输入的信息插入表中。使用 `callproc` 方法调用存储过程，接着使用 `fetchall()` 方法获取存储过程的输出结果（这里起到清空输入缓冲区的作用），然后再调用 `execute` 方法执行 `SELECT` 操作，获取 `OUT` 参数。

   需要注意的是，不能直接 `SELECT <@参数名称>` ，而是 `SELECT <@_{存储过程名称}_{参数序号}>` ，并且使用 `fetchone()` 方法得到一个字典，参数的值为字典中键 `'@_{存储过程名称}_{参数序号}'` 对应的值。

```python
if request.method == 'POST':
    id = request.form['cardCode']
    name = request.form['name']
    phone_number = request.form['mobileNo']
    password = request.form['encryptedPassword']

    db = get_db()
    cursor = db.cursor()

    try:
        cursor.callproc('RegisterPassenger', (id, name, phone_number,
password, "@result_message"))
        cursor.fetchall()
        cursor.execute("SELECT @_RegisterPassenger_4;")
        result_message = cursor.fetchone()['@_RegisterPassenger_4']
        print(result_message)
        flash(result_message)
        db.commit()
    except pymysql.MySQLError as e:
        db.rollback()
        if e.args[0] == 1644:  # SQLSTATE 45000 corresponds to error
code 1644
            flash("乘客已存在，无法重复注册")
        else:
            print(e)
            flash("数据库异常，注册失败")
    db.close()
```

2. 数据删除、修改

```python
class ModifyInfo:
    def __init__(self, form: Dict[str, str]):
        self.id = form['cardCode']
```

```
        modifyType = form['modifyType']
        self.new_password = form['encryptedNewPassword']
        self.phone_number = form['mobileNo'] if form['mobileNo'] != ""
else "11111111111"
        modifyType2command = {
            '1': 'delete account',
            '2': 'modify Password',
            '3': 'modify Phone_Number'
        }
        self.command = modifyType2command[modifyType]

    def get_args(self):
        return (self.id, self.command, self.new_password,
self.phone_number, "@result_message")

    def get_ok_message(self, cursor):
        cursor.execute("SELECT @_ModifyPassengerInfo_4;")
        return cursor.fetchone()['@_ModifyPassengerInfo_4']


modifyInfo = ModifyInfo(request.form)
try:
    cursor.callproc('ModifyPassengerInfo', modifyInfo.get_args())
    cursor.fetchall()
    db.commit()
    flash(modifyInfo.get_ok_message(cursor))
except pymysql.MySQLError as e:
    db.rollback()
    if e.args[0] == 1644:  # SQLSTATE 45000 corresponds to error code
1644
        flash("用户不存在，无法修改")
    else:
        print(e)
        flash("数据库异常，修改失败")
db.close()
```

完整代码见附件。

1.3. 触发器实现

在用户注册、修改账户数据之前，必须验证用户是否存在。

- 用户注册时，若用户ID已存在在表中，则不可再插入相同的ID的数据。

```
CREATE TRIGGER BeforeInsertPassenger
BEFORE INSERT ON passengers
FOR EACH ROW
BEGIN
    DECLARE id_exist INT DEFAULT 0;

    SELECT COUNT(*) INTO id_exist
    FROM passengers
    WHERE ID = NEW.ID;

    IF id_exist != 0 THEN
```

```
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = '乘客已存在，无法重复注册';
    END IF;
END;
```

- 用户修改数据时，若用户ID不存在在表中，则不可修改。

```
CREATE TRIGGER BeforeModifyPassengerInfo
BEFORE UPDATE ON passengers
FOR EACH ROW
BEGIN
    DECLARE id_exist INT DEFAULT 0;

    SELECT COUNT(*) INTO id_exist
    FROM passengers
    WHERE ID = NEW.ID;

    IF id_exist = 0 THEN
        SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = '用户不存在，无法修改';
    END IF;
END;
```

经过测试，结果与实验四一致，系统能够正确向用户发出错误警告。

2. 题目2

> **索引实验**
>
> 1. 结合作业#3，针对你的数据库中的一个表，编写简单的数据查询（查询语句应包括单个涉及非主属性等值比较的查询条件，设该非主属性为A，具体属性结合业务背景）和数据插入语句，程序应能在终端或服务器以文件形式记录每次数据读写操作的耗时。
> 2. 无索引测试：执行查询（查询条件不包含主码，且不存在针对属性A建立的索引），记录不同数据规模下的查询时间，
> 3. 有索引测试：针对属性A建立索引，采用与2）中相同的查询，记录不同数据规模下的查询时间。
> 4. 分析实验数据，制作图表，比较有索引和无索引的情况下，查询时间随数据量增加的变化情况，分析导致实验结果的原因。

2.1. 编写程序

调库，设置数据量为 100,000 。

```
import pymysql
import random
from tqdm import tqdm
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

N = 100000  # 十万条数据
```

连接数据库。

```
db = pymysql.connect(
    host='127.0.0.1', user='kejingfan',
    password='PASSWORD', database='DBLab_7_2'
)
cursor = db.cursor()
```

初始化数据库，创建两个表。两个表的主属性为 `ID`，用于测试的非主属性是 `Phone_number`。

- 在表 `passengers_no_index` 中**不针对** `Phone_number` 建立索引;
- 在表 `passengers_with_index` 中**针对** `Phone_number` 建立索引。

使用 `SET profiling = 1;` 设置使用 `PROFILES`，记录近15次操作中每次操作的时间花费。该时间花费是 MySQL 内部的计时，没有网络延迟等误差影响计算结果。

```
sql_statements = [
    "SET profiling = 1;",

    "DROP TABLE IF EXISTS passengers_no_index;",
    "DROP TABLE IF EXISTS passengers_with_index;",
    """
    CREATE TABLE passengers_no_index (
        ID BIGINT PRIMARY KEY,
        `Name` VARCHAR (255) NOT NULL,
        Phone_number BIGINT NOT NULL,
        `Password` VARCHAR (255) NOT NULL,
        CHECK (ID REGEXP '^\\\\d{18}$'),
        CHECK (Phone_number REGEXP '^\\\\d{11}$')
    );
    """,
    """
    CREATE TABLE passengers_with_index (
        ID BIGINT PRIMARY KEY,
        `Name` VARCHAR (255) NOT NULL,
        Phone_number BIGINT NOT NULL,
        `Password` VARCHAR (255) NOT NULL,
        CHECK (ID REGEXP '^\\\\d{18}$'),
        CHECK (Phone_number REGEXP '^\\\\d{11}$')
    );
    """,
    "CREATE INDEX idx_phone_number ON passengers_with_index (Phone_number);",
]

for sql in sql_statements:
    cursor.execute(sql)
db.commit()
```

初始化写入的数据，并定义数组存储数据。

```python
id_list = random.sample(range(1000000000000000000, 1000000000000000000), N)
phone_number_list = random.sample(range(10000000000, 20000000000), N)

insert_times = {
    'passengers_no_index': [],
    'passengers_with_index': []
}
query_times = {
    'passengers_no_index': [],
    'passengers_with_index': []
}
```

分别对表 `passengers_no_index` 和表 `passengers_with_index` 进行插入操作，并读取 `PROFILES` 中的计时数据，写入列表中。

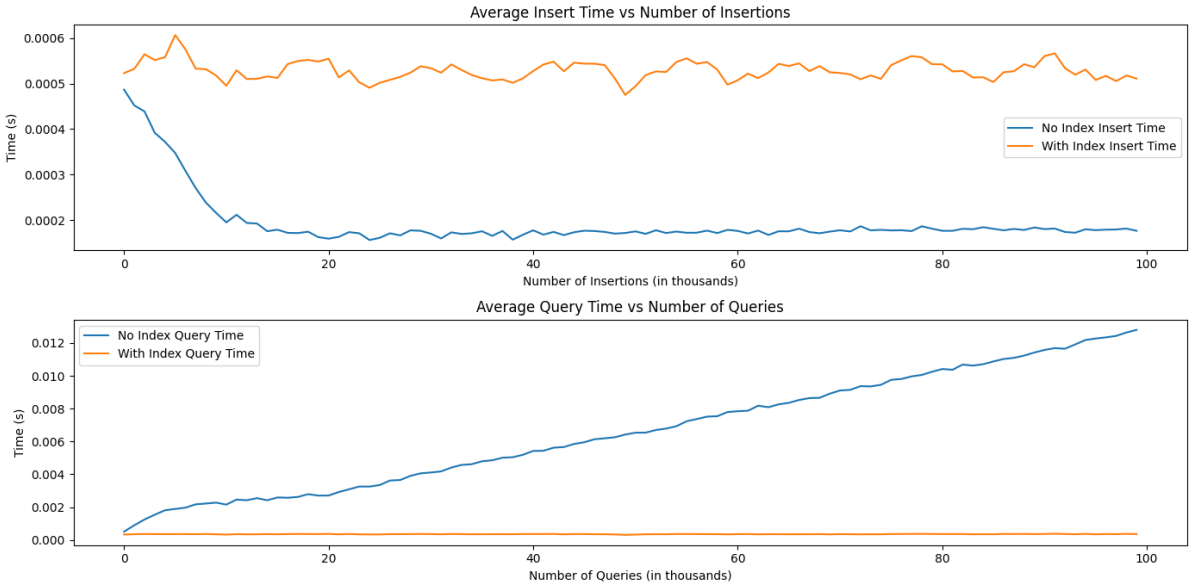
```python
for table_name in ['passengers_no_index', 'passengers_with_index']:
    print(f"操作数据表 {table_name} : ")
    insert_sql = f'''
        INSERT INTO {table_name} (ID, `Name`, Phone_number, `Password`)
        VALUES (%s, %s, %s, %s);
    '''
    query_sql = f'''
        SELECT * FROM {table_name}
        WHERE Phone_number = %s;
    '''

    for i in tqdm(range(N)):
        cursor.execute(insert_sql, (id_list[i], 'kejingfan',
phone_number_list[i], 'password'))
        db.commit()
        cursor.execute(query_sql, (phone_number_list[random.randint(0,
i)],))
        cursor.execute("SHOW PROFILES;")
        profiles = cursor.fetchall()[-3:]
        profile = profiles[0]
        if "INSERT INTO" in profile[2]:
            insert_times[table_name].append(profile[1])
        profile = profiles[-1]
        if "SELECT * FROM" in profile[2]:
            query_times[table_name].append(profile[1])

cursor.close()
db.close()
```

```
操作数据表 passengers_no_index :
100%|
████████████████████████████████████████████████████████████████████████
███| 100000/100000 [20:55<00:00, 79.67it/s]
操作数据表 passengers_with_index :
100%|
████████████████████████████████████████████████████████████████████████
██| 100000/100000 [13:12<00:00, 126.20it/s]
```

导出数据为 csv 文件，并画图（将 100,000 条数据每 100 条做平均，使得折线图平滑易读）：

```
data = {
    'insert_times_no_index': insert_times['passengers_no_index'],
    'insert_times_with_index': insert_times['passengers_with_index'],
    'query_times_no_index': query_times['passengers_no_index'],
    'query_times_with_index': query_times['passengers_with_index']
}
df = pd.DataFrame(data)
df.to_csv('performance_times.csv', index=True)

def get_average_per_n(data, n):
    return [np.mean(data[i:i + n]) for i in range(0, len(data), n)]

avg_insert_times_no_index =
get_average_per_n(insert_times['passengers_no_index'], 1000)
avg_insert_times_with_index =
get_average_per_n(insert_times['passengers_with_index'], 1000)
avg_query_times_no_index =
get_average_per_n(query_times['passengers_no_index'], 1000)
avg_query_times_with_index =
get_average_per_n(query_times['passengers_with_index'], 1000)

plt.figure(figsize=(14, 7))

plt.subplot(2, 1, 1)
plt.plot(range(len(avg_insert_times_no_index)), avg_insert_times_no_index,
label='No Index Insert Time')
plt.plot(range(len(avg_insert_times_with_index)),
avg_insert_times_with_index, label='With Index Insert Time')
plt.xlabel('Number of Insertions (in thousands)')
plt.ylabel('Time (s)')
plt.title('Average Insert Time vs Number of Insertions')
plt.legend()

plt.subplot(2, 1, 2)
plt.plot(range(len(avg_query_times_no_index)), avg_query_times_no_index,
label='No Index Query Time')
plt.plot(range(len(avg_query_times_with_index)), avg_query_times_with_index,
label='With Index Query Time')
plt.xlabel('Number of Queries (in thousands)')
plt.ylabel('Time (s)')
plt.title('Average Query Time vs Number of Queries')
plt.legend()

plt.tight_layout()
plt.show()
```

2.2. 实验结果及比较



可见对于查询操作：

- **有索引**的表耗时为常数；
- **无索引**的表耗时与数据量呈正比例增长。

对于插入操作：

- **无索引**的表插入耗时一开始较多，最后趋于平稳；
- **有索引**的表插入耗时变化较为平稳，但始终比无索引的用时多约 10 倍。

以上运行结果的实验环境1为：

操作系统	CPU
Ubuntu 22.04.1（6.5.0-35-generic)	12th Gen Intel(R) Core(TM) i7-12700H（20 核)

我又用以下的实验环境2重新运行实验：

操作系统	CPU
5.15.146.1-microsoft-standard-WSL2	11th Gen Intel(R) Core(TM) i7-1165G7（8 核)

结果为：

对于查询操作，结果用时比实验环境1运行用时多（硬件性能导致），但相对性能相近：

- **有索引**的表耗时为常数；
- **无索引**的表耗时与数据量呈正比例增长。

对于插入操作：

- 两个表的插入用时均不稳定；
- **有索引**的表比**无索引**的用时少。

2.3. 实验结论

综合以上两个实验环境的运行结果，得出结论：

在重复数据少的属性上，建立索引有助于减少查询时间。但是建立索引可能对插入耗时造成无稳定的影响（具体影响因硬件和操作系统不同而不同）。