176. Second Highest Salary

Write 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                 |
+---------------------+
```

MY SQL

```
SELECT
(SELECT DISTINCT Salary
FROM Employee
ORDER BY Salary DESC
LIMIT 1 OFFSET 1) AS SecondHighestSalary;
```

MY SQL

```
SELECT
   IFNULL(
     (SELECT DISTINCT Salary
      FROM Employee
      ORDER BY Salary DESC
       LIMIT 1 OFFSET 1),
   NULL) AS SecondHighestSalary
```

175. Combine Two Tables

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

我的：

```
SELECT Firstname, LastName, City, State

FROM Person LEFT JOIN Address

ON Person.PersonID = Address.PersonID;
```

181. Employees Earning More Than Their Managers

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

MySQL

```
SELECT a.Name AS 'Employee'

FROM Employee AS a, Employee AS b

WHERE a.ManagerId = b.Id

AND a.Salary > b.Salary;
```

```
SELECT a.NAME AS Employee

FROM Employee AS a JOIN Employee AS b

ON a.ManagerId = b.Id

AND a.Salary > b.Salary;
```

```
SELECT E.name as Employee from Employee E

JOIN Employee as M

ON E.ManagerId = M.Id

WHERE E.Salary > M.Salary;
```

196. Delete Duplicate Emails

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  |
```

MY SQL

```
DELETE p1
FROM Person p1, Person p2
WHERE p1.Email = p2.Email
AND p1.Id > p2.Id;
```

182. Duplicate Emails

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.

```
SELECT DISTINCT p1.Email
FROM Person p1, Person p2
WHERE p1.Email = p2.Email and p1.id != p2.id;
```

183. Customers Who Never Order

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

```
SELECT Name AS 'Customers'

FROM Customers C

LEFT JOIN Orders O

ON C.Id = O.CustomerId

WHERE O.CustomerId IS NULL;
```

197. Rising Temperature

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).

SELECT weather.id AS 'Id'

FROM weather

JOIN weather w ON DATEDIFF(weather.recordDate, w.recordDate) = 1

AND weather.Temperature > w.Temperature;

511. Game Play Analysis I

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 (possibly 0) before logging out on some day using some device.

Write 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  |
+-----------+-------------+
```

SELECT player_id, MIN(event_date) as first_login

FROM Activity

GROUP BY player_id

ORDER BY player_id;

512. Game Play Analysis II

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 (possibly 0) before logging out on some day
using some device.
```

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

```
SELECT a1.player_id, a1.device_id
FROM
   (SELECT player_id, MIN(event_date), device_id
    FROM Activity
    GROUP BY player_id) AS a1 ;
```

从最早`login date`的`subquery`里面选`device id`

577. Employee Bonus

Select 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  |
```


```
select E.name, B.bonus
from Employee E
left join Bonus B
on B.empId = E.empId
where B.empId not in(select empId from Bonus where bonus >=
1000) or B.empId is null
```

627. Swap Salary

Given a table `salary`, such as the one below, that has m=male and f=female values. 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.

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

Example:

```
| id | name | sex | salary |
|----|------|-----|--------|
| 1  | A    | m   | 2500   |
| 2  | B    | f   | 1500   |
```

```
| 3   | C     | m    | 5500    |
| 4   | D     | f    | 500     |
```

After running your **update** statement, the above salary table should have the following rows:

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

UPDATE salary

SET sex =

CASE sex

WHEN 'm' THEN ‘f’

ELSE 'm'

END;

1179. Reformat Department Table

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","Oct","Nov","Dec"].

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 the months).

```
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;
```

1517. Find Users With Valid E-Mails

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-mails are invalid.

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.

```
SELECT user_id, name, mail
FROM users
WHERE mail REGEXP '^[a-zA-Z][a-zA-Z0-9_.-]*@leetcode.com$';
```

597. Friend Requests I: Overall Acceptance Rate

Table: FriendRequest

+----------------+---------+

| Column Name | Type |

+----------------+---------+

| sender_id | int |

| send_to_id | int |

| request_date | date |

+----------------+---------+

There is no primary key for this table, it may contain duplicates.

This table contains the ID of the user who sent the request, the ID of the user who received the request, and the date of the request.

Table: RequestAccepted

+----------------+---------+

| Column Name | Type |

+----------------+---------+

| requester_id | int |

| accepter_id | int |

| accept_date | date |

+----------------+---------+

There is no primary key for this table, it may contain duplicates.

This table contains the ID of the user who sent the request, the ID of the user who received the request, and the date when the request was accepted.

Write an SQL query to find the overall acceptance rate of requests, which is the number of acceptance divided by the number of requests. Return the answer rounded to 2 decimals places.

Note that:

- 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 are no requests at all, you should return 0.00 as the accept_rate.

The query result format is in the following example:

FriendRequest table:

+-----------+------------+--------------+

| 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 |

+-----------+------------+--------------+

RequestAccepted table:

+--------------+-------------+-------------+

| 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 |

+--------------+-------------+-------------+

Result table:

+-------------+

| accept_rate |

+-------------+

| 0.8 |

+-------------+

There are 4 unique accepted requests, and there are 5 requests in total. So the rate is 0.80.

```
SELECT  ROUND(IFNULL( COUNT(DISTINCT r.requester_id, r.accepter_id)
                    / COUNT(DISTINCT f.sender_id, f.send_to_id), 0), 2) AS accept_rate
FROM FriendRequest f, RequestAccepted r;
```

1435. Create a Session Bar Chart

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           | 199           |
| 3           | 299           |
| 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 less than 5 minutes.

For session_id 4 has a duration greater or equal than 5 minutes and less than 10 minutes.

There are no session with a duration greater or equial than 10 minutes and less than 15 minutes.

For session_id 5 has a duration greater or equal than 15 minutes.

```
SELECT'[0-5>' AS bin, COUNT(duration) AS total
FROM Sessions
WHERE duration < 5 * 60
```

```
UNION

SELECT '[5-10>' AS bin,
COUNT(duration) AS total
FROM Sessions
WHERE duration >= 5 * 60 AND duration < 10 * 60

UNION

SELECT '[10-15>' AS bin, COUNT(duration) AS total
FROM Sessions
WHERE duration >= 10 * 60 AND duration < 15 * 60

UNION

SELECT '15 or more' AS bin, COUNT(duration) AS total
FROM Sessions
WHERE duration >= 15 * 60;
```

610. Triangle Judgement

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 |

1.

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

2.

```
SELECT *,
IF(x + y > z AND x + z > y AND y + z > x, 'Yes', 'No') AS triangle
FROM triangle;
```

1173. Immediate Food Delivery I

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 at some date and specify a preferred delivery date (on the same order date or after it).

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 scheduled.

1.

```
WITH CTE AS
(SELECT *
FROM Delivery
WHERE order_date = customer_pref_delivery_date)

SELECT round((COUNT(CTE.delivery_id) / COUNT(d.delivery_id) *100),2) AS
immediate_percentage
FROM Delivery d
LEFT JOIN CTE
ON d.delivery_id = CTE.delivery_id;
```

2.

```
SELECT round((select count(delivery_id)
FROM delivery
WHERE order_date = customer_pref_delivery_date) / count(delivery_id) * 100,2) AS immediate_percentage
FROM delivery;
```

1141. User Activity for the Past 30 Days I

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', 'scroll_down', 'send_message').

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.

1.

```
SELECT activity_date AS day, COUNT(DISTINCT user_id) AS active_users
FROM activity
WHERE (activity_date BETWEEN '2019-06-28' AND '2019-07-27')
GROUP BY activity_date;
```

2.

```
SELECT activity_date AS day,
COUNT(DISTINCT user_id) AS active_users
FROM Activity
GROUP BY 1
HAVING DATEDIFF('2019-07-27', day) < 30 AND active_users >= 1;
```

3.

```
SELECT activity_date AS day, COUNT(DISTINCT user_id) AS active_users
FROM Activity
WHERE activity_date BETWEEN date_add("2019-07-27", INTERVAL -29 Day) AND "2019-07-27"
GROUP BY activity_date;
```

4.

```
SELECT activity_date AS day, COUNT(DISTINCT user_id) AS active_users
FROM Activity
WHERE DATEDIFF('2019-07-27', activity_date) BETWEEN 0 AND 30
GROUP BY activity_date;
```

1084. Sales Analysis III

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 **products** that were **only** sold in spring 2019. That is, between **2019-01-01** and **2019-03-31** inclusive.

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:

+-------------+--------------+

| product_id | product_name |

+-------------+--------------+

| 1 | S8 |

+-------------+--------------+

The product with id 1 was only sold in spring 2019 while the other two were sold after.

```
SELECT sales.product_id, product_name
```

```
FROM product
        JOIN sales ON product.product_id = sales.product_id
GROUP BY product_id
HAVING SUM(sale_date BETWEEN '2019-01-01' AND '2019-03-31') = COUNT(sale_date);
```

2021/1/8

1350. Students With Invalid Departments

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 the id of the department he/she studies at.

Write an SQL query to find the id and the name of all students who are enrolled in departments that no longer exist.

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 77 respectively. department 14, 33, 74 and 77 doesn't exist in the Departments table.

```
SELECT id, name
FROM Students
WHERE department_id NOT IN
(SELECT id FROM Departments);
```

1083. Sales Analysis II

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 3 bought both.

```
SELECT DISTINCT buyer_id
FROM sales JOIN product ON sales.product_id = product.product_id
WHERE product_name = 'S8' AND buyer_id NOT IN
(SELECT DISTINCT buyer_id
 FROM sales JOIN product ON sales.product_id = product.product_id
 WHERE product_name = 'iPhone' );
```

1251. Average Selling Price

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 from start_date to end_date.

For each product_id there will be no two overlapping periods. That means there will be no two intersecting periods for the same product_id.

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 product sold.

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

SELECT a.product_id,ROUND(SUM(b.units*a.price)/SUM(b.units),2) AS average_price

FROM Prices AS a

JOIN UnitsSold AS b

ON a.product_id = b.product_id AND (b.purchase_date BETWEEN a.start_date AND a.end_date)

GROUP BY product_id;

584. Find Customer Referee

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

SELECT name

FROM Customer

`WHERE referee_id != 2 OR referee_id IS NULL;` !=可以换成<>

620. Not Boring Movies

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

```
SELECT *
FROM cinema
WHERE MOD(id,2) = 1 AND description <> 'boring'
ORDER BY rating DESC;
```

```
SELECT *
FROM cinema
WHERE (id % 2 = 1) AND (description <> 'boring')
ORDER BY rating DESC;
```

1068. Product Sales Analysis I

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 |

+--------------+-------+-------+

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

```
SELECT DISTINCT P.product_name, S.year, S.price
FROM
(SELECT DISTINCT product_id, year, price FROM Sales) S
INNER JOIN Product AS P ON S.product_id = P.product_id;
```

586. Customer Placing the Largest Number of Orders

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	status	comment
1	1	2017-04-09	2017-04-13	2017-04-12	Closed	
2	2	2017-04-15	2017-04-20	2017-04-18	Closed	
3	3	2017-04-16	2017-04-25	2017-04-20	Closed	

| 4 | 3 | 2017-04-18 | 2017-04-28 | 2017-04-25 | Closed | |

Sample Output

| customer_number |

|-----------------|

| 3 |

Explanation

The customer with number '3' has two orders, which is greater than either customer '1' or '2' because each of them only has one order.

So the result is customer_number '3'.

```
SELECT customer_number
FROM orders
GROUP BY customer_number
ORDER BY COUNT(*) DESC
LIMIT 1;
```

```
SELECT customer_number
FROM orders
GROUP BY customer_number
ORDER BY count(*) desc limit 1;
```

1543. Fix Product Name Format

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**.

The 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 case sensitive and may contain spaces.

In Februery, 2 LCKeychains and 1 LCPhone were sold.

In March, 1 matryoshka was sold.

```
SELECT LOWER(TRIM(product_name)) product_name, DATE_FORMAT(sale_date,
"%Y-%m") sale_date, count(sale_id) AS total
FROM sales
GROUP BY 1, 2
ORDER BY 1, 2;
```

595. Big Countries

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

```
SELECT name, population, area
FROM World
WHERE area > 3000000 OR population > 25000000;
```

1211. Queries Quality and Percentage

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 poor query.

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

```
SELECT
    query_name,
    ROUND(AVG(rating / position), 2) AS quality,
    ROUND(AVG(rating < 3) * 100, 2) AS poor_query_percentage
FROM
    Queries
GROUP BY
    query_name;
```

2020/1/10

577. Employee Bonus

Select 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 |

+-------+-------+

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

596. Classes More Than 5 Students

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

Note:

The students should not be counted duplicate in each course.

```
SELECT class
FROM
(SELECT class, COUNT(DISTINCT student) AS num
FROM courses
GROUP BY class) AS temp_table
WHERE num >= 5;
```

```
SELECT class
FROM courses
GROUP BY class
HAVING COUNT (distinct student) >= 5;
```

603. Consecutive Available Seats

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.

```
SELECT distinct a.seat_id
FROM cinema a JOIN cinema b
ON ABS (a.seat_id - b.seat_id) = 1
AND a.free = true AND b.free=true
ORDER BY a.seat_id;
```

607. Sales Person

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	order_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 'Pam' have sales to company 'RED',

so we need to output all the other **names** in the table salesperson.

```
SELECT salesperson.name

FROM salesperson

WHERE salesperson.sales_id NOT IN
```

```
(SELECT orders.sales_id
FROM orders LEFT JOIN company ON orders.com_id = company.com_id
WHERE company.name = 'RED');


SELECT salesperson.name
FROM orders o JOIN company c ON (o.com_id = c.com_id AND c.name = 'RED')
RIGHT JOIN salesperson ON salesperson.sales_id = o.sales_id
WHERE o.sales_id IS NULL;
```

613. Shortest Distance in a Line

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.

```
SELECT MIN(ABS(p1.x - p2.x)) AS shortest
FROM point p1 JOIN point p2 ON p1.x != p2.x;
```

619. Biggest Single Number

Table my_numbers 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**.

```
 SELECT(
  SELECT num
  FROM my_numbers
  GROUP BY num
  HAVING COUNT (*) = 1
  ORDER BY num DESC LIMIT 1
) AS num;
```

1050. Actors and Directors Who Cooperated At Least Three Times

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.

```
SELECT actor_id, director_id
FROM ActorDirector
GROUP BY actor_id, director_id
HAVING COUNT(actor_id) >= 3;
```

1069. Product Sales Analysis II

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 |

+--------------+----------------+

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

1075. Project Employees I

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 and for the second project is (3 + 2) / 2 = 2.50

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

这里是交集用`inner join`

1076. Project Employees II

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.

```
SELECT project_id
FROM project
GROUP BY project_id
HAVING COUNT(employee_id) =
(
    SELECT count(employee_id) AS cnt
    FROM project
    GROUP BY project_id
    ORDER BY cnt desc LIMIT 1
```

);

2021/1/11

1082. Sales Analysis I

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.

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

1113. Reported Posts

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', 'report', 'share').

The extra column has optional information about the action such as a reason for report or a type of reaction.

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.

```
SELECT extra AS report_reason, COUNT(DISTINCT post_id) AS report_count
FROM Actions
WHERE action_date = '2019-07-04' AND action = 'report'
GROUP BY extra;
```

1142. User Activity for the Past 30 Days II

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', 'scroll_down', 'send_message').

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 sessions so the average is (1 + 1 + 2) / 3 = 1.33.

```
SELECT IFNULL(ROUND(COUNT(DISTINCT user_id,session_id)/COUNT(DISTINCT user_id),2),0) AS average_sessions_per_user
FROM Activity
```

WHERE activity_date BETWEEN '2019-06-28' AND '2019-07-27';

1148. Article Views I

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 by some author) on some date.

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 |

+------+

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

1241. Number of Comments per Post

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 comment with id 3 is repeated in the table, we counted it **only once**.

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 it.

```
SELECT S1.sub_id AS post_id, COUNT(DISTINCT S2.sub_id) AS number_of_comments
FROM Submissions S1 LEFT JOIN Submissions S2 ON S1.sub_id = S2.parent_id
```

```
WHERE S1.parent_id IS NULL
GROUP BY S1.sub_id;
```

1280. Students and Examinations

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 school.

Table: Subjects

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| subject_name | varchar |
+--------------+---------+
```

subject_name is the primary key for this table.

Each row of this table contains the 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 the Students table takes every course from Subjects table.

Each row of this table indicates that a student with ID student_id attended the exam of subject_name.

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 time.

Bob attended Math exam 1 time, Programming exam 1 time and didn't attend the Physics exam.

Alex didn't attend any exam.

John attended Math exam 1 time, Physics exam 1 time and Programming exam 1 time.

```
SELECT a.student_id, a.student_name, b.subject_name,COUNT(c.subject_name)
AS attended_exams
FROM Students a CROSS JOIN Subjects b LEFT JOIN Examinations c
ON a.student_id = c.student_id AND b.subject_name = c.subject_name
GROUP BY a.student_id, b.subject_name
ORDER BY student_id,subject_name;
```

1294. Weather Type in Each Country

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 Cold.

Average weather_state in Austraila in November is (-2 + 0 + 3) / 3 = 0.333 so weather type is Cold.

Average weather_state in Peru in November is (25) / 1 = 25 so weather type is Hot.

Average weather_state in China in November is (16 + 18 + 21) / 3 = 18.333 so weather type is Warm.

Average weather_state in Morocco in November is (25 + 27 + 31) / 3 = 27.667 so weather type is Hot.

We know nothing about average weather_state in Spain in November so we don't include it in the result table.

```
SELECT a.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 a JOIN Weather b ON a.country_id = b.country_id
WHERE b.day BETWEEN "2019-11-01" AND "2019-11-30"
GROUP BY a.country_name;
```

1303. Find the Team Size

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 team.

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.

```
SELECT e.employee_id,

(SELECT COUNT(team_id)
 FROM Employee a
 WHERE e.team_id = a.team_id) AS team_size
```

FROM Employee e;

WINDOW FUNCTION:

SELECT employee_id, COUNT(*) over(PARTITION BY team_id) AS team_size

FROM employee;

1322. Ads Performance

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 action taken by this user regarding this Ad.

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.

Performance of the Ad is measured using Click-Through Rate (CTR) where:

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

Write an SQL query to find the ctr of each Ad.

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

```
SELECT ad_id, IFNULL(ROUND(AVG(CASE WHEN action = 'Clicked' THEN1
                                    WHEN action = 'Viewed' THEN 0
                               ELSE NULL
                               END)*100,2),0) AS ctr
FROM Ads
GROUP BY ad_id
ORDER BY ctr DESC, ad_id ASC;
```

1327. List the Products Ordered in a Period

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) = 130.

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) = 100.

```
SELECT product_name, SUM(unit) AS unit
FROM Products a LEFT JOIN Orders b ON a.product_id = b.product_id
WHERE month(order_date) = 2 AND year(order_date) = '2020'
GROUP BY a.product_id
HAVING unit >= 100;
```

2021/1/12

1378. Replace Employee ID With The Unique Identifier

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 company.

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 employee in the company.

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.

```
SELECT b.unique_id, a.name
FROM Employees a LEFT JOIN EmployeeUNI b ON a.id = b.id;
```

1407. Top Travellers

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.

user_id is the id of the user who travelled the distance "distance".

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 alphabetically smaller than Lee.

Bob, Jonathan, Alex and Alice have only one ride and we just order them by the total distances of the ride.

Donald didn't have any rides, the distance travelled by him is 0.

```
SELECT u.name, ifnull (sum(r.distance), 0) AS travelled_distance
FROM users u LEFT JOIN rides r ON u.id = r.user_id
GROUP BY r.user_id
ORDER BY travelled_distance DESC, u.name ASC;
```

1484. Group Sold Products By The Date

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 a market.

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 them lexicographically and separate them by comma.

For 2020-06-01, Sold items were (Pencil, Bible), we sort them lexicographically and separate them by comma.

For 2020-06-02, Sold item is (Mask), we just return it.

```
SELECT sell_date,
        COUNT(DISTINCT(product)) AS num_sold,
        GROUP_CONCAT(DISTINCT (product) ORDER BY product ASC SEPARATOR ',') AS products
FROM Activities
```

```
GROUP BY sell_date
ORDER BY sell_date ASC;
```

1495. Friendly Movies Streamed Last Month

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.

The 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.

```
SELECT DISTINCT c.title
FROM TVProgram p
JOIN Content c ON p.content_id = c.content_id
WHERE
        (p.program_date BETWEEN '2020-06-01' AND '2020-06-30')
AND c.Kids_content = 'Y'
```

```
AND c.content_type = 'Movies';
```

1511. Customer Order Frequency

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 "product_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 2020.

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.

```
SELECT c.customer_id, c.name
FROM Customers c JOIN Orders o ON c.customer_id = o.customer_id
                 JOIN Product p ON o.product_id = p.product_id
GROUP BY c.customer_id
HAVING SUM(IF(LEFT(order_date, 7) = '2020-06', quantity, 0) * price) >= 100
   AND SUM(IF(LEFT(order_date, 7) = '2020-07', quantity, 0) * price) >= 100;
```

2021/1/13

1527. Patients With a Condition

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.

```
SELECT *
FROM PATIENTS
WHERE CONDITIONS LIKE '% DIAB1%' OR CONDITIONS LIKE 'DIAB1%';
```

1565. Unique Orders and Customers Per Month

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 customers** 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 > $20.

In October 2020 we have two orders from 1 customer, and only one of the two orders has invoice > $20.

In November 2020 we have two orders from 2 different customers but invoices < $20, so we don't include that month.

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 of them with invoice > $20.

```
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

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 warehouse.

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 (Width, Lenght and Height) in feets of each product.

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.

The 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.

```
SELECT w.name AS warehouse_name, SUM(units * Width * Length * Height)
AS volume
FROM Warehouse w LEFT JOIN Products p
ON w.product_id = p.product_id
GROUP BY warehouse_name;
```

1581. Customer Who Visited but Did Not Make Any Transactions

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 transactions made during the visit_id.

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 order**.

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 the visit with id = 12.

Customer with id = 9 visited the mall once and made one transaction during the visit with id = 13.

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 not make any transactions, and during one visit they made 3 transactions.

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 making any transactions. Also user 54 visited the mall twice and did not make any transactions.

```
SELECT customer_id,COUNT(v.visit_id) AS count_no_trans
FROM Visits v LEFT JOIN Transactions t ON v.visit_id = t.visit_id
WHERE t.visit_id IS NULL
GROUP BY customer_id;
```

1587. Bank Account Summary II

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 transferred money.

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.

```
SELECT a.name, SUM(b.amount) AS balance
FROM Users a JOIN Transactions b ON a.account = b.account
GROUP BY a.account
HAVING balance > 10000;
```

1607. Sellers With No Sales

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 WebStore.

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 (customer_id) and the seller (seller_id).

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.

```
SELECT seller_name
FROM Seller s LEFT JOIN Orders o ON s.seller_id = o.seller_id AND
LEFT(sale_date, 4) = '2020'
WHERE o.seller_id IS NULL
ORDER BY seller_name;
```

1623. All Valid Triplets That Can Represent a Country

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.

Table: 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**.

The 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 name and the same ID.

- (Alice, Tom, Jerry) --> Valid triplet.

- (Alice, Tom, Alice) --> Rejected because member_A and member_C have the same name.

- (Bob, Tom, Tom) --> Rejected because member_B and member_C have the same name and the same ID.

- (Bob, Tom, Jerry) --> Rejected because member_A and member_C have the same ID.

- (Bob, Tom, Alice) --> Valid triplet.

```
SELECT a.student_name AS member_A, b.student_name AS member_B,
c.student_name AS member_C
FROM schoola a CROSS JOIN schoolb b CROSS JOIN schoolc c
    WHERE a.student_name != b.student_name
    AND a.student_name != c.student_name
    AND b.student_name != c.student_name
    AND a.student_id != c.student_id
    AND b.student_id != c.student_id
    AND a.student_id != b.student_id;
```

```
SELECT a.student_name AS member_A, b.student_name AS member_B,
c.student_name AS member_C
FROM SchoolA a, SchoolB b, SchoolC c
WHERE (a.student_name<>b.student_name AND
a.student_name<>c.student_name AND b.student_name<>c.student_name)
AND (a.student_id<>b.student_id AND a.student_id<>c.student_id AND
b.student_id<>c.student_id);
```

1633. Percentage of Users Attended a Contest

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 registered into.

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 100% and we sort them in the answer table by contest_id in ascending order.

Alice and Alex registered in contest 215 and the percentage is ((2/3) * 100) = 66.67%

Bob registered in contest 207 and the percentage is ((1/3) * 100) = 33.33%

SELECT contest_id, ROUND(COUNT(DISTINCT user_id) * 100 / (SELECT COUNT(*) FROM Users), 2) AS percentage

FROM Register

GROUP BY contest_id

ORDER BY percentage DESC, contest_id ASC;

1661. Average Time of Process per Machine

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' means the machine ends the process at the given timestamp.

The 'start' timestamp will always be before the 'end' timestamp for every (machine_id, process_id) pair.

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

```
SELECT s.machine_id, ROUND(AVG(e.timestamp-s.timestamp), 3) AS processing_time
FROM Activity s JOIN Activity e ON s.machine_id = e.machine_id AND s.process_id = e.process_id AND s.activity_type = 'start' AND e.activity_type = 'end'
GROUP BY s.machine_id;
```

1667. Fix Names in a Table

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 only lowercase and uppercase characters.

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

```
SELECT user_id, concat(upper(substring(name, 1,1)), lower(substring(name,2)))
AS name
FROM users
ORDER BY user_id;
```

2021/1/15

1677. Product's Worth Over Invoices

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 only lowercase English letters. No two products have the same name.

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

```
SELECT name, SUM(rest) AS rest, SUM(paid) AS paid, SUM(canceled) AS canceled, SUM(refunded) AS refunded
FROM Product p JOIN Invoice i ON p.product_id = i.product_id
GROUP BY name
ORDER BY name;
```

1683. Invalid Tweets

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.

```
SELECT tweet_id
FROM Tweets
WHERE CHAR_LENGTH(content) > 15;
```

1693. Daily Leads and Partners

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 lead and partner it was sold to.

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 honda gets leads = [1, 2] and partners = [1, 2].

For 2020-12-7, toyota gets leads = [0] and partners = [1, 2] while honda gets leads = [0, 1, 2] and partners = [1, 2].

```
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;
```