数据库实验

第四五次上机实验

```sql
CREATE DATABASE travel_system;
USE travel_system;
```

根据题目中的需求，设计如下表结构：

```sql
-- 航班信息表
CREATE TABLE FLIGHTS (
    flightNum VARCHAR(10) PRIMARY KEY,    -- 航班编号
    price INT,                            -- 价格
    numSeats INT,                         -- 总座位数
    numAvail INT,                         -- 可用座位数
    FromCity VARCHAR(50),                 -- 出发城市
    ArivCity VARCHAR(50)                  -- 到达城市
);

-- 宾馆信息表
CREATE TABLE HOTELS (
    location VARCHAR(50) PRIMARY KEY,     -- 地理位置
    price INT,                            -- 价格
    numRooms INT,                         -- 总房间数
    numAvail INT                          -- 可用房间数
);

-- 大巴信息表
CREATE TABLE BUS (
    location VARCHAR(50) PRIMARY KEY,     -- 地理位置
    price INT,                            -- 价格
    numBus INT,                           -- 总大巴数量
    numAvail INT                          -- 可用大巴数量
);

-- 客户信息表
CREATE TABLE CUSTOMERS (
    custID INT PRIMARY KEY AUTO_INCREMENT,  -- 客户ID
    custName VARCHAR(50) NOT NULL           -- 客户姓名
);

-- 预订信息表
CREATE TABLE RESERVATIONS (
    resvID INT PRIMARY KEY AUTO_INCREMENT,  -- 预订ID
    custID INT,                             -- 客户ID，外键
    resvType INT,                           -- 预订类型 (1: 航班, 2: 宾馆, 3: 大巴)
    resvKey VARCHAR(50),                    -- 预订关键字 (如航班号, 宾馆位置, 大巴位置)
    FOREIGN KEY (custID) REFERENCES CUSTOMERS(custID)
);
```

1.2. 数据库一致性约束

为了保持表FLIGHTS、HOTELS和BUS的一致性，确保每个航班、酒店房间或大巴的可用数量与预订数量匹配，我们可以通过触发器（trigger）来实现。假设我们希望在每次插入或删除预订时，自动调整`numAvail`的值。

```
-- 更改分隔符为 //
DELIMITER //

-- 航班预订触发器：插入预订时减少航班可用座位
CREATE TRIGGER flight_reservation_insert AFTER INSERT ON RESERVATIONS
FOR EACH ROW
BEGIN
    IF NEW.resvType = 1 THEN
        UPDATE FLIGHTS
        SET numAvail = numAvail - 1
        WHERE flightNum = NEW.resvKey;
    END IF;
END;
//

-- 航班预订取消触发器：删除预订时增加航班可用座位
CREATE TRIGGER flight_reservation_delete AFTER DELETE ON RESERVATIONS
FOR EACH ROW
BEGIN
    IF OLD.resvType = 1 THEN
        UPDATE FLIGHTS
        SET numAvail = numAvail + 1
        WHERE flightNum = OLD.resvKey;
    END IF;
END;
//

-- 宾馆预订触发器：插入预订时减少宾馆可用房间
CREATE TRIGGER hotel_reservation_insert AFTER INSERT ON RESERVATIONS
FOR EACH ROW
BEGIN
    IF NEW.resvType = 2 THEN
        UPDATE HOTELS
        SET numAvail = numAvail - 1
        WHERE location = NEW.resvKey;
    END IF;
END;
//

-- 宾馆预订取消触发器：删除预订时增加宾馆可用房间
CREATE TRIGGER hotel_reservation_delete AFTER DELETE ON RESERVATIONS
FOR EACH ROW
BEGIN
    IF OLD.resvType = 2 THEN
        UPDATE HOTELS
        SET numAvail = numAvail + 1
        WHERE location = OLD.resvKey;
    END IF;
END;
```

```
//

-- 大巴预订触发器：插入预订时减少大巴可用数量
CREATE TRIGGER bus_reservation_insert AFTER INSERT ON RESERVATIONS
FOR EACH ROW
BEGIN
    IF NEW.resvType = 3 THEN
        UPDATE BUS
        SET numAvail = numAvail - 1
        WHERE location = NEW.resvKey;
    END IF;
END;
//

-- 大巴预订取消触发器：删除预订时增加大巴可用数量
CREATE TRIGGER bus_reservation_delete AFTER DELETE ON RESERVATIONS
FOR EACH ROW
BEGIN
    IF OLD.resvType = 3 THEN
        UPDATE BUS
        SET numAvail = numAvail + 1
        WHERE location = OLD.resvKey;
    END IF;
END;
//

-- 恢复默认分隔符为 ;
DELIMITER ;
```

2. **基本功能实现**

2.1. 航班、大巴车、宾馆房间和客户数据的入库与更新

插入新的航班、大巴、宾馆房间和客户数据的SQL语句可以如下：

```
-- 插入航班数据
INSERT INTO FLIGHTS (flightNum, price, numSeats, numAvail, FromCity,
ArivCity)
VALUES ('CA123', 500, 100, 100, 'Beijing', 'Shanghai');

-- 插入宾馆数据
INSERT INTO HOTELS (location, price, numRooms, numAvail)
VALUES ('Beijing', 300, 50, 50);

-- 插入大巴数据
INSERT INTO BUS (location, price, numBus, numAvail)
VALUES ('Beijing', 50, 5, 5);

-- 插入客户数据
INSERT INTO CUSTOMERS (custName)
VALUES ('John Doe');
```

更新航班、大巴、宾馆房间和客户数据的SQL语句可以如下：

```
-- 更新航班价格
```

```sql
UPDATE FLIGHTS
SET price = 550
WHERE flightNum = 'CA123';

-- 更新宾馆房间价格
UPDATE HOTELS
SET price = 350
WHERE location = 'Beijing';

-- 更新大巴价格
UPDATE BUS
SET price = 60
WHERE location = 'Beijing';

-- 更新客户姓名
UPDATE CUSTOMERS
SET custName = 'Jane Doe'
WHERE custID = 1;
```

2.2. 预订航班、大巴车、宾馆房间

为了预订航班、大巴车或宾馆房间，可以通过往`RESERVATIONS`表中插入记录：

```sql
-- 预订航班
INSERT INTO RESERVATIONS (custID, resvType, resvKey)
VALUES (1, 1, 'CA123');  -- 1表示航班

-- 预订宾馆房间
INSERT INTO RESERVATIONS (custID, resvType, resvKey)
VALUES (1, 2, 'Beijing');  -- 2表示宾馆

-- 预订大巴车
INSERT INTO RESERVATIONS (custID, resvType, resvKey)
VALUES (1, 3, 'Beijing');  -- 3表示大巴
```

2.3. 查询航班、大巴车、宾馆房间、客户和预订信息

查询航班、大巴车、宾馆房间信息：

```sql
-- 查询所有航班信息
SELECT * FROM FLIGHTS;

-- 查询所有宾馆信息
SELECT * FROM HOTELS;

-- 查询所有大巴信息
SELECT * FROM BUS;

-- 查询所有客户信息
SELECT * FROM CUSTOMERS;

-- 查询所有预订信息
SELECT * FROM RESERVATIONS;
```

查询某个客户的所有预订信息：

```sql
-- 查询某个客户的预订信息
SELECT R.resvID, R.resvType, R.resvKey
FROM RESERVATIONS R
JOIN CUSTOMERS C ON R.custID = C.custID
WHERE C.custName = 'John Doe';
```

2.4. 查询某个客户的旅行线路

旅行线路包括航班、大巴车和宾馆房间的预订信息。以下SQL查询可以列出某个客户的所有预订信息：

```sql
-- 查询某个客户的旅行线路
SELECT
    CASE
        WHEN R.resvType = 1 THEN 'Flight'
        WHEN R.resvType = 2 THEN 'Hotel'
        WHEN R.resvType = 3 THEN 'Bus'
    END AS ReservationType,
    R.resvKey
FROM RESERVATIONS R
JOIN CUSTOMERS C ON R.custID = C.custID
WHERE C.custName = 'John Doe';
```

检查预定线路的完整性

为了检查某个客户是否预订了完整的旅行线路（即航班、大巴车、宾馆房间都预订了），可以通过以下SQL查询来检查：

```sql
-- 检查客户是否有完整的预订（航班、大巴、宾馆房间）
SELECT custName,
       SUM(CASE WHEN resvType = 1 THEN 1 ELSE 0 END) AS FlightCount,
       SUM(CASE WHEN resvType = 2 THEN 1 ELSE 0 END) AS HotelCount,
       SUM(CASE WHEN resvType = 3 THEN 1 ELSE 0 END) AS BusCount
FROM RESERVATIONS R
JOIN CUSTOMERS C ON R.custID = C.custID
WHERE C.custName = 'John Doe'
GROUP BY custName
HAVING FlightCount > 0 AND HotelCount > 0 AND BusCount > 0;
```

该查询将返回客户的预订信息，只有当客户预订了航班、大巴和宾馆房间时，结果才会显示。如果返回空结果集，则说明客户没有完整的预订。

样例如下：

```
mysql> -- 查看航班信息表的所有数据
mysql> SELECT * FROM FLIGHTS;
+-----------+-------+----------+----------+-----------+-------------+
| flightNum | price | numSeats | numAvail | FromCity  | ArivCity    |
+-----------+-------+----------+----------+-----------+-------------+
| BA200     |  1200 |      180 |       59 | London    | New York    |
| CA123     |   550 |      100 |       96 | Beijing   | Shanghai    |
| CZ789     |   600 |      120 |       49 | Guangzhou | Beijing     |
| MU456     |   800 |      150 |       74 | Shanghai  | Guangzhou   |
| UA100     |  1000 |      200 |      149 | New York  | Los Angeles |
+-----------+-------+----------+----------+-----------+-------------+
5 rows in set (0.00 sec)

mysql>
mysql> -- 查看宾馆信息表的所有数据
mysql> SELECT * FROM HOTELS;
+-----------+-------+----------+----------+
| location  | price | numRooms | numAvail |
+-----------+-------+----------+----------+
| Beijing   |   350 |       50 |       46 |
| Guangzhou |   350 |       40 |       19 |
| London    |   900 |       80 |       49 |
| New York  |  1000 |      100 |       79 |
| Shanghai  |   400 |       60 |       29 |
+-----------+-------+----------+----------+
5 rows in set (0.00 sec)

mysql>
mysql> -- 查看大巴信息表的所有数据
mysql> SELECT * FROM BUS;
+-----------+-------+--------+----------+
| location  | price | numBus | numAvail |
+-----------+-------+--------+----------+
| Beijing   |    60 |      5 |        3 |
| Guangzhou |    55 |     12 |        4 |
| London    |    95 |     18 |        9 |
| New York  |   100 |     20 |       14 |
| Shanghai  |    60 |     15 |        9 |
+-----------+-------+--------+----------+
5 rows in set (0.00 sec)

mysql>
mysql> -- 查看客户信息表的所有数据
mysql> SELECT * FROM CUSTOMERS;
+--------+---------------+
| custID | custName      |
+--------+---------------+
|      1 | Jane Doe      |
|      2 | John Doe      |
|      3 | Jane Smith    |
|      4 | Alice Johnson |
|      5 | Bob Lee       |
|      6 | Charlie Brown |
|      7 | Chen Lu       |
|      8 | Qin Che       |
+--------+---------------+
8 rows in set (0.00 sec)
```

```
mysql> SELECT
    ->     r.resvID,
    ->     c.custName,
    ->     CASE r.resvType
    ->         WHEN 1 THEN '航班'
    ->         WHEN 2 THEN '宾馆'
    ->         WHEN 3 THEN '大巴'
    ->     END as 预订类型,
    ->     r.resvKey as 预订详情
    -> FROM RESERVATIONS r
    -> JOIN CUSTOMERS c ON r.custID = c.custID
    -> WHERE c.custID IN (7,8);
+--------+----------+--------------+--------------+
| resvID | custName | 预订类型     | 预订详情     |
+--------+----------+--------------+--------------+
|     24 | Chen Lu  | 航班         | CA123        |
|     25 | Chen Lu  | 宾馆         | Shanghai     |
|     26 | Qin Che  | 航班         | CZ789        |
|     27 | Qin Che  | 宾馆         | Beijing      |
|     28 | Qin Che  | 大巴         | Beijing      |
+--------+----------+--------------+--------------+
5 rows in set (0.00 sec)
```

第三次上机实验

1.创建一个函数，为所有存款账户增加 1%的利息。

```
DELIMITER //

CREATE FUNCTION add_interest()
RETURNS INT
DETERMINISTIC
BEGIN
    UPDATE account
    SET balance = balance * 1.01;
    RETURN 1;
END //

DELIMITER ;
```

显示一下结果

```
-- 重新插入account表的数据
INSERT INTO account (account_number, branch_name, balance) VALUES
('A-101', 'Downtown', 500),
('A-102', 'Perryridge', 400),
('A-201', 'Brighton', 900),
('A-215', 'Mianus', 700),
('A-217', 'Brighton', 750),
('A-222', 'Redwood', 700),
('A-305', 'Round Hill', 350);

-- 然后再次执行查看和更新操作
-- 1. 查看原始余额
SELECT * FROM account;

-- 2. 执行增加利息的函数
SELECT add_interest();

-- 3. 查看增加利息后的余额
SELECT * FROM account;

-- 4. 对比显示
SELECT
    account_number,
    branch_name,
    balance AS original_balance,
    balance * 1.01 AS balance_after_interest,
    balance * 0.01 AS interest_earned
FROM account;
```

2.创建一个新表 branch_total，用于存储各个支行的存款总额（表中有branch_name 和 total_balance 两个属性）。然后在这个表上，创建一个触发器，实现当有用户存款变动（包括增加、删除和更新）时，brach_total表中的存款总额跟着自动更新.

首先创建branch_total表并初始化数据：

```sql
CREATE TABLE branch_total (
    branch_name VARCHAR(50) PRIMARY KEY,
    total_balance DECIMAL(12, 2)
);

-- 初始化branch_total表的数据
INSERT INTO branch_total (branch_name, total_balance)
SELECT branch_name, COALESCE(SUM(balance), 0)
FROM account
GROUP BY branch_name;
```

然后创建触发器：

```sql
DELIMITER //

-- 当插入新账户时
CREATE TRIGGER update_branch_total_insert
AFTER INSERT ON account
FOR EACH ROW
BEGIN
    UPDATE branch_total
    SET total_balance = total_balance + NEW.balance
    WHERE branch_name = NEW.branch_name;
END //

-- 当更新账户时
CREATE TRIGGER update_branch_total_update
AFTER UPDATE ON account
FOR EACH ROW
BEGIN
    IF NEW.branch_name = OLD.branch_name THEN
        UPDATE branch_total
        SET total_balance = total_balance - OLD.balance + NEW.balance
        WHERE branch_name = NEW.branch_name;
    ELSE
        UPDATE branch_total
        SET total_balance = total_balance - OLD.balance
        WHERE branch_name = OLD.branch_name;

        UPDATE branch_total
        SET total_balance = total_balance + NEW.balance
        WHERE branch_name = NEW.branch_name;
    END IF;
END //

-- 当删除账户时
CREATE TRIGGER update_branch_total_delete
AFTER DELETE ON account
FOR EACH ROW
BEGIN
    UPDATE branch_total
    SET total_balance = total_balance - OLD.balance
    WHERE branch_name = OLD.branch_name;
END //
```

```
42
43 DELIMITER ;
```

我们可以通过以下步骤来验证触发器的功能：

```sql
-- 1. 首先查看当前branch_total表的状态
SELECT * FROM branch_total ORDER BY branch_name;

-- 2. 查看当前account表的状态
SELECT * FROM account ORDER BY account_number;

-- 3. 测试插入新账户
INSERT INTO account (account_number, branch_name, balance)
VALUES ('A-999', 'Downtown', 1000);

-- 查看branch_total的变化（Downtown分行的总额应该增加1000）
SELECT * FROM branch_total ORDER BY branch_name;

-- 4. 测试更新账户余额
UPDATE account
SET balance = 2000
WHERE account_number = 'A-999';

-- 查看branch_total的变化（Downtown分行的总额应该再增加1000）
SELECT * FROM branch_total ORDER BY branch_name;

-- 5. 测试更改账户的分行
UPDATE account
SET branch_name = 'Brighton'
WHERE account_number = 'A-999';

-- 查看branch_total的变化（Downtown总额应减少2000，Brighton总额应增加2000）
SELECT * FROM branch_total ORDER BY branch_name;

-- 6. 测试删除账户
DELETE FROM account
WHERE account_number = 'A-999';

-- 查看branch_total的变化（Brighton总额应减少2000）
SELECT * FROM branch_total ORDER BY branch_name;
```

第二次上机实验

1.创建视图

```sql
CREATE DATABASE IF NOT EXISTS bank_database;
USE bank_database;

CREATE TABLE account (
account_number VARCHAR(10) PRIMARY KEY,
branch_name VARCHAR(50),
balance DECIMAL(10, 2)
);
INSERT INTO account (account_number, branch_name, balance) VALUES
```

```
('A-101', 'Downtown', 500),
('A-102', 'Perryridge', 400),
('A-201', 'Brighton', 900),
('A-215', 'Mianus', 700),
('A-217', 'Brighton', 750),
('A-222', 'Redwood', 700),
('A-305', 'Round Hill', 350);

CREATE TABLE branch (
branch_name VARCHAR(50) PRIMARY KEY,
branch_city VARCHAR(50),
assets DECIMAL(12, 2)
);
INSERT INTO branch (branch_name, branch_city, assets) VALUES
('Brighton', 'Brooklyn', 7100000),
('Downtown', 'Brooklyn', 9000000),
('Mianus', 'Horseneck', 400000),
('North Town', 'Rye', 3700000),
('Perryridge', 'Horseneck', 1700000),
('Pownal', 'Bennington', 300000),
('Redwood', 'Palo Alto', 2100000),
('Round Hill', 'Horseneck', 8000000);

CREATE TABLE customer (
customer_name VARCHAR(50) PRIMARY KEY,
customer_street VARCHAR(50),
customer_city VARCHAR(50)
);
INSERT INTO customer (customer_name, customer_street, customer_city) VALUES
('Adams', 'Spring', 'Pittsfield'),
('Brooks', 'Senator', 'Brooklyn'),
('Curry', 'North', 'Rye'),
('Glenn', 'Sand Hill', 'Woodside'),
('Green', 'Walnut', 'Stamford'),
('Hayes', 'Main', 'Harrison'),
('Johnson', 'Alma', 'Palo Alto'),
('Jones', 'Main', 'Harrison'),
('Lindsay', 'Park', 'Pittsfield'),
('Smith', 'North', 'Rye'),
('Turner', 'Putnam', 'Stamford'),
('Williams', 'Nassau', 'Princeton');

CREATE TABLE depositor (
customer_name VARCHAR(50),
account_number VARCHAR(10),
PRIMARY KEY (customer_name, account_number)
);
INSERT INTO depositor (customer_name, account_number) VALUES
('Hayes', 'A-102'),
('Johnson', 'A-101'),
('Johnson', 'A-201'),
('Jones', 'A-217'),
('Lindsay', 'A-222'),
('Smith', 'A-215'),
('Turner', 'A-305');

```

```sql
CREATE TABLE loan (
loan_number VARCHAR(10) PRIMARY KEY,
branch_name VARCHAR(50),
amount DECIMAL(10, 2)
);
INSERT INTO loan (loan_number, branch_name, amount) VALUES
('L-11', 'Round Hill', 900),
('L-14', 'Downtown', 1500),
('L-15', 'Perryridge', 1500),
('L-16', 'Perryridge', 1300),
('L-17', 'Downtown', 1000),
('L-23', 'Redwood', 2000),
('L-93', 'Mianus', 500);

CREATE TABLE borrower (
customer_name VARCHAR(50),
loan_number VARCHAR(10),
PRIMARY KEY (customer_name, loan_number)
);
INSERT INTO borrower (customer_name, loan_number) VALUES
('Adams', 'L-16'),
('Curry', 'L-93'),
('Hayes', 'L-15'),
('Jackson', 'L-14'),
('Jones', 'L-17'),
('Smith', 'L-11'),
('Smith', 'L-23'),
('Williams', 'L-17');
```

以下是正确的 SQL 语句:

```sql
CREATE VIEW branch_detail AS
SELECT
  br.branch_name,
  COUNT(DISTINCT d.customer_name) AS deposit_customer_count,
  SUM(a.balance) AS deposit_total,
  COUNT(DISTINCT b.customer_name) AS loan_customer_count,
  SUM(l.amount) AS loan_total
FROM
  branch br
  LEFT JOIN depositor d ON d.account_number IN (
    SELECT account_number
    FROM account
    WHERE branch_name = br.branch_name
  )
  LEFT JOIN account a ON br.branch_name = a.branch_name
  LEFT JOIN borrower b ON b.loan_number IN (
    SELECT loan_number
    FROM loan
    WHERE branch_name = br.branch_name
  )
  LEFT JOIN loan l ON br.branch_name = l.branch_name
GROUP BY
  br.branch_name;
```

我们可以使用以下 SQL 语句来查看视图的内容:

```
SELECT * FROM branch_detail;
```

这将返回一个包含以下列的结果集:

- `branch_name`: 支行名称
- `deposit_customer_count`: 该支行的存款客户数量
- `deposit_total`: 该支行的存款总额
- `loan_customer_count`: 该支行的贷款客户数量
- `loan_total`: 该支行的贷款总额

2.比较速度

首先,让我们先清空 `account` 表中的数据:

```
TRUNCATE TABLE account;
```

然后,我们再插入 1000 条记录:

```
START TRANSACTION;
SET @i = 1;
WHILE @i <= 1000 DO
  INSERT INTO account (account_number, branch_name, balance)
  VALUES (CONCAT('A-', LPAD(@i, 3, '0')), 'Branch', 1000.00);
  SET @i = @i + 1;
END WHILE;
COMMIT;
```

接下来,我们来测试有索引和无索引的查询时间差异:

无索引查询:

```
SET @start_time = NOW();
SELECT * FROM account WHERE account_number = 'A-500';
SELECT TIMEDIFF(NOW(), @start_time) AS query_time;
```

这将输出无索引查询的耗时。

有索引查询:

```
CREATE INDEX idx_account_number ON account (account_number);

SET @start_time = NOW();
SELECT * FROM account WHERE account_number = 'A-500';
SELECT TIMEDIFF(NOW(), @start_time) AS query_time;
```

这将输出有索引查询的耗时。

3.插入删除更新的权限

1. 首先,创建branch_manager角色:

```
CREATE ROLE 'branch_manager';
```

然后,为branch_manager角色授予对branch表的INSERT、DELETE和UPDATE权限:

```
GRANT INSERT, DELETE, UPDATE ON bank_database.branch TO 'branch_manager';
```

如果您希望branch_manager也能查看branch表的数据,可以额外授予SELECT权限:

```
GRANT SELECT ON bank_database.branch TO 'branch_manager';
```

创建一个用户并将其与branch_manager角色关联(假设用户名为'manager1'):

```
CREATE USER 'manager1'@'localhost' IDENTIFIED BY 'password';
GRANT 'branch_manager' TO 'manager1'@'localhost';
```

最后,使角色对用户生效:

```
SET DEFAULT ROLE 'branch_manager' TO 'manager1'@'localhost';
```

要验证您创建的角色和权限是否正确设置，您可以遵循以下步骤:

首先，以管理员身份登录MySQL:

```
mysql -u root -p
```

查看创建的角色:

```
SELECT * FROM mysql.user WHERE account_locked = 'Y';
```

这将显示所有角色，包括 'branch_manager'。

检查角色权限:

```
SHOW GRANTS FOR 'branch_manager';
```

这将显示分配给 'branch_manager' 角色的所有权限。

检查用户权限:

```
SHOW GRANTS FOR 'manager1'@'localhost';
```

这将显示分配给 'manager1' 用户的所有权限，包括通过角色获得的权限。

1. 验证用户是否可以执行允许的操作:
   首先，以 'manager1' 身份登录:

```
mysql -u manager1 -p
```

然后尝试执行一些操作:

```
USE bank_database;
```

```sql

-- 尝试 SELECT（如果授予了这个权限）
SELECT * FROM branch;

-- 尝试 INSERT
INSERT INTO branch (branch_name, branch_city, assets) VALUES ('Test Branch', 'Test City', 1000000);

-- 尝试 UPDATE
UPDATE branch SET assets = 1100000 WHERE branch_name = 'Test Branch';

-- 尝试 DELETE
DELETE FROM branch WHERE branch_name = 'Test Branch';

-- 尝试对其他表进行操作（应该被拒绝）
SELECT * FROM account;
```

最后，不要忘记清理测试数据：

```sql
DELETE FROM branch WHERE branch_name = 'Test Branch';
```

第一次上机实验

首先，创建一个新的数据库：

```sql
CREATE DATABASE employee_db;
```

然后，选择这个数据库：

```sql
USE employee_db;
```

现在，您可以运行之前提供的所有创建表、插入数据和查询的SQL语句了。

让我们创建四张表:

```sql
CREATE TABLE employee (
    employee_name VARCHAR(50) PRIMARY KEY,
    street VARCHAR(100),
    city VARCHAR(50)
);

CREATE TABLE works (
    employee_name VARCHAR(50),
    company_name VARCHAR(100),
    salary DECIMAL(10, 2),
    PRIMARY KEY (employee_name, company_name),
    FOREIGN KEY (employee_name) REFERENCES employee(employee_name)
);

CREATE TABLE company (
    company_name VARCHAR(100) PRIMARY KEY,
    city VARCHAR(50)
);
```

```
19
20 CREATE TABLE managers (
21     employee_name VARCHAR(50),
22     manager_name VARCHAR(50),
23     PRIMARY KEY (employee_name, manager_name),
24     FOREIGN KEY (employee_name) REFERENCES employee(employee_name)
25 );
```

接下来,让我们插入一些示例数据:

```
1  INSERT INTO employee VALUES
2  ('John Doe', '123 Main St', 'New York'),
3  ('Jane Smith', '456 Elm St', 'Chicago'),
4  ('Bob Johnson', '789 Oak St', 'Los Angeles'),
5  ('Alice Brown', '321 Pine St', 'New York');
6
7  INSERT INTO works VALUES
8  ('John Doe', 'First Bank Corporation', 80000),
9  ('Jane Smith', 'Small Bank Corporation', 70000),
10 ('Bob Johnson', 'First Bank Corporation', 95000),
11 ('Alice Brown', 'Small Bank Corporation', 65000);
12
13 INSERT INTO company VALUES
14 ('First Bank Corporation', 'New York'),
15 ('Small Bank Corporation', 'Chicago');
16
17 INSERT INTO managers VALUES
18 ('John Doe', 'Jane Smith'),
19 ('Bob Johnson', 'Alice Brown');
```

a. 找出所有为"First Bank Corporation"工作的雇员名字及其居住城市。

```
1 SELECT e.employee_name, e.city
2 FROM employee e
3 JOIN works w ON e.employee_name = w.employee_name
4 WHERE w.company_name = 'First Bank Corporation';
```

b. 找出所有为"First Bank Corporation"工作且薪金超过10000美元的雇员名字、居住街道和城市。

```
1 SELECT e.employee_name, e.street, e.city
2 FROM employee e
3 JOIN works w ON e.employee_name = w.employee_name
4 WHERE w.company_name = 'First Bank Corporation' AND w.salary > 10000;
```

c. 找出数据库中所有不为"First Bank Corporation"工作的雇员。

```
1 SELECT e.employee_name
2 FROM employee e
3 LEFT JOIN works w ON e.employee_name = w.employee_name AND w.company_name =
  'First Bank Corporation'
4 WHERE w.company_name IS NULL OR w.company_name != 'First Bank Corporation';
```

d. 找出数据库中工资高于"Small Bank Corporation"的每个雇员的所有雇员。

```sql
SELECT DISTINCT w1.employee_name
FROM works w1
WHERE w1.salary > ALL (
    SELECT w2.salary
    FROM works w2
    WHERE w2.company_name = 'Small Bank Corporation'
);
```

e. 假设一个公司可以在好几个城市有分部。找出位于"Small Bank Corporation"所有分在城市的所有公司。

```sql
SELECT DISTINCT c1.company_name
FROM company c1
WHERE NOT EXISTS (
    SELECT c2.city
    FROM company c2
    WHERE c2.company_name = 'Small Bank Corporation'
    AND c2.city NOT IN (
        SELECT c3.city
        FROM company c3
        WHERE c3.company_name = c1.company_name
    )
);
```

1. 最外层查询:

   ```sql
   SELECT DISTINCT c1.company_name
   FROM company c1
   WHERE NOT EXISTS (...)
   ```

   这部分是在寻找满足某个条件的公司名称。

2. NOT EXISTS 子查询:

   ```sql
   NOT EXISTS (
       SELECT c2.city
       FROM company c2
       WHERE c2.company_name = 'Small Bank Corporation'
       AND c2.city NOT IN (...)
   )
   ```

   这个子查询的作用是确保不存在这样的情况：Small Bank Corporation 在某个城市有分部，而当前考虑的公司（c1）在该城市没有分部。

3. 最内层子查询:

   ```sql
   SELECT c3.city
   FROM company c3
   WHERE c3.company_name = c1.company_name这个查询获取当前考虑的公司（c1）所在的所有城市。
   ```

查询的逻辑步骤：

1. 对于 company 表中的每个公司（c1）：

2. 查找 Small Bank Corporation 所在的所有城市（c2）。
3. 检查这些城市是否都包含在当前公司（c1）的城市列表中。
4. 如果所有 Small Bank Corporation 的城市都在当前公司的城市列表中，则该公司满足条件。

换句话说，要找不存在Small Bank Corporation的城市不在c1的所有公司所在的城市中。找存在Small Bank Corporation的城市在c1的所有公司的城市中。

即这个查询在寻找那些在每个 Small Bank Corporation 有分部的城市中都有分部的公司。

这种查询方式使用了"双重否定"的逻辑：

- "不存在" (NOT EXISTS)
- "Small Bank Corporation 的城市不在当前公司的城市列表中" (NOT IN)

f. 找出雇员最多的公司。

```sql
SELECT company_name
FROM works
GROUP BY company_name
HAVING COUNT(*) = (
    SELECT COUNT(*)
    FROM works
    GROUP BY company_name
    ORDER BY COUNT(*) DESC
    LIMIT 1
);
```

g. 找出平均工资高于"First Bank Corporation"平均工资的那些公司。

```sql
SELECT company_name
FROM works
GROUP BY company_name
HAVING AVG(salary) > (
    SELECT AVG(salary)
    FROM works
    WHERE company_name = 'First Bank Corporation'
);
```

这些查询覆盖了图片中要求的所有问题。每个查询都针对特定的要求进行了优化。让我简要解释一下每个查询的思路:

a. 使用JOIN连接employee和works表,找出在First Bank Corporation工作的员工及其城市。

b. 类似于a,但增加了薪资条件和返回街道信息。

c. 使用LEFT JOIN和IS NULL来找出不在First Bank Corporation工作的员工。

d. 使用子查询和ALL关键字来比较薪资。

e. 使用NOT EXISTS和子查询来找出在所有Small Bank Corporation所在城市都有分部的公司。

f. 使用GROUP BY和子查询找出雇员数量最多的公司。

g. 使用子查询计算First Bank Corporation的平均工资,然后比较其他公司的平均工资。

join语句

JOIN 是 SQL 中用于组合两个或多个表中的行的操作。它基于这些表之间的相关列。让我详细解释一下不同类型的 JOIN：

1. INNER JOIN（内连接）：
   - 返回两个表中匹配的行。
   - 语法：`SELECT * FROM table1 INNER JOIN table2 ON table1.column = table2.column;`
   - 例子：

```sql
SELECT employee.name, works.salary
FROM employee
INNER JOIN works ON employee.name = works.employee_name;
```

   - 这会返回所有在 employee 表和 works 表中都有记录的员工信息。
2. LEFT JOIN（左连接）：
   - 返回左表中的所有行，即使右表中没有匹配。
   - 语法：`SELECT * FROM table1 LEFT JOIN table2 ON table1.column = table2.column;`
   - 例子：

```sql
SELECT employee.name, works.salary
FROM employee
LEFT JOIN works ON employee.name = works.employee_name;
```

   - 这会返回所有员工信息，即使某些员工没有工资记录。
3. RIGHT JOIN（右连接）：
   - 返回右表中的所有行，即使左表中没有匹配。
   - 语法：`SELECT * FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;`
   - 例子：

```sql
SELECT employee.name, works.salary
FROM employee
RIGHT JOIN works ON employee.name = works.employee_name;
```

   - 这会返回所有工资记录，即使某些工资记录没有对应的员工信息。
4. FULL JOIN（全连接）：
   - 返回左表和右表中的所有行，不论是否匹配。
   - 语法：`SELECT * FROM table1 FULL JOIN table2 ON table1.column = table2.column;`
   - 注意：MySQL 不直接支持 FULL JOIN，但可以通过 LEFT JOIN 和 UNION 来模拟。
5. CROSS JOIN（交叉连接）：
   - 返回两个表的笛卡尔积（所有可能的行组合）。

- 语法：`SELECT * FROM table1 CROSS JOIN table2;`
- 使用时要小心，因为结果集可能非常大。

JOIN 的关键点：

1. ON 子句：指定连接条件。
2. 多表连接：可以连接两个以上的表。
3. 别名：可以给表和列使用别名，使查询更清晰。
4. 性能：JOIN 操作可能影响查询性能，特别是在大型表上。