# 175. Combine Two Tables

Easy

SQL Schema

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**

**SELECT firstname,**

**lastname,**

**city,**

**state**

**FROM person**

**LEFT JOIN address**

**ON person.personid = address.personid;**

# 176. Second Highest Salary

Easy

SQL Schema

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 |

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

**SELECT isnull(**

**(**

**SELECT DISTINCT salary**

**FROM employee**

**ORDER BY salary DESC**

**offset 1 rows**

**FETCH next 1 rows only ), NULL) AS secondhighestsalary**

# 181. Employees Earning More Than Their Managers

Easy

SQL Schema

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 |

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

**SELECT a.name AS 'Employee'**

**FROM employee AS a,**

**employee AS b**

**WHERE a.managerid = b.id**

**AND a.salary > b.salary;**

# 182. Duplicate Emails

Easy

SQL Schema

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 email**

**FROM person**

**GROUP BY email**

**HAVING Count(email) > 1;**

# 183. Customers Who Never Order

Easy

SQL Schema

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 'Customers'**

**FROM customers c**

**LEFT JOIN orders o**

**ON c.id = o.customerid**

**WHERE o.customerid IS NULL**

**ORDER BY NAME ASC;**

# 196. Delete Duplicate Emails

Easy

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

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

| Id | Email |

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

| 1 | john@example.com |

| 2 | bob@example.com |

| 3 | john@example.com |

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

Id is the primary key column for this table.

For example, after running your query, the above Person table should have the following rows:

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

| Id | Email |

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

| 1 | john@example.com |

| 2 | bob@example.com |

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

Note:

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

**DELETE a**

**FROM person a,**

**person b**

**WHERE a.email = b.email**

**AND a.id > b.id;**

# 197. Rising Temperature

Easy

SQL Schema

Given a Weather table, write a SQL query to find all dates' Ids with higher temperature compared to its previous (yesterday's) dates.

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

| Id(INT) | RecordDate(DATE) | Temperature(INT) |

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

| 1 | 2015-01-01 | 10 |

| 2 | 2015-01-02 | 25 |

| 3 | 2015-01-03 | 20 |

| 4 | 2015-01-04 | 30 |

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

For example, return the following Ids for the above Weather table:

+----+

| Id |

+----+

| 2 |

| 4 |

+----+

SELECT weather.id

FROM weather

JOIN weather AS w

ON w.recorddate = Subdate(weather.recorddate, 1)

WHERE weather.temperature > w.temperature;

# 511. Game Play Analysis I

Easy

SQL Schema

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 a.player\_id,**

**a.event\_date AS 'first\_login'**

**FROM activity a**

**INNER JOIN (SELECT player\_id,**

**Min(event\_date) AS 'ed'**

**FROM activity**

**GROUP BY player\_id) b**

**ON a.player\_id = b.player\_id**

**WHERE a.event\_date = b.ed;**

# 512. Game Play Analysis II

Easy

SQL Schema

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 a.player\_id,**

**a.device\_id**

**FROM activity a**

**RIGHT JOIN (SELECT player\_id,**

**Min(event\_date) AS ev**

**FROM activity**

**GROUP BY player\_id) z**

**ON a.player\_id = z.player\_id**

**WHERE a.event\_date = z.ev**

# 577. Employee Bonus

Easy

SQL Schema

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 AS 'name',**

**b.bonus AS 'bonus'**

**FROM employee e**

**LEFT JOIN bonus b**

**ON e.empid = b.empid**

**WHERE e.supervisor IS NOT NULL**

**AND b.bonus < 1000**

**OR b.bonus IS NULL;**

# 584. Find Customer Referee

Easy

SQL Schema

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 IS NULL**

**OR referee\_id != 2**；

# 586. Customer Placing the Largest Number of Orders

Easy

SQL Schema

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'.

Follow up: What if more than one customer have the largest number of orders, can you find all the customer\_number in this case?

**SELECT customer\_number**

**FROM orders**

**GROUP BY customer\_number**

**ORDER BY Count(order\_number) DESC**

**LIMIT 1;**

follow up question:

**SELECT customer\_number**

**FROM orders**

**GROUP BY customer\_number**

**HAVING Count(order\_number) = (SELECT Max(numoforder)**

**FROM (SELECT customer\_number,**

**Count(order\_number) AS numOfOrder**

**FROM orders**

**GROUP BY customer\_number) AS a)**

# 595. Big Countries

Easy

SQL Schema

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 / 1000000 ) > 3**

**OR ( population / 1000000 ) > 25;**

# 596. Classes More Than 5 Students

Easy

SQL Schema

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 courses**

**GROUP BY class**

**HAVING Count(DISTINCT student) >= 5;**

# 597. Friend Requests I: Overall Acceptance Rate

Easy

SQL Schema

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

Table: friend\_request

| sender\_id | send\_to\_id |request\_date|

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

| 1 | 2 | 2016\_06-01 |

| 1 | 3 | 2016\_06-01 |

| 1 | 4 | 2016\_06-01 |

| 2 | 3 | 2016\_06-02 |

| 3 | 4 | 2016-06-09 |

Table: request\_accepted

| requester\_id | accepter\_id |accept\_date |

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

| 1 | 2 | 2016\_06-03 |

| 1 | 3 | 2016-06-08 |

| 2 | 3 | 2016-06-08 |

| 3 | 4 | 2016-06-09 |

| 3 | 4 | 2016-06-10 |

Write a query to find the overall acceptance rate of requests rounded to 2 decimals, which is the number of acceptance divide the number of requests.

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

|accept\_rate|

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

| 0.80|

Note:

The accepted requests are not necessarily from the table friend\_request. In this case, you just need to simply count the total accepted requests (no matter whether they are in the original requests), and divide it by the number of requests to get the acceptance rate.

It is possible that a sender sends multiple requests to the same receiver, and a request could be accepted more than once. In this case, the ‘duplicated’ requests or acceptances are only counted once.

If there is no requests at all, you should return 0.00 as the accept\_rate.

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

Follow-up:

Can you write a query to return the accept rate but for every month?

How about the cumulative accept rate for every day?

**SELECT Round(Ifnull((SELECT Count(\*)**

**FROM (SELECT DISTINCT requester\_id,**

**accepter\_id**

**FROM request\_accepted) AS b) / (SELECT Count(\*)**

**FROM**

**(SELECT DISTINCT sender\_id,**

**send\_to\_id**

**FROM friend\_request) AS a), 0), 2) AS 'accept\_rate';**

follow up question:

1.

**SELECT r.month AS month,**

**Round(Ifnull(ct\_accept / ct\_request, 0), 2) AS accept\_rate**

**FROM (SELECT Month(request\_date) AS month,**

**Count(DISTINCT sender\_id, send\_to\_id) AS ct\_request**

**FROM friend\_request**

**GROUP BY Month(request\_date)) r**

**LEFT JOIN (SELECT Month(accept\_date) AS month,**

**Count(DISTINCT requester\_id, accepter\_id) AS ct\_accept**

**FROM request\_accepted**

**GROUP BY Month(accept\_date)) a**

**ON r.month = a.month**

**GROUP BY r.month**

2.

**SELECT a.date,**

**Ifnull(Round(a.ct\_accept / r.ct\_request, 2), 0) accept\_rate**

**FROM (SELECT Date\_format(accept\_date, '%Y-%m') date,**

**Count(DISTINCT requester\_id, accepter\_id) ct\_accept**

**FROM request\_accepted**

**GROUP BY Date\_format(accept\_date, '%Y-%m')) a**

**JOIN (SELECT Date\_format(request\_date, '%Y-%m') date,**

**Count(DISTINCT sender\_id, send\_to\_id) ct\_request**

**FROM friend\_request**

**GROUP BY Date\_format(request\_date, '%Y-%m')) r**

**ON a.date = r.date**

# 603. Consecutive Available Seats

Easy

SQL Schema

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**

**where abs(a.seat\_id - b.seat\_id)=1**

**AND a.free =1**

**AND b.free=1**

**ORDER BY a.seat\_id ASC;**

# 607. Sales Person

Easy

SQL Schema

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 sa.name**

**FROM salesperson sa**

**WHERE sa.sales\_id NOT IN (SELECT DISTINCT s.sales\_id**

**FROM salesperson s**

**LEFT JOIN orders o**

**ON s.sales\_id = o.sales\_id**

**LEFT JOIN company c**

**ON c.com\_id = o.com\_id**

**WHERE c.name = 'RED');**

# 610. Triangle Judgement

Easy

SQL Schema

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 |

**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;**

# 613. Shortest Distance in a Line

Easy

SQL Schema

Table point holds the x coordinate of some points on x-axis in a plane, which are all integers.

Write a query to find the shortest distance between two points in these points.

| x |

|-----|

| -1 |

| 0 |

| 2 |

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

| shortest|

|---------|

| 1 |

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

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

**SELECT Min(d) AS 'shortest'**

**FROM (SELECT Abs(a.x - b.x) AS d**

**FROM point a**

**JOIN point b**

**ON a.x != b.x) AS c;**

follow up:

**SELECT Min(b.x - a.x) AS shortest**

**FROM point a,**

**point b**

**WHERE b.x > a.x**

**LIMIT 1**

# 619. Biggest Single Number

Easy

SQL Schema

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 Ifnull((SELECT Max(n)**

**FROM (SELECT DISTINCT num AS n**

**FROM my\_numbers**

**GROUP BY num**

**HAVING Count(num) = 1) AS a), NULL) AS 'num';**

# 620. Not Boring Movies

Easy

SQL Schema

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 description != 'boring'**

**AND MOD(id, 2) = 1**

**ORDER BY id DESC;**

# 627. Swap Salary

Easy

SQL Schema

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;**

# 1050. Actors and Directors Who Cooperated At Least Three Times

Easy

SQL Schema

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(\*) >= 3;**

# 1068. Product Sales Analysis I

Easy

SQL Schema

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 p.product\_name,**

**s.year,**

**s.price**

**FROM product p**

**RIGHT JOIN sales s**

**ON p.product\_id = s.product\_id;**

# 1069. Product Sales Analysis II

Easy

SQL Schema

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 s.product\_id,**

**Sum(quantity) AS 'total\_quantity'**

**FROM sales s**

**LEFT JOIN product p**

**ON s.product\_id = p.product\_id**

**GROUP BY product\_id**

**ORDER BY Sum(quantity) DESC;**

# 1075. Project Employees I

Easy

SQL Schema

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(Sum(experience\_years) / Count(p.employee\_id), 2) AS 'average\_years'**

**FROM project p**

**LEFT JOIN employee e**

**ON p.employee\_id = e.employee\_id**

**GROUP BY p.project\_id**

**ORDER BY p.project\_id ASC;**

# 1076. Project Employees II

Easy

SQL Schema

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 a AS 'project\_id'**

**FROM (SELECT p.project\_id a,**

**Count(DISTINCT p.employee\_id) b**

**FROM project p**

**LEFT JOIN employee e**

**ON p.employee\_id = e.employee\_id**

**GROUP BY p.project\_id**

**ORDER BY b DESC**

**LIMIT 1) n;**

# 1082. Sales Analysis I

Easy

SQL Schema

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 Sum(price) DESC**

**LIMIT 1)**

# 1083. Sales Analysis II

Easy

SQL Schema

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 s.buyer\_id**

**FROM sales s**

**LEFT JOIN product p**

**ON s.product\_id = p.product\_id**

**WHERE p.product\_name = 'S8'**

**AND s.buyer\_id NOT IN (SELECT s.buyer\_id**

**FROM sales s**

**LEFT JOIN product p**

**ON s.product\_id = p.product\_id**

**WHERE p.product\_name = 'iPhone')**

# 1084. Sales Analysis III

Easy

SQL Schema

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 DISTINCT s.product\_id,**

**p.product\_name**

**FROM sales s**

**LEFT JOIN product p**

**ON s.product\_id = p.product\_id**

**WHERE s.sale\_date >= '2019-01-01'**

**AND s.sale\_date <= '2019-03-31'**

**AND p.product\_id NOT IN (SELECT DISTINCT s.product\_id**

**FROM sales s**

**LEFT JOIN product p**

**ON s.product\_id = p.product\_id**

**WHERE s.sale\_date < '2019-01-01'**

**OR s.sale\_date > '2019-03-31')**

# 1113. Reported Posts

Easy

SQL Schema

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 report\_reason,**

**Count(DISTINCT post\_id) report\_count**

**FROM (SELECT post\_id,**

**extra**

**FROM actions**

**WHERE action\_date = Date\_sub('2019-07-05', INTERVAL 1 day)**

**AND action = 'report') AS tmp**

**GROUP BY extra**

# 1141. User Activity for the Past 30 Days I

Easy

SQL Schema

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.

**SELECT activity\_date day,**

**Count(DISTINCT user\_id) AS active\_users**

**FROM activity**

**WHERE activity\_date BETWEEN Date\_sub('2019-07-27', INTERVAL 30 day) AND Date(**

**'2019-7-27')**

**GROUP BY activity\_date**

# 1142. User Activity for the Past 30 Days II

Easy

SQL Schema

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 Round(( Count(DISTINCT session\_id) / Count(DISTINCT user\_id) ), 2) AS**

**average\_sessions\_per\_user**

**FROM activity**

**WHERE activity\_date BETWEEN Date\_sub('2019-07-27', INTERVAL 30 day) AND Date(**

**'2019-7-27')**

# 1148. Article Views I

Easy

SQL Schema

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 id**

**FROM views**

**WHERE author\_id = viewer\_id**

**ORDER BY author\_id ASC**

# 1173. Immediate Food Delivery I

Easy

SQL Schema

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.

**SELECT Round((SELECT Count(\*)**

**FROM delivery**

**WHERE order\_date = customer\_pref\_delivery\_date) /**

**(SELECT Count(\*)**

**FROM delivery) \***

**100, 2) immediate\_percentage;**

# 1179. Reformat Department Table

Easy

SQL Schema

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(CASE `month`**

**WHEN 'Jan' THEN revenue**

**ELSE NULL**

**end) AS Jan\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Feb' THEN revenue**

**ELSE NULL**

**end) AS Feb\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Mar' THEN revenue**

**ELSE NULL**

**end) AS Mar\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Apr' THEN revenue**

**ELSE NULL**

**end) AS Apr\_Revenue,**

**Sum(CASE `month`**

**WHEN 'May' THEN revenue**

**ELSE NULL**

**end) AS May\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Jun' THEN revenue**

**ELSE NULL**

**end) AS Jun\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Jul' THEN revenue**

**ELSE NULL**

**end) AS Jul\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Aug' THEN revenue**

**ELSE NULL**

**end) AS Aug\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Sep' THEN revenue**

**ELSE NULL**

**end) AS Sep\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Oct' THEN revenue**

**ELSE NULL**

**end) AS Oct\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Nov' THEN revenue**

**ELSE NULL**

**end) AS Nov\_Revenue,**

**Sum(CASE `month`**

**WHEN 'Dec' THEN revenue**

**ELSE NULL**

**end) AS Dec\_Revenue**

**FROM department**

**GROUP BY id**

# 1211. Queries Quality and Percentage

Easy

SQL Schema

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) quality,**

**Round(( Sum(IF(rating < 3, 1, 0)) / Count(\*) ) \* 100, 2)**

**poor\_query\_percentage**

**FROM queries**

**GROUP BY query\_name;**

# 1241. Number of Comments per Post

Easy

SQL Schema

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 post\_id,**

**Count(DISTINCT S2.sub\_id) AS number\_of\_comments**

**FROM (SELECT DISTINCT sub\_id AS post\_id**

**FROM submissions**

**WHERE parent\_id IS NULL) S1**

**LEFT JOIN submissions S2**

**ON S1.post\_id = S2.parent\_id**

**GROUP BY S1.post\_id**

# 1251. Average Selling Price

Easy

SQL Schema

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(a.units \* b.price) / Sum(a.units), 2) AS average\_price**

**FROM unitssold a**

**JOIN prices b**

**ON ( a.product\_id = b.product\_id**

**AND a.purchase\_date >= b.start\_date**

**AND a.purchase\_date <= b.end\_date )**

**GROUP BY product\_id**

# 1280. Students and Examinations

Easy

SQL Schema

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 s.student\_id,**

**s.student\_name,**

**s.subject\_name,**

**Count(e.student\_id) attended\_exams**

**FROM examinations e**

**RIGHT JOIN (SELECT \***

**FROM students**

**CROSS JOIN subjects) s**

**ON e.student\_id = s.student\_id**

**AND e.subject\_name = s.subject\_name**

**GROUP BY s.student\_id,**

**s.subject\_name**

**ORDER BY s.student\_id,**

**s.subject\_name**

# 1294. Weather Type in Each Country

Easy

SQL Schema

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 DISTINCT country\_name,**

**CASE**

**WHEN Avg(weather\_state) <= 15 THEN 'Cold'**

**WHEN Avg(weather\_state) >= 25 THEN 'Hot'**

**ELSE 'Warm'**

**end weather\_type**

**FROM weather w**

**LEFT JOIN countries c**

**ON c.country\_id = w.country\_id**

**WHERE day BETWEEN Date('2019-11-01') AND Date('2019-11-30')**

**GROUP BY w.country\_id**

# 1303. Find the Team Size

Easy

SQL Schema

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 a.employee\_id,**

**Count(\*) team\_size**

**FROM employee a**

**LEFT JOIN employee b**

**ON a.team\_id = b.team\_id**

**GROUP BY a.employee\_id;**

# 1322. Ads Performance

Easy

SQL Schema

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:

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(Sum(action = 'Clicked') / ( Sum(action = 'Clicked')**

**+ Sum(action = 'Viewed') ) \* 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

Easy

SQL Schema

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) unit**

**FROM products p**

**LEFT JOIN orders o**

**ON ( p.product\_id = o.product\_id**

**AND order\_date >= Date('2020-02-01')**

**AND order\_date <= Date('2020-02-29') )**

**GROUP BY p.product\_id**

**ORDER BY Sum(unit) DESC**

**LIMIT 2;**

# 1350. Students With Invalid Departments

Easy

SQL Schema

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 exists.

Return the result table in any order.

The query result format is in the following example:

Departments table:

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

| id | name |

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

| 1 | Electrical Engineering |

| 7 | Computer Engineering |

| 13 | Bussiness Administration |

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

Students table:

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

| id | name | department\_id |

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

| 23 | Alice | 1 |

| 1 | Bob | 7 |

| 5 | Jennifer | 13 |

| 2 | John | 14 |

| 4 | Jasmine | 77 |

| 3 | Steve | 74 |

| 6 | Luis | 1 |

| 8 | Jonathan | 7 |

| 7 | Daiana | 33 |

| 11 | Madelynn | 1 |

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

Result table:

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

| id | name |

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

| 2 | John |

| 7 | Daiana |

| 4 | Jasmine |

| 3 | Steve |

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

John, Daiana, Steve and Jasmine are enrolled in departments 14, 33, 74 and 77 respectively. department 14, 33, 74 and 77 doesn't exist in the Departments table.

**SELECT s.id,**

**s.name**

**FROM students s**

**LEFT JOIN departments d**

**ON s.department\_id = d.id**

**WHERE d.name IS NULL;**

# 1378. Replace Employee ID With The Unique Identifier

Easy

SQL Schema

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

Easy

SQL Schema

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 DISTINCT u.name,**

**Ifnull(Sum(distance), 0) travelled\_distance**

**FROM users u**

**LEFT JOIN rides r**

**ON u.id = r.user\_id**

**GROUP BY u.id**

**ORDER BY Sum(distance) DESC,**

**name ASC**

# 1435. Create a Session Bar Chart

Easy

SQL Schema

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(session\_id) AS total**

**FROM sessions**

**WHERE duration < 300**

**UNION**

**SELECT '[5-10>' AS bin,**

**Count(session\_id) AS total**

**FROM sessions**

**WHERE duration > 300**

**AND duration < 600**

**UNION**

**SELECT '[10-15>' AS bin,**

**Count(session\_id) AS total**

**FROM sessions**

**WHERE duration > 600**

**AND duration < 900**

**UNION**

**SELECT '15 or more' AS bin,**

**Count(session\_id) AS total**

**FROM sessions**

**WHERE duration > 900;**

# 1484. Group Sold Products By The Date

Easy

SQL Schema

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)AS products**

**FROM activities**

**GROUP BY sell\_date**

**ORDER BY sell\_date;**

# 1495. Friendly Movies Streamed Last Month

Easy

SQL Schema

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 c.title**

**FROM tvprogram t**

**JOIN content c**

**ON t.content\_id = c.content\_id**

**WHERE kids\_content = 'Y'**

**AND program\_date BETWEEN ( '2020-06-01 00:00' ) AND Date(**

**'2020-06-30 23:59')**

**AND content\_type = 'Movies';**

# 1511. Customer Order Frequency

Easy

SQL Schema

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 temp.customer\_id,**

**temp.name**

**FROM (SELECT c.customer\_id,**

**c.name,**

**Month(o.order\_date) AS mnth**

**FROM orders o**

**LEFT JOIN product p**

**ON o.product\_id = p.product\_id**

**LEFT JOIN customers c**

**ON c.customer\_id = o.customer\_id**

**WHERE o.order\_date BETWEEN '2020-06-01' AND '2020-07-31'**

**GROUP BY o.customer\_id,**

**Month(o.order\_date)**

**HAVING Sum(p.price \* o.quantity) >= 100) temp**

**GROUP BY temp.customer\_id**

**HAVING Count(mnth) = 2**

# 1517. Find Users With Valid E-Mails

Easy

SQL Schema

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 \***

**FROM users**

**WHERE mail REGEXP '^[a-z][[:alnum:]|\_|.|-]+@leetcode\.com$';**