MTHREE TRAINING [SQL]

DAY 2

---

*Topic 1- Permissions using Binary Operations

In SQL, permissions are often managed using **binary operations** to combine or evaluate multiple permission states. Permissions are typically represented as bits, and **binary operations** (like AND, OR, NOT, etc.) are applied to manipulate or check permission sets.

Common Use Case for Binary Operations with Permissions

1.Permission Representation:
Permissions are often represented as integers or bit masks. Each bit in a binary number corresponds to a specific permission:

- 001 → Read
- 010 → Write
- 100 → Execute

2. Granting Permissions:
Use the **binary OR (|)** operation to combine permissions.
Example:
If you want to grant both Read (001) and Write (010), you perform:
001 | 010 = 011 (Read and Write)

3. Revoking Permissions:
Use the **binary AND with NOT (& ~)** operation to remove specific permissions.
Example:
To revoke Write (010) from a user with Read and Write (011):
011 & ~010 = 001 (Only Read)

4. Checking Permissions:
Use the **binary AND (&)** operation to check if specific permissions exist.
Example:
To check if Write (010) is granted in 011:
011 & 010 = 010 (Write exists)

- **From class-**

```
CREATE TABLE permissions (
    user_id INT PRIMARY KEY,
    username VARCHAR(50),
    permission_flags INT  -- Will store permission bits );
-- Insert sample data
INSERT INTO permissions (user_id, username, permission_flags) VALUES
(1, 'admin', 7),     -- Binary: 111 (Read: 1, Write: 1, Execute: 1)
(2, 'developer', 6), -- Binary: 110 (Read: 1, Write: 1, Execute: 0)
(3, 'viewer', 4),    -- Binary: 100 (Read: 1, Write: 0, Execute: 0)
(4, 'guest', 1);     -- Binary: 001 (Read: 0, Write: 0, Execute: 1)
```

To check if the user has specific permissions [here Bitwise AND(&) is used.

Question 1: If user has read permission, permission_flag=4

```
select username,
permission_flags & 4 as has_read_permission,
case
  when permission_flags & 4 > 0 then 'Yes'
  else 'No'
end as can_read
from permissions;
```

Question 2: Add write permission to all users who don't have it.

As write permission correspond to 2 doing an OR with 2 will solve this question using subquery.

```
update permissions
set permission_flags = permission_flags | 2
where (permission_flags & 2) =0
```

Question 3: Toggle the execute permission for the user.

```
select * from permissions

#toggle the execute permission for user
update permissions
set permission_flags =permission_flags ^ 1
where (permission_flags & 1)=0;
```

Little explanation for above questions-
This SQL query is toggling the "Execute" permission for a user in the permissions table using **bitwise XOR (^)**.

Query Explanation

- A combination of permissions would look like 0111 (Read + Write + Execute).

Step-by-Step Breakdown:

1. **Condition: (permission_flags & 1) = 0**
    - This checks if the "Execute" permission (1st bit) is currently **off** (not set).
    - permission_flags & 1 isolates the least significant bit (LSB) using the **bitwise AND** operation:
        - If the result is 0, the "Execute" permission is **off**.
        - If the result is 1, the "Execute" permission is **on**.
2. **Example:**
    - 0110 (6) AND 0001 (1) → 0000 (Execute is off).
    - 0111 (7) AND 0001 (1) → 0001 (Execute is on).
3. **Purpose:** This ensures the query only updates rows where "Execute" is **off**.
2. **Action: SET permission_flags = permission_flags ^ 1**
    - The **bitwise XOR (^)** operation toggles the "Execute" permission (1st bit):
        - If the bit is 0, XOR with 1 sets it to 1 (turning it on).

---

*Topic 2- Bit Shifting Operations

Bit shifting operations in SQL are used to shift the bits of an integer value either to the left or right. These operations are commonly used in scenarios like permission management, flag handling, or other bitmask-related computations.

Bit Shifting Operations

1. **Left Shift (<<)**

Shifts the bits of an integer to the left by the specified number of positions.
0010 (2) << 1 → 0100 (4)

2. **Right Shift (>>)**

Shifts the bits of an integer to the right by the specified number of positions.
0100 (4) >> 1 → 0010 (2)

- **From class- [important for the interviews]**
- Left shift, Right shift,
- When shift 1 num*2
- When left shift by 2 , num*4
- [if someone asks u to multiply without using* use this,computer does this]
- [Left shift increases the number and right shift decreases]

Question 1. Left Shift

```
CREATE TABLE bit_shift_demo (
   id INT PRIMARY KEY,
   value INT
);

INSERT INTO bit_shift_demo (id, value) VALUES
(1, 8),   -- Binary: 1000
(2, 12),  -- Binary: 1100
(3, 16);  -- Binary: 10000

select
id,
value,
value << 1 as left_shft_1,
value << 2 as left_shift_2
from bit_shift_demo;
```

Question 2. Right Shift

```
select
id,
value,
value << 1 as left_shft_1,
value << 2 as left_shift_2
from bit_shift_demo;
```

From class-

MAJOR QUESTION -

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100),
    Country VARCHAR(50),
    IsActive BIT,
    CreditLimit DECIMAL(10,2)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    TotalAmount DECIMAL(10,2),
    Status VARCHAR(20)
);

CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(100),
    Category VARCHAR(50),
    Price DECIMAL(10,2),
    InStock BIT
);

CREATE TABLE OrderDetails (
    OrderID INT,
    ProductID INT,
    Quantity INT,
    UnitPrice DECIMAL(10,2),
    PRIMARY KEY (OrderID, ProductID)
```

```
);



-- Insert sample data
INSERT INTO Customers VALUES
(1, 'John Doe', 'USA', 1, 5000.00),
(2, 'Jane Smith', 'Canada', 1, 3000.00),
(3, 'Bob Johnson', 'USA', 0, 2000.00),
(4, 'Alice Brown', 'UK', 1, 4000.00),
(5, 'Charlie Wilson', 'Canada', 1, 6000.00);

INSERT INTO Orders VALUES
(1, 1, '2024-01-01', 1500.00, 'Delivered'),
(2, 1, '2024-01-15', 2000.00, 'Pending'),
(3, 2, '2024-01-20', 1000.00, 'Delivered'),
(4, 3, '2024-02-01', 500.00, 'Cancelled'),
(5, 4, '2024-02-15', 3000.00, 'Processing');

INSERT INTO Products VALUES
(1, 'Laptop', 'Electronics', 1200.00, 1),
(2, 'Smartphone', 'Electronics', 800.00, 1),
(3, 'Desk Chair', 'Furniture', 200.00, 0),
(4, 'Coffee Maker', 'Appliances', 100.00, 1),
(5, 'Headphones', 'Electronics', 150.00, 1);

INSERT INTO OrderDetails VALUES
(1, 1, 1, 1200.00),
(1, 2, 1, 800.00),
(2, 3, 2, 200.00),
(3, 4, 1, 100.00),
(4, 5, 2, 150.00);
```

Question 1. Retrieve customer name, country where country is in USA & Canada

```
Select name,country
From Customers
Where country in('USA','Canada')
```

Question 2. Anyone who has ordered greater than USA

```
select distinct c.Name,c.Country
from Customers c
```

```
join orders o on
c.customerID =o.customerID
where c.country<>'USA' and o.totalamount >
ANY(
select totalamount
from orders o2
join customers c2 on c2.customerID=o2.customerId
where c2.country='USA'
)
```

Question 3. Products not in stock.

```
Select productName
From Products
Where not in stock; #not in stock gives negation whose boolean values are 0
```

Question 4. Order amount between 1k and 3k.

```
Select oprderID, Amount
From orders
Where totalAmount between 1000 and 3000;
```

Question 5. List all customer distinct order status.

```
Select country from customer
Union
Select distinct status from customers;
```

Question 6. Find products that are in stock true and have been ordered.

```
SELECT ProductName
FROM Products P
WHERE P.InStock = 1
AND EXISTS (SELECT 1 FROM OrderDetails OD WHERE OD.ProductID = P.ProductID);
##Same can be done with join##

SELECT DISTINCT p.ProductID, p.ProductName, p.Category, p.Price
FROM Products p
JOIN OrderDetails od ON p.ProductID = od.ProductID
WHERE p.InStock = 1;
```

---

*Topic 3- Exist Keyword

The **EXISTS** keyword in SQL is used to check whether a subquery returns any rows. It's commonly used in conditional queries to test the existence of rows in a related table or dataset.

How EXISTS Works

- EXISTS evaluates to TRUE if the subquery returns **at least one row**.
- EXISTS evaluates to FALSE if the subquery returns **no rows**.
- It is often used in WHERE clauses to filter results based on the existence of related records.

- **From class-**
- To find the customers who have placed at least one order [this could be done easily with join but try exist] exist always works with subquery
- Select 1 from customers or using only select 1 = this will always return 1
- But when places with a condition like where o.customerID=c.customerID it checks if value occurs in both or not

```
Select name

From customers c

Where exists(

Select *

From orders o

Where o.customerID=c.customerID
```

---

*Topic 4- Case in SQL

From class-

Question. Categorize the customer based on credit limit
5000=premium
>3000=gold
Less standard

```
Select Name,
Case
  When credit_limit>=5000
        Then ‘premium’
```

```
  When credit_limit>=3000
       Then 'Gold'
   Else 'Standard'
End as CustomerTier
From Customers
```

---

*Topic 5- RANK() / DENSE RANK()

1. RANK

- **Definition**: Assigns a unique rank to each row within a partition, but **skips ranks** when there are ties.
- **Behavior with Ties**: If two rows tie for a rank, the next rank skips the tied number of rows.

Syntax

RANK() OVER (PARTITION BY column ORDER BY column)

2. DENSE_RANK

- **Definition**: Similar to RANK, but it **does not skip ranks** when there are ties.
- **Behavior with Ties**: If two rows tie for a rank, the next rank is assigned without skipping any numbers.

Syntax

DENSE_RANK() OVER (PARTITION BY column ORDER BY column)

From class-[Most famous question for fresher,mostly used in companies]

Question.

```
Select name,department, salary,rank() over (order by salary desc)as SalaryRank,
Dense_rank() over(order by salary desc) as SalaryDenserank
From employees;
```

Now i want to get it dept wise:

```
Select name,department, salary,rank() over (partition by Department order by salary desc)as SalaryRank,
Dense_rank() over(partiton by order by salary desc) as SalaryDenserank
From employees;
```

Question. Both managers and their employees

```
Select e.name as Ename,
E.department, m. Name as ManagerName
```

```
From employees e
Left join employees m on e.managerid=m.empID
```

eg of self join as well as left join or inner join but inner join is more expensive .

---

*Topic 6- Monthly Trend Question / LAG

The **LAG** function in SQL is a **window function** used to access data from a previous row in the result set. It provides the value of a column from a specified number of rows "before" the current row within the same result set.

Key Features of LAG

1. **Retrieve Previous Row Values**: It allows you to compare a current row with a previous one.
2. **Flexible Offset**: You can specify how many rows back to look.
3. **Default Value**: You can provide a default value in case the previous row does not exist (e.g., for the first row).
4. **Works Within Partitions**: Often used with PARTITION BY to group rows before applying the function.

Syntax

```
LAG(column_name, offset, default_value)
OVER (PARTITION BY partition_column ORDER BY order_column)
```

- **column_name**: The column whose previous value you want to retrieve.
- **offset**: How many rows back to look (default is 1).
- **default_value**: A value to return if no row exists at the specified offset (optional, defaults to NULL).
- **PARTITION BY**: Divides the rows into partitions (optional).
- **ORDER BY**: Specifies the order of rows within each partition.

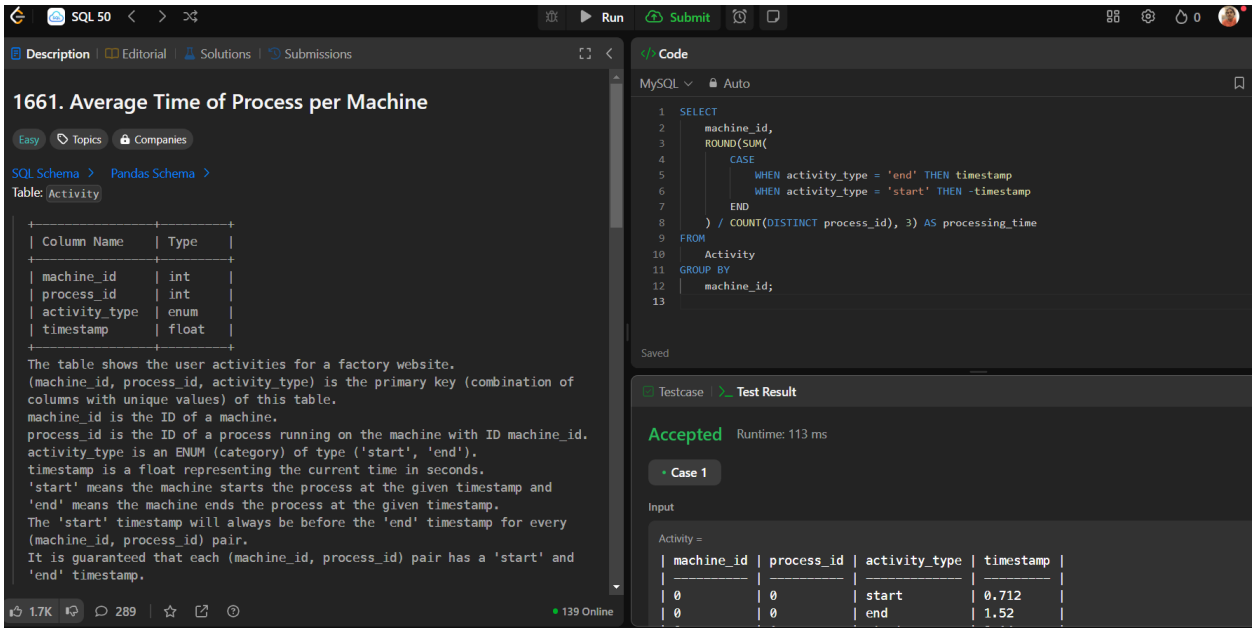
```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(100),
    Email VARCHAR(100)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    CustomerID INT,
    OrderDate DATE,
    TotalAmount DECIMAL(10,2)
```

```
);

-- Insert sample data
INSERT INTO Customers VALUES
(1, 'John Doe', 'john@email.com'),
(2, 'Jane Smith', 'jane@email.com'),
(3, 'Bob Wilson', 'bob@email.com');

INSERT INTO Orders VALUES
(1, 1, '2024-01-01', 100.00),
(2, 1, '2024-01-15', 150.00),
(3, 2, '2024-01-20', 200.00),
(4, 1, '2024-02-01', 120.00),
(5, 3, '2024-02-05', 180.00),
(6, 2, '2024-02-10', 250.00),
(7, 1, '2024-02-15', 300.00);

#monthly sales trend[since we need to check how its starting every month we need to use lag)
select
date_format(orderdate, '%Y-%m') as yearmonth,
count(*) as totalorders,
sum(totalamount) as totalsales,
avg(totalamount) as avgordervalue,
(SUM(totalAmount)-LAG(SUM(totalAmount)) over (order by date_format(orderdate,
'%Y-%m')))/(LAG(SUM(totalAmount)) over (order by date_format(orderdate, '%Y-%m'))) *100 as MoMGrowth
from orders
group by date_format(orderdate, '%Y-%m')
order by yearmonth;
```

*Codes

---

1. Leetcode Question No. 1

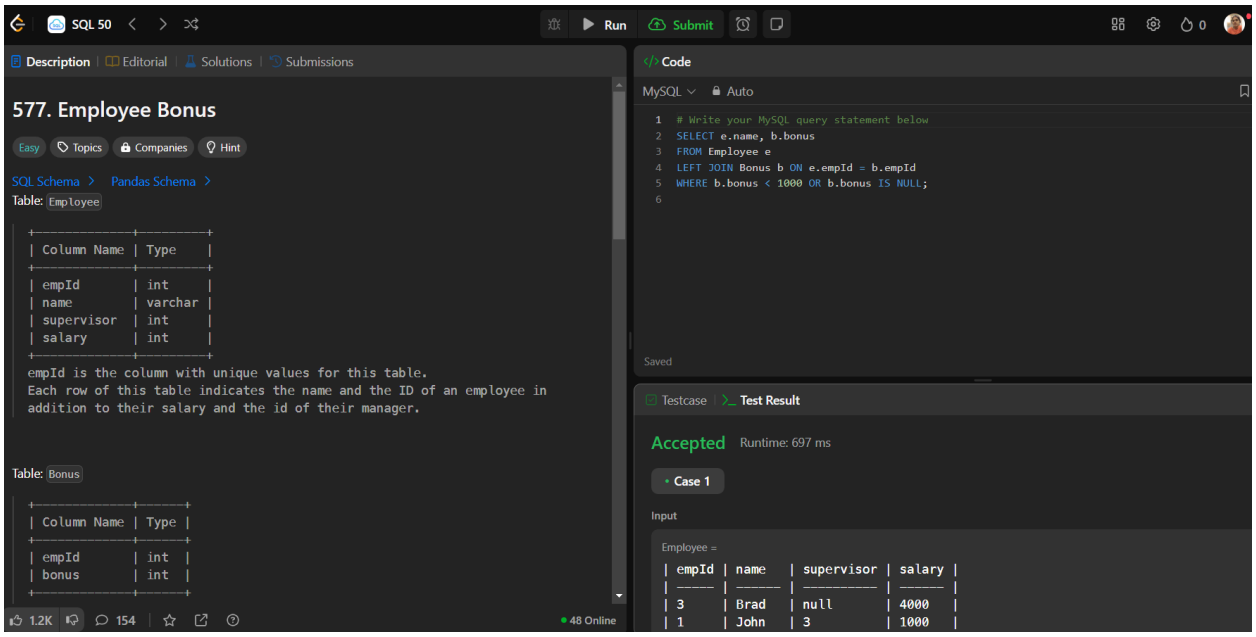


Solution:

```
SELECT
  machine_id,
  ROUND(SUM(
    CASE
      WHEN activity_type = 'end' THEN timestamp
      WHEN activity_type = 'start' THEN -timestamp
    END
  ) / COUNT(DISTINCT process_id), 3) AS processing_time
FROM
  Activity
GROUP BY
  machine_id;
```

Link of the question:
https://leetcode.com/problems/average-time-of-process-per-machine/?envType=study-plan-v2&envId=top-sql-50

2. Leetcode Question No. 2

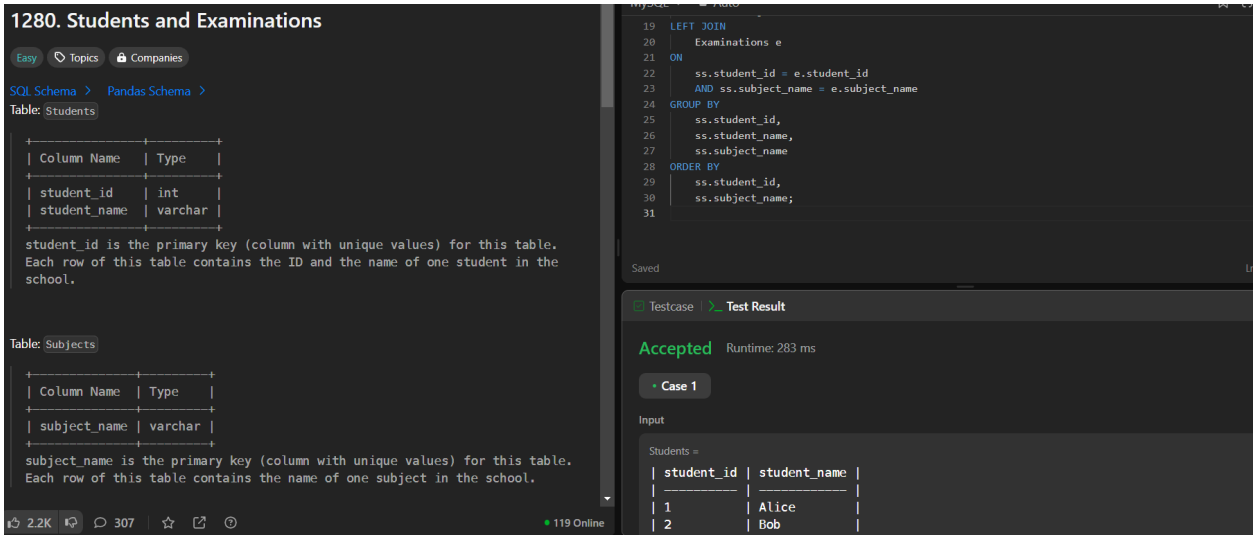


Solution:

```
SELECT e.name, b.bonus
FROM Employee e
LEFT JOIN Bonus b ON e.empId = b.empId
WHERE b.bonus < 1000 OR b.bonus IS NULL;
```

Link of the question:
https://leetcode.com/problems/employee-bonus/?envType=study-plan-v2&envId=top-sql-50

3. Leetcode Question No. 3



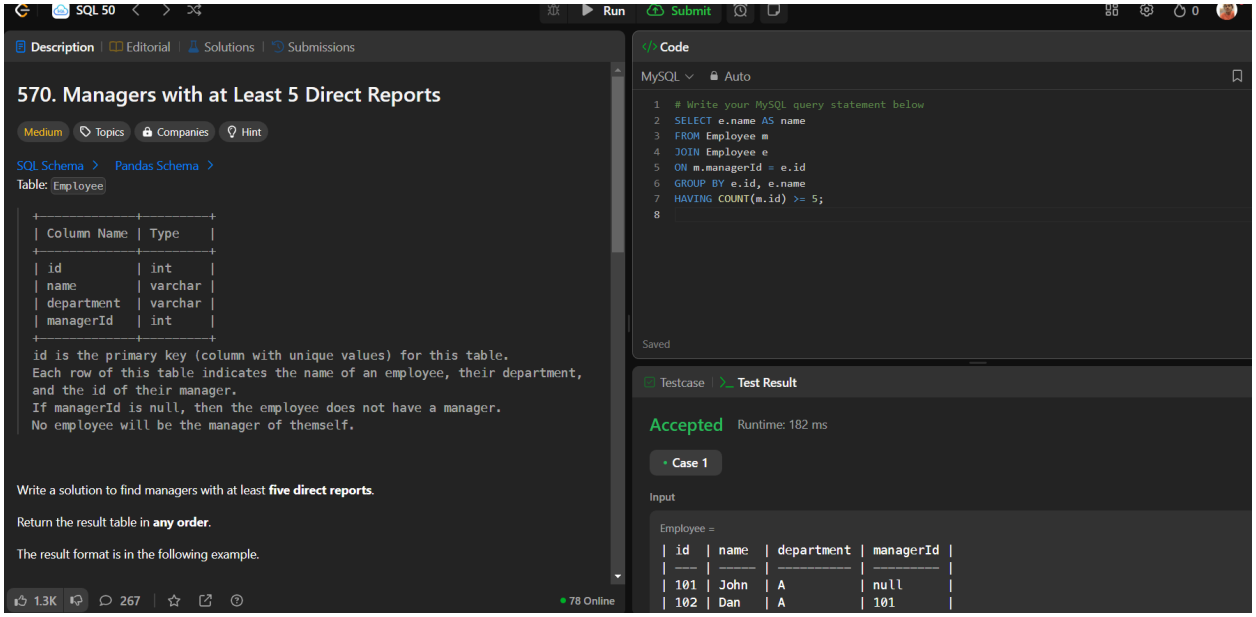
Solution:

```
# Write your MySQL query statement below
WITH StudentSubjects AS (
    SELECT
        s.student_id,
        s.student_name,
        sub.subject_name
    FROM
        Students s
    CROSS JOIN
        Subjects sub
)
SELECT
    ss.student_id,
    ss.student_name,
    ss.subject_name,
    COUNT(e.subject_name) AS attended_exams
FROM
    StudentSubjects ss
LEFT JOIN
    Examinations e
ON
    ss.student_id = e.student_id
    AND ss.subject_name = e.subject_name
GROUP BY
```

```
  ss.student_id,
  ss.student_name,
  ss.subject_name
ORDER BY
  ss.student_id,
  ss.subject_name;
```

Link of the question:
https://leetcode.com/problems/students-and-examinations/description/?envType=study-plan-v2&envId=top-sql-50

4. Leetcode Question No. 4

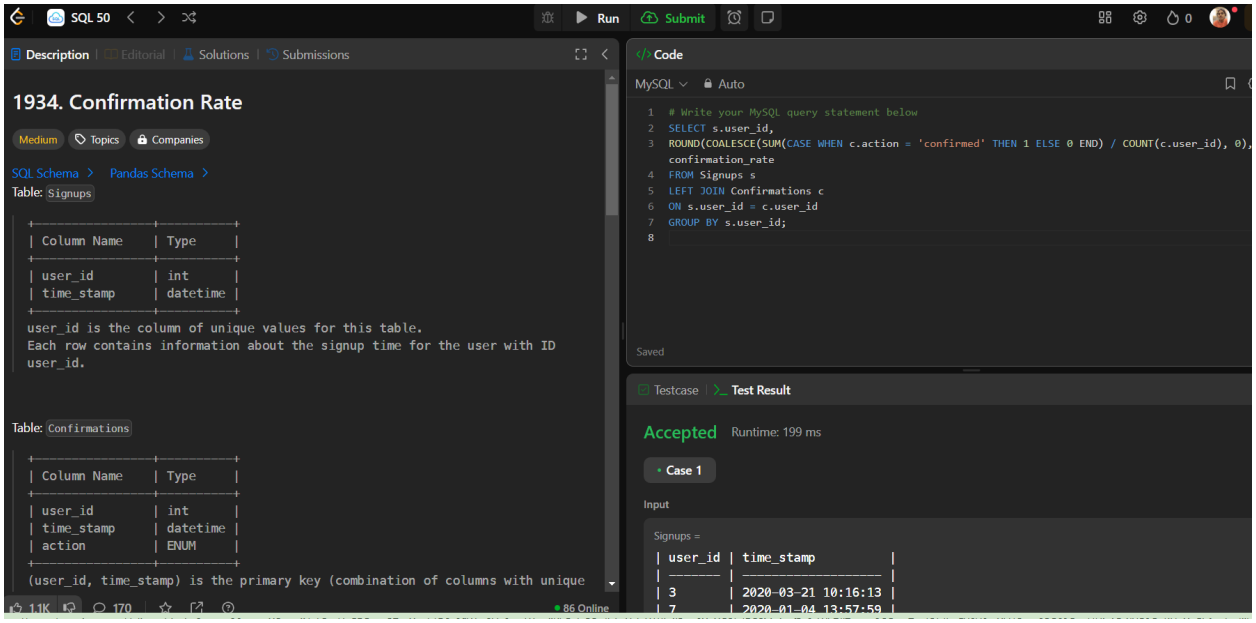


Solution:

```
SELECT e.name AS name
FROM Employee m
JOIN Employee e
ON m.managerId = e.id
GROUP BY e.id, e.name
HAVING COUNT(m.id) >= 5;
```

Link of the question:
https://leetcode.com/problems/managers-with-at-least-5-direct-reports/description/?envType=study-plan-v2&envId=top-sql-50

5. **Leetcode Question No. 5**



Solution:

```
SELECT s.user_id,
ROUND(COALESCE(SUM(CASE WHEN c.action = 'confirmed' THEN 1 ELSE 0 END) / COUNT(c.user_id), 0),
2) AS confirmation_rate
FROM Signups s
LEFT JOIN Confirmations c
ON s.user_id = c.user_id
GROUP BY s.user_id;
```

Link of the question:
https://leetcode.com/problems/confirmation-rate/?envType=study-plan-v2&envId=top-sql-50