|  |  |
| --- | --- |
| Column Name | Type |
| name | varchar |
| continent | varchar |
| area | int |
| population | int |
| gdp | int |

name is the primary key column for this table.

Each row of this table gives information about the name of a country, the continent to which it belongs, its area, the population, and its GDP value.

A country is big if:

* it has an area of at least three million (i.e., 3000000 km2), or
* it has a population of at least twenty-ﬁve million (i.e., 25000000).

Write an SQL query to report the name, population, and area of the big countries. Return the result table in any order.

The query result format is in the following example.

Input: World table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| name | continent | Area | population | gdp |
| Afghanistan | Asia | 652230 | 25500100 | 20343000000 |
| Albania | Europe | 28748 | 2831741 | 12960000000 |
| Algeria | Africa | 2381741 | 37100000 | 188681000000 |
| Andorra | Europe | 468 | 78115 | 3712000000 |
| Angola | Africa | 1246700 | 20609294 | 100990000000 |

Output:

|  |  |  |
| --- | --- | --- |
| name | population | Area |
| Afghanistan | 25500100 | 652230 |
| Algeria | 37100000 | 2381741 |

|  |
| --- |
| SELECT |
|  | name, population, area |
|  | FROM |
|  | world |
|  | WHERE |
|  | area > 3000000 OR population > 25000000 |
|  | ; |

Table: Customer

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |
| referee\_id | int |

id is the primary key column for this table.

Each row of this table indicates the id of a customer, their name, and the id of the customer who referred them.

Write an SQL query to report the names of the customer that are not referred by the customer with id

= 2.

Return the result table in any order.

The query result format is in the following example.

Input: Customer table:

|  |  |  |
| --- | --- | --- |
| id | name | referee\_id |
| 1 | Will | null |
| 2 | Jane | null |
| 3 | Alex | 2 |
| 4 | Bill | null |
| 5 | Zack | 1 |
| 6 | Mark | 2 |

Output:

|  |
| --- |
| name |
| Will |
| Jane |
| Bill |
| Zack |

|  |
| --- |
| select name |
|  | from customer |
|  | where referee\_id != 2 or referee\_id is NULL |

Table: Customers

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |

id is the primary key column for this table.

Each row of this table indicates the ID and name of a customer.

Table: Orders

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| customerId | int |

id is the primary key column for this table.

customerId is a foreign key of the ID from the Customers table.

Each row of this table indicates the ID of an order and the ID of the customer who ordered it.

Write an SQL query to report all customers who never order anything. Return the result table in any order.

The query result format is in the following example.

Input: Customers table:

|  |  |
| --- | --- |
| id | name |
| 1 | Joe |
| 2 | Henry |
| 3 | Sam |
| 4 | Max |

Orders table:

|  |  |
| --- | --- |
| id | CustomerId |
| 1 | 3 |
| 2 | 1 |

Output:

|  |
| --- |
| Customers |
| Henry |
| Max |

SELECT Name as Customers from Customers LEFT JOIN Orders ON Customers.Id = Orders.CustomerId WHERE Orders.CustomerId IS NULL;

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.

Input:

Employee Table:

|  |  |
| --- | --- |
| employee\_id | team\_id |
| 1 | 8 |
| 2 | 8 |
| 3 | 8 |
| 4 | 7 |
| 5 | 9 |
| 6 | 9 |

Output:

|  |  |
| --- | --- |
| employee\_id | team\_size |
| 1 | 3 |
| 2 | 3 |
| 3 | 3 |
| 4 | 1 |
| 5 | 2 |
| 6 | 2 |

Explanation:

Employees with Id 1,2,3 are part of a team with team\_id = 8. Employee 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 EMPLOYEE\_ID, COUNT(TEAM\_ID) OVER (PARTITION BY TEAM\_ID) TEAM\_SIZE FROM EMPLOYEE;

|  |
| --- |
|  |
|  |

# Q55

Table Person:

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |
| phone\_number | varchar |

id is the primary key for this table.

Each row of this table contains the name of a person and their phone number.

Phone number will be in the form 'xxx-yyyyyyy' where xxx is the country code (3 characters) and yyyyyyy is the phone number (7 characters) where x and y are digits. Both can contain leading zeros.

Table Country:

|  |  |
| --- | --- |
| Column Name | Type |
| name | varchar |
| country\_code | varchar |

country\_code is the primary key for this table.

Each row of this table contains the country name and its code. country\_code will be in the form 'xxx' where x is digits.

Table Calls:

|  |  |
| --- | --- |
| Column Name | Type |
| caller\_id | int |
| callee\_id | int |
| duration | int |

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

Each row of this table contains the caller id, caller id and the duration of the call in minutes. caller\_id

!= callee\_id

A telecommunications company wants to invest in new countries. The company intends to invest in the countries where the average call duration of the calls in this country is strictly greater than the global average call duration.

Write an SQL query to ﬁnd the countries where this company can invest. Return the result table in any order.

The query result format is in the following example.

Input: Person table:

|  |  |  |
| --- | --- | --- |
| id | name | phone\_number |
| 3 | Jonathan | 051-1234567 |
| 12 | Elvis | 051-7654321 |
| 1 | Moncef | 212-1234567 |
| 2 | Maroua | 212-6523651 |
| 7 | Meir | 972-1234567 |
| 9 | Rachel | 972-0011100 |

Country table:

|  |  |
| --- | --- |
| name | country\_code |
| Peru | 51 |
| Israel | 972 |
| Morocco | 212 |
| Germany | 49 |
| Ethiopia | 251 |

Calls table:

|  |  |  |
| --- | --- | --- |
| caller\_id | callee\_id | duration |
| 1 | 9 | 33 |
| 2 | 9 | 4 |
| 1 | 2 | 59 |
| 3 | 12 | 102 |
| 3 | 12 | 330 |
| 12 | 3 | 5 |
| 7 | 9 | 13 |
| 7 | 1 | 3 |
| 9 | 7 | 1 |
| 1 | 7 | 7 |

Output:

Peru

country

Explanation:

The average call duration for Peru is (102 + 102 + 330 + 330 + 5 + 5) / 6 = 145.666667

The average call duration for Israel is (33 + 4 + 13 + 13 + 3 + 1 + 1 + 7) / 8 = 9.37500

The average call duration for Morocco is (33 + 4 + 59 + 59 + 3 + 7) / 6 = 27.5000

Global call duration average = (2 \* (33 + 4 + 59 + 102 + 330 + 5 + 13 + 3 + 1 + 7)) / 20 = 55.70000 Since Peru is the only country where the average call duration is greater than the global average, it is the only recommended country.

**select** **c**.name **as** country

**from** Person p

**inner** **join** Country **c**

**on** **left** (p.phone\_number,3) **=** **c**.country\_code

**inner** **join** (**select** caller\_id **as** id, duration

**from** Calls

**union** **all**

**select** callee\_id **as** id, duration

**from** Calls) phn

**on** p.id **=** phn.id

**group** **by** country

**having** **avg**(duration) **>** (**select** **avg**(duration) **from** Calls)

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 games.

Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on someday using some device.

Write an SQL query to report the device that is ﬁrst logged in for each player. Return the result table in any order.

The query result format is in the following example.

Input: 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 |

Output:

|  |  |
| --- | --- |
| player\_id | device\_id |
| 1 | 2 |
| 2 | 3 |
| 3 | 1 |

|  |
| --- |
| select player\_id, min(event\_date) as first\_login |
|  | from Activity |
|  | group by player\_id |

Table: Orders

|  |  |
| --- | --- |
| Column Name | Type |
| order\_number | Int |
| customer\_number | Int |

order\_number is the primary key for this table.

This table contains information about the order ID and the customer ID.

Write an SQL query to ﬁnd the customer\_number for the customer who has placed the largest number of orders.

The test cases are generated so that exactly one customer will have placed more orders than any other customer.

The query result format is in the following example.

Input: Orders table:

|  |  |
| --- | --- |
| order\_number | ustomer\_numbe |
| 1 | 1 |
| 2 | 2 |
| 3 | 3 |
| 4 | 3 |

Output:

3

customer\_number

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 has 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(customer\_number) DESC |
|  | LIMIT 1; |

Table: Cinema

|  |  |
| --- | --- |
| Column Name | Type |
| seat\_id | Int |
| Free | Bool |

seat\_id is an auto-increment primary key column for this table.

Each row of this table indicates whether the ith seat is free or not. 1 means free while 0 means occupied.

Write an SQL query to report all the consecutive available seats in the cinema. Return the result table ordered by seat\_id in ascending order.

The test cases are generated so that more than two seats are consecutively available. The query result format is in the following example.

Input: Cinema table:

|  |  |
| --- | --- |
| seat\_id | Free |
| 1 | 1 |
| 2 | 0 |
| 3 | 1 |
| 4 | 1 |
| 5 | 1 |

Output:

|  |
| --- |
| seat\_id |
| 3 |
| 4 |
| 5 |

SELECT

DISTINCT(a.seat\_id)

FROM cinema a

INNER JOIN cinema b

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

WHERE a.free = 1 and b.free = 1

ORDER BY a.seat\_id

Table: SalesPerson

|  |  |
| --- | --- |
| Column Name | Type |
| sales\_id | Int |
| Name | Varchar |
| Salary | Int |
| commission\_rate | Int |
| hire\_date | Date |

sales\_id is the primary key column for this table.

Each row of this table indicates the name and the ID of a salesperson alongside their salary, commission rate, and hire date.

Table: Company

|  |  |
| --- | --- |
| Column Name | Type |
| com\_id | Int |
| Name | Varchar |
| City | Varchar |

com\_id is the primary key column for this table.

Each row of this table indicates the name and the ID of a company and the city in which the company is located.

Table: Orders

|  |  |
| --- | --- |
| Column Name | Type |
| order\_id | Int |
| order\_date | Date |
| com\_id | Int |
| sales\_id | Int |
| Amount | Int |

order\_id is the primary key column for this table.

com\_id is a foreign key to com\_id from the Company table. sales\_id is a foreign key to sales\_id from the SalesPerson table.

Each row of this table contains information about one order. This includes the ID of the company, the ID of the salesperson, the date of the order, and the amount paid.

Write an SQL query to report the names of all the salespersons who did not have any orders related to the company with the name "RED".

Return the result table in any order.

The query result format is in the following example.

Input:

SalesPerson table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sales\_id | name | salary | commission\_rate | hire\_date |
| 1 | John | 100000 | 6 | 4/1/2006 |
| 2 | Amy | 12000 | 5 | 5/1/2010 |
| 3 | Mark | 65000 | 12 | 12/25/2008 |
| 4 | Pam | 25000 | 25 | 1/1/2005 |
| 5 | Alex | 5000 | 10 | 2/3/2007 |

Company table:

|  |  |  |
| --- | --- | --- |
| com\_id | name | city |
| 1 | RED | Boston |
| 2 | ORANGE | New York |
| 3 | YELLOW | Boston |
| 4 | GREEN | Austin |

Orders table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| order\_id | order\_date | com\_id | sales\_id | amount |
| 1 | 1/1/2014 | 3 | 4 | 10000 |
| 2 | 2/1/2014 | 4 | 5 | 5000 |
| 3 | 3/1/2014 | 1 | 1 | 50000 |
| 4 | 4/1/2014 | 1 | 4 | 25000 |

Output:

|  |
| --- |
| name |
| Amy |
|  |
| Mark |
| Alex |

Explanation:

According to orders 3 and 4 in the Orders table, it is easy to tell that only salesperson John and Pam have sales to company RED, so we report all the other names in the table salesperson.

SELECT name

FROM salesperson

WHERE sales\_id

NOT IN (

SELECT s.sales\_id FROM orders o

INNER JOIN salesperson s ON o.sales\_id = s.sales\_id

INNER JOIN company c ON o.com\_id = c.com\_id

WHERE c.name = 'RED'

);

Table: Triangle

|  |  |
| --- | --- |
| Column Name | Type |
| x | Int |
| y | int |
| z | int |

(x, y, z) is the primary key column for this table.

Each row of this table contains the lengths of three line segments.

Write an SQL query to report for every three line segments whether they can form a triangle. Return the result table in any order.

The query result format is in the following example.

Input: Triangle table:

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

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| x | y | z | triangle |
| 13 | 15 | 30 | No |
| 10 | 20 | 15 | Yes |

SELECT

x,

y,

z,

IF(x + y > z AND y + z > x AND z + x > y, 'Yes', 'No') triangle

FROM

triangle;

Table: Point

|  |  |
| --- | --- |
| Column Name | Type |
| x | int |

x is the primary key column for this table.

Each row of this table indicates the position of a point on the X-axis.

Write an SQL query to report the shortest distance between any two points from the Point table. The query result format is in the following example.

Input: Point table:

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

Output:

1

shortest

Explanation:

The shortest distance is between points -1 and 0 which is |(-1) - 0| = 1.

Follow up: How could you optimise your query if the Point table is ordered in ascending order?

select

min(abs(p1.x - p2.x)) shortest

from point p1, point p2

where p1.x <> p2.x

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 has cooperated with the director at least three times.

Return the result table in any order.

The query result format is in the following example.

Input:

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 |

Output:

|  |  |
| --- | --- |
| actor\_id | director\_id |
| 1 | 1 |

Explanation:

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 |

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 the Product table.

Each row of this table shows a sale on the product product\_id in a certain year. 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.

Each row of this table indicates the product name of each product.

Write an SQL query that reports the product\_name, year, and price for each sale\_id in the Sales table. Return the resulting table in any order.

The query result format is in the following example.

Input: 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 |

Output:

|  |  |  |
| --- | --- | --- |
| product\_name | year | price |
| Nokia | 2008 | 5000 |
| Nokia | 2009 | 5000 |
| Apple | 2011 | 9000 |

Explanation:

From sale\_id = 1, we can conclude that Nokia was sold for 5000 in the year 2008. From sale\_id = 2, we can conclude that Nokia was sold for 5000 in the year 2009. From sale\_id = 7, we can conclude that Apple was sold for 9000 in the year 2011.

|  |
| --- |
| select p.product\_name, s.year, s.price |
|  | from Product p |
|  | join Sales s |
|  | on s.product\_id = p.product\_id |

# Q64.

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 the Employee table.

Each row of this table indicates that the employee with employee\_id is working on the project with project\_id.

Table: Employee

|  |  |
| --- | --- |
| Column Name | Type |
| employee\_id | int |
| name | varchar |
| experience\_years | int |

employee\_id is the primary key of this table.

Each row of this table contains information about one employee.

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

Return the result table in any order.

The query result format is in the following example.

Input: 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 |

Output:

|  |  |
| --- | --- |
| project\_id | average\_years |
| 1 | 2 |
| 2 | 2.5 |

Explanation:

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 project\_id , round(avg(experience\_years), 2) as average\_years |
|  | from project as p |
|  | left join employee as e |
|  | on p.employee\_id = e.employee\_id |
|  | group by project\_id |

# Q65.

Table: Product

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| product\_name | varchar |
| unit\_price | int |

product\_id is the primary key of this table.

Each row of this table indicates the name and the price of each product. 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 the Product table.

Each row of this table contains some information about one sale.

Write an SQL query that reports the best seller by total sales price, If there is a tie, report them all. Return the result table in any order.

The query result format is in the following example.

Input: 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 |

Output:

|  |
| --- |
| seller\_id |
| 1 |
| 3 |

Explanation: Both sellers with id 1 and 3 sold products with the most total price of 2800.

**select** **distinct** seller\_id

**from** Sales

**group** **by** seller\_id

**having** **sum**(price) **=** (

**select** **sum**(price) **as** max\_price

**from** Sales

**group** **by** seller\_id

**order** **by** max\_price **desc**

**limit** 1)

# Q66.

Table: Product

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| product\_name | varchar |
| unit\_price | int |

product\_id is the primary key of this table.

Each row of this table indicates the name and the price of each product. 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 the Product table.

Each row of this table contains some information about one sale.

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.

Return the result table in any order.

The query result format is in the following example.

Input: 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 |

Output:

1

buyer\_id

Explanation:

The buyer with id 1 bought an S8 but did not buy an iPhone. The buyer with id 3 bought both. Orders table:

|  |  |  |  |
| --- | --- | --- | --- |
| order\_id | book\_id | quantity | dispatch\_date |
| 1 | 1 | 2 | 2018-07-26 |
| 2 | 1 | 1 | 2018-11-05 |
| 3 | 3 | 8 | 2019-06-11 |
| 4 | 4 | 6 | 2019-06-05 |
| 5 | 4 | 5 | 2019-06-20 |
| 6 | 5 | 9 | 2009-02-02 |
| 7 | 5 | 8 | 2010-04-13 |

Output:

|  |  |
| --- | --- |
| book\_id | name |
| 1 | "Kalila And Demna" |
| 2 | "28 Letters" |
| 5 | "The Hunger Games" |

|  |
| --- |
| select distinct buyer\_id from Sales s |
|  | join Product p |
|  | on p.product\_id = s.product\_id |
|  | where p.product\_name = 'S8' |
|  | and buyer\_id not in |
|  | ( |
|  | select buyer\_id from Sales s |
|  | join Product p on p.product\_id = s.product\_id |
|  | where p.product\_name = 'iPhone' |
|  | ) |

# Q67.

Table: Customer

|  |  |
| --- | --- |
| Column Name | Type |
| customer\_id | int |
| name | varchar |
| visited\_on | date |
| amount | int |

(customer\_id, visited\_on) is the primary key for this table.

This table contains data about customer transactions in a restaurant.

visited\_on is the date on which the customer with ID (customer\_id) has visited the restaurant. amount is the total paid by a customer.

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

Write an SQL query to compute the moving average of how much the customer paid in a seven days window (i.e., current day + 6 days before). average\_amount should be rounded to two decimal places. Return result table ordered by visited\_on in ascending order.

The query result format is in the following example.

Input: Customer table:

|  |  |  |  |
| --- | --- | --- | --- |
| customer\_id | name | visited\_on | amount |
| 1 | Jhon | 2019-01-01 | 100 |
| 2 | Daniel | 2019-01-02 | 110 |
| 3 | Jade | 2019-01-03 | 120 |
| 4 | Khaled | 2019-01-04 | 130 |
| 5 | Winston | 2019-01-05 | 110 |
| 6 | Elvis | 2019-01-06 | 140 |
| 7 | Anna | 2019-01-07 | 150 |
| 8 | Maria | 2019-01-08 | 80 |
| 9 | Jaze | 2019-01-09 | 110 |
| 1 | Jhon | 2019-01-10 | 130 |
| 3 | Jade | 2019-01-10 | 150 |

Output:

|  |  |  |
| --- | --- | --- |
| visited\_on | amount | average\_amount |
| 2019-01-07 | 860 | 122.86 |

|  |  |  |
| --- | --- | --- |
| 2019-01-08 | 840 | 120 |
| 2019-01-09 | 840 | 120 |
| 2019-01-10 | 1000 | 142.86 |

Explanation:

1st moving average from 2019-01-01 to 2019-01-07 has an average\_amount of (100 + 110 + 120 + 130 + 110 + 140 + 150)/7 = 122.86

2nd moving average from 2019-01-02 to 2019-01-08 has an average\_amount of (110 + 120 + 130 + 110 + 140 + 150 + 80)/7 = 120

3rd moving average from 2019-01-03 to 2019-01-09 has an average\_amount of (120 + 130 + 110 + 140 + 150 + 80 + 110)/7 = 120

4th moving average from 2019-01-04 to 2019-01-10 has an average\_amount of (130 + 110 + 140 + 150 + 80 + 110 + 130 + 150)/7 = 142.86

**select** c1.visited\_on, **sum**(c2.amount) **as** amount,

round(**avg**(c2.amount), 2) **as** average\_amount

**from** (**select** visited\_on, **sum**(amount) **as** amount

**from** customer **group** **by** visited\_on) c1

**join** (**select** visited\_on, **sum**(amount) **as** amount

**from** customer **group** **by** visited\_on) c2

**on** datediff(c1.visited\_on, c2.visited\_on) **between** 0 **and** 6

**group** **by** c1.visited\_on

**having** **count**(c2.amount) **=** 7

# Q68.

Table: Scores

|  |  |
| --- | --- |
| Column Name | Type |
| player\_name | Varchar |
| gender | Varchar |
| day | Date |
| score\_points | Int |

(gender, day) is the primary key for this table.

A competition is held between the female team and the male team.

Each row of this table indicates that a player\_name and with gender has scored score\_point in someday.

Gender is 'F' if the player is in the female team and 'M' if the player is in the male team.

Write an SQL query to ﬁnd the total score for each gender on each day. Return the result table ordered by gender and day in ascending order. The query result format is in the following example.

Input: Scores table:

|  |  |  |  |
| --- | --- | --- | --- |
| player\_name | Gender | day | score\_points |
| Aron | F | 2020-01-01 | 17 |
| Alice | F | 2020-01-07 | 23 |
| Bajrang | M | 2020-01-07 | 7 |
| Khali | M | 2019-12-25 | 11 |
| Slaman | M | 2019-12-30 | 13 |
| Joe | M | 2019-12-31 | 3 |
| Jose | M | 2019-12-18 | 2 |
| Priya | F | 2019-12-31 | 23 |
| Priyanka | F | 2019-12-30 | 17 |

Output:

|  |  |  |
| --- | --- | --- |
| gender | day | total |
| F | 2019-12-30 | 17 |
| F | 2019-12-31 | 40 |
| F | 2020-01-01 | 57 |
| F | 2020-01-07 | 80 |
| M | 2019-12-18 | 2 |
| M | 2019-12-25 | 13 |

|  |  |  |
| --- | --- | --- |
| M | 2019-12-30 | 26 |
| M | 2019-12-31 | 29 |
| M | 2020-01-07 | 36 |

Explanation:

For the female team:

The ﬁrst day is 2019-12-30, Priyanka scored 17 points and the total score for the team is 17. The second day is 2019-12-31, Priya scored 23 points and the total score for the team is 40. The third day is 2020-01-01, Aron scored 17 points and the total score for the team is 57.

The fourth day is 2020-01-07, Alice scored 23 points and the total score for the team is 80.

For the male team:

The ﬁrst day is 2019-12-18, Jose scored 2 points and the total score for the team is 2.

The second day is 2019-12-25, Khali scored 11 points and the total score for the team is 13. The third day is 2019-12-30, Slaman scored 13 points and the total score for the team is 26. The fourth day is 2019-12-31, Joe scored 3 points and the total score for the team is 29.

The ﬁfth day is 2020-01-07, Bajrang scored 7 points and the total score for the team is 36.

|  |
| --- |
| select s1.gender, s1.day, sum(s2.score\_points) as total from Scores s1, Scores s2 |
|  | where s1.gender = s2.gender and s1.day >= s2.day |
|  | group by s1.gender, s1.day |
|  | order by s1.gender, s1.day |

**Q69**.

Table: Logs

|  |  |
| --- | --- |
| Column Name | Type |
| log\_id | int |

log\_id is the primary key for this table.

Each row of this table contains the ID in a log Table.

Write an SQL query to ﬁnd the start and end number of continuous ranges in the table Logs. Return the result table ordered by start\_id.

The query result format is in the following example.

Input: Logs table:

|  |
| --- |
| log\_id |
| 1 |
| 2 |
| 3 |
| 7 |
| 8 |
| 10 |

Output:

|  |  |
| --- | --- |
| start\_id | end\_id |
| 1 | 3 |
| 7 | 8 |
| 10 | 10 |

Explanation:

The result table should contain all ranges in table Logs. From 1 to 3 is contained in the table.

From 4 to 6 is missing in the table From 7 to 8 is contained in the table. Number 9 is missing from the table. Number 10 is contained in the table.

**select** log\_start.log\_id **as** START\_ID, **min**(log\_end.log\_id) **as** END\_ID **from**

(**select** log\_id **from** logs **where** log\_id **-** 1 **not** **in** (**select** **\*** **from** Logs)) log\_start,

(**select** log\_id **from** logs **where** log\_id **+** 1 **not** **in** (**select** **\*** **from** Logs)) log\_end

**where** log\_start.log\_id **<=** log\_end.log\_id

**group** **by** log\_start.log\_id;

# Q70.

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 the 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. Return the result table ordered by student\_id and subject\_name.

The query result format is in the following example.

Input: 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 |

Output:

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

Explanation:

The result table should contain all students and all subjects.

Alice attended the Math exam 3 times, the Physics exam 2 times, and the Programming exam 1 time. Bob attended the Math exam 1 time, the Programming exam 1 time, and did not attend the Physics exam.

Alex did not attend any exams.

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

|  |
| --- |
| select a.student\_id, a.student\_name, b.subject\_name, count(c.subject\_name) as attended\_exams |
|  | from Students as a |
|  | join Subjects as b |
|  | left join Examinations as c |
|  | on a.student\_id = c.student\_id and b.subject\_name = c.subject\_name |
|  | group by a.student\_id, b.subject\_name; |

# Q71.

Table: Employees

|  |  |
| --- | --- |
| Column Name | Type |
| employee\_id | int |
| employee\_name | varchar |
| manager\_id | int |

employee\_id is the primary key for this table.

Each row of this table indicates that the employee with ID employee\_id and name employee\_name reports his work to his/her direct manager with manager\_id

The head of the company is the employee with employee\_id = 1.

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

The indirect relation between managers will not exceed three managers as the company is small. Return the result table in any order.

The query result format is in the following example.

Input: Employees table:

|  |  |  |
| --- | --- | --- |
| employee\_id | employee\_nam e | manager\_id |
| 1 | Boss | 1 |
| 3 | Alice | 3 |
| 2 | Bob | 1 |
| 4 | Daniel | 2 |
| 7 | Luis | 4 |
| 8 | Jhon | 3 |
| 9 | Angela | 8 |
| 77 | Robert | 1 |

Output:

|  |
| --- |
| employee\_id |
| 2 |
| 77 |
| 4 |
| 7 |

Explanation:

The head of the company is the employee with employee\_id 1.

The employees with employee\_id 2 and 77 report their work directly to the head of the company.

The employee with employee\_id 4 reports their work indirectly to the head of the company 4 --> 2 --> 1. The employee with employee\_id 7 reports their work indirectly to the head of the company 7 --> 4 --> 2

--> 1.

The employees with employee\_id 3, 8, and 9 do not report their work to the head of the company directly or indirectly.

**select** employee\_id **as** EMPLOYEE\_ID **from** Employees **where** manager\_id **in**

(**select** employee\_id **from** Employees **WHERE** manager\_id **in**

(**select** employee\_id **from** Employees **where** manager\_id **=**1))

**and** employee\_id **!=**1

Table: Transactions

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| country | varchar |
| state | enum |
| amount | int |
| trans\_date | date |

id is the primary key of this table.

The table has information about incoming transactions.

The state column is an enum of type ["approved", "declined"].

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

Return the result table in any order.

The query result format is in the following example.

Input: Transactions table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Id | country | state | amount | trans\_date |
| 121 | US | approved | 1000 | 2018-12-18 |
| 122 | US | declined | 2000 | 2018-12-19 |
| 123 | US | approved | 2000 | 2019-01-01 |
| 124 | DE | approved | 2000 | 2019-01-07 |

Output:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| month | country | trans\_count | approved\_cou nt | trans\_total\_a mount | roved\_total\_am |
| 2018-12 | US | 2 | 1 | 3000 | 1000 |
| 2019-01 | US | 1 | 1 | 2000 | 2000 |
| 2019-01 | DE | 1 | 1 | 2000 | 2000 |

SELECT Date\_format(trans\_date, '%Y-%m') AS month,

country,

Count(id) AS trans\_count,

Count(IF(state = 'approved', 1, NULL)) AS approved\_count,

SUM(amount) AS trans\_total\_amount,

SUM(IF(state = 'approved', amount, 0)) AS approved\_total\_amount

FROM transactions

GROUP BY Date\_format(trans\_date, '%Y-%m'),

country

ORDER BY NULL

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 the report or a type of reaction.

Table: Removals

|  |  |
| --- | --- |
| Column Name | Type |
| post\_id | int |
| remove\_date | date |

post\_id is the primary key of this table.

Each row in this table indicates that some post was removed due to being reported or as a result of an admin review.

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

The query result format is in the following example.

Input: Actions table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| user\_id | post\_id | action\_date | action | extra |
| 1 | 1 | 2019-07-01 | view | null |
| 1 | 1 | 2019-07-01 | like | null |
| 1 | 1 | 2019-07-01 | share | null |
| 2 | 2 | 2019-07-04 | view | null |
| 2 | 2 | 2019-07-04 | report | spam |
| 3 | 4 | 2019-07-04 | view | null |
| 3 | 4 | 2019-07-04 | report | spam |
| 4 | 3 | 2019-07-02 | view | null |
| 4 | 3 | 2019-07-02 | report | spam |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 2 | 2019-07-03 | view | null |
| 5 | 2 | 2019-07-03 | report | racism |
| 5 | 5 | 2019-07-03 | view | null |
| 5 | 5 | 2019-07-03 | report | racism |

Removals table:

|  |  |
| --- | --- |
| post\_id | remove\_date |
| 2 | 2019-07-20 |
| 3 | 2019-07-18 |

Output:

75

average\_daily\_percent

Explanation:

The percentage for 2019-07-04 is 50% because only one post of two spam reported posts were removed.

The percentage for 2019-07-02 is 100% because one post was reported as spam and it was removed. The other days had no spam reports so the average is (50 + 100) / 2 = 75%

Note that the output is only one number and that we do not care about the remove dates.

WITH PostRemovalRate

AS (

-- Perform a LEFT JOIN so we can calculate daily removal rate of reported spams

SELECT A.action\_date

,count(R.post\_id) / cast(count(A.post\_id) AS FLOAT) AS removal\_rate

FROM Actions AS A

LEFT JOIN Removals AS R ON A.post\_id = R.post\_id

WHERE A.action = 'report'

AND A.extra = 'spam'

GROUP BY A.action\_date

)

-- Find the average for daily percentage of posts that got removed after being reported as spam

-- Round to 2 decimal places

SELECT round(avg(removal\_rate), 2) \* 100 AS average\_daily\_percent

FROM PostRemovalRate

|  |  |
| --- | --- |
| 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 games.

Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on someday using some device.

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

The query result format is in the following example.

Input: Activity table:

|  |  |  |  |
| --- | --- | --- | --- |
| player\_id | device\_id | event\_date | games\_played |
| 1 | 2 | 2016-03-01 | 5 |
| 1 | 2 | 2016-03-02 | 6 |
| 2 | 3 | 2017-06-25 | 1 |
| 3 | 1 | 2016-03-02 | 0 |
| 3 | 4 | 2018-07-03 | 5 |

Output:

0.33

fraction

Explanation:

Only the player with id 1 logged back in after the ﬁrst day he had logged in so the answer is 1/3 = 0.33

SELECT  
round((count(distinct c.player\_id) / (select count(distinct player\_id) from activity)),2)as fraction  
FROM  
CTE c  
JOIN Activity a  
on c.player\_id = a.player\_id  
and datediff(c.event\_start\_date, a.event\_date) = -1

|  |  |
| --- | --- |
| 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 games.

Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on someday using some device.

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

The query result format is in the following example.

Input: Activity table:

|  |  |  |  |
| --- | --- | --- | --- |
| player\_id | device\_id | event\_date | games\_played |
| 1 | 2 | 2016-03-01 | 5 |
| 1 | 2 | 2016-03-02 | 6 |
| 2 | 3 | 2017-06-25 | 1 |
| 3 | 1 | 2016-03-02 | 0 |
| 3 | 4 | 2018-07-03 | 5 |

Output:

0.33

fraction

Explanation:Only the player with id 1 logged back in after the ﬁrst day he had logged in so the answer is 1/3 = 0.33

WITH CTE AS (  
SELECT  
player\_id, min(event\_date) as event\_start\_date  
from  
Activity  
group by player\_id )

SELECT  
round((count(distinct c.player\_id) / (select count(distinct player\_id) from activity)),2)as fraction  
FROM  
CTE c  
JOIN Activity a  
on c.player\_id = a.player\_id  
and datediff(c.event\_start\_date, a.event\_date) = -1

Table Salaries:

|  |  |
| --- | --- |
| Column Name | Type |
| company\_id | int |
| employee\_id | int |
| employee\_name | varchar |
| salary | int |

(company\_id, employee\_id) is the primary key for this table.

This table contains the company id, the id, the name, and the salary for an employee.

Write an SQL query to ﬁnd the salaries of the employees after applying taxes. Round the salary to the nearest integer.

The tax rate is calculated for each company based on the following criteria:

* 0% If the max salary of any employee in the company is less than $1000.
* 24% If the max salary of any employee in the company is in the range [1000, 10000] inclusive.
* 49% If the max salary of any employee in the company is greater than $10000. Return the result table in any order.

The query result format is in the following example.

Input: Salaries table:

|  |  |  |  |
| --- | --- | --- | --- |
| company\_id | employee\_id | employee\_nam e | salary |
| 1 | 1 | Tony | 2000 |
| 1 | 2 | Pronub | 21300 |
| 1 | 3 | Tyrrox | 10800 |
| 2 | 1 | Pam | 300 |
| 2 | 7 | Bassem | 450 |
| 2 | 9 | Hermione | 700 |
| 3 | 7 | Bocaben | 100 |
| 3 | 2 | Ognjen | 2200 |
| 3 | 13 | Nyan Cat | 3300 |
| 3 | 15 | Morning Cat | 7777 |

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| company\_id | employee\_id | employee\_name | salary |
| 1 | 1 | Tony | 1020 |
| 1 | 2 | Pronub | 10863 |
| 1 | 3 | Tyrrox | 5508 |
| 2 | 1 | Pam | 300 |
| 2 | 7 | Bassem | 450 |
| 2 | 9 | Hermione | 700 |
| 3 | 7 | Bocaben | 76 |
| 3 | 2 | Ognjen | 1672 |
| 3 | 13 | Nyan Cat | 2508 |
| 3 | 15 | Morning Cat | 5911 |

Explanation:

For company 1, Max salary is 21300. Employees in company 1 have taxes = 49% For company 2, Max salary is 700. Employees in company 2 have taxes = 0% For company 3, Max salary is 7777. Employees in company 3 have taxes = 24% The salary after taxes = salary - (taxes percentage / 100) \* salary

For example, Salary for Morning Cat (3, 15) after taxes = 7777 - 7777 \* (24 / 100) = 7777 - 1866.48 =

5910.52, which is rounded to 5911.

select company\_id, employee\_id, employee\_name, round(salary - salary\*tax, 0) as salary  
from  
( select \*, case when max(salary) over(partition by company\_id) < 1000 then 0 when max(salary) over(partition by company\_id) between 1000  
 and 10000 then 0.24  
 else 0.49 end as tax from Salaries  
) x

# Q77.

Table Variables:

|  |  |
| --- | --- |
| Column Name | Type |
| name | varchar |
| value | int |

name is the primary key for this table.

This table contains the stored variables and their values.

Table Expressions:

|  |  |
| --- | --- |
| Column Name | Type |
| left\_operand | varchar |
| operator | enum |
| right\_operand | varchar |

(left\_operand, operator, right\_operand) is the primary key for this table. This table contains a boolean expression that should be evaluated. operator is an enum that takes one of the values ('<', '>', '=')

The values of left\_operand and right\_operand are guaranteed to be in the Variables table.

Write an SQL query to evaluate the boolean expressions in Expressions table. Return the result table in any order.

The query result format is in the following example.

Input: Variables table:

|  |  |
| --- | --- |
| name | value |
| x | 66 |
| y | 77 |

Expressions table:

|  |  |  |
| --- | --- | --- |
| left\_operand | operator | right\_operand |
| x | > | y |
| x | < | y |
| x | = | y |
| y | > | x |
| y | < | x |
| x | = | x |

Output:

|  |  |  |  |
| --- | --- | --- | --- |
| left\_operand | operator | right\_operand | value |
| x | > | y | false |
| x | < | y | true |
| x | = | y | false |
| y | > | x | true |
| y | < | x | false |
| x | = | x | true |

Explanation:

As shown, you need to ﬁnd the value of each boolean expression in the table using the variables table.

**select** e.left\_operand, e.**operator**, e.right\_operand,

**case**

**when** e.**operator** **=** '<' **then** if(l.value **<** r.value,'true','false')

**when** e.**operator** **=** '>' **then** if(l.value **>** r.value,'true','false')

**else** if(l.value **=** r.value,'true','false')

**end** **as** value

**from** expressions e

**left** **join** variables l **on** e.left\_operand **=** l.name

**left** **join** variables r **on** e.right\_operand **=** r.name

**Q78**.

Table Person:

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |
| phone\_number | varchar |

id is the primary key for this table.

Each row of this table contains the name of a person and their phone number.

Phone number will be in the form 'xxx-yyyyyyy' where xxx is the country code (3 characters) and yyyyyyy is the phone number (7 characters) where x and y are digits. Both can contain leading zeros.

Table Country:

|  |  |
| --- | --- |
| Column Name | Type |
| name | varchar |
| country\_code | varchar |

country\_code is the primary key for this table.

Each row of this table contains the country name and its code. country\_code will be in the form 'xxx' where x is digits.

Table Calls:

|  |  |
| --- | --- |
| Column Name | Type |
| caller\_id | int |
| callee\_id | int |
| duration | int |

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

Each row of this table contains the caller id, callee id and the duration of the call in minutes. caller\_id

!= callee\_id

A telecommunications company wants to invest in new countries. The company intends to invest in the countries where the average call duration of the calls in this country is strictly greater than the global average call duration.

Write an SQL query to ﬁnd the countries where this company can invest. Return the result table in any order.

The query result format is in the following example.

Input: Person table:

|  |  |  |
| --- | --- | --- |
| id | name | phone\_number |
| 3 | Jonathan | 051-1234567 |
| 12 | Elvis | 051-7654321 |
| 1 | Moncef | 212-1234567 |
| 2 | Maroua | 212-6523651 |
| 7 | Meir | 972-1234567 |
| 9 | Rachel | 972-0011100 |

Country table:

|  |  |
| --- | --- |
| name | country\_code |
| Peru | 51 |
| Israel | 972 |
| Morocco | 212 |
| Germany | 49 |
| Ethiopia | 251 |

Calls table:

|  |  |  |
| --- | --- | --- |
| caller\_id | callee\_id | duration |
| 1 | 9 | 33 |
| 2 | 9 | 4 |
| 1 | 2 | 59 |
| 3 | 12 | 102 |
| 3 | 12 | 330 |
| 12 | 3 | 5 |
| 7 | 9 | 13 |

|  |  |  |
| --- | --- | --- |
| 7 | 1 | 3 |
| 9 | 7 | 1 |
| 1 | 7 | 7 |

Output:

Peru

country

Explanation:

The average call duration for Peru is (102 + 102 + 330 + 330 + 5 + 5) / 6 = 145.666667

The average call duration for Israel is (33 + 4 + 13 + 13 + 3 + 1 + 1 + 7) / 8 = 9.37500

The average call duration for Morocco is (33 + 4 + 59 + 59 + 3 + 7) / 6 = 27.5000

Global call duration average = (2 \* (33 + 4 + 59 + 102 + 330 + 5 + 13 + 3 + 1 + 7)) / 20 = 55.70000 Since Peru is the only country where the average call duration is greater than the global average, it is

the only recommended country.

**select** **c**.name **as** country

**from** Person p

**inner** **join** Country **c**

**on** **left** (p.phone\_number,3) **=** **c**.country\_code

**inner** **join** (**select** caller\_id **as** id, duration

**from** Calls

**union** **all**

**select** callee\_id **as** id, duration

**from** Calls) phn

**on** p.id **=** phn.id

**group** **by** country

**having** **avg**(duration) **>** (**select** **avg**(duration) **from** Calls)

# Q79.

Write a query that prints a list of employee names (i.e.: the name attribute) from the Employee table in alphabetical order.

Level - Easy

Hint - Use ORDER BY Input Format

The Employee table containing employee data for a company is described as follows:



where employee\_id is an employee's ID number, name is their name, months is the total number of months they've been working for the company, and salary is their monthly salary.

Sample Input



# Sample Output

Angela Bonnie Frank Joe Kimberly Lisa Michael Patrick Rose Todd

SELECT NAME FROM EMPLOYEE ORDER BY NAME;

# Q80.

Assume you are given the table below containing information on user transactions for particular products. Write a query to obtain the year-on-year growth rate for the total spend of each product for each year.

Output the year (in ascending order) partitioned by product id, current year's spend, previous year's spend and year-on-year growth rate (percentage rounded to 2 decimal places).

Level - Hard

Hint - Use extract function

user\_transactions Table:

|  |  |
| --- | --- |
| Column Name | Type |
| transaction\_id | integer |
| product\_id | integer |
| spend | decimal |
| transaction\_date | datetime |

user\_transactions Example Input:

|  |  |  |  |
| --- | --- | --- | --- |
| transaction\_i  d | product\_i d | spend | transaction\_date |
| 1341 | 123424 | 1500.60 | 12/31/2019 12:00:00 |
| 1423 | 123424 | 1000.20 | 12/31/2020 12:00:00 |
| 1623 | 123424 | 1246.44 | 12/31/2021 12:00:00 |
| 1322 | 123424 | 2145.32 | 12/31/2022 12:00:00 |

Example Output:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| y | product\_i d | curr\_year\_spend | prev\_year\_spend | yoy\_rate |
| 2 | 123424 | 1500.60 |  |  |
| 2 | 123424 | 1000.20 | 1500.60 | -33.35 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 123424 | 1246.44 | 1000.20 | 24.62 |
| 2 | 123424 | 2145.32 | 1246.44 | 72.12 |

# Q81.

Amazon wants to maximise the number of items it can stock in a 500,000 square feet warehouse. It wants to stock as many prime items as possible, and afterwards use the remaining square footage to stock the most number of non-prime items.

Write a SQL query to ﬁnd the number of prime and non-prime items that can be stored in the 500,000 square feet warehouse. Output the item type and number of items to be stocked.

Hint - create a table containing a summary of the necessary ﬁelds such as item type ('prime\_eligible', 'not\_prime'), SUM of square footage, and COUNT of items grouped by the item type.

https://www.linkedin.com/posts/abhishek-u-3b7703147\_hard-hard-thankyou-activity-7000380721798254592-4VNt/?trk=public\_profile\_like\_view&originalSubdomain=in

inventory table:

|  |  |
| --- | --- |
| Column Name | Type |
| item\_id | integer |
| item\_type | string |
| item\_category | string |
| square\_footage | decimal |

inventory Example Input:

|  |  |  |  |
| --- | --- | --- | --- |
| item\_id | item\_type | item\_category | square\_footage |
| 1374 | prime\_eligible | mini refrigerator | 68.00 |
| 4245 | not\_prime | standing lamp | 26.40 |
| 2452 | prime\_eligible | television | 85.00 |
| 3255 | not\_prime | side table | 22.60 |
| 1672 | prime\_eligible | laptop | 8.50 |

Example Output:

|  |  |
| --- | --- |
| item\_type | item\_count |
| prime\_eligible | 9285 |
| not\_prime | 6 |

# Q82.

Assume you have the table below containing information on Facebook user actions. Write a query to obtain the active user retention in July 2022. Output the month (in numerical format 1, 2, 3) and the number of monthly active users (MAUs).

Hint: An active user is a user who has user action ("sign-in", "like", or "comment") in the current month and last month.

Hint- Use generic correlated subquery user\_actions Table:

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | integer |
| event\_id | integer |
| event\_type | string ("sign-in, "like", "comment") |
| event\_date | datetime |

user\_actionsExample Input:

|  |  |  |  |
| --- | --- | --- | --- |
| user\_id | event\_id | event\_type | event\_date |
| 445 | 7765 | sign-in | 05/31/2022 12:00:00 |
| 742 | 6458 | sign-in | 06/03/2022 12:00:00 |
| 445 | 3634 | like | 06/05/2022 12:00:00 |
| 742 | 1374 | comment | 06/05/2022 12:00:00 |
| 648 | 3124 | like | 06/18/2022 12:00:00 |

Example Output for June 2022:

|  |  |
| --- | --- |
| month | monthly\_active\_users |
| 6 | 1 |

SELECT user\_id,   
 EXTRACT(month from event\_date) as m\_num  
from user\_actions;

# Q83.

Google's marketing team is making a Superbowl commercial and needs a simple statistic to put on their TV ad: the median number of searches a person made last year.

However, at Google scale, querying the 2 trillion searches is too costly. Luckily, you have access to the summary table which tells you the number of searches made last year and how many Google users fall into that bucket.

Write a query to report the median of searches made by a user. Round the median to one decimal point.

Hint- Write a subquery or common table expression (CTE) to generate a series of data (that's keyword for column) starting at the ﬁrst search and ending at some point with an optional incremental value.

search\_frequency Table:

|  |  |
| --- | --- |
| Column Name | Type |
| searches | integer |
| num\_users | integer |

search\_frequency Example Input:

|  |  |
| --- | --- |
| searches | num\_users |
| 1 | 2 |
| 2 | 2 |
| 3 | 3 |
| 4 | 1 |

Example Output:

2.5

median

# Q84.

Write a query to update the Facebook advertiser's status using the daily\_pay table. Advertiser is a two-column table containing the user id and their payment status based on the last payment and daily\_pay table has current information about their payment. Only advertisers who paid will show up in this table.

Output the user id and current payment status sorted by the user id.

Hint- Query the daily\_pay table and check through the advertisers in this table. .

advertiser Table:

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | string |
| status | string |

advertiser Example Input:

|  |  |
| --- | --- |
| user\_id | status |
| bing | NEW |
| yahoo | NEW |
| alibaba | EXISTING |

daily\_pay Table:

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | string |
| paid | decimal |

daily\_pay Example Input:

|  |  |
| --- | --- |
| user\_id | paid |
| yahoo | 45.00 |

|  |  |
| --- | --- |
| alibaba | 100.00 |
| target | 13.00 |

Deﬁnition of advertiser status:

* New: users registered and made their ﬁrst payment.
* Existing: users who paid previously and recently made a current payment.
* Churn: users who paid previously, but have yet to make any recent payment.
* Resurrect: users who did not pay recently but may have made a previous payment and have made payment again recently.

Example Output:

|  |  |
| --- | --- |
| user\_id | new\_status |
| bing | CHURN |
| yahoo | EXISTING |
| alibaba | EXISTING |

Bing's updated status is CHURN because no payment was made in the daily\_pay table whereas Yahoo which made a payment is updated as EXISTING.

The dataset you are querying against may have different input & output - this is just an example! Read this before proceeding to solve the question

For better understanding of the advertiser's status, we're sharing with you a table of possible transitions based on the payment status.

|  |  |  |  |
| --- | --- | --- | --- |
| # | Start | End | Condition |
| 1 | NEW | EXISTING | Paid on day T |
| 2 | NEW | CHURN | No pay on day T |
| 3 | EXISTING | EXISTING | Paid on day T |
| 4 | EXISTING | CHURN | No pay on day T |
| 5 | CHURN | RESURRECT | Paid on day T |
| 6 | CHURN | CHURN | No pay on day T |
| 7 | RESURRECT | EXISTING | Paid on day T |

|  |  |  |  |
| --- | --- | --- | --- |
| 8 | RESURRECT | CHURN | No pay on day T |

1. Row 2, 4, 6, 8: As long as the user has not paid on day T, the end status is updated to CHURN regardless of the previous status.
2. Row 1, 3, 5, 7: When the user paid on day T, the end status is updated to either EXISTING or RESURRECT, depending on their previous state. RESURRECT is only possible when the previous state is CHURN. When the previous state is anything else, the status is updated to EXISTING.

# Q85.

Amazon Web Services (AWS) is powered by ﬂeets of servers. Senior management has requested data-driven solutions to optimise server usage.

Write a query that calculates the total time that the ﬂeet of servers was running. The output should be in units of full days.

Level - Hard Hint-

1. Calculate individual uptimes
2. Sum those up to obtain the uptime of the whole ﬂeet, keeping in mind that the result must be output in units of full days

Assumptions:

* + Each server might start and stop several times.
  + The total time in which the server ﬂeet is running can be calculated as the sum of each server's uptime.

server\_utilization Table:

|  |  |
| --- | --- |
| Column Name | Type |
| server\_id | integer |
| status\_time | timestamp |
| session\_status | string |

server\_utilization Example Input:

|  |  |  |
| --- | --- | --- |
| server\_id | status\_time | session\_status |
| 1 | 08/02/2022 10:00:00 | start |
| 1 | 08/04/2022 10:00:00 | stop |
| 2 | 08/17/2022 10:00:00 | start |
| 2 | 08/24/2022 10:00:00 | stop |

Example Output:

21

total\_uptime\_days

# Q86.

Sometimes, payment transactions are repeated by accident; it could be due to user error, API failure or a retry error that causes a credit card to be charged twice.

Using the transactions table, identify any payments made at the same merchant with the same credit card for the same amount within 10 minutes of each other. Count such repeated payments.

Level - Hard

Hint- Use Partition and order by

Assumptions:

* + The ﬁrst transaction of such payments should not be counted as a repeated payment. This means, if there are two transactions performed by a merchant with the same credit card and for the same amount within 10 minutes, there will only be 1 repeated payment.

transactions Table:

|  |  |
| --- | --- |
| Column Name | Type |
| transaction\_id | integer |
| merchant\_id | integer |
| credit\_card\_id | integer |
| amount | integer |

|  |  |
| --- | --- |
| transaction\_timestamp | datetime |

transactions Example Input:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| transaction\_id | merchant\_id | credit\_card\_id | amount | transaction\_timestamp |
| 1 | 101 | 1 | 100 | 09/25/2022 12:00:00 |
| 2 | 101 | 1 | 100 | 09/25/2022 12:08:00 |
| 3 | 101 | 1 | 100 | 09/25/2022 12:28:00 |
| 4 | 102 | 2 | 300 | 09/25/2022 12:00:00 |
| 6 | 102 | 2 | 400 | 09/25/2022 14:00:00 |

Example Output:

1

payment\_count

# Q87.

DoorDash's Growth Team is trying to make sure new users (those who are making orders in their ﬁrst 14 days) have a great experience on all their orders in their 2 weeks on the platform.

Unfortunately, many deliveries are being messed up because:

* + the orders are being completed incorrectly (missing items, wrong order, etc.)
  + the orders aren't being received (wrong address, wrong drop off spot)
  + the orders are being delivered late (the actual delivery time is 30 minutes later than when the order was placed). Note that the estimated\_delivery\_timestamp is automatically set to 30 minutes after the order\_timestamp.

Hint- Use Where Clause and joins

Write a query to ﬁnd the bad experience rate in the ﬁrst 14 days for new users who signed up in June 2022. Output the percentage of bad experience rounded to 2 decimal places.

orders Table:

|  |  |
| --- | --- |
| Column Name | Type |
| order\_id | integer |

|  |  |
| --- | --- |
| customer\_id | integer |
| trip\_id | integer |
| status | string ('completed successfully', 'completed incorrectly', 'never received') |
| order\_timestamp | timestamp |

orders Example Input:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| order\_id | customer\_id | trip\_id | status | order\_timestamp |
| 727424 | 8472 | 100463 | completed successfully | 06/05/2022 09:12:00 |
| 242513 | 2341 | 100482 | completed incorrectly | 06/05/2022 14:40:00 |
| 141367 | 1314 | 100362 | completed incorrectly | 06/07/2022 15:03:00 |
| 582193 | 5421 | 100657 | never\_received | 07/07/2022 15:22:00 |
| 253613 | 1314 | 100213 | completed successfully | 06/12/2022 13:43:00 |

trips Table:

|  |  |
| --- | --- |
| Column Name | Type |
| dasher\_id | integer |
| trip\_id | integer |
| estimated\_delivery\_timestamp | timestamp |
| actual\_delivery\_timestamp | timestamp |

trips Example Input:

|  |  |  |  |
| --- | --- | --- | --- |
| dasher\_id | trip\_id | estimated\_delivery\_timestamp | actual\_delivery\_timestamp |
| 101 | 100463 | 06/05/2022 09:42:00 | 06/05/2022 09:38:00 |
| 102 | 100482 | 06/05/2022 15:10:00 | 06/05/2022 15:46:00 |

|  |  |  |  |
| --- | --- | --- | --- |
| 101 | 100362 | 06/07/2022 15:33:00 | 06/07/2022 16:45:00 |
| 102 | 100657 | 07/07/2022 15:52:00 | - |
| 103 | 100213 | 06/12/2022 14:13:00 | 06/12/2022 14:10:00 |

customers Table:

|  |  |
| --- | --- |
| Column Name | Type |
| customer\_id | integer |
| signup\_timestamp | timestamp |

customers Example Input:

|  |  |
| --- | --- |
| customer\_id | signup\_timestamp |
| 8472 | 05/30/2022 00:00:00 |
| 2341 | 06/01/2022 00:00:00 |
| 1314 | 06/03/2022 00:00:00 |
| 1435 | 06/05/2022 00:00:00 |
| 5421 | 06/07/2022 00:00:00 |

Example Output:

75.00

bad\_experience\_pct

**with totorders as(  
select o.order\_id, o.customer\_id, o.trip\_id,o.status, o.order\_timestamp,  
t.estimated\_delivery\_timestamp as etimestamp, t.actual\_delivery\_timestamp as atimestamp,  
c.signup\_timestamp  
from orders as o  
join trips as t on t.trip\_id = o.trip\_id  
join customers as c on c.customer\_id = o.customer\_id  
where extract(month from c.signup\_timestamp) = 06  
AND extract(year from c.signup\_timestamp) = 2022  
and c.signup\_timestamp+interval '14 days' > o.order\_timestamp  
),  
badorders as(  
select \*  
from totorders  
where not status = 'completed successfully' or  
atimestamp>etimestamp or  
atimestamp is null  
)  
select round(100.0\*count(order\_id)/(select count(order\_id)from totorders),2) as bad\_experience\_pct  
from badorders**

# Q88

Table: Scores

|  |  |
| --- | --- |
| Column Name | Type |
| player\_name | varchar |
| gender | varchar |
| day | date |
| score\_points | int |

(gender, day) is the primary key for this table.

A competition is held between the female team and the male team.

Each row of this table indicates that a player\_name and with gender has scored score\_point in someday.

Gender is 'F' if the player is in the female team and 'M' if the player is in the male team.

Write an SQL query to ﬁnd the total score for each gender on each day. Return the result table ordered by gender and day in ascending order. The query result format is in the following example.

Input: Scores table:

|  |  |  |  |
| --- | --- | --- | --- |
| player\_name | gender | day | score\_points |
| Aron | F | 2020-01-01 | 17 |
| Alice | F | 2020-01-07 | 23 |
| Bajrang | M | 2020-01-07 | 7 |
| Khali | M | 2019-12-25 | 11 |
| Slaman | M | 2019-12-30 | 13 |
| Joe | M | 2019-12-31 | 3 |
| Jose | M | 2019-12-18 | 2 |
| Priya | F | 2019-12-31 | 23 |
| Priyanka | F | 2019-12-30 | 17 |

Output:

|  |  |  |
| --- | --- | --- |
| gender | day | total |
| F | 2019-12-30 | 17 |
| F | 2019-12-31 | 40 |
| F | 2020-01-01 | 57 |
| F | 2020-01-07 | 80 |
| M | 2019-12-18 | 2 |
| M | 2019-12-25 | 13 |

|  |  |  |
| --- | --- | --- |
| M | 2019-12-30 | 26 |
| M | 2019-12-31 | 29 |
| M | 2020-01-07 | 36 |

Explanation:

For the female team:

The ﬁrst day is 2019-12-30, Priyanka scored 17 points and the total score for the team is 17. The second day is 2019-12-31, Priya scored 23 points and the total score for the team is 40. The third day is 2020-01-01, Aron scored 17 points and the total score for the team is 57.

The fourth day is 2020-01-07, Alice scored 23 points and the total score for the team is 80.

For the male team:

The ﬁrst day is 2019-12-18, Jose scored 2 points and the total score for the team is 2.

The second day is 2019-12-25, Khali scored 11 points and the total score for the team is 13. The third day is 2019-12-30, Slaman scored 13 points and the total score for the team is 26. The fourth day is 2019-12-31, Joe scored 3 points and the total score for the team is 29.

The ﬁfth day is 2020-01-07, Bajrang scored 7 points and the total score for the team is 36.

**select** s.gender, s.**day**, (**select** **sum**(score\_points) **from** Scores **where** gender **=** s.gender **and** **day** **<=** s.**day**) **as** total

**from** Scores s

**group** **by** gender, **day**

**order** **by** gender, **day**;

# Q89.

Table Person:

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |
| phone\_number | varchar |

id is the primary key for this table.

Each row of this table contains the name of a person and their phone number.

Phone number will be in the form 'xxx-yyyyyyy' where xxx is the country code (3 characters) and yyyyyyy is the phone number (7 characters) where x and y are digits. Both can contain leading zeros.

Table Country:

|  |  |
| --- | --- |
| Column Name | Type |
| name | varchar |
| country\_code | varchar |

country\_code is the primary key for this table.

Each row of this table contains the country name and its code. country\_code will be in the form 'xxx' where x is digits.

Table Calls:

|  |  |
| --- | --- |
| Column Name | Type |
| caller\_id | int |
| callee\_id | int |
| duration | int |

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

Each row of this table contains the caller id, callee id and the duration of the call in minutes. caller\_id

!= callee\_id

A telecommunications company wants to invest in new countries. The company intends to invest in the countries where the average call duration of the calls in this country is strictly greater than the global average call duration.

Write an SQL query to ﬁnd the countries where this company can invest. Return the result table in any order.

The query result format is in the following example.

Input: Person table:

|  |  |  |
| --- | --- | --- |
| id | name | phone\_number |
| 3 | Jonathan | 051-1234567 |
| 12 | Elvis | 051-7654321 |
| 1 | Moncef | 212-1234567 |
| 2 | Maroua | 212-6523651 |
| 7 | Meir | 972-1234567 |
| 9 | Rachel | 972-0011100 |

Country table:

|  |  |
| --- | --- |
| name | country\_code |
| Peru | 51 |
| Israel | 972 |
| Morocco | 212 |
| Germany | 49 |
| Ethiopia | 251 |
| Ethiopia | 251 |

Calls table:

|  |  |  |
| --- | --- | --- |
| caller\_id | callee\_id | duration |
| 1 | 9 | 33 |
| 2 | 9 | 4 |

|  |  |  |
| --- | --- | --- |
| 1 | 2 | 59 |
| 3 | 12 | 102 |
| 3 | 12 | 330 |
| 12 | 3 | 5 |
| 7 | 9 | 13 |
| 7 | 1 | 3 |
| 9 | 7 | 1 |
| 1 | 7 | 7 |

Output:

Peru

country

Explanation:

The average call duration for Peru is (102 + 102 + 330 + 330 + 5 + 5) / 6 = 145.666667

The average call duration for Israel is (33 + 4 + 13 + 13 + 3 + 1 + 1 + 7) / 8 = 9.37500

The average call duration for Morocco is (33 + 4 + 59 + 59 + 3 + 7) / 6 = 27.5000

Global call duration average = (2 \* (33 + 4 + 59 + 102 + 330 + 5 + 13 + 3 + 1 + 7)) / 20 = 55.70000 Since Peru is the only country where the average call duration is greater than the global average, it is the only recommended country.

|  |
| --- |
| select s1.gender, s1.day, sum(s2.score\_points) as total from Scores s1, Scores s2 |
|  | where s1.gender = s2.gender and s1.day >= s2.day |
|  | group by s1.gender, s1.day |
|  | order by s1.gender, s1.day |

# Q90.

Table: Numbers

|  |  |
| --- | --- |
| Column Name | Type |
| num | int |
| frequency | int |

num is the primary key for this table.

Each row of this table shows the frequency of a number in the database.

The median is the value separating the higher half from the lower half of a data sample.

Write an SQL query to report the median of all the numbers in the database after decompressing the Numbers table. Round the median to one decimal point.

The query result format is in the following example.

|  |  |
| --- | --- |
| num | frequency |
| 0 | 7 |
| 1 | 1 |
| 2 | 3 |
| 3 | 1 |

Output:

0

median

Explanation:

If we decompose the Numbers table, we will get [0, 0, 0, 0, 0, 0, 0, 1, 2, 2, 2, 3], so the median is (0 + 0) /

2 = 0.

select avg(number \* 1.0)

from (select t.\*,

sum(freq) over (order by number asc) as sum\_freq,

sum(freq) over () as cnt

from t

) t

where cnt - 1 <= 2 \* sum\_freq and

cnt - 1 >= 2 \* (sum\_freq - freq)

# Q91.

Table: Salary

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| employee\_id | int |
| amount | int |
| pay\_date | date |

id is the primary key column for this table.

Each row of this table indicates the salary of an employee in one month. employee\_id is a foreign key from the Employee table.

Table: Employee

|  |  |
| --- | --- |
| Column Name | Type |
| employee\_id | int |
| department\_id | int |

employee\_id is the primary key column for this table.

Each row of this table indicates the department of an employee.

Write an SQL query to report the comparison result (higher/lower/same) of the average salary of employees in a department to the company's average salary.

Return the result table in any order.

The query result format is in the following example.

select

pay\_month,

department\_id,

case when dept\_avg > comp\_avg then 'higher' when dept\_avg < comp\_avg then 'lower' else 'same' end comparison

from (select date\_format(b.pay\_date, '%Y-%m') pay\_month, a.department\_id, avg(b.amount) dept\_avg, d.comp\_avg

from employee a

inner join salary b

on (a.employee\_id = b.employee\_id)

inner join (select date\_format(c.pay\_date, '%Y-%m') pay\_month, avg(c.amount) comp\_avg

from salary c

group by date\_format(c.pay\_date, '%Y-%m')) d

on ( date\_format(b.pay\_date, '%Y-%m') = d.pay\_month)

group by date\_format(b.pay\_date, '%Y-%m'), department\_id, d.comp\_avg) fi

|  |  |  |  |
| --- | --- | --- | --- |
| id | employee\_id | amount | pay\_date |
| 1 | 1 | 9000 | 2017/03/31 |
| 2 | 2 | 6000 | 2017/03/31 |
| 3 | 3 | 10000 | 2017/03/31 |
| 4 | 1 | 7000 | 2017/02/28 |
| 5 | 2 | 6000 | 2017/02/28 |
| 6 | 3 | 8000 | 2017/02/28 |

Employee table:

|  |  |
| --- | --- |
| employee\_id | department\_id |
| 1 | 1 |
| 2 | 2 |
| 3 | 2 |

Output:

|  |  |  |
| --- | --- | --- |
| pay\_month | department\_id | comparison |
| 2017-02 | 1 | same |
| 2017-03 | 1 | higher |
| 2017-02 | 2 | same |
| 2017-03 | 2 | lower |

Explanation:

In March, the company's average salary is (9000+6000+10000)/3 = 8333.33...

The average salary for department '1' is 9000, which is the salary of employee\_id '1' since there is only one employee in this department. So the comparison result is 'higher' since 9000 > 8333.33 obviously. The average salary of department '2' is (6000 + 10000)/2 = 8000, which is the average of employee\_id '2' and '3'. So the comparison result is 'lower' since 8000 < 8333.33.

With the same formula for the average salary comparison in February, the result is 'same' since both the departments '1' and '2' have the same average salary with the company, which is 7000.

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 games.

Each row is a record of a player who logged in and played a number of games (possibly 0) before logging out on someday using some device.

The install date of a player is the ﬁrst login day of that player.

We deﬁne day one retention of some date x to be the number of players whose install date is x and they logged back in on the day right after x, divided by the number of players whose install date is x, rounded to 2 decimal places.

Write an SQL query to report for each install date, the number of players that installed the game on that day, and the day one retention.

Return the result table in any order.

The query result format is in the following example.

Input:

Activity table:

|  |  |  |  |
| --- | --- | --- | --- |
| player\_id | device\_id | event\_date | games\_played |
| 1 | 2 | 2016-03-01 | 5 |
| 1 | 2 | 2016-03-02 | 6 |
| 2 | 3 | 2017-06-25 | 1 |
| 3 | 1 | 2016-03-01 | 0 |
| 3 | 4 | 2016-07-03 | 5 |

Output:

|  |  |  |
| --- | --- | --- |
| install\_dt | installs | Day1\_retention |
| 2016-03-01 | 2 | 0.5 |
| 2017-06-25 | 1 | 0 |

Explanation:

Player 1 and 3 installed the game on 2016-03-01 but only player 1 logged back in on 2016-03-02 so the day 1 retention of 2016-03-01 is 1 / 2 = 0.50

Player 2 installed the game on 2017-06-25 but didn't log back in on 2017-06-26 so the day 1 retention of 2017-06-25 is 0 / 1 = 0.00

**select** a1.event\_date **as** install\_dt, **count**(a1.player\_id) **as** installs, round(**count**(a3.player\_id) **/** **count**(a1.player\_id), 2) **as** Day1\_retention

**from** Activity a1 **left** **join** Activity a2

**on** a1.player\_id **=** a2.player\_id **and** a1.event\_date **>** a2.event\_date

**left** **join** Activity a3

**on** a1.player\_id **=** a3.player\_id **and** datediff(a3.event\_date, a1.event\_date) **=** 1

**where** a2.event\_date **is** **null**

**group** **by** a1.event\_date;

Table: Players

|  |  |
| --- | --- |
| Column Name | Type |
| player\_id | int |
| group\_id | int |

player\_id is the primary key of this table.

Each row of this table indicates the group of each player.

Table: Matches

|  |  |
| --- | --- |
| Column Name | Type |
| match\_id | int |
| ﬁrst\_player | int |
| second\_player | int |
| ﬁrst\_score | int |
| second\_score | int |

match\_id is the primary key of this table.

Each row is a record of a match, ﬁrst\_player and second\_player contain the player\_id of each match. ﬁrst\_score and second\_score contain the number of points of the ﬁrst\_player and second\_player respectively.

You may assume that, in each match, players belong to the same group.

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

Write an SQL query to ﬁnd the winner in each group. Return the result table in any order.

The query result format is in the following example.

Input: Players table:

|  |  |
| --- | --- |
| player\_id | group\_id |
| 15 | 1 |
| 25 | 1 |
| 30 | 1 |
| 45 | 1 |
| 10 | 2 |
| 35 | 2 |
| 50 | 2 |

|  |  |
| --- | --- |
| 20 | 3 |
| 40 | 3 |

Matches table:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| match\_id | ﬁrst\_player | second\_player | ﬁrst\_score | second\_score |
| 1 | 15 | 45 | 3 | 0 |
| 2 | 30 | 25 | 1 | 2 |
| 3 | 30 | 15 | 2 | 0 |
| 4 | 40 | 20 | 5 | 2 |
| 5 | 35 | 50 | 1 | 1 |

Output:

|  |  |  |
| --- | --- | --- |
| group\_id | | player\_id |
| 1 | | 15 |
| 2 | | 35 |
| 3 | | 40 |
| SELECT group\_id, | |
|  | | player\_id | | |
|  | | FROM (SELECT p.group\_id, | | |
|  | | ps.player\_id, | | |
|  | | Sum(ps.score) AS score | | |
|  | | FROM players p INNER JOIN | | |
|  | | (SELECT first\_player AS player\_id, | | |
|  | | first\_score AS score | | |
|  | | FROM matches | | |
|  | | UNION ALL | | |
|  | | SELECT second\_player AS player\_id, | | |
|  | | second\_score AS score | | |
|  | | FROM matches) ps | | |
|  | | ON p.player\_id = ps.player\_id | | |
|  | | GROUP BY ps.player\_id | | |
|  | | ORDER BY group\_id, | | |
|  | | score DESC, | | |
|  | | player\_id) top\_scores | | |
|  | | GROUP BY group\_id | | |

# Q94.

Table: Student

|  |  |
| --- | --- |
| Column Name | Type |
| student\_id | int |
| student\_name | varchar |

student\_id is the primary key for this table. student\_name is the name of the student.

Table: Exam

|  |  |
| --- | --- |
| Column Name | Type |
| exam\_id | int |
| student\_id | int |
| score | int |

(exam\_id, student\_id) is the primary key for this table.

Each row of this table indicates that the student with student\_id had a score points in the exam with id exam\_id.

A quiet student is the one who took at least one exam and did not score the high or the low score. Write an SQL query to report the students (student\_id, student\_name) being quiet in all exams. Do not return the student who has never taken any exam.

Return the result table ordered by student\_id.

The query result format is in the following example.

Input:

Student table:

|  |  |
| --- | --- |
| student\_id | student\_name |
| 1 | Daniel |
| 2 | Jade |
| 3 | Stella |
| 4 | Jonathan |
| 5 | Will |

Exam table:

|  |  |  |
| --- | --- | --- |
| exam\_id | student\_id | score |
| 10 | 1 | 70 |
| 10 | 2 | 80 |
| 10 | 3 | 90 |
| 20 | 1 | 80 |
| 30 | 1 | 70 |
| 30 | 3 | 80 |
| 30 | 4 | 90 |
| 40 | 1 | 60 |
| 40 | 2 | 70 |
| 40 | 4 | 80 |

Output:

|  |  |
| --- | --- |
| student\_id | student\_name |
| 2 | Jade |

Explanation:

For exam 1: Student 1 and 3 hold the lowest and high scores respectively. For exam 2: Student 1 holds both the highest and lowest score.

For exam 3 and 4: Student 1 and 4 hold the lowest and high scores respectively. Students 2 and 5 have never got the highest or lowest in any of the exams.

Since student 5 is not taking any exam, he is excluded from the result. So, we only return the information of Student 2.

|  |
| --- |
| select distinct Student.\* |
|  | from Student inner join Exam |
|  | on Student.student\_id = Exam.student\_id |
|  | where student.student\_id not in |
|  | (select e1.student\_id |
|  | from Exam as e1 inner join |
|  | (select exam\_id, min(score) as min\_score, max(score) as max\_score |
|  | from Exam |
|  | group by exam\_id) as e2 |
|  | on e1.exam\_id = e2.exam\_id |
|  | where e1.score = e2.min\_score or e1.score = e2.max\_score) |
|  | order by student\_id |

# Q95.

Table: Student

|  |  |
| --- | --- |
| Column Name | Type |
| student\_id | int |
| student\_name | varchar |

student\_id is the primary key for this table. student\_name is the name of the student.

Table: Exam

|  |  |
| --- | --- |
| Column Name | Type |
| exam\_id | int |
| student\_id | int |
| score | int |

(exam\_id, student\_id) is the primary key for this table.

Each row of this table indicates that the student with student\_id had a score points in the exam with id exam\_id.

A quiet student is the one who took at least one exam and did not score the high or the low score. Write an SQL query to report the students (student\_id, student\_name) being quiet in all exams. Do not return the student who has never taken any exam.

Return the result table ordered by student\_id.

The query result format is in the following example.

Input: Student table:

|  |  |
| --- | --- |
| student\_id | student\_name |
| 1 | Daniel |
| 2 | Jade |
| 3 | Stella |
| 4 | Jonathan |
| 5 | Will |

Exam table:

|  |  |  |
| --- | --- | --- |
| exam\_id | student\_id | score |
| 10 | 1 | 70 |
| 10 | 2 | 80 |
| 10 | 3 | 90 |
| 20 | 1 | 80 |
| 30 | 1 | 70 |

|  |  |  |
| --- | --- | --- |
| 30 | 3 | 80 |
| 30 | 4 | 90 |
| 40 | 1 | 60 |
| 40 | 2 | 70 |
| 40 | 4 | 80 |

Output:

|  |  |
| --- | --- |
| student\_id | student\_name |
| 2 | Jade |

Explanation:

For exam 1: Student 1 and 3 hold the lowest and high scores respectively. For exam 2: Student 1 holds both the highest and lowest score.

For exam 3 and 4: Student 1 and 4 hold the lowest and high scores respectively. Students 2 and 5 have never got the highest or lowest in any of the exams.

Since student 5 is not taking any exam, he is excluded from the result. So, we only return the information of Student 2.

WITH cte AS (

SELECT student\_id

FROM (

SELECT \*,

(MIN(score) OVER (PARTITION BY exam\_id) = score) +

(MAX(score) OVER (PARTITION BY exam\_id) = score) flag

FROM exam

) t

GROUP BY student\_id

HAVING SUM(flag) = 0

)

SELECT \*

FROM student

WHERE student\_id IN (SELECT student\_id FROM cte)

# Q96.

You're given two tables on Spotify users' streaming data. songs\_history table contains the historical streaming data and songs\_weekly table contains the current week's streaming data.

Write a query to output the user id, song id, and cumulative count of song plays as of 4 August 2022 sorted in descending order.

Hint- Use group by Deﬁnitions:

* + song\_weekly table currently holds data from 1 August 2022 to 7 August 2022.
  + songs\_history table currently holds data up to to 31 July 2022. The output should include the historical data in this table.

Assumption:

* + There may be a new user or song in the songs\_weekly table not present in the songs\_history table.

songs\_history Table:

|  |  |
| --- | --- |
| Column Name | Type |
| history\_id | integer |
| user\_id | integer |
| song\_id | integer |
| song\_plays | integer |

songs\_history Example Input:

|  |  |  |  |
| --- | --- | --- | --- |
| history\_id | user\_id | song\_id | song\_plays |
| 10011 | 777 | 1238 | 11 |
| 12452 | 695 | 4520 | 1 |

song\_plays: Refers to the historical count of streaming or song plays by the user.

songs\_weekly Table:

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | integer |
| song\_id | integer |
| listen\_time | datetime |

songs\_weekly Example Input:

|  |  |  |
| --- | --- | --- |
| user\_id | song\_id | listen\_time |
| 777 | 1238 | 08/01/2022 12:00:00 |
| 695 | 4520 | 08/04/2022 08:00:00 |

|  |  |  |
| --- | --- | --- |
| 125 | 9630 | 08/04/2022 16:00:00 |
| 695 | 9852 | 08/07/2022 12:00:00 |

|  |  |  |
| --- | --- | --- |
| user\_id | song\_id | song\_plays |
| 777 | 1238 | 12 |
| 695 | 4520 | 2 |
| 125 | 9630 | 1 |

with maintable as( select co.customer\_id, co.num\_seats, co.yearly\_seat\_cost, cu.employee\_count, case when cu.employee\_count < 100 then 'SMB' when cu.employee\_count BETWEEN 100 and 999 then 'Mid-Market' else 'Enterprise' end as seg, (num\_seats \* yearly\_seat\_cost) as tot\_cost from contracts as co join customers as cu on cu.customer\_id = co.customer\_id ), revcustomer as( select seg, sum(tot\_cost)/count(distinct customer\_id) as avg\_rev\_per\_cust from maintable group by seg ) select sum(avg\_rev\_per\_cust) filter(where seg = 'SMB') as avg\_smb, sum(avg\_rev\_per\_cust) filter(where seg = 'Mid-Market') as avg\_mmd, sum(avg\_rev\_per\_cust) filter(where seg = 'Enterprise') as avg\_ent from revcustomer

# Q97.

New TikTok users sign up with their emails, so each signup requires a text conﬁrmation to activate the new user's account.

Write a query to ﬁnd the conﬁrmation rate of users who conﬁrmed their signups with text messages. Round the result to 2 decimal places.

Hint- Use Joins Assumptions:

* + A user may fail to conﬁrm several times with text. Once the signup is conﬁrmed for a user, they will not be able to initiate the signup again.
  + A user may not initiate the signup conﬁrmation process at all.

emails Table:

|  |  |
| --- | --- |
| Column Name | Type |
| email\_id | integer |
| user\_id | integer |
| signup\_date | datetime |

emails Example Input:

|  |  |  |
| --- | --- | --- |
| email\_id | user\_id | signup\_date |
| 125 | 7771 | 06/14/2022 00:00:00 |

|  |  |  |
| --- | --- | --- |
| 236 | 6950 | 07/01/2022 00:00:00 |
| 433 | 1052 | 07/09/2022 00:00:00 |

|  |  |
| --- | --- |
| Column Name | Type |
| text\_id | integer |
| email\_id | integer |
| signup\_action | varchar |

texts Example Input:

|  |  |  |
| --- | --- | --- |
| text\_id | email\_id | signup\_action |
| 6878 | 125 | Conﬁrmed |
| 6920 | 236 | Not Conﬁrmed |
| 6994 | 236 | Conﬁrmed |

Example Output:

0.67

conﬁrm\_rate

with CTE as ( select \* from  (SELECT count(\*) as ToTal\_Count FROM emails as A inner JOIN texts as B on A.email\_id = B.email\_id) as Total, (SELECT count(\*) as ToTal\_Confirmation\_Count FROM emails as A inner JOIN texts as B on A.email\_id = B.email\_id where B.signup\_action = 'Confirmed') as ToTal\_Confirmation ) select round(cast(ToTal\_Confirmation\_Count as decimal(10,2))/cast(ToTal\_Count as decimal(10,2)),2) from CTE

# Q98.

The table below contains information about tweets over a given period of time. Calculate the 3-day rolling average of tweets published by each user for each date that a tweet was posted. Output the user id, tweet date, and rolling averages rounded to 2 decimal places.

Hint- Use Count and group by Important Assumptions:

* + Rows in this table are *consecutive* and ordered by date.
  + Each row represents a different day
  + A day that does not correspond to a row in this table is not counted. The most recent day is the next row above the current row.

Note: Rolling average is a metric that helps us analyze data points by creating a series of averages based on different subsets of a dataset. It is also known as a moving average, running average, moving mean, or rolling mean.

tweets Table:

|  |  |
| --- | --- |
| Column Name | Type |
| tweet\_id | integer |
| user\_id | integer |
| tweet\_date | timestamp |

tweets Example Input:

|  |  |  |
| --- | --- | --- |
| tweet\_id | user\_id | tweet\_date |
| 214252 | 111 | 06/01/2022 12:00:00 |
| 739252 | 111 | 06/01/2022 12:00:00 |
| 846402 | 111 | 06/02/2022 12:00:00 |
| 241425 | 254 | 06/02/2022 12:00:00 |
| 137374 | 111 | 06/04/2022 12:00:00 |

Example Output:

|  |  |  |
| --- | --- | --- |
| user\_id | tweet\_date | rolling\_avg\_3days |
| 111 | 06/01/2022 12:00:00 | 2.00 |
| 111 | 06/02/2022 12:00:00 | 1.50 |
| 111 | 06/04/2022 12:00:00 | 1.33 |
| 254 | 06/02/2022 12:00:00 | 1.00 |

**select** user\_id, tweet\_date,

round(avg(total\_tweets) over(partition **by** user\_id **order** **by** tweet\_date **asc**

**rows** between 2 preceding and **current** row),2) **as** rolling\_avg\_3d

**from** (

**select** user\_id, tweet\_date, count(\*) total\_tweets **from** tweets **group**

user\_id,tweet\_date **order** **by** tweet\_date **asc** x

# Q99.

Assume you are given the tables below containing information on Snapchat users, their ages, and their time spent sending and opening snaps. Write a query to obtain a breakdown of the time spent sending vs. opening snaps (as a percentage of total time spent on these activities) for each age group.

Hint- Use join and case

Output the age bucket and percentage of sending and opening snaps. Round the percentage to 2 decimal places.

Notes:

* + You should calculate these percentages:
    - time sending / (time sending + time opening)
    - time opening / (time sending + time opening)
  + To avoid integer division in percentages, multiply by 100.0 and not 100.

activities Table:

|  |  |
| --- | --- |
| Column Name | Type |
| activity\_id | integer |
| user\_id | integer |
| activity\_type | string ('send', 'open', 'chat') |
| time\_spent | ﬂoat |
| activity\_date | datetime |

activities Example Input:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| activity\_id | user\_id | activity\_type | time\_spent | activity\_date |
| 7274 | 123 | open | 4.50 | 06/22/2022 12:00:00 |
| 2425 | 123 | send | 3.50 | 06/22/2022 12:00:00 |
| 1413 | 456 | send | 5.67 | 06/23/2022 12:00:00 |
| 1414 | 789 | chat | 11.00 | 06/25/2022 12:00:00 |
| 2536 | 456 | open | 3.00 | 06/25/2022 12:00:00 |

age\_breakdown Table:

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | integer |
| age\_bucket | string ('21-25', '26-30', '31-25') |

age\_breakdown Example Input:

|  |  |
| --- | --- |
| user\_id | age\_bucket |
| 123 | 31-35 |
| 456 | 26-30 |
| 789 | 21-25 |

Example Output:

|  |  |  |
| --- | --- | --- |
| age\_bucket | send\_perc | open\_perc |
| 26-30 | 65.40 | 34.60 |
| 31-35 | 43.75 | 56.25 |

WITH total\_spents AS (SELECT age.age\_bucket,            SUM(CASE WHEN act.activity\_type = 'send' THEN time\_spent end) as sending\_time,            SUM(CASE WHEN act.activity\_type = 'open' THEN time\_spent end) as opening\_time            FROM activities act             JOIN age\_breakdown age            on act.user\_id = age.user\_id            GROUP BY age.age\_bucket) SELECT age\_bucket,     ROUND(CAST(sending\_time/(sending\_time + opening\_time) :: FLOAT \* 100.00 as numeric), 2) as send\_perc,     ROUND(CAST (opening\_time/(sending\_time + opening\_time) :: FLOAT \* 100.00 as numeric), 2) as open\_perc FROM total\_spents

# Q100 .

The LinkedIn Creator team is looking for power creators who use their personal proﬁle as a company or inﬂuencer page. This means that if someone's Linkedin page has more followers than all the companies they work for, we can safely assume that person is a Power Creator. Keep in mind that if a person works at multiple companies, we should take into account the company with the most followers.

Level - Medium

Hint- Use join and group by

Write a query to return the IDs of these LinkedIn power creators in ascending order. Assumptions:

* + A person can work at multiple companies.
  + In the case of multiple companies, use the one with largest follower base.

personal\_proﬁles Table:

|  |  |
| --- | --- |
| Column Name | Type |
| proﬁle\_id | integer |
| name | string |
| followers | integer |

personal\_proﬁles Example Input:

|  |  |  |
| --- | --- | --- |
| proﬁle\_id | name | followers |
| 1 | Nick Singh | 92,000 |
| 2 | Zach Wilson | 199,000 |
| 3 | Daliana Liu | 171,000 |
| 4 | Ravit Jain | 107,000 |
| 5 | Vin Vashishta | 139,000 |
| 6 | Susan Wojcicki | 39,000 |

employee\_company Table:

|  |  |
| --- | --- |
| Column Name | Type |
| personal\_proﬁle\_id | integer |
| company\_id | integer |

employee\_company Example Input:

|  |  |
| --- | --- |
| personal\_proﬁle\_id | company\_id |
| 1 | 4 |
| 1 | 9 |
| 2 | 2 |
| 3 | 1 |
| 4 | 3 |
| 5 | 6 |
| 6 | 5 |

company\_pages Table:

|  |  |
| --- | --- |
| Column Name | Type |
| company\_id | integer |

|  |  |
| --- | --- |
| name | string |
| followers | integer |

company\_pages Example Input:

|  |  |  |
| --- | --- | --- |
| company\_id | name | followers |
| 1 | The Data Science Podcast | 8,000 |
| 2 | Airbnb | 700,000 |
| 3 | The Ravit Show | 6,000 |
| 4 | DataLemur | 200 |
| 5 | YouTube | 1,6000,000 |
| 6 | DataScience.Vin | 4,500 |
| 9 | Ace The Data Science Interview | 4479 |

Example Output:

|  |
| --- |
| proﬁle\_id |
| 1 |
| 3 |
| 4 |
| 5 |

Select \* from personal\_profiles pf

Join company\_pages cp

On pf.employer\_id = cp.company\_id