November 6, 2020 52831 words 265 min read 21,725 views

LeetCode SQL Problem Solving Questions With Solutions

Related Topics: SQL SQL-Problem-Solving LeetCode

175. Combine Two Tables | Easy | LeetCode

Table: `Person`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
```

```
| PersonId    | int     |
| FirstName   | varchar |
| LastName    | varchar |
+-------------+---------+
PersonId is the primary key column for this table.
```

Table: `Address`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| AddressId   | int     |
| PersonId    | int     |
| City        | varchar |
| State       | varchar |
+-------------+---------+
AddressId is the primary key column for this table.
```

Write a SQL query for a report that provides the following information for each person in the Person table, regardless if there is an address for each of those people:

```
FirstName, LastName, City, State
```

Solution

sql

```sql
SELECT p.FirstName, p.LastName, a.City, a.State
FROM Person p
LEFT JOIN Address a
ON p.PersonId = a.PersonId;
```

176. Second Highest Salary | Easy | LeetCode

.ite a SQL query to get the second highest salary from the `Employee` table.

```
+----+--------+
| Id | Salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

For example, given the above Employee table, the query should return `200` as the second highest salary. If there is no second highest salary, then the query should return `null`.

```
+---------------------+
| SecondHighestSalary |
+---------------------+
| 200                 |
+---------------------+
```

Solution

sql

```sql
#Solution 1:
SELECT Max(Salary) SecondHighestSalary
FROM Employee WHERE Salary < (SELECT MAX(Salary) FROM Employee)

#Solution 2:
WITH CTE AS (SELECT DISTINCT Salary
FROM Employee
ORDER BY Salary DESC
LIMIT 2)

SELECT Salary as SecondHighestSalary
FROM CTE
ORDER BY Salary Asc
LIMIT 1;

#Solution 3:
WITH CTE AS
```

```
(
    SELECT Salary,
           DENSE_RANK() OVER (ORDER BY Salary DESC) AS DENSERANK
    FROM Employee
)
SELECT Salary SecondHighestSalary
FROM CTE
WHERE DENSERANK = 2;
```

177. Nth Highest Salary | Medium | LeetCode

Write a SQL query to get the nth highest salary from the Employee table.

```
+----+--------+
| Id | Salary |
+----+--------+
| 1  | 100    |
| 2  | 200    |
| 3  | 300    |
+----+--------+
```

For example, given the above Employee table, the nth highest salary where n = 2 is 200. If there is no nth highest salary, then the query should return null.

```
+------------------------+
| getNthHighestSalary(2) |
+------------------------+
| 200                    |
+------------------------+
```

Solution

sql

```
CREATE FUNCTION getNthHighestSalary(N INT) RETURNS INT
BEGIN
SET N = N-1;
```

```
    RETURN(
        SELECT DISTINCT Salary FROM Employee ORDER BY Salary DESC
        LIMIT 1 OFFSET N
    );
END
```

178. Rank Scores | Medium | LeetCode

Write a SQL query to rank scores. If there is a tie between two scores, both should have the same ranking. Note that after a tie, the next ranking number should be the next consecutive integer value. In other words, there should be no "holes" between ranks.

```
+----+-------+
| Id | Score |
+----+-------+
| 1  | 3.50  |
| 2  | 3.65  |
| 3  | 4.00  |
| 4  | 3.85  |
| 5  | 4.00  |
| 6  | 3.65  |
+----+-------+
```

For example, given the above `Scores` table, your query should generate the following report (order by highest score):

```
+-------+---------+
| score | Rank    |
+-------+---------+
| 4.00  | 1       |
| 4.00  | 1       |
| 3.85  | 2       |
| 3.65  | 3       |
| 3.65  | 3       |
| 3.50  | 4       |
+-------+---------+
```

Important Note: For MySQL solutions, to escape reserved words used as column names, you can use an apostrophe before and after the keyword. For example `Rank`.

Solution

```sql
SELECT score, DENSE_RANK() OVER (ORDER By Score DESC) AS "Rank"
FROM Scores;
```

180. Consecutive Numbers | Medium | LeetCode

Table: `Logs`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| num         | varchar |
+-------------+---------+
id is the primary key for this table.
```

Write an SQL query to find all numbers that appear at least three times consecutively.

Return the result table in any order.

The query result format is in the following example:

```
Logs table:
+----+-----+
| Id | Num |
+----+-----+
| 1  | 1   |
| 2  | 1   |
| 3  | 1   |
| 4  | 2   |
| 5  | 1   |
| 6  | 2   |
```

```
| 7  | 2   |
+----+-----+
```

```
Result table:
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
1 is the only number that appears consecutively for at least three times.
```

Solution

sql

```sql
SELECT a.Num as ConsecutiveNums
FROM Logs a
JOIN Logs b
ON a.id = b.id+1 AND a.num = b.num
JOIN Logs c
ON a.id = c.id+2 AND a.num = c.num;
```

181. Employees Earning More Than Their Managers | Easy | LeetCode

The `Employee` table holds all employees including their managers. Every employee has an Id, and there is also a column for the manager Id.

```
+----+-------+--------+-----------+
| Id | Name  | Salary | ManagerId |
+----+-------+--------+-----------+
| 1  | Joe   | 70000  | 3         |
| 2  | Henry | 80000  | 4         |
| 3  | Sam   | 60000  | NULL      |
| 4  | Max   | 90000  | NULL      |
+----+-------+--------+-----------+
```

Given the `Employee` table, write a SQL query that finds out employees who earn more than their managers. For the above table, Joe is the only employee who earns more than his manager.

```
+----------+
| Employee |
+----------+
| Joe      |
+----------+
```

Solution

```sql
SELECT E.Name as "Employee"
FROM Employee E
JOIN Employee M
ON E.ManagerId = M.Id
AND E.Salary > M.Salary;
```

182. Duplicate Emails | Easy | LeetCode

Write a SQL query to find all duplicate emails in a table named `Person`.

```
+----+---------+
| Id | Email   |
+----+---------+
| 1  | a@b.com |
| 2  | c@d.com |
| 3  | a@b.com |
+----+---------+
```

For example, your query should return the following for the above table:

```
+---------+
| Email   |
```

```
+---------+
| a@b.com |
+---------+
```

Note: All emails are in lowercase.

Solution

sql

```sql
#Solution- 1:
SELECT Email
FROM Person
GROUP BY Email
HAVING count(*) > 1

#Solution- 2:
WITH CTE AS(
SELECT Email, ROW_NUMBER() OVER(PARTITION BY Email ORDER BY Email) AS RN
    FROM Person
)

SELECT Email
FROM CTE
WHERE RN > 1;
```

183. Customers Who Never Order | Easy | LeetCode

Suppose that a website contains two tables, the `Customers` table and the `Orders` table. Write a SQL query to find all customers who never order anything.

Table: `Customers` .

```
+----+-------+
| Id | Name  |
+----+-------+
| 1  | Joe   |
| 2  | Henry |
| 3  | Sam   |
```

```
| 4  | Max   |
+----+-------+
```

Table: `Orders`.

```
+----+------------+
| Id | CustomerId |
+----+------------+
| 1  | 3          |
| 2  | 1          |
+----+------------+
```

Using the above tables as example, return the following:

```
+-----------+
| Customers |
+-----------+
| Henry     |
| Max       |
+-----------+
```

Solution

sql

```sql
#Solution- 1:
SELECT Name AS Customers
FROM Customers
LEFT JOIN Orders
ON Customers.Id = Orders.CustomerId
WHERE CustomerId IS NULL;

#Solution- 2:
SELECT Name as Customers
FROM Customers
WHERE Id NOT IN(
    SELECT CustomerId
    FROM Orders
```

```
)
```

184. Department Highest Salary | Medium | LeetCode

The `Employee` table holds all employees. Every employee has an Id, a salary, and there is also a column for the department Id.

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 70000  | 1            |
| 2  | Jim   | 90000  | 1            |
| 3  | Henry | 80000  | 2            |
| 4  | Sam   | 60000  | 2            |
| 5  | Max   | 90000  | 1            |
+----+-------+--------+--------------+
```

The `Department` table holds all departments of the company.

```
+----+----------+
| Id | Name     |
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

Write a SQL query to find employees who have the highest salary in each of the departments. For the above tables, your SQL query should return the following rows (order of rows does not matter).

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| IT         | Jim      | 90000  |
| Sales      | Henry    | 80000  |
```

```
+------------+----------+--------+
```

Explanation:

Max and Jim both have the highest salary in the IT department and Henry has the highest salary in the Sales department.

Solution

sql

```sql
SELECT Department.Name AS Department, Employee.Name AS Employee, Salary
FROM Employee
JOIN Department
ON Employee.DepartmentId = Department.Id
WHERE (DepartmentId, Salary) IN(
        SELECT  DepartmentId, MAX(Salary) AS Salary
        FROM Employee
        GROUP BY DepartmentId
        );
```

185. Department Top Three Salaries | Hard | LeetCode

The `Employee` table holds all employees. Every employee has an Id, and there is also a column for the department Id.

```
+----+-------+--------+--------------+
| Id | Name` | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 85000  | 1            |
| 2  | Henry | 80000  | 2            |
| 3  | Sam   | 60000  | 2            |
| 4  | Max   | 90000  | 1            |
| 5  | Janet | 69000  | 1            |
| 6  | Randy | 85000  | 1            |
| 7  | Will  | 70000  | 1            |
+----+-------+--------+--------------+
```

The `Department` table holds all departments of the company.

```
+----+----------+
| Id | Name     |
+----+----------+
| 1  | IT       |
| 2  | Sales    |
+----+----------+
```

Write a SQL query to find employees who earn the top three salaries in each of the department. For the above tables, your SQL query should return the following rows (order of rows does not matter).

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
| IT         | Randy    | 85000  |
| IT         | Joe      | 85000  |
| IT         | Will     | 70000  |
| Sales      | Henry    | 80000  |
| Sales      | Sam      | 60000  |
+------------+----------+--------+
```

Explanation:
In IT department, Max earns the highest salary, both Randy and Joe earn the second highest salary, and Will earns the third highest salary. There are only two employees in the Sales department, Henry earns the highest salary while Sam earns the second highest salary.

Solution

sql

```sql
WITH department_ranking AS (
SELECT Name AS Employee, Salary ,DepartmentId
  ,DENSE_RANK() OVER (PARTITION BY DepartmentId ORDER BY Salary DESC) AS rnk
FROM Employee
```

```
)

SELECT d.Name AS Department, r.Employee, r.Salary
FROM department_ranking AS r
JOIN Department AS d
ON r.DepartmentId = d.Id
WHERE r.rnk <= 3
ORDER BY d.Name ASC, r.Salary DESC;
```

196. Delete Duplicate Emails | Easy | LeetCode

Write a SQL query to delete all duplicate email entries in a table named `Person`, keeping only unique emails based on its smallest Id.

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
| 3  | john@example.com |
+----+------------------+
```

Id is the primary key column for this table. For example, after running your query, the above `Person` table should have the following rows:

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
+----+------------------+
```

Note:
Your output is the whole `Person` table after executing your sql. Use `delete` statement.

Solution

```sql
DELETE p2
FROM Person p1
JOIN Person p2
ON p1.Email = p2.Email
AND p1.id < p2.id
```

197. Rising Temperature | Easy | LeetCode

Table: `Weather`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| recordDate    | date    |
| temperature   | int     |
+---------------+---------+

id is the primary key for this table.
This table contains information about the temperature in a certain day.
```

Write an SQL query to find all dates' `id` with higher temperature compared to its previous dates (yesterday).

Return the result table in any order.

The query result format is in the following example:

```
Weather
+----+------------+-------------+
| id | recordDate | Temperature |
+----+------------+-------------+
| 1  | 2015-01-01 | 10          |
| 2  | 2015-01-02 | 25          |
| 3  | 2015-01-03 | 20          |
| 4  | 2015-01-04 | 30          |
+----+------------+-------------+
```

```
Result table:
+----+
| id |
+----+
| 2  |
| 4  |
+----+

In 2015-01-02, temperature was higher than the previous day (10 -> 25).
In 2015-01-04, temperature was higher than the previous day (20 -> 30).
```

Solution

sql

```sql
#Solution- 1:
SELECT t.Id
FROM Weather AS t, Weather AS y
WHERE DATEDIFF(t.RecordDate, y.RecordDate) = 1
AND t.Temperature > y.Temperature;

#Solution- 2:
SELECT t.Id
FROM Weather t
JOIN Weather y
ON DATEDIFF(t.recordDate, y.recordDate) = 1 AND
t.temperature > y.temperature;
```

262. Trips and Users | Hard | LeetCode

Table: **Trips**

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| Id          | int      |
| Client_Id   | int      |
| Driver_Id   | int      |
```

```
| City_Id     | int      |
| Status      | enum     |
| Request_at  | date     |
+-------------+----------+
Id is the primary key for this table.
The table holds all taxi trips. Each trip has a unique Id, while Client_Id ar
Status is an ENUM type of (‘completed’, ‘cancelled_by_driver’, ‘cancelled_by_
```

Table: **Users**

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| Users_Id    | int      |
| Banned      | enum     |
| Role        | enum     |
+-------------+----------+
Users_Id is the primary key for this table.
The table holds all users. Each user has a unique Users_Id, and Role is an EN
Status is an ENUM type of (‘Yes’, ‘No’).
```

Write a SQL query to find the cancellation rate of requests with unbanned users (both client and driver must not be banned) each day between `"2013-10-01"` and `"2013-10-03"`.

The cancellation rate is computed by dividing the number of canceled (by client or driver) requests with unbanned users by the total number of requests with unbanned users on that day.

Return the result table in any order. Round `Cancellation Rate` to two decimal points.

The query result format is in the following example:

```
Trips table:
+----+-----------+-----------+---------+---------------------+------------+
| Id | Client_Id | Driver_Id | City_Id | Status              | Request_at |
+----+-----------+-----------+---------+---------------------+------------+
| 1  | 1         | 10        | 1       | completed           | 2013-10-01 |
| 2  | 2         | 11        | 1       | cancelled_by_driver | 2013-10-01 |
```

3	3	12	6	completed	2013-10-01
4	4	13	6	cancelled_by_client	2013-10-01
5	1	10	1	completed	2013-10-02
6	2	11	6	completed	2013-10-02
7	3	12	6	completed	2013-10-02
8	2	12	12	completed	2013-10-03
9	3	10	12	completed	2013-10-03
10	4	13	12	cancelled_by_driver	2013-10-03
+----+-----------+-----------+---------+---------------------+------------+

Users table:
+----------+--------+--------+
| Users_Id | Banned | Role |
+----------+--------+--------+
1	No	client
2	Yes	client
3	No	client
4	No	client
10	No	driver
11	No	driver
12	No	driver
13	No	driver
+----------+--------+--------+

Result table:
+------------+-------------------+
| Day | Cancellation Rate |
+------------+-------------------+
2013-10-01	0.33
2013-10-02	0.00
2013-10-03	0.50
+------------+-------------------+

On 2013-10-01:

- There were 4 requests in total, 2 of which were canceled.
- However, the request with Id=2 was made by a banned client (User_Id=2), so
- Hence there are 3 unbanned requests in total, 1 of which was canceled.
- The Cancellation Rate is (1 / 3) = 0.33
  On 2013-10-02:
- There were 3 requests in total, 0 of which were canceled.
- The request with Id=6 was made by a banned client, so it is ignored.

```
- Hence there are 2 unbanned requests in total, 0 of which were canceled.
- The Cancellation Rate is (0 / 2) = 0.00
  On 2013-10-03:
- There were 3 requests in total, 1 of which was canceled.
- The request with Id=8 was made by a banned client, so it is ignored.
- Hence there are 2 unbanned request in total, 1 of which were canceled.
- The Cancellation Rate is (1 / 2) = 0.50
```

Solution

sql

```sql
SELECT Request_at AS Day,
ROUND(SUM(IF(Status<>"completed", 1, 0))/COUNT(Status),2) AS "Cancellation Ra
FROM Trips
WHERE Request_at BETWEEN "2013-10-01" AND "2013-10-03"
AND Client_Id NOT IN (SELECT Users_Id FROM Users WHERE Banned = 'Yes')
AND Driver_Id NOT IN (SELECT Users_Id FROM Users WHERE Banned = 'Yes')
GROUP BY Request_at;
```

511. Game Play Analysis I | Easy | LeetCode

Table: **Activity**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| player_id    | int     |
| device_id    | int     |
| event_date   | date    |
| games_played | int     |
+--------------+---------+
(player_id, event_date) is the primary key of this table.
This table shows the activity of players of some game.
Each row is a record of a player who logged in and played a number of games
```

..rite an SQL query that reports the **first login date** for each player.

The query result format is in the following example:

```
Activity table:
+-----------+-----------+------------+--------------+
| player_id | device_id | event_date | games_played |
+-----------+-----------+------------+--------------+
| 1         | 2         | 2016-03-01 | 5            |
| 1         | 2         | 2016-05-02 | 6            |
| 2         | 3         | 2017-06-25 | 1            |
| 3         | 1         | 2016-03-02 | 0            |
| 3         | 4         | 2018-07-03 | 5            |
+-----------+-----------+------------+--------------+

Result table:
+-----------+-------------+
| player_id | first_login |
+-----------+-------------+
| 1         | 2016-03-01  |
| 2         | 2017-06-25  |
| 3         | 2016-03-02  |
+-----------+-------------+
```

Solution

sql

```sql
SELECT player_id, MIN(event_date) as first_login
FROM Activity
GROUP BY player_id
```

512. Game Play Analysis II | Easy | LeetCode

Table: **Activity**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| player_id    | int     |
```

```
| device_id    | int     |
| event_date   | date    |
| games_played | int     |
+--------------+---------+
(player_id, event_date) is the primary key of this table.
This table shows the activity of players of some game.
Each row is a record of a player who logged in and played a number of games (
```

Write a SQL query that reports the device that is first logged in for each player.

The query result format is in the following example:

```
Activity table:
+-----------+-----------+------------+--------------+
| player_id | device_id | event_date | games_played |
+-----------+-----------+------------+--------------+
| 1         | 2         | 2016-03-01 | 5            |
| 1         | 2         | 2016-05-02 | 6            |
| 2         | 3         | 2017-06-25 | 1            |
| 3         | 1         | 2016-03-02 | 0            |
| 3         | 4         | 2018-07-03 | 5            |
+-----------+-----------+------------+--------------+

Result table:
+-----------+-----------+
| player_id | device_id |
+-----------+-----------+
| 1         | 2         |
| 2         | 3         |
| 3         | 1         |
+-----------+-----------+
```

Solution

sql

```sql
#Solution- 1:
SELECT DISTINCT player_id, device_id
FROM Activity
WHERE (player_id, event_date) in (
```

```
    SELECT player_id, min(event_date)
    FROM Activity
    GROUP BY player_id)
```

```
#Solution- 2:
SELECT a.player_id, b.device_id
FROM
(SELECT player_id, MIN(event_date) AS event_date FROM Activity
GROUP BY player_id) a
JOIN Activity b
ON a.player_id = b.player_id AND a.event_date = b.event_date;

#Solution- 3:
SELECT player_id, device_id
FROM
(SELECT player_id, device_id, event_date,
ROW_NUMBER() OVER (PARTITION BY player_id ORDER BY event_date) AS r
FROM Activity) lookup
WHERE r = 1;
```

534. Game Play Analysis III | Medium | LeetCode

Table: **Activity**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| player_id    | int     |
| device_id    | int     |
| event_date   | date    |
| games_played | int     |
+--------------+---------+
(player_id, event_date) is the primary key of this table.
This table shows the activity of players of some game.
Each row is a record of a player who logged in and played a number of games (
```

ite an SQL query that reports for each player and date, how many games played so far by the player. That is, the total number of games played by the player until that date. Check the example for clarity.

The query result format is in the following example:

```
Activity table:
+-----------+-----------+------------+--------------+
| player_id | device_id | event_date | games_played |
+-----------+-----------+------------+--------------+
| 1         | 2         | 2016-03-01 | 5            |
| 1         | 2         | 2016-05-02 | 6            |
| 1         | 3         | 2017-06-25 | 1            |
| 3         | 1         | 2016-03-02 | 0            |
| 3         | 4         | 2018-07-03 | 5            |
+-----------+-----------+------------+--------------+

Result table:
+-----------+------------+---------------------+
| player_id | event_date | games_played_so_far |
+-----------+------------+---------------------+
| 1         | 2016-03-01 | 5                   |
| 1         | 2016-05-02 | 11                  |
| 1         | 2017-06-25 | 12                  |
| 3         | 2016-03-02 | 0                   |
| 3         | 2018-07-03 | 5                   |
+-----------+------------+---------------------+
For the player with id 1, 5 + 6 = 11 games played by 2016-05-02, and 5 + 6 +
For the player with id 3, 0 + 5 = 5 games played by 2018-07-03.
Note that for each player we only care about the days when the player logged
```

Solution

sql

```sql
#Solution- 1:
SELECT t1.player_id, t1.event_date, SUM(t2.games_played) as games_played_so_
FROM Activity t1
JOIN Activity t2
ON t1.player_id = t2.player_id
WHERE t1.event_date >= t2.event_date
GROUP BY t1.player_id, t1.event_date;

#Solution- 2:
SELECT player_id, event_date,
```

```
    SUM(games_played) OVER (PARTITION BY player_id ORDER BY event_date) AS games_
    FROM Activity;
```

550. Game Play Analysis IV | Medium | LeetCode

Table: Activity

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| player_id    | int     |
| device_id    | int     |
| event_date   | date    |
| games_played | int     |
+--------------+---------+
(player_id, event_date) is the primary key of this table.
This table shows the activity of players of some game.
Each row is a record of a player who logged in and played a number of games (
```

Write an SQL query that reports the fraction of players that logged in again on the day after the day they first logged in, rounded to 2 decimal places. In other words, you need to count the number of players that logged in for at least two consecutive days starting from their first login date, then divide that number by the total number of players.

The query result format is in the following example:

```
Activity table:
+-----------+-----------+------------+--------------+
| player_id | device_id | event_date | games_played |
+-----------+-----------+------------+--------------+
| 1         | 2         | 2016-03-01 | 5            |
| 1         | 2         | 2016-03-02 | 6            |
| 2         | 3         | 2017-06-25 | 1            |
| 3         | 1         | 2016-03-02 | 0            |
| 3         | 4         | 2018-07-03 | 5            |
+-----------+-----------+------------+--------------+

Result table:
```

```
+-----------+
| fraction  |
+-----------+
| 0.33      |
+-----------+
Only the player with id 1 logged back in after the first day he had logged in
```

Solution

sql

```sql
#Solution- 1:
SELECT ROUND(sum(CASE WHEN t1.event_date = t2.first_event+1 THEN 1 ELSE 0 END
FROM Activity t1
JOIN
    (SELECT player_id, MIN(event_date) AS first_event
    FROM Activity
    GROUP BY player_id) t2
    ON t1.player_id = t2.player_id;

#Solution- 2:
SELECT ROUND(COUNT(DISTINCT b.player_id)/COUNT(DISTINCT a.player_id),2) AS fr
FROM
  (SELECT player_id, MIN(event_date) AS event_date FROM Activity
  GROUP BY player_id) a
  LEFT JOIN Activity b
  ON a.player_id = b.player_id AND a.event_date+1 = b.event_date;
```

569. Median Employee Salary | Hard | LeetCode

The `Employee` table holds all employees. The employee table has three columns: Employee Id, Company Name, and Salary.

```
+-----+------------+--------+
|Id   | Company    | Salary |
+-----+------------+--------+
|1    | A          | 2341   |
|2    | A          | 341    |
|3    | A          | 15     |
```

```
|4    | A           | 15314  |
|5    | A           | 451    |
|6    | A           | 513    |
|7    | B           | 15     |
|8    | B           | 13     |
|9    | B           | 1154   |
|10   | B           | 1345   |
|11   | B           | 1221   |
|12   | B           | 234    |
|13   | C           | 2345   |
|14   | C           | 2645   |
|15   | C           | 2645   |
|16   | C           | 2652   |
|17   | C           | 65     |
+-----+-------------+--------+
```

Write a SQL query to find the median salary of each company. Bonus points if you can solve it without using any built-in SQL functions.

```
+-----+-------------+--------+
|Id   | Company     | Salary |
+-----+-------------+--------+
|5    | A           | 451    |
|6    | A           | 513    |
|12   | B           | 234    |
|9    | B           | 1154   |
|14   | C           | 2645   |
+-----+-------------+--------+
```

Solution

sql

```sql
SELECT t1.Id AS Id, t1.Company, t1.Salary
FROM Employee AS t1 JOIN Employee AS t2
ON t1.Company = t2.Company
GROUP BY t1.Id
HAVING abs(sum(CASE WHEN t2.Salary<t1.Salary THEN 1
                    WHEN t2.Salary>t1.Salary THEN -1
                    WHEN t2.Salary=t1.Salary AND t2.Id<t1.Id THEN 1
```

```
                    WHEN t2.Salary=t1.Salary AND t2.Id>t1.Id THEN -1
                    ELSE 0 END)) <= 1
ORDER BY t1.Company, t1.Salary, t1.Id
```

570. Managers with at Least 5 Direct Reports | Medium | LeetCode

The `Employee` table holds all employees including their managers. Every employee has an Id, and there is also a column for the manager Id.

```
+------+----------+-----------+----------+
|Id    |Name      |Department |ManagerId |
+------+----------+-----------+----------+
|101   |John      |A          |null      |
|102   |Dan       |A          |101       |
|103   |James     |A          |101       |
|104   |Amy       |A          |101       |
|105   |Anne      |A          |101       |
|106   |Ron       |B          |101       |
+------+----------+-----------+----------+
```

Given the `Employee` table, write a SQL query that finds out managers with at least 5 direct report. For the above table, your SQL query should return:

```
+-------+
| Name  |
+-------+
| John  |
+-------+
```

Note: No one would report to himself.

Solution

sql

```
SELECT Name
```

```
FROM Employee
WHERE id IN
   (SELECT ManagerId
    FROM Employee
    GROUP BY ManagerId
    HAVING COUNT(DISTINCT Id) >= 5)
```

571. Find Median Given Frequency of Numbers | LeetCode

The `Numbers` table keeps the value of number and its frequency.

```
+----------+-------------+
|  Number  |  Frequency  |
+----------+-------------|
|  0       |  7          |
|  1       |  1          |
|  2       |  3          |
|  3       |  1          |
+----------+-------------+
```

In this table, the numbers are 0, 0, 0, 0, 0, 0, 0, 1, 2, 2, 2, 3, so the median is (0 + 0) / 2 = 0.

```
+--------+
| median |
+--------|
| 0.0000 |
+--------+
```

Write a query to find the median of all numbers and name the result as median.

Solution

sql

```
SELECT avg(t3.Number) as median
FROM Numbers as t3
```

```
JOIN
    (SELECT t1.Number,
        abs(SUM(CASE WHEN t1.Number>t2.Number THEN t2.Frequency ELSE 0 END) -
            SUM(CASE WHEN t1.Number<t2.Number THEN t2.Frequency ELSE 0 END))
    FROM numbers AS t1, numbers AS t2
    GROUP BY t1.Number) AS t4
ON t3.Number = t4.Number
WHERE t3.Frequency>=t4.count_diff
```

574. Winning Candidate | Medium | LeetCode

Table: **Candidate**

```
+-----+---------+
| id  | Name    |
+-----+---------+
| 1   | A       |
| 2   | B       |
| 3   | C       |
| 4   | D       |
| 5   | E       |
+-----+---------+
```

Table: **Vote**

```
+-----+--------------+
| id  | CandidateId  |
+-----+--------------+
| 1   |     2        |
| 2   |     4        |
| 3   |     3        |
| 4   |     2        |
| 5   |     5        |
+-----+--------------+
```

 is the auto-increment primary key, **CandidateId** is the id appeared in Candidate table. Write a sql to find the name of the winning candidate, the above example will

return the winner B.

```
+------+
| Name |
+------+
| B    |
+------+
```

Notes: You may assume there is no tie, in other words there will be at most one winning candidate.

Solution

sql

```sql
SELECT Name
FROM Candidate
WHERE id = (SELECT CandidateId
     FROM Vote
     GROUP BY CandidateId
     ORDER BY COUNT(1) desc
     LIMIT 1)

## Assumption: if we have two candidates with the same votes, we choose the
# SELECT Name
# FROM Candidate JOIN
#     (SELECT CandidateId
#     FROM Vote
#     GROUP BY CandidateId
#     ORDER BY count(1) DESC
#     LIMIT 1) AS t
# ON Candidate.id = t.CandidateId
```

577. Employee Bonus | Easy | LeetCode

lect all employee's name and bonus whose bonus is < 1000.

Table:Employee

```
+-------+--------+-----------+--------+
| empId |  name  | supervisor| salary |
+-------+--------+-----------+--------+
|   1   | John   |  3        | 1000   |
|   2   | Dan    |  3        | 2000   |
|   3   | Brad   |  null     | 4000   |
|   4   | Thomas |  3        | 4000   |
+-------+--------+-----------+--------+
empId is the primary key column for this table.
```

Table: Bonus

```
+-------+-------+
| empId | bonus |
+-------+-------+
| 2     | 500   |
| 4     | 2000  |
+-------+-------+
empId is the primary key column for this table.
```

Example ouput:

```
+-------+-------+
| name  | bonus |
+-------+-------+
| John  | null  |
| Dan   | 500   |
| Brad  | null  |
+-------+-------+
```

Solution

sql

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

578. Get Highest Answer Rate Question | Medium | LeetCode

Get the highest answer rate question from a table survey_log with these columns: uid, action, question_id, answer_id, q_num, timestamp.

uid means user id; action has these kind of values: “show”, “answer”, “skip”; answer_id is not null when action column is “answer”, while is null for “show” and “skip”; q_num is the numeral order of the question in current session.

Write a sql query to identify the question which has the highest answer rate.

Example: Input:

```
+------+-----------+--------------+------------+-----------+------------+
| uid  | action    | question_id  | answer_id  | q_num     | timestamp  |
+------+-----------+--------------+------------+-----------+------------+
| 5    | show      | 285          | null       | 1         | 123        |
| 5    | answer    | 285          | 124124     | 1         | 124        |
| 5    | show      | 369          | null       | 2         | 125        |
| 5    | skip      | 369          | null       | 2         | 126        |
+------+-----------+--------------+------------+-----------+------------+
```

Output:

```
+-------------+
| survey_log  |
+-------------+
|    285      |
+-------------+
```

Explanation: question 285 has answer rate 1/1, while question 369 has 0/1 answer rate, so output 285.

Note: The highest answer rate meaning is: answer number’s ratio in show number in the same question.

Solution

sql

```sql
#Solution- 1::
SELECT question_id AS survey_log FROM
(SELECT question_id,
        SUM(IF(action='show', 1, 0)) AS num_show,
        SUM(IF(action='answer', 1, 0)) AS num_answer
 FROM survey_log GROUP BY question_id) AS t
ORDER BY (num_answer/num_show) DESC LIMIT 1;

#Solution- 2:
SELECT question_id AS survey_log
FROM (SELECT question_id,
      sum(CASE WHEN  action='show' THEN 1 ELSE 0 END) AS show_count,
      sum(CASE WHEN  action='answer' THEN 1 ELSE 0 END) AS answer_count
    FROM survey_log
    GROUP BY question_id) AS t
ORDER BY answer_count/show_count DESC LIMIT 1;
```

579. Find Cumulative Salary of an Employee | Hard | LeetCode

The `Employee` table holds the salary information in a year.

Write a SQL to get the cumulative sum of an employee's salary over a period of 3 months but exclude the most recent month.

The result should be displayed by 'Id' ascending, and then by 'Month' descending.

Example Input

```
| Id | Month | Salary |
|----|-------|--------|
| 1  | 1     | 20     |
| 2  | 1     | 20     |
| 1  | 2     | 30     |
| 2  | 2     | 30     |
```

```
| 3  | 2    | 40    |
| 1  | 3    | 40    |
| 3  | 3    | 60    |
| 1  | 4    | 60    |
| 3  | 4    | 70    |
```

Output

```
| Id | Month | Salary |
|----|-------|--------|
| 1  | 3     | 90     |
| 1  | 2     | 50     |
| 1  | 1     | 20     |
| 2  | 1     | 20     |
| 3  | 3     | 100    |
| 3  | 2     | 40     |
```

Explanation Employee ‘1’ has 3 salary records for the following 3 months except the most recent month ‘4’: salary 40 for month ‘3’, 30 for month ‘2’ and 20 for month ‘1’ So the cumulative sum of salary of this employee over 3 months is 90(40+30+20), 50(30+20) and 20 respectively.

```
| Id | Month | Salary |
|----|-------|--------|
| 1  | 3     | 90     |
| 1  | 2     | 50     |
| 1  | 1     | 20     |
```

Employee ‘2’ only has one salary record (month ‘1’) except its most recent month ‘2’.

```
| Id | Month | Salary |
|----|-------|--------|
| 2  | 1     | 20     |
```

ploy ‘3’ has two salary records except its most recent pay month ‘4’: month ‘3’ with 60 and month ‘2’ with 40. So the cumulative salary is as following.

```
| Id | Month | Salary |
|----|-------|--------|
| 3  | 3     | 100    |
| 3  | 2     | 40     |
```

Solution

```sql
SELECT
    a.id,
    a.month,
    SUM(b.salary) Salary
FROM
    Employee a JOIN Employee b ON
    a.id = b.id AND
    a.month - b.month >= 0 AND
    a.month - b.month < 3
GROUP BY
    a.id, a.month
HAVING
    (a.id, a.month) NOT IN (SELECT id, MAX(month) FROM Employee GROUP BY id)
ORDER BY
    a.id, a.month DESC
```

580. Count Student Number in Departments | Medium | LeetCode

A university uses 2 data tables, `student` and `department`, to store data about its students and the departments associated with each major.

Write a query to print the respective department name and number of students majoring in each department for all departments in the department table (even ones with no current students).

Sort your results by descending number of students; if two or more departments have the same number of students, then sort those departments alphabetically by department name.

The `student` is described as follow:

```
| Column Name  | Type      |
|--------------|-----------|
| student_id   | Integer   |
| student_name | String    |
| gender       | Character |
| dept_id      | Integer   |
```

where student_id is the student's ID number, student_name is the student's name, gender is their gender, and dept_id is the department ID associated with their declared major.

And the department table is described as below:

```
| Column Name | Type    |
|-------------|---------|
| dept_id     | Integer |
| dept_name   | String  |
```

where dept_id is the department's ID number and dept_name is the department name.

Here is an example input: `student` table:

```
| student_id | student_name | gender | dept_id |
|------------|--------------|--------|---------|
| 1          | Jack         | M      | 1       |
| 2          | Jane         | F      | 1       |
| 3          | Mark         | M      | 2       |
```

`department` table:

```
| dept_id | dept_name   |
|---------|-------------|
| 1       | Engineering |
| 2       | Science     |
```

```
| 3       | Law         |
```

The Output should be:

```
| dept_name   | student_number |
|-------------|----------------|
| Engineering | 2              |
| Science     | 1              |
| Law         | 0              |
```

Solution

sql

```sql
SELECT dept_name,
  SUM(CASE WHEN student_id IS NULL THEN 0 ELSE 1 END) AS student_number
FROM department
LEFT JOIN student
ON department.dept_id = student.dept_id
GROUP BY department.dept_id
ORDER BY student_number DESC, dept_name
```

584. Find Customer Referee | Easy | LeetCode

Given a table `customer` holding customers information and the referee.

```
+------+------+-----------+
| id   | name | referee_id|
+------+------+-----------+
|    1 | Will |      NULL |
|    2 | Jane |      NULL |
|    3 | Alex |         2 |
|    4 | Bill |      NULL |
|    5 | Zack |         1 |
|    6 | Mark |         2 |
+------+------+-----------+
```

Write a query to return the list of customers NOT referred by the person with id '2'.

For the sample data above, the result is:

```
+------+
| name |
+------+
| Will |
| Jane |
| Bill |
| Zack |
+------+
```

Solution

```sql
SELECT name
FROM customer
WHERE referee_id != '2' OR referee_id IS NULL;
```

585. Investments in 2016 | Medium | LeetCode

Write a query to print the sum of all total investment values in 2016 (TIV_2016), to a scale of 2 decimal places, for all policy holders who meet the following criteria:

1. Have the same TIV_2015 value as one or more other policyholders.
2. Are not located in the same city as any other policyholder (i.e.: the (latitude, longitude) attribute pairs must be unique). Input Format: The insurance table is described as follows:

```
| Column Name | Type          |
|-------------|---------------|
| PID         | INTEGER(11)   |
| TIV_2015    | NUMERIC(15,2) |
| TIV_2016    | NUMERIC(15,2) |
| LAT         | NUMERIC(5,2)  |
```

```
| LON         | NUMERIC(5,2)  |
```

where PID is the policyholder's policy ID, TIV_2015 is the total investment value in 2015, TIV_2016 is the total investment value in 2016, LAT is the latitude of the policy holder's city, and LON is the longitude of the policy holder's city.

Sample Input

```
| PID | TIV_2015 | TIV_2016 | LAT | LON |
|-----|----------|----------|-----|-----|
| 1   | 10       | 5        | 10  | 10  |
| 2   | 20       | 20       | 20  | 20  |
| 3   | 10       | 30       | 20  | 20  |
| 4   | 10       | 40       | 40  | 40  |
```

Sample Output

```
| TIV_2016 |
|----------|
| 45.00    |
```

Explanation

```
The first record in the table, like the last record, meets both of the two cr
The TIV_2015 value '10' is as the same as the third and forth record, and its

The second record does not meet any of the two criteria. Its TIV_2015 is not

And its location is the same with the third record, which makes the third rec

So, the result is the sum of TIV_2016 of the first and last record, which is
```

Solution

sql

```sql
SELECT SUM(TIV_2016) AS TIV_2016
```

```
FROM insurance
WHERE CONCAT(LAT, ',', LON)
    IN (SELECT CONCAT(LAT, ',', LON)
       FROM insurance
       GROUP BY LAT, LON
       HAVING COUNT(1) = 1)
AND TIV_2015 in
    (SELECT TIV_2015
    FROM insurance
    GROUP BY TIV_2015
    HAVING COUNT(1)>1)
```

586. Customer Placing the Largest Number of Orders | Easy | LeetCode

Query the customer_number from the orders table for the customer who has placed the largest number of orders.

It is guaranteed that exactly one customer will have placed more orders than any other customer.

The orders table is defined as follows:

```
| Column            | Type      |
|-------------------|-----------|
| order_number (PK) | int       |
| customer_number   | int       |
| order_date        | date      |
| required_date     | date      |
| shipped_date      | date      |
| status            | char(15)  |
| comment           | char(200) |
```

Sample Input

```
| order_number | customer_number | order_date | required_date | shipped_date
|--------------|-----------------|------------|---------------|-------------
| 1            | 1               | 2017-04-09 | 2017-04-13    | 2017-04-12
```

```
| 2           | 2             | 2017-04-15 | 2017-04-20    | 2017-04-18
| 3           | 3             | 2017-04-16 | 2017-04-25    | 2017-04-20
| 4           | 3             | 2017-04-18 | 2017-04-28    | 2017-04-25
```

Sample Output

```
| customer_number |
|-----------------|
| 3               |
```

Explanation

```
The customer with number '3' has two orders, which is greater than either cus
So the result is customer_number '3'.
```

Solution

sql

```sql
# assume: only one match
SELECT customer_number FROM orders
GROUP BY customer_number
ORDER BY COUNT(1) DESC
LIMIT 1

## assume: multiple matches
##  1 1
##  2 1
##  3 1
##
##  1 1 1 1
##  1 1 2 1
##  1 1 3 1
##
##  SELECT t1.customer_number
##  FROM (SELECT customer_number, COUNT(1) AS count
##        FROM orders GROUP BY customer_number) AS t1,
##        (SELECT customer_number, COUNT(1) AS count
##        FROM orders GROUP BY customer_number) AS t2
```

```
##  GROUP BY t1.customer_number
##  HAVING max(t1.count) = max(t2.count)
```

595. Big Countries | Easy | LeetCode

There is a table `World`

```
+-----------------+------------+------------+--------------+---------------+
| name            | continent  | area       | population   | gdp           |
+-----------------+------------+------------+--------------+---------------+
| Afghanistan     | Asia       | 652230     | 25500100     | 20343000      |
| Albania         | Europe     | 28748      | 2831741      | 12960000      |
| Algeria         | Africa     | 2381741    | 37100000     | 188681000     |
| Andorra         | Europe     | 468        | 78115        | 3712000       |
| Angola          | Africa     | 1246700    | 20609294     | 100990000     |
+-----------------+------------+------------+--------------+---------------+
```

A country is big if it has an area of bigger than 3 million square km or a population of more than 25 million.

Write a SQL solution to output big countries' name, population and area.

For example, according to the above table, we should output:

```
+--------------+-------------+--------------+
| name         | population  | area         |
+--------------+-------------+--------------+
| Afghanistan  | 25500100    | 652230       |
| Algeria      | 37100000    | 2381741      |
+--------------+-------------+--------------+
```

Solution

sql

```sql
SELECT name, population, area
FROM World
```

```
WHERE area >= 3000000 OR population > 25000000;
```

596. Classes More Than 5 Students | Easy | LeetCode

There is a table `courses` with columns: **student** and **class**

Please list out all classes which have more than or equal to 5 students.

For example, the table:

```
+---------+------------+
| student | class      |
+---------+------------+
| A       | Math       |
| B       | English    |
| C       | Math       |
| D       | Biology    |
| E       | Math       |
| F       | Computer   |
| G       | Math       |
| H       | Math       |
| I       | Math       |
+---------+------------+
```

Should output:

```
+---------+
| class   |
+---------+
| Math    |
+---------+
```

Solution

sql

```
SELECT class
FROM courses
```

```
GROUP BY class
HAVING count(DISTINCT Student)>=5;
```

597. Friend Requests I: Overall Acceptance Rate | Easy | LeetCode

In social network like Facebook or Twitter, people send friend requests and accept others' requests as well. Now given two tables as below: Table: `friend_request`

```
| sender_id | send_to_id |request_date|
|-----------|------------|------------|
| 1         | 2          | 2016_06-01 |
| 1         | 3          | 2016_06-01 |
| 1         | 4          | 2016_06-01 |
| 2         | 3          | 2016_06-02 |
| 3         | 4          | 2016-06-09 |
```

Table: `request_accepted`

```
| requester_id | accepter_id |accept_date |
|--------------|-------------|------------|
| 1            | 2           | 2016_06-03 |
| 1            | 3           | 2016-06-08 |
| 2            | 3           | 2016-06-08 |
| 3            | 4           | 2016-06-09 |
| 3            | 4           | 2016-06-10 |
```

Write a query to find the overall acceptance rate of requests rounded to 2 decimals, which is the number of acceptance divide the number of requests. For the sample data above, your query should return the following result.

```
|accept_rate|
|-----------|
|       0.80|
```

Note:

The accepted requests are not necessarily from the table friend_request. In this case, you just need to simply count the total accepted requests (no matter whether they are in the original requests), and divide it by the number of requests to get the acceptance rate. It is possible that a sender sends multiple requests to the same receiver, and a request could be accepted more than once. In this case, the 'duplicated' requests or acceptances are only counted once. If there is no requests at all, you should return 0.00 as the accept_rate. Explanation: There are 4 unique accepted requests, and there are 5 requests in total. So the rate is 0.80.

Follow-up:

Can you write a query to return the accept rate but for every month? How about the cumulative accept rate for every day?

Solution

sql

```sql
SELECT IFNULL((round(accepts/requests, 2)), 0.0) AS accept_rate
FROM
    (SELECT count(DISTINCT sender_id, send_to_id) AS requests FROM friend_re
    (SELECT count(DISTINCT requester_id, accepter_id) AS accepts FROM request
```

601. Human Traffic of Stadium | Hard | LeetCode

Table: `Stadium`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| visit_date    | date    |
| people        | int     |
+---------------+---------+
```

visit_date is the primary key for this table. Each row of this table contains the visit date [redacted]d visit id to the stadium with the number of people during the visit. No two rows will have the same visit_date, and as the id increases, the dates increase as well.

Write an SQL query to display the records with three or more rows with **consecutive** `id` 's, and the number of people is greater than or equal to 100 for each.

Return the result table ordered by `visit_date` in **ascending order**.

The query result format is in the following example.

```
Stadium table:
+------+------------+-----------+
| id   | visit_date | people    |
+------+------------+-----------+
| 1    | 2017-01-01 | 10        |
| 2    | 2017-01-02 | 109       |
| 3    | 2017-01-03 | 150       |
| 4    | 2017-01-04 | 99        |
| 5    | 2017-01-05 | 145       |
| 6    | 2017-01-06 | 1455      |
| 7    | 2017-01-07 | 199       |
| 8    | 2017-01-09 | 188       |
+------+------------+-----------+

Result table:
+------+------------+-----------+
| id   | visit_date | people    |
+------+------------+-----------+
| 5    | 2017-01-05 | 145       |
| 6    | 2017-01-06 | 1455      |
| 7    | 2017-01-07 | 199       |
| 8    | 2017-01-09 | 188       |
+------+------------+-----------+
The four rows with ids 5, 6, 7, and 8 have consecutive ids and each of them h
The rows with ids 2 and 3 are not included because we need at least three cor
```

lution

sql

```
SELECT DISTINCT s1.*
FROM Stadium s1 JOIN Stadium s2 JOIN Stadium s3
ON (s1.id = s2.id-1 AND s1.id = s3.id-2) OR
 (s1.id = s2.id+1 AND s1.id = s3.id-1) OR
 (s1.id = s2.id+1 AND s1.id = s3.id+2)
    WHERE s1.people >= 100 AND s2.people >= 100 AND s3.people>=100
  ORDER BY visit_date
```

602. Friend Requests II: Who Has the Most Friends | Medium | LeetCode

In social network like Facebook or Twitter, people send friend requests and accept others' requests as well. Table `request_accepted` holds the data of friend acceptance, while requester_id and accepter_id both are the id of a person.

```
| requester_id | accepter_id | accept_date|
|--------------|-------------|------------|
| 1            | 2           | 2016_06-03 |
| 1            | 3           | 2016-06-08 |
| 2            | 3           | 2016-06-08 |
| 3            | 4           | 2016-06-09 |
```

Write a query to find the the people who has most friends and the most friends number. For the sample data above, the result is:

```
| id | num |
|----|-----|
| 3  | 3   |
```

Note:

It is guaranteed there is only 1 people having the most friends. The friend request could only been accepted once, which mean there is no multiple records with the same requester_id and accepter_id value. Explanation: The person with id '3' is a friend of

people '1', '2' and '4', so he has 3 friends in total, which is the most number than any [redacted]ers.

Follow-up: In the real world, multiple people could have the same most number of friends, can you find all these people in this case?

```sql
SELECT t.id, sum(t.num) AS num
FROM (
      (SELECT requester_id AS id, COUNT(1) AS num
       FROM request_accepted
       GROUP BY requester_id)
      union all
       (SELECT accepter_id AS id, COUNT(1) AS num
        FROM request_accepted
        GROUP BY accepter_id)) AS t
GROUP BY t.id
ORDER BY num DESC
LIMIT 1;
```

603. Consecutive Available Seats | Easy | LeetCode

Several friends at a cinema ticket office would like to reserve consecutive available seats. Can you help to query all the consecutive available seats order by the seat_id using the following cinema table?

```
| seat_id | free |
|---------|------|
| 1       | 1    |
| 2       | 0    |
| 3       | 1    |
| 4       | 1    |
| 5       | 1    |
```

Your query should return the following result for the sample case above.

```
| seat_id |
```

```
|---------|
| 3       |
| 4       |
| 5       |
```

Note:

The seat_id is an auto increment int, and free is bool ('1' means free, and '0' means occupied.). Consecutive available seats are more than 2(inclusive) seats consecutively available.

Solution

sql

```sql
SELECT DISTINCT t1.seat_id
FROM cinema AS t1 JOIN cinema AS t2
ON abs(t1.seat_id-t2.seat_id)=1
WHERE t1.free='1' AND t2.free='1'
ORDER BY t1.seat_id
```

607.Sales Person | Easy | LeetCode

Description

Given three tables: `salesperson` , `company` , `orders` . Output all the names in the table salesperson, who didn't have sales to company 'RED'.

Example Input

Table: `salesperson`

```
+----------+------+--------+-----------------+-----------+
| sales_id | name | salary | commission_rate | hire_date |
+----------+------+--------+-----------------+-----------+
|   1      | John | 100000 |     6           | 4/1/2006  |
|   2      | Amy  | 120000 |     5           | 5/1/2010  |
|   3      | Mark | 65000  |     12          | 12/25/2008|
```

```
|   4      | Pam  | 25000  |     25          | 1/1/2005  |
|   5      | Alex | 50000  |     10          | 2/3/2007  |
+----------+------+--------+-----------------+-----------+
```

The table salesperson holds the salesperson information. Every salesperson has a sales_id and a name. Table: `company`

```
+---------+--------+------------+
| com_id  |  name  |    city    |
+---------+--------+------------+
|   1     |  RED   |   Boston   |
|   2     | ORANGE |   New York |
|   3     | YELLOW |   Boston   |
|   4     | GREEN  |   Austin   |
+---------+--------+------------+
```

The table company holds the company information. Every company has a com_id and a name. Table: `orders`

```
+----------+----------+---------+----------+--------+
| order_id |  date    | com_id  | sales_id | amount |
+----------+----------+---------+----------+--------+
| 1        | 1/1/2014 |    3    |    4     | 100000 |
| 2        | 2/1/2014 |    4    |    5     | 5000   |
| 3        | 3/1/2014 |    1    |    1     | 50000  |
| 4        | 4/1/2014 |    1    |    4     | 25000  |
+----------+----------+---------+----------+--------+
```

The table orders holds the sales record information, salesperson and customer company are represented by sales_id and com_id. output

```
+------+
| name |
+------+
| Amy  |
| Mark |
| Alex |
```

```
+------+
```

Explanation

According to order '3' and '4' in table orders, it is easy to tell only salesperson 'John' and 'Alex' have sales to company 'RED', so we need to output all the other names in table salesperson.

Solution

sql

```sql
SELECT name
FROM salesperson
WHERE name NOT IN
    (SELECT DISTINCT salesperson.name
    FROM salesperson, orders, company
    WHERE company.name = 'RED'
    AND salesperson.sales_id = orders.sales_id
    AND orders.com_id = company.com_id)
```

608. Tree Node | Medium | LeetCode

Given a table tree, id is identifier of the tree node and p_id is its parent node's id.

```
+----+------+
| id | p_id |
+----+------+
| 1  | null |
| 2  | 1    |
| 3  | 1    |
| 4  | 2    |
| 5  | 2    |
+----+------+
```

ch node in the tree can be one of three types:

Leaf: if the node is a leaf node. Root: if the node is the root of the tree. Inner: If the node is neither a leaf node nor a root node. Write a query to print the node id and the type of the node. Sort your output by the node id. The result for the above sample is:

```
+----+------+
| id | Type |
+----+------+
| 1  | Root |
| 2  | Inner|
| 3  | Leaf |
| 4  | Leaf |
| 5  | Leaf |
+----+------+
```

Explanation

Node '1' is root node, because its parent node is NULL and it has child node '2' and '3'. Node '2' is inner node, because it has parent node '1' and child node '4' and '5'. Node '3', '4' and '5' is Leaf node, because they have parent node and they don't have child node. And here is the image of the sample tree as below:

```
1
      /   \
    2       3
  /   \
4       5
```

Note

If there is only one node on the tree, you only need to output its root attributes.

Solution

sql

```
## Basic Ideas: LEFT JOIN
# In tree, each node can only one parent or no parent
## | id | p_id | id (child) |
```

```
## |----+------+------------|
## |  1 | null |          1 |
## |  1 | null |          2 |
## |  2 |    1 |          4 |
## |  2 |    1 |          5 |
## |  3 |    1 |       null |
## |  4 |    2 |       null |
## |  5 |    2 |       null |
```

```
SELECT t1.id,
    CASE
        WHEN ISNULL(t1.p_id) THEN 'Root'
        WHEN ISNULL(MAX(t2.id)) THEN 'Leaf'
        ELSE 'Inner'
    END AS Type
FROM tree AS t1 LEFT JOIN tree AS t2
ON t1.id = t2.p_id
GROUP BY t1.id, t1.p_id
```

610. Triangle Judgement | Easy | LeetCode

A pupil Tim gets homework to identify whether three line segments could possibly form a triangle. However, this assignment is very heavy because there are hundreds of records to calculate. Could you help Tim by writing a query to judge whether these three sides can form a triangle, assuming table triangle holds the length of the three sides x, y and z.

```
| x  | y  | z  |
|----|----|----|
| 13 | 15 | 30 |
| 10 | 20 | 15 |
```

For the sample data above, your query should return the follow result:

```
| x  | y  | z  | triangle |
|----|----|----|----------|
| 13 | 15 | 30 | No       |
```

```
| 10 | 20 | 15 | Yes      |
```

Solution

```sql
SELECT x, y, z,
       CASE
           WHEN x+y>z AND y+z>x AND x+z>y THEN 'Yes'
           ELSE 'No'
       END AS triangle
FROM triangle
```

612. Shortest Distance in a Plane | Medium | LeetCode

Table point_2d holds the coordinates (x,y) of some unique points (more than two) in a plane. Write a query to find the shortest distance between these points rounded to 2 decimals.

```
| x  | y  |
|----|----|
| -1 | -1 |
| 0  | 0  |
| -1 | -2 |
```

The shortest distance is 1.00 from point (-1,-1) to (-1,2). So the output should be:

```
| shortest |
|----------|
| 1.00     |
```

Note: The longest distance among all the points are less than 10000.

lution

```sql
SELECT ROUND(MIN(SQRT((t1.x-t2.x)*(t1.x-t2.x) + (t1.y-t2.y)*(t1.y-t2.y))), 2)
FROM point_2d AS t1, point_2d AS t2
WHERE t1.x!=t2.x OR t1.y!=t2.y

# SELECT ROUND(SQRT((t1.x-t2.x)*(t1.x-t2.x) + (t1.y-t2.y)*(t1.y-t2.y)), 2) AS
# FROM point_2d AS t1, point_2d AS t2
# WHERE t1.x!=t2.x OR t1.y!=t2.y
# ORDER BY shortest ASC
# LIMIT 1
```

613. Shortest Distance in a Line | Easy | LeetCode

Table point holds the x coordinate of some points on x-axis in a plane, which are all integers. Write a query to find the shortest distance between two points in these points.

```
| x   |
|-----|
| -1  |
| 0   |
| 2   |
```

The shortest distance is '1' obviously, which is from point '-1' to '0'. So the output is as below:

```
| shortest|
|---------|
| 1       |
```

Note: Every point is unique, which means there is no duplicates in table point.

Follow-up: What if all these points have an id and are arranged from the left most to the right most of x axis?

Solution

```sql
SELECT t1.x-t2.x AS shortest
FROM point AS t1 JOIN point AS t2
WHERE t1.x>t2.x
ORDER BY (t1.x-t2.x) ASC
LIMIT 1
```

614. Second Degree Follower | Medium | LeetCode

In facebook, there is a follow table with two columns: followee, follower.

Please write a sql query to get the amount of each follower's follower if he/she has one.

For example:

```
+-------------+------------+
| followee    | follower   |
+-------------+------------+
|     A       |     B      |
|     B       |     C      |
|     B       |     D      |
|     D       |     E      |
+-------------+------------+
```

should output:

```
+-------------+------------+
| follower    | num        |
+-------------+------------+
|     B       |  2         |
|     D       |  1         |
+-------------+------------+
```

Explanation: Both B and D exist in the follower list, when as a followee, B's follower is C and D, and D's follower is E. A does not exist in follower list.

Note: Followee would not follow himself/herself in all cases. Please display the result in [redacted]ower's alphabet order.

Solution

sql

```sql
## Explain the business logic
##   A follows B. Then A is follwer, B is followee
## What are second degree followers?
##   A follows B, and B follows C.
##   Then A is the second degree followers of C

SELECT f1.follower, COUNT(DISTINCT f2.follower) AS num
FROM follow AS f1 JOIN follow AS f2
ON f1.follower = f2.followee
GROUP BY f1.follower;
```

615. Average Salary: Departments VS Company | Hard | LeetCode

Given two tables as below, write a query to display the comparison result (higher/lower/same) of the average salary of employees in a department to the company's average salary. Table: salary

```
| id | employee_id | amount | pay_date   |
|----|-------------|--------|------------|
| 1  | 1           | 9000   | 2017-03-31 |
| 2  | 2           | 6000   | 2017-03-31 |
| 3  | 3           | 10000  | 2017-03-31 |
| 4  | 1           | 7000   | 2017-02-28 |
| 5  | 2           | 6000   | 2017-02-28 |
| 6  | 3           | 8000   | 2017-02-28 |
```

The employee_id column refers to the employee_id in the following table employee.

```
| employee_id | department_id |
```

```
|-------------|---------------|
| 1           | 1             |
| 2           | 2             |
| 3           | 2             |
```

So for the sample data above, the result is:

```
| pay_month | department_id | comparison  |
|-----------|---------------|-------------|
| 2017-03   | 1             | higher      |
| 2017-03   | 2             | lower       |
| 2017-02   | 1             | same        |
| 2017-02   | 2             | same        |
```

Explanation In March, the company's average salary is (9000+6000+10000)/3 = 8333.33... The average salary for department '1' is 9000, which is the salary of employee_id '1' since there is only one employee in this department. So the comparison result is 'higher' since 9000 > 8333.33 obviously. The average salary of department '2' is (6000 + 10000)/2 = 8000, which is the average of employee_id '2' and '3'. So the comparison result is 'lower' since 8000 < 8333.33. With he same formula for the average salary comparison in February, the result is 'same' since both the department '1' and '2' have the same average salary with the company, which is 7000.

Solution

sql

```sql
SELECT t1.pay_month, t1.department_id,
    (CASE WHEN t1.amount = t2.amount THEN 'same'
          WHEN t1.amount > t2.amount THEN 'higher'
          WHEN t1.amount < t2.amount THEN 'lower' END) AS comparison
FROM
    (SELECT left(pay_date, 7) AS pay_month, department_id, avg(amount) AS amo
    FROM salary JOIN employee
    ON salary.employee_id = employee.employee_id
    GROUP BY pay_month, department_id
    ORDER BY pay_month DESC, department_id) AS t1
    JOIN
    (SELECT left(pay_date, 7) AS pay_month, avg(amount) AS amount
```

```
        FROM salary JOIN employee
        ON salary.employee_id = employee.employee_id
        GROUP BY pay_month) AS t2
        ON t1.pay_month = t2.pay_month
```

618. Students Report By Geography | Hard | LeetCode

A U.S graduate school has students from Asia, Europe and America. The students' location information are stored in table student as below.

```
| name   | continent |
|--------|-----------|
| Jack   | America   |
| Pascal | Europe    |
| Xi     | Asia      |
| Jane   | America   |
```

Pivot the continent column in this table so that each name is sorted alphabetically and displayed underneath its corresponding continent. The output headers should be America, Asia and Europe respectively. It is guaranteed that the student number from America is no less than either Asia or Europe. For the sample input, the output is:

```
| America | Asia | Europe |
|---------|------|--------|
| Jack    | Xi   | Pascal |
| Jane    |      |        |
```

Follow-up: If it is unknown which continent has the most students, can you write a query to generate the student report?

Solution

sql

```
SELECT t1.name AS America, t2.name AS Asia, t3.name AS Europe
FROM
```

```
(SELECT (@cnt1 := @cnt1 + 1) AS id, name
FROM student
CROSS JOIN (SELECT @cnt1 := 0) AS dummy
WHERE continent='America'
ORDER BY name) AS t1
LEFT JOIN
(SELECT (@cnt2 := @cnt2 + 1) AS id, name
FROM student
CROSS JOIN (SELECT @cnt2 := 0) AS dummy
WHERE continent='Asia'
ORDER BY name) AS t2
ON t1.id = t2.id
LEFT JOIN
(SELECT (@cnt3 := @cnt3 + 1) AS id, name
FROM student
CROSS JOIN (SELECT @cnt3 := 0) AS dummy
WHERE continent='Europe'
ORDER BY name) AS t3
ON t1.id = t3.id
```

619. Biggest Single Number | Easy | LeetCode

Table number contains many numbers in column num including duplicated ones. Can you write a SQL query to find the biggest number, which only appears once.

```
+---+
|num|
+---+
| 8 |
| 8 |
| 3 |
| 3 |
| 1 |
| 4 |
| 5 |
| 6 |
```

For the sample data above, your query should return the following result:

```
+---+
|num|
+---+
| 6 |
```

Note: If there is no such number, just output null.

Solution

```sql
SELECT IFNULL((
    SELECT num
    FROM number
    GROUP BY num
    HAVING count(1) = 1
    ORDER BY num DESC
    LIMIT 0, 1), NULL) AS num
```

620. Not Boring Movies | Easy | LeetCode

X city opened a new cinema, many people would like to go to this cinema. The cinema also gives out a poster indicating the movies' ratings and descriptions. Please write a SQL query to output movies with an odd numbered ID and a description that is not 'boring'. Order the result by rating.

For example, table `cinema`:

```
+---------+-----------+--------------+-----------+
|   id    | movie     |  description |  rating   |
+---------+-----------+--------------+-----------+
|   1     | War       |   great 3D   |   8.9     |
|   2     | Science   |   fiction    |   8.5     |
|   3     | irish     |   boring     |   6.2     |
|   4     | Ice song  |   Fantacy    |   8.6     |
|   5     | House card|   Interesting|   9.1     |
+---------+-----------+--------------+-----------+
```

For the example above, the output should be:

```
+---------+-----------+--------------+-----------+
|   id    | movie     |  description |  rating   |
+---------+-----------+--------------+-----------+
|   5     | House card|   Interesting|   9.1     |
|   1     | War       |   great 3D   |   8.9     |
+---------+-----------+--------------+-----------+
```

Solution

sql

```sql
SELECT *
FROM Cinema
WHERE description <> 'boring' AND ID % 2 = 1
ORDER BY rating DESC;
```

626. Exchange Seats | Medium | LeetCode

Mary is a teacher in a middle school and she has a table `seat` storing students' names and their corresponding seat ids.

The column id is continuous increment.

Mary wants to change seats for the adjacent students.

Can you write a SQL query to output the result for Mary?

```
+---------+---------+
|    id   | student |
+---------+---------+
|    1    | Abbot   |
|    2    | Doris   |
|    3    | Emerson |
|    4    | Green   |
|    5    | Jeames  |
+---------+---------+
```

For the sample input, the output is:

```
+---------+---------+
|    id   | student |
+---------+---------+
|    1    | Doris   |
|    2    | Abbot   |
|    3    | Green   |
|    4    | Emerson |
|    5    | Jeames  |
+---------+---------+
```

Note:
If the number of students is odd, there is no need to change the last one's seat.

Solution

sql

```sql
SELECT
IF(id<(SELECT MAX(id) FROM seat),IF(id%2=0,id-1, id+1),IF(id%2=0, id-1, id))
FROM seat
ORDER BY id;
```

627. Swap Salary | LeetCode

Table: **Salary**

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| id          | int      |
| name        | varchar  |
| sex         | ENUM     |
| salary      | int      |
+-------------+----------+

id is the primary key for this table.
```

```
The sex column is ENUM value of type ('m', 'f').
The table contains information about an employee.
```

Write an SQL query to swap all `'f'` and `'m'` values (i.e., change all `'f'` values to `'m'` and vice versa) with a single update statement and no intermediate temp table(s).

Note that you must write a single update statement, DO NOT write any select statement for this problem.

The query result format is in the following example:

```
Salary table:
+----+------+-----+--------+
| id | name | sex | salary |
+----+------+-----+--------+
| 1  | A    | m   | 2500   |
| 2  | B    | f   | 1500   |
| 3  | C    | m   | 5500   |
| 4  | D    | f   | 500    |
+----+------+-----+--------+

Result table:
+----+------+-----+--------+
| id | name | sex | salary |
+----+------+-----+--------+
| 1  | A    | f   | 2500   |
| 2  | B    | m   | 1500   |
| 3  | C    | f   | 5500   |
| 4  | D    | m   | 500    |
+----+------+-----+--------+
(1, A) and (2, C) were changed from 'm' to 'f'.
(2, B) and (4, D) were changed from 'f' to 'm'.
```

Solution

sql

```sql
# With IF
UPDATE Salary SET sex = IF(sex='m', 'f', 'm')
```

```
# With CASE
UPDATE Salary SET sex = CASE WHEN sex='m' THEN 'f' ELSE 'm' END
```

1045. Customers Who Bought All Products | Medium | LeetCode

Table: `Customer`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| customer_id | int     |
| product_key | int     |
+-------------+---------+
```

product_key is a foreign key to Product table. Table: `Product`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_key | int     |
+-------------+---------+
product_key is the primary key column for this table.
```

Write an SQL query for a report that provides the customer ids from the Customer table that bought all the products in the Product table.

For example:

```
Customer table:
+-------------+-------------+
| customer_id | product_key |
+-------------+-------------+
| 1           | 5           |
| 2           | 6           |
| 3           | 5           |
```

```
| 3           | 6           |
| 1           | 6           |
+-------------+-------------+
```

```
Product table:
+-------------+
| product_key |
+-------------+
| 5           |
| 6           |
+-------------+

Result table:
+-------------+
| customer_id |
+-------------+
| 1           |
| 3           |
+-------------+
The customers who bought all the products (5 and 6) are customers with id 1 a
```

Solution

sql

```sql
SELECT customer_id
FROM Customer
GROUP NY customer_id
HAVING count(DISTINCT product_key) = (
    SELECT count(1)
    FROM Product)
```

1050. Actors and Directors Who Cooperated At Least Three Times | Easy | LeetCode

Table: **ActorDirector**

```
+-------------+---------+
| Column Name | Type    |
```

```
+-------------+---------+
| actor_id    | int     |
| director_id | int     |
| timestamp   | int     |
+-------------+---------+
timestamp is the primary key column for this table.
```

Write a SQL query for a report that provides the pairs (actor_id, director_id) where the actor have cooperated with the director at least 3 times.

Example:

```
ActorDirector table:
+-------------+-------------+-------------+
| actor_id    | director_id | timestamp   |
+-------------+-------------+-------------+
| 1           | 1           | 0           |
| 1           | 1           | 1           |
| 1           | 1           | 2           |
| 1           | 2           | 3           |
| 1           | 2           | 4           |
| 2           | 1           | 5           |
| 2           | 1           | 6           |
+-------------+-------------+-------------+

Result table:
+-------------+-------------+
| actor_id    | director_id |
+-------------+-------------+
| 1           | 1           |
+-------------+-------------+
The only pair is (1, 1) where they cooperated exactly 3 times.
```

Solution

sql

```sql
SELECT actor_id, director_id
FROM ActorDirector
GROUP BY actor_id, director_id
```

```
HAVING COUNT(1)>=3
```

1068. Product Sales Analysis I | Easy | LeetCode

Table: **Sales**

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| sale_id     | int   |
| product_id  | int   |
| year        | int   |
| quantity    | int   |
| price       | int   |
+-------------+-------+
(sale_id, year) is the primary key of this table.
product_id is a foreign key to Product table.
Note that the price is per unit.
```

Table: **Product**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
+--------------+---------+
product_id is the primary key of this table.
```

Write an SQL query that reports all product names of the products in the Sales table along with their selling year and price.

For example:

```
Sales table:
+---------+------------+------+----------+-------+
| sale_id | product_id | year | quantity | price |
```

```
+---------+------------+------+----------+-------+
| 1       | 100        | 2008 | 10       | 5000  |
| 2       | 100        | 2009 | 12       | 5000  |
| 7       | 200        | 2011 | 15       | 9000  |
+---------+------------+------+----------+-------+

Product table:
+------------+--------------+
| product_id | product_name |
+------------+--------------+
| 100        | Nokia        |
| 200        | Apple        |
| 300        | Samsung      |
+------------+--------------+

Result table:
+--------------+-------+-------+
| product_name | year  | price |
+--------------+-------+-------+
| Nokia        | 2008  | 5000  |
| Nokia        | 2009  | 5000  |
| Apple        | 2011  | 9000  |
+--------------+-------+-------+
```

Solution

sql

```sql
SELECT product_name, year, price
FROM Sales JOIN Product
ON Product.product_id = Sales.product_id
```

1069. Product Sales Analysis II | Easy | LeetCode

Table: **Sales**

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
```

```
| sale_id     | int   |
| product_id  | int   |
| year        | int   |
| quantity    | int   |
| price       | int   |
+-------------+-------+
sale_id is the primary key of this table.
product_id is a foreign key to Product table.
Note that the price is per unit.
```

Table: **Product**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
+--------------+---------+
product_id is the primary key of this table.
```

Write an SQL query that reports the total quantity sold for every product id.

The query result format is in the following example:

```
Sales table:
+---------+------------+------+----------+-------+
| sale_id | product_id | year | quantity | price |
+---------+------------+------+----------+-------+
| 1       | 100        | 2008 | 10       | 5000  |
| 2       | 100        | 2009 | 12       | 5000  |
| 7       | 200        | 2011 | 15       | 9000  |
+---------+------------+------+----------+-------+

Product table:
+------------+--------------+
| product_id | product_name |
+------------+--------------+
| 100        | Nokia        |
| 200        | Apple        |
```

```
| 300        | Samsung      |
+------------+--------------+
```

```
Result table:
+--------------+----------------+
| product_id   | total_quantity |
+--------------+----------------+
| 100          | 22             |
| 200          | 15             |
+--------------+----------------+
```

Solution

sql

```sql
SELECT product_id, sum(quantity) AS total_quantity
FROM Sales
GROUP BY product_id;
```

1070. Product Sales Analysis III | Medium | LeetCode

Table: **Sales**

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| sale_id     | int   |
| product_id  | int   |
| year        | int   |
| quantity    | int   |
| price       | int   |
+-------------+-------+
sale_id is the primary key of this table.
product_id is a foreign key to Product table.
Note that the price is per unit.
```

Table: **Product**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
+--------------+---------+
product_id is the primary key of this table.
```

Write an SQL query that selects the product id, year, quantity, and price for the first year of every product sold.

The query result format is in the following example:

```
Sales table:
+---------+------------+------+----------+-------+
| sale_id | product_id | year | quantity | price |
+---------+------------+------+----------+-------+
| 1       | 100        | 2008 | 10       | 5000  |
| 2       | 100        | 2009 | 12       | 5000  |
| 7       | 200        | 2011 | 15       | 9000  |
+---------+------------+------+----------+-------+

Product table:
+------------+--------------+
| product_id | product_name |
+------------+--------------+
| 100        | Nokia        |
| 200        | Apple        |
| 300        | Samsung      |
+------------+--------------+

Result table:
+------------+------------+----------+-------+
| product_id | first_year | quantity | price |
+------------+------------+----------+-------+
| 100        | 2008       | 10       | 5000  |
| 200        | 2011       | 15       | 9000  |
+------------+------------+----------+-------+
```

Solution

```sql
SELECT
    product_id,
    year first_year,
    quantity,
    price
FROM Sales
WHERE (product_id, year) IN (SELECT product_id, MIN(year)
                             FROM Sales
                             GROUP BY product_id)
```

1075. Project Employees I | Easy | LeetCode

Table: **Project**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| project_id  | int     |
| employee_id | int     |
+-------------+---------+
(project_id, employee_id) is the primary key of this table.
employee_id is a foreign key to Employee table.
```

Table: **Employee**

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| employee_id      | int     |
| name             | varchar |
| experience_years | int     |
+------------------+---------+
employee_id is the primary key of this table.
```

Write an SQL query that reports the average experience years of all the employees for each project, rounded to 2 digits.

The query result format is in the following example:

```
Project table:
+-------------+-------------+
| project_id  | employee_id |
+-------------+-------------+
| 1           | 1           |
| 1           | 2           |
| 1           | 3           |
| 2           | 1           |
| 2           | 4           |
+-------------+-------------+

Employee table:
+-------------+--------+------------------+
| employee_id | name   | experience_years |
+-------------+--------+------------------+
| 1           | Khaled | 3                |
| 2           | Ali    | 2                |
| 3           | John   | 1                |
| 4           | Doe    | 2                |
+-------------+--------+------------------+

Result table:
+-------------+---------------+
| project_id  | average_years |
+-------------+---------------+
| 1           | 2.00          |
| 2           | 2.50          |
+-------------+---------------+
The average experience years for the first project is (3 + 2 + 1) / 3 = 2.00
```

Solution

sql

```sql
SELECT
    p.project_id,
```

```
    ROUND(AVG(e.experience_years),2) average_years
FROM
    Project p JOIN Employee e ON
    p.employee_id = e.employee_id
GROUP BY
    p.project_id
```

1076. Project Employees II | Easy | LeetCode

Table: **Project**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| project_id  | int     |
| employee_id | int     |
+-------------+---------+
(project_id, employee_id) is the primary key of this table.
employee_id is a foreign key to Employee table.
```

Table: **Employee**

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| employee_id      | int     |
| name             | varchar |
| experience_years | int     |
+------------------+---------+
employee_id is the primary key of this table.
```

Write an SQL query that reports all the projects that have the most employees.

The query result format is in the following example:

```
Project table:
+-------------+-------------+
```

```
| project_id  | employee_id |
+-------------+-------------+
| 1           | 1           |
| 1           | 2           |
| 1           | 3           |
| 2           | 1           |
| 2           | 4           |
+-------------+-------------+

Employee table:
+-------------+--------+------------------+
| employee_id | name   | experience_years |
+-------------+--------+------------------+
| 1           | Khaled | 3                |
| 2           | Ali    | 2                |
| 3           | John   | 1                |
| 4           | Doe    | 2                |
+-------------+--------+------------------+

Result table:
+-------------+
| project_id  |
+-------------+
| 1           |
+-------------+
The first project has 3 employees while the second one has 2.
```

sql

```sql
SELECT project_id
FROM Project
GROUP BY project_id
HAVING COUNT(employee_id) = (SELECT COUNT(employee_id)
                             FROM Project
                             GROUP BY project_id
                             ORDER BY COUNT(employee_id) DESC
                             LIMIT 1)
```

)77. Project Employees III | Medium | LeetCode

Table: **Project**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| project_id  | int     |
| employee_id | int     |
+-------------+---------+
(project_id, employee_id) is the primary key of this table.
employee_id is a foreign key to Employee table.
```

Table: **Employee**

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| employee_id      | int     |
| name             | varchar |
| experience_years | int     |
+------------------+---------+
employee_id is the primary key of this table.
```

Write an SQL query that reports the most experienced employees in each project. In case of a tie, report all employees with the maximum number of experience years.

The query result format is in the following example:

```
Project table:
+-------------+-------------+
| project_id  | employee_id |
+-------------+-------------+
| 1           | 1           |
| 1           | 2           |
| 1           | 3           |
| 2           | 1           |
| 2           | 4           |
+-------------+-------------+
```

Employee table:

```
+-------------+--------+------------------+
| employee_id | name   | experience_years |
+-------------+--------+------------------+
| 1           | Khaled | 3                |
| 2           | Ali    | 2                |
| 3           | John   | 3                |
| 4           | Doe    | 2                |
+-------------+--------+------------------+
```

Result table:

```
+-------------+---------------+
| project_id  | employee_id   |
+-------------+---------------+
| 1           | 1             |
| 1           | 3             |
| 2           | 1             |
+-------------+---------------+
```

Both employees with id 1 and 3 have the most experience among the employees c

Solution

sql

```sql
SELECT
    p.project_id,
    e.employee_id
FROM
    Project p LEFT JOIN Employee e ON
    p.employee_id = e.employee_id
WHERE (p.project_id,
       e.experience_years) IN (SELECT
                                   p.project_id,
                                   MAX(e.experience_years)
                               FROM
                                   Project p JOIN Employee e ON
                                   p.employee_id = e.employee_id
                               GROUP BY
                                   p.project_id)
```

1082. Sales Analysis I | Easy | LeetCode

Table: `Product`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
| unit_price   | int     |
+--------------+---------+
product_id is the primary key of this table.
```

Table: Sales

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| seller_id   | int     |
| product_id  | int     |
| buyer_id    | int     |
| sale_date   | date    |
| quantity    | int     |
| price       | int     |
+------ ------+---------+
This table has no primary key, it can have repeated rows.
product_id is a foreign key to Product table.
```

Write an SQL query that reports the best seller by total sales price, If there is a tie, report them all.

The query result format is in the following example:

```
Product table:
+------------+--------------+------------+
| product_id | product_name | unit_price |
+------------+--------------+------------+
| 1          | S8           | 1000       |
```

```
| 2          | G4           | 800        |
| 3          | iPhone       | 1400       |
+------------+--------------+------------+
```

```
Sales table:
+-----------+------------+---------+------------+----------+-------+
| seller_id | product_id | buyer_id | sale_date  | quantity | price |
+-----------+------------+---------+------------+----------+-------+
| 1         | 1          | 1        | 2019-01-21 | 2        | 2000  |
| 1         | 2          | 2        | 2019-02-17 | 1        | 800   |
| 2         | 2          | 3        | 2019-06-02 | 1        | 800   |
| 3         | 3          | 4        | 2019-05-13 | 2        | 2800  |
+-----------+------------+---------+------------+----------+-------+

Result table:
+-------------+
| seller_id   |
+-------------+
| 1           |
| 3           |
+-------------+
Both sellers with id 1 and 3 sold products with the most total price of 2800.
```

Solution

sql

```sql
SELECT seller_id
FROM Sales
GROUP BY seller_id
HAVING SUM(price) = (SELECT SUM(price)
                     FROM Sales
                     GROUP BY seller_id
                     ORDER BY SUM(price) DESC
                     LIMIT 1)
```

1083. Sales Analysis II | Easy | LeetCode

Table: **Product**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
| unit_price   | int     |
+--------------+---------+
product_id is the primary key of this table.
```

Table: **Sales**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| seller_id   | int     |
| product_id  | int     |
| buyer_id    | int     |
| sale_date   | date    |
| quantity    | int     |
| price       | int     |
+------ ------+---------+
This table has no primary key, it can have repeated rows.
product_id is a foreign key to Product table.
```

Write an SQL query that reports the buyers who have bought S8 but not iPhone. Note that S8 and iPhone are products present in the Product table.

The query result format is in the following example:

```
Product table:
+------------+--------------+------------+
| product_id | product_name | unit_price |
+------------+--------------+------------+
| 1          | S8           | 1000       |
| 2          | G4           | 800        |
| 3          | iPhone       | 1400       |
+------------+--------------+------------+

Sales table:
```

```
+-----------+------------+---------+------------+----------+-------+
| seller_id | product_id | buyer_id | sale_date  | quantity | price |
+-----------+------------+---------+------------+----------+-------+
| 1         | 1          | 1        | 2019-01-21 | 2        | 2000  |
| 1         | 2          | 2        | 2019-02-17 | 1        | 800   |
| 2         | 1          | 3        | 2019-06-02 | 1        | 800   |
| 3         | 3          | 3        | 2019-05-13 | 2        | 2800  |
+-----------+------------+---------+------------+----------+-------+

Result table:
+-------------+
| buyer_id    |
+-------------+
| 1           |
+-------------+
The buyer with id 1 bought an S8 but didn't buy an iPhone. The buyer with id
```

Solution

sql

```sql
SELECT DISTINCT s.buyer_id
FROM Sales s LEFT JOIN Product p ON
    s.product_id = p.product_id
WHERE p.product_name = 'S8' AND
      s.buyer_id NOT IN (SELECT s.buyer_id
                         FROM Sales s LEFT JOIN Product p ON
                             s.product_id = p.product_id
                         WHERE p.product_name = 'iPhone')
```

1084. Sales Analysis III | Easy | LeetCode

Reports the products that were only sold in spring 2019. That is, between 2019-01-01 and 2019-03-31 inclusive. Select the product that were only sold in spring 2019.

```
Product table:
+------------+--------------+------------+
| product_id | product_name | unit_price |
+------------+--------------+------------+
```

```
| 1          | S8           | 1000       |
| 2          | G4           | 800        |
| 3          | iPhone       | 1400       |
+------------+--------------+------------+

Sales table:
+-----------+------------+----------+------------+----------+-------+
| seller_id | product_id | buyer_id | sale_date  | quantity | price |
+-----------+------------+----------+------------+----------+-------+
| 1         | 1          | 1        | 2019-01-21 | 2        | 2000  |
| 1         | 2          | 2        | 2019-02-17 | 1        | 800   |
| 2         | 2          | 3        | 2019-06-02 | 1        | 800   |
| 3         | 3          | 4        | 2019-05-13 | 2        | 2800  |
+-----------+------------+----------+------------+----------+-------+

Result table:
+-------------+--------------+
| product_id  | product_name |
+-------------+--------------+
| 1           | S8           |
+-------------+--------------+
The product with id 1 was only sold in spring 2019 while the other two were s
```

Solution

sql

```sql
(SELECT DISTINCT s.product_id, p.product_name
FROM Sales s LEFT JOIN Product p ON
    s.product_id = p.product_id
WHERE s.sale_date >= '2019-01-01' AND
      s.sale_date <= '2019-03-31')
EXCEPT -- MINUS if Oracle
(SELECT DISTINCT s.product_id, p.product_name
FROM Sales s LEFT JOIN Product p ON
    s.product_id = p.product_id
WHERE s.sale_date < '2019-01-01' OR
      s.sale_date > '2019-03-31')
```

1097. Game Play Analysis V | Hard | LeetCode

We define the install date of a player to be the first login day of that player. We also define day 1 retention of some date X to be the number of players whose install date is X and they logged back in on the day right after X , divided by the number of players whose install date is X, rounded to 2 decimal places. Write an SQL query that reports for each install date, the number of players that installed the game on that day and the day 1 retention. The query result format is in the following example:

```
Activity table:
+-----------+-----------+------------+--------------+
| player_id | device_id | event_date | games_played |
+-----------+-----------+------------+--------------+
| 1         | 2         | 2016-03-01 | 5            |
| 1         | 2         | 2016-03-02 | 6            |
| 2         | 3         | 2017-06-25 | 1            |
| 3         | 1         | 2016-03-01 | 0            |
| 3         | 4         | 2016-07-03 | 5            |
+-----------+-----------+------------+--------------+

Result table:
+------------+----------+----------------+
| install_dt | installs | Day1_retention |
+------------+----------+----------------+
| 2016-03-01 | 2        | 0.50           |
| 2017-06-25 | 1        | 0.00           |
+------------+----------+----------------+

Player 1 and 3 installed the game on 2016-03-01 but only player 1 logged bacl
Player 2 installed the game on 2017 -06-25 but didn't log back in on 2017-06-
```

Solution

sql

```sql
SELECT
    install_dt,
    COUNT(player_id) installs,
    ROUND(COUNT(retention)/COUNT(player_id),2) Day1_retention  --the number o
FROM
    (
    SELECT a.player_id, a.install_dt, b.event_date retention -- id, the reco
```

```
    FROM
        (SELECT player_id, MIN(event_date) install_dt   --subquery 1 take the
        FROM Activity
        GROUP BY player_id) a LEFT JOIN Activity b ON   --sql left join the
            a.player_id = b.player_id AND
            a.install_dt + 1=b.event_date
    ) AS tmp
GROUP BY
    install_dt
```

1098. Unpopular Books | Medium | LeetCode

Table: **Books**

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| book_id        | int     |
| name           | varchar |
| available_from | date    |
+----------------+---------+
book_id is the primary key of this table.
```

Table: **Orders**

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| order_id       | int     |
| book_id        | int     |
| quantity       | int     |
| dispatch_date  | date    |
+----------------+---------+
order_id is the primary key of this table.
book_id is a foreign key to the Books table.
```

Write an SQL query that reports the books that have sold less than 10 copies in the last [redacted]r, excluding books that have been available for less than 1 month from today. Assume today is 2019-06-23.

The query result format is in the following example:

```
Books table:
+---------+--------------------+----------------+
| book_id | name               | available_from |
+---------+--------------------+----------------+
| 1       | "Kalila And Demna" | 2010-01-01     |
| 2       | "28 Letters"       | 2012-05-12     |
| 3       | "The Hobbit"       | 2019-06-10     |
| 4       | "13 Reasons Why"   | 2019-06-01     |
| 5       | "The Hunger Games" | 2008-09-21     |
+---------+--------------------+----------------+

Orders table:
+----------+---------+----------+---------------+
| order_id | book_id | quantity | dispatch_date |
+----------+---------+----------+---------------+
| 1        | 1       | 2        | 2018-07-26    |
| 2        | 1       | 1        | 2018-11-05    |
| 3        | 3       | 8        | 2019-06-11    |
| 4        | 4       | 6        | 2019-06-05    |
| 5        | 4       | 5        | 2019-06-20    |
| 6        | 5       | 9        | 2009-02-02    |
| 7        | 5       | 8        | 2010-04-13    |
+----------+---------+----------+---------------+

Result table:
+-----------+--------------------+
| book_id   | name               |
+-----------+--------------------+
| 1         | "Kalila And Demna" |
| 2         | "28 Letters"       |
| 5         | "The Hunger Games" |
+-----------+--------------------+
```

Solution

```sql
SELECT
    b.book_id, b.name
FROM
    Books b LEFT JOIN (                    -- subquery calculates last year's
        SELECT book_id, SUM(quantity) nsold
        FROM Orders
        WHERE dispatch_date BETWEEN '2018-06-23' AND '2019-06-23'
        GROUP BY book_id
    ) o
    ON b.book_id = o.book_id
WHERE
    (o.nsold < 10 OR o.nsold IS NULL) AND          -- Sales less than 10 or
    DATEDIFF('2019-06-23', b.available_from) > 30   -- Not a new book within
```

1107. New Users Daily Count | Medium | LeetCode

Table: **Traffic**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| activity      | enum    |
| activity_date | date    |
+---------------+---------+
There is no primary key for this table, it may have duplicate rows.
The activity column is an ENUM type of ('login', 'logout', 'jobs', 'groups',
```

Write an SQL query that reports for every date within at most 90 days from today, the number of users that logged in for the first time on that date. Assume today is 2019-06-30.

The query result format is in the following example:

```
Traffic table:
+---------+----------+---------------+
| user_id | activity | activity_date |
```

```
+---------+----------+---------------+
| 1       | login    | 2019-05-01    |
| 1       | homepage | 2019-05-01    |
| 1       | logout   | 2019-05-01    |
| 2       | login    | 2019-06-21    |
| 2       | logout   | 2019-06-21    |
| 3       | login    | 2019-01-01    |
| 3       | jobs     | 2019-01-01    |
| 3       | logout   | 2019-01-01    |
| 4       | login    | 2019-06-21    |
| 4       | groups   | 2019-06-21    |
| 4       | logout   | 2019-06-21    |
| 5       | login    | 2019-03-01    |
| 5       | logout   | 2019-03-01    |
| 5       | login    | 2019-06-21    |
| 5       | logout   | 2019-06-21    |
+---------+----------+---------------+

Result table:
+------------+-------------+
| login_date | user_count  |
+------------+-------------+
| 2019-05-01 | 1           |
| 2019-06-21 | 2           |
+------------+-------------+
Note that we only care about dates with non zero user count.
The user with id 5 first logged in on 2019-03-01 so he's not counted on 2019-
```

Solution

sql

```sql
#Solution- 1:
SELECT login_date, COUNT(user_id) AS user_count
FROM (SELECT user_id, MIN(activity_date) AS login_date
    FROM Traffic
    WHERE activity = 'login'
    GROUP BY user_id) AS t
WHERE login_date >= DATE_ADD('2019-06-30', INTERVAL -90 DAY) AND login_date <
GROUP BY login_date

#Solution- 2:
```

```
SELECT login_date, COUNT(user_id) user_count
FROM
    (SELECT user_id, MIN(activity_date) as login_date
    FROM Traffic
    WHERE activity='login'
    GROUP BY user_id) as t
WHERE DATEDIFF('2019-06-30', login_date) <= 90
GROUP BY login_date
```

1112. Highest Grade For Each Student | Medium | LeetCode

Table: **Enrollments**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| student_id    | int     |
| course_id     | int     |
| grade         | int     |
+---------------+---------+
(student_id, course_id) is the primary key of this table.
```

Write a SQL query to find the highest grade with its corresponding course for each student. In case of a tie, you should find the course with the smallest course_id. The output must be sorted by increasing student_id.

The query result format is in the following example:

```
Enrollments table:
+------------+-------------------+
| student_id | course_id | grade |
+------------+-----------+-------+
| 2          | 2         | 95    |
| 2          | 3         | 95    |
| 1          | 1         | 90    |
| 1          | 2         | 99    |
| 3          | 1         | 80    |
```

```
| 3          | 2          | 75    |
| 3          | 3          | 82    |
+------------+------------+-------+
```

```
Result table:
+------------+------------------+
| student_id | course_id | grade |
+------------+-----------+-------+
| 1          | 2         | 99    |
| 2          | 2         | 95    |
| 3          | 3         | 82    |
+------------+-----------+-------+
```

Solution

sql

```sql
SELECT student_id, MIN(course_id) course_id, grade
FROM Enrollments
WHERE (student_id, grade) IN
        (SELECT student_id, MAX(grade)
        FROM Enrollments
        GROUP BY student_id)
GROUP BY student_id
ORDER BY student_id;
```

1113.Reported Posts | Easy | LeetCode

Table: **Actions**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| post_id       | int     |
| action_date   | date    |
| action        | enum    |
| extra         | varchar |
+---------------+---------+
There is no primary key for this table, it may have duplicate rows.
```

```
The action column is an ENUM type of ('view', 'like', 'reaction', 'comment',
The extra column has optional information about the action such as a reason t
```

Write an SQL query that reports the number of posts reported yesterday for each report reason. Assume today is 2019-07-05.

The query result format is in the following example:

```
Actions table:
+---------+---------+-------------+--------+--------+
| user_id | post_id | action_date | action | extra  |
+---------+---------+-------------+--------+--------+
| 1       | 1       | 2019-07-01  | view   | null   |
| 1       | 1       | 2019-07-01  | like   | null   |
| 1       | 1       | 2019-07-01  | share  | null   |
| 2       | 4       | 2019-07-04  | view   | null   |
| 2       | 4       | 2019-07-04  | report | spam   |
| 3       | 4       | 2019-07-04  | view   | null   |
| 3       | 4       | 2019-07-04  | report | spam   |
| 4       | 3       | 2019-07-02  | view   | null   |
| 4       | 3       | 2019-07-02  | report | spam   |
| 5       | 2       | 2019-07-04  | view   | null   |
| 5       | 2       | 2019-07-04  | report | racism |
| 5       | 5       | 2019-07-04  | view   | null   |
| 5       | 5       | 2019-07-04  | report | racism |
+---------+---------+-------------+--------+--------+
Result table:
+---------------+--------------+
| report_reason | report_count |
+---------------+--------------+
| spam          | 1            |
| racism        | 2            |
+---------------+--------------+
```

Note that we only care about report reasons with non zero number of reports.

lution

sql

```
SELECT extra report_reason, COUNT(DISTINCT post_id) report_count
FROM
    (SELECT post_id, extra
    FROM Actions
    WHERE action_date = DATE_SUB('2019-07-05', INTERVAL 1 DAY) AND
          action = 'report') AS tmp
GROUP BY extra
```

1126. Active Businesses | Medium | LeetCode

Table: **Events**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| business_id   | int     |
| event_type    | varchar |
| occurences    | int     |
+---------------+---------+
(business_id, event_type) is the primary key of this table.
Each row in the table logs the info that an event of some type occured at sor
```

Write an SQL query to find all active businesses.

An active business is a business that has more than one event type with occurences greater than the average occurences of that event type among all businesses.

The query result format is in the following example:

```
Events table:
+-------------+------------+------------+
| business_id | event_type | occurences |
+-------------+------------+------------+
| 1           | reviews    | 7          |
| 3           | reviews    | 3          |
| 1           | ads        | 11         |
| 2           | ads        | 7          |
```

```
| 3           | ads        | 6           |
| 1           | page views | 3           |
| 2           | page views | 12          |
+-------------+------------+-------------+

Result table:
+-------------+
| business_id |
+-------------+
| 1           |
+-------------+
Average for 'reviews', 'ads' and 'page views' are (7+3)/2=5, (11+7+6)/3=8, (3
Business with id 1 has 7 'reviews' events (more than 5) and 11 'ads' events (
```

Solution

sql

```sql
SELECT business_id
FROM (SELECT a.business_id, a.event_type, a.occurences, b.event_avg  -- sub 2
      FROM Events a LEFT JOIN
        (SELECT event_type, AVG(occurences) event_avg   -- sub 1
         FROM Events
         GROUP BY event_type) b ON
      a.event_type = b.event_type) tmp
WHERE occurences > event_avg
GROUP BY business_id
HAVING COUNT(event_type) > 1
```

1127. User Purchase Platform | Hard | LeetCode

Table: **Spending**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| user_id     | int     |
| spend_date  | date    |
| platform    | enum    |
```

```
| amount      | int     |
+-------------+---------+
The table logs the spendings history of users that make purchases from an on
(user_id, spend_date, platform) is the primary key of this table.
The platform column is an ENUM type of ('desktop', 'mobile').
```

Write an SQL query to find the total number of users and the total amount spent using mobile only, desktop only and both mobile and desktop together for each date.

The query result format is in the following example:

```
Spending table:
+---------+------------+----------+--------+
| user_id | spend_date | platform | amount |
+---------+------------+----------+--------+
| 1       | 2019-07-01 | mobile   | 100    |
| 1       | 2019-07-01 | desktop  | 100    |
| 2       | 2019-07-01 | mobile   | 100    |
| 2       | 2019-07-02 | mobile   | 100    |
| 3       | 2019-07-01 | desktop  | 100    |
| 3       | 2019-07-02 | desktop  | 100    |
+---------+------------+----------+--------+

Result table:
+------------+----------+--------------+-------------+
| spend_date | platform | total_amount | total_users |
+------------+----------+--------------+-------------+
| 2019-07-01 | desktop  | 100          | 1           |
| 2019-07-01 | mobile   | 100          | 1           |
| 2019-07-01 | both     | 200          | 1           |
| 2019-07-02 | desktop  | 100          | 1           |
| 2019-07-02 | mobile   | 100          | 1           |
| 2019-07-02 | both     | 0            | 0           |
+------------+----------+--------------+-------------+
On 2019-07-01, user 1 purchased using both desktop and mobile, user 2 purchas
On 2019-07-02, user 2 purchased using mobile only, user 3 purchased using des
```

lution

```sql
SELECT aa.spend_date,
       aa.platform,
       COALESCE(bb.total_amount, 0) total_amount,
       COALESCE(bb.total_users,0) total_users
FROM
    (SELECT DISTINCT(spend_date), a.platform   -- table aa
    FROM Spending JOIN
        (SELECT 'desktop' AS platform UNION
        SELECT 'mobile' AS platform UNION
        SELECT 'both' AS platform
        ) a
    ) aa
    LEFT JOIN
    (SELECT spend_date,                        -- table bb
            platform,
            SUM(amount) total_amount,
            COUNT(user_id) total_users
    FROM
        (SELECT spend_date,
                user_id,
                (CASE COUNT(DISTINCT platform)
                    WHEN 1 THEN platform
                    WHEN 2 THEN 'both'
                    END) platform,
                SUM(amount) amount
        FROM Spending
        GROUP BY spend_date, user_id
        ) b
    GROUP BY spend_date, platform
    ) bb
    ON aa.platform = bb.platform AND
    aa.spend_date = bb.spend_date
```

1132. Reported Posts II | Medium | LeetCode

Table: **Actions**

```
+---------------+---------+
```

```
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| post_id       | int     |
| action_date   | date    |
| action        | enum    |
| extra         | varchar |
+---------------+---------+
There is no primary key for this table, it may have duplicate rows.
The action column is an ENUM type of ('view', 'like', 'reaction', 'comment',
The extra column has optional information about the action such as a reason t
```

Table: **Removals**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| post_id       | int     |
| remove_date   | date    |
+---------------+---------+
post_id is the primary key of this table.
Each row in this table indicates that some post was removed as a result of be
```

Write an SQL query to find the average for daily percentage of posts that got removed after being reported as spam, rounded to 2 decimal places.

The query result format is in the following example:

```
Actions table:
+---------+---------+-------------+--------+--------+
| user_id | post_id | action_date | action | extra  |
+---------+---------+-------------+--------+--------+
| 1       | 1       | 2019-07-01  | view   | null   |
| 1       | 1       | 2019-07-01  | like   | null   |
| 1       | 1       | 2019-07-01  | share  | null   |
| 2       | 2       | 2019-07-04  | view   | null   |
| 2       | 2       | 2019-07-04  | report | spam   |
| 3       | 4       | 2019-07-04  | view   | null   |
| 3       | 4       | 2019-07-04  | report | spam   |
```

```
| 4       | 3       | 2019-07-02  | view   | null   |
| 4       | 3       | 2019-07-02  | report | spam   |
| 5       | 2       | 2019-07-03  | view   | null   |
| 5       | 2       | 2019-07-03  | report | racism |
| 5       | 5       | 2019-07-03  | view   | null   |
| 5       | 5       | 2019-07-03  | report | racism |
+---------+---------+-------------+--------+--------+

Removals table:
+---------+-------------+
| post_id | remove_date |
+---------+-------------+
| 2       | 2019-07-20  |
| 3       | 2019-07-18  |
+---------+-------------+

Result table:
+----------------------+
| average_daily_percent |
+----------------------+
| 75.00                |
+----------------------+
The percentage for 2019-07-04 is 50% because only one post of two spam report
The percentage for 2019-07-02 is 100% because one post was reported as spam a
The other days had no spam reports so the average is (50 + 100) / 2 = 75%
Note that the output is only one number and that we do not care about the rer
```

Solution

sql

```sql
WITH t1 AS(
SELECT a.action_date, (COUNT(DISTINCT r.post_id))/(COUNT(DISTINCT a.post_id))
FROM (SELECT action_date, post_id
FROM actions
WHERE extra = 'spam' AND action = 'report') a
LEFT JOIN
removals r
ON a.post_id = r.post_id
GROUP BY a.action_date)

SELECT ROUND(AVG(t1.result)*100,2) AS  average_daily_percent
```

```
FROM t1
```

1141. User Activity for the Past 30 Days I | Easy | LeetCode

Table: **Activity**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| session_id    | int     |
| activity_date | date    |
| activity_type | enum    |
+---------------+---------+
There is no primary key for this table, it may have duplicate rows.
The activity_type column is an ENUM of type ('open_session', 'end_session',
The table shows the user activities for a social media website.
Note that each session belongs to exactly one user.
```

Write an SQL query to find the daily active user count for a period of 30 days ending 2019-07-27 inclusively. A user was active on some day if he/she made at least one activity on that day.

The query result format is in the following example:

```
Activity table:
+---------+------------+---------------+---------------+
| user_id | session_id | activity_date | activity_type |
+---------+------------+---------------+---------------+
| 1       | 1          | 2019-07-20    | open_session  |
| 1       | 1          | 2019-07-20    | scroll_down   |
| 1       | 1          | 2019-07-20    | end_session   |
| 2       | 4          | 2019-07-20    | open_session  |
| 2       | 4          | 2019-07-21    | send_message  |
| 2       | 4          | 2019-07-21    | end_session   |
| 3       | 2          | 2019-07-21    | open_session  |
| 3       | 2          | 2019-07-21    | send_message  |
```

```
| 3       | 2           | 2019-07-21    | end_session   |
| 4       | 3           | 2019-06-25    | open_session  |
| 4       | 3           | 2019-06-25    | end_session   |
+---------+-------------+---------------+---------------+
```

```
Result table:
+------------+--------------+
| day        | active_users |
+------------+--------------+
| 2019-07-20 | 2            |
| 2019-07-21 | 2            |
+------------+--------------+
Note that we do not care about days with zero active users.
```

Solution

sql

```sql
SELECT activity_date AS day, COUNT(DISTINCT user_id) AS active_users
FROM activity
WHERE activity_date > '2019-06-26' AND activity_date < '2019-07-27'
GROUP BY activity_date
```

1142. User Activity for the Past 30 Days II | Easy | LeetCode

Table: **Activity**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| session_id    | int     |
| activity_date | date    |
| activity_type | enum    |
+---------------+---------+
There is no primary key for this table, it may have duplicate rows.
The activity_type column is an ENUM of type ('open_session', 'end_session',
The table shows the user activities for a social media website.
```

```
Note that each session belongs to exactly one user.
```

Write an SQL query to find the average number of sessions per user for a period of 30 days ending 2019-07-27 inclusively, rounded to 2 decimal places. The sessions we want to count for a user are those with at least one activity in that time period.

The query result format is in the following example:

```
Activity table:
+---------+------------+---------------+---------------+
| user_id | session_id | activity_date | activity_type |
+---------+------------+---------------+---------------+
| 1       | 1          | 2019-07-20    | open_session  |
| 1       | 1          | 2019-07-20    | scroll_down   |
| 1       | 1          | 2019-07-20    | end_session   |
| 2       | 4          | 2019-07-20    | open_session  |
| 2       | 4          | 2019-07-21    | send_message  |
| 2       | 4          | 2019-07-21    | end_session   |
| 3       | 2          | 2019-07-21    | open_session  |
| 3       | 2          | 2019-07-21    | send_message  |
| 3       | 2          | 2019-07-21    | end_session   |
| 3       | 5          | 2019-07-21    | open_session  |
| 3       | 5          | 2019-07-21    | scroll_down   |
| 3       | 5          | 2019-07-21    | end_session   |
| 4       | 3          | 2019-06-25    | open_session  |
| 4       | 3          | 2019-06-25    | end_session   |
+---------+------------+---------------+---------------+

Result table:
+---------------------------+
| average_sessions_per_user |
+---------------------------+
| 1.33                      |
+---------------------------+
User 1 and 2 each had 1 session in the past 30 days while user 3 had 2 sessio
```

Solution

sql

```
SELECT IFNULL(ROUND(AVG(a.num),2),0) AS average_sessions_per_user
FROM (
SELECT COUNT(DISTINCT session_id) AS num
FROM activity
WHERE activity_date BETWEEN '2019-06-28' AND '2019-07-27'
GROUP BY user_id) a
```

1148. Article Views I | Easy | LeetCode

Table: **Views**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| article_id    | int     |
| author_id     | int     |
| viewer_id     | int     |
| view_date     | date    |
+---------------+---------+
There is no primary key for this table, it may have duplicate rows.
Each row of this table indicates that some viewer viewed an article (written
Note that equal author_id and viewer_id indicate the same person.
```

Write an SQL query to find all the authors that viewed at least one of their own articles, sorted in ascending order by their id.

The query result format is in the following example:

```
Views table:
+------------+-----------+-----------+------------+
| article_id | author_id | viewer_id | view_date  |
+------------+-----------+-----------+------------+
| 1          | 3         | 5         | 2019-08-01 |
| 1          | 3         | 6         | 2019-08-02 |
| 2          | 7         | 7         | 2019-08-01 |
| 2          | 7         | 6         | 2019-08-02 |
| 4          | 7         | 1         | 2019-07-22 |
```

```
| 3          | 4         | 4         | 2019-07-21 |
| 3          | 4         | 4         | 2019-07-21 |
+------------+-----------+-----------+------------+
```

```
Result table:
+------+
| id   |
+------+
| 4    |
| 7    |
+------+
```

Solution

sql

```sql
SELECT DISTINCT author_id AS id
FROM Views
WHERE author_id = viewer_id
ORDER BY author_id
```

1149. Article Views II | Medium | LeetCode

Table: **Views**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| article_id    | int     |
| author_id     | int     |
| viewer_id     | int     |
| view_date     | date    |
+---------------+---------+
There is no primary key for this table, it may have duplicate rows.
Each row of this table indicates that some viewer viewed an article (written
Note that equal author_id and viewer_id indicate the same person.
```

Write an SQL query to find all the people who viewed more than one article on the same [redacted]e, sorted in ascending order by their id.

The query result format is in the following example:

```
Views table:
+------------+-----------+-----------+------------+
| article_id | author_id | viewer_id | view_date  |
+------------+-----------+-----------+------------+
| 1          | 3         | 5         | 2019-08-01 |
| 3          | 4         | 5         | 2019-08-01 |
| 1          | 3         | 6         | 2019-08-02 |
| 2          | 7         | 7         | 2019-08-01 |
| 2          | 7         | 6         | 2019-08-02 |
| 4          | 7         | 1         | 2019-07-22 |
| 3          | 4         | 4         | 2019-07-21 |
| 3          | 4         | 4         | 2019-07-21 |
+------------+-----------+-----------+------------+

Result table:
+------+
| id   |
+------+
| 5    |
| 6    |
+------+
```

Solution

sql

```sql
SELECT DISTINCT viewer_id AS id#, COUNT(DISTINCT article_id) AS total
FROM views
GROUP BY viewer_id, view_date
HAVING count(DISTINCT article_id)>1
ORDER BY 1
```

1158. Market Analysis I | Medium | LeetCode

Table: **Users**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| join_date     | date    |
| favorite_brand | varchar |
+---------------+---------+
user_id is the primary key of this table.
This table has the info of the users of an online shopping website where user
```

Table: **Orders**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| order_date    | date    |
| item_id       | int     |
| buyer_id      | int     |
| seller_id     | int     |
+---------------+---------+
order_id is the primary key of this table.
item_id is a foreign key to the Items table.
buyer_id and seller_id are foreign keys to the Users table.
```

Table: **Items**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| item_id       | int     |
| item_brand    | varchar |
+---------------+---------+
item_id is the primary key of this table.
```

Write an SQL query to find for each user, the join date and the number of orders they made as a buyer in 2019.

The query result format is in the following example:

```
Users table:
+---------+------------+---------------+
| user_id | join_date  | favorite_brand |
+---------+------------+---------------+
| 1       | 2018-01-01 | Lenovo         |
| 2       | 2018-02-09 | Samsung        |
| 3       | 2018-01-19 | LG             |
| 4       | 2018-05-21 | HP             |
+---------+------------+---------------+

Orders table:
+----------+------------+---------+----------+-----------+
| order_id | order_date | item_id | buyer_id | seller_id |
+----------+------------+---------+----------+-----------+
| 1        | 2019-08-01 | 4       | 1        | 2         |
| 2        | 2018-08-02 | 2       | 1        | 3         |
| 3        | 2019-08-03 | 3       | 2        | 3         |
| 4        | 2018-08-04 | 1       | 4        | 2         |
| 5        | 2018-08-04 | 1       | 3        | 4         |
| 6        | 2019-08-05 | 2       | 2        | 4         |
+----------+------------+---------+----------+-----------+

Items table:
+---------+------------+
| item_id | item_brand |
+---------+------------+
| 1       | Samsung    |
| 2       | Lenovo     |
| 3       | LG         |
| 4       | HP         |
+---------+------------+

Result table:
+-----------+------------+----------------+
| buyer_id  | join_date  | orders_in_2019 |
+-----------+------------+----------------+
```

```
| 1          | 2018-01-01 | 1              |
| 2          | 2018-02-09 | 2              |
| 3          | 2018-01-19 | 0              |
| 4          | 2018-05-21 | 0              |
+------------+------------+----------------+
```

Solution

sql

```sql
SELECT user_id AS buyer_id, join_date, coalesce(a.orders_in_2019,0)
FROM users
LEFT JOIN
(
SELECT buyer_id, coalesce(count(*), 0) AS orders_in_2019
FROM orders o
JOIN users u
ON u.user_id = o.buyer_id
WHERE extract('year' FROM order_date) = 2019
GROUP BY buyer_id) a
ON users.user_id = a.buyer_id
```

1159. Market Analysis II | Hard | LeetCode

Table: **Users**

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| user_id        | int     |
| join_date      | date    |
| favorite_brand | varchar |
+----------------+---------+
user_id is the primary key of this table.
This table has the info of the users of an online shopping website where user
```

Table: **Orders**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| order_date    | date    |
| item_id       | int     |
| buyer_id      | int     |
| seller_id     | int     |
+---------------+---------+
order_id is the primary key of this table.
item_id is a foreign key to the Items table.
buyer_id and seller_id are foreign keys to the Users table.
```

Table: **Items**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| item_id       | int     |
| item_brand    | varchar |
+---------------+---------+
item_id is the primary key of this table.
```

Write an SQL query to find for each user, whether the brand of the second item (by date) they sold is their favorite brand. If a user sold less than two items, report the answer for that user as no.

It is guaranteed that no seller sold more than one item on a day.

The query result format is in the following example:

```
Users table:
+---------+------------+----------------+
| user_id | join_date  | favorite_brand |
+---------+------------+----------------+
| 1       | 2019-01-01 | Lenovo         |
| 2       | 2019-02-09 | Samsung        |
| 3       | 2019-01-19 | LG             |
| 4       | 2019-05-21 | HP             |
```

```
+---------+------------+---------------+
```

Orders table:

```
+----------+------------+---------+----------+-----------+
| order_id | order_date | item_id | buyer_id | seller_id |
+----------+------------+---------+----------+-----------+
| 1        | 2019-08-01 | 4       | 1        | 2         |
| 2        | 2019-08-02 | 2       | 1        | 3         |
| 3        | 2019-08-03 | 3       | 2        | 3         |
| 4        | 2019-08-04 | 1       | 4        | 2         |
| 5        | 2019-08-04 | 1       | 3        | 4         |
| 6        | 2019-08-05 | 2       | 2        | 4         |
+----------+------------+---------+----------+-----------+
```

Items table:

```
+---------+------------+
| item_id | item_brand |
+---------+------------+
| 1       | Samsung    |
| 2       | Lenovo     |
| 3       | LG         |
| 4       | HP         |
+---------+------------+
```

Result table:

```
+-----------+--------------------+
| seller_id | 2nd_item_fav_brand |
+-----------+--------------------+
| 1         | no                 |
| 2         | yes                |
| 3         | yes                |
| 4         | no                 |
+-----------+--------------------+
```

The answer for the user with id 1 is no because they sold nothing.
The answer for the users with id 2 and 3 is yes because the brands of their s
The answer for the user with id 4 is no because the brand of their second so

lution

sql

```
#Solution- 1:
SELECT user_id AS seller_id,
       IF(ISNULL(item_brand), "no", "yes") AS 2nd_item_fav_brand
FROM Users LEFT JOIN
(SELECT seller_id, item_brand
FROM Orders INNER JOIN Items
ON Orders.item_id = Items.item_id
WHERE (seller_id, order_date) IN
(SELECT seller_id, MIN(order_date) AS order_date
 FROM Orders
 WHERE (seller_id, order_date) NOT IN
 (SELECT seller_id, MIN(order_date) FROM Orders GROUP BY seller_id)
GROUP BY seller_id)
 ) AS t
ON Users.user_id = t.seller_id and favorite_brand = item_brand

#Solution- 2:
WITH t1 AS(
SELECT user_id,
CASE WHEN favorite_brand = item_brand THEN "yes"
ELSE "no"
END AS 2nd_item_fav_brand
FROM users u LEFT JOIN
(SELECT o.item_id, seller_id, item_brand, RANK() OVER(PARTITION BY seller_id
FROM orders o join items i
USING (item_id)) a
ON u.user_id = a.seller_id
WHERE a.rk = 2)

SELECT u.user_id AS seller_id, COALESCE(2nd_item_fav_brand,"no") AS 2nd_item_
FROM users u LEFT JOIN t1
USING(user_id)
```

1164. Product Price at a Given Date | Medium | LeetCode

Table: **Products**

```
+---------------+---------+
```

```
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| new_price     | int     |
| change_date   | date    |
+---------------+---------+
(product_id, change_date) is the primary key of this table.
Each row of this table indicates that the price of some product was changed t
```

Write an SQL query to find the prices of all products on 2019-08-16. Assume the price of all products before any change is 10.

The query result format is in the following example:

```
Products table:
+------------+-----------+-------------+
| product_id | new_price | change_date |
+------------+-----------+-------------+
| 1          | 20        | 2019-08-14  |
| 2          | 50        | 2019-08-14  |
| 1          | 30        | 2019-08-15  |
| 1          | 35        | 2019-08-16  |
| 2          | 65        | 2019-08-17  |
| 3          | 20        | 2019-08-18  |
+------------+-----------+-------------+

Result table:
+------------+-------+
| product_id | price |
+------------+-------+
| 2          | 50    |
| 1          | 35    |
| 3          | 10    |
+------------+-------+
```

Solution

sql

```
#Solution- 1:
```

```
WITH t1 AS (
SELECT a.product_id, new_price
FROM(
SELECT product_id, max(change_date) AS date
FROM products
WHERE change_date<='2019-08-16'
GROUP BY product_id) a
JOIN products p
ON a.product_id = p.product_id AND a.date = p.change_date),

t2 AS (
SELECT distinct product_id
    FROM products)

SELECT t2.product_id, coalesce(new_price,10) AS price
FROM t2 LEFT JOIN t1
ON t2.product_id = t1.product_id
ORDER BY price DESC

#Solution- 2:
SELECT t1.product_id AS product_id, IF(ISNULL(t2.price), 10, t2.price) AS pr
FROM
  (SELECT distinct product_id
  FROM Products) AS t1 LEFT JOIN
  (SELECT product_id, new_price AS price
   FROM Products
   WHERE (product_id, change_date) in
      (SELECT product_id, max(change_date)
      FROM Products
      WHERE change_date <='2019-08-16'
      GROUP BY product_id)) AS t2
ON t1.product_id = t2.product_id
```

1173. Immediate Food Delivery I | Easy | LeetCode

Table: **Delivery**

```
+-----------------------------+---------+
| Column Name                 | Type    |
+-----------------------------+---------+
```

```
| delivery_id                 | int     |
| customer_id                 | int     |
| order_date                  | date    |
| customer_pref_delivery_date | date    |
+-----------------------------+---------+
delivery_id is the primary key of this table.
The table holds information about food delivery to customers that make orders
```

If the preferred delivery date of the customer is the same as the order date then the order is called immediate otherwise it's called scheduled.

Write an SQL query to find the percentage of immediate orders in the table, **rounded to 2 decimal places**.

The query result format is in the following example:

```
Delivery table:
+-------------+-------------+------------+-----------------------------+
| delivery_id | customer_id | order_date | customer_pref_delivery_date |
+-------------+-------------+------------+-----------------------------+
| 1           | 1           | 2019-08-01 | 2019-08-02                  |
| 2           | 5           | 2019-08-02 | 2019-08-02                  |
| 3           | 1           | 2019-08-11 | 2019-08-11                  |
| 4           | 3           | 2019-08-24 | 2019-08-26                  |
| 5           | 4           | 2019-08-21 | 2019-08-22                  |
| 6           | 2           | 2019-08-11 | 2019-08-13                  |
+-------------+-------------+------------+-----------------------------+

Result table:
+----------------------+
| immediate_percentage |
+----------------------+
| 33.33                |
+----------------------+
The orders with delivery id 2 and 3 are immediate while the others are schedu
```

Solution

sql

```sql
#Solution- 1:
SELECT
ROUND(SUM(CASE WHEN order_date=customer_pref_delivery_date THEN 1 ELSE 0 END)
FROM Delivery;

#Solution- 2:
SELECT
ROUND(avg(CASE WHEN order_date=customer_pref_delivery_date THEN 1 ELSE 0 END)
FROM delivery
```

1174. Immediate Food Delivery II | Medium | LeetCode

Table: **Delivery**

```
+-----------------------------+---------+
| Column Name                 | Type    |
+-----------------------------+---------+
| delivery_id                 | int     |
| customer_id                 | int     |
| order_date                  | date    |
| customer_pref_delivery_date | date    |
+-----------------------------+---------+
delivery_id is the primary key of this table.
The table holds information about food delivery to customers that make orders
```

If the preferred delivery date of the customer is the same as the order date then the order is called immediate otherwise it's called scheduled.

The first order of a customer is the order with the earliest order date that customer made. It is guaranteed that a customer has exactly one first order.

Write an SQL query to find the percentage of immediate orders in the first orders of all customers, rounded to 2 decimal places.

e query result format is in the following example:

Delivery table:

delivery_id	customer_id	order_date	customer_pref_delivery_date
1	1	2019-08-01	2019-08-02
2	2	2019-08-02	2019-08-02
3	1	2019-08-11	2019-08-12
4	3	2019-08-24	2019-08-24
5	3	2019-08-21	2019-08-22
6	2	2019-08-11	2019-08-13
7	4	2019-08-09	2019-08-09

Result table:

immediate_percentage
50.00

The customer id 1 has a first order with delivery id 1 and it is scheduled.
The customer id 2 has a first order with delivery id 2 and it is immediate.
The customer id 3 has a first order with delivery id 5 and it is scheduled.
The customer id 4 has a first order with delivery id 7 and it is immediate.
Hence, half the customers have immediate first orders.

Solution

sql

```sql
#Solution- 1:
SELECT ROUND(SUM(CASE WHEN order_date=customer_pref_delivery_date THEN 1 ELSE
FROM Delivery
WHERE (customer_id, order_date) IN
    (SELECT customer_id, MIN(order_date)
    FROM Delivery
    GROUP BY customer_id)
#Solution- 2:
SELECT ROUND(AVG(CASE WHEN order_date = customer_pref_delivery_date THEN 1 EL
immediate_percentage
FROM
(SELECT *,
```

```
 RANK() OVER(PARTITION BY customer_id ORDER BY order_date) AS rk
FROM delivery) a
WHERE a.rk=1
```

1179. Reformat Department Table | Easy | LeetCode

Table: **Department**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| revenue       | int     |
| month         | varchar |
+---------------+---------+
(id, month) is the primary key of this table.
The table has information about the revenue of each department per month.
The month has values in ["Jan","Feb","Mar","Apr","May","Jun","Jul","Aug","Sep
```

Write an SQL query to reformat the table such that there is a department id column and a revenue column for each month.

The query result format is in the following example:

```
Department table:
+------+---------+-------+
| id   | revenue | month |
+------+---------+-------+
| 1    | 8000    | Jan   |
| 2    | 9000    | Jan   |
| 3    | 10000   | Feb   |
| 1    | 7000    | Feb   |
| 1    | 6000    | Mar   |
+------+---------+-------+

Result table:
+------+-------------+-------------+-------------+-----+-------------+
| id   | Jan_Revenue | Feb_Revenue | Mar_Revenue | ... | Dec_Revenue |
```

```
+------+------------+------------+------------+-----+------------+
| 1    | 8000       | 7000       | 6000       | ... | null       |
| 2    | 9000       | null       | null       | ... | null       |
| 3    | null       | 10000      | null       | ... | null       |
+------+------------+------------+------------+-----+------------+
```

Note that the result table has 13 columns (1 for the department id + 12 for t

Solution

sql

```sql
SELECT id,
SUM(IF(month='Jan', revenue, NULL)) AS Jan_Revenue,
SUM(IF(month='Feb', revenue, NULL)) AS Feb_Revenue,
SUM(IF(month='Mar', revenue, NULL)) AS Mar_Revenue,
SUM(IF(month='Apr', revenue, NULL)) AS Apr_Revenue,
SUM(IF(month='May', revenue, NULL)) AS May_Revenue,
SUM(IF(month='Jun', revenue, NULL)) AS Jun_Revenue,
SUM(IF(month='Jul', revenue, NULL)) AS Jul_Revenue,
SUM(IF(month='Aug', revenue, NULL)) AS Aug_Revenue,
SUM(IF(month='Sep', revenue, NULL)) AS Sep_Revenue,
SUM(IF(month='Oct', revenue, NULL)) AS Oct_Revenue,
SUM(IF(month='Nov', revenue, NULL)) AS Nov_Revenue,
SUM(IF(month='Dec', revenue, NULL)) AS Dec_Revenue
FROM Department
Group BY id;
```

1193. Monthly Transactions I | Medium | LeetCode

Table: **Transactions**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| country       | varchar |
| state         | enum    |
| amount        | int     |
```

```
| trans_date    | date    |
+---------------+---------+
id is the primary key of this table.
The table has information about incoming transactions.
The state column is an enum of type ["approved", "declined"].
```

Write an SQL query to find for each month and country, the number of transactions and their total amount, the number of approved transactions and their total amount.

The query result format is in the following example:

```
Transactions table:
+------+---------+----------+--------+------------+
| id   | country | state    | amount | trans_date |
+------+---------+----------+--------+------------+
| 121  | US      | approved | 1000   | 2018-12-18 |
| 122  | US      | declined | 2000   | 2018-12-19 |
| 123  | US      | approved | 2000   | 2019-01-01 |
| 124  | DE      | approved | 2000   | 2019-01-07 |
+------+---------+----------+--------+------------+

Result table:
+----------+---------+-------------+----------------+--------------------+---
| month    | country | trans_count | approved_count | trans_total_amount | ap
+----------+---------+-------------+----------------+--------------------+---
| 2018-12  | US      | 2           | 1              | 3000               | 10
| 2019-01  | US      | 1           | 1              | 2000               | 20
| 2019-01  | DE      | 1           | 1              | 2000               | 20
+----------+---------+-------------+----------------+--------------------+---
```

Solution

sql

```sql
WITH t1 AS(
SELECT DATE_FORMAT(trans_date,'%Y-%m') AS month, country, COUNT(state) AS tra
FROM transactions
GROUP BY country, month(trans_date)),

t2 AS (
```

```
SELECT DATE_FORMAT(trans_date,'%Y-%m') AS month, country, COUNT(state) AS app
FROM transactions
WHERE state = 'approved'
GROUP BY country, month(trans_date))

SELECT t1.month, t1.country, COALESCE(t1.trans_count,0) AS trans_count, COALE
FROM t1 LEFT JOIN t2
ON t1.country = t2.country and t1.month = t2.month
```

1194. Tournament Winners | Hard | LeetCode

Table: **Players**

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| player_id   | int   |
| group_id    | int   |
+-------------+-------+
player_id is the primary key of this table.
Each row of this table indicates the group of each player.
```

Table: **Matches**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| match_id      | int     |
| first_player  | int     |
| second_player | int     |
| first_score   | int     |
| second_score  | int     |
+---------------+---------+
match_id is the primary key of this table.
Each row is a record of a match, first_player and second_player contain the p
first_score and second_score contain the number of points of the first_player
You may assume that, in each match, players belongs to the same group.
```

The winner in each group is the player who scored the maximum total points within the group. In the case of a tie, the lowest player_id wins.

Write an SQL query to find the winner in each group.

The query result format is in the following example:

Players table:

player_id	group_id
15	1
25	1
30	1
45	1
10	2
35	2
50	2
20	3
40	3

Matches table:

match_id	first_player	second_player	first_score	second_score
1	15	45	3	0
2	30	25	1	2
3	30	15	2	0
4	40	20	5	2
5	35	50	1	1

Result table:

group_id	player_id
1	15
2	35
3	40

Solution

sql

```sql
WITH t1 AS(
SELECT first_player, SUM(first_score) AS total
FROM
(SELECT first_player, first_score
FROM matches
UNION ALL
SELECT second_player, second_score
FROM matches) a
GROUP BY 1),

t2 AS(
SELECT *, COALESCE(total,0) AS score
FROM players p LEFT JOIN t1
ON p.player_id = t1.first_player)

SELECT group_id, player_id
FROM
(SELECT *, ROW_NUMBER() OVER(PARTITION BY group_id ORDER BY group_id, score [
FROM t2) b
WHERE b.rn = 1
```

1204. Last Person to Fit in the Elevator | Medium | LeetCode

Table: **Queue**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| person_id   | int     |
| person_name | varchar |
| weight      | int     |
| turn        | int     |
+-------------+---------+
person_id is the primary key column for this table.
This table has the information about all people waiting for an elevator.
```

```
The person_id and turn columns will contain all numbers from 1 to n, where n
```

The maximum weight the elevator can hold is 1000.

Write an SQL query to find the person_name of the last person who will fit in the elevator without exceeding the weight limit. It is guaranteed that the person who is first in the queue can fit in the elevator.

The query result format is in the following example:

```
Queue table
+-----------+-------------------+--------+------+
| person_id | person_name       | weight | turn |
+-----------+-------------------+--------+------+
| 5         | George Washington | 250    | 1    |
| 3         | John Adams        | 350    | 2    |
| 6         | Thomas Jefferson  | 400    | 3    |
| 2         | Will Johnliams    | 200    | 4    |
| 4         | Thomas Jefferson  | 175    | 5    |
| 1         | James Elephant    | 500    | 6    |
+-----------+-------------------+--------+------+

Result table
+-------------------+
| person_name       |
+-------------------+
| Thomas Jefferson  |
+-------------------+

Queue table is ordered by turn in the example for simplicity.
In the example George Washington(id 5), John Adams(id 3) and Thomas Jefferso
Thomas Jefferson(id 6) is the last person to fit in the elevator because he h
```

Solution

sql

```sql
WITH t1 AS
(
SELECT *,
```

```
    SUM(weight) OVER(ORDER BY turn) AS cum_weight
    FROM queue
    ORDER BY turn)

    SELECT t1.person_name
    FROM t1
    WHERE turn = (SELECT MAX(turn) FROM t1 WHERE t1.cum_weight<=1000)
```

1205. Monthly Transactions II | Medium | LeetCode

Table: **Transactions**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| country       | varchar |
| state         | enum    |
| amount        | int     |
| trans_date    | date    |
+---------------+---------+
id is the primary key of this table.
The table has information about incoming transactions.
The state column is an enum of type ["approved", "declined"].
```

Table: **Chargebacks**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| trans_id      | int     |
| charge_date   | date    |
+---------------+---------+
Chargebacks contains basic information regarding incoming chargebacks from s
trans_id is a foreign key to the id column of Transactions table.
Each chargeback corresponds to a transaction made previously even if they we
```

Write an SQL query to find for each month and country, the number of approved transactions and their total amount, the number of chargebacks and their total amount.

Note: In your query, given the month and country, ignore rows with all zeros.

The query result format is in the following example:

```
Transactions table:
+------+---------+----------+--------+------------+
| id   | country | state    | amount | trans_date |
+------+---------+----------+--------+------------+
| 101  | US      | approved | 1000   | 2019-05-18 |
| 102  | US      | declined | 2000   | 2019-05-19 |
| 103  | US      | approved | 3000   | 2019-06-10 |
| 104  | US      | approved | 4000   | 2019-06-13 |
| 105  | US      | approved | 5000   | 2019-06-15 |
+------+---------+----------+--------+------------+

Chargebacks table:
+------------+------------+
| trans_id   | trans_date |
+------------+------------+
| 102        | 2019-05-29 |
| 101        | 2019-06-30 |
| 105        | 2019-09-18 |
+------------+------------+

Result table:
+----------+---------+----------------+-----------------+--------------------
| month    | country | approved_count | approved_amount | chargeback_count
+----------+`---------+----------------+-----------------+--------------------
| 2019-05  | US      | 1              | 1000            | 1
| 2019-06  | US      | 3              | 12000           | 1
| 2019-09  | US      | 0              | 0               | 1
+----------+---------+----------------+-----------------+--------------------
```

Solution

sql

```
#Solution 1:
```

```
WITH t1 AS
(SELECT country, extract('month' FROM trans_date), state, COUNT(*) AS approve
FROM transactions
WHERE state = 'approved'
GROUP BY 1, 2, 3),
t2 AS(
SELECT t.country, extract('month' FROM c.trans_date), SUM(amount) AS chargeba
FROM chargebacks c LEFT JOIN transactions t
ON trans_id = id
GROUP BY t.country, extract('month' FROM c.trans_date)),

t3 AS(
SELECT t2.date_part, t2.country, COALESCE(approved_count,0) AS approved_count
FROM t2 LEFT JOIN t1
ON t2.date_part = t1.date_part AND t2.country = t1.country),

t4 AS(
SELECT t1.date_part, t1.country, COALESCE(approved_count,0) AS approved_count
FROM t2 RIGHT JOIN t1
ON t2.date_part = t1.date_part AND t2.country = t1.country)

SELECT *
FROM t3
UNION
SELECT *
FROM t4

#Solution 2:
SELECT month, country,
    SUM(CASE WHEN type='approved' THEN 1 ELSE 0 END) AS approved_count,
    SUM(CASE WHEN type='approved' THEN amount ELSE 0 END) AS approved_amount,
    SUM(CASE WHEN type='chargeback' THEN 1 ELSE 0 END) AS chargeback_count,
    SUM(CASE WHEN type='chargeback' THEN amount ELSE 0 END) AS chargeback_amo
FROM (
    (
    SELECT left(t.trans_date, 7) AS month, t.country, amount,'approved' AS ty
    FROM Transactions AS t
    WHERE state='approved'
    )
    UNION ALL (
    SELECT left(c.trans_date, 7) AS month, t.country, amount,'chargeback' AS
    FROM Transactions AS t JOIN Chargebacks AS c
```

```
    ON t.id = c.trans_id
    )
) AS tt
GROUP BY tt.month, tt.country

#Solution 3:
SELECT month, country,
    SUM(CASE WHEN type='approved' THEN count ELSE 0 END) AS approved_count,
    SUM(CASE WHEN type='approved' THEN amount ELSE 0 END) AS approved_amount,
    SUM(CASE WHEN type='chargeback' THEN count ELSE 0 END) AS chargeback_cou
    SUM(CASE WHEN type='chargeback' THEN amount ELSE 0 END) AS chargeback_am
FROM (
    (
    SELECT LEFT(t.trans_date, 7) AS month, t.country,
    COUNT(1) AS count, SUM(amount) AS amount,'approved' AS type
    FROM Transactions AS t LEFT JOIN Chargebacks AS c
    ON t.id = c.trans_id
    WHERE state='approved'
    GROUP BY LEFT(t.trans_date, 7), t.country
    )
    union (
    SELECT LEFT(c.trans_date, 7) AS month, t.country,
        COUNT(1) AS count, SUM(amount) AS amount,'chargeback' AS type
    FROM Transactions AS t JOIN Chargebacks AS c
    ON t.id = c.trans_id
    GROUP BY LEFT(c.trans_date, 7), t.country
    )
) AS tt
GROUP BY tt.month, tt.country
```

1211. Queries Quality and Percentage | Easy | LeetCode

Table: **Queries**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| query_name  | varchar |
| result      | varchar |
| position    | int     |
```

```
| rating      | int     |
+-------------+---------+
There is no primary key for this table, it may have duplicate rows.
This table contains information collected from some queries on a database.
The position column has a value from 1 to 500.
The rating column has a value from 1 to 5. Query with rating less than 3 is a
```

We define query `quality` as:

- The average of the ratio between query rating and its position.

We also define `poor query percentage` as:

- The percentage of all queries with rating less than 3.

Write an SQL query to find each `query_name`, the `quality` and `poor_query_percentage`.

Both `quality` and `poor_query_percentage` should be rounded to **2 decimal places**.

The query result format is in the following example:

```
Queries table:
+------------+-------------------+----------+--------+
| query_name | result            | position | rating |
+------------+-------------------+----------+--------+
| Dog        | Golden Retriever  | 1        | 5      |
| Dog        | German Shepherd   | 2        | 5      |
| Dog        | Mule              | 200      | 1      |
| Cat        | Shirazi           | 5        | 2      |
| Cat        | Siamese           | 3        | 3      |
| Cat        | Sphynx            | 7        | 4      |
+------------+-------------------+----------+--------+

Result table:
+------------+---------+-----------------------+
| query_name | quality | poor_query_percentage |
+------------+---------+-----------------------+
| Dog        | 2.50    | 33.33                 |
| Cat        | 0.66    | 33.33                 |
+------------+---------+-----------------------+
```

```
Dog queries quality is ((5 / 1) + (5 / 2) + (1 / 200)) / 3 = 2.50
Dog queries poor_ query_percentage is (1 / 3) * 100 = 33.33

Cat queries quality equals ((2 / 5) + (3 / 3) + (4 / 7)) / 3 = 0.66
Cat queries poor_ query_percentage is (1 / 3) * 100 = 33.33
```

Solution

sql

```sql
#Solution 1:
SELECT query_name, ROUND(SUM(rating/position)/COUNT(*),2) AS quality,
ROUND(AVG(CASE WHEN rating<3 THEN 1 ELSE 0 END)*100,2) AS poor_query_percenta
FROM queries
GROUP BY query_name

#Solution 2:
SELECT query_name, ROUND(AVG(rating/position), 2) AS quality,
       ROUND(100*SUM(CASE WHEN rating<3 THEN 1 ELSE 0 END)/COUNT(1), 2) AS p
FROM Queries
GROUP BY query_name
```

1212. Team Scores in Football Tournament | Medium | LeetCode

Table: **Teams**

```
+---------------+----------+
| Column Name   | Type     |
+---------------+----------+
| team_id       | int      |
| team_name     | varchar  |
+---------------+----------+
team_id is the primary key of this table.
Each row of this table represents a single football team.
```

Table: **Matches**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| match_id      | int     |
| host_team     | int     |
| guest_team    | int     |
| host_goals    | int     |
| guest_goals   | int     |
+---------------+---------+
match_id is the primary key of this table.
Each row is a record of a finished match between two different teams.
Teams host_team and guest_team are represented by their IDs in the teams tab
```

You would like to compute the scores of all teams after all matches. Points are awarded as follows:

- A team receives three points if they win a match (Score strictly more goals than the opponent team).
- A team receives one point if they draw a match (Same number of goals as the opponent team).
- A team receives no points if they lose a match (Score less goals than the opponent team).

  Write an SQL query that selects the team_id, team_name and num_points of each team in the tournament after all described matches. Result table should be ordered by num_points (decreasing order). In case of a tie, order the records by team_id (increasing order).

The query result format is in the following example:

```
Teams table:
+-----------+--------------+
| team_id   | team_name    |
+-----------+--------------+
| 10        | Leetcode FC  |
| 20        | NewYork FC   |
| 30        | Atlanta FC   |
| 40        | Chicago FC   |
```

```
| 50        | Toronto FC   |
+-----------+--------------+
```

```
Matches table:
+------------+--------------+---------------+-------------+--------------+
| match_id   | host_team    | guest_team    | host_goals  | guest_goals  |
+------------+--------------+---------------+-------------+--------------+
| 1          | 10           | 20            | 3           | 0            |
| 2          | 30           | 10            | 2           | 2            |
| 3          | 10           | 50            | 5           | 1            |
| 4          | 20           | 30            | 1           | 0            |
| 5          | 50           | 30            | 1           | 0            |
+------------+--------------+---------------+-------------+--------------+

Result table:
+------------+--------------+---------------+
| team_id    | team_name    | num_points    |
+------------+--------------+---------------+
| 10         | Leetcode FC  | 7             |
| 20         | NewYork FC   | 3             |
| 50         | Toronto FC   | 3             |
| 30         | Atlanta FC   | 1             |
| 40         | Chicago FC   | 0             |
+------------+--------------+---------------+
```

Solution

sql

```sql
#Solution 1:
SELECT Teams.team_id, Teams.team_name,
    SUM(CASE WHEN team_id=host_team AND host_goals>guest_goals THEN 3 ELSE 0
    SUM(CASE WHEN team_id=host_team AND host_goals=guest_goals THEN 1 ELSE 0
    SUM(CASE WHEN team_id=guest_team AND host_goals<guest_goals THEN 3 ELSE (
    SUM(CASE WHEN team_id=guest_team AND host_goals=guest_goals THEN 1 ELSE (
FROM Teams LEFT JOIN Matches
ON Teams.team_id = Matches.host_team OR Teams.team_id = Matches.guest_team
GROUP BY Teams.team_id
ORDER BY num_points DESC, Teams.team_id ASC

#Solution 2:
SELECT Teams.team_id, Teams.team_name, SUM(if(isnull(num_points), 0, num_poi
```

```
FROM Teams LEFT JOIN
    (
        SELECT host_team AS team_id,
            SUM(CASE WHEN host_goals>guest_goals THEN 3
                     WHEN host_goals=guest_goals THEN 1
                     ELSE 0 END) AS num_points
        FROM Matches
        GROUP BY host_team
        UNION ALL
        SELECT guest_team AS team_id,
            SUM(CASE WHEN host_goals<guest_goals THEN 3
                     WHEN host_goals=guest_goals THEN 1
                     ELSE 0 END) AS num_points
        FROM Matches
        GROUP BY guest_team
    ) AS tt
ON Teams.team_id = tt.team_id
GROUP BY Teams.team_id
ORDER BY num_points DESC, Teams.team_id ASC

#Solution 3:
SELECT Teams.team_id, Teams.team_name, IFNULL(SUM(num_points), 0) AS num_poi
FROM Teams LEFT JOIN
    (
        SELECT host_team AS team_id,
            SUM(CASE WHEN host_goals>guest_goals THEN 3
                     WHEN host_goals=guest_goals THEN 1
                     ELSE 0 END) AS num_points
        FROM Matches
        GROUP BY host_team
        UNION ALL
        SELECT guest_team AS team_id,
            SUM(CASE WHEN host_goals<guest_goals THEN 3
                     WHEN host_goals=guest_goals THEN 1
                     ELSE 0 END) AS num_points
        FROM Matches
        GROUP BY guest_team
    ) AS tt
ON Teams.team_id = tt.team_id
GROUP BY Teams.team_id
ORDER BY num_points DESC, Teams.team_id ASC
```

```sql
#Solution 4:
WITH t1 AS(
SELECT c.host_id, c.host_name, c.host_points
FROM(
SELECT a.match_id, a.team_id AS host_id, a.team_name AS host_name, b.team_id
CASE WHEN a.host_goals > a.guest_goals THEN 3
     WHEN a.host_goals = a.guest_goals THEN 1
     ELSE 0 END AS host_points,
CASE WHEN a.host_goals < a.guest_goals THEN 3
     WHEN a.host_goals = a.guest_goals THEN 1
     ELSE 0 END AS guest_points
FROM(
SELECT *
FROM matches m
JOIN teams t
ON t.team_id = m.host_team) a
JOIN
(SELECT *
FROM matches m
JOIN teams t
ON t.team_id = m.guest_team) b
ON a.match_id = b.match_id) c
UNION ALL
SELECT d.guest_id, d.guest_name, d.guest_points
FROM(
SELECT a.match_id, a.team_id AS host_id, a.team_name AS host_name, b.team_id
CASE WHEN a.host_goals > a.guest_goals THEN 3
     WHEN a.host_goals = a.guest_goals THEN 1
     ELSE 0 END AS host_points,
CASE WHEN a.host_goals < a.guest_goals THEN 3
     WHEN a.host_goals = a.guest_goals THEN 1
     ELSE 0 END AS guest_points
FROM(
SELECT *
FROM matches m
JOIN teams t
ON t.team_id = m.host_team) a
JOIN
(SELECT *
FROM matches m
JOIN teams t
ON t.team_id = m.guest_team) b
```

```
ON a.match_id = b.match_id) d)

SELECT team_id, team_name, coalesce(total,0) AS num_points
FROM teams t2
LEFT JOIN(
SELECT host_id, host_name, SUM(host_points) AS total
FROM t1
GROUP BY host_id, host_name) e
ON t2.team_id = e.host_id
ORDER BY num_points DESC, team_id
```

1225. Report Contiguous Dates | Hard | LeetCode

Table: **Failed**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| fail_date    | date    |
+--------------+---------+
Primary key for this table is fail_date.
Failed table contains the days of failed tasks.
```

Table: **Succeeded**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| success_date | date    |
+--------------+---------+
Primary key for this table is success_date.
Succeeded table contains the days of succeeded tasks.
```

A system is running one task every day. Every task is independent of the previous tasks.
e tasks can fail or succeed.

Write an SQL query to generate a report of period_state for each continuous interval of days in the period from 2019-01-01 to 2019-12-31.

period_state is 'failed' if tasks in this interval failed or 'succeeded' if tasks in this interval succeeded. Interval of days are retrieved as start_date and end_date.

Order result by start_date.

The query result format is in the following example:

```
Failed table:
+-------------------+
| fail_date         |
+-------------------+
| 2018-12-28        |
| 2018-12-29        |
| 2019-01-04        |
| 2019-01-05        |
+-------------------+

Succeeded table:
+-------------------+
| success_date      |
+-------------------+
| 2018-12-30        |
| 2018-12-31        |
| 2019-01-01        |
| 2019-01-02        |
| 2019-01-03        |
| 2019-01-06        |
+-------------------+


Result table:
+--------------+--------------+--------------+
| period_state | start_date   | end_date     |
+--------------+--------------+--------------+
| succeeded    | 2019-01-01   | 2019-01-03   |
| failed       | 2019-01-04   | 2019-01-05   |
| succeeded    | 2019-01-06   | 2019-01-06   |
+--------------+--------------+--------------+
```

```
The report ignored the system state in 2018 as we care about the system in th
From 2019-01-01 to 2019-01-03 all tasks succeeded and the system state was "s
From 2019-01-04 to 2019-01-05 all tasks failed and system state was "failed".
From 2019-01-06 to 2019-01-06 all tasks succeeded and system state was "succe
```

Solution

sql

```sql
#Solution 1:
WITH t1 AS(
SELECT MIN(success_date) AS start_date, MAX(success_date) AS end_date, state
FROM(
SELECT *, date_sub(success_date, interval ROW_NUMBER() OVER(ORDER BY success_
FROM succeeded
WHERE success_date BETWEEN "2019-01-01" AND "2019-12-31") a
GROUP BY diff),

t2 AS(
SELECT MIN(fail_date) AS start_date, MAX(fail_date) AS end_date, state
FROM(
SELECT *, date_sub(fail_date, interval ROW_NUMBER() OVER(ORDER BY fail_date)
FROM failed
WHERE fail_date BETWEEN "2019-01-01" AND "2019-12-31") b
GROUP BY diff)


SELECT
CASE WHEN c.state = 1 THEN "succeeded"
ELSE "failed"
END AS period_state,start_date, end_date
FROM(
SELECT *
FROM t1

UNION ALL

SELECT *
FROM t2) c
ORDER BY start_date
```

```
#Solution 2:
## First generate a list of dates
##   succeeded 2019-01-01
##   succeeded 2019-01-02
##   ...
##   failed 2019-01-04
##   ...
## Add group id for contiguous ranges
## Notice: dates themselves are contiguous
##
SELECT period_state, MIN(date) AS start_date, MAX(date) AS end_date
FROM (
    SELECT period_state, date,
         @rank := CASE WHEN @prev = period_state THEN @rank ELSE @rank+1 END
         @prev := period_state AS prev
    FROM (
        SELECT 'failed' AS period_state, fail_date AS date
        FROM Failed
        WHERE fail_date BETWEEN '2019-01-01' AND '2019-12-31'
        UNION
        SELECT 'succeeded' AS period_state, success_date AS date
        FROM Succeeded
        WHERE success_date BETWEEN '2019-01-01' AND '2019-12-31') AS t,
        (SELECT @rank:=0, @prev:='') AS rows
    ORDER BY date ASC) AS tt
GROUP BY rank
ORDER BY rank
```

1241. Number of Comments per Post | Easy | LeetCode

Table: **Submissions**

```
+---------------+----------+
| Column Name   | Type     |
+---------------+----------+
| sub_id        | int      |
| parent_id     | int      |
+---------------+----------+
There is no primary key for this table, it may have duplicate rows.
Each row can be a post or comment on the post.
```

```
parent_id is null for posts.
parent_id for comments is sub_id for another post in the table.
```

Write an SQL query to find number of comments per each post.

Result table should contain `post_id` and its corresponding `number_of_comments`, and must be sorted by `post_id` in ascending order.

`Submissions` may contain duplicate comments. You should count the number of unique comments per post.

`Submissions` may contain duplicate posts. You should treat them as one post.

The query result format is in the following example:

```
Submissions table:
+---------+------------+
| sub_id  | parent_id  |
+---------+------------+
| 1       | Null       |
| 2       | Null       |
| 1       | Null       |
| 12      | Null       |
| 3       | 1          |
| 5       | 2          |
| 3       | 1          |
| 4       | 1          |
| 9       | 1          |
| 10      | 2          |
| 6       | 7          |
+---------+------------+

Result table:
+---------+--------------------+
| post_id | number_of_comments |
+---------+--------------------+
| 1       | 3                  |
| 2       | 2                  |
| 12      | 0                  |
+---------+--------------------+
```

```
The post with id 1 has three comments in the table with id 3, 4 and 9. The c
The post with id 2 has two comments in the table with id 5 and 10.
The post with id 12 has no comments in the table.
The comment with id 6 is a comment on a deleted post with id 7 so we ignored
```

Solution

sql

```sql
SELECT a.sub_id AS post_id, coalesce(b.number_of_comments,0) AS number_of_cor
FROM(
SELECT DISTINCT sub_id FROM submissions WHERE parent_id IS NULL) a
LEFT JOIN(
SELECT parent_id, count(DISTINCT(sub_id)) AS number_of_comments
FROM submissions
GROUP BY parent_id
HAVING parent_id = any(SELECT sub_id from submissions WHERE parent_id IS NULL
ON a.sub_id = b.parent_id
ORDER BY post_id
```

1251. Average Selling Price | Easy | LeetCode

Table: **Prices**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| start_date    | date    |
| end_date      | date    |
| price         | int     |
+---------------+---------+
(product_id, start_date, end_date) is the primary key for this table.
Each row of this table indicates the price of the product_id in the period f
For each product_id there will be no two overlapping periods. That means the
```

Table: **UnitsSold**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| purchase_date | date   |
| units        | int     |
+--------------+---------+
There is no primary key for this table, it may contain duplicates.
Each row of this table indicates the date, units and product_id of each produ
```

Write an SQL query to find the average selling price for each product.

average_price should be rounded to 2 decimal places.

The query result format is in the following example:

```
Prices table:
+------------+------------+------------+--------+
| product_id | start_date | end_date   | price  |
+------------+------------+------------+--------+
| 1          | 2019-02-17 | 2019-02-28 | 5      |
| 1          | 2019-03-01 | 2019-03-22 | 20     |
| 2          | 2019-02-01 | 2019-02-20 | 15     |
| 2          | 2019-02-21 | 2019-03-31 | 30     |
+------------+------------+------------+--------+

UnitsSold table:
+------------+---------------+-------+
| product_id | purchase_date | units |
+------------+---------------+-------+
| 1          | 2019-02-25    | 100   |
| 1          | 2019-03-01    | 15    |
| 2          | 2019-02-10    | 200   |
| 2          | 2019-03-22    | 30    |
+------------+---------------+-------+

Result table:
+------------+--------------+
| product_id | average_price |
+------------+--------------+
```

```
| 1           | 6.96          |
| 2           | 16.96         |
+-------------+---------------+
```

```
Average selling price = Total Price of Product / Number of products sold.
Average selling price for product 1 = ((100 * 5) + (15 * 20)) / 115 = 6.96
Average selling price for product 2 = ((200 * 15) + (30 * 30)) / 230 = 16.96
```

Solution

sql

```sql
SELECT UnitsSold.product_id, ROUND(SUM(units*price)/SUM(units), 2) AS average
FROM UnitsSold INNER JOIN Prices
ON UnitsSold.product_id = Prices.product_id
AND UnitsSold.purchase_date BETWEEN Prices.start_date AND Prices.end_date
GROUP BY UnitsSold.product_id
```

1264. Page Recommendations | Medium | LeetCode

Table: **Friendship**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user1_id      | int     |
| user2_id      | int     |
+---------------+---------+
(user1_id, user2_id) is the primary key for this table.
Each row of this table indicates that there is a friendship relation between
```

Table: **Likes**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| user_id     | int     |
| page_id     | int     |
+-------------+---------+
```

```
(user_id, page_id) is the primary key for this table.
Each row of this table indicates that user_id likes page_id.
```

Write an SQL query to recommend pages to the user with `user_id` = 1 using the pages that your friends liked. It should not recommend pages you already liked.

Return result table in any order without duplicates.

The query result format is in the following example:

```
Friendship table:
+----------+----------+
| user1_id | user2_id |
+----------+----------+
| 1        | 2        |
| 1        | 3        |
| 1        | 4        |
| 2        | 3        |
| 2        | 4        |
| 2        | 5        |
| 6        | 1        |
+----------+----------+

Likes table:
+---------+---------+
| user_id | page_id |
+---------+---------+
| 1       | 88      |
| 2       | 23      |
| 3       | 24      |
| 4       | 56      |
| 5       | 11      |
| 6       | 33      |
| 2       | 77      |
| 3       | 77      |
| 6       | 88      |
+---------+---------+

Result table:
+-----------------+
| recommended_page |
```

```
+------------------+
| 23               |
| 24               |
| 56               |
| 33               |
| 77               |
+------------------+
User one is friend with users 2, 3, 4 and 6.
Suggested pages are 23 from user 2, 24 from user 3, 56 from user 3 and 33 fr
Page 77 is suggested from both user 2 and user 3.
Page 88 is not suggested because user 1 already likes it.
```

Solution

sql

```sql
SELECT DISTINCT page_id AS recommended_page
FROM Likes
WHERE user_id IN (SELECT user2_id
    FROM Friendship
    WHERE user1_id=1
    UNION
    SELECT user1_id
    FROM Friendship
    WHERE user2_id=1)
    AND page_id NOT IN
        (SELECT page_id
        FROM Likes
        WHERE user_id=1)
```

1270. All People Report to the Given Manager | Medium | LeetCode

Table: **Employees**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| employee_id   | int     |
```

```
| employee_name | varchar |
| manager_id    | int     |
+---------------+---------+
employee_id is the primary key for this table.
Each row of this table indicates that the employee with ID employee_id and na
The head of the company is the employee with employee_id = 1.
```

Write an SQL query to find `employee_id` of all employees that directly or indirectly report their work to the head of the company.

The indirect relation between managers will not exceed 3 managers as the company is small.

Return result table in any order without duplicates.

The query result format is in the following example:

```
Employees table:
+-------------+---------------+------------+
| employee_id | employee_name | manager_id |
+-------------+---------------+------------+
| 1           | Boss          | 1          |
| 3           | Alice         | 3          |
| 2           | Bob           | 1          |
| 4           | Daniel        | 2          |
| 7           | Luis          | 4          |
| 8           | Jhon          | 3          |
| 9           | Angela        | 8          |
| 77          | Robert        | 1          |
+-------------+---------------+------------+

Result table:
+-------------+
| employee_id |
+-------------+
| 2           |
| 77          |
| 4           |
| 7           |
+-------------+
```

```
The head of the company is the employee with employee_id 1.
The employees with employee_id 2 and 77 report their work directly to the hea
The employee with employee_id 4 report his work indirectly to the head of the
The employee with employee_id 7 report his work indirectly to the head of the
The employees with employee_id 3, 8 and 9 don't report their work to head of
```

Solution

sql

```sql
#Solution 1:
## t3: directly report to employee_id 1
## t2: directly report to t3
## t1: directly report to t2
SELECT t1.employee_id
FROM Employees AS t1 INNER JOIN Employees AS t2
ON t1.manager_id = t2.employee_id
JOIN Employees AS t3
ON t2.manager_id = t3.employee_id
WHERE t3.manager_id = 1 AND t1.employee_id != 1

#Solution 2:
SELECT distinct employee_id
FROM (
SELECT employee_id
FROM Employees
WHERE manager_id IN
(SELECT employee_id
FROM Employees
WHERE manager_id IN
    (SELECT employee_id
    FROM Employees
    WHERE manager_id = 1))
UNION
SELECT employee_id
FROM Employees
WHERE manager_id IN
    (SELECT employee_id
    FROM Employees
    WHERE manager_id = 1)
UNION
SELECT employee_id
```

```
    FROM Employees
    WHERE manager_id = 1) AS t
WHERE employee_id != 1
```

```
#Solution 3:
SELECT employee_id
FROM employees
WHERE manager_id = 1 AND employee_id != 1
UNION
SELECT employee_id
FROM employees
WHERE manager_id = any (SELECT employee_id
FROM employees
WHERE manager_id = 1 AND employee_id != 1)
UNION
SELECT employee_id
FROM employees
WHERE manager_id = any (SELECT employee_id
FROM employees
WHERE manager_id = any (SELECT employee_id
FROM employees
WHERE manager_id = 1 AND employee_id != 1))
```

1280. Students and Examinations | Easy | LeetCode

Table: **Students**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| student_id    | int     |
| student_name  | varchar |
+---------------+---------+
student_id is the primary key for this table.
Each row of this table contains the ID and the name of one student in the sc
```

ble: **Subjects**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| subject_name | varchar |
+--------------+---------+
subject_name is the primary key for this table.
Each row of this table contains a name of one subject in the school.
```

Table: **Examinations**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| student_id   | int     |
| subject_name | varchar |
+--------------+---------+
There is no primary key for this table. It may contain duplicates.
Each student from Students table takes every course from Subjects table.
Each row of this table indicates that a student with ID student_id attended t
```

Write an SQL query to find the number of times each student attended each exam.

Order the result table by student_id and subject_name.

The query result format is in the following example:

```
Students table:
+------------+--------------+
| student_id | student_name |
+------------+--------------+
| 1          | Alice        |
| 2          | Bob          |
| 13         | John         |
| 6          | Alex         |
+------------+--------------+
Subjects table:
+--------------+
| subject_name |
+--------------+
```

```
| Math         |
| Physics      |
| Programming  |
+--------------+
```

Examinations table:

```
+------------+--------------+
| student_id | subject_name |
+------------+--------------+
| 1          | Math         |
| 1          | Physics      |
| 1          | Programming  |
| 2          | Programming  |
| 1          | Physics      |
| 1          | Math         |
| 13         | Math         |
| 13         | Programming  |
| 13         | Physics      |
| 2          | Math         |
| 1          | Math         |
+------------+--------------+
```

Result table:

```
+------------+--------------+--------------+----------------+
| student_id | student_name | subject_name | attended_exams |
+------------+--------------+--------------+----------------+
| 1          | Alice        | Math         | 3              |
| 1          | Alice        | Physics      | 2              |
| 1          | Alice        | Programming  | 1              |
| 2          | Bob          | Math         | 1              |
| 2          | Bob          | Physics      | 0              |
| 2          | Bob          | Programming  | 1              |
| 6          | Alex         | Math         | 0              |
| 6          | Alex         | Physics      | 0              |
| 6          | Alex         | Programming  | 0              |
| 13         | John         | Math         | 1              |
| 13         | John         | Physics      | 1              |
| 13         | John         | Programming  | 1              |
+------------+--------------+--------------+----------------+
```

The result table should contain all students and all subjects.
Alice attended Math exam 3 times, Physics exam 2 times and Programming exam 1
Bob attended Math exam 1 time, Programming exam 1 time and didn't attend the
Alex didn't attend any exam.
John attended Math exam 1 time, Physics exam 1 time and Programming exam 1 t

Solution

sql

```sql
#Solution 1: count with null
SELECT Students.student_id, student_name, Subjects.subject_name, COUNT(Exami
FROM Students JOIN Subjects
LEFT JOIN Examinations
ON Students.student_id = Examinations.student_id AND Subjects.subject_name =
GROUP BY Students.student_id, subject_name

#Solution 2: using ISNULL
SELECT Students.student_id, student_name, Subjects.subject_name, SUM(IF(ISNUl
FROM Students JOIN Subjects
LEFT JOIN Examinations
ON Students.student_id = Examinations.student_id AND Subjects.subject_name =
GROUP BY Students.student_id, subject_name

#Solution 3: coalesce
SELECT a.student_id AS student_id, a.student_name AS student_name, a.subject_
FROM(
SELECT *
FROM students
CROSS JOIN subjects
GROUP BY student_id, student_name, subject_name) a
LEFT JOIN
(SELECT e.student_id, student_name, subject_name, COUNT(*) AS attended_exams
FROM examinations e JOIN students s
ON e.student_id = s.student_id
GROUP BY e.student_id, student_name, subject_name) b
ON a.student_id = b.student_id AND a.subject_name =b.subject_name
ORDER BY a.student_id ASC, a.subject_name ASC
```

1285. Find the Start and End Number of Continuous Ranges | Medium | LeetCode

Table: **Logs**

```
+--------------+---------+
| Column Name  | Type    |
```

```
+---------------+---------+
| log_id        | int     |
+---------------+---------+
id is the primary key for this table.
Each row of this table contains the ID in a log Table.
```

Since some IDs have been removed from Logs. Write an SQL query to find the start and end number of continuous ranges in table Logs.

Order the result table by start_id.

The query result format is in the following example:

```
Logs table:
+------------+
| log_id     |
+------------+
| 1          |
| 2          |
| 3          |
| 7          |
| 8          |
| 10         |
+------------+

Result table:
+------------+--------------+
| start_id   | end_id       |
+------------+--------------+
| 1          | 3            |
| 7          | 8            |
| 10         | 10           |
+------------+--------------+
The result table should contain all ranges in table Logs.
From 1 to 3 is contained in the table.
From 4 to 6 is missing in the table
From 7 to 8 is contained in the table.
Number 9 is missing in the table.
Number 10 is contained in the table.
```

Solution

sql

```sql
#Solution 1:
SELECT MIN(log_id) AS start_id, MAX(log_id) AS end_id
FROM(
SELECT log_id, log_id-ROW_NUMBER() OVER (ORDER BY log_id) AS rk
FROM logs) a
GROUP BY rk

#Solution 2: Add temporary columns of rank and prev
SELECT MIN(log_id) AS START_ID, MAX(log_id) AS END_ID
FROM (SELECT log_id,
        @rank := CASE WHEN @prev = log_id-1 THEN @rank ELSE @rank+1 END AS ra
        @prev := log_id AS prev
    FROM Logs,
       (SELECT @rank:=0, @prev:=-1) AS rows) AS tt
GROUP BY rank
ORDER BY START_ID

# Solution 3: Find the starting and ending sequences, then merge two AS one t
## find the starting sequence: 1, 7, 10
## find the ending sequence: 3, 8, 10
## merge them AS one table
SELECT start_id, MIN(end_id) AS end_id
FROM (SELECT t1.log_id AS start_id
    FROM logs AS t1 LEFT JOIN logs AS t2
    ON t1.log_id-1 = t2.log_id
    WHERE t2.log_id IS NULL) tt_start join
    (SELECT t1.log_id AS end_id
    FROM logs AS t1 LEFT JOIN logs AS t2
    ON t1.log_id+1 = t2.log_id
    WHERE t2.log_id IS NULL) tt_end
WHERE start_id<=end_id
GROUP BY start_id
```

1294. Weather Type in Each Country | Easy | LeetCode

Table: **Countries**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| country_id    | int     |
| country_name  | varchar |
+---------------+---------+
country_id is the primary key for this table.
Each row of this table contains the ID and the name of one country.
```

Table: **Weather**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| country_id    | int     |
| weather_state | varchar |
| day           | date    |
+---------------+---------+
(country_id, day) is the primary key for this table.
Each row of this table indicates the weather state in a country for one day.
```

Write an SQL query to find the type of weather in each country for November 2019.

The type of weather is Cold if the average weather_state is less than or equal 15, Hot if the average weather_state is greater than or equal 25 and Warm otherwise.

Return result table in any order.

The query result format is in the following example:

```
Countries table:
+------------+--------------+
| country_id | country_name |
+------------+--------------+
| 2          | USA          |
| 3          | Australia    |
| 7          | Peru         |
| 5          | China        |
| 8          | Morocco      |
```

| 9 | Spain |
+------------+--------------+

Weather table:

country_id	weather_state	day
2	15	2019-11-01
2	12	2019-10-28
2	12	2019-10-27
3	-2	2019-11-10
3	0	2019-11-11
3	3	2019-11-12
5	16	2019-11-07
5	18	2019-11-09
5	21	2019-11-23
7	25	2019-11-28
7	22	2019-12-01
7	20	2019-12-02
8	25	2019-11-05
8	27	2019-11-15
8	31	2019-11-25
9	7	2019-10-23
9	3	2019-12-23

Result table:

country_name	weather_type
USA	Cold
Austraila	Cold
Peru	Hot
China	Warm
Morocco	Hot

Average weather_state in USA in November is (15) / 1 = 15 so weather type is
Average weather_state in Austraila in November is (-2 + 0 + 3) / 3 = 0.333 s
Average weather_state in Peru in November is (25) / 1 = 25 so weather type is
Average weather_state in China in November is (16 + 18 + 21) / 3 = 18.333 so
Average weather_state in Morocco in November is (25 + 27 + 31) / 3 = 27.667 s
We know nothing about average weather_state in Spain in November so we don't

Solution

```sql
SELECT country_name, CASE WHEN AVG(weather_state) <= 15 THEN "Cold"
                          WHEN AVG(weather_state) >= 25 THEN "Hot"
                          ELSE "Warm" END AS weather_type
FROM Countries INNER JOIN Weather
ON Countries.country_id = Weather.country_id
WHERE MONTH(day) = 11
GORUP BY country_name
```

1303. Find the Team Size | Easy | LeetCode

Table: **Employee**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| employee_id   | int     |
| team_id       | int     |
+---------------+---------+
employee_id is the primary key for this table.
Each row of this table contains the ID of each employee and their respective
```

Write an SQL query to find the team size of each of the employees.

Return result table in any order.

The query result format is in the following example:

```
Employee Table:
+-------------+------------+
| employee_id | team_id    |
+-------------+------------+
|     1       |     8      |
|     2       |     8      |
|     3       |     8      |
```

```
|     4       |     7      |
|     5       |     9      |
|     6       |     9      |
+-------------+------------+
Result table:
+-------------+------------+
| employee_id | team_size  |
+-------------+------------+
|     1       |     3      |
|     2       |     3      |
|     3       |     3      |
|     4       |     1      |
|     5       |     2      |
|     6       |     2      |
+-------------+------------+
Employees with Id 1,2,3 are part of a team with team_id = 8.
Employees with Id 4 is part of a team with team_id = 7.
Employees with Id 5,6 are part of a team with team_id = 9.
```

Solution

sql

```sql
SELECT employee_id, b.team_size
FROM employee e
JOIN
(
SELECT team_id, count(team_id) AS team_size
FROM employee
GROUP BY team_id) b
ON e.team_id = b.team_id
```

1308. Running Total for Different Genders | Medium | LeetCode

Table: **Scores**

```
+---------------+---------+
| Column Name   | Type    |
```

```
+---------------+---------+
| player_name   | varchar |
| gender        | varchar |
| day           | date    |
| score_points  | int     |
+---------------+---------+
(gender, day) is the primary key for this table.
A competition is held between females team and males team.
Each row of this table indicates that a player_name and with gender has score
Gender is 'F' if the player is in females team and 'M' if the player is in ma
```

Write an SQL query to find the total score for each gender at each day.

Order the result table by gender and day

The query result format is in the following example:

```
Scores table:
+-------------+--------+------------+--------------+
| player_name | gender | day        | score_points |
+-------------+--------+------------+--------------+
| Aron        | F      | 2020-01-01 | 17           |
| Alice       | F      | 2020-01-07 | 23           |
| Bajrang     | M      | 2020-01-07 | 7            |
| Khali       | M      | 2019-12-25 | 11           |
| Slaman      | M      | 2019-12-30 | 13           |
| Joe         | M      | 2019-12-31 | 3            |
| Jose        | M      | 2019-12-18 | 2            |
| Priya       | F      | 2019-12-31 | 23           |
| Priyanka    | F      | 2019-12-30 | 17           |
+-------------+--------+------------+--------------+
Result table:
+--------+------------+-------+
| gender | day        | total |
+--------+------------+-------+
| F      | 2019-12-30 | 17    |
| F      | 2019-12-31 | 40    |
| F      | 2020-01-01 | 57    |
| F      | 2020-01-07 | 80    |
| M      | 2019-12-18 | 2     |
| M      | 2019-12-25 | 13    |
```

```
| M      | 2019-12-30 | 26    |
| M      | 2019-12-31 | 29    |
| M      | 2020-01-07 | 36    |
+--------+------------+-------+
For females team:
First day is 2019-12-30, Priyanka scored 17 points and the total score for th
Second day is 2019-12-31, Priya scored 23 points and the total score for the
Third day is 2020-01-01, Aron scored 17 points and the total score for the te
Fourth day is 2020-01-07, Alice scored 23 points and the total score for the
For males team:
First day is 2019-12-18, Jose scored 2 points and the total score for the tea
Second day is 2019-12-25, Khali scored 11 points and the total score for the
Third day is 2019-12-30, Slaman scored 13 points and the total score for the
Fourth day is 2019-12-31, Joe scored 3 points and the total score for the tea
Fifth day is 2020-01-07, Bajrang scored 7 points and the total score for the
```

Solution

sql

```sql
#Solution 1:
SELECT gender, day,
SUM(score_points) OVER(PARTITION BY gender ORDER BY day) AS total
FROM scores
GROUP BY 1,2
ORDER BY 1,2

#Solution 2:
SELECT t1.gender, t1.day, SUM(t2.score_points) AS total
FROM Scores AS t1 JOIN Scores AS t2
ON t1.gender = t2.gender
AND t1.day>=t2.day
GROUP BY t1.gender, t1.day
```

1321. Restaurant Growth | Medium | LeetCode

Table: **Customer**

```
+---------------+---------+
```

```
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| name          | varchar |
| visited_on    | date    |
| amount        | int     |
+---------------+---------+
(customer_id, visited_on) is the primary key for this table.
This table contains data about customer transactions in a restaurant.
visited_on is the date on which the customer with ID (customer_id) have visit
amount is the total paid by a customer.
```

You are the restaurant owner and you want to analyze a possible expansion (there will be at least one customer every day).

Write an SQL query to compute moving average of how much customer paid in a 7 days window (current day + 6 days before) .

The query result format is in the following example:

Return result table ordered by visited_on.

`average_amount` should be rounded to 2 decimal places, all dates are in the format ('YYYY-MM-DD').

```
Customer table:
+-------------+--------------+--------------+-------------+
| customer_id | name         | visited_on   | amount      |
+-------------+--------------+--------------+-------------+
| 1           | Jhon         | 2019-01-01   | 100         |
| 2           | Daniel       | 2019-01-02   | 110         |
| 3           | Jade         | 2019-01-03   | 120         |
| 4           | Khaled       | 2019-01-04   | 130         |
| 5           | Winston      | 2019-01-05   | 110         |
| 6           | Elvis        | 2019-01-06   | 140         |
| 7           | Anna         | 2019-01-07   | 150         |
| 8           | Maria        | 2019-01-08   | 80          |
| 9           | Jaze         | 2019-01-09   | 110         |
| 1           | Jhon         | 2019-01-10   | 130         |
| 3           | Jade         | 2019-01-10   | 150         |
```

```
+------------+--------------+-------------+-------------+
```

Result table:

```
+--------------+--------------+----------------+
| visited_on   | amount       | average_amount |
+--------------+--------------+----------------+
| 2019-01-07   | 860          | 122.86         |
| 2019-01-08   | 840          | 120            |
| 2019-01-09   | 840          | 120            |
| 2019-01-10   | 1000         | 142.86         |
+--------------+--------------+----------------+
```

```
1st moving average from 2019-01-01 to 2019-01-07 has an average_amount of (1(
2nd moving average from 2019-01-02 to 2019-01-08 has an average_amount of (1
3rd moving average from 2019-01-03 to 2019-01-09 has an average_amount of (1
4th moving average from 2019-01-04 to 2019-01-10 has an average_amount of (1
```

Solution

sql

```sql
#Solution 1:
SELECT visited_on, SUM(amount) OVER(ORDER BY visited_on ROWS 6 PRECEDING),
round(avg(amount) OVER(ORDER BY visited_on ROWS 6 PRECEDING),2)
FROM
(
    SELECT visited_on, SUM(amount) AS amount
    FROM customer
    GROUP BY visited_on
    ORDER BY visited_on
) a
ORDER BY visited_on offset 6 ROWS

#Solution 2:
SELECT t1.visited_on,
    SUM(t2.amount) AS amount,
    round(avg(t2.amount), 2) AS average_amount
FROM (
    SELECT visited_on, SUM(amount) AS amount
    FROM Customer
    GROUP BY visited_on) AS t1
    inner join
```

```
    (
    SELECT visited_on, SUM(amount) AS amount
    FROM Customer
    GROUP BY visited_on) AS t2
ON t2.visited_on BETWEEN DATE_SUB(t1.visited_on, INTERVAL 6 DAY) and t1.visit
GROUP BY t1.visited_on
HAVING COUNT(1)=7
```

1322. Ads Performance | Easy | LeetCode

Table: **Ads**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| ad_id         | int     |
| user_id       | int     |
| action        | enum    |
+---------------+---------+
(ad_id, user_id) is the primary key for this table.
Each row of this table contains the ID of an Ad, the ID of a user and the act
The action column is an ENUM type of ('Clicked', 'Viewed', 'Ignored').
```

A company is running Ads and wants to calculate the performance of each Ad.

$$CTR = \begin{cases} 0, & \text{if Ad total clicks + Ad total views = 0} \\ \frac{\text{Ad total clicks}}{\text{Ad total clicks + Ad total views}} \times 100, & \text{otherwise} \end{cases}$$

Round `ctr` to 2 decimal points. **Order** the result table by `ctr` in descending order and by `ad_id` in ascending order in case of a tie.

The query result format is in the following example:

```
Ads table:
+-------+---------+---------+
| ad_id | user_id | action  |
+-------+---------+---------+
```

```
| 1     | 1       | Clicked |
| 2     | 2       | Clicked |
| 3     | 3       | Viewed  |
| 5     | 5       | Ignored |
| 1     | 7       | Ignored |
| 2     | 7       | Viewed  |
| 3     | 5       | Clicked |
| 1     | 4       | Viewed  |
| 2     | 11      | Viewed  |
| 1     | 2       | Clicked |
+-------+---------+---------+
Result table:
+-------+-------+
| ad_id | ctr   |
+-------+-------+
| 1     | 66.67 |
| 3     | 50.00 |
| 2     | 33.33 |
| 5     | 0.00  |
+-------+-------+
for ad_id = 1, ctr = (2/(2+1)) * 100 = 66.67
for ad_id = 2, ctr = (1/(1+2)) * 100 = 33.33
for ad_id = 3, ctr = (1/(1+1)) * 100 = 50.00
for ad_id = 5, ctr = 0.00, Note that ad_id = 5 has no clicks or views.
Note that we don't care about Ignored Ads.
Result table is ordered by the ctr. in case of a tie we order them by ad_id
```

Solution

sql

```sql
#Solution 1:
SELECT ad_id,
    (CASE WHEN clicks+views = 0 THEN 0 ELSE ROUND(clicks/(clicks+views)*100,
FROM
    (SELECT ad_id,
        SUM(CASE WHEN action='Clicked' THEN 1 ELSE 0 END) AS clicks,
        SUM(CASE WHEN action='Viewed' THEN 1 ELSE 0 END) AS views
    FROM Ads
    GROUP BY ad_id) AS t
ORDER BY ctr DESC, ad_id ASC
```

```
#Solution 2:
WITH t1 AS(
SELECT ad_id, SUM(CASE WHEN action in ('Clicked') THEN 1 ELSE 0 END) AS clicl
FROM ads
GROUP BY ad_id
)

, t2 AS
(
SELECT ad_id AS ad, SUM(CASE WHEN action in ('Clicked','Viewed') THEN 1 ELSE
FROM ads
GROUP BY ad_id
)

SELECT a.ad_id, coalesce(round((clicked +0.0)/nullif((total +0.0),0)*100,2),(
FROM
(
select *
FROM t1 JOIN t2
ON t1.ad_id = t2.ad) a
ORDER BY ctr DESC, ad_id
```

1327. List the Products Ordered in a Period | Easy | LeetCode

Table: **Products**

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| product_id       | int     |
| product_name     | varchar |
| product_category | varchar |
+------------------+---------+
product_id is the primary key for this table.
This table contains data about the company's products.
```

Table: **Orders**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| order_date    | date    |
| unit          | int     |
+---------------+---------+
There is no primary key for this table. It may have duplicate rows.
product_id is a foreign key to Products table.
unit is the number of products ordered in order_date.
```

Write an SQL query to get the names of products with greater than or equal to 100 units ordered in February 2020 and their amount.

Return result table in any order.

The query result format is in the following example:

```
Products table:
+-------------+----------------------+------------------+
| product_id  | product_name         | product_category |
+-------------+----------------------+------------------+
| 1           | Leetcode Solutions   | Book             |
| 2           | Jewels of Stringology | Book            |
| 3           | HP                   | Laptop           |
| 4           | Lenovo               | Laptop           |
| 5           | Leetcode Kit         | T-shirt          |
+-------------+----------------------+------------------+

Orders table:
+--------------+--------------+----------+
| product_id   | order_date   | unit     |
+--------------+--------------+----------+
| 1            | 2020-02-05   | 60       |
| 1            | 2020-02-10   | 70       |
| 2            | 2020-01-18   | 30       |
| 2            | 2020-02-11   | 80       |
| 3            | 2020-02-17   | 2        |
| 3            | 2020-02-24   | 3        |
| 4            | 2020-03-01   | 20       |
```

```
| 4            | 2020-03-04   | 30       |
| 4            | 2020-03-04   | 60       |
| 5            | 2020-02-25   | 50       |
| 5            | 2020-02-27   | 50       |
| 5            | 2020-03-01   | 50       |
+--------------+--------------+----------+

Result table:
+--------------------+---------+
| product_name       | unit    |
+--------------------+---------+
| Leetcode Solutions | 130     |
| Leetcode Kit       | 100     |
+--------------------+---------+

Products with product_id = 1 is ordered in February a total of (60 + 70) = 13
Products with product_id = 2 is ordered in February a total of 80.
Products with product_id = 3 is ordered in February a total of (2 + 3) = 5.
Products with product_id = 4 was not ordered in February 2020.
Products with product_id = 5 is ordered in February a total of (50 + 50) = 10
```

Solution

sql

```sql
#Solution 1:
SELECT a.product_name, a.unit
FROM
(SELECT p.product_name, SUM(unit) AS unit
FROM orders o
JOIN products p
ON o.product_id = p.product_id
WHERE MONTH(order_date)=2 and YEAR(order_date) = 2020
GROUP BY o.product_id) a
WHERE a.unit>=100

#Solution 2:
SELECT product_name, SUM(unit) AS unit
FROM Products JOIN Orders
ON Products.product_id = Orders.product_id
WHERE left(order_date, 7) = "2020-02"
GROUP BY Products.product_id
```

```
HAVING SUM(unit)>=100
```

1336. Number of Transactions per Visit | Hard | LeetCode

Table: `Visits`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| visit_date    | date    |
+---------------+---------+
(user_id, visit_date) is the primary key for this table.
Each row of this table indicates that user_id has visited the bank in visit_
```

Table: `Transactions`

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| user_id          | int     |
| transaction_date | date    |
| amount           | int     |
+------------------+---------+
There is no primary key for this table, it may contain duplicates.
Each row of this table indicates that user_id has done a transaction of amou
It is guaranteed that the user has visited the bank in the transaction_date.
```

A bank wants to draw a chart of the number of transactions bank visitors did in one visit to the bank and the corresponding number of visitors who have done this number of transaction in one visit.

Write an SQL query to find how many users visited the bank and didn't do any transactions, how many visited the bank and did one transaction and so on.

The result table will contain two columns:

- `transactions_count` which is the number of transactions done in one visit.
  `visits_count` which is the corresponding number of users who did transactions_count in one visit to the bank.
  `transactions_count` should take all values from `0` to `max(transactions_count)` done by one or more users.

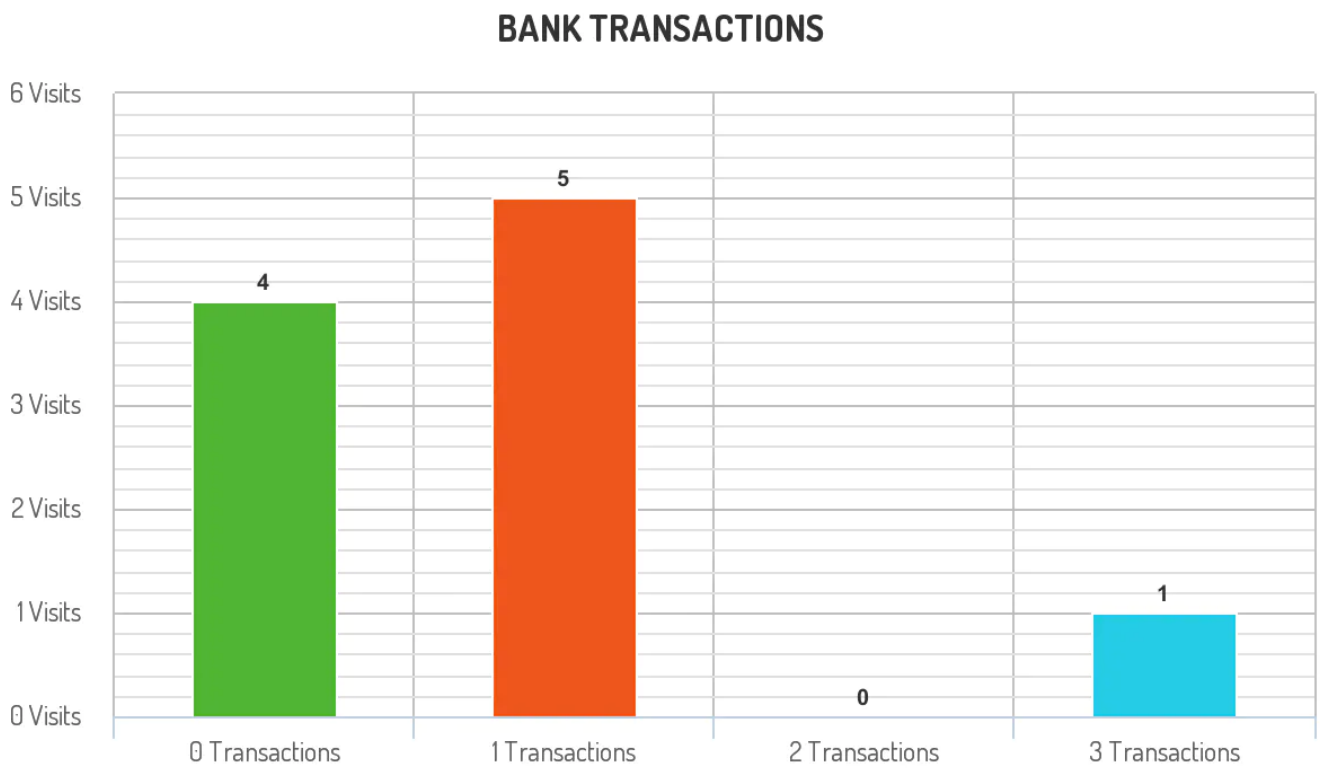
Order the result table by `transactions_count` .

The query result format is in the following example:

```
Visits table:
+---------+------------+
| user_id | visit_date |
+---------+------------+
| 1       | 2020-01-01 |
| 2       | 2020-01-02 |
| 12      | 2020-01-01 |
| 19      | 2020-01-03 |
| 1       | 2020-01-02 |
| 2       | 2020-01-03 |
| 1       | 2020-01-04 |
| 7       | 2020-01-11 |
| 9       | 2020-01-25 |
| 8       | 2020-01-28 |
+---------+------------+
Transactions table:
+---------+------------------+--------+
| user_id | transaction_date | amount |
+---------+------------------+--------+
| 1       | 2020-01-02       | 120    |
| 2       | 2020-01-03       | 22     |
| 7       | 2020-01-11       | 232    |
| 1       | 2020-01-04       | 7      |
| 9       | 2020-01-25       | 33     |
| 9       | 2020-01-25       | 66     |
| 8       | 2020-01-28       | 1      |
| 9       | 2020-01-25       | 99     |
+---------+------------------+--------+
Result table:
+--------------------+--------------+
| transactions_count | visits_count |
```

```
+--------------------+-------------+
| 0                  | 4           |
| 1                  | 5           |
| 2                  | 0           |
| 3                  | 1           |
+--------------------+-------------+
* For transactions_count = 0, The visits (1, "2020-01-01"), (2, "2020-01-02"
* For transactions_count = 1, The visits (2, "2020-01-03"), (7, "2020-01-11"
* For transactions_count = 2, No customers visited the bank and did two trans
* For transactions_count = 3, The visit (9, "2020-01-25") did three transact
* For transactions_count >= 4, No customers visited the bank and did more tha
```

The chart drawn for this example is as follows:



Solution

sql

```sql
WITH RECURSIVE t1 AS(
                    SELECT visit_date,
                           COALESCE(num_visits,0) as num_visits,
                           COALESCE(num_trans,0) as num_trans
                    FROM ((
```

```
                        SELECT visit_date, user_id, COUNT(*) as num_visits
                        FROM visits
                        GROUP BY 1, 2) AS a
                       LEFT JOIN
                        (
                         SELECT transaction_date,
                               user_id,
                               count(*) as num_trans
                          FROM transactions
                        GROUP BY 1, 2) AS b
                       ON a.visit_date = b.transaction_date and a.user_id =
                    ),

            t2 AS (
                    SELECT MAX(num_trans) as trans
                      FROM t1
                    UNION ALL
                    SELECT trans-1
                      FROM t2
                    WHERE trans >= 1)

SELECT trans as transactions_count,
       COALESCE(visits_count,0) as visits_count
  FROM t2 LEFT JOIN (
                   SELECT num_trans as transactions_count, COALESCE(COUNT(*)
                   FROM t1
                   GROUP BY 1
                   ORDER BY 1) AS a
ON a.transactions_count = t2.trans
ORDER BY 1
```

1341. Movie Rating | Medium | LeetCode

Table: **Movies**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| movie_id      | int     |
| title         | varchar |
```

```
+---------------+---------+
movie_id is the primary key for this table.
title is the name of the movie.
```

Table: **Users**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| name          | varchar |
+---------------+---------+
user_id is the primary key for this table.
```

Table: **Movie_Rating**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| movie_id      | int     |
| user_id       | int     |
| rating        | int     |
| created_at    | date    |
+---------------+---------+
(movie_id, user_id) is the primary key for this table.
This table contains the rating of a movie by a user in their review.
created_at is the user's review date.
```

Write the following SQL query:

- Find the name of the user who has rated the greatest number of the movies.
  In case of a tie, return lexicographically smaller user name.
- Find the movie name with the highest average rating in February 2020.
  In case of a tie, return lexicographically smaller movie name..

uery is returned in 2 rows, the query result format is in the following example:

Movies table:

movie_id	title
1	Avengers
2	Frozen 2
3	Joker

Users table:

user_id	name
1	Daniel
2	Monica
3	Maria
4	James

Movie_Rating table:

movie_id	user_id	rating	created_at
1	1	3	2020-01-12
1	2	4	2020-02-11
1	3	2	2020-02-12
1	4	1	2020-01-01
2	1	5	2020-02-17
2	2	2	2020-02-01
2	3	2	2020-03-01
3	1	3	2020-02-22
3	2	4	2020-02-25

Result table:

results
Daniel
Frozen 2

Daniel and Maria have rated 3 movies ("Avengers", "Frozen 2" and "Joker") but
Frozen 2 and Joker have a rating average of 3.5 in February but Frozen 2 is s

Solution

sql

```sql
#Solution 1:
(SELECT name AS results
FROM Movie_Rating JOIN Users
ON Movie_Rating.user_id = Users.user_id
GROUP BY Movie_Rating.user_id
ORDER BY count(1) DESC, name
LIMIT 1)
UNION ALL
(SELECT title AS results
FROM Movie_Rating JOIN Movies
ON Movie_Rating.movie_id = Movies.movie_id
WHERE left(created_at, 7) = "2020-02"
GROUP BY Movie_Rating.movie_id
ORDER BY avg(rating) DESC, title
LIMIT 1
)

#Solution 2:
SELECT name AS results
FROM(
(SELECT a.name
FROM(
SELECT name, count(*),
rank() OVER(ORDER BY count(*) DESC) AS rk
FROM movie_rating m
JOIN users u
ON m.user_id = u.user_id
GROUP BY name, m.user_id
ORDER BY rk, name) a
LIMIT 1)
UNION
(SELECT title
FROM(
SELECT title, round(avg(rating),1) AS rnd
FROM movie_rating m
```

```
JOIN movies u
on m.movie_id = u.movie_id
WHERE month(created_at) = 2
GROUP BY title
ORDER BY rnd DESC, title) b
LIMIT 1)) AS d
```

1350. Students With Invalid Departments | Easy | LeetCode

Table: **Departments**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| name          | varchar |
+---------------+---------+
id is the primary key of this table.
The table has information about the id of each department of a university.
```

Table: **Students**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| name          | varchar |
| department_id | int     |
+---------------+---------+
id is the primary key of this table.
The table has information about the id of each student at a university and th
```

Write an SQL query to find the id and the name of all students who are enrolled in
)artments that no longer exists.

Return the result table in any order.

The query result format is in the following example:

```
Departments table:
+------+--------------------------+
| id   | name                     |
+------+--------------------------+
| 1    | Electrical Engineering   |
| 7    | Computer Engineering     |
| 13   | Bussiness Administration |
+------+--------------------------+

Students table:
+------+----------+---------------+
| id   | name     | department_id |
+------+----------+---------------+
| 23   | Alice    | 1             |
| 1    | Bob      | 7             |
| 5    | Jennifer | 13            |
| 2    | John     | 14            |
| 4    | Jasmine  | 77            |
| 3    | Steve    | 74            |
| 6    | Luis     | 1             |
| 8    | Jonathan | 7             |
| 7    | Daiana   | 33            |
| 11   | Madelynn | 1             |
+------+----------+---------------+

Result table:
+------+----------+
| id   | name     |
+------+----------+
| 2    | John     |
| 7    | Daiana   |
| 4    | Jasmine  |
| 3    | Steve    |
+------+----------+

John, Daiana, Steve and Jasmine are enrolled in departments 14, 33, 74 and 7
```

Solution

```sql
#Solution 1:
SELECT s.id, s.name
FROM students s LEFT JOIN
departments d
ON s.department_id = d.id
WHERE d.name IS NULL;

#Solution 2:
SELECT id, name
FROM Students
WHERE department_id NOT IN
    (SELECT id FROM Departments)
```

1355. Activity Participants | Medium | LeetCode

Table: **Friends**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| name          | varchar |
| activity      | varchar |
+---------------+---------+
id is the id of the friend and primary key for this table.
name is the name of the friend.
activity is the name of the activity which the friend takes part in.
```

Table: **Activities**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| name          | varchar |
+---------------+---------+
id is the primary key for this table.
```

```
name is the name of the activity.
```

Write an SQL query to find the names of all the activities with neither maximum, nor minimum number of participants.

Return the result table in any order. Each activity in table Activities is performed by any person in the table Friends.

The query result format is in the following example:

```
Friends table:
+------+--------------+---------------+
| id   | name         | activity      |
+------+--------------+---------------+
| 1    | Jonathan D.  | Eating        |
| 2    | Jade W.      | Singing       |
| 3    | Victor J.    | Singing       |
| 4    | Elvis Q.     | Eating        |
| 5    | Daniel A.    | Eating        |
| 6    | Bob B.       | Horse Riding  |
+------+--------------+---------------+

Activities table:
+------+--------------+
| id   | name         |
+------+--------------+
| 1    | Eating       |
| 2    | Singing      |
| 3    | Horse Riding |
+------+--------------+

Result table:
+--------------+
| results      |
+--------------+
| Singing      |
+--------------+

Eating activity is performed by 3 friends, maximum number of participants, (
Horse Riding activity is performed by 1 friend, minimum number of participant
```

Singing is performed by 2 friends (Victor J. and Jade W.)

Solution

sql

```sql
#Solution 1:
WITH CTE AS
(SELECT COUNT(*) AS cnt, activity FROM Friends GROUP BY activity)

SELECT activity FROM CTE
WHERE cnt NOT IN
    (SELECT MAX(cnt) FROM CTE
    UNION ALL
    SELECT MIN(cnt) FROM CTE)

#Solution 2:
WITH t1 AS(
SELECT MAX(a.total) AS total
FROM(
    SELECT activity, COUNT(*) AS total
    FROM friends
    GROUP BY activity) a
    UNION ALL
    SELECT MIN(b.total) AS low
    FROM(
    SELECT activity, COUNT(*) AS total
    FROM friends
    GROUP BY activity) b),
t2 AS
(
    SELECT activity, COUNT(*) AS total
    FROM friends
    GROUP BY activity
)

SELECT activity
FROM t1 RIGHT JOIN t2
ON t1.total = t2.total
WHERE t1.total is null
```

1364. Number of Trusted Contacts of a Customer | Medium | LeetCode

Table: **Customers**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| customer_name | varchar |
| email         | varchar |
+---------------+---------+
customer_id is the primary key for this table.
Each row of this table contains the name and the email of a customer of an or
```

Table: **Contacts**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | id      |
| contact_name  | varchar |
| contact_email | varchar |
+---------------+---------+
(user_id, contact_email) is the primary key for this table.
Each row of this table contains the name and email of one contact of customer
This table contains information about people each customer trust. The contact
```

Table: **Invoices**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| invoice_id  | int     |
| price       | int     |
| user_id     | int     |
+-------------+---------+
invoice_id is the primary key for this table.
```

Each row of this table indicates that user_id has an invoice with invoice_id

Write an SQL query to find the following for each invoice_id:

- customer_name: The name of the customer the invoice is related to.
- price: The price of the invoice.
- contacts_cnt: The number of contacts related to the customer.
- trusted_contacts_cnt: The number of contacts related to the customer and at the same time they are customers to the shop. (i.e His/Her email exists in the Customers table.) Order the result table by invoice_id.

The query result format is in the following example:

```
Customers table:
+-------------+---------------+--------------------+
| customer_id | customer_name | email              |
+-------------+---------------+--------------------+
| 1           | Alice         | alice@leetcode.com |
| 2           | Bob           | bob@leetcode.com   |
| 13          | John          | john@leetcode.com  |
| 6           | Alex          | alex@leetcode.com  |
+-------------+---------------+--------------------+
Contacts table:
+-------------+--------------+--------------------+
| user_id     | contact_name | contact_email      |
+-------------+--------------+--------------------+
| 1           | Bob          | bob@leetcode.com   |
| 1           | John         | john@leetcode.com  |
| 1           | Jal          | jal@leetcode.com   |
| 2           | Omar         | omar@leetcode.com  |
| 2           | Meir         | meir@leetcode.com  |
| 6           | Alice        | alice@leetcode.com |
+-------------+--------------+--------------------+
Invoices table:
+------------+-------+---------+
| invoice_id | price | user_id |
+------------+-------+---------+
| 77         | 100   | 1       |
| 88         | 200   | 1       |
```

```
| 99          | 300    | 2        |
| 66          | 400    | 2        |
| 55          | 500    | 13       |
| 44          | 60     | 6        |
+-------------+--------+----------+
Result table:
+-------------+---------------+--------+--------------+----------------------+
| invoice_id  | customer_name | price  | contacts_cnt | trusted_contacts_cnt |
+-------------+---------------+--------+--------------+----------------------+
| 44          | Alex          | 60     | 1            | 1                    |
| 55          | John          | 500    | 0            | 0                    |
| 66          | Bob           | 400    | 2            | 0                    |
| 77          | Alice         | 100    | 3            | 2                    |
| 88          | Alice         | 200    | 3            | 2                    |
| 99          | Bob           | 300    | 2            | 0                    |
+-------------+---------------+--------+--------------+----------------------+
Alice has three contacts, two of them are trusted contacts (Bob and John).
Bob has two contacts, none of them is a trusted contact.
Alex has one contact and it is a trusted contact (Alice).
John doesn't have any contacts.
```

Solution

sql

```sql
SELECT invoice_id, customer_name, price,
 COUNT(Contacts.user_id) AS contacts_cnt,
 SUM(CASE WHEN Contacts.contact_name IN
    (SELECT customer_name FROM Customers)
    THEN 1 ELSE 0 END) AS trusted_contacts_cnt
FROM Invoices INNER JOIN Customers ON Invoices.user_id = Customers.customer_
LEFT JOIN Contacts ON Customers.customer_id = Contacts.user_id
GROUP BY Invoices.invoice_id, customer_name
ORDER BY Invoices.invoice_id
```

1369. Get the Second Most Recent Activity | Hard | LeetCode

Table: `UserActivity`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| username      | varchar |
| activity      | varchar |
| startDate     | Date    |
| endDate       | Date    |
+---------------+---------+
This table does not contain primary key.
This table contain information about the activity performed of each user in a
A person with username performed a activity from startDate to endDate.
```

Write an SQL query to show the second most recent activity of each user.

If the user only has one activity, return that one.

A user can't perform more than one activity at the same time. Return the result table in any order.

The query result format is in the following example:

```
UserActivity table:
+------------+--------------+-------------+-------------+
| username   | activity     | startDate   | endDate     |
+------------+--------------+-------------+-------------+
| Alice      | Travel       | 2020-02-12  | 2020-02-20  |
| Alice      | Dancing      | 2020-02-21  | 2020-02-23  |
| Alice      | Travel       | 2020-02-24  | 2020-02-28  |
| Bob        | Travel       | 2020-02-11  | 2020-02-18  |
+------------+--------------+-------------+-------------+

Result table:
+------------+--------------+-------------+-------------+
| username   | activity     | startDate   | endDate     |
+------------+--------------+-------------+-------------+
| Alice      | Dancing      | 2020-02-21  | 2020-02-23  |
| Bob        | Travel       | 2020-02-11  | 2020-02-18  |
+------------+--------------+-------------+-------------+

The most recent activity of Alice is Travel from 2020-02-24 to 2020-02-28, be
```

```
Bob only has one record, we just take that one.
```

Solution

```sql
(SELECT *
FROM UserActivity
GROUP BY username
HAVING count(1) = 1)
UNION
(SELECT a.*
FROM UserActivity AS a LEFT JOIN UserActivity AS b
on a.username = b.username AND a.endDate<b.endDate
GROUP BY a.username, a.endDate
HAVING count(b.endDate) = 1)
```

1378. Replace Employee ID With The Unique Identifier | Easy | LeetCode

Table: **Employees**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| id           | int     |
| name         | varchar |
+--------------+---------+
id is the primary key for this table.
Each row of this table contains the id and the name of an employee in a compa
```

Table: **EmployeeUNI**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| id           | int     |
```

```
| unique_id     | int     |
+---------------+---------+
(id, unique_id) is the primary key for this table.
Each row of this table contains the id and the corresponding unique id of an
```

Write an SQL query to show the unique ID of each user, If a user doesn't have a unique ID replace just show null.

Return the result table in any order.

The query result format is in the following example:

```
Employees table:
+----+----------+
| id | name     |
+----+----------+
| 1  | Alice    |
| 7  | Bob      |
| 11 | Meir     |
| 90 | Winston  |
| 3  | Jonathan |
+----+----------+

EmployeeUNI table:
+----+-----------+
| id | unique_id |
+----+-----------+
| 3  | 1         |
| 11 | 2         |
| 90 | 3         |
+----+-----------+

EmployeeUNI table:
+-----------+----------+
| unique_id | name     |
+-----------+----------+
| null      | Alice    |
| null      | Bob      |
| 2         | Meir     |
| 3         | Winston  |
| 1         | Jonathan |
```

```
+----------+----------+

Alice and Bob don't have a unique ID, We will show null instead.
The unique ID of Meir is 2.
The unique ID of Winston is 3.
The unique ID of Jonathan is 1.
```

Solution

sql

```sql
SELECT unique_id, name
FROM Employees
LEFT JOIN EmployeeUNI
ON Employees.id = EmployeeUNI.id
```

1384. Total Sales Amount by Year | Hard | LeetCode

Table: **Product**

+---------------+---------+ | Column Name | Type | +---------------+---------+ | product_id | int | | product_name | varchar | +---------------+---------+ product_id is the primary key for this table. product_name is the name of the product.

Table: **Sales**

```
+---------------------+---------+
| Column Name         | Type    |
+---------------------+---------+
| product_id          | int     |
| period_start        | varchar |
| period_end          | date    |
| average_daily_sales | int     |
+---------------------+---------+
product_id is the primary key for this table.
period_start and period_end indicates the start and end date for sales perio
The average_daily_sales column holds the average daily sales amount of the i
```

Write an SQL query to report the Total sales amount of each item for each year, with corresponding product name, product_id, product_name and report_year.

Dates of the sales years are between 2018 to 2020. Return the result table **ordered** by product_id and report_year.

The query result format is in the following example:

```
Product table:
+------------+--------------+
| product_id | product_name |
+------------+--------------+
| 1          | LC Phone     |
| 2          | LC T-Shirt   |
| 3          | LC Keychain  |
+------------+--------------+

Sales table:
+------------+--------------+-------------+---------------------+
| product_id | period_start | period_end  | average_daily_sales |
+------------+--------------+-------------+---------------------+
| 1          | 2019-01-25   | 2019-02-28  | 100                 |
| 2          | 2018-12-01   | 2020-01-01  | 10                  |
| 3          | 2019-12-01   | 2020-01-31  | 1                   |
+------------+--------------+-------------+---------------------+

Result table:
+------------+--------------+-------------+--------------+
| product_id | product_name | report_year | total_amount |
+------------+--------------+-------------+--------------+
| 1          | LC Phone     |    2019     | 3500         |
| 2          | LC T-Shirt   |    2018     | 310          |
| 2          | LC T-Shirt   |    2019     | 3650         |
| 2          | LC T-Shirt   |    2020     | 10           |
| 3          | LC Keychain  |    2019     | 31           |
| 3          | LC Keychain  |    2020     | 31           |
+------------+--------------+-------------+--------------+
LC Phone was sold for the period of 2019-01-25 to 2019-02-28, and there are 3
LC T-shirt was sold for the period of 2018-12-01 to 2020-01-01, and there are
LC Keychain was sold for the period of 2019-12-01 to 2020-01-31, and there a
```

Solution

sql

```sql
SELECT
    b.product_id,
    a.product_name,
    a.yr AS report_year,
    CASE
        WHEN YEAR(b.period_start)=YEAR(b.period_end) AND a.yr=YEAR(b.period_s
        WHEN a.yr=YEAR(b.period_start) THEN DATEDIFF(DATE_FORMAT(b.period_sta
        WHEN a.yr=YEAR(b.period_end) THEN DAYOFYEAR(b.period_end)
        WHEN a.yr>YEAR(b.period_start) AND a.yr<YEAR(b.period_end) THEN 365
        ELSE 0
    END * average_daily_sales AS total_amount
FROM
    (SELECT product_id,product_name,'2018' AS yr FROM Product
    UNION
    SELECT product_id,product_name,'2019' AS yr FROM Product
    UNION
    SELECT product_id,product_name,'2020' AS yr FROM Product) a
    JOIN
    Sales b
    ON a.product_id=b.product_id
HAVING total_amount > 0
ORDER BY b.product_id,a.yr
```

1393. Capital Gain/Loss | Medium | LeetCode

Table: **Stocks**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| stock_name    | varchar |
| operation     | enum    |
| operation_day | int     |
| price         | int     |
+---------------+---------+
(stock_name, day) is the primary key for this table.
```

```
The operation column is an ENUM of type ('Sell', 'Buy')
Each row of this table indicates that the stock which has stock_name had an o
It is guaranteed that each 'Sell' operation for a stock has a corresponding
```

Write an SQL query to report the Capital gain/loss for each stock.

The capital gain/loss of a stock is total gain or loss after buying and selling the stock one or many times.

Return the result table in any order.

The query result format is in the following example:

```
Stocks table:
+---------------+-----------+---------------+--------+
| stock_name    | operation | operation_day | price  |
+---------------+-----------+---------------+--------+
| Leetcode      | Buy       | 1             | 1000   |
| Corona Masks  | Buy       | 2             | 10     |
| Leetcode      | Sell      | 5             | 9000   |
| Handbags      | Buy       | 17            | 30000  |
| Corona Masks  | Sell      | 3             | 1010   |
| Corona Masks  | Buy       | 4             | 1000   |
| Corona Masks  | Sell      | 5             | 500    |
| Corona Masks  | Buy       | 6             | 1000   |
| Handbags      | Sell      | 29            | 7000   |
| Corona Masks  | Sell      | 10            | 10000  |
+---------------+-----------+---------------+--------+

Result table:
+---------------+-------------------+
| stock_name    | capital_gain_loss |
+---------------+-------------------+
| Corona Masks  | 9500              |
| Leetcode      | 8000              |
| Handbags      | -23000            |
+---------------+-------------------+
Leetcode stock was bought at day 1 for 1000$ and was sold at day 5 for 9000$.
Handbags stock was bought at day 17 for 30000$ and was sold at day 29 for 70(
Corona Masks stock was bought at day 1 for 10$ and was sold at day 3 for 1010
```

Solution

sql

```sql
#Solution 1:
SELECT stock_name,
       SUM(CASE WHEN operation = 'Buy' THEN -price ELSE price END) AS capita
FROM Stocks
GROUP BY stock_name;

#Solution 2:
SELECT stock_name, (one-two) AS capital_gain_loss
FROM(
(SELECT stock_name, sum(price) AS one
FROM stocks
WHERE operation = 'Sell'
GROUP BY stock_name) b
LEFT JOIN
(SELECT stock_name AS name, sum(price) AS two
FROM stocks
WHERE operation = 'Buy'
GROUP BY stock_name) c
ON b.stock_name = c.name)
ORDER BY capital_gain_loss DESC;
```

1398. Customers Who Bought Products A and B but Not C | Medium | LeetCode

Table: **Customers**

```
+---------------------+---------+
| Column Name         | Type    |
+---------------------+---------+
| customer_id         | int     |
| customer_name       | varchar |
+---------------------+---------+
customer_id is the primary key for this table.
customer_name is the name of the customer.
```

Table: **Orders**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| order_id     | int     |
| customer_id  | int     |
| product_name | varchar |
+--------------+---------+
order_id is the primary key for this table.
customer_id is the id of the customer who bought the product "product_name".
```

Write an SQL query to report the customer_id and customer_name of customers who bought products "A", "B" but did not buy the product "C" since we want to recommend them buy this product.

Return the result table ordered by customer_id.

The query result format is in the following example.

```
Customers table:
+-------------+---------------+
| customer_id | customer_name |
+-------------+---------------+
| 1           | Daniel        |
| 2           | Diana         |
| 3           | Elizabeth     |
| 4           | Jhon          |
+-------------+---------------+

Orders table:
+------------+--------------+---------------+
| order_id   | customer_id  | product_name  |
+------------+--------------+---------------+
| 10         |     1        |     A         |
| 20         |     1        |     B         |
| 30         |     1        |     D         |
| 40         |     1        |     C         |
| 50         |     2        |     A         |
| 60         |     3        |     A         |
```

```
| 70          |     3        |     B         |
| 80          |     3        |     D         |
| 90          |     4        |     C         |
+-------------+--------------+---------------+

Result table:
+-------------+---------------+
| customer_id | customer_name |
+-------------+---------------+
| 3           | Elizabeth     |
+-------------+---------------+
Only the customer_id with id 3 bought the product A and B but not the product
```

Solution

sql

```sql
#Solution 1:
WITH t1 AS
(
SELECT customer_id
FROM orders
WHERE product_name = 'B' AND
customer_id IN (SELECT customer_id
FROM orders
WHERE product_name = 'A'))

SELECT t1.customer_id, c.customer_name
FROM t1 JOIN customers c
ON t1.customer_id = c.customer_id
WHERE t1.customer_id != all(SELECT customer_id
FROM orders
WHERE product_name = 'C')

#Solution 2:
SELECT *
FROM Customers
WHERE customer_id IN
    (SELECT DISTINCT customer_id
     FROM Orders
     WHERE product_name = 'A'
    ) AND
```

```
    customer_id IN
    (SELECT DISTINCT customer_id
     FROM Orders
     WHERE product_name = 'B'
    ) AND
    customer_id NOT IN
    (SELECT DISTINCT customer_id
     FROM Orders
     WHERE product_name = 'C'
    )
ORDER BY customer_id

#Solution 3:
SELECT Customers.*
FROM (
    SELECT customer_id,
     sum(CASE WHEN product_name = 'A' THEN 1 ELSE 0 END) AS product_a,
     sum(CASE WHEN product_name = 'B' THEN 1 ELSE 0 END) AS product_b
    FROM Orders
    GROUP BY customer_id) AS t JOIN Customers
ON t.customer_id = Customers.customer_id
WHERE t.product_a>0 AND product_b >0 AND Customers.customer_id NOT IN (
    SELECT DISTINCT customer_id
    FROM Orders
    WHERE product_name = 'C')
ORDER BY Customers.customer_id
```

1407. Top Travellers | Easy | LeetCode

Table: **Users**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| name          | varchar |
+---------------+---------+
id is the primary key for this table.
name is the name of the user.
```

Table: **Rides**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| user_id       | int     |
| distance      | int     |
+---------------+---------+
id is the primary key for this table.
city_id is the id of the city who bought the product "product_name".
```

Write an SQL query to report the distance travelled by each user.

Return the result table ordered by `travelled_distance` in descending order, if two or more users travelled the same distance, order them by their `name` in ascending order.

The query result format is in the following example.

```
Users table:
+------+-----------+
| id   | name      |
+------+-----------+
| 1    | Alice     |
| 2    | Bob       |
| 3    | Alex      |
| 4    | Donald    |
| 7    | Lee       |
| 13   | Jonathan  |
| 19   | Elvis     |
+------+-----------+

Rides table:
+------+----------+----------+
| id   | user_id  | distance |
+------+----------+----------+
| 1    | 1        | 120      |
| 2    | 2        | 317      |
| 3    | 3        | 222      |
| 4    | 7        | 100      |
```

```
| 5    | 13       | 312      |
| 6    | 19       | 50       |
| 7    | 7        | 120      |
| 8    | 19       | 400      |
| 9    | 7        | 230      |
+------+----------+----------+

Result table:
+----------+--------------------+
| name     | travelled_distance |
+----------+--------------------+
| Elvis    | 450                |
| Lee      | 450                |
| Bob      | 317                |
| Jonathan | 312                |
| Alex     | 222                |
| Alice    | 120                |
| Donald   | 0                  |
+----------+--------------------+
Elvis and Lee travelled 450 miles, Elvis is the top traveller as his name is
Bob, Jonathan, Alex and Alice have only one ride and we just order them by th
Donald didn't have any rides, the distance travelled by him is 0.
```

Solution

sql

```sql
#Solution 1:
SELECT U.name AS name, COALESCE(SUM(R.distance),0) AS travelled_distance
FROM Users U LEFT JOIN Rides R
ON R.user_id = U.id
GROUP BY name
ORDER BY travelled_distance DESC, name

#Solution 2:
SELECT name, IFNULL(SUM(distance), 0) AS travelled_distance
FROM Users LEFT JOIN Rides
ON Users.id = Rides.user_id
GROUP BY Users.id
ORDER BY travelled_distance DESC, name

#Solution 3:
```

```sql
SELECT name, SUM(IF(ISNULL(distance), 0, distance)) AS travelled_distance
FROM Users LEFT JOIN Rides
ON Users.id = Rides.user_id
GROUP BY Users.id
ORDER BY travelled_distance DESC, name
```

1412. Find the Quiet Students in All Exams | Hard | LeetCode

Table: **Student**

```
+---------------------+---------+
| Column Name         | Type    |
+---------------------+---------+
| student_id          | int     |
| student_name        | varchar |
+---------------------+---------+
student_id is the primary key for this table.
student_name is the name of the student.
```

Table: **Exam**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| exam_id       | int     |
| student_id    | int     |
| score         | int     |
+---------------+---------+
(exam_id, student_id) is the primary key for this table.
Student with student_id got score points in exam with id exam_id.
```

A "quite" student is the one who took at least one exam and didn't score neither the high score nor the low score.

Write an SQL query to report the students (student_id, student_name) being "quiet" in **ALL** exams.

Don't return the student who has never taken any exam. Return the result table ordered by student_id.

The query result format is in the following example.

Student table:

student_id	student_name
1	Daniel
2	Jade
3	Stella
4	Jonathan
5	Will

Exam table:

exam_id	student_id	score
10	1	70
10	2	80
10	3	90
20	1	80
30	1	70
30	3	80
30	4	90
40	1	60
40	2	70
40	4	80

Result table:

student_id	student_name
2	Jade

For exam 1: Student 1 and 3 hold the lowest and high score respectively.
For exam 2: Student 1 hold both highest and lowest score.

For exam 3 and 4: Studnet 1 and 4 hold the lowest and high score respectively
Student 2 and 5 have never got the highest or lowest in any of the exam.
Since student 5 is not taking any exam, he is excluded from the result.
So, we only return the information of Student 2.

Solution

sql

```sql
#Solution 1:
WITH t1 AS(
SELECT student_id
FROM
(SELECT *,
MIN(score) OVER(PARTITION BY exam_id) AS least,
MAX(score) OVER(PARTITION BY exam_id) AS most
FROM exam) a
WHERE least = score OR most = score)

SELECT DISTINCT student_id, student_name
FROM exam JOIN student
USING (student_id)
WHERE student_id != all(SELECT student_id FROM t1)
order by 1

#Solution 2:
SELECT DISTINCT Student.*
FROM Student INNER JOIN Exam
ON Student.student_id = Exam.student_id
WHERE student.student_id NOT IN
    (SELECT e1.student_id
    FROM Exam AS e1 INNER JOIN
        (SELECT exam_id, MIN(score) AS min_score, MAX(score) AS max_score
        FROM Exam
        GROUP BY exam_id) AS e2
    ON e1.exam_id = e2.exam_id
    WHERE e1.score = e2.min_score OR e1.score = e2.max_score)
ORDER BY student_id
```

1421. NPV Queries | Medium | LeetCode

Table: **NPV**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| year          | int     |
| npv           | int     |
+---------------+---------+
(id, year) is the primary key of this table.
The table has information about the id and the year of each inventory and the
```

Table: **Queries**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| year          | int     |
+---------------+---------+
(id, year) is the primary key of this table.
The table has information about the id and the year of each inventory query.
```

Write an SQL query to find the npv of all each query of queries table.

Return the result table in any order.

The query result format is in the following example:

```
NPV table:
+------+--------+--------+
| id   | year   | npv    |
+------+--------+--------+
| 1    | 2018   | 100    |
| 7    | 2020   | 30     |
| 13   | 2019   | 40     |
| 1    | 2019   | 113    |
| 2    | 2008   | 121    |
| 3    | 2009   | 12     |
```

```
| 11   | 2020   | 99     |
| 7    | 2019   | 0      |
+------+--------+--------+
```

Queries table:

```
+------+--------+
| id   | year   |
+------+--------+
| 1    | 2019   |
| 2    | 2008   |
| 3    | 2009   |
| 7    | 2018   |
| 7    | 2019   |
| 7    | 2020   |
| 13   | 2019   |
+------+--------+
```

Result table:

```
+------+--------+--------+
| id   | year   | npv    |
+------+--------+--------+
| 1    | 2019   | 113    |
| 2    | 2008   | 121    |
| 3    | 2009   | 12     |
| 7    | 2018   | 0      |
| 7    | 2019   | 0      |
| 7    | 2020   | 30     |
| 13   | 2019   | 40     |
+------+--------+--------+
```

The npv value of (7, 2018) is not present in the NPV table, we consider it 0.
The npv values of all other queries can be found in the NPV table.

Solution

sql

```sql
#Solution 1:
SELECT q.id, q.year, COALESCE(n.npv,0) AS npv
FROM queries q
LEFT JOIN npv n
ON q.id = n.id AND q.year=n.year
```

```
#Solution 2:
SELECT Queries.*, IF(ISNULL(npv), 0, npv) AS npv
FROM Queries LEFT JOIN NPV
ON Queries.id = NPV.id AND Queries.year = NPV.year
```

1435. Create a Session Bar Chart | Easy | LeetCode

Table: **Sessions**

```
+---------------------+---------+
| Column Name         | Type    |
+---------------------+---------+
| session_id          | int     |
| duration            | int     |
+---------------------+---------+
session_id is the primary key for this table.
duration is the time in seconds that a user has visited the application.
```

You want to know how long a user visits your application. You decided to create bins of "[0-5>", "[5-10>", "[10-15>" and "15 minutes or more" and count the number of sessions on it.

Write an SQL query to report the (bin, total) in **any** order.

```
The query result format is in the following example.

Sessions table:
+-------------+---------------+
| session_id  | duration      |
+-------------+---------------+
| 1           | 30            |
| 2           | 299           |
| 3           | 340           |
| 4           | 580           |
| 5           | 1000          |
+-------------+---------------+
```

Result table:

```
+--------------+--------------+
| bin          | total        |
+--------------+--------------+
| [0-5>        | 3            |
| [5-10>       | 1            |
| [10-15>      | 0            |
| 15 or more   | 1            |
+--------------+--------------+
```

For session_id 1, 2 and 3 have a duration greater or equal than 0 minutes and
For session_id 4 has a duration greater or equal than 5 minutes and less than
There are no session with a duration greater or equial than 10 minutes and le
For session_id 5 has a duration greater or equal than 15 minutes.

Solution

sql

```sql
#Solution 1:
(SELECT '[0-5>' AS bin,
 SUM(CASE WHEN duration/60 < 5 THEN 1 ELSE 0 END) AS total FROM Sessions)
 UNION
(SELECT '[5-10>' AS bin,
 SUM(CASE WHEN ((duration/60 >= 5) AND (duration/60 < 10)) THEN 1 ELSE 0 END)
 UNION
(SELECT '[10-15>' AS bin,
 SUM(CASE WHEN ((duration/60 >= 10) AND (duration/60 < 15)) THEN 1 ELSE 0 END
 UNION
(SELECT '15 or more' AS bin,
 SUM(CASE WHEN duration/60 >= 15 THEN 1 ELSE 0 END) AS total FROM Sessions)

#Solution 2:
SELECT '[0-5>' AS bin, count(1) AS total
FROM Sessions
WHERE duration>=0 AND duration < 300
UNION
SELECT '[5-10>' AS bin, count(1) AS total
FROM Sessions
WHERE duration>=300 AND duration < 600
UNION
SELECT '[10-15>' AS bin, count(1) AS total
```

```
FROM Sessions
WHERE duration>=600 AND duration < 900
UNION
SELECT '15 or more' AS bin, count(1) AS total
FROM Sessions
WHERE duration >= 900
```

1440. Evaluate Boolean Expression | Medium | LeetCode

Table `Variables`:

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| name          | varchar |
| value         | int     |
+---------------+---------+
name is the primary key for this table.
This table contains the stored variables and their values.
```

Table `Expressions`:

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| left_operand  | varchar |
| operator      | enum    |
| right_operand | varchar |
+---------------+---------+
(left_operand, operator, right_operand) is the primary key for this table.
This table contains a boolean expression that should be evaluated.
operator is an enum that takes one of the values ('<', '>', '=')
The values of left_operand and right_operand are guaranteed to be in the Var
```

Write an SQL query to evaluate the boolean expressions in `Expressions` table.

Return the result table in any order.

The query result format is in the following example.

```
Variables table:
+------+-------+
| name | value |
+------+-------+
| x    | 66    |
| y    | 77    |
+------+-------+

Expressions table:
+--------------+----------+---------------+
| left_operand | operator | right_operand |
+--------------+----------+---------------+
| x            | >        | y             |
| x            | <        | y             |
| x            | =        | y             |
| y            | >        | x             |
| y            | <        | x             |
| x            | =        | x             |
+--------------+----------+---------------+

Result table:
+--------------+----------+---------------+-------+
| left_operand | operator | right_operand | value |
+--------------+----------+---------------+-------+
| x            | >        | y             | false |
| x            | <        | y             | true  |
| x            | =        | y             | false |
| y            | >        | x             | true  |
| y            | <        | x             | false |
| x            | =        | x             | true  |
+--------------+----------+---------------+-------+
As shown, you need find the value of each boolean exprssion in the table usi
```

Solution

sql

```sql
#Solution 1:
WITH t1 AS(
```

```
SELECT e.left_operand, e.operator, e.right_operand, v.value AS left_val, v_1.
FROM expressions e
JOIN variables v
ON v.name = e.left_operand
JOIN variables v_1
ON v_1.name = e.right_operand)

SELECT t1.left_operand, t1.operator, t1.right_operand,
CASE WHEN t1.operator = '<' THEN (SELECT t1.left_val< t1.right_val)
WHEN t1.operator = '>' THEN (SELECT t1.left_val > t1.right_val)
WHEN t1.operator = '=' THEN (SELECT t1.left_val = t1.right_val)
ELSE FALSE
END AS VALUE
FROM t1

#Solution 2:
# nested INNER JOIN can trim the volume of the intermediate table, which give
SELECT t.left_operand, t.operator, t.right_operand,
    (CASE WHEN v1_value>v2.value AND operator = '>' THEN "true"
          WHEN v1_value<v2.value AND operator = '<' THEN "true"
          WHEN v1_value=v2.value AND operator = '=' THEN "true"
          ELSE "false"
          END) AS value
FROM
   (SELECT e.*, v1.value AS v1_value
    FROM Expressions AS e INNER JOIN Variables AS v1
    ON e.left_operand = v1.name) AS t INNER JOIN Variables AS v2
    ON t.right_operand = v2.name

#Solution 3:
SELECT t.left_operand, t.operator, t.right_operand,
    (CASE WHEN operator = '>' THEN IF(v1_value>v2.value, "true", "false")
          WHEN operator = '<' THEN IF(v1_value<v2.value, "true", "false")
          WHEN operator = '=' THEN IF(v1_value=v2.value, "true", "false")
          END) AS value
FROM
   (SELECT e.*, v1.value AS v1_value
    FROM Expressions AS e INNER JOIN Variables AS v1
    ON e.left_operand = v1.name) AS t INNER JOIN Variables AS v2
    ON t.right_operand = v2.name
```

1445. Apples & Oranges | Medium | LeetCode

Table: `Sales`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| sale_date     | date    |
| fruit         | enum    |
| sold_num      | int     |
+---------------+---------+
(sale_date,fruit) is the primary key for this table.
This table contains the sales of "apples" and "oranges" sold each day.
```

Write an SQL query to report the difference between number of **apples** and **oranges** sold each day.

Return the result table ordered by sale_date in format ('YYYY-MM-DD').

The query result format is in the following example:

```
Sales table:
+------------+------------+-------------+
| sale_date  | fruit      | sold_num    |
+------------+------------+-------------+
| 2020-05-01 | apples     | 10          |
| 2020-05-01 | oranges    | 8           |
| 2020-05-02 | apples     | 15          |
| 2020-05-02 | oranges    | 15          |
| 2020-05-03 | apples     | 20          |
| 2020-05-03 | oranges    | 0           |
| 2020-05-04 | apples     | 15          |
| 2020-05-04 | oranges    | 16          |
+------------+------------+-------------+

Result table:
+------------+--------------+
| sale_date  | diff         |
+------------+--------------+
| 2020-05-01 | 2            |
```

```
| 2020-05-02 | 0           |
| 2020-05-03 | 20          |
| 2020-05-04 | -1          |
+------------+-------------+

Day 2020-05-01, 10 apples and 8 oranges were sold (Difference  10 - 8 = 2).
Day 2020-05-02, 15 apples and 15 oranges were sold (Difference 15 - 15 = 0).
Day 2020-05-03, 20 apples and 0 oranges were sold (Difference 20 - 0 = 20).
Day 2020-05-04, 15 apples and 16 oranges were sold (Difference 15 - 16 = -1).
```

Solution

sql

```sql
#Solution 1:
SELECT sale_date, sum(CASE WHEN fruit='apples' THEN sold_num ELSE -sold_num E
FROM Sales
GROUP BY sale_date

#Solution 2:
SELECT sale_date, sold_num-sold AS diff
FROM
((SELECT *
FROM sales
WHERE fruit = 'apples') a
JOIN
(SELECT sale_date AS sale, fruit, sold_num AS sold
FROM sales
WHERE fruit = 'oranges') b
ON a.sale_date = b.sale)
```

1454. Active Users | Medium | LeetCode

Table `Accounts`:

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
```

```
| name          | varchar |
+---------------+---------+
the id is the primary key for this table.
This table contains the account id and the user name of each account.
```

Table `Logins` :

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| login_date    | date    |
+---------------+---------+
There is no primary key for this table, it may contain duplicates.
This table contains the account id of the user who logged in and the login da
```

Write an SQL query to find the id and the name of active users.

Active users are those who logged in to their accounts for 5 or more consecutive days.

Return the result table **ordered** by the id.

The query result format is in the following example:

```
Accounts table:
+----+----------+
| id | name     |
+----+----------+
| 1  | Winston  |
| 7  | Jonathan |
+----+----------+

Logins table:
+----+------------+
| id | login_date |
+----+------------+
| 7  | 2020-05-30 |
| 1  | 2020-05-30 |
| 7  | 2020-05-31 |
```

```
| 7  | 2020-06-01 |
| 7  | 2020-06-02 |
| 7  | 2020-06-02 |
| 7  | 2020-06-03 |
| 1  | 2020-06-07 |
| 7  | 2020-06-10 |
+----+------------+

Result table:
+----+----------+
| id | name     |
+----+----------+
| 7  | Jonathan |
+----+----------+
User Winston with id = 1 logged in 2 times only in 2 different days, so, Wins
User Jonathan with id = 7 logged in 7 times in 6 different days, five of ther
```

Follow up question: Can you write a general solution if the active users are those who logged in to their accounts for n or more consecutive days?

Solution

sql

```sql
#Solution 1:
WITH t1 AS (
SELECT id,login_date,
lead(login_date,4) OVER(PARTITION BY id ORDER BY login_date) date_5
FROM (SELECT DISTINCT * FROM Logins) b
)

SELECT DISTINCT a.id, a.name FROM t1
INNER JOIN accounts a
ON t1.id = a.id
WHERE DATEDIFF(t1.date_5,login_date) = 4
ORDER BY id

#Soltion 2:
SELECT *
FROM Accounts
WHERE id IN
```

```
    (SELECT DISTINCT t1.id
    FROM Logins AS t1 INNER JOIN Logins AS t2
    ON t1.id = t2.id AND datediff(t1.login_date, t2.login_date) BETWEEN 1 ANI
    GROUP BY t1.id, t1.login_date
    HAVING count(DISTINCT(t2.login_date)) = 4)
ORDER BY id
```

1459. Rectangles Area | Medium | LeetCode

Table: **Points**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| x_value       | int     |
| y_value       | int     |
+---------------+---------+
id is the primary key for this table.
Each point is represented as a 2D Dimensional (x_value, y_value).
```

Write an SQL query to report of all possible rectangles which can be formed by any two points of the table.

Each row in the result contains three columns (p1, p2, area) where:

- **p1** and **p2** are the id of two opposite corners of a rectangle and p1 < p2.
- Area of this rectangle is represented by the column **area**. Report the query in descending order by area in case of tie in ascending order by p1 and p2.

```
Points table:
+----------+-------------+-------------+
| id       | x_value     | y_value     |
+----------+-------------+-------------+
| 1        | 2           | 8           |
| 2        | 4           | 7           |
| 3        | 2           | 10          |
```

```
+---------+------------+------------+
```

Result table:

```
+---------+------------+------------+
| p1      | p2         | area       |
+---------+------------+------------+
| 2       | 3          | 6          |
| 1       | 2          | 2          |
+---------+------------+------------+
```

p1 should be less than p2 and area greater than 0.
p1 = 1 and p2 = 2, has an area equal to |2-4| * |8-7| = 2.
p1 = 2 and p2 = 3, has an area equal to |4-2| * |7-10| = 2.
p1 = 1 and p2 = 3 It's not possible because has an area equal to 0.

Solution

sql

```sql
SELECT t1.id AS p1, t2.id AS p2, ABS(t1.x_value-t2.x_value)*ABS(t1.y_value-t2
FROM Points AS t1 INNER JOIN Points AS t2
ON t1.id < t2.id
AND t1.x_value != t2.x_value AND t1.y_value != t2.y_value
ORDER BY area DESC, p1, p2
```

1468. Calculate Salaries | Medium | LeetCode

Table `Salaries`:

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| company_id   | int     |
| employee_id  | int     |
| employee_name| varchar |
| salary       | int     |
+--------------+---------+
(company_id, employee_id) is the primary key for this table.
This table contains the company id, the id, the name and the salary for an em
```

Write an SQL query to find the salaries of the employees after applying taxes.

The tax rate is calculated for each company based on the following criteria:

- 0% If the max salary of any employee in the company is less than 1000$.
- 24% If the max salary of any employee in the company is in the range [1000, 10000] inclusive.
- 49% If the max salary of any employee in the company is greater than 10000$.

Return the result table **in any order**. Round the salary to the nearest integer.

The query result format is in the following example:

Salaries table:

company_id	employee_id	employee_name	salary
1	1	Tony	2000
1	2	Pronub	21300
1	3	Tyrrox	10800
2	1	Pam	300
2	7	Bassem	450
2	9	Hermione	700
3	7	Bocaben	100
3	2	Ognjen	2200
3	13	Nyancat	3300
3	15	Morninngcat	1866

Result table:

company_id	employee_id	employee_name	salary
1	1	Tony	1020
1	2	Pronub	10863
1	3	Tyrrox	5508
2	1	Pam	300
2	7	Bassem	450
2	9	Hermione	700
3	7	Bocaben	76
3	2	Ognjen	1672

```
| 3          | 13          | Nyancat       | 2508   |
| 3          | 15          | Morninngcat   | 5911   |
+------------+-------------+---------------+--------+
For company 1, Max salary is 21300. Employees in company 1 have taxes = 49%
For company 2, Max salary is 700. Employees in company 2 have taxes = 0%
For company 3, Max salary is 7777. Employees in company 3 have taxes = 24%
The salary after taxes = salary - (taxes percentage / 100) * salary
For example, Salary for Morninngcat (3, 15) after taxes = 7777 - 7777 * (24 /
```

Solution

sql

```sql
#Solution 1:
WITH t1 AS (
SELECT company_id, employee_id, employee_name, salary AS sa, MAX(salary) OVEI
FROM salaries)

SELECT company_id, employee_id, employee_name,
CASE WHEN t1.maximum<1000 THEN t1.sa
WHEN t1.maximum BETWEEN 1000 AND 10000 THEN ROUND(t1.sa*.76,0)
ELSE ROUND(t1.sa*.51,0)
END AS salary
FROM t1

#Soltion 2:
SELECT Salaries.company_id, Salaries.employee_id, Salaries.employee_name,
    ROUND(CASE WHEN salary_max<1000 THEN Salaries.salary
               WHEN salary_max>=1000 AND salary_max<=10000 THEN Salaries.sala
               ELSE Salaries.salary * 0.51 END, 0) AS salary
FROM Salaries INNER JOIN (
    SELECT company_id, MAX(salary) AS salary_max
    FROM Salaries
    GROUP BY company_id) AS t
ON Salaries.company_id = t.company_id
```

1479. Sales by Day of the Week | Hard | LeetCode

Table: **Orders**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| customer_id   | int     |
| order_date    | date    |
| item_id       | varchar |
| quantity      | int     |
+---------------+---------+
(ordered_id, item_id) is the primary key for this table.
This table contains information of the orders placed.
order_date is the date when item_id was ordered by the customer with id custo
```

Table: **Items**

```
+--------------------+---------+
| Column Name        | Type    |
+--------------------+---------+
| item_id            | varchar |
| item_name          | varchar |
| item_category      | varchar |
+--------------------+---------+
item_id is the primary key for this table.
item_name is the name of the item.
item_category is the category of the item.
```

You are the business owner and would like to obtain a sales report for category items and day of the week.

Write an SQL query to report how many units in each category have been ordered on each **day of the week**.

Return the result table **ordered** by category.

The query result format is in the following example:

```
Orders table:
+------------+-------------+-------------+-------------+-------------+
| order_id   | customer_id | order_date  | item_id     | quantity    |
```

```
+------------+--------------+-------------+--------------+-------------+
| 1          | 1            | 2020-06-01  | 1            | 10          |
| 2          | 1            | 2020-06-08  | 2            | 10          |
| 3          | 2            | 2020-06-02  | 1            | 5           |
| 4          | 3            | 2020-06-03  | 3            | 5           |
| 5          | 4            | 2020-06-04  | 4            | 1           |
| 6          | 4            | 2020-06-05  | 5            | 5           |
| 7          | 5            | 2020-06-05  | 1            | 10          |
| 8          | 5            | 2020-06-14  | 4            | 5           |
| 9          | 5            | 2020-06-21  | 3            | 5           |
+------------+--------------+-------------+--------------+-------------+
```

Items table:

```
+------------+----------------+---------------+
| item_id    | item_name      | item_category |
+------------+----------------+---------------+
| 1          | LC Alg. Book   | Book          |
| 2          | LC DB. Book    | Book          |
| 3          | LC SmarthPhone | Phone         |
| 4          | LC Phone 2020  | Phone         |
| 5          | LC SmartGlass  | Glasses       |
| 6          | LC T-Shirt XL  | T-Shirt       |
+------------+----------------+---------------+
```

Result table:

```
+------------+-----------+-----------+-----------+-----------+-----------+--
| Category   | Monday    | Tuesday   | Wednesday | Thursday  | Friday    | Sa
+------------+-----------+-----------+-----------+-----------+-----------+--
| Book       | 20        | 5         | 0         | 0         | 10        | 0
| Glasses    | 0         | 0         | 0         | 0         | 5         | 0
| Phone      | 0         | 0         | 5         | 1         | 0         | 0
| T-Shirt    | 0         | 0         | 0         | 0         | 0         | 0
+------------+-----------+-----------+-----------+-----------+-----------+--
On Monday (2020-06-01, 2020-06-08) were sold a total of 20 units (10 + 10) ir
On Tuesday (2020-06-02) were sold a total of 5 units  in the category Book (:
On Wednesday (2020-06-03) were sold a total of 5 units in the category Phone
On Thursday (2020-06-04) were sold a total of 1 unit in the category Phone (:
On Friday (2020-06-05) were sold 10 units in the category Book (ids: 1, 2) ar
On Saturday there are no items sold.
On Sunday (2020-06-14, 2020-06-21) were sold a total of 10 units (5 +5) in th
There are no sales of T-Shirt.
```

Solution

sql

```
WITH t1 AS(
SELECT DISTINCT item_category,
CASE WHEN dayname(order_date)='Monday' THEN SUM(quantity) OVER(PARTITION BY :
CASE WHEN dayname(order_date)='Tuesday' THEN SUM(quantity) OVER(PARTITION BY
CASE WHEN dayname(order_date)='Wednesday' THEN SUM(quantity) OVER(PARTITION E
CASE WHEN dayname(order_date)='Thursday' THEN SUM(quantity) OVER(PARTITION BY
CASE WHEN dayname(order_date)='Friday' THEN SUM(quantity) OVER(PARTITION BY :
CASE WHEN dayname(order_date)='Saturday' THEN SUM(quantity) OVER(PARTITION BY
CASE WHEN dayname(order_date)='Sunday' THEN SUM(quantity) OVER(PARTITION BY :
FROM orders o
RIGHT JOIN items i
USING (item_id))

SELECT item_category AS category, SUM(Monday) AS Monday, SUM(Tuesday) AS Tues
SUM(Friday) Friday, SUM(Saturday) Saturday, SUM(Sunday) Sunday
FROM t1
GROUP BY item_category
```

1484. Group Sold Products By The Date | Easy | LeetCode

Table **Activities**:

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| sell_date   | date    |
| product     | varchar |
+-------------+---------+
There is no primary key for this table, it may contains duplicates.
Each row of this table contains the product name and the date it was sold in
```

Write an SQL query to find for each date, the number of distinct products sold and their names.

The sold-products names for each date should be sorted lexicographically.

Return the result table ordered by `sell_date`.

The query result format is in the following example.

```
Activities table:
+------------+-------------+
| sell_date  | product     |
+------------+-------------+
| 2020-05-30 | Headphone   |
| 2020-06-01 | Pencil      |
| 2020-06-02 | Mask        |
| 2020-05-30 | Basketball  |
| 2020-06-01 | Bible       |
| 2020-06-02 | Mask        |
| 2020-05-30 | T-Shirt     |
+------------+-------------+

Result table:
+------------+----------+------------------------------+
| sell_date  | num_sold | products                     |
+------------+----------+------------------------------+
| 2020-05-30 | 3        | Basketball,Headphone,T-shirt |
| 2020-06-01 | 2        | Bible,Pencil                 |
| 2020-06-02 | 1        | Mask                         |
+------------+----------+------------------------------+
For 2020-05-30, Sold items were (Headphone, Basketball, T-shirt), we sort the
For 2020-06-01, Sold items were (Pencil, Bible), we sort them lexicographica
For 2020-06-02, Sold item is (Masks), we just return it.
```

Solution

sql

```sql
SELECT sell_date, COUNT(DISTINCT product) AS num_sold, group_concat(DISTINCT
FROM activities
GROUP BY 1
ORDER BY 1
```

1495. Friendly Movies Streamed Last Month | Easy | LeetCode

Table: `TVProgram`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| program_date  | date    |
| content_id    | int     |
| channel       | varchar |
+---------------+---------+
(program_date, content_id) is the primary key for this table.
This table contains information of the programs on the TV.
content_id is the id of the program in some channel on the TV.
```

Table: `Content`

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| content_id       | varchar |
| title            | varchar |
| Kids_content     | enum    |
| content_type     | varchar |
+------------------+---------+
content_id is the primary key for this table.
Kids_content is an enum that takes one of the values ('Y', 'N') where:
'Y' means is content for kids otherwise 'N' is not content for kids.
content_type is the category of the content as movies, series, etc.
```

Write an SQL query to report the distinct titles of the kid-friendly movies streamed in June 2020.

Return the result table in any order.

.e query result format is in the following example.

TVProgram table:

program_date	content_id	channel
2020-06-10 08:00	1	LC-Channel
2020-05-11 12:00	2	LC-Channel
2020-05-12 12:00	3	LC-Channel
2020-05-13 14:00	4	Disney Ch
2020-06-18 14:00	4	Disney Ch
2020-07-15 16:00	5	Disney Ch

Content table:

content_id	title	Kids_content	content_type
1	Leetcode Movie	N	Movies
2	Alg. for Kids	Y	Series
3	Database Sols	N	Series
4	Aladdin	Y	Movies
5	Cinderella	Y	Movies

Result table:

title
Aladdin

"Leetcode Movie" is not a content for kids.
"Alg. for Kids" is not a movie.
"Database Sols" is not a movie
"Alladin" is a movie, content for kids and was streamed in June 2020.
"Cinderella" was not streamed in June 2020.

Solution

sql

```sql
SELCT DISTINCT title
FROM
```

```
(SELCT content_id, title
FROM content
WHERE kids_content = 'Y' AND content_type = 'Movies') a
JOIN
tvprogram USING (content_id)
WHERE month(program_date) = 6
```

1501. Countries You Can Safely Invest In | Medium | LeetCode

Table **Person**:

```
+---------------+---------+
| Column Name    | Type    |
+---------------+---------+
| id             | int     |
| name           | varchar |
| phone_number   | varchar |
+---------------+---------+
id is the primary key for this table.
Each row of this table contains the name of a person and their phone number.
Phone number will be in the form 'xxx-yyyyyyy' where xxx is the country code
```

Table **Country**:

```
+---------------+---------+
| Column Name    | Type    |
+---------------+---------+
| name           | varchar |
| country_code   | varchar |
+---------------+---------+
country_code is the primary key for this table.
Each row of this table contains the country name and its code. country_code v
```

ble **Calls**:

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| caller_id   | int  |
| callee_id   | int  |
| duration    | int  |
+-------------+------+
There is no primary key for this table, it may contain duplicates.
Each row of this table contains the caller id, callee id and the duration of
A telecommunications company wants to invest in new countries. The country ir
```

Write an SQL query to find the countries where this company can invest.

Return the result table in any order.

The query result format is in the following example.

```
Person table:
+----+----------+--------------+
| id | name     | phone_number |
+----+----------+--------------+
| 3  | Jonathan | 051-1234567  |
| 12 | Elvis    | 051-7654321  |
| 1  | Moncef   | 212-1234567  |
| 2  | Maroua   | 212-6523651  |
| 7  | Meir     | 972-1234567  |
| 9  | Rachel   | 972-0011100  |
+----+----------+--------------+

Country table:
+----------+--------------+
| name     | country_code |
+----------+--------------+
| Peru     | 051          |
| Israel   | 972          |
| Morocco  | 212          |
| Germany  | 049          |
| Ethiopia | 251          |
+----------+--------------+
```

Calls table:

caller_id	callee_id	duration
1	9	33
2	9	4
1	2	59
3	12	102
3	12	330
12	3	5
7	9	13
7	1	3
9	7	1
1	7	7

Result table:

country
Peru

The average call duration for Peru is (102 + 102 + 330 + 330 + 5 + 5) / 6 = 1
The average call duration for Israel is (33 + 4 + 13 + 13 + 3 + 1 + 1 + 7) /
The average call duration for Morocco is (33 + 4 + 59 + 59 + 3 + 7) / 6 = 27.
Global call duration average = (2 * (33 + 3 + 59 + 102 + 330 + 5 + 13 + 3 + 1
Since Peru is the only country where average call duration is greater than th

Solution

sql

```sql
WITH t1 AS(
SELECT caller_id AS id, duration AS total
FROM
(SELECT caller_id, duration
FROM calls
UNION ALL
SELECT callee_id, duration
FROM calls) a
)
SELECT name AS country
```

```
FROM
(SELECT distinct avg(total) OVER(PARTITION BY code) AS avg_call, avg(total) (
FROM
((SELECT *, coalesce(total,0) AS duration, SUBSTRING(phone_number FROM 1 for
FROM person RIGHT JOIN t1
USING (id)) b
join country c
ON c.country_code = b.code)) d
WHERE avg_call > global_avg
```

1511. Customer Order Frequency | Easy | LeetCode

Table: **Customers**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| name          | varchar |
| country       | varchar |
+---------------+---------+
customer_id is the primary key for this table.
This table contains information of the customers in the company.
```

Table: **Product**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| description   | varchar |
| price         | int     |
+---------------+---------+
product_id is the primary key for this table.
This table contains information of the products in the company.
price is the product cost.
```

Table: **Orders**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| order_id     | int     |
| customer_id  | int     |
| product_id   | int     |
| order_date   | date    |
| quantity     | int     |
+--------------+---------+
order_id is the primary key for this table.
This table contains information on customer orders.
customer_id is the id of the customer who bought "quantity" products with id
Order_date is the date in format ('YYYY-MM-DD') when the order was shipped.
```

Write an SQL query to report the customer_id and customer_name of customers who have spent at least $100 in each month of June and July 2020.

Return the result table in any order.

The query result format is in the following example.

```
Customers
+--------------+-----------+-------------+
| customer_id  | name      | country     |
+--------------+-----------+-------------+
| 1            | Winston   | USA         |
| 2            | Jonathan  | Peru        |
| 3            | Moustafa  | Egypt       |
+--------------+-----------+-------------+

Product
+--------------+-------------+-------------+
| product_id   | description | price       |
+--------------+-------------+-------------+
| 10           | LC Phone    | 300         |
| 20           | LC T-Shirt  | 10          |
| 30           | LC Book     | 45          |
| 40           | LC Keychain | 2           |
```

```
+--------------+-------------+-------------+
```

Orders

```
+--------------+-------------+-------------+-------------+-----------+
| order_id     | customer_id | product_id  | order_date  | quantity  |
+--------------+-------------+-------------+-------------+-----------+
| 1            | 1           | 10          | 2020-06-10  | 1         |
| 2            | 1           | 20          | 2020-07-01  | 1         |
| 3            | 1           | 30          | 2020-07-08  | 2         |
| 4            | 2           | 10          | 2020-06-15  | 2         |
| 5            | 2           | 40          | 2020-07-01  | 10        |
| 6            | 3           | 20          | 2020-06-24  | 2         |
| 7            | 3           | 30          | 2020-06-25  | 2         |
| 9            | 3           | 30          | 2020-05-08  | 3         |
+--------------+-------------+-------------+-------------+-----------+
```

Result table:

```
+--------------+------------+
| customer_id  | name       |
+--------------+------------+
| 1            | Winston    |
+--------------+------------+
```

Winston spent $300 (300 * 1) in June and $100 (10 * 1 + 45 * 2) in July 202(
Jonathan spent $600 (300 * 2) in June and $20 (2 * 10) in July 2020.
Moustafa spent $110 (10 * 2 + 45 * 2) in June and $0 in July 2020.

Solution

sql

```sql
#Solution 1:
SELECT o.customer_id, name
JOIN Product p
ON o.product_id = p.product_id
JOIN Customers c
ON o.customer_id = c.customer_id
GROUP BY 1, 2
HAVING SUM(CASE WHEN date_format(order_date, '%Y-%m')='2020-06'
THEN price*quantity END) >= 100
AND
SUM(CASE WHEN date_format(order_date, '%Y-%m')='2020-07'
THEN price*quantity END) >= 100;
```

```
#Solution 2:
SELECT customer_id, name
FROM
(
    SELECT o.customer_id, c.name,
        sum(CASE WHEN left(o.order_date,7) = '2020-06' THEN p.price * o.quant
        sum(CASE WHEN left(o.order_date,7) = '2020-07' THEN p.price * o.quant
    FROM Orders o
    LEFT JOIN Customers c ON o.customer_id = c.customer_id
    lEFT JOIN Product p ON o.product_id = p.product_id
    GROUP BY o.customer_id
    HAVING JuneSpend >= 100 AND JulySpend >= 100
) AS temp

#Solution 3:
SELECT o.customer_id, c.name
FROM Customers c, Product p, Orders o
WHERE c.customer_id = o.customer_id AND p.product_id = o.product_id
GROUP BY o.customer_id
HAVING
(
    SUM(CASE WHEN o.order_date LIKE '2020-06%' THEN o.quantity*p.price ELSE (
    and
    SUM(CASE WHEN o.order_date LIKE '2020-07%' THEN o.quantity*p.price ELSE (
);
```

1517. Find Users With Valid E-Mails | Easy | LeetCode

Table: **Users**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| name          | varchar |
| mail          | varchar |
+---------------+---------+
user_id is the primary key for this table.
```

```
This table contains information of the users signed up in a website. Some e-r
```

Write an SQL query to find the users who have **valid emails**.

A valid e-mail has a prefix name and a domain where:

- **The prefix name** is a string that may contain letters (upper or lower case), digits, underscore `'_'`, period `'.'` and/or dash `'-'`. The prefix name **must** start with a letter.
- The domain is `'@leetcode.com'`.
  Return the result table in any order.

The query result format is in the following example.

```
Users
+---------+-----------+-------------------------+
| user_id | name      | mail                    |
+---------+-----------+-------------------------+
| 1       | Winston   | winston@leetcode.com    |
| 2       | Jonathan  | jonathanisgreat         |
| 3       | Annabelle | bella-@leetcode.com     |
| 4       | Sally     | sally.come@leetcode.com |
| 5       | Marwan    | quarz#2020@leetcode.com |
| 6       | David     | david69@gmail.com       |
| 7       | Shapiro   | .shapo@leetcode.com     |
+---------+-----------+-------------------------+

Result table:
+---------+-----------+-------------------------+
| user_id | name      | mail                    |
+---------+-----------+-------------------------+
| 1       | Winston   | winston@leetcode.com    |
| 3       | Annabelle | bella-@leetcode.com     |
| 4       | Sally     | sally.come@leetcode.com |
+---------+-----------+-------------------------+
The mail of user 2 doesn't have a domain.
The mail of user 5 has # sign which is not allowed.
The mail of user 6 doesn't have leetcode domain.
The mail of user 7 starts with a period.
```

Solution

sql

```sql
#Solution 1:
SELECT user_id, name, mail
FROM Users
WHERE mail regexp "^[a-zA-Z]+[a-zA-Z0-9_\\./\\-]{0,}@leetcode\\.com$"
ORDER BY user_id

#Solution 2:
SELECT * FROM Users
WHERE regexp_like(mail, '^[A-Za-z]+[A-Za-z0-9\_\.\-]*@leetcode.com')
```

1527. Patients With a Condition | Easy | LeetCode

Table: **Patients**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| patient_id   | int     |
| patient_name | varchar |
| conditions   | varchar |
+--------------+---------+
patient_id is the primary key for this table.
'conditions' contains 0 or more code separated by spaces.
This table contains information of the patients in the hospital.
```

Write an SQL query to report the patient_id, patient_name all conditions of patients who have Type I Diabetes. Type I Diabetes always starts with **DIAB1** prefix

Return the result table in any order.

The query result format is in the following example.

```
Patients
+------------+--------------+--------------+
```

```
| patient_id | patient_name | conditions   |
+------------+--------------+--------------+
| 1          | Daniel       | YFEV COUGH   |
| 2          | Alice        |              |
| 3          | Bob          | DIAB100 MYOP |
| 4          | George       | ACNE DIAB100 |
| 5          | Alain        | DIAB201      |
+------------+--------------+--------------+

Result table:
+------------+--------------+--------------+
| patient_id | patient_name | conditions   |
+------------+--------------+--------------+
| 3          | Bob          | DIAB100 MYOP |
| 4          | George       | ACNE DIAB100 |
+------------+--------------+--------------+
Bob and George both have a condition that starts with DIAB1.
```

Solution

sql

```sql
SELECT patient_id, patient_name, conditions
FROM Patients
WHERE conditions LIKE '%DIAB1%'
```

1532. The Most Recent Three Orders | Medium | LeetCode

Table: `Customers`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| name          | varchar |
+---------------+---------+
customer_id is the primary key for this table.
```

```
This table contains information about customers.
```

Table: **Orders**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| order_id     | int     |
| order_date   | date    |
| customer_id  | int     |
| cost         | int     |
+--------------+---------+
order_id is the primary key for this table.
This table contains information about the orders made customer_id.
Each customer has one order per day.
```

Write an SQL query to find the most recent 3 orders of each user. If a user ordered less than 3 orders return all of their orders.

Return the result table sorted by `customer_name` in ascending order and in case of a tie by the `customer_id` in ascending order. If there still a tie, order them by the `order_date` in descending order.

The query result format is in the following example:

```
Customers
+-------------+-----------+
| customer_id | name      |
+-------------+-----------+
| 1           | Winston   |
| 2           | Jonathan  |
| 3           | Annabelle |
| 4           | Marwan    |
| 5           | Khaled    |
+-------------+-----------+

Orders
+----------+------------+-------------+------+
| order_id | order_date | customer_id | cost |
```

```
+----------+------------+-------------+------+
| 1        | 2020-07-31 | 1           | 30   |
| 2        | 2020-07-30 | 2           | 40   |
| 3        | 2020-07-31 | 3           | 70   |
| 4        | 2020-07-29 | 4           | 100  |
| 5        | 2020-06-10 | 1           | 1010 |
| 6        | 2020-08-01 | 2           | 102  |
| 7        | 2020-08-01 | 3           | 111  |
| 8        | 2020-08-03 | 1           | 99   |
| 9        | 2020-08-07 | 2           | 32   |
| 10       | 2020-07-15 | 1           | 2    |
+----------+------------+-------------+------+
```

Result table:

```
+---------------+-------------+----------+------------+
| customer_name | customer_id | order_id | order_date |
+---------------+-------------+----------+------------+
| Annabelle     | 3           | 7        | 2020-08-01 |
| Annabelle     | 3           | 3        | 2020-07-31 |
| Jonathan      | 2           | 9        | 2020-08-07 |
| Jonathan      | 2           | 6        | 2020-08-01 |
| Jonathan      | 2           | 2        | 2020-07-30 |
| Marwan        | 4           | 4        | 2020-07-29 |
| Winston       | 1           | 8        | 2020-08-03 |
| Winston       | 1           | 1        | 2020-07-31 |
| Winston       | 1           | 10       | 2020-07-15 |
+---------------+-------------+----------+------------+
```

Winston has 4 orders, we discard the order of "2020-06-10" because it is the

Annabelle has only 2 orders, we return them.

Jonathan has exactly 3 orders.

Marwan ordered only one time.

We sort the result table by customer_name in ascending order, by customer_id

Follow-up:

Can you write a general solution for the most recent `n` orders?

Solution

sql

```sql
WITH tmp AS (
SELECT a.name, a.customer_id, b.order_id, b.order_date,
ROW_NUMBER() OVER(PARTITION BY a.name, a.customer_id ORDER BY b.order_date DI
```

```
FROM Customers AS a
JOIN Orders AS b
ON a.customer_id = b.customer_id
)

SELECT name AS customer_name, customer_id, order_id, order_date
FROM tmp
WHERE rnk <= 3
ORDER BY customer_name, customer_id, order_date DESC;
```

1543. Fix Product Name Format | Easy | LeetCode

Table: `Sales`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| sale_id      | int     |
| product_name | varchar |
| sale_date    | date    |
+--------------+---------+
sale_id is the primary key for this table.
Each row of this table contains the product name and the date it was sold.
```

Since table Sales was filled manually in the year 2000, `product_name` may contain leading and/or trailing white spaces, also they are case-insensitive.

Write an SQL query to report

- `product_name` in lowercase without leading or trailing white spaces.
- `sale_date` in the format `('YYYY-MM')`
- `total` the number of times the product was sold in this month.
  Return the result table ordered by `product_name` in **ascending order**, in case of a tie order it by `sale_date` in ascending order.

.e query result format is in the following example.

Sales

```
+------------+-----------------+-------------+
| sale_id    | product_name    | sale_date   |
+------------+-----------------+-------------+
| 1          |      LCPHONE    | 2000-01-16  |
| 2          |    LCPhone      | 2000-01-17  |
| 3          |     LcPhOnE     | 2000-02-18  |
| 4          |      LCKeyCHAiN | 2000-02-19  |
| 5          |   LCKeyChain    | 2000-02-28  |
| 6          | Matryoshka      | 2000-03-31  |
+------------+-----------------+-------------+
```

Result table:

```
+--------------+--------------+----------+
| product_name | sale_date    | total    |
+--------------+--------------+----------+
| lcphone      | 2000-01      | 2        |
| lckeychain   | 2000-02      | 2        |
| lcphone      | 2000-02      | 1        |
| matryoshka   | 2000-03      | 1        |
+--------------+--------------+----------+
```

In January, 2 LcPhones were sold, please note that the product names are not
In Februery, 2 LCKeychains and 1 LCPhone were sold.
In March, 1 matryoshka was sold.

Solution

sql

```sql
SELECT TRIM(LOWER(product_name)) AS product_name,
       DATE_FORMAT(sale_date, '%Y-%m') AS sale_date,
       COUNT(*) AS total
FROM Sales
GROUP BY 1, DATE_FORMAT(sale_date, '%Y-%m')
ORDER BY 1, 2;
```

3549. The Most Recent Orders for Each Product | Medium | LeetCode

Table: **Customers**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| name          | varchar |
+---------------+---------+
customer_id is the primary key for this table.
This table contains information about the customers.
```

Table: **Orders**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| order_date    | date    |
| customer_id   | int     |
| product_id    | int     |
+---------------+---------+
order_id is the primary key for this table.
This table contains information about the orders made by customer_id.
There will be no product ordered by the same user more than once in one day.
```

Table: **Products**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| product_name  | varchar |
| price         | int     |
+---------------+---------+
product_id is the primary key for this table.
This table contains information about the Products.
```

Write an SQL query to find the most recent order(s) of each product.

Return the result table sorted by `product_name` in **ascending** order and in case of a tie the `product_id` in **ascending** order. If there still a tie, order them by the `order_id` in **ascending** order.

The query result format is in the following example:

```
Customers
+-------------+-----------+
| customer_id | name      |
+-------------+-----------+
| 1           | Winston   |
| 2           | Jonathan  |
| 3           | Annabelle |
| 4           | Marwan    |
| 5           | Khaled    |
+-------------+-----------+

Orders
+----------+------------+-------------+------------+
| order_id | order_date | customer_id | product_id |
+----------+------------+-------------+------------+
| 1        | 2020-07-31 | 1           | 1          |
| 2        | 2020-07-30 | 2           | 2          |
| 3        | 2020-08-29 | 3           | 3          |
| 4        | 2020-07-29 | 4           | 1          |
| 5        | 2020-06-10 | 1           | 2          |
| 6        | 2020-08-01 | 2           | 1          |
| 7        | 2020-08-01 | 3           | 1          |
| 8        | 2020-08-03 | 1           | 2          |
| 9        | 2020-08-07 | 2           | 3          |
| 10       | 2020-07-15 | 1           | 2          |
+----------+------------+-------------+------------+

Products
+------------+--------------+-------+
| product_id | product_name | price |
+------------+--------------+-------+
| 1          | keyboard     | 120   |
| 2          | mouse        | 80    |
| 3          | screen       | 600   |
| 4          | hard disk    | 450   |
```

```
+------------+-------------+-------+
```

Result table:

```
+--------------+------------+----------+------------+
| product_name | product_id | order_id | order_date |
+--------------+------------+----------+------------+
| keyboard     | 1          | 6        | 2020-08-01 |
| keyboard     | 1          | 7        | 2020-08-01 |
| mouse        | 2          | 8        | 2020-08-03 |
| screen       | 3          | 3        | 2020-08-29 |
+--------------+------------+----------+------------+
keyboard's most recent order is in 2020-08-01, it was ordered two times this
mouse's most recent order is in 2020-08-03, it was ordered only once this da
screen's most recent order is in 2020-08-29, it was ordered only once this d
The hard disk was never ordered and we don't include it in the result table.
```

Solution

sql

```sql
SELECT p.product_name, o.product_id, o.order_id, o.order_date
FROM(
    SELECT product_id, order_id, order_date,
    RANK() OVER(PARTITION BY product_id ORDER BY order_date DESC) AS seq
    FROM orders
) o
LEFT JOIN products p
    ON o.product_id = p.product_id
WHERE o.seq = 1
ORDER BY 1,2,3
```

1555. Bank Account Summary | Medium | LeetCode

Table: **Users**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| user_id     | int     |
```

```
| user_name    | varchar |
| credit       | int     |
+--------------+---------+
user_id is the primary key for this table.
Each row of this table contains the current credit information for each user.
```

Table: **Transaction**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| trans_id      | int     |
| paid_by       | int     |
| paid_to       | int     |
| amount        | int     |
| transacted_on | date    |
+---------------+---------+
trans_id is the primary key for this table.
Each row of this table contains the information about the transaction in the
User with id (paid_by) transfer money to user with id (paid_to).
```

Leetcode Bank (LCB) helps its coders in making virtual payments. Our bank records all transactions in the table Transaction, we want to find out the current balance of all users and check wheter they have breached their credit limit (If their current credit is less than 0).

Write an SQL query to report.

- **user_id**
- **user_name**
- **credit** , current balance after performing transactions.
- **credit_limit_breached** , check credit_limit ("Yes" or "No") Return the result table in **any** order.

The query result format is in the following example.

```
Users table:
```

```
+------------+--------------+-------------+
| user_id    | user_name    | credit      |
+------------+--------------+-------------+
| 1          | Moustafa     | 100         |
| 2          | Jonathan     | 200         |
| 3          | Winston      | 10000       |
| 4          | Luis         | 800         |
+------------+--------------+-------------+

Transaction table:
+------------+------------+------------+---------+---------------+
| trans_id   | paid_by    | paid_to    | amount  | transacted_on |
+------------+------------+------------+---------+---------------+
| 1          | 1          | 3          | 400     | 2020-08-01    |
| 2          | 3          | 2          | 500     | 2020-08-02    |
| 3          | 2          | 1          | 200     | 2020-08-03    |
+------------+------------+------------+---------+---------------+

Result table:
+------------+------------+------------+----------------------+
| user_id    | user_name  | credit     | credit_limit_breached |
+------------+------------+------------+----------------------+
| 1          | Moustafa   | -100       | Yes                  |
| 2          | Jonathan   | 500        | No                   |
| 3          | Winston    | 9990       | No                   |
| 4          | Luis       | 800        | No                   |
+------------+------------+------------+----------------------+
Moustafa paid $400 on "2020-08-01" and received $200 on "2020-08-03", credit
Jonathan received $500 on "2020-08-02" and paid $200 on "2020-08-08", credit
Winston received $400 on "2020-08-01" and paid $500 on "2020-08-03", credit
Luis didn't received any transfer, credit = $800
```

Solution

sql

```sql
SELECT Users.user_id AS user_id
    , Users.user_name AS user_name
    , credit+IFNULL(SUM(trans),0) AS credit
    , CASE WHEN credit+IFNULL(SUM(trans),0)>0 THEN 'No' ELSE 'Yes' END AS cre
FROM(
    SELECT paid_by AS user_id, -amount AS trans
```

```
    FROM Transaction
    UNION ALL
    SELECT paid_to AS user_id, amount AS trans
    FROM Transaction
) t RIGHT JOIN users ON t.user_id=users.user_id
GROUP BY user_id
```

1565. Unique Orders and Customers Per Month | Easy | LeetCode

Table: `Orders`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| order_date    | date    |
| customer_id   | int     |
| invoice       | int     |
+---------------+---------+
order_id is the primary key for this table.
This table contains information about the orders made by customer_id.
```

Write an SQL query to find the number of **unique orders** and the number of **unique users** with invoices > **$20** for each **different month**.

Return the result table sorted in **any order**.

The query result format is in the following example:

```
Orders
+----------+------------+-------------+------------+
| order_id | order_date | customer_id | invoice    |
+----------+------------+-------------+------------+
| 1        | 2020-09-15 | 1           | 30         |
| 2        | 2020-09-17 | 2           | 90         |
| 3        | 2020-10-06 | 3           | 20         |
| 4        | 2020-10-20 | 3           | 21         |
```

```
| 5         | 2020-11-10 | 1           | 10          |
| 6         | 2020-11-21 | 2           | 15          |
| 7         | 2020-12-01 | 4           | 55          |
| 8         | 2020-12-03 | 4           | 77          |
| 9         | 2021-01-07 | 3           | 31          |
| 10        | 2021-01-15 | 2           | 20          |
+-----------+------------+-------------+-------------+

Result table:
+---------+-------------+----------------+
| month   | order_count | customer_count |
+---------+-------------+----------------+
| 2020-09 | 2           | 2              |
| 2020-10 | 1           | 1              |
| 2020-12 | 2           | 1              |
| 2021-01 | 1           | 1              |
+---------+-------------+----------------+
In September 2020 we have two orders from 2 different customers with invoices
In October 2020 we have two orders from 1 customer, and only one of the two
In November 2020 we have two orders from 2 different customers but invoices
In December 2020 we have two orders from 1 customer both with invoices > $20.
In January 2021 we have two orders from 2 different customers, but only one
```

Solution

sql

```sql
#Solution 1:
SELECT DATE_FORMAT(order_date, '%Y-%m') AS month, COUNT(DISTINCT order_id) AS
    FROM Orders
    WHERE invoice > 20
    GROUP BY YEAR(order_date), MONTH(order_date);

#Solution 2:
SELECT LEFT(order_date, 7) AS month, COUNT(DISTINCT order_id) AS order_count,
    COUNT(DISTINCT customer_id) AS customer_count
FROM orders
WHERE invoice > 20
GROUP BY month
```

1571. Warehouse Manager | Easy | LeetCode

Table: `Warehouse`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| name         | varchar |
| product_id   | int     |
| units        | int     |
+--------------+---------+
(name, product_id) is the primary key for this table.
Each row of this table contains the information of the products in each warel
```

Table: `Products`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| product_name  | varchar |
| Width         | int     |
| Length        | int     |
| Height        | int     |
+---------------+---------+
product_id is the primary key for this table.
Each row of this table contains the information about the product dimensions
```

Write an SQL query to report, How much cubic feet of **volume** does the inventory occupy in each warehouse.

- warehouse_name
- volume
  Return the result table in **any** order.

e query result format is in the following example.

Warehouse table:

name	product_id	units
LCHouse1	1	1
LCHouse1	2	10
LCHouse1	3	5
LCHouse2	1	2
LCHouse2	2	2
LCHouse3	4	1

Products table:

product_id	product_name	Width	Length	Height
1	LC-TV	5	50	40
2	LC-KeyChain	5	5	5
3	LC-Phone	2	10	10
4	LC-T-Shirt	4	10	20

Result table:

warehouse_name	volume
LCHouse1	12250
LCHouse2	20250
LCHouse3	800

Volume of product_id = 1 (LC-TV), 5x50x40 = 10000
Volume of product_id = 2 (LC-KeyChain), 5x5x5 = 125
Volume of product_id = 3 (LC-Phone), 2x10x10 = 200
Volume of product_id = 4 (LC-T-Shirt), 4x10x20 = 800
LCHouse1: 1 unit of LC-TV + 10 units of LC-KeyChain + 5 units of LC-Phone.
Total volume: 1*10000 + 10*125 + 5*200 = 12250 cubic feet
LCHouse2: 2 units of LC-TV + 2 units of LC-KeyChain.
Total volume: 2*10000 + 2*125 = 20250 cubic feet
LCHouse3: 1 unit of LC-T-Shirt.
Total volume: 1*800 = 800 cubic feet.

Solution

```sql
SELECT a.name AS warehouse_name,
SUM(a.units * b.Width * b.Length * b.Height) AS volume
FROM Warehouse AS a
LEFT JOIN Products AS b
ON a.product_id = b.product_id
GROUP BY a.name;
```

1581. Customer Who Visited but Did Not Make Any Transactions | Easy | LeetCode

Table: **Visits**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| visit_id    | int     |
| customer_id | int     |
+-------------+---------+
visit_id is the primary key for this table.
This table contains information about the customers who visited the mall.
```

Table: **Transactions**

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| transaction_id | int     |
| visit_id       | int     |
| amount         | int     |
+----------------+---------+
transaction_id is the primary key for this table.
This table contains information about the customers who visited the mall.
```

Write an SQL query to find the IDs of the users who visited without making any transactions and the number of times they made these types of visits.

Return the result table sorted in **any orders**.

The query result format is in the following example:

```
Visits
+----------+-------------+
| visit_id | customer_id |
+----------+-------------+
| 1        | 23          |
| 2        | 9           |
| 4        | 30          |
| 5        | 54          |
| 6        | 96          |
| 7        | 54          |
| 8        | 54          |
+----------+-------------+

Transactions
+----------------+----------+--------+
| transaction_id | visit_id | amount |
+----------------+----------+--------+
| 2              | 5        | 310    |
| 3              | 5        | 300    |
| 9              | 5        | 200    |
| 12             | 1        | 910    |
| 13             | 2        | 970    |
+----------------+----------+--------+

Result table:
+-------------+---------------+
| customer_id | count_no_trans |
+-------------+---------------+
| 54          | 2             |
| 30          | 1             |
| 96          | 1             |
+-------------+---------------+
Customer with id = 23 visited the mall once and made one transaction during t
Customer with id = 9 visited the mall once and made one transaction during th
```

```
Customer with id = 30 visited the mall once and did not make any transactions
Customer with id = 54 visited the mall three times. During 2 visits they did
Customer with id = 96 visited the mall once and did not make any transactions
As we can see, users with IDs 30 and 96 visited the mall one time without mak
```

Solution

sql

```sql
#Solution 1:
SELECT a.customer_id, COUNT(a.visit_id) AS count_no_trans FROM Visits AS a
LEFT JOIN Transactions AS b
ON a.visit_id = b.visit_id
WHERE b.transaction_id IS NULL
GROUP BY a.customer_id;

#Solution 2:
SELECT customer_id, count(visit_id) AS count_no_trans
FROM Visits
WHERE visit_id NOT IN
    (SELECT visit_id
    FROM Transactions
    GROUP BY visit_id)
GROUP BY customer_id
```

1587. Bank Account Summary II | Easy | LeetCode

Table: **Users**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| account      | int     |
| name         | varchar |
+--------------+---------+
account is the primary key for this table.
Each row of this table contains the account number of each user in the bank.
```

Table: **Transactions**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| trans_id      | int     |
| account       | int     |
| amount        | int     |
| transacted_on | date    |
+---------------+---------+
trans_id is the primary key for this table.
Each row of this table contains all changes made to all accounts.
amount is positive if the user received money and negative if they transferr
All accounts start with a balance 0.
```

Write an SQL query to report the name and balance of users with a balance higher than 10000. The balance of an account is equal to the sum of the amounts of all transactions involving that account.

Return the result table in **any** order.

The query result format is in the following example.

```
Users table:
+------------+--------------+
| account    | name         |
+------------+--------------+
| 900001     | Alice        |
| 900002     | Bob          |
| 900003     | Charlie      |
+------------+--------------+

Transactions table:
+------------+------------+------------+---------------+
| trans_id   | account    | amount     | transacted_on |
+------------+------------+------------+---------------+
| 1          | 900001     | 7000       |  2020-08-01   |
| 2          | 900001     | 7000       |  2020-09-01   |
| 3          | 900001     | -3000      |  2020-09-02   |
| 4          | 900002     | 1000       |  2020-09-12   |
```

```
| 5           | 900003     | 6000        |  2020-08-07   |
| 6           | 900003     | 6000        |  2020-09-07   |
| 7           | 900003     | -4000       |  2020-09-11   |
+-------------+------------+-------------+---------------+

Result table:
+------------+------------+
| name       | balance    |
+------------+------------+
| Alice      | 11000      |
+------------+------------+
Alice's balance is (7000 + 7000 - 3000) = 11000.
Bob's balance is 1000.
Charlie's balance is (6000 + 6000 - 4000) = 8000.
```

Solution

sql

```sql
#Solution 1:
SELECT u.name AS NAME,SUM(t.amount) AS BALANCE
FROM Transactions t LEFT JOIN Users u
ON u.account = t.account
GROUP BY u.account
HAVING SUM(t.amount)>10000;


#Solution 2:
WITH tmp AS(
SELECT t.account, u.name, SUM(amount) AS balance
FROM Transactions t
LEFT JOIN Users u ON t.account = u.account
GROUP BY account )

SELECT name, balance
FROM tmp
WHERE balance > 10000
```

1596. The Most Frequently Ordered Products for Each Customer | Medium | LeetCode

Table: **Customers**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| name          | varchar |
+---------------+---------+
customer_id is the primary key for this table.
This table contains information about the customers.
```

Table: **Orders**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| order_date    | date    |
| customer_id   | int     |
| product_id    | int     |
+---------------+---------+
order_id is the primary key for this table.
This table contains information about the orders made by customer_id.
No customer will order the same product more than once in a single day.
```

Table: **Products**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| product_name  | varchar |
| price         | int     |
+---------------+---------+
product_id is the primary key for this table.
This table contains information about the products.
```

Write an SQL query to find the most frequently ordered product(s) for each customer.

The result table should have the `product_id` and `product_name` for each `customer_id` who ordered at least one order. Return the result table in **any order**.

The query result format is in the following example:

Customers

```
+-------------+-------+
| customer_id | name  |
+-------------+-------+
| 1           | Alice |
| 2           | Bob   |
| 3           | Tom   |
| 4           | Jerry |
| 5           | John  |
+-------------+-------+
```

Orders

```
+----------+------------+-------------+------------+
| order_id | order_date | customer_id | product_id |
+----------+------------+-------------+------------+
| 1        | 2020-07-31 | 1           | 1          |
| 2        | 2020-07-30 | 2           | 2          |
| 3        | 2020-08-29 | 3           | 3          |
| 4        | 2020-07-29 | 4           | 1          |
| 5        | 2020-06-10 | 1           | 2          |
| 6        | 2020-08-01 | 2           | 1          |
| 7        | 2020-08-01 | 3           | 3          |
| 8        | 2020-08-03 | 1           | 2          |
| 9        | 2020-08-07 | 2           | 3          |
| 10       | 2020-07-15 | 1           | 2          |
+----------+------------+-------------+------------+
```

Products

```
+------------+--------------+-------+
| product_id | product_name | price |
+------------+--------------+-------+
| 1          | keyboard     | 120   |
| 2          | mouse        | 80    |
| 3          | screen       | 600   |
| 4          | hard disk    | 450   |
+------------+--------------+-------+
```

```
Result table:
+-------------+------------+--------------+
| customer_id | product_id | product_name |
+-------------+------------+--------------+
| 1           | 2          | mouse        |
| 2           | 1          | keyboard     |
| 2           | 2          | mouse        |
| 2           | 3          | screen       |
| 3           | 3          | screen       |
| 4           | 1          | keyboard     |
+-------------+------------+--------------+

Alice (customer 1) ordered the mouse three times and the keyboard one time, s
Bob (customer 2) ordered the keyboard, the mouse, and the screen one time, sc
Tom (customer 3) only ordered the screen (two times), so that is the most fr
Jerry (customer 4) only ordered the keyboard (one time), so that is the most
John (customer 5) did not order anything, so we do not include them in the re
```

Solution

sql

```sql
#Solution 1:
SELECT customer_id, Products.product_id, Products.product_name FROM
(SELECT customer_id, product_id, order_count, RANK() OVER(PARTITION BY custor
(SELECT customer_id, product_id, COUNT(DISTINCT order_id) AS order_count FROM
GROUP BY customer_id, product_id) order_counts) order_counts_ranked
JOIN Products ON order_counts_ranked.product_id = Products.product_id
WHERE r = 1;

#solution- 2:

SELECT customer_id, T.product_id, product_name
FROM(
    SELECT customer_id, product_id,
    RANK() OVER( PARTITION BY customer_id ORDER BY COUNT(*) DESC ) AS RK
    FROM Orders o
    GROUP BY customer_id, product_id
) T
LEFT JOIN Products p on p.product_id = t.product_id
WHERE RK=1
```

```
#Solution-3:

WITH

tmp AS (
  SELECT a.customer_id, b.product_id, c.product_name,
  COUNT(b.order_id) OVER(PARTITION BY a.customer_id, b.product_id) AS freq
  FROM Customers AS a
  JOIN Orders AS b
  ON a.customer_id = b.customer_id
  JOIN Products AS c
  ON b.product_id = c.product_id
),

tmp1 AS (
  SELECT customer_id, product_id, product_name, freq,
  DENSE_RANK() OVER(PARTITION BY customer_id ORDER BY freq DESC) AS rnk
  FROM tmp
)

SELECT DISTINCT customer_id, product_id, product_name FROM tmp1
WHERE rnk = 1;
```

1607. Sellers With No Sales | Easy | LeetCode

Table: **Customer**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| customer_name | varchar |
+---------------+---------+
customer_id is the primary key for this table.
Each row of this table contains the information of each customer in the WebS
```

Table: **Orders**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| order_id      | int     |
| sale_date     | date    |
| order_cost    | int     |
| customer_id   | int     |
| seller_id     | int     |
+---------------+---------+
order_id is the primary key for this table.
Each row of this table contains all orders made in the webstore.
sale_date is the date when the transaction was made between the customer (cus
```

Table: `Seller`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| seller_id     | int     |
| seller_name   | varchar |
+---------------+---------+
seller_id is the primary key for this table.
Each row of this table contains the information of each seller.
```

Write an SQL query to report the names of all sellers who did not make any sales in 2020.

Return the result table ordered by `seller_name` in ascending order.

The query result format is in the following example.

```
Customer table:
+-------------+---------------+
| customer_id | customer_name |
+-------------+---------------+
| 101         | Alice         |
| 102         | Bob           |
| 103         | Charlie       |
+-------------+---------------+
```

Orders table:

order_id	sale_date	order_cost	customer_id	seller_id
1	2020-03-01	1500	101	1
2	2020-05-25	2400	102	2
3	2019-05-25	800	101	3
4	2020-09-13	1000	103	2
5	2019-02-11	700	101	2

Seller table:

seller_id	seller_name
1	Daniel
2	Elizabeth
3	Frank

Result table:

seller_name
Frank

Daniel made 1 sale in March 2020.
Elizabeth made 2 sales in 2020 and 1 sale in 2019.
Frank made 1 sale in 2019 but no sales in 2020.

Solution

sql

```sql
SELECT seller_name FROM Seller
WHERE seller_id NOT IN (
SELECT DISTINCT seller_id FROM Orders
WHERE YEAR(sale_date)='2020'
)
ORDER BY seller_name;
```

1613. Find the Missing IDs | Medium | LeetCode

Table: `Customers`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| customer_name | varchar |
+---------------+---------+
customer_id is the primary key for this table.
Each row of this table contains the name and the id customer.
```

Write an SQL query to find the missing customer IDs. The missing IDs are ones that are not in the `Customers` table but are in the range between `1` and the **maximum** `customer_id` present in the table.

Notice that the maximum `customer_id` will not exceed 100.

Return the result table ordered by **ids** in ***ascending order***.

The query result format is in the following example.

```
Customer table:
+-------------+---------------+
| customer_id | customer_name |
+-------------+---------------+
| 1           | Alice         |
| 4           | Bob           |
| 5           | Charlie       |
+-------------+---------------+

Result table:
+-----+
| ids |
+-----+
| 2   |
| 3   |
+-----+
```

```
The maximum customer_id present in the table is 5, so in the range [1,5], IDs
```

Solution

sql

```sql
WITH RECURSIVE CTE AS(
    SELECT 1 AS 'id', MAX(c.customer_id) AS 'Max_Id'
    FROM Customers c
    UNION ALL
    SELECT id+1, Max_Id
    FROM CTE
    WHERE id < Max_id
)

SELECT id AS 'ids'
FROM CTE c
WHERE c.id NOT IN (SELECT customer_id FROM Customers)
ORDER BY 1 ASC
```

1623. All Valid Triplets That Can Represent a Country | Easy | LeetCode

Table: **SchoolA**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| student_id    | int     |
| student_name  | varchar |
+---------------+---------+
student_id is the primary key for this table.
Each row of this table contains the name and the id of a student in school A.
All student_name are distinct.
```

ble: **SchoolB**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| student_id    | int     |
| student_name  | varchar |
+---------------+---------+
student_id is the primary key for this table.
Each row of this table contains the name and the id of a student in school B.
All student_name are distinct.
```

Table: `SchoolC`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| student_id    | int     |
| student_name  | varchar |
+---------------+---------+
student_id is the primary key for this table.
Each row of this table contains the name and the id of a student in school C.
All student_name are distinct.
```

There is a country with three schools, where each student is enrolled in **exactly one** school. The country is joining a competition and wants to select one student from each school to represent the country such that:

- `member_A` is selected from `SchoolA`,
- `member_B` is selected from `SchoolB`,
- `member_C` is selected from `SchoolC`, and The selected students' names and IDs are pairwise distinct (i.e. no two students share the same name, and no two students share the same ID). Write an SQL query to find all the possible triplets representing the country under the given constraints.

Return the result table in **any order**.

ɜ query result format is in the following example.

SchoolA table:

```
+------------+--------------+
| student_id | student_name |
+------------+--------------+
| 1          | Alice        |
| 2          | Bob          |
+------------+--------------+
```

SchoolB table:

```
+------------+--------------+
| student_id | student_name |
+------------+--------------+
| 3          | Tom          |
+------------+--------------+
```

SchoolC table:

```
+------------+--------------+
| student_id | student_name |
+------------+--------------+
| 3          | Tom          |
| 2          | Jerry        |
| 10         | Alice        |
+------------+--------------+
```

Result table:

```
+----------+----------+----------+
| member_A | member_B | member_C |
+----------+----------+----------+
| Alice    | Tom      | Jerry    |
| Bob      | Tom      | Alice    |
+----------+----------+----------+
```

Let us see all the possible triplets.
- (Alice, Tom, Tom) --> Rejected because member_B and member_C have the same
- (Alice, Tom, Jerry) --> Valid triplet.
- (Alice, Tom, Alice) --> Rejected because member_A and member_C have the sar
- (Bob, Tom, Tom) --> Rejected because member_B and member_C have the same na
- (Bob, Tom, Jerry) --> Rejected because member_A and member_C have the same
- (Bob, Tom, Alice) --> Valid triplet.

Solution

```sql
SELECT a.student_name AS 'member_A',
b.student_name AS 'member_B',
c.student_name AS 'member_C'
FROM SchoolA AS a
JOIN SchoolB AS b
ON a.student_id <> b.student_id
AND a.student_name <> b.student_name
JOIN SchoolC AS c
ON a.student_id <> c.student_id
AND b.student_id <> c.student_id
AND a.student_name <> c.student_name
AND b.student_name <> c.student_name;
```

1633. Percentage of Users Attended a Contest | Easy | LeetCode

Table: **Users**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| user_id     | int     |
| user_name   | varchar |
+-------------+---------+
user_id is the primary key for this table.
Each row of this table contains the name and the id of a user.
```

Table: **Register**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| contest_id  | int     |
| user_id     | int     |
+-------------+---------+
(contest_id, user_id) is the primary key for this table.
```

Each row of this table contains the id of a user and the contest they registe

Write an SQL query to find the percentage of the users registered in each contest rounded to two decimals.

Return the result table ordered by percentage in descending order. In case of a tie, order it by contest_id in ascending order.

The query result format is in the following example.

```
Users table:
+---------+-----------+
| user_id | user_name |
+---------+-----------+
| 6       | Alice     |
| 2       | Bob       |
| 7       | Alex      |
+---------+-----------+

Register table:
+------------+---------+
| contest_id | user_id |
+------------+---------+
| 215        | 6       |
| 209        | 2       |
| 208        | 2       |
| 210        | 6       |
| 208        | 6       |
| 209        | 7       |
| 209        | 6       |
| 215        | 7       |
| 208        | 7       |
| 210        | 2       |
| 207        | 2       |
| 210        | 7       |
+------------+---------+

Result table:
+------------+------------+
| contest_id | percentage |
+------------+------------+
```

```
| 208         | 100.0      |
| 209         | 100.0      |
| 210         | 100.0      |
| 215         | 66.67      |
| 207         | 33.33      |
+-------------+------------+
All the users registered in contests 208, 209, and 210. The percentage is 10(
Alice and Alex registered in contest 215 and the percentage is ((2/3) * 100)
Bob registered in contest 207 and the percentage is ((1/3) * 100) = 33.33%
```

Solution

sql

```sql
SELECT contest_id, ROUND(COUNT(user_id)*100.00/(SELECT COUNT(*) FROM users),2
FROM register
GROUP BY contest_id
ORDER BY percentage desc, contest_id
```

1635. Hopper Company Queries I | Hard | LeetCode

Table: **Drivers**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| driver_id   | int     |
| join_date   | date    |
+-------------+---------+
driver_id is the primary key for this table.
Each row of this table contains the driver's ID and the date they joined the
```

Table: **Rides**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| ride_id      | int     |
```

```
| user_id      | int     |
| requested_at | date    |
+--------------+---------+
ride_id is the primary key for this table.
Each row of this table contains the ID of a ride, the user's ID that requeste
There may be some ride requests in this table that were not accepted.
```

Table: **AcceptedRides**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| ride_id       | int     |
| driver_id     | int     |
| ride_distance | int     |
| ride_duration | int     |
+---------------+---------+
ride_id is the primary key for this table.
Each row of this table contains some information about an accepted ride.
It is guaranteed that each accepted ride exists in the Rides table.
```

Write an SQL query to report the following statistics for each month of **2020**:

The number of drivers currently with the Hopper company by the end of the month (**active_drivers**). The number of accepted rides in that month (**accepted_rides**). Return the result table ordered by **month** in ascending order, where **month** is the month's number (January is **1**, February is **2**, etc.).

The query result format is in the following example.

```
Drivers table:
+-----------+------------+
| driver_id | join_date  |
+-----------+------------+
| 10        | 2019-12-10 |
| 8         | 2020-1-13  |
| 5         | 2020-2-16  |
| 7         | 2020-3-8   |
| 4         | 2020-5-17  |
```

| 1 | 2020-10-24 |
| 6 | 2021-1-5 |

Rides table:

ride_id	user_id	requested_at
6	75	2019-12-9
1	54	2020-2-9
10	63	2020-3-4
19	39	2020-4-6
3	41	2020-6-3
13	52	2020-6-22
7	69	2020-7-16
17	70	2020-8-25
20	81	2020-11-2
5	57	2020-11-9
2	42	2020-12-9
11	68	2021-1-11
15	32	2021-1-17
12	11	2021-1-19
14	18	2021-1-27

AcceptedRides table:

ride_id	driver_id	ride_distance	ride_duration
10	10	63	38
13	10	73	96
7	8	100	28
17	7	119	68
20	1	121	92
5	7	42	101
2	4	6	38
11	8	37	43
15	8	108	82
12	8	38	34
14	1	90	74

Result table:

month	active_drivers	accepted_rides
1	2	0
2	3	0
3	4	1
4	4	0
5	5	0
6	5	1
7	5	1
8	5	1
9	5	0
10	6	0
11	6	2
12	6	1

By the end of January --> two active drivers (10, 8) and no accepted rides.
By the end of February --> three active drivers (10, 8, 5) and no accepted r
By the end of March --> four active drivers (10, 8, 5, 7) and one accepted r
By the end of April --> four active drivers (10, 8, 5, 7) and no accepted ri
By the end of May --> five active drivers (10, 8, 5, 7, 4) and no accepted r
By the end of June --> five active drivers (10, 8, 5, 7, 4) and one accepted
By the end of July --> five active drivers (10, 8, 5, 7, 4) and one accepted
By the end of August --> five active drivers (10, 8, 5, 7, 4) and one accepte
By the end of Septemeber --> five active drivers (10, 8, 5, 7, 4) and no acce
By the end of October --> six active drivers (10, 8, 5, 7, 4, 1) and no accep
By the end of November --> six active drivers (10, 8, 5, 7, 4, 1) and two ac
By the end of December --> six active drivers (10, 8, 5, 7, 4, 1) and one ac

Solution

sql

```sql
SELECT t.month,
    COUNT(DISTINCT driver_id) active_drivers,
    COUNT(DISTINCT rides.ride_id) accepted_rides
FROM
    ((SELECT 1 AS month)
    UNION (SELECT 2 AS month)
    UNION (SELECT 3 AS month)
```

```
    UNION (SELECT 4 AS month)
    UNION (SELECT 5 AS month)
    UNION (SELECT 6 AS month)
    UNION (SELECT 7 AS month)
    UNION (SELECT 8 AS month)
    UNION (SELECT 9 AS month)
    UNION (SELECT 10 AS month)
    UNION (SELECT 11 AS month)
    UNION (SELECT 12 AS month)) t
LEFT JOIN
    (SELECT driver_id,
    (CASE WHEN year(join_date) = 2019 THEN '1' ELSE month(join_date) END) `mo
    FROM Drivers
    WHERE year(join_date) <= 2020) d
ON d.month <= t.month
LEFT JOIN
    (SELECT month(requested_at) AS `month`, a.ride_id
    FROM AcceptedRides a
    JOIN Rides r
    ON r.ride_id = a.ride_id
    WHERE year(requested_at) = 2020) rides
ON t.month = rides.month
GROUP BY t.month
ORDER BY t.month
```

1645. Hopper Company Queries II | Hard | LeetCode

Table: **Drivers**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| driver_id   | int     |
| join_date   | date    |
+-------------+---------+
driver_id is the primary key for this table.
Each row of this table contains the driver's ID and the date they joined the
```

Table: **Rides**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| ride_id      | int     |
| user_id      | int     |
| requested_at | date    |
+--------------+---------+
ride_id is the primary key for this table.
Each row of this table contains the ID of a ride, the user's ID that requeste
There may be some ride requests in this table that were not accepted.
```

Table: `AcceptedRides`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| ride_id       | int     |
| driver_id     | int     |
| ride_distance | int     |
| ride_duration | int     |
+---------------+---------+
ride_id is the primary key for this table.
Each row of this table contains some information about an accepted ride.
It is guaranteed that each accepted ride exists in the Rides table.
```

Write an SQL query to report the percentage of working drivers (`working_percentage`) for each month of **2020** where:

$$percentage_{month} = \frac{\#\ drivers\ that\ accepted\ at\ least\ one\ ride\ during\ the\ month}{\#\ available\ drivers\ during\ the\ month} \times 100.0$$

Note that if the number of available drivers during a month is zero, we consider the `working_percentage` to be `0` .

Return the result table ordered by `month` in **ascending** order, where **month** is the month's number (January is **1**, February is **2**, etc.). Round **working_percentage** to the nearest **2 decimal places**.

The query result format is in the following example.

Drivers table:

driver_id	join_date
10	2019-12-10
8	2020-1-13
5	2020-2-16
7	2020-3-8
4	2020-5-17
1	2020-10-24
6	2021-1-5

Rides table:

ride_id	user_id	requested_at
6	75	2019-12-9
1	54	2020-2-9
10	63	2020-3-4
19	39	2020-4-6
3	41	2020-6-3
13	52	2020-6-22
7	69	2020-7-16
17	70	2020-8-25
20	81	2020-11-2
5	57	2020-11-9
2	42	2020-12-9
11	68	2021-1-11
15	32	2021-1-17
12	11	2021-1-19
14	18	2021-1-27

AcceptedRides table:

ride_id	driver_id	ride_distance	ride_duration
10	10	63	38
13	10	73	96
7	8	100	28
17	7	119	68

```
| 20      | 1         | 121           | 92           |
| 5       | 7         | 42            | 101          |
| 2       | 4         | 6             | 38           |
| 11      | 8         | 37            | 43           |
| 15      | 8         | 108           | 82           |
| 12      | 8         | 38            | 34           |
| 14      | 1         | 90            | 74           |
+---------+-----------+---------------+--------------+
```

Result table:

```
+-------+--------------------+
| month | working_percentage |
+-------+--------------------+
| 1     | 0.00               |
| 2     | 0.00               |
| 3     | 25.00              |
| 4     | 0.00               |
| 5     | 0.00               |
| 6     | 20.00              |
| 7     | 20.00              |
| 8     | 20.00              |
| 9     | 0.00               |
| 10    | 0.00               |
| 11    | 33.33              |
| 12    | 16.67              |
+-------+--------------------+
```

```
By the end of January --> two active drivers (10, 8) and no accepted rides. T
By the end of February --> three active drivers (10, 8, 5) and no accepted ri
By the end of March --> four active drivers (10, 8, 5, 7) and one accepted ri
By the end of April --> four active drivers (10, 8, 5, 7) and no accepted rid
By the end of May --> five active drivers (10, 8, 5, 7, 4) and no accepted ri
By the end of June --> five active drivers (10, 8, 5, 7, 4) and one accepted
By the end of July --> five active drivers (10, 8, 5, 7, 4) and one accepted
By the end of August --> five active drivers (10, 8, 5, 7, 4) and one accepte
By the end of Septemeber --> five active drivers (10, 8, 5, 7, 4) and no acce
By the end of October --> six active drivers (10, 8, 5, 7, 4, 1) and no accep
By the end of November --> six active drivers (10, 8, 5, 7, 4, 1) and two acc
By the end of December --> six active drivers (10, 8, 5, 7, 4, 1) and one acc
```

Solution

```sql
SELECT months_drivers.month, ROUND(COALESCE(100 * COALESCE(total_active_drive
FROM
(
    SELECT month, COUNT(driver_id) AS total_drivers
    FROM Drivers AS a
    RIGHT JOIN
    (
        SELECT "2020-1-31" AS day, 1 AS month
        UNION SELECT "2020-2-29", 2
        UNION SELECT "2020-3-31", 3
        UNION SELECT "2020-4-30", 4
        UNION SELECT "2020-5-31", 5
        UNION SELECT "2020-6-30", 6
        UNION SELECT "2020-7-31", 7
        UNION SELECT "2020-8-31", 8
        UNION SELECT "2020-9-30", 9
        UNION SELECT "2020-10-31", 10
        UNION SELECT "2020-11-30", 11
        UNION SELECT "2020-12-31", 12
    ) AS months
    ON join_date <= day
    GROUP BY month
) months_drivers
LEFT JOIN
(
    SELECT month, COUNT(DISTINCT b.driver_id) AS total_active_drivers
    FROM
    (
        SELECT ride_id, CAST(substr(requested_at, 6, 2) AS unsigned) AS month
        FROM Rides
        WHERE substr(requested_at, 1, 4) = "2020"
    ) month_rides
    JOIN AcceptedRides AS b
    ON month_rides.ride_id = b.ride_id
    GROUP BY month
) months_active_drivers
ON months_drivers.month = months_active_drivers.month;
```

1651. Hopper Company Queries III | Hard | LeetCode

Table: **Drivers**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| driver_id   | int     |
| join_date   | date    |
+-------------+---------+
driver_id is the primary key for this table.
Each row of this table contains the driver's ID and the date they joined the
```

Table: **Rides**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| ride_id      | int     |
| user_id      | int     |
| requested_at | date    |
+--------------+---------+
ride_id is the primary key for this table.
Each row of this table contains the ID of a ride, the user's ID that requeste
There may be some ride requests in this table that were not accepted.
```

Table: **AcceptedRides**

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| ride_id       | int     |
| driver_id     | int     |
| ride_distance | int     |
| ride_duration | int     |
+---------------+---------+
ride_id is the primary key for this table.
Each row of this table contains some information about an accepted ride.
It is guaranteed that each accepted ride exists in the Rides table.
```

Write an SQL query to compute the `average_ride_distance` and `average_ride_duration` of every 3-month window starting from **January - March 2020 to October - December 2020**. Round `average_ride_distance` and `average_ride_duration` to the nearest two decimal places.

The `average_ride_distance` is calculated by summing up the total `ride_distance` values from the three months and dividing it by `3`. The `average_ride_duration` is calculated in a similar way.

Return the result table ordered by `month` in ascending order, where `month` is the starting month's number (January is `1`, February is `2`, etc.).

The query result format is in the following example.

```
Drivers table:
+-----------+------------+
| driver_id | join_date  |
+-----------+------------+
| 10        | 2019-12-10 |
| 8         | 2020-1-13  |
| 5         | 2020-2-16  |
| 7         | 2020-3-8   |
| 4         | 2020-5-17  |
| 1         | 2020-10-24 |
| 6         | 2021-1-5   |
+-----------+------------+

Rides table:
+---------+---------+--------------+
| ride_id | user_id | requested_at |
+---------+---------+--------------+
| 6       | 75      | 2019-12-9    |
| 1       | 54      | 2020-2-9     |
| 10      | 63      | 2020-3-4     |
| 19      | 39      | 2020-4-6     |
| 3       | 41      | 2020-6-3     |
| 13      | 52      | 2020-6-22    |
| 7       | 69      | 2020-7-16    |
| 17      | 70      | 2020-8-25    |
| 20      | 81      | 2020-11-2    |
| 5       | 57      | 2020-11-9    |
```

2	42	2020-12-9
11	68	2021-1-11
15	32	2021-1-17
12	11	2021-1-19
14	18	2021-1-27

AcceptedRides table:

ride_id	driver_id	ride_distance	ride_duration
10	10	63	38
13	10	73	96
7	8	100	28
17	7	119	68
20	1	121	92
5	7	42	101
2	4	6	38
11	8	37	43
15	8	108	82
12	8	38	34
14	1	90	74

Result table:

month	average_ride_distance	average_ride_duration
1	21.00	12.67
2	21.00	12.67
3	21.00	12.67
4	24.33	32.00
5	57.67	41.33
6	97.33	64.00
7	73.00	32.00
8	39.67	22.67
9	54.33	64.33
10	56.33	77.00

By the end of January --> average_ride_distance = (0+0+63)/3=21, average_ride
By the end of February --> average_ride_distance = (0+63+0)/3=21, average_ri

```
By the end of March --> average_ride_distance = (63+0+0)/3=21, average_ride_d
By the end of April --> average_ride_distance = (0+0+73)/3=24.33, average_rid
By the end of May --> average_ride_distance = (0+73+100)/3=57.67, average_rid
By the end of June --> average_ride_distance = (73+100+119)/3=97.33, average_
By the end of July --> average_ride_distance = (100+119+0)/3=73.00, average_r
By the end of August --> average_ride_distance = (119+0+0)/3=39.67, average_r
By the end of Septemeber --> average_ride_distance = (0+0+163)/3=54.33, avera
By the end of October --> average_ride_distance = (0+163+6)/3=56.33, average_
```

Solution

sql

```sql
SELECT month,
    COALESCE(ROUND(SUM(ride_distance)/3,2),0) AS average_ride_distance,
    COALESCE(ROUND(SUM(ride_duration)/3,2),0) AS average_ride_duration
FROM
(
    SELECT months.month, ride_id
    FROM Rides
    RIGHT JOIN
    (
        SELECT "2020-1-1" AS start, "2020-3-31" AS last, 1 AS month
        UNION SELECT "2020-2-1", "2020-4-30", 2
        UNION SELECT "2020-3-1", "2020-5-31", 3
        UNION SELECT "2020-4-1", "2020-6-30", 4
        UNION SELECT "2020-5-1", "2020-7-31", 5
        UNION SELECT "2020-6-1", "2020-8-31", 6
        UNION SELECT "2020-7-1", "2020-9-30", 7
        UNION SELECT "2020-8-1", "2020-10-31", 8
        UNION SELECT "2020-9-1", "2020-11-30", 9
        UNION SELECT "2020-10-1", "2020-12-31", 10
    ) AS months
    ON months.start <= requested_at AND months.last >= requested_at
) total
LEFT JOIN AcceptedRides AS a
ON total.ride_id=a.ride_id
GROUP BY month
ORDER BY month;
```

1661. Average Time of Process per Machine | Easy | LeetCode

Table: `Activity`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| machine_id     | int     |
| process_id     | int     |
| activity_type  | enum    |
| timestamp      | float   |
+----------------+---------+
The table shows the user activities for a factory website.
(machine_id, process_id, activity_type) is the primary key of this table.
machine_id is the ID of a machine.
process_id is the ID of a process running on the machine with ID machine_id.
activity_type is an ENUM of type ('start', 'end').
timestamp is a float representing the current time in seconds.
'start' means the machine starts the process at the given timestamp and 'end
The 'start' timestamp will always be before the 'end' timestamp for every (ma
```

There is a factory website that has several machines each running the **same number of processes**. Write an SQL query to find the **average time** each machine takes to complete a process.

The time to complete a process is the `'end' timestamp` minus the `'start' timestamp`. The average time is calculated by the total time to complete every process on the machine divided by the number of processes that were run.

The resulting table should have the `machine_id` along with the **average time** as `processing_time`, which should be **rounded to 3 decimal places**.

The query result format is in the following example:

```
Activity table:
+------------+------------+---------------+-----------+
| machine_id | process_id | activity_type | timestamp |
+------------+------------+---------------+-----------+
```

```
| 0          | 0          | start         | 0.712     |
| 0          | 0          | end           | 1.520     |
| 0          | 1          | start         | 3.140     |
| 0          | 1          | end           | 4.120     |
| 1          | 0          | start         | 0.550     |
| 1          | 0          | end           | 1.550     |
| 1          | 1          | start         | 0.430     |
| 1          | 1          | end           | 1.420     |
| 2          | 0          | start         | 4.100     |
| 2          | 0          | end           | 4.512     |
| 2          | 1          | start         | 2.500     |
| 2          | 1          | end           | 5.000     |
+------------+------------+---------------+-----------+

Result table:
+------------+-----------------+
| machine_id | processing_time |
+------------+-----------------+
| 0          | 0.894           |
| 1          | 0.995           |
| 2          | 1.456           |
+------------+-----------------+

There are 3 machines running 2 processes each.
Machine 0's average time is ((1.520 - 0.712) + (4.120 - 3.140)) / 2 = 0.894
Machine 1's average time is ((1.550 - 0.550) + (1.420 - 0.430)) / 2 = 0.995
Machine 2's average time is ((4.512 - 4.100) + (5.000 - 2.500)) / 2 = 1.456
```

Solution

sql

```sql
SELECT machine_id,
    ROUND(SUM(IF(activity_type='start', -timestamp, timestamp)) / COUNT(DIST
FROM Activity
GROUP BY machine_id
ORDER BY machine_id
```

1667. Fix Names in a Table | Easy | LeetCode

Table: `Users`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| user_id        | int     |
| name           | varchar |
+----------------+---------+
user_id is the primary key for this table.
This table contains the ID and the name of the user. The name consists of on
```

Write an SQL query to fix the names so that only the first character is uppercase and the rest are lowercase.

Return the result table ordered by `user_id`.

The query result format is in the following example:

```
Users table:
+---------+-------+
| user_id | name  |
+---------+-------+
| 1       | aLice |
| 2       | bOB   |
+---------+-------+

Result table:
+---------+-------+
| user_id | name  |
+---------+-------+
| 1       | Alice |
| 2       | Bob   |
+---------+-------+
```

Solution

sql

```
select user_id,
```

```
    CONCAT(UPPER(LEFT(name,1)),LOWER(SUBSTRING(name,2))) AS name
FROM Users
ORDER BY user_id
```

1677. Product's Worth Over Invoices | Easy | LeetCode

Table: **Product**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_id  | int     |
| name        | varchar |
+-------------+---------+
product_id is the primary key for this table.
This table contains the ID and the name of the product. The name consists of
```

Table: **Invoice**

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| invoice_id  | int  |
| product_id  | int  |
| rest        | int  |
| paid        | int  |
| canceled    | int  |
| refunded    | int  |
+-------------+------+
invoice_id is the primary key for this table and the id of this invoice.
product_id is the id of the product for this invoice.
rest is the amount left to pay for this invoice.
paid is the amount paid for this invoice.
canceled is the amount canceled for this invoice.
refunded is the amount refunded for this invoice.
```

Write an SQL query that will, for all products, return each product name with total amount due, paid, canceled, and refunded across all invoices.

Return the result table ordered by `product_name`.

The query result format is in the following example:

```
Product table:
+------------+-------+
| product_id | name  |
+------------+-------+
| 0          | ham   |
| 1          | bacon |
+------------+-------+
Invoice table:
+------------+------------+------+------+----------+----------+
| invoice_id | product_id | rest | paid | canceled | refunded |
+------------+------------+------+------+----------+----------+
| 23         | 0          | 2    | 0    | 5        | 0        |
| 12         | 0          | 0    | 4    | 0        | 3        |
| 1          | 1          | 1    | 1    | 0        | 1        |
| 2          | 1          | 1    | 0    | 1        | 1        |
| 3          | 1          | 0    | 1    | 1        | 1        |
| 4          | 1          | 1    | 1    | 1        | 0        |
+------------+------------+------+------+----------+----------+
Result table:
+-------+------+------+----------+----------+
| name  | rest | paid | canceled | refunded |
+-------+------+------+----------+----------+
| bacon | 3    | 3    | 3        | 3        |
| ham   | 2    | 4    | 5        | 3        |
+-------+------+------+----------+----------+
- The amount of money left to pay for bacon is 1 + 1 + 0 + 1 = 3
- The amount of money paid for bacon is 1 + 0 + 1 + 1 = 3
- The amount of money canceled for bacon is 0 + 1 + 1 + 1 = 3
- The amount of money refunded for bacon is 1 + 1 + 1 + 0 = 3
- The amount of money left to pay for ham is 2 + 0 = 2
- The amount of money paid for ham is 0 + 4 = 4
- The amount of money canceled for ham is 5 + 0 = 5
- The amount of money refunded for ham is 0 + 3 = 3
```

Solution

```sql
SELECT p.name AS name,
    SUM(i.rest) AS rest,
    SUM(i.paid) AS paid,
    SUM(i.canceled) AS canceled,
    SUM(i.refunded) AS refunded
FROM Invoice i
LEFT JOIN Product p ON p.product_id = i.product_id
GROUP BY name
ORDER BY name;
```

1683. Invalid Tweets | Easy | LeetCode

Table: `Tweets`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| tweet_id       | int     |
| content        | varchar |
+----------------+---------+
tweet_id is the primary key for this table.
This table contains all the tweets in a social media app.
```

Write an SQL query to find the IDs of the invalid tweets. The tweet is invalid if the number of characters used in the content of the tweet is **strictly greater** than **15**.

Return the result table **in any order**.

The query result format is in the following example:

```
Tweets table:
+----------+----------------------------------+
| tweet_id | content                          |
+----------+----------------------------------+
```

```
| 1        | Vote for Biden                  |
| 2        | Let us make America great again! |
+----------+---------------------------------+
```

```
Result table:
+----------+
| tweet_id |
+----------+
| 2        |
+----------+
Tweet 1 has length = 14. It is a valid tweet.
Tweet 2 has length = 32. It is an invalid tweet.
```

Solution

sql

```sql
SELECT tweet_id
FROM Tweets
WHERE LENGTH(content) > 15;
```

1693. Daily Leads and Partners | Easy | LeetCode

Table: **DailySales**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| date_id     | date    |
| make_name   | varchar |
| lead_id     | int     |
| partner_id  | int     |
+-------------+---------+
This table does not have a primary key.
This table contains the date and the name of the product sold and the IDs of
The name consists of only lowercase English letters.
```

Write an SQL query that will, for each `date_id` and `make_name`, return the number of distinct `lead_id`'s and distinct `partner_id`'s.

Return the result table in any order.

The query result format is in the following example:

```
DailySales table:
+-----------+-----------+---------+------------+
| date_id   | make_name | lead_id | partner_id |
+-----------+-----------+---------+------------+
| 2020-12-8 | toyota    | 0       | 1          |
| 2020-12-8 | toyota    | 1       | 0          |
| 2020-12-8 | toyota    | 1       | 2          |
| 2020-12-7 | toyota    | 0       | 2          |
| 2020-12-7 | toyota    | 0       | 1          |
| 2020-12-8 | honda     | 1       | 2          |
| 2020-12-8 | honda     | 2       | 1          |
| 2020-12-7 | honda     | 0       | 1          |
| 2020-12-7 | honda     | 1       | 2          |
| 2020-12-7 | honda     | 2       | 1          |
+-----------+-----------+---------+------------+
Result table:
+-----------+-----------+--------------+-----------------+
| date_id   | make_name | unique_leads | unique_partners |
+-----------+-----------+--------------+-----------------+
| 2020-12-8 | toyota    | 2            | 3               |
| 2020-12-7 | toyota    | 1            | 2               |
| 2020-12-8 | honda     | 2            | 2               |
| 2020-12-7 | honda     | 3            | 2               |
+-----------+-----------+--------------+-----------------+
For 2020-12-8, toyota gets leads = [0, 1] and partners = [0, 1, 2] while hon
For 2020-12-7, toyota gets leads = [0] and partners = [1, 2] while honda get
```

Solution

sql

```sql
SELECT date_id, make_name,
    COUNT(DISTINCT lead_id) AS unique_leads,
    COUNT(DISTINCT partner_id) AS unique_partners
```

```
FROM DailySales
GROUP BY date_id, make_name
```

1699. Number of Calls Between Two Persons | Medium | LeetCode

Table: `Calls`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| from_id     | int     |
| to_id       | int     |
| duration    | int     |
+-------------+---------+
This table does not have a primary key, it may contain duplicates.
This table contains the duration of a phone call between from_id and to_id.
from_id != to_id
```

Write an SQL query to report the number of calls and the total call duration between each pair of distinct persons `(person1, person2)` where `person1 < person2`.

Return the result table in any order.

The query result format is in the following example:

```
Calls table:
+---------+-------+----------+
| from_id | to_id | duration |
+---------+-------+----------+
| 1       | 2     | 59       |
| 2       | 1     | 11       |
| 1       | 3     | 20       |
| 3       | 4     | 100      |
| 3       | 4     | 200      |
| 3       | 4     | 200      |
| 4       | 3     | 499      |
+---------+-------+----------+
```

Result table:

```
+---------+---------+------------+----------------+
| person1 | person2 | call_count | total_duration |
+---------+---------+------------+----------------+
| 1       | 2       | 2          | 70             |
| 1       | 3       | 1          | 20             |
| 3       | 4       | 4          | 999            |
+---------+---------+------------+----------------+
Users 1 and 2 had 2 calls and the total duration is 70 (59 + 11).
Users 1 and 3 had 1 call and the total duration is 20.
Users 3 and 4 had 4 calls and the total duration is 999 (100 + 200 + 200 + 49
```

Solution

sql

```sql
#Solution 1:
SELECT from_id AS person1,to_id AS person2,
    COUNT(duration) AS call_count, SUM(duration) AS total_duration
FROM (SELECT *
      FROM Calls

      UNION ALL

      SELECT to_id, from_id, duration
      FROM Calls) t1
WHERE from_id < to_id
GROUP BY person1, person2

#Solution 2:
SELECT
    IF(from_id<to_id,from_id,to_id) person1,
    IF(from_id>to_id,from_id,to_id) person2,
    COUNT(*) call_count,
    SUM(duration) total_duration
FROM
    Calls
GROUP BY
    IF(from_id<to_id,from_id,to_id),
    IF(from_id>to_id,from_id,to_id);
```

1709. Biggest Window Between Visits | Medium | [redacted]eetCode

Table: `UserVisits`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| user_id     | int  |
| visit_date  | date |
+-------------+------+
This table does not have a primary key.
This table contains logs of the dates that users vistied a certain retailer.
```

Assume today's date is `'2021-1-1'`.

Write an SQL query that will, for each `user_id`, find out the largest `window` of days between each visit and the one right after it (or today if you are considering the last visit).

Return the result table ordered by `user_id`.

The query result format is in the following example:

```
UserVisits table:
+---------+------------+
| user_id | visit_date |
+---------+------------+
| 1       | 2020-11-28 |
| 1       | 2020-10-20 |
| 1       | 2020-12-3  |
| 2       | 2020-10-5  |
| 2       | 2020-12-9  |
| 3       | 2020-11-11 |
+---------+------------+
Result table:
+---------+---------------+
| user_id | biggest_window|
+---------+---------------+
```

```
| 1       | 39           |
| 2       | 65           |
| 3       | 51           |
+---------+--------------+
For the first user, the windows in question are between dates:
    - 2020-10-20 and 2020-11-28 with a total of 39 days.
    - 2020-11-28 and 2020-12-3 with a total of 5 days.
    - 2020-12-3 and 2021-1-1 with a total of 29 days.
Making the biggest window the one with 39 days.
For the second user, the windows in question are between dates:
    - 2020-10-5 and 2020-12-9 with a total of 65 days.
    - 2020-12-9 and 2021-1-1 with a total of 23 days.
Making the biggest window the one with 65 days.
For the third user, the only window in question is between dates 2020-11-11 a
```

Solution

sql

```sql
SELECT user_id, max(diff) AS biggest_window
FROM
(
    SELECT user_id,
       datediff(coalesce(lead(visit_date) OVER (PARTITION BY user_id ORDER BY
    FROM userVisits
) t
GROUP BY user_id
ORDER BY user_id
```

1715. Count Apples and Oranges | Medium | LeetCode

Table: **Boxes**

```
+--------------+------+
| Column Name  | Type |
+--------------+------+
| box_id       | int  |
| chest_id     | int  |
| apple_count  | int  |
```

```
| orange_count | int  |
+--------------+------+
box_id is the primary key for this table.
chest_id is a foreign key of the chests table.
This table contains information about the boxes and the number of oranges and
```

Table: **Chests**

```
+--------------+------+
| Column Name  | Type |
+--------------+------+
| chest_id     | int  |
| apple_count  | int  |
| orange_count | int  |
+--------------+------+
chest_id is the primary key for this table.
This table contains information about the chests we have, and the correspond
```

Write an SQL query to count the number of apples and oranges in all the boxes. If a box contains a chest, you should also include the number of apples and oranges it has.

Return the result table in **any order**.

The query result format is in the following example:

```
Boxes table:
+--------+----------+-------------+--------------+
| box_id | chest_id | apple_count | orange_count |
+--------+----------+-------------+--------------+
| 2      | null     | 6           | 15           |
| 18     | 14       | 4           | 15           |
| 19     | 3        | 8           | 4            |
| 12     | 2        | 19          | 20           |
| 20     | 6        | 12          | 9            |
| 8      | 6        | 9           | 9            |
| 3      | 14       | 16          | 7            |
+--------+----------+-------------+--------------+

Chests table:
```

```
+----------+-------------+--------------+
| chest_id | apple_count | orange_count |
+----------+-------------+--------------+
| 6        | 5           | 6            |
| 14       | 20          | 10           |
| 2        | 8           | 8            |
| 3        | 19          | 4            |
| 16       | 19          | 19           |
+----------+-------------+--------------+

Result table:
+-------------+--------------+
| apple_count | orange_count |
+-------------+--------------+
| 151         | 123          |
+-------------+--------------+
box 2 has 6 apples and 15 oranges.
box 18 has 4 + 20 (from the chest) = 24 apples and 15 + 10 (from the chest) =
box 19 has 8 + 19 (from the chest) = 27 apples and 4 + 4 (from the chest) = 8
box 12 has 19 + 8 (from the chest) = 27 apples and 20 + 8 (from the chest) =
box 20 has 12 + 5 (from the chest) = 17 apples and 9 + 6 (from the chest) = 1
box 8 has 9 + 5 (from the chest) = 14 apples and 9 + 6 (from the chest) = 15
box 3 has 16 + 20 (from the chest) = 36 apples and 7 + 10 (from the chest) =
Total number of apples = 6 + 24 + 27 + 27 + 17 + 14 + 36 = 151
Total number of oranges = 15 + 25 + 8 + 28 + 15 + 15 + 17 = 123
```

Solution

sql

```sql
SELECT sum(IFNULL(box.apple_count, 0) + IFNULL(chest.apple_count, 0)) AS app
    sum(IFNULL(box.orange_count, 0) + IFNULL(chest.orange_count, 0)) AS orang
    FROM Boxes AS box
    LEFT JOIN Chests AS chest
    ON box.chest_id = chest.chest_id;
```

1729. Find Followers Count | Easy | LeetCode

Table: `Followers`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| user_id     | int  |
| follower_id | int  |
+-------------+------+
(user_id, follower_id) is the primary key for this table.
This table contains the IDs of a user and a follower in a social media app wh
```

Write an SQL query that will, for each user, return the number of followers.

Return the result table ordered by user_id.

The query result format is in the following example:

```
Followers table:
+---------+-------------+
| user_id | follower_id |
+---------+-------------+
| 0       | 1           |
| 1       | 0           |
| 2       | 0           |
| 2       | 1           |
+---------+-------------+
Result table:
+---------+----------------+
| user_id | followers_count|
+---------+----------------+
| 0       | 1              |
| 1       | 1              |
| 2       | 2              |
+---------+----------------+
The followers of 0 are {1}
The followers of 1 are {0}
The followers of 2 are {0,1}
```

lution

sql

```
SELECT user_id, COUNT(DISTINCT follower_id) followers_count
FROM followers
GROUP BY user_id
ORDER BY user_id
```

1731. The Number of Employees Which Report to Each Employee | Easy | LeetCode

Table: `Employees`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| employee_id | int      |
| name        | varchar  |
| reports_to  | int      |
| age         | int      |
+-------------+----------+
employee_id is the primary key for this table.
This table contains information about the employees and the id of the manager
```

For this problem, we will consider a **manager** an employee who has at least 1 other employee reporting to them.

Write an SQL query to report the ids and the names of all **managers**, the number of employees who report **directly** to them, and the average age of the reports rounded to the nearest integer.

Return the result table ordered by `employee_id`.

The query result format is in the following example:

```
Employees table:
+-------------+---------+------------+-----+
| employee_id | name    | reports_to | age |
+-------------+---------+------------+-----+
```

```
| 9           | Hercy   | null       | 43  |
| 6           | Alice   | 9          | 41  |
| 4           | Bob     | 9          | 36  |
| 2           | Winston | null       | 37  |
+-------------+---------+------------+-----+

Result table:
+-------------+-------+---------------+-------------+
| employee_id | name  | reports_count | average_age |
+-------------+-------+---------------+-------------+
| 9           | Hercy | 2             | 39          |
+-------------+-------+---------------+-------------+
Hercy has 2 people report directly to him, Alice and Bob. Their average age :
```

Solution

sql

```sql
SELECT e1.reports_to AS employee_id,
       e2.name,
       COUNT(e1.reports_to) AS reports_count,
       ROUND(AVG(e1.age),0) AS average_age
FROM employees e1
JOIN employees e2
ON e1.reports_to=e2.employee_id
GROUP BY e1.reports_to
ORDER BY e1.reports_to
```

1741. Find Total Time Spent by Each Employee | Easy | LeetCode

Table: **Employees**

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| emp_id      | int  |
| event_day   | date |
| in_time     | int  |
```

```
| out_time    | int  |
+-------------+------+
(emp_id, event_day, in_time) is the primary key of this table.
The table shows the employees' entries and exits in an office.
event_day is the day at which this event happened and in_time is the minute a
It's guaranteed that no two events on the same day intersect in time.
```

Write an SQL query to calculate the total time **in minutes** spent by each employee on each day at the office. Note that within one day, an employee can enter and leave more than once.

Return the result table in **any order**.

The query result format is in the following example:

```
Employees table:
+--------+------------+---------+----------+
| emp_id | event_day  | in_time | out_time |
+--------+------------+---------+----------+
| 1      | 2020-11-28 | 4       | 32       |
| 1      | 2020-11-28 | 55      | 200      |
| 1      | 2020-12-03 | 1       | 42       |
| 2      | 2020-11-28 | 3       | 33       |
| 2      | 2020-12-09 | 47      | 74       |
+--------+------------+---------+----------+
Result table:
+------------+--------+------------+
| day        | emp_id | total_time |
+------------+--------+------------+
| 2020-11-28 | 1      | 173        |
| 2020-11-28 | 2      | 30         |
| 2020-12-03 | 1      | 41         |
| 2020-12-09 | 2      | 27         |
+------------+--------+------------+
Employee 1 has three events two on day 2020-11-28 with a total of (32 - 4) +
Employee 2 has two events one on day 2020-11-28 with a total of (33-3) = 30 a
```

lution

```sql
SELECT event_day AS day, emp_id, SUM(out_time - in_time) AS total_time
FROM Employees
GROUP BY day, emp_id
```

1747. Leetflex Banned Accounts | Medium | LeetCode

Table: **LogInfo**

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| account_id  | int      |
| ip_address  | int      |
| login       | datetime |
| logout      | datetime |
+-------------+----------+
There is no primary key for this table, and it may contain duplicates.
The table contains information about the login and logout dates of Leetflex a
It is guaranteed that the logout time is after the login time.
```

Write an SQL query to find the **account_id** of the accounts that should be banned from Leetflex. An account should be banned if it was logged in at some moment from two different IP addresses.

Return the result table in any order.

The query result format is in the following example:

```
LogInfo table:
+------------+------------+---------------------+---------------------+
| account_id | ip_address | login               | logout              |
+------------+------------+---------------------+---------------------+
| 1          | 1          | 2021-02-01 09:00:00 | 2021-02-01 09:30:00 |
| 1          | 2          | 2021-02-01 08:00:00 | 2021-02-01 11:30:00 |
| 2          | 6          | 2021-02-01 20:30:00 | 2021-02-01 22:00:00 |
| 2          | 7          | 2021-02-02 20:30:00 | 2021-02-02 22:00:00 |
| 3          | 9          | 2021-02-01 16:00:00 | 2021-02-01 16:59:59 |
```

```
| 3          | 13         | 2021-02-01 17:00:00 | 2021-02-01 17:59:59 |
| 4          | 10         | 2021-02-01 16:00:00 | 2021-02-01 17:00:00 |
| 4          | 11         | 2021-02-01 17:00:00 | 2021-02-01 17:59:59 |
+------------+------------+---------------------+---------------------+

Result table:
+------------+
| account_id |
+------------+
| 1          |
| 4          |
+------------+
Account ID 1 --> The account was active from "2021-02-01 09:00:00" to "2021-(
Account ID 2 --> The account was active from two different addresses (6, 7) t
Account ID 3 --> The account was active from two different addresses (9, 13)
Account ID 4 --> The account was active from "2021-02-01 17:00:00" to "2021-(
```

Solution

sql

```sql
SELECT DISTINCT l1.account_id
FROM LogInfo l1
JOIN LogInfo l2
ON l1.account_id = l2.account_id AND l1.ip_address != l2.ip_address
WHERE NOT (l1.login > l2.logout OR l1.logout < l2.login)
```

1757. Recyclable and Low Fat Products | Easy | LeetCode

Table: **Products**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_id  | int     |
| low_fats    | enum    |
| recyclable  | enum    |
+-------------+---------+
product_id is the primary key for this table.
```

```
low_fats is an ENUM of type ('Y', 'N') where 'Y' means this product is low fa
recyclable is an ENUM of types ('Y', 'N') where 'Y' means this product is rec
```

Write an SQL query to find the ids of products that are both low fat and recyclable.

Return the result table in **any order**.

The query result format is in the following example:

```
Products table:
+-------------+----------+------------+
| product_id  | low_fats | recyclable |
+-------------+----------+------------+
| 0           | Y        | N          |
| 1           | Y        | Y          |
| 2           | N        | Y          |
| 3           | Y        | Y          |
| 4           | N        | N          |
+-------------+----------+------------+
Result table:
+-------------+
| product_id  |
+-------------+
| 1           |
| 3           |
+-------------+
Only products 1 and 3 are both low fat and recyclable.
```

Solution

sql

```sql
SELECT product_id
FROM Products
WHERE low_fats = "Y" AND recyclable = "Y"
```

/67. Find the Subtasks That Did Not Execute | Hard | LeetCode

Table: `Tasks`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| task_id        | int     |
| subtasks_count | int     |
+----------------+---------+
task_id is the primary key for this table.
Each row in this table indicates that task_id was divided into subtasks_count
It is guaranteed that 2 <= subtasks_count <= 20.
```

Table: `Executed`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| task_id       | int     |
| subtask_id    | int     |
+---------------+---------+
(task_id, subtask_id) is the primary key for this table.
Each row in this table indicates that for the task task_id, the subtask with
It is guaranteed that subtask_id <= subtasks_count for each task_id.
```

Write an SQL query to report the IDs of the missing subtasks for each `task_id`.

Return the result table in **any order**.

The query result format is in the following example:

```
Tasks table:
+---------+----------------+
| task_id | subtasks_count |
+---------+----------------+
| 1       | 3              |
| 2       | 2              |
| 3       | 4              |
+---------+----------------+
```

Executed table:

```
+---------+------------+
| task_id | subtask_id |
+---------+------------+
| 1       | 2          |
| 3       | 1          |
| 3       | 2          |
| 3       | 3          |
| 3       | 4          |
+---------+------------+
```

Result table:

```
+---------+------------+
| task_id | subtask_id |
+---------+------------+
| 1       | 1          |
| 1       | 3          |
| 2       | 1          |
| 2       | 2          |
+---------+------------+
Task 1 was divided into 3 subtasks (1, 2, 3). Only subtask 2 was executed suc
Task 2 was divided into 2 subtasks (1, 2). No subtask was executed successfu
Task 3 was divided into 4 subtasks (1, 2, 3, 4). All of the subtasks were exe
```

Solution

sql

```sql
WITH RECURSIVE CTE AS
  (SELECT 1 AS subtask_id
   UNION ALL SELECT subtask_id + 1
   FROM CTE
   WHERE subtask_id <
       (SELECT MAX(subtasks_count)
        FROM Tasks) )
SELECT Tasks.task_id,
       CTE.subtask_id
FROM CTE
INNER JOIN Tasks ON CTE.subtask_id <= Tasks.subtasks_count
LEFT JOIN Executed ON Tasks.task_id = Executed.task_id
                      AND CTE.subtask_id = Executed.subtask_id
WHERE Executed.subtask_id IS NULL
```

```
ORDER BY NULL
```

1777. Product's Price for Each Store | Easy | LeetCode

Table: **Products**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_id  | int     |
| store       | enum    |
| price       | int     |
+-------------+---------+
(product_id,store) is the primary key for this table.
store is an ENUM of type ('store1', 'store2', 'store3') where each represents
price is the price of the product at this store.
```

Write an SQL query to find the price of each product in each store.

Return the result table in **any order**.

The query result format is in the following example:

```
Products table:
+-------------+--------+-------+
| product_id  | store  | price |
+-------------+--------+-------+
| 0           | store1 | 95    |
| 0           | store3 | 105   |
| 0           | store2 | 100   |
| 1           | store1 | 70    |
| 1           | store3 | 80    |
+-------------+--------+-------+
Result table:
+-------------+--------+--------+--------+
| product_id  | store1 | store2 | store3 |
+-------------+--------+--------+--------+
| 0           | 95     | 100    | 105    |
```

```
| 1           | 70     | null   | 80     |
+-------------+--------+--------+--------+
Product 0 price's are 95 for store1, 100 for store2 and, 105 for store3.
Product 1 price's are 70 for store1, 80 for store3 and, it's not sold in sto
```

Solution

sql

```sql
SELECT product_id,
       SUM(CASE WHEN store='store1' THEN price END) AS store1,
       SUM(CASE WHEN store='store2' THEN price END) AS store2,
       SUM(CASE WHEN store='store3' THEN price END) AS store3
FROM Products
GROUP BY product_id
```

1783. Grand Slam Titles | Medium | LeetCode

Table: **Players**

```
+---------------+---------+
| Column Name    | Type    |
+---------------+---------+
| player_id      | int     |
| player_name    | varchar |
+---------------+---------+
player_id is the primary key for this table.
Each row in this table contains the name and the ID of a tennis player.
```

Table: **Championships**

```
+--------------+---------+
| Column Name   | Type    |
+--------------+---------+
| year          | int     |
| Wimbledon     | int     |
| Fr_open       | int     |
| US_open       | int     |
```

```
| Au_open       | int     |
+---------------+---------+
year is the primary key for this table.
Each row of this table containts the IDs of the players who won one each tenn
```

Write an SQL query to report the number of grand slam tournaments won by each player. Do not include the players who did not win any tournament.

Return the result table in **any order**.

The query result format is in the following example:

```
Players table:
+-----------+-------------+
| player_id | player_name |
+-----------+-------------+
| 1         | Nadal       |
| 2         | Federer     |
| 3         | Novak       |
+-----------+-------------+

Championships table:
+------+-----------+---------+---------+---------+
| year | Wimbledon | Fr_open | US_open | Au_open |
+------+-----------+---------+---------+---------+
| 2018 | 1         | 1       | 1       | 1       |
| 2019 | 1         | 1       | 2       | 2       |
| 2020 | 2         | 1       | 2       | 2       |
+------+-----------+---------+---------+---------+

Result table:
+-----------+-------------+-------------------+
| player_id | player_name | grand_slams_count |
+-----------+-------------+-------------------+
| 2         | Federer     | 5                 |
| 1         | Nadal       | 7                 |
+-----------+-------------+-------------------+

Player 1 (Nadal) won 7 titles: Wimbledon (2018, 2019), Fr_open (2018, 2019, 2
Player 2 (Federer) won 5 titles: Wimbledon (2020), US_open (2019, 2020), and
```

Player 3 (Novak) did not win anything, we did not include them in the result

Solution

sql

```sql
#Solution 1:
SELECT player_id, player_name,
    SUM((IF(Wimbledon = player_id,1,0) +
        IF(Fr_open = player_id,1,0) +
        IF(US_open = player_id,1,0) +
        IF(Au_open = player_id,1,0))) as grand_slams_count
FROM Players INNER JOIN Championships
ON Wimbledon = player_id OR Fr_open = player_id OR US_open = player_id OR Au_
GROUP BY player_id;

#Solution 2:
WITH cte
     AS (SELECT wimbledon AS id
         FROM   championships
         UNION ALL
         SELECT fr_open AS id
         FROM   championships
         UNION ALL
         SELECT us_open AS id
         FROM   championships
         UNION ALL
         SELECT au_open AS id
         FROM   championships)
SELECT player_id,
       player_name,
       Count(*) AS grand_slams_count
FROM   players
       INNER JOIN cte
               ON players.player_id = cte.id
GROUP  BY 1, 2
ORDER  BY NULL;
```

1789. Primary Department for Each Employee | Easy | LeetCode

Table: **Employee**

```
+-------------+---------+
| Column Name  |  Type   |
+-------------+---------+
| employee_id  | int     |
| deprtment_id | int     |
| primary_flag | varchar |
+-------------+---------+
(employee_id, department_id) is the primary key for this table.
employee_id is the id of the employee.
department_id is the id of the department to which the employee belongs.
primary_flag is an ENUM of type ('Y', 'N'). If the flag is 'Y', the departmer
```

Employees can belong to multiple departments. When the employee joins other departments, they need to decide which department is their primary department. Note that when an employee belongs to only one department, their primary column is `'N'`.

Write an SQL query to report all the employees with their primary department. For employees who belong to one department, report their only department.

Return the result table in any order.

The query result format is in the following example.

```
Employee table:
+-------------+---------------+-------------+
| employee_id | department_id | primary_flag |
+-------------+---------------+-------------+
| 1           | 1             | N           |
| 2           | 1             | Y           |
| 2           | 2             | N           |
| 3           | 3             | N           |
| 4           | 2             | N           |
| 4           | 3             | Y           |
| 4           | 4             | N           |
+-------------+---------------+-------------+

Result table:
+-------------+---------------+
```

```
| employee_id | department_id |
+-------------+---------------+
| 1           | 1             |
| 2           | 1             |
| 3           | 3             |
| 4           | 3             |
+-------------+---------------+
- The Primary department for employee 1 is 1.
- The Primary department for employee 2 is 1.
- The Primary department for employee 3 is 3.
- The Primary department for employee 4 is 3.
```

Solution

sql

```sql
#Solution 1:
SELECT employee_id,department_id
FROM employee
WHERE primary_flag = 'Y' OR employee_id IN
    (SELECT employee_id
     FROM employee
     GROUP BY employee_id
     HAVING COUNT(department_id) = 1)

#Solution 2:
(SELECT employee_id,
      department_id
FROM Employee
WHERE primary_flag = 'Y')
UNION
(SELECT employee_id,
      department_id
FROM Employee
GROUP BY employee_id
HAVING COUNT(employee_id) = 1
ORDER BY NULL);
```

← HackerRank SQL Problem Solving Questions With Solutions