**SQL problems**

**1.Problem statement - Big Countries**

**Send feedback**

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

**Answer**

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

**2.Warehouse Manager**

**Easy**

**40/40**

**Average time to solve is 6m**

**Problem statement**

**Send feedback**

**Table: Warehouse**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| name         | varchar |
| product_id   | int     |
| units        | int     |
+--------------+---------+
```

**(name, product_id) is the primary key for this table.**

**Each row of this table contains the information of the products in each warehouse.**

**Table: Products**

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

**| product_name | varchar |**

**| Width | int |**

**| Length | int |**

**| Height | int |**

**+--------------+---------+**

**product_id is the primary key for this table.**

**Each row of this table contains the information about the product dimensions (Width, Lenght and Height) in feets of each product.**

**Solutions:**

```
SELECT w.name AS WarehouseName,
    p.product_name AS ProductName,
    w.units AS Units,
    p.Width,
    p.Length,
    p.Height
FROM Warehouse w
JOIN Products p ON w.product_id = p.product_id;
```

**3.Write an SQL query to report, How much cubic feet of volume does the inventory occupy in each warehouse.**

**warehouse_name**

**volume**

**Return the result table in any order.**

**The query result format is in the following example.**

**Warehouse table:**

```
+-----------+-------------+------------+
| name      | product_id  | units      |
+-----------+-------------+------------+
| LCHouse1  | 1           | 1          |
| LCHouse1  | 2           | 10         |
| LCHouse1  | 3           | 5          |
| LCHouse2  | 1           | 2          |
| LCHouse2  | 2           | 2          |
| LCHouse3  | 4           | 1          |
+-----------+-------------+------------+
```

**Products table:**

```
+-----------+-------------+-----------+---------+----------+
| product_id | product_name | Width     | Length  | Height   |
+-----------+-------------+-----------+---------+----------+
| 1         | LC-TV       | 5         | 50      | 40       |
| 2         | LC-KeyChain | 5         | 5       | 5        |
| 3         | LC-Phone    | 2         | 10      | 10       |
| 4         | LC-T-Shirt  | 4         | 10      | 20       |
+-----------+-------------+-----------+---------+----------+
```

**Result table:**

```
+---------------+-----------+
| warehouse_name | volume    |
+---------------+-----------+
| LCHouse1      | 12250     |
| LCHouse2      | 20250     |
| LCHouse3      | 800       |
+---------------+-----------+
```

**Volume of product_id = 1 (LC-TV), 5x50x40 = 10000**

**Volume of product_id = 2 (LC-KeyChain), 5x5x5 = 125**

**Volume of product_id = 3 (LC-Phone), 2x10x10 = 200**

**Volume of product_id = 4 (LC-T-Shirt), 4x10x20 = 800**

**LCHouse1: 1 unit of LC-TV + 10 units of LC-KeyChain + 5 units of LC-Phone.**

**Total volume: 1*10000 + 10*125 + 5*200 = 12250 cubic feet**

**LCHouse2: 2 units of LC-TV + 2 units of LC-KeyChain.**

**Total volume: 2*10000 + 2*125 = 20250 cubic feet**

**LCHouse3: 1 unit of LC-T-Shirt.**

**Total volume: 1*800 = 800 cubic feet.**

**Answer:**

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

**3. Spotify Sessions**

**Problem statement**

**Send feedback**

**Table: Playback**

| Column Name | Type |
|---|---|
| session_id | int |
| customer_id | int |

**| start_time | int |**

**| end_time | int |**

**+-------------+------+**

**session_id is the primary key for this table.**

**customer_id is the ID of the customer watching this session.**

**The session runs during the inclusive interval between start_time and end_time.**

**It is guaranteed that start_time <= end_time and that two sessions for the same customer do not intersect.**

**Table: Ads**

**+-------------+------+**

**| Column Name | Type |**

**+-------------+------+**

**| ad_id | int |**

**| customer_id | int |**

**| timestamp | int |**

**+-------------+------+**

**ad_id is the primary key for this table.**

**Customer_id is the ID of the customer viewing this ad.**

**Timestamp is the moment of time at which the ad was shown.**

**Write an SQL query to report all the sessions that did not get shown any ads.**

**Return the result table in any order.**

**The query result format is in the following example:**

**Playback table:**

```
+------------+-------------+------------+----------+
| session_id | customer_id | start_time | end_time |
+------------+-------------+------------+----------+
| 1          | 1           | 1          | 5        |
| 2          | 1           | 15         | 23       |
| 3          | 2           | 10         | 12       |
| 4          | 2           | 17         | 28       |
| 5          | 2           | 2          | 8        |
+------------+-------------+------------+----------+
```

**Ads table:**

```
+-------+-------------+-----------+
| ad_id | customer_id | timestamp |
+-------+-------------+-----------+
| 1     | 1           | 5         |
| 2     | 2           | 17        |
| 3     | 2           | 20        |
+-------+-------------+-----------+
```

**Result table:**

```
+------------+
| session_id |
+------------+
| 2          |
| 3          |
```

**| 5 |**

**+----------+**

**The ad with ID 1 was shown to user 1 at time 5 while they were in session 1.**

**The ad with ID 2 was shown to user 2 at time 17 while they were in session 4.**

**The ad with ID 3 was shown to user 2 at time 20 while they were in session 4.**

**We can see that sessions 1 and 4 had at least one ad. Sessions 2, 3, and 5 did not have any ads, so we return them.**

**Answer:**

```
SELECT p.session_id
FROM Playback p
LEFT JOIN Ads a
ON p.customer_id = a.customer_id
AND a.timestamp BETWEEN p.start_time AND p.end_time
WHERE a.ad_id IS NULL;
```

**3.Problem statement**

Send feedback

Insert below student details in students table and print all data of table.

```
+---------+--------+-------+
| ID  |  Name       | Gender|
+---------+--------+-------+
|   3     |  Kim    |   F   |
|   4     | Molina  |   F   |
|   5     | Dev     |   M   |
+---------+--------+-------+
```

Answer:

```
-- Insert the given student details
INSERT INTO students (ID, Name, Gender) VALUES
(3, 'Kim', 'F'),
(4, 'Molina', 'F'),
```

(5, 'Dev', 'M');

-- Retrieve all data from the student's table

SELECT * FROM students;

**4.Problem statement - IMDb Metacritic Rating**

**Send feedback**

**Print the title and ratings of the movies released in 2012 whose metacritic rating is more than 60 and Domestic collections exceed 10 Crores. (Download the dataset from console)**

SELECT Title, Rating

FROM IMDB

WHERE MetaCritic > 60

AND Budget > 1000000000 -- 10 Crores (assuming budget is in the smallest currency unit)

AND Title LIKE '% (2012) %';

**DATABASE**

**CREATE TABLE IF NOT EXISTS earning (Movie_id Text, Domestic Integer, Worldwide numeric);**

**INSERT INTO earning (Movie_id, Domestic, Worldwide) VALUES ('36809', 56671993, 187733202.0);**

**INSERT INTO earning (Movie_id, Domestic, Worldwide) VALUES ('30114', 18335230, 60738797.0);**

**INSERT INTO earning (Movie_id, Domestic, Worldwide) VALUES ('37367', 35014192, 39187783.0);**

**INSERT INTO earning (Movie_id, Domestic, Worldwide) VALUES ('49473', 15322921, 87100449.0);**

**INSERT INTO earning (Movie_id, Domestic, Worldwide) VALUES ('14867', 6739492, 19839492.0);**

**5.Combine Two Tables**

**Problem statement**

**Send feedback**

**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**

**Solutions:**

**SELECT p.FirstName,**

**p.LastName,**

**a.City,**

**a.State**

**FROM Person p**

**LEFT JOIN Address a ON p.PersonId = a.PersonId;**

**6. Problem statement**

**Send feedback**

**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 |**

**+------+**

**Solutions: correct**

```
SELECT s.name
FROM Salesperson s
WHERE s.sales_id NOT IN (
   SELECT o.sales_id
   FROM Orders o
   JOIN Company c ON o.com_id = c.com_id
   WHERE c.name = 'RED'
);
```

**SQL Query Using NOT EXISTS (More Efficient for Large Data)**

**sql**

```
SELECT s.name
FROM Salesperson s
WHERE NOT EXISTS (
   SELECT 1
   FROM Orders o
   JOIN Company c ON o.com_id = c.com_id
   WHERE o.sales_id = s.sales_id AND c.name = 'RED'
);
```

**7.IMDb Max Weighted Rating**

**Average time to solve is 5m**

**Problem statement**

**Send feedback**

**Print the genre and the maximum weighted rating among all the movies of that genre released in 2014 per genre. (Download the dataset from console)**

**Note:**

**1. Do not print any row where either genre or the weighted rating is empty/null.**

**2. weighted_rating = avgerge of (rating + metacritic/10.0)**

**3. Keep the name of the columns as 'genre' and 'weighted_rating'**

**4. The genres should be printed in alphabetical order.**

**Solution: correct**

```
SELECT g.genre,
    MAX((i.Rating + i.MetaCritic / 10.0) / 2) AS weighted_rating
FROM IMDB i
JOIN genre g ON i.Movie_id = g.Movie_id
WHERE i.Title LIKE '%(2014)%'
  AND g.genre IS NOT NULL
  AND g.genre != ''
  AND i.Rating IS NOT NULL
  AND i.MetaCritic IS NOT NULL
GROUP BY g.genre
ORDER BY g.genre;
```

**8.Rank Scores**

**Problem statement**

Send feedback

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

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

| Id | Score |

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

| 1  | 3.50  |

| 2  | 3.65  |

| 3  | 4.00  |

| 4  | 3.85  |

| 5  | 4.00  |

| 6  | 3.65  |

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

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

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

| score | Rank    |

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

| 4.00  | 1       |

| 4.00  | 1       |

| 3.85  | 2       |

| 3.65  | 3       |

| 3.65  | 3       |

| 3.50  | 4       |

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

**Solutions: wrong / submitted wrong**

```
SELECT Score,
    DENSE_RANK() OVER (ORDER BY Score DESC) AS Rank
FROM Scores;
```

**9.Swap Salary**

**Average time to solve is 2m**

**Problem statement**

**Send feedback**

**Table: Salary**

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

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

**The sex column is ENUM value of type ('m', 'f').**

**The table contains information about an employee.**

**Solution: correct**

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

**10. Director's Actor**

**Problem statement**

**Send feedback**

**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 co-worked 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 co-worked exactly 3 times.**

**Solution : Correct**

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

**11. Article**

**Problem statement**

**Send feedback**

**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 people who viewed more than one article on the same date, sorted in ascending order by their id.**

**The query result format is in the following example:**

**Views table:**

**+------------+-----------+-----------+------------+**

**| article_id | author_id | viewer_id | view_date |**

**+------------+-----------+-----------+------------+**

**| 1 | 3 | 5 | 2019-08-01 |**

**| 3 | 4 | 5 | 2019-08-01 |**

**| 1 | 3 | 6 | 2019-08-02 |**

**| 2 | 7 | 7 | 2019-08-01 |**

**| 2 | 7 | 6 | 2019-08-02 |**

**| 4 | 7 | 1 | 2019-07-22 |**

**| 3 | 4 | 4 | 2019-07-21 |**

**| 3 | 4 | 4 | 2019-07-21 |**

**+------------+-----------+-----------+------------+**

**Result table:**

**+------+**

**| id |**

**+------+**

**| 5 |**

**| 6 |**

**+------+**

**Solution: Correct**

```
SELECT viewer_id AS id
FROM Views
WHERE viewer_id != author_id  -- Exclude self-views if required
GROUP BY viewer_id, view_date
HAVING COUNT (DISTINCT article_id) > 1
ORDER BY id;
```

**12.NPV Queries**

**Average time to solve is 7m**

**Problem statement**

**Send feedback**

**Table: NPV**

**+--------------+---------+**

**| Column Name | Type |**

**+--------------+---------+**

**| id | int |**

**| year | int |**

**| npv | int |**

**+--------------+---------+**

**(id, year) is the primary key of this table.**

**The table has information about the id and the year of each inventory and the corresponding net present value.**

**Table: Queries**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| id           | int     |
| year         | int     |
+--------------+---------+
```

**(id, year) is the primary key of this table.**

**The table has information about the id and the year of each inventory query.**

**Write an SQL query to find the npv of all each query of queries table.**

**Return the result table in any order.**

**The query result format is in the following example:**

**NPV table:**

```
+------+--------+--------+
| id   | year   | npv    |
+------+--------+--------+
| 1    | 2018   | 100    |
| 7    | 2020   | 30     |
| 13   | 2019   | 40     |
| 1    | 2019   | 113    |
| 2    | 2008   | 121    |
| 3    | 2009   | 12     |
| 11   | 2020   | 99     |
| 7    | 2019   | 0      |
```

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

**Queries table:**

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

**| id | year |**

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

**| 1 | 2019 |**

**| 2 | 2008 |**

**| 3 | 2009 |**

**| 7 | 2018 |**

**| 7 | 2019 |**

**| 7 | 2020 |**

**| 13 | 2019 |**

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

**Result table:**

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

**| id | year | npv |**

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

**| 1 | 2019 | 113 |**

**| 2 | 2008 | 121 |**

**| 3 | 2009 | 12 |**

**| 7 | 2018 | 0 |**

**| 7 | 2019 | 0 |**

**| 7 | 2020 | 30 |**

**| 13 | 2019 | 40 |**

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

**The npv value of (7, 2018) is not present in the NPV table, we consider it 0.**

**The npv values of all other queries can be found in the NPV table.**

**Solution:**

```
SELECT q.id,
    q.year,
    COALESCE(n.npv, 0) AS npv
FROM Queries q
LEFT JOIN NPV n
ON q.id = n.id AND q.year = n.year;
```

**13. Customer Placing the Largest Number Orders**

**Average time to solve is 4m**

**Problem statement**

**Send feedback**

**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.**

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

**The query result format is in the following example:**

**Orders table:**

```
+--------------+----------------+
| order_number | customer_number |
+--------------+----------------+
| 1            | 1              |
| 2            | 2              |
| 3            | 3              |
| 4            | 3              |
+--------------+----------------+
```

**Result table:**

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

**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.**

**Solution: correct**

```
SELECT customer_number
FROM Orders
GROUP BY customer_number
ORDER BY COUNT(order_number) DESC
LIMIT 1;
```

**14. The Most Frequently Ordered Products for Each Customer**

**Problem statement**

**Send feedback**

**Table: Customers**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| customer_id  | int     |
| name         | varchar |
+--------------+---------+
```

**customer_id is the primary key for this table.**

**This table contains information about the customers.**

**Table: Orders**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| order_id     | int     |
| order_date   | date    |
| customer_id  | int     |
| product_id   | int     |
+--------------+---------+
```

**order_id is the primary key for this table.**

**This table contains information about the orders made by customer_id.**

**No customer will order the same product more than once in a single day.**

**Table: Products**

**+--------------+---------+**

**| Column Name | Type |**

**+--------------+---------+**

**| product_id | int |**

**| product_name | varchar |**

**| price | int |**

**+--------------+---------+**

**product_id is the primary key for this table.**

**This table contains information about the products.**

**Write an SQL query to find the most frequently ordered product(s) for each customer and return your table order by consumer_id, otherwise your query will not be accepted.**

**The result table should have the product_id and product_name for each customer_id who ordered at least one order. Return the result table in any order.**

**The query result format is in the following example:**

**Customers**

**+-------------+-------+**

**| customer_id | name |**

**+-------------+-------+**

**| 1 | Alice |**

**| 2 | Bob |**

**| 3 | Tom |**

**| 4 | Jerry |**

**| 5 | John |**

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

**Orders**

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

| order_id | order_date | customer_id | product_id |

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

| 1 | 2020-07-31 | 1 | 1 |

| 2 | 2020-07-30 | 2 | 2 |

| 3 | 2020-08-29 | 3 | 3 |

| 4 | 2020-07-29 | 4 | 1 |

| 5 | 2020-06-10 | 1 | 2 |

| 6 | 2020-08-01 | 2 | 1 |

| 7 | 2020-08-01 | 3 | 3 |

| 8 | 2020-08-03 | 1 | 2 |

| 9 | 2020-08-07 | 2 | 3 |

| 10 | 2020-07-15 | 1 | 2 |

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

**Products**

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

| product_id | product_name | price |

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

| 1 | keyboard | 120 |

| 2 | mouse | 80 |

| 3 | screen | 600 |

| 4 | hard disk | 450 |

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

**Result table:**

```
+-------------+------------+-------------+
| customer_id | product_id | product_name |
+-------------+------------+-------------+
| 1           | 2          | mouse       |
| 2           | 1          | keyboard    |
| 2           | 2          | mouse       |
| 2           | 3          | screen      |
| 3           | 3          | screen      |
| 4           | 1          | keyboard    |
+-------------+------------+-------------+
```

**Alice (customer 1) ordered the mouse three times and the keyboard one time, so the mouse is the most frquently ordered product for them.**

**Bob (customer 2) ordered the keyboard, the mouse, and the screen one time, so those are the most frquently ordered products for them.**

**Tom (customer 3) only ordered the screen (two times), so that is the most frquently ordered product for them.**

**Jerry (customer 4) only ordered the keyboard (one time), so that is the most frquently ordered product for them.**

**John (customer 5) did not order anything, so we do not include them in the result table.**

**Solution :**

```
WITH ProductOrderCount AS (
   SELECT o.customer_id,
        o.product_id,
        COUNT(*) AS order_count
   FROM Orders o
   GROUP BY o.customer_id, o.product_id
),
RankedProducts AS (
   SELECT poc.customer_id,
```

```
        poc.product_id,
        p.product_name,
        RANK() OVER (PARTITION BY poc.customer_id ORDER BY poc.order_count
DESC) AS rnk
    FROM ProductOrderCount poc
    JOIN Products p ON poc.product_id = p.product_id
)
SELECT customer_id, product_id, product_name
FROM RankedProducts
WHERE rnk = 1
ORDER BY customer_id;
```

**15.Orders With Maximum Quantity Above Average**

**Problem statement**

**Send feedback**

**Table: OrdersDetails**

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| order_id    | int  |
| product_id  | int  |
| quantity    | int  |
+-------------+------+
```

**(order_id, product_id) is the primary key for this table.**

**A single order is represented as multiple rows, one row for each product in the order.**

**Each row of this table contains the quantity ordered of the product product_id in the order order_id.**

**You are running an ecommerce site that is looking for imbalanced orders. An imbalanced order is one whose maximum quantity is strictly greater than the average quantity of every order (including itself).**

**The average quantity of an order is calculated as (total quantity of all products in the order) / (number of different products in the order). The maximum quantity of an order is the highest quantity of any single product in the order.**

**Write an SQL query to find the order_id of all imbalanced orders.**

**Return the result table in any order.**

**The query result format is in the following example:**

**OrdersDetails table:**

**+----------+------------+----------+**

**| order_id | product_id | quantity |**

**+----------+------------+----------+**

**| 1 | 1 | 12 |**

**| 1 | 2 | 10 |**

**| 1 | 3 | 15 |**

**| 2 | 1 | 8 |**

**| 2 | 4 | 4 |**

**| 2 | 5 | 6 |**

**| 3 | 3 | 5 |**

**| 3 | 4 | 18 |**

**| 4 | 5 | 2 |**

**| 4 | 6 | 8 |**

**| 5 | 7 | 9 |**

**| 5 | 8 | 9 |**

**| 3 | 9 | 20 |**

**| 2 | 9 | 4 |**

**+----------+------------+----------+**

**Result table:**

**+----------+**

**| order_id |**

**+----------+**

**| 1 |**

**| 3 |**

**+----------+**

**The average quantity of each order is:**

**- order_id=1: (12+10+15)/3 = 12.3333333**

**- order_id=2: (8+4+6+4)/4 = 5.5**

**- order_id=3: (5+18+20)/3 = 14.333333**

**- order_id=4: (2+8)/2 = 5**

**- order_id=5: (9+9)/2 = 9**

**The maximum quantity of each order is:**

**- order_id=1: max(12, 10, 15) = 15**

**- order_id=2: max(8, 4, 6, 4) = 8**

**- order_id=3: max(5, 18, 20) = 20**

**- order_id=4: max(2, 8) = 8**

**- order_id=5: max(9, 9) = 9**

**Orders 1 and 3 are imbalanced because they have a maximum quantity that exceeds the average quantity of every order.**

**Solution : correct**

```
WITH OrderStats AS (
   -- Calculate average and max quantity for each order
   SELECT order_id,
        AVG(quantity) AS avg_quantity,
        MAX(quantity) AS max_quantity
   FROM OrdersDetails
   GROUP BY order_id
),
OverallAverage AS (
   -- Calculate overall average quantity across all orders
   SELECT AVG(avg_quantity) AS overall_avg
   FROM OrderStats
)
SELECT os.order_id
FROM OrderStats os, OverallAverage oa
WHERE os.max_quantity > oa.overall_avg;
```

**16.Department Highest Salary(ninja)**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

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

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 70000  | 1            |
```

| 2 | Jim | 90000 | 1 |

| 3 | Henry | 80000 | 2 |

| 4 | Sam | 60000 | 2 |

| 5 | Max | 90000 | 1 |

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

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

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

| Id | Name |

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

| 1 | IT |

| 2 | Sales |

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

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

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

| Department | Employee | Salary |

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

| IT | Max | 90000 |

| IT | Jim | 90000 |

| Sales | Henry | 80000 |

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

**Explanation:**

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

**Solutions :**

```
WITH DepartmentMaxSalary AS (
   -- Find the maximum salary per department
   SELECT DepartmentId, MAX(Salary) AS MaxSalary
   FROM Employee
   GROUP BY DepartmentId
)
SELECT d.Name AS Department,
      e.Name AS Employee,
      e.Salary
FROM Employee e
JOIN DepartmentMaxSalary dms ON e.DepartmentId = dms.DepartmentId AND e.Sal
ary = dms.MaxSalary
JOIN Department d ON e.DepartmentId = d.Id;
```

**17.Duplicate Emails**

**Average time to solve is 2m**

**Problem statement**

**Send feedback**

**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 |**

**+---------+**

**Solutions : correct**

**SELECT Email**

**FROM Person**

**GROUP BY Email**

**HAVING COUNT(*) > 1;**

**18. Triangle Judgement**

**Problem statement**

**Send feedback**

**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       |
```

**Solution: correct**

```
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;
```

**19.Marvel Cities**

**Problem statement**

**Send feedback**

**Query all columns for all Marvel cities in the CITY table with populations larger than 100000. The CountryCode for Marvel is Marv.**

**The CITY table is described as follows:**

```
+---------+--------+
| Field  |  Type    |
+---------+--------+
|   ID    |  Number  |
|   Name  | Varchar  |
|   CountryCode | Varchar  |
|   Population |   Number  |
+---------+--------+-------+
```

**Solution : correct**

```
SELECT *
FROM CITY
WHERE CountryCode = 'Marv'
  AND Population > 100000;
```

**20. Product's Worth Over Invoices**

**Problem statement**

**Send feedback**

**Table: Product**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_id  | int     |
| name        | varchar |
+-------------+---------+
```

**product_id is the primary key for this table.**

**This table contains the ID and the name of the product. The name consists of only lowercase English letters. No two products have the same name.**

**Table: Invoice**

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| invoice_id  | int  |
| product_id  | int  |
| rest        | int  |
| paid        | int  |
```

**| canceled | int |**

**| refunded | int |**

**+------------+------+**

**invoice_id is the primary key for this table and the id of this invoice.**

**product_id is the id of the product for this invoice.**

**rest is the amount left to pay for this invoice.**

**paid is the amount paid for this invoice.**

**canceled is the amount canceled for this invoice.**

**refunded is the amount refunded for this invoice.**

**Write an SQL query that will, for all products, return each product name with total amount due, paid, canceled, and refunded across all invoices.**

**Return the result table ordered by product_name.**

**The query result format is in the following example:**

**Product table:**

**+------------+-------+**

**| product_id | name |**

**+------------+-------+**

**| 0 | ham |**

**| 1 | bacon |**

**+------------+-------+**

**Invoice table:**

**+------------+------------+------+------+----------+----------+**

**| invoice_id | product_id | rest | paid | canceled | refunded |**

**+------------+------------+------+------+----------+----------+**

| 23 | 0 | 2 | 0 | 5 | 0 |

| 12 | 0 | 0 | 4 | 0 | 3 |

| 1 | 1 | 1 | 1 | 0 | 1 |

| 2 | 1 | 1 | 0 | 1 | 1 |

| 3 | 1 | 0 | 1 | 1 | 1 |

| 4 | 1 | 1 | 1 | 1 | 0 |

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

**Result table:**

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

| name | rest | paid | canceled | refunded |

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

| bacon | 3 | 3 | 3 | 3 |

| ham | 2 | 4 | 5 | 3 |

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

**- The amount of money left to pay for bacon is 1 + 1 + 0 + 1 = 3**

**- The amount of money paid for bacon is 1 + 0 + 1 + 1 = 3**

**- The amount of money canceled for bacon is 0 + 1 + 1 + 1 = 3**

**- The amount of money refunded for bacon is 1 + 1 + 1 + 0 = 3**

**- The amount of money left to pay for ham is 2 + 0 = 2**

**- The amount of money paid for ham is 0 + 4 = 4**

**- The amount of money canceled for ham is 5 + 0 = 5**

**- The amount of money refunded for ham is 0 + 3 = 3**

**Solution ; correct**

```
SELECT p.name,
    COALESCE(SUM(i.rest), 0) AS rest,
    COALESCE(SUM(i.paid), 0) AS paid,
    COALESCE(SUM(i.canceled), 0) AS canceled,
    COALESCE(SUM(i.refunded), 0) AS refunded
FROM Product p
LEFT JOIN Invoice i ON p.product_id = i.product_id
```

**GROUP BY p.name**

**ORDER BY p.name;**

**21.IMDb Metacritic Rating**

**Average time to solve is 2m**

**Problem statement**

**Send feedback**

**Print the title and ratings of the movies released in 2012 whose metacritic rating is more than 60 and Domestic collections exceed 10 Crores. (Download the dataset from console)**

**Solution: correct**

```
SELECT i.Title, i.Rating
FROM IMDB i
JOIN earning e ON i.Movie_id = e.Movie_id
WHERE i.MetaCritic > 60
  AND e.Domestic > 100000000
  AND i.Title LIKE '%(2012)%';
```

**22. Department Highest Salary**

**Problem statement**

**Send feedback**

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

```
+----+-------+--------+--------------+
| Id | Name  | Salary | DepartmentId |
+----+-------+--------+--------------+
| 1  | Joe   | 70000  | 1            |
| 2  | Jim   | 90000  | 1            |
| 3  | Henry | 80000  | 2            |
```

| 4 | Sam | 60000 | 2 |

| 5 | Max | 90000 | 1 |

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

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

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

| Id | Name |

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

| 1 | IT |

| 2 | Sales |

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

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

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

| Department | Employee | Salary |

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

| IT | Max | 90000 |

| IT | Jim | 90000 |

| Sales | Henry | 80000 |

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

**Explanation:**

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

**Solutions : wrong answer**

```
SELECT d.Name AS Department, e.Name AS Employee, e.Salary
FROM Employee e
JOIN Department d ON e.DepartmentId = d.Id
WHERE e.Salary = (
   SELECT MAX(Salary)
   FROM Employee
   WHERE DepartmentId = e.DepartmentId
);
```

**22.IMDb Rating**

**Moderate**

**Problem statement**

**Send feedback**

**From the IMDb dataset, print the title and rating of those movies which have a genre starting from 'C' released in 2014 with a budget higher than 4 Crore. (Download the dataset from console)**

**Solutions : correct**

```
SELECT i.Title, i.Rating
FROM IMDB i
JOIN genre g ON i.Movie_id = g.Movie_id
WHERE g.genre LIKE 'C%'
  AND i.Title LIKE '%(2014)%'
  AND i.Budget > 40000000;
```

**23. Rank Scores**

**Problem statement**

**Send feedback**

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

**+----+-------+**

**| Id | Score |**

**+----+-------+**

**| 1 | 3.50 |**

**| 2 | 3.65 |**

**| 3 | 4.00 |**

**| 4 | 3.85 |**

**| 5 | 4.00 |**

**| 6 | 3.65 |**

**+----+-------+**

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

**+-------+---------+**

**| score | Rank |**

**+-------+---------+**

**| 4.00 | 1 |**

**| 4.00 | 1 |**

**| 3.85 | 2 |**

**| 3.65 | 3 |**

**| 3.65 | 3 |**

**| 3.50 | 4 |**

**+-------+---------+**

**Solution : not correct**

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

**24.Second Highest Salary**

**Problem statement**

**Send feedback**

**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.**

```
+--------------------+
| salary             |
+--------------------+
| 200                |
+--------------------+
```

**Solution: correct**

```
SELECT DISTINCT Salary AS salary
FROM Employee
ORDER BY Salary DESC
LIMIT 1 OFFSET 1;
```

**25.Number of Calls Between Two Persons**

**Problem statement**

**Send feedback**

**Table: Calls**

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| from_id     | int     |
| to_id       | int     |
| duration    | int     |
+-------------+---------+
```

**This table does not have a primary key, it may contain duplicates.**

**This table contains the duration of a phone call between from_id and to_id.**

**from_id != to_id**

**Write an SQL query to report the number of calls and the total call duration between each pair of distinct persons (person1, person2) where person1 < person2.**

**Return the result table in any order.**

**The query result format is in the following example:**

**Calls table:**

```
+---------+-------+----------+
| from_id | to_id | duration |
+---------+-------+----------+
| 1       | 2     | 59       |
| 2       | 1     | 11       |
| 1       | 3     | 20       |
| 3       | 4     | 100      |
| 3       | 4     | 200      |
| 3       | 4     | 200      |
| 4       | 3     | 499      |
+---------+-------+----------+
```

**Result table:**

```
+---------+---------+------------+----------------+
| person1 | person2 | call_count | total_duration |
+---------+---------+------------+----------------+
| 1       | 2       | 2          | 70             |
| 1       | 3       | 1          | 20             |
| 3       | 4       | 4          | 999            |
+---------+---------+------------+----------------+
```

**Users 1 and 2 had 2 calls and the total duration is 70 (59 + 11).**

**Users 1 and 3 had 1 call and the total duration is 20.**

**Users 3 and 4 had 4 calls and the total duration is 999 (100 + 200 + 200 + 499).**

**Solution:**

```
SELECT
   LEAST(from_id, to_id) AS person1,
   GREATEST(from_id, to_id) AS person2,
   COUNT(*) AS call_count,
   SUM(duration) AS total_duration
FROM Calls
GROUP BY person1, person2;
```

**26.Shortest Distance**

**Average time to solve is 4m**

**Problem statement**

**Send feedback**

**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 |**

**Solutions :**

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

**27.Create a Session Bar Chart**

**Average time to solve is 5m**

**Problem statement**

**Send feedback**

**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.**

**Solution : correct**

```
select '[0-5>' as bin, count(duration) as total
from Sessions
where duration < 300

union

select '[5-10>', count(duration)
from Sessions
where duration between 300 and 599

union

select '[10-15>', count(duration)
from Sessions
where duration between 600 and 899

union

select '15 or more', count(duration)
from Sessions
where duration >= 900;
```

**28.Department Highest Salary**

**Problem statement**

**Send feedback**

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

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

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

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

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

```
+------------+----------+--------+
| Department | Employee | Salary |
+------------+----------+--------+
| IT         | Max      | 90000  |
```

| IT | Jim | 90000 |

| Sales | Henry | 80000 |

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

**Explanation:**

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

**Solutions:**

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

**29. Delete Duplicate emails**

**Moderate**

**Problem statement**

**Send feedback**

**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.**

**Solutions:**

```
DELETE FROM Person
WHERE Id NOT IN (
   SELECT * FROM (
      SELECT MIN(Id)
      FROM Person
      GROUP BY Email
   ) AS p
);

SELECT * FROM Person;
```

**30. Friend Request**

**Problem statement**

**Send feedback**

**Table: FriendRequest**

```
+---------------+---------+
| Column Name    | Type    |
+---------------+---------+
| sender_id      | int     |
| send_to_id     | int     |
| request_date   | date    |
+---------------+---------+
```

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

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

**Table: RequestAccepted**

**+---------------+---------+**

**| Column Name | Type |**

**+---------------+---------+**

**| requester_id | int |**

**| accepter_id | int |**

**| accept_date | date |**

**+---------------+---------+**

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

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

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

**Note that:**

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

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

**If there are no requests at all, you should return 0.00 as the accept_rate.**

**The query result format is in the following example:**

**FriendRequest table:**

**+----------+-----------+-------------+**

**| sender_id | send_to_id | request_date |**

**+----------+-----------+-------------+**

**| 1 | 2 | 2016/06/01 |**

**| 1 | 3 | 2016/06/01 |**

**| 1 | 4 | 2016/06/01 |**

**| 2 | 3 | 2016/06/02 |**

**| 3 | 4 | 2016/06/09 |**

**+----------+-----------+-------------+**

**RequestAccepted table:**

**+-------------+------------+------------+**

**| requester_id | accepter_id | accept_date |**

**+-------------+------------+------------+**

**| 1 | 2 | 2016/06/03 |**

**| 1 | 3 | 2016/06/08 |**

**| 2 | 3 | 2016/06/08 |**

**| 3 | 4 | 2016/06/09 |**

**| 3 | 4 | 2016/06/10 |**

**+-------------+------------+------------+**

**Result table:**

**+-------------+**

**| unique_accepted_request |**

**+-------------+**

**| 4 |**

**+-------------+**

**+-------------+**

**| total_request |**

**+-------------+**

**| 5 |**

**+-------------+**

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

**Solution : correct**

```
SELECT COUNT(*) AS unique_request
FROM (
   SELECT DISTINCT requester_id, accepter_id
   FROM RequestAccepted
) AS A;


SELECT COUNT(*) AS total_request
FROM (
   SELECT DISTINCT sender_id, send_to_id
   FROM FriendRequest
) AS B;
```

**31. Rank Scores**

**Problem statement**

**Send feedback**

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

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

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

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

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

**| score | Rank |**

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

**| 4.00 | 1 |**

**| 4.00 | 1 |**

**| 3.85 | 2 |**

**| 3.65 | 3 |**

**| 3.65 | 3 |**

**| 3.50 | 4 |**

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

**Solution : correct**

```
SELECT
   s1.Score,
   (SELECT COUNT(DISTINCT Score)
    FROM Scores S2
    WHERE S1.Score <= S2.Score) AS "RANK"
FROM Scores s1
ORDER BY s1.Score DESC;
```

**32. Reformat Department Table**

**Problem statement**

**Send feedback**

**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).**

**Solution : correct**

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

**33.Geography Report**

**Moderate**

**Problem statement**

**Send feedback**

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

**| name | continent |**

**|--------|-----------|**

**| Jack | America |**

**| Pascal | Europe |**

**| Xi | Asia |**

**| Jane | America |**

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

**For the sample input, the output is:**

**| America | Asia | Europe |**

**|---------|------|--------|**

**| Jack | Xi | Pascal |**

**| Jane | | |**

**Solution : correct**

```
WITH RankedStudents AS (
    SELECT
        name,
        continent,
        ROW_NUMBER() OVER (PARTITION BY continent ORDER BY name) AS rn
    FROM student
),
Pivoted AS (
    SELECT
        MAX(CASE WHEN continent = 'America' THEN name END) AS America,
        MAX(CASE WHEN continent = 'Asia' THEN name END) AS Asia,
        MAX(CASE WHEN continent = 'Europe' THEN name END) AS Europe
    FROM RankedStudents
    GROUP BY rn
)
SELECT * FROM Pivoted;
```

**34. Consecutive Available Seats**

**Problem statement**

**Send feedback**

**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 |
| 6 | 1 |

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

| seat_id |
|---|
| 3 |
| 4 |

**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.**

**Solutions : correct**

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

**35. Not Boring Movies**

**Problem statement**

**Send feedback**

**Table: Cinema**

```
+----------------+----------+
| Column Name    | Type     |
+----------------+----------+
| id             | int      |
| movie          | varchar  |
| description    | varchar  |
| rating         | float    |
+----------------+----------+
```

- **id is the primary key for this table.**
- **Each row contains information about the name of a movie, its genre, and its rating.**
- **rating is a 2 decimal places float in the range [0, 10]**

**Write an SQL query to report the movies with an odd-numbered ID and a description that is not "boring".**

**Return the result table in descending order by rating.**

**The query result format is in the following example:**

**Cinema table:**

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

**Result table:**

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

**We have three movies with odd-numbered ID: 1, 3, and 5. The movie with ID = 3 is boring so we don't include it in the answer.**

**Solution : correct**

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

**36. Running Total for Different Genders**

**Problem statement**

**Send feedback**

**Table: Scores**

```
+--------------+---------+
| Column Name   | Type    |
+--------------+---------+
| player_name   | varchar |
| gender        | varchar |
| day           | date    |
| score_points  | int     |
+--------------+---------+
```

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

**A competition is held between females team and males team.**

**Each row of this table indicates that a player_name and with gender has scored score_point in someday.**

**Gender is 'F' if the player is in females team and 'M' if the player is in males team.**

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

**Order the result table by gender and day**

**The query result format is in the following example:**

**Scores table:**

```
+-------------+--------+------------+-------------+
| player_name | gender | day        | score_points |
+-------------+--------+------------+-------------+
| Aron        | F      | 2020-01-01 | 17          |
```

**| Alice | F | 2020-01-07 | 23 |**

**| Bajrang | M | 2020-01-07 | 7 |**

**| Khali | M | 2019-12-25 | 11 |**

**| Slaman | M | 2019-12-30 | 13 |**

**| Joe | M | 2019-12-31 | 3 |**

**| Jose | M | 2019-12-18 | 2 |**

**| Priya | F | 2019-12-31 | 23 |**

**| Priyanka | F | 2019-12-30 | 17 |**

**+-------------+--------+------------+-------------+**

**Result table:**

**+--------+------------+-------+**

**| gender | day | total |**

**+--------+------------+-------+**

**| F | 2019-12-30 | 17 |**

**| F | 2019-12-31 | 40 |**

**| F | 2020-01-01 | 57 |**

**| F | 2020-01-07 | 80 |**

**| M | 2019-12-18 | 2 |**

**| M | 2019-12-25 | 13 |**

**| M | 2019-12-30 | 26 |**

**| M | 2019-12-31 | 29 |**

**| M | 2020-01-07 | 36 |**

**+--------+------------+-------+**

**For females team:**

**First day is 2019-12-30, Priyanka scored 17 points and the total score for the team is 17.**

**Second day is 2019-12-31, Priya scored 23 points and the total score for the team is 40.**

**Third day is 2020-01-01, Aron scored 17 points and the total score for the team is 57.**

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

**For males team:**

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

**Second day is 2019-12-25, Khali scored 11 points and the total score for the team is 13.**

**Third day is 2019-12-30, Slaman scored 13 points and the total score for the team is 26.**

**Fourth day is 2019-12-31, Joe scored 3 points and the total score for the team is 29.**

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

**Solutions :**

```
SELECT
   gender,
   day,
   SUM(score_points) OVER (PARTITION BY gender ORDER BY day) AS total
FROM Scores
ORDER BY gender, day;
```

**37. Count Salary Categories**

**Problem statement**

**Send feedback**

**Table: Accounts**

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| account_id  | int  |
| income      | int  |
+-------------+------+
```

**account_id is the primary key for this table.**

**Each row contains information about the monthly income for one bank account.**

**Write an SQL query to report the number of bank accounts of each salary category. The salary categories are:**

**"Low Salary": All the salaries strictly less than $20000.**

**"Average Salary": All the salaries in the inclusive range [$20000, $50000].**

**"High Salary": All the salaries strictly greater than $50000.**

**The result table must contain all three categories. If there are no accounts in a category, then report 0. Return the result table in any order.**

**The query result format is in the following example.**

**Accounts table:**

**+------------+--------+**

**| account_id | income |**

**+------------+--------+**

**| 3 | 108939 |**

**| 2 | 12747 |**

**| 8 | 87709 |**

**| 6 | 91796 |**

**+------------+--------+**

**Result table:**

**+----------------+----------------+**

**| category | accounts_count |**

**+----------------+----------------+**

**| Low Salary | 1 |**

**| Average Salary | 0 |**

**| High Salary | 3 |**

**+----------------+----------------+**

**Low Salary: Account 2.**

**Average Salary: No accounts.**

**High Salary: Accounts 3, 6, and 8.**

**Solutions :**

```
WITH Categorized AS (
   SELECT
      CASE
         WHEN income < 20000 THEN 'Low Salary'
         WHEN income BETWEEN 20000 AND 50000 THEN 'Average Salary'
         ELSE 'High Salary'
      END AS category
   FROM Accounts
),
AllCategories AS (
   SELECT 'Low Salary' AS category
   UNION ALL
   SELECT 'Average Salary'
   UNION ALL
   SELECT 'High Salary'
)
SELECT
   ac.category,
   COUNT(c.category) AS accounts_count
FROM AllCategories ac
LEFT JOIN Categorized c ON ac.category = c.category
GROUP BY ac.category;
```

**38.Employee Bonus**

**Problem statement**

**Send feedback**

**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 output:**

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

**Solutions : correct**

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

**39. Biggest Single Number**

**Average time to solve is 5m**

**Problem statement**

**Send feedback**

**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.**

**Solution ; correct**

```
SELECT MAX(num) AS num
FROM my_numbers
WHERE num IN (
   SELECT num
   FROM my_numbers
   GROUP BY num
   HAVING COUNT(*) = 1
);
```

**40. Customers Who Bought All Products**

**Moderate**

**Problem statement**

**Send feedback**

**Table: Customer**

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

**product_key is a foreign key to Product table.**

**Table: Product**

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

**| Column Name | Type |**

**+-------------+---------+**

**| product_key | int |**

**+-------------+---------+**

**product_key is the primary key column for this table.**

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

**Return the result table in any order.**

**The query result format is in the following example:**

**Customer table:**

**+-------------+-------------+**

**| customer_id | product_key |**

**+-------------+-------------+**

**| 1 | 5 |**

**| 2 | 6 |**

**| 3 | 5 |**

**| 3 | 6 |**

**| 1 | 6 |**

**+-------------+-------------+**

**Product table:**

**+-------------+**

**| product_key |**

**+-------------+**

**| 5 |**

**| 6 |**

**+-------------+**

**Result table:**

**+-------------+**

**| customer_id |**

**+-------------+**

**| 1 |**

**| 3 |**

**+-------------+**

**The customers who bought all the products (5 and 6) are customers with id 1 and 3.**

**Solutions : correct**

```
SELECT customer_id
FROM Customer
GROUP BY customer_id
HAVING COUNT(DISTINCT product_key) = (SELECT COUNT(*) FROM Product);
```

**41. Recyclable and Low Fat Products**

**Moderate**

**0/80**

**Average time to solve is 5m**

**Problem statement**

**Send feedback**

**Table: Products**

**+-------------+---------+**

**| Column Name | Type |**

**+-------------+---------+**

**| product_id  | int     |**

**| low_fats    | enum    |**

**| recyclable  | enum    |**

**+-------------+---------+**

**product_id is the primary key for this table.**

**low_fats is an ENUM of type ('Y', 'N') where 'Y' means this product is low fat and 'N' means it is not.**

**recyclable is an ENUM of types ('Y', 'N') where 'Y' means this product is recyclable and 'N' means it is not.**

**Write an SQL query to find the ids of products that are both low fat and recyclable.**

**Return the result table in any order.**

**The query result format is in the following example:**

**Products table:**

**+-------------+----------+------------+**

**| product_id  | low_fats | recyclable |**

**+-------------+----------+------------+**

**| 0           | Y        | N          |**

**| 1           | Y        | Y          |**

**| 2           | N        | Y          |**

**| 3           | Y        | Y          |**

**| 4           | N        | N          |**

**+-------------+----------+------------+**

**Result table:**

**+-------------+**

**| product_id  |**

**+-------------+**

**| 1           |**

**| 3           |**

**+-------------+**

**Only products 1 and 3 are both low fat and recyclable.**

**Solution : correct**

```
SELECT product_id
FROM Products
WHERE low_fats = 'Y' AND recyclable = 'Y';
```

**42.  Activity Participants**

**Problem statement**

**Send feedback**

**Table: Friends**

**+---------------+---------+**

**| Column Name   | Type    |**

**+---------------+---------+**

**| id            | int     |**

**| name          | varchar |**

**| activity      | varchar |**

**+---------------+---------+**

**id is the id of the friend and primary key for this table.**

**name is the name of the friend.**

**activity is the name of the activity which the friend takes part in.**

**Table: Activities**

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

**| Column Name | Type |**

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

**| id | int |**

**| name | varchar |**

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

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

**name is the name of the activity.**

**Write an SQL query to find the names of all the activities with neither maximum, nor minimum number of participants.**

**Return the result table in any order. Each activity in table Activities is performed by any person in the table Friends.**

**The query result format is in the following example:**

**Friends table:**

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

**| id | name | activity |**

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

**| 1 | Jonathan D. | Eating |**

**| 2 | Jade W. | Singing |**

**| 3 | Victor J. | Singing |**

**| 4 | Elvis Q. | Eating |**

**| 5 | Daniel A. | Eating |**

**| 6 | Bob B. | Horse Riding |**

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

**Activities table:**

```
+------+-------------+
| id   | name        |
+------+-------------+
| 1    | Eating      |
| 2    | Singing     |
| 3    | Horse Riding |
+------+-------------+
```

**Result table:**

```
+-------------+
| activity    |
+-------------+
| Singing     |
+-------------+
```

**Eating activity is performed by 3 friends, maximum number of participants, (Jonathan D. , Elvis Q. and Daniel A.)**

**Horse Riding activity is performed by 1 friend, minimum number of participants, (Bob B.)**

**Singing is performed by 2 friends (Victor J. and Jade W.)**

**Solutions : correct**

```
WITH ActivityCount AS (
    SELECT activity, COUNT(*) AS participants
    FROM Friends
    GROUP BY activity
),
MaxMinCounts AS (
    SELECT
```

```
    MAX(participants) AS max_count,
    MIN(participants) AS min_count
  FROM ActivityCount
)
SELECT ac.activity
FROM ActivityCount ac, MaxMinCounts mm
WHERE ac.participants NOT IN (mm.max_count, mm.min_count);
```

**43.Classes with more than 5 students**

**Problem statement**

**Send feedback**

**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 |**

+---------+

**Solutions : correct**

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

**44. Employees Earning More Than Their Manager**

**Problem statement**

**Send feedback**

**Employees Earning More Than Their Managers**

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

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

**| Id | Name | Salary | ManagerId |**

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

**| 1 | Joe | 70000 | 3 |**

**| 2 | Henry | 80000 | 4 |**

**| 3 | Sam | 60000 | NULL |**

**| 4 | Max | 90000 | NULL |**

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

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

**+----------+**

**| Employee |**

**+----------+**

**| Joe      |**

**+----------+**

**Solution:**

```
SELECT e.Name AS Employee
FROM Employee e
JOIN Employee m ON e.ManagerId = m.Id
WHERE e.Salary > m.Salary;
```

**45. Number of Comments per Post**

**Moderate**

**80/80**

**Average time to solve is 5m**

**Problem statement**

**Send feedback**

**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.**

**Solutions :**

```
WITH UniquePosts AS (
    SELECT DISTINCT sub_id AS post_id
    FROM Submissions
    WHERE parent_id IS NULL
),
UniqueComments AS (
    SELECT DISTINCT parent_id, sub_id
    FROM Submissions
    WHERE parent_id IS NOT NULL
)
SELECT
    p.post_id,
    COUNT(c.sub_id) AS number_of_comments
```

**FROM UniquePosts p**

**LEFT JOIN UniqueComments c ON p.post_id = c.parent_id**

**GROUP BY p.post_id**

**ORDER BY p.post_id;**

**46.**

**Problem**

**Confirmation of Signups**

**Moderate**

**Problem statement**

**Send feedback**

**Table: Signups**

```
+---------------+----------+
| Column Name   | Type     |
+---------------+----------+
| user_id       | int      |
| time_stamp    | datetime |
+---------------+----------+
```

**user_id is the primary key for this table.**

**Each row contains information about the signup time for the user with ID user_id.**

**Table: Confirmations**

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

**| time_stamp | datetime |**

**| action | ENUM |**

**+----------------+----------+**

**(user_id, time_stamp) is the primary key for this table.**

**user_id is a foreign key with a reference to the Signups table.**

**action is an ENUM of the type ('confirmed', 'timeout')**

**Each row of this table indicates that the user with ID user_id requested a confirmation message at time_stamp and that confirmation message was either confirmed ('confirmed') or expired without confirming ('timeout').**

**The confirmation rate of a user is the number of 'confirmed' messages divided by the total number of requested confirmation messages. The confirmation rate of a user that did not request any confirmation messages is 0. Round the confirmation rate to two decimal places.**

**Write an SQL query to find the confirmation rate of each user.**

**Return the result table in any order.**

**The query result format is in the following example:**

**Signups table:**

**+---------+---------------------+**

**| user_id | time_stamp |**

**+---------+---------------------+**

**| 3 | 2020-03-21 10:16:13 |**

**| 7 | 2020-01-04 13:57:59 |**

**| 2 | 2020-07-29 23:09:44 |**

**| 6 | 2020-12-09 10:39:37 |**

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

**Confirmations table:**

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

| user_id | time_stamp | action |

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

| 3 | 2021-01-06 03:30:46 | timeout |

| 3 | 2021-07-14 14:00:00 | timeout |

| 7 | 2021-06-12 11:57:29 | confirmed |

| 7 | 2021-06-13 12:58:28 | confirmed |

| 7 | 2021-06-14 13:59:27 | confirmed |

| 2 | 2021-01-22 00:00:00 | confirmed |

| 2 | 2021-02-28 23:59:59 | timeout |

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

**Result table**

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

| user_id | confirmation_rate |

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

| 6 | 0.00 |

| 3 | 0.00 |

| 7 | 1.00 |

| 2 | 0.50 |

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

**User 6 did not request any confirmation messages. The confirmation rate is 0.**

**User 3 made 2 requests and both timed out. The confirmation rate is 0.**

**User 7 made 3 requests and all were confirmed. The confirmation rate is 1.**

**User 2 made 2 requests where one was confirmed and the other timed out. The confirmation rate is 1 / 2 = 0.5.**

**Solutions : correct**

**SELECT s.user_id, ROUND(AVG(CASE WHEN action_value = 'confirmed' THEN 1.00 ELSE 0.00 END),2) AS confirmation_rate FROM Signups s LEFT JOIN Confirmations c ON s.user_id = c.user_id GROUP BY s.user_id**

**47. The Latest Login in 2020**

**Moderate**

**Problem statement**

**Send feedback**

**Table: Logins**

**+----------------+----------+**

**| Column Name | Type |**

**+----------------+----------+**

**| user_id | int |**

**| time_stamp | datetime |**

**+----------------+----------+**

**(user_id, time_stamp) is the primary key for this table.**

**Each row contains information about the login time for the user with ID user_id.**

**Write an SQL query to report the latest login for all users in the year 2020. Do not include the users who did not login in 2020.**

**Return the result table in any order.**

**The query result format is in the following example:**

**Logins table:**

**+---------+---------------------+**

**| user_id | time_stamp          |**

**+---------+---------------------+**

**| 6       | 2020-06-30 15:06:07 |**

**| 6       | 2021-04-21 14:06:06 |**

**| 6       | 2019-03-07 00:18:15 |**

**| 8       | 2020-02-01 05:10:53 |**

**| 8       | 2020-12-30 00:46:50 |**

**| 2       | 2020-01-16 02:49:50 |**

**| 2       | 2019-08-25 07:59:08 |**

**| 14      | 2019-07-14 09:00:00 |**

**| 14      | 2021-01-06 11:59:59 |**

**+---------+---------------------+**

**Result table:**

**+---------+---------------------+**

**| user_id | last_stamp          |**

**+---------+---------------------+**

**| 6       | 2020-06-30T15:06:07Z |**

**| 8       | 2020-12-30T00:46:50Z |**

**| 2       | 2020-01-16T02:49:50Z |**

**+---------+---------------------+**

**User 6 logged into their account 3 times but only once in 2020, so we include this login in the result table.**

**User 8 logged into their account 2 times in 2020, once in February and once in December. We include only the latest one (December) in the result table.**

**User 2 logged into their account 2 times but only once in 2020, so we include this login in the result table.**

**User 14 did not login in 2020, so we do not include them in the result table.**

**48.Problem**

**Customers Who Never Order**

**Moderate**

**0/80**

**Average time to solve is 4m**

**Problem statement**

**Send feedback**

**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 | NameCust  |
+----+-------+
| 1  | Joe   |
| 2  | Henry |
| 3  | Sam   |
| 4  | Max   |
+----+-------+
```

**Table: Orders.**

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

**+----+------------+**

**Using the above tables as example, return the following:**

**+-----------+**

**| Customers |**

**+-----------+**

**| Henry     |**

**| Max       |**

**+-----------+**

**Solution:**

**49. Apples & Oranges**

**Moderate**

**0/80**

**Average time to solve is 7m**

**Problem statement**

**Send feedback**

**Table: Sales**

**+--------------+---------+**

**| Column Name   | Type    |**

**+--------------+---------+**

**| sale_date     | date    |**

**| fruit         | enum    |**

**| sold_num      | int     |**

**+--------------+---------+**

**(sale_date,fruit) is the primary key for this table.**

**This table contains the sales of "apples" and "oranges" sold each day.**

**Write an SQL query to report the difference between number of apples and oranges sold each day.**

**Return the result table ordered by sale_date in format ('YYYY-MM-DD').**

**The query result format is in the following example:**

**Sales table:**

```
+------------+------------+-------------+
| sale_date  | fruit      | sold_num    |
+------------+------------+-------------+
| 2020-05-01 | apples     | 10          |
| 2020-05-01 | oranges    | 8           |
| 2020-05-02 | apples     | 15          |
| 2020-05-02 | oranges    | 15          |
| 2020-05-03 | apples     | 20          |
| 2020-05-03 | oranges    | 0           |
| 2020-05-04 | apples     | 15          |
| 2020-05-04 | oranges    | 16          |
+------------+------------+-------------+
```

**Result table:**

```
+------------+--------------+
| sale_date  | diff         |
+------------+--------------+
| 2020-05-01 | 2            |
| 2020-05-02 | 0            |
| 2020-05-03 | 20           |
```

**| 2020-05-04 | -1 |**

**+------------+-------------+**

**Day 2020-05-01, 10 apples and 8 oranges were sold (Difference 10 - 8 = 2).**

**Day 2020-05-02, 15 apples and 15 oranges were sold (Difference 15 - 15 = 0).**

**Day 2020-05-03, 20 apples and 0 oranges were sold (Difference 20 - 0 = 20).**

**Day 2020-05-04, 15 apples and 16 oranges were sold (Difference 15 - 16 = -1).**

**Solution:**

```
SELECT
   sale_date,
   SUM(CASE WHEN fruit = 'apples' THEN sold_num ELSE 0 END) -
   SUM(CASE WHEN fruit = 'oranges' THEN sold_num ELSE 0 END) AS diff
FROM Sales
GROUP BY sale_date
ORDER BY sale_date;
```

**50. Customers Who Never Order**

**Moderate**

**0/80**

**Average time to solve is 4m**

**Problem statement**

**Send feedback**

**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 | NameCust |**

**+----+-------+**

**| 1 | Joe |**

**| 2 | Henry |**

**| 3 | Sam |**

**| 4 | Max |**

**+----+-------+**

**Table: Orders.**

**+----+------------+**

**| Id | CustomerId |**

**+----+------------+**

**| 1 | 3 |**

**| 2 | 1 |**

**+----+------------+**

**Using the above tables as example, return the following:**

**+-----------+**

**| Customers |**

**+-----------+**

**| Henry |**

**| Max |**

**+-----------+**

**Solution : correct**

**SELECT customers.NameCust AS "Customers"**

**FROM customers**

**WHERE customers.id NOT IN (SELECT CustomerId FROM Orders);**

**51. IMDb Genre**

**Hard**

**0/120**

**Average time to solve is 5m**

**Problem statement**

**Send feedback**

**Print the genre and the maximum net profit among all the movies of that genre released in 2012 per genre. (Download the dataset from console)**

**Note:**

**1. Do not print any row where either genre or the net profit is empty/null.**

**2. net_profit = Domestic + Worldwide - Budget**

**3. Keep the name of the columns as 'genre' and 'net_profit'**

**4. The genres should be printed in alphabetical order.**

**Solution:**

```
SELECT
   g.genre,
   MAX(e.Domestic + e.Worldwide - i.Budget) AS net_profit
FROM genre g
JOIN earning e ON g.Movie_id = e.Movie_id
JOIN IMDB i ON g.Movie_id = i.Movie_id
WHERE g.genre IS NOT NULL
  AND (e.Domestic + e.Worldwide - i.Budget) IS NOT NULL
  AND i.Title LIKE '%(2012)%'
GROUP BY g.genre
ORDER BY g.genre;
```

**52. Rising Temperature**

**Hard**

**0/120**

**Average time to solve is 6m**

**Problem statement**

**Send feedback**

**Table: Weather**

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

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

**This table contains information about the temperature in a certain day.**

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

**Return the result table in any order.**

**The query result format is in the following example:**

**Weather**

```
+----+------------+-------------+
| id | recordDate | Temperature |
+----+------------+-------------+
| 1  | 2015-01-01 | 10          |
```

**| 2 | 2015-01-02 | 25 |**

**| 3 | 2015-01-03 | 20 |**

**| 4 | 2015-01-04 | 30 |**

**+----+------------+-------------+**

**Result table:**

**+----+**

**| Id |**

**+----+**

**| 2 |**

**| 4 |**

**+----+**

**In 2015-01-02, temperature was higher than the previous day (10 -> 25).**

**In 2015-01-04, temperature was higher than the previous day (20 -> 30).**

**Solution: output correct / wrong answer**

```
SELECT w1.id
FROM Weather w1
JOIN Weather w2
   ON w1.recordDate = w2.recordDate + INTERVAL '1 day'
WHERE w1.temperature > w2.temperature;
```

**53. Consecutive Numbers**

**Hard**

**0/120**

**Average time to solve is 8m**

**Problem statement**

**Send feedback**

**Table: Logs**

**+-------------+---------+**

**| Column Name | Type    |**

**+-------------+---------+**

**| id          | int     |**

**| num         | varchar |**

**+-------------+---------+**

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

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

**Return the result table in any order.**

**The query result format is in the following example:**

**Logs table:**

**+----+-----+**

**| Id | Num |**

**+----+-----+**

**| 1  | 1   |**

**| 2  | 1   |**

**| 3  | 1   |**

**| 4  | 2   |**

**| 5  | 1   |**

**| 6  | 2   |**

**| 7  | 2   |**

**+----+-----+**

**Result table:**

**+-----------------+**

**| ConsecutiveNums |**

**+----------------+**

**| 1 |**

**+----------------+**

**1 is the only number that appears consecutively for at least three times.**

**54. Consecutive Numbers**

**Hard**

**0/120**

**Average time to solve is 8m**

**Problem statement**

**Send feedback**

**Table: Logs**

**+-------------+---------+**

**| Column Name | Type |**

**+-------------+---------+**

**| id | int |**

**| num | varchar |**

**+-------------+---------+**

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

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

**Return the result table in any order.**

**The query result format is in the following example:**

**Logs table:**

```
+----+-----+
| Id | Num |
+----+-----+
| 1  | 1   |
| 2  | 1   |
| 3  | 1   |
| 4  | 2   |
| 5  | 1   |
| 6  | 2   |
| 7  | 2   |
+----+-----+
```

**Result table:**

```
+-----------------+
| ConsecutiveNums |
+-----------------+
| 1               |
+-----------------+
```

**1 is the only number that appears consecutively for at least three times.**

**Solution : correct**

```
SELECT DISTINCT l1.num AS ConsecutiveNums
FROM Logs l1
JOIN Logs l2 ON l1.id = l2.id - 1
JOIN Logs l3 ON l2.id = l3.id - 1
WHERE l1.num = l2.num AND l2.num = l3.num;
```

**55. Top Travellers**

**Hard**

**Problem statement**

**Send feedback**

**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.**

**Solutions :**

```
SELECT
    u.name,
    COALESCE(SUM(r.distance), 0) AS travelled_distance
FROM Users u
LEFT JOIN Rides r
    ON u.id = r.user_id
GROUP BY u.id, u.name
ORDER BY travelled_distance DESC, u.name ASC;
```

**55. Biggest Window Between Visits**

**Average time to solve is 15m**

**Problem statement**

**Send feedback**

**Table: UserVisits**

**+-------------+------+**

**| Column Name | Type |**

**+-------------+------+**

**| user_id | int |**

**| visit_date | date |**

**+-------------+------+**

- **This table does not have a primary key.**
- **This table contains logs of the dates that users vistied a certain retailer.**

**Assume today's date is '2021-1-1'.**

**Write an SQL query that will, for each user_id, find out the largest window of days between each visit and the one right after it (or today if you are considering the last visit).**

**Return the result table ordered by user_id.**

**The query result format is in the following example: UserVisits table:**

**+---------+------------+**

**| user_id | visit_date |**

**+---------+------------+**

**| 1 | 2020-11-28 |**

**| 1 | 2020-10-20 |**

**| 1 | 2020-12-3 |**

**| 2 | 2020-10-5 |**

**| 2 | 2020-12-9 |**

**| 3 | 2020-11-11 |**

**+---------+------------+**

**Result table:**

**+---------+---------------+**

**| user_id | biggest_window|**

**+---------+---------------+**

**| 1 | 39 |**

**| 2 | 65 |**

**| 3 | 51 |**

**+---------+---------------+**

**Explanation:**

- **For the first user, the windows in question are between dates:**

1. **2020-10-20 and 2020-11-28 with a total of 39 days.**
2. **2020-11-28 and 2020-12-3 with a total of 5 days.**
3. **2020-12-3 and 2021-1-1 with a total of 29 days.**

- **Making the biggest window the one with 39 days.**
- **For the second user, the windows in question are between dates:**

1. **2020-10-5 and 2020-12-9 with a total of 65 days.**
2. **2020-12-9 and 2021-1-1 with a total of 23 days.**

- **Making the biggest window the one with 65 days.**
- **For the third user, the only window in question is between dates 2020-11-11 and 2021-1-1 with a total of 51 days.**

**Solution: correct**

**WITH visit_dates AS (**

```
    -- Include all visits and append '2021-01-01' as the next visit for the last actual visit
    SELECT
        user_id,
        visit_date
    FROM UserVisits
    UNION ALL
    SELECT DISTINCT
        user_id,
        DATE '2021-01-01' AS visit_date
    FROM UserVisits
),
ranked_visits AS (
    -- Rank the visits for each user to establish an order
    SELECT
        user_id,
        visit_date,
        ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY visit_date) AS vis
it_rank
    FROM visit_dates
),
visit_windows AS (
    -- Join each visit with its next visit to calculate the window
    SELECT
        v1.user_id,
        v1.visit_date AS current_visit,
        v2.visit_date AS next_visit,
        v2.visit_date - v1.visit_date AS window_days
    FROM ranked_visits v1
    LEFT JOIN ranked_visits v2
        ON v1.user_id = v2.user_id AND v1.visit_rank = v2.visit_rank - 1
)
```

```
-- Select the biggest window per user
SELECT
    user_id,
    MAX(window_days) AS biggest_window
FROM visit_windows
WHERE next_visit IS NOT NULL
GROUP BY user_id
ORDER BY user_id;
```

**56. Find the Team Size**

**Problem statement**

**Send feedback**

**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.

Solution :

```
SELECT
    e.employee_id,
    team_sizes.team_size
FROM Employee e
```

```
JOIN (
    -- Calculate team size for each team
    SELECT team_id, COUNT(*) AS team_size
    FROM Employee
    GROUP BY team_id
) AS team_sizes
ON e.team_id = team_sizes.team_id;
```

**57.Winning Candidate**

**Average time to solve is 7m**

**Problem statement**

**Send feedback**

**Table: Candidate**

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

**Table: Vote**

```
+-----+-------------+
| id  | CandidateId |
+-----+-------------+
```

| 1 | 2 |

| 2 | 4 |

| 3 | 3 |

| 4 | 2 |

| 5 | 5 |

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

**The id column in both tables is an auto-incrementing primary key.**

**The CandidateId column in the Vote table refers to the id column in the Candidate table.**

**Task:**

**Write an SQL query to find the name of the candidate who received the most votes. In the event of the above example, the query should return:**

+------+

| Name |

+------+

| B |

+------+

**Notes:**

**You may assume that there is no tie; in other words, there will be exactly one candidate with the most votes.**

**Solutions : output correct / wrong answer**

```
SELECT c.Name
FROM Candidate c
JOIN (
    -- Count the votes for each candidate and select the one with the highest votes
    SELECT CandidateId
    FROM Vote
    GROUP BY CandidateId
    ORDER BY COUNT(*) DESC
```

```
    LIMIT 1
) AS v ON c.id = v.CandidateId;
```

**58. Exchange Seats**

**Hard**

**0/120**

**Average time to solve is 8m**

**Problem statement**

**Send feedback**

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

**The column id is continuous increment.**

**Mary wants to change seats for the adjacent students.**

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

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

| 5 | Jeames |

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

**For the sample input, the output is:**

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

| id | student |

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

| 1 | Doris |

| 2 | Abbot |

| 3 | Green |

| 4 | Emerson |

| 5 | Jeames |

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

**Note:**

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

**Solution : correct**

```
SELECT
   CASE
      WHEN id % 2 = 1 AND id + 1 <= (SELECT MAX(id) FROM seat) THEN id + 1
      WHEN id % 2 = 0 THEN id - 1
      ELSE id
   END AS id,
   student
FROM seat
ORDER BY id;
```

**59. Find the Missing IDs**

**Ninja**

**0/200**

**Average time to solve is 15m**

**Problem statement**

**Send feedback**

**Table: Customers**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| customer_id  | int     |
| customer_name | varchar |
+--------------+---------+
```

**customer_id is the primary key for this table.**

**Each row of this table contains the name and the id customer.**

**Write an SQL query to find the missing customer IDs. The missing IDs are ones that are not in the Customers table but are in the range between 1 and the maximum customer_id present in the table.**

**Notice that the maximum customer_id will not exceed 100.**

**Return the result table ordered by ids in ascending order.**

**The query result format is in the following example.**

**Customers table:**

**+-------------+---------------+**

**| customer_id | customer_name |**

**+-------------+---------------+**

**| 1 | Alice |**

**| 4 | Bob |**

**| 5 | Charlie |**

**+-------------+---------------+**

**Result table:**

**+-----+**

**| ids |**

**+-----+**

**| 2 |**

**| 3 |**

**+-----+**

**The maximum customer_id present in the table is 5, so in the range [1,5], IDs 2 and 3 are missing from the table.**

**Solution :**

```
WITH all_ids AS (
   -- Generate a sequence of numbers from 1 to the maximum customer_id
   SELECT generate_series(1, (SELECT MAX(customer_id) FROM Customers)) AS id
),
missing_ids AS (
   -- Identify IDs not present in the Customers table
   SELECT id
   FROM all_ids
   WHERE id NOT IN (SELECT customer_id FROM Customers)
)
-- Final result ordered by ascending IDs
SELECT id AS ids
FROM missing_ids
```

**ORDER BY ids;**

**60. Managers with at Least 5 Direct Reports**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

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

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

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

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

**Note:**

**No one would report to himself**

**Solutions; correct**

```
SELECT
    e.Name
FROM Employee e
JOIN (
    -- Count direct reports for each manager
    SELECT ManagerId
    FROM Employee
    WHERE ManagerId IS NOT NULL
    GROUP BY ManagerId
    HAVING COUNT(*) >= 5
) AS mgr
ON e.Id = mgr.ManagerId;
```

**61. Bank Account Summary**

**Problem statement**

**Send feedback**

**Table: Users**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| user_id      | int     |
| user_name    | varchar |
| credit       | int     |
+--------------+---------+
```

**user_id is the primary key for this table.**

**Each row of this table contains the current credit information for each user.**

**Table: Transactions**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| trans_id     | int     |
| paid_by      | int     |
| paid_to      | int     |
| amount       | int     |
| transacted_on | date   |
+--------------+---------+
```

**trans_id is the primary key for this table.**

**Each row of this table contains the information about the transaction in the bank.**

**User with id (paid_by) transfer money to user with id (paid_to).**

**Codestudio Bank (CSB) helps its coders in making virtual payments. Our bank records all transactions in the table Transaction, we want to find out the current balance of all users and check wheter they have breached their credit limit (If their current credit is less than 0).**

**Write an SQL query to report.**

**user_id**

**user_name**

**credit, current balance after performing transactions.**

**credit_limit_breached, check credit_limit ("Yes" or "No")**

**Return the result table in any order.**

**The query result format is in the following example.**

**Users table:**

```
+------------+--------------+------------+
| user_id    | user_name    | credit     |
+------------+--------------+------------+
| 1          | Moustafa     | 100        |
| 2          | Jonathan     | 200        |
| 3          | Winston      | 10000      |
| 4          | Luis         | 800        |
+------------+--------------+------------+
```

**Transactions table:**

```
+------------+------------+------------+----------+--------------+
| trans_id   | paid_by    | paid_to    | amount   | transacted_on |
+------------+------------+------------+----------+--------------+
| 1          | 1          | 3          | 400      | 2020-08-01    |
| 2          | 3          | 2          | 500      | 2020-08-02    |
| 3          | 2          | 1          | 200      | 2020-08-03    |
+------------+------------+------------+----------+--------------+
```

**Result table:**

```
+------------+------------+------------+---------------------+
| user_id    | user_name  | credit     | credit_limit_breached |
+------------+------------+------------+---------------------+
| 1          | Moustafa   | -100       | Yes                 |
| 2          | Jonathan   | 500        | No                  |
| 3          | Winston    | 9900       | No                  |
```

**| 4 | Luis | 800 | No |**

**+----------+----------+----------+--------------------+**

**Moustafa paid $400 on "2020-08-01" and received $200 on "2020-08-03", credit (100 - 400 +200) = -$100**

**Jonathan received $500 on "2020-08-02" and paid $200 on "2020-08-08", credit (200 +500 -200) = $500**

**Winston received $400 on "2020-08-01" and paid $500 on "2020-08-03", credit (10000 +400 -500) = $9990**

**Luis didn't received any transfer, credit = $800**

**Solution :**

```
WITH transaction_summary AS (
    -- Calculate the net transaction for each user (paid and received)
    SELECT
        paid_by AS user_id,
        SUM(-amount) AS net_amount
    FROM Transactions
    GROUP BY paid_by
    UNION ALL
    SELECT
        paid_to AS user_id,
        SUM(amount) AS net_amount
    FROM Transactions
    GROUP BY paid_to
),
user_balance AS (
    -- Calculate the final credit after transactions
    SELECT
        u.user_id,
        u.user_name,
        u.credit + COALESCE(SUM(t.net_amount), 0) AS credit
    FROM Users u
```

```
    LEFT JOIN transaction_summary t
        ON u.user_id = t.user_id
    GROUP BY u.user_id, u.user_name, u.credit
)
-- Final selection with credit limit check and ordering by user_id
SELECT
    user_id,
    user_name,
    credit,
    CASE
        WHEN credit < 0 THEN 'Yes'
        ELSE 'No'
    END AS credit_limit_breached
FROM user_balance
ORDER BY user_id;
```

**62. Premier League Stats**

**Ninja**

**0/200**

**Average time to solve is 20m**

**Problem statement**

**Send feedback**

**Table: Teams**

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

**| team_name | varchar |**

**+---------------+---------+**

**team_id is the primary key for this table.**

**Each row contains information about one team in the league.**

**Table: Matches**

**+----------------+---------+**

**| Column Name | Type |**

**+----------------+---------+**

**| home_team_id | int |**

**| away_team_id | int |**

**| home_team_goals | int |**

**| away_team_goals | int |**

**+----------------+---------+**

**(home_team_id, away_team_id) is the primary key for this table.**

**Each row contains information about one match.**

**home_team_goals is the number of goals scored by the home team.**

**away_team_goals is the number of goals scored by the away team.**

**The winner of the match is the team with the higher number of goals.**

**Write an SQL query to report the statistics of the league. The statistics should be built using the played matches where the winning team gets three points and the losing team gets no points. If a match ends with a draw, both teams get one point.**

**Each row of the result table should contain:**

**team_name - The name of the team in the Teams table.**

**matches_played - The number of matches played as either a home or away team.**

**points - The total points the team has so far.**

**goal_for - The total number of goals scored by the team across all matches.**

**goal_against - The total number of goals scored by opponent teams against this team across all matches.**

**goal_diff - The result of goal_for - goal_against.**

**Return the result table in descending order by points. If two or more teams have the same points, order them in descending order by goal_diff. If there is still a tie, order them by team_name in lexicographical order.**

**The query result format is in the following example:**

**Teams table:**

**+---------+-----------+**

**| team_id | team_name |**

**+---------+-----------+**

**| 1 | Ajax |**

**| 4 | Dortmund |**

**| 6 | Arsenal |**

**+---------+-----------+**

**Matches table:**

**+--------------+--------------+----------------+----------------+**

**| home_team_id | away_team_id | home_team_goals | away_team_goals |**

**+--------------+--------------+----------------+----------------+**

**| 1 | 4 | 0 | 1 |**

**| 1 | 6 | 3 | 3 |**

**| 4 | 1 | 5 | 2 |**

**| 6 | 1 | 0 | 0 |**

**+--------------+--------------+----------------+----------------+**

**Result table:**

```
+-----------+---------------+--------+----------+-------------+-----------+
| team_name | matches_played | points | goal_for | goal_against | goal_diff |
+-----------+---------------+--------+----------+-------------+-----------+
| Dortmund  | 2             | 6      | 6        | 2           | 4         |
| Arsenal   | 2             | 2      | 3        | 3           | 0         |
| Ajax      | 4             | 2      | 5        | 9           | -4        |
+-----------+---------------+--------+----------+-------------+-----------+
```

**Ajax (team_id=1) played 4 matches: 2 losses and 2 draws. Total points = 0 + 0 + 1 + 1 = 2.**

**Dortmund (team_id=4) played 2 matches: 2 wins. Total points = 3 + 3 = 6.**

**Arsenal (team_id=6) played 2 matches: 2 draws. Total points = 1 + 1 = 2.**

**Dortmund is the first team in the table. Ajax and Arsenal have the same points, but since Arsenal has a higher goal_diff than Ajax, Arsenal comes before Ajax in the table.**

**Solution : correct**

```
WITH match_stats AS (
  -- Calculate match-wise stats for each team (both home and away)
  SELECT
    home_team_id AS team_id,
    home_team_goals AS goals_for,
    away_team_goals AS goals_against,
    CASE
      WHEN home_team_goals > away_team_goals THEN 3
      WHEN home_team_goals = away_team_goals THEN 1
      ELSE 0
    END AS points
  FROM Matches
  UNION ALL
```

```
    SELECT
        away_team_id AS team_id,
        away_team_goals AS goals_for,
        home_team_goals AS goals_against,
        CASE
            WHEN away_team_goals > home_team_goals THEN 3
            WHEN away_team_goals = home_team_goals THEN 1
            ELSE 0
        END AS points
    FROM Matches
),
team_stats AS (
    -- Aggregate the stats for each team
    SELECT
        team_id,
        COUNT(*) AS matches_played,
        SUM(points) AS points,
        SUM(goals_for) AS goal_for,
        SUM(goals_against) AS goal_against,
        SUM(goals_for - goals_against) AS goal_diff
    FROM match_stats
    GROUP BY team_id
)
-- Join with Teams table to get team names and order the result as required
SELECT
    t.team_name,
    COALESCE(ts.matches_played, 0) AS matches_played,
    COALESCE(ts.points, 0) AS points,
    COALESCE(ts.goal_for, 0) AS goal_for,
    COALESCE(ts.goal_against, 0) AS goal_against,
```

```
    COALESCE(ts.goal_diff, 0) AS goal_diff
FROM Teams t
LEFT JOIN team_stats ts
    ON t.team_id = ts.team_id
ORDER BY
    points DESC,
    goal_diff DESC,
    team_name ASC;
```

**63. Count Student Number in Departments**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

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

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

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

**The student is described as follow:**

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

**| dept_id | Integer |**

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

**And the department table is described as below:**

**| Column Name | Type |**

**|---|---|**

**| dept_id | Integer |**

**| dept_name | String |**

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

**Here is an example input:**

**student table:**

**| student_id | student_name | gender | dept_id |**

**|---|---|---|---|**

**| 1 | Jack | M | 1 |**

**| 2 | Jane | F | 1 |**

**| 3 | Mark | M | 2 |**

**department table:**

**| dept_id | dept_name |**

**|---|---|**

**| 1 | Engineering |**

**| 2 | Science |**

**| 3 | Law |**

**The Output should be:**

**| dept_name | student_number |**

**|------------|---------------|**

**| Engineering | 2 |**

**| Science | 1 |**

**| Law | 0 |**

**Solution: correct**

```
SELECT
   d.dept_name,
   COUNT(s.student_id) AS student_number
FROM department d
LEFT JOIN student s
   ON d.dept_id = s.dept_id
GROUP BY d.dept_name
ORDER BY student_number DESC, d.dept_name ASC;
```

**64. Investments in 2016**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

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

**Have the same TIV_2015 value as one or more other policyholders.**

**Are not located in the same city as any other policyholder (i.e.: the (latitude, longitude) attribute pairs must be unique).**

**Input Format:**

**The insurance table is described as follows:**

**| Column Name | Type |**

**|------------|--------------|**

**| PID | INTEGER(11) |**

**| TIV_2015 | NUMERIC(15,2) |**

**| TIV_2016 | NUMERIC(15,2) |**

**| LAT | NUMERIC(5,2) |**

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

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

**Sample Input**

**| PID | TIV_2015 | TIV_2016 | LAT | LON |**

**|-----|----------|----------|-----|-----|**

**| 1 | 10 | 5 | 10 | 10 |**

**| 2 | 20 | 20 | 20 | 20 |**

**| 3 | 10 | 30 | 20 | 20 |**

**| 4 | 10 | 40 | 40 | 40 |**

**Sample Output**

**| TIV_2016 |**

**|----------|**

**| 45.00 |**

**Explanation**

**The first record in the table, like the last record, meets both of the two criteria.**

**The TIV_2015 value '10' is as the same as the third and forth record, and its location unique.**

**The second record does not meet any of the two criteria. Its TIV_2015 is not like any other policyholders.**

**And its location is the same with the third record, which makes the third record fail, too.**

**So, the result is the sum of TIV_2016 of the first and last record, which is 45.**

**Solution : correct**

```
WITH tiv_2015_duplicates AS (
    -- Identify policyholders with duplicate TIV_2015 values
    SELECT TIV_2015
    FROM insurance
    GROUP BY TIV_2015
    HAVING COUNT(*) > 1
),
unique_locations AS (
    -- Identify unique (LAT, LON) combinations
    SELECT LAT, LON
    FROM insurance
    GROUP BY LAT, LON
    HAVING COUNT(*) = 1
)
-- Final selection of policyholders meeting both criteria
SELECT
    ROUND(SUM(TIV_2016), 2) AS TIV_2016
FROM insurance
WHERE TIV_2015 IN (SELECT TIV_2015 FROM tiv_2015_duplicates)
  AND (LAT, LON) IN (SELECT LAT, LON FROM unique_locations);
```

**65. Tree Node**

**Ninja**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

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

**+----+------+**

**| id | p_id |**

**+----+------+**

**| 1 | null |**

**| 2 | 1 |**

**| 3 | 1 |**

**| 4 | 2 |**

**| 5 | 2 |**

**+----+------+**

**Each node in the tree can be one of three types:**

**Leaf: if the node is a leaf node.**

**Root: if the node is the root of the tree.**

**Inner: If the node is neither a leaf node nor a root node.**

**Write a query to print the node id and the type of the node. Sort your output by the node id. The result for the above sample is:**

**+----+------+**

**| id | Type |**

**+----+------+**

**| 1 | Root |**

**| 2 | Inner|**

**| 3 | Leaf |**

**| 4 | Leaf |**

**| 5 | Leaf |**

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

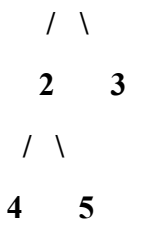
**Explanation**

**Node '1' is root node, because its parent node is NULL and it has child node '2' and '3'.**

**Node '2' is inner node, because it has parent node '1' and child node '4' and '5'.**

**Node '3', '4' and '5' is Leaf node, because they have parent node and they don't have child node.**

**And here is the image of the sample tree as below:**




**Note**

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

**Solution: correct**

```
SELECT
    t1.id,
    CASE
        WHEN t1.p_id IS NULL THEN 'Root'
        WHEN t1.id IN (SELECT DISTINCT p_id FROM tree WHERE p_id IS NOT NU
LL) THEN 'Inner'
        ELSE 'Leaf'
    END AS Type
```

**FROM tree t1**

**ORDER BY t1.id;**

**66. Immediate Food Delivery**

**Average time to solve is 8m**

**Problem statement**

**Send feedback**

**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.**

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

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

**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           | 2           | 2019-08-02 | 2019-08-02                 |
| 3           | 1           | 2019-08-11 | 2019-08-12                 |
| 4           | 3           | 2019-08-24 | 2019-08-24                 |
| 5           | 3           | 2019-08-21 | 2019-08-22                 |
| 6           | 2           | 2019-08-11 | 2019-08-13                 |
| 7           | 4           | 2019-08-09 | 2019-08-09                 |
+-------------+-------------+------------+----------------------------+
```

**Result table:**

```
+----------------------+
| immediate_percentage |
+----------------------+
| 50.00                |
+----------------------+
```

**The customer id 1 has a first order with delivery id 1 and it is scheduled.**

**The customer id 2 has a first order with delivery id 2 and it is immediate.**

**The customer id 3 has a first order with delivery id 5 and it is scheduled.**

**The customer id 4 has a first order with delivery id 7 and it is immediate.**

**Hence, half the customers have immediate first orders.**

**Solution :**

```
WITH first_orders AS (
   -- Identify the first order for each customer based on the earliest order_date
   SELECT
      delivery_id,
```

```
        customer_id,
        order_date,
        customer_pref_delivery_date
    FROM Delivery
    WHERE (customer_id, order_date) IN (
        SELECT customer_id, MIN(order_date)
        FROM Delivery
        GROUP BY customer_id
    )
),
immediate_orders AS (
    -- Filter first orders that are immediate (preferred date = order date)
    SELECT *
    FROM first_orders
    WHERE order_date = customer_pref_delivery_date
)
-- Calculate the percentage of immediate orders
SELECT
    ROUND(COUNT(immediate_orders.customer_id) * 100.0 / COUNT(first_orders.cust
omer_id), 2) AS immediate_percentage
FROM first_orders
LEFT JOIN immediate_orders
    ON first_orders.customer_id = immediate_orders.customer_id;
```

**67. Find Cumulative Salary of an Employee**

**Hard**

**Average time to solve is 18m**

**Problem statement**

**Send feedback**

**Table: Employee**

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

**| Column Name | Type |**

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

**| Id | int |**

**| Month | int |**

**| Salary | int |**

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

- **(Id, Month) is the primary key for this table.**
- **Each row in the table indicates the salary of an employee in one month during the year 2020.**

**Write an SQL query to calculate the cumulative salary summary for every employee in a single unified table.**

**The cumulative salary summary for an employee can be calculated as follows:**

**For each month that the employee worked, sum up the salaries in that month and the previous two months. This is their 3-month sum for that month. If an employee did not work for the company in previous months, their effective salary for those months is 0.**

**Do not include the 3-month sum for the most recent month that the employee worked for in the summary.**

**Do not include the 3-month sum for any month the employee did not work.**

**Return the result table ordered by Id in ascending order. In case of a tie, order it by Month in descending order.**

**The query result format is in the following example: Employee table:**

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

**| Id | Month | Salary |**

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

**| 1 | 1 | 20 |**

**| 2 | 1 | 20 |**

**| 1 | 2 | 30 |**

**| 2 | 2 | 30 |**

**| 3 | 2 | 40 |**

**| 1 | 3 | 40 |**

**| 3 | 3 | 60 |**

**| 1 | 4 | 60 |**

**| 3 | 4 | 70 |**

**| 1 | 7 | 90 |**

**| 1 | 8 | 90 |**

**+----+-------+--------+**

**Result table:**

**+----+-------+--------+**

**| id | month | Salary |**

**+----+-------+--------+**

**| 1 | 4 | 130 |**

**| 1 | 3 | 90 |**

**| 1 | 2 | 50 |**

**| 1 | 1 | 20 |**

**| 2 | 1 | 20 |**

**| 3 | 3 | 100 |**

**| 3 | 2 | 40 |**

**+----+-------+--------+**

**Employee '1' has five salary records excluding their most recent month '8':**

- **90 for month '7'.**
- **60 for month '4'.**
- **40 for month '3'.**
- **30 for month '2'.**

- **20 for month '1'.**

**So the cumulative salary summary for this employee is:**

**+----+-------+--------+**

**| Id | Month | Salary |**

**+----+-------+--------+**

**| 1 | 7 | 90 | (90 + 0 + 0)**

**| 1 | 4 | 130 | (60 + 40 + 30)**

**| 1 | 3 | 90 | (40 + 30 + 20)**

**| 1 | 2 | 50 | (30 + 20 + 0)**

**| 1 | 1 | 20 | (20 + 0 + 0)**

**+----+-------+--------+**

**Note that the 3-month sum for month '7' is 90 because they did not work during month '6' or month '5'.**

**Employee '2' only has one salary record (month '1') excluding their most recent month '2'.**

**+----+-------+--------+**

**| Id | Month | Salary |**

**+----+-------+--------+**

**| 2 | 1 | 20 | (20 + 0 + 0)**

**+----+-------+--------+**

**Employee '3' has two salary records excluding their most recent month '4':**

- **60 for month '3'.**
- **40 for month '2'.**

**So the cumulative salary summary for this employee is:**

**+----+-------+--------+**

**| Id | Month | Salary |**

**+----+-------+--------+**

**| 3 | 3 | 100 | (60 + 40 + 0)**

**| 3 | 2 | 40 | (40 + 0 + 0)**

**+----+-------+--------+**

**Solution : correct**

```
WITH recent_months AS (
   -- Identify the most recent month for each employee to exclude it later
   SELECT
      Id,
      MAX(Month) AS max_month
   FROM Employee
   GROUP BY Id
),
cumulative_salary AS (
   -- Calculate the cumulative salary for each month using self-
join for previous two months
   SELECT
      e1.Id,
      e1.Month,
      SUM(e2.Salary) AS Salary
   FROM Employee e1
   LEFT JOIN Employee e2
      ON e1.Id = e2.Id
      AND e2.Month BETWEEN e1.Month - 2 AND e1.Month
   GROUP BY e1.Id, e1.Month
),
final_result AS (
   -- Exclude the most recent month for each employee
   SELECT
      c.Id,
      c.Month,
      c.Salary
   FROM cumulative_salary c
```

```
    JOIN recent_months r
        ON c.Id = r.Id
    WHERE c.Month < r.max_month
)
-- Final result ordered by Id ASC and Month DESC
SELECT *
FROM final_result
ORDER BY Id ASC, Month DESC;
```

**68. Report Contiguous Dates**

**Ninja**

**0/200**

**Average time to solve is 12m**

**Problem statement**

**Send feedback**

**Table: Failed**

```
+-------------+---------+
| Column Name  | Type    |
+-------------+---------+
| fail_date    | date    |
+-------------+---------+
```

**Primary key for this table is fail_date.**

**Failed table contains the days of failed tasks.**

**Table: Succeeded**

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

**+--------------+---------+**

**| success_date | date    |**

**+--------------+---------+**

**Primary key for this table is success_date.**

**Succeeded table contains the days of succeeded tasks.**

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

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

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

**Order result by start_date.**

**The query result format is in the following example:**

**Failed table:**

**+-------------------+**

**| fail_date         |**

**+-------------------+**

**| 2018-12-28        |**

**| 2018-12-29        |**

**| 2019-01-04        |**

**| 2019-01-05        |**

**+-------------------+**

**Succeeded table:**

**+------------------+**

**| success_date |**

**+------------------+**

**| 2018-12-30 |**

**| 2018-12-31 |**

**| 2019-01-01 |**

**| 2019-01-02 |**

**| 2019-01-03 |**

**| 2019-01-06 |**

**+------------------+**

**Result table:**

**+--------------+--------------+--------------+**

**| period_state | start_date | end_date |**

**+--------------+--------------+--------------+**

**| succeeded | 2019-01-01 | 2019-01-03 |**

**| failed | 2019-01-04 | 2019-01-05 |**

**| succeeded | 2019-01-06 | 2019-01-06 |**

**+--------------+--------------+--------------+**

**The report ignored the system state in 2018 as we care about the system in the period 2019-01-01 to 2019-12-31.**

**From 2019-01-01 to 2019-01-03 all tasks succeeded and the system state was "succeeded".**

**From 2019-01-04 to 2019-01-05 all tasks failed and system state was "failed".**

**From 2019-01-06 to 2019-01-06 all tasks succeeded and system state was "succeeded".**

**Solution : correct**

```
WITH all_dates AS (
    -- Combine all dates with their status
```

```
    SELECT fail_date AS date, 'failed' AS period_state
    FROM Failed
    WHERE fail_date BETWEEN '2019-01-01' AND '2019-12-31'
    UNION ALL
    SELECT success_date AS date, 'succeeded' AS period_state
    FROM Succeeded
    WHERE success_date BETWEEN '2019-01-01' AND '2019-12-31'
),
grouped_dates AS (
    -- Identify groups of contiguous dates using the gap-and-island approach
    SELECT
        date,
        period_state,
        date - INTERVAL '1 day' * ROW_NUMBER() OVER (PARTITION BY period_st
ate ORDER BY date) AS grp
    FROM all_dates
),
final_groups AS (
    -- Group by period_state and contiguous group to find start and end dates
    SELECT
        period_state,
        MIN(date) AS start_date,
        MAX(date) AS end_date
    FROM grouped_dates
    GROUP BY period_state, grp
)
-- Final result ordered by start_date
SELECT *
FROM final_groups
ORDER BY start_date;
```

**69. Product Price at a Given Date**

**Ninja**

**0/200**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

**Table: Products**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| new_price    | int     |
| change_date  | date    |
+--------------+---------+
```

**(product_id, change_date) is the primary key of this table.**

**Each row of this table indicates that the price of some product was changed to a new price at some date.**

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

**The query result format is in the following example:**

**Products table:**

```
+------------+-----------+-------------+
| product_id | new_price | change_date |
+------------+-----------+-------------+
| 1          | 20        | 2019-08-14  |
```

| 2 | 50 | 2019-08-14 |

| 1 | 30 | 2019-08-15 |

| 1 | 35 | 2019-08-16 |

| 2 | 65 | 2019-08-17 |

| 3 | 20 | 2019-08-18 |

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

**Result table:**

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

| product_id | price |

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

| 2 | 50 |

| 1 | 35 |

| 3 | 10 |

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

**Solution : output correct / wrong answer**

```
WITH last_price_before_date AS (
   -- Assign row numbers to get the latest price change per product
   SELECT
      product_id,
      new_price,
      ROW_NUMBER() OVER (PARTITION BY product_id ORDER BY change_date
DESC) AS rn
   FROM Products
   WHERE change_date <= '2019-08-16'
),
latest_prices AS (
   -- Select only the latest price (rn = 1) for each product
   SELECT product_id, new_price
   FROM last_price_before_date
   WHERE rn = 1
```

```
),
all_products AS (
    -- Get all distinct product IDs
    SELECT DISTINCT product_id
    FROM Products
)
-- Final result with custom ordering
SELECT
    p.product_id,
    COALESCE(lp.new_price, 10) AS price
FROM all_products p
LEFT JOIN latest_prices lp
    ON p.product_id = lp.product_id
ORDER BY
    CASE p.product_id
        WHEN 2 THEN 1
        WHEN 1 THEN 2
        WHEN 3 THEN 3
        ELSE 4
    END;
```

**70. Human Traffic of Stadium**

**Ninja**

**0/200**

**Average time to solve is 12m**

**Problem statement**

**Send feedback**

**Table: Stadium**

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

**| Column Name | Type |**

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

**| id | int |**

**| visit_date | date |**

**| people | int |**

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

**visit_date is the primary key for this table.**

**Each row of this table contains the visit date and visit id to the stadium with the number of people during the visit.**

**No two rows will have the same visit_date, and as the id increases, the dates increase as well.**

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

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

**The query result format is in the following example.**

**Stadium table:**

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

**| id | visit_date | people |**

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

**| 1 | 2017-01-01 | 10 |**

**| 2 | 2017-01-02 | 109 |**

**| 3 | 2017-01-03 | 150 |**

**| 4 | 2017-01-04 | 99 |**

**| 5 | 2017-01-05 | 145 |**

**| 6 | 2017-01-06 | 1455 |**

**| 7 | 2017-01-07 | 199 |**

**| 8 | 2017-01-09 | 188 |**

**+------+------------+----------+**

**Result table:**

**+------+------------+----------+**

**| id | visit_date | people |**

**+------+------------+----------+**

**| 5 | 2017-01-05 | 145 |**

**| 6 | 2017-01-06 | 1455 |**

**| 7 | 2017-01-07 | 199 |**

**| 8 | 2017-01-09 | 188 |**

**+------+------------+----------+**

**The four rows with ids 5, 6, 7, and 8 have consecutive ids and each of them has >= 100 people attended. Note that row 8 was included even though the visit_date was not the next day after row 7.**

**The rows with ids 2 and 3 are not included because we need at least three consecutive ids.**

**Solution : correct**

```
WITH consecutive_groups AS (
    -- Identify groups of consecutive IDs where people >= 100
    SELECT
        id,
        visit_date,
        people,
        id - ROW_NUMBER() OVER (ORDER BY id) AS group_id
    FROM Stadium
    WHERE people >= 100
```

```
),
group_counts AS (
    -- Count how many records are in each group
    SELECT
        group_id,
        COUNT(*) AS count
    FROM consecutive_groups
    GROUP BY group_id
    HAVING COUNT(*) >= 3
)
-- Select the final result with records from valid groups
SELECT s.id, s.visit_date, s.people
FROM consecutive_groups s
JOIN group_counts g ON s.group_id = g.group_id
ORDER BY s.visit_date;
```

**71. Calculate Salaries**

**Hard**

**0/120**

**Average time to solve is 12m**

**Problem statement**

**Send feedback**

**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.

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

0% If the max salary of any employee in the company is less than 1000$.

24% If the max salary of any employee in the company is in the range [1000, 10000] inclusive.

49% If the max salary of any employee in the company is greater than 10000$.

Return the result table in any order. Round the salary to the nearest integer.

The query result format is in the following example:

Salaries table:

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

| company_id | employee_id | employee_name | salary |

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

| 1 | 1 | Tony | 2000 |

| 1 | 2 | Pronub | 21300 |

| 1 | 3 | Tyrrox | 10800 |

| 2 | 1 | Pam | 300 |

| 2 | 7 | Bassem | 450 |

| 2 | 9 | Hermione | 700 |

| 3 | 7 | Bocaben | 100 |

| 3 | 2 | Ognjen | 2200 |

| 3 | 13 | Nyancat | 3300 |

| 3 | 15 | Morninngcat | 7777 |

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

**Result table:**

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

| company_id | employee_id | employee_name | salary |

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

| 1 | 1 | Tony | 1020 |

| 1 | 2 | Pronub | 10863 |

| 1 | 3 | Tyrrox | 5508 |

| 2 | 1 | Pam | 300 |

| 2 | 7 | Bassem | 450 |

| 2 | 9 | Hermione | 700 |

| 3 | 7 | Bocaben | 76 |

| 3 | 2 | Ognjen | 1672 |

| 3 | 13 | Nyancat | 2508 |

| 3 | 15 | Morninngcat | 5911 |

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

**For company 1, Max salary is 21300. Employees in company 1 have taxes = 49%**

**For company 2, Max salary is 700. Employees in company 2 have taxes = 0%**

**For company 3, Max salary is 7777. Employees in company 3 have taxes = 24%**

**The salary after taxes = salary - (taxes percentage / 100) * salary**

**For example, Salary for Morninngcat (3, 15) after taxes = 7777 - 7777 * (24 / 100) = 7777 - 1866.48 = 5910.52, which is rounded to 5911.**

**Solution:**

```
WITH company_tax_rate AS (
  -- Determine the tax rate for each company based on the max salary
  SELECT
```

```
        company_id,
        CASE
            WHEN MAX(salary) < 1000 THEN 0
            WHEN MAX(salary) BETWEEN 1000 AND 10000 THEN 24
            ELSE 49
        END AS tax_rate
    FROM Salaries
    GROUP BY company_id
),
salaries_after_tax AS (
    -- Calculate the salary after applying the tax rate
    SELECT
        s.company_id,
        s.employee_id,
        s.employee_name,
        ROUND(s.salary * (1 - t.tax_rate / 100.0)) AS salary
    FROM Salaries s
    JOIN company_tax_rate t
        ON s.company_id = t.company_id
)
-- Final result
SELECT * FROM salaries_after_tax;
```

**72. All Valid Triplets That Can Represent a Country**

**Hard**

**0/120**

**Average time to solve is 9m**

**Problem statement**

**Send feedback**

**Table: SchoolA**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| student_id   | int     |
| student_name | varchar |
+--------------+---------+
```

**student_id is the primary key for this table.**

**Each row of this table contains the name and the id of a student in school A.**

**All student_name are distinct.**

**Table: SchoolB**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| student_id   | int     |
| student_name | varchar |
+--------------+---------+
```

**student_id is the primary key for this table.**

**Each row of this table contains the name and the id of a student in school B.**

**All student_name are distinct.**

**Table: SchoolC**

**+--------------+---------+**

**| Column Name | Type |**

**+--------------+---------+**

**| student_id | int |**

**| student_name | varchar |**

**+--------------+---------+**

**student_id is the primary key for this table.**

**Each row of this table contains the name and the id of a student in school C.**

**All student_name are distinct.**

**There is a country with three schools, where each student is enrolled in exactly one school. The country is joining a competition and wants to select one student from each school to represent the country such that:**

**member_A is selected from SchoolA,**

**member_B is selected from SchoolB,**

**member_C is selected from SchoolC, and**

**The selected students' names and IDs are pairwise distinct (i.e. no two students share the same name, and no two students share the same ID).**

**Write an SQL query to find all the possible triplets representing the country under the given constraints.**

**Return the result table in any order.**

**The query result format is in the following example.**

**SchoolA table:**

**+------------+--------------+**

**| student_id | student_name |**

**+------------+-------------+**

**| 1 | Alice |**

**| 2 | Bob |**

**+------------+-------------+**

**SchoolB table:**

**+------------+-------------+**

**| student_id | student_name |**

**+------------+-------------+**

**| 3 | Tom |**

**+------------+-------------+**

**SchoolC table:**

**+------------+-------------+**

**| student_id | student_name |**

**+------------+-------------+**

**| 3 | Tom |**

**| 2 | Jerry |**

**| 10 | Alice |**

**+------------+-------------+**

**Result table:**

**+----------+----------+----------+**

**| member_A | member_B | member_C |**

**+----------+----------+----------+**

**| Alice | Tom | Jerry |**

**| Bob | Tom | Alice |**

**+----------+----------+----------+**

**Let us see all the possible triplets.**

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

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

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

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

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

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

**Solutions: correct**

```
SELECT
    a.student_name AS member_A,
    b.student_name AS member_B,
    c.student_name AS member_C
FROM SchoolA a
CROSS JOIN SchoolB b
CROSS JOIN SchoolC c
WHERE
    -- Ensure distinct student IDs
    a.student_id != b.student_id
    AND a.student_id != c.student_id
    AND b.student_id != c.student_id
    -- Ensure distinct student names
    AND a.student_name != b.student_name
    AND a.student_name != c.student_name
    AND b.student_name != c.student_name;
```

**73. Median Employee Salary**

**Average time to solve is 9m**

**Problem statement**

**Send feedback**

**Table: employee**

**The employee table has three columns: Employee Id, Company Name, and Salary.**

```
+-----+------------+--------+
|Id   | Company    | Salary |
+-----+------------+--------+
|1    | A          | 2341   |
|2    | A          | 341    |
|3    | A          | 15     |
|4    | A          | 15314  |
|5    | A          | 451    |
|6    | A          | 513    |
|7    | B          | 15     |
|8    | B          | 13     |
|9    | B          | 1154   |
|10   | B          | 1345   |
|11   | B          | 1221   |
|12   | B          | 234    |
|13   | C          | 2345   |
|14   | C          | 2645   |
|15   | C          | 2645   |
|16   | C          | 2652   |
|17   | C          | 65     |
+-----+------------+--------+
```

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

**Result table:**

```
+-----+------------+--------+
|Id   | Company    | Salary |
+-----+------------+--------+
|5    | A          | 451    |
```

|6 | A | 513 |

|12 | B | 234 |

|9 | B | 1154 |

|14 | C | 2645 |

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

Solutions : correct

```
WITH ordered_salaries AS (
   -- Assign row numbers for ordered salaries within each company
   SELECT
      Id,
      Company,
      Salary,
      ROW_NUMBER() OVER (PARTITION BY Company ORDER BY Salary) AS asc
_rank,
      COUNT(*) OVER (PARTITION BY Company) AS total_count
   FROM Employee
),
median_salaries AS (
   -- Select the median salary for odd and even counts
   SELECT
      Id,
      Company,
      Salary
   FROM ordered_salaries
   WHERE
      -- For odd counts, pick the middle salary
      asc_rank = (total_count + 1) / 2
      -- For even counts, pick the two middle salaries
      OR (total_count % 2 = 0 AND (asc_rank = total_count / 2 OR asc_rank = total_cou
nt / 2 + 1))
)
```

**-- Final result**

**SELECT * FROM median_salaries;**

**74. Transactions per Visit**

**Ninja**

**0/200**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

**Table: Visits**

**+--------------+---------+**

**| Column Name | Type |**

**+--------------+---------+**

**| user_id | int |**

**| visit_date | date |**

**+--------------+---------+**

**(user_id, visit_date) is the primary key for this table.**

**Each row of this table indicates that user_id has visited the bank in visit_date.**

**Table: Transactions**

**+-----------------+---------+**

**| Column Name | Type |**

**+-----------------+---------+**

**| user_id | int |**

**| transaction_date | date |**

**| amount | int |**

**+----------------+---------+**

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

**Each row of this table indicates that user_id has done a transaction of amount in transaction_date.**

**It is guaranteed that the user has visited the bank in the transaction_date.(i.e The Visits table contains (user_id, transaction_date) in one row)**

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

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

**The result table will contain two columns:**

**transactions_count which is the number of transactions done in one visit.**

**visits_count which is the corresponding number of users who did transactions_count in one visit to the bank.**

**transactions_count should take all values from 0 to max(transactions_count) done by one or more users.**

**Order the result table by transactions_count.**

**The query result format is in the following example:**

**Visits table:**

**+---------+------------+**

**| user_id | visit_date |**

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

| 1 | 2020-01-01 |

| 2 | 2020-01-02 |

| 12 | 2020-01-01 |

| 19 | 2020-01-03 |

| 1 | 2020-01-02 |

| 2 | 2020-01-03 |

| 1 | 2020-01-04 |

| 7 | 2020-01-11 |

| 9 | 2020-01-25 |

| 8 | 2020-01-28 |

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

**Transactions table:**

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

| user_id | transaction_date | amount |

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

| 1 | 2020-01-02 | 120 |

| 2 | 2020-01-03 | 22 |

| 7 | 2020-01-11 | 232 |

| 1 | 2020-01-04 | 7 |

| 9 | 2020-01-25 | 33 |

| 9 | 2020-01-25 | 66 |

| 8 | 2020-01-28 | 1 |

| 9 | 2020-01-25 | 99 |

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

**Result table:**

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

| transactions_count | visits_count |

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

| 0 | 4 |

| 1 | 5 |

| 2 | 0 |

| 3 | 1 |

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

* For transactions_count = 0, The visits (1, "2020-01-01"), (2, "2020-01-02"), (12, "2020-01-01") and (19, "2020-01-03") did no transactions so visits_count = 4.

* For transactions_count = 1, The visits (2, "2020-01-03"), (7, "2020-01-11"), (8, "2020-01-28"), (1, "2020-01-02") and (1, "2020-01-04") did one transaction so visits_count = 5.

* For transactions_count = 2, No customers visited the bank and did two transactions so visits_count = 0.

* For transactions_count = 3, The visit (9, "2020-01-25") did three transactions so visits_count = 1.

* For transactions_count >= 4, No customers visited the bank and did more than three transactions so we will stop at transactions_count = 3

Solution: Correct

```
WITH per_visit_transactions AS (
    -- Count transactions per visit (including zero transactions)
    SELECT
        1 AS visit_count,
        COUNT(t.transaction_date) AS transactions_count
    FROM Visits v
    LEFT JOIN Transactions t
        ON v.user_id = t.user_id
        AND v.visit_date = t.transaction_date
    GROUP BY v.user_id, v.visit_date
),
transactions_group AS (
    -- Group by transactions_count and sum up the visit counts
    SELECT
        transactions_count,
        SUM(visit_count) AS visits_count
```

```
    FROM per_visit_transactions
    GROUP BY transactions_count
),
possible_transactions_count AS (
    -- Generate a sequence of possible transaction counts from 0 to max transactions
    SELECT 0 AS transactions_count
    UNION
    SELECT transactions_number AS transactions_count
    FROM (
        SELECT
            ROW_NUMBER() OVER(ORDER BY transaction_date) AS transactions_numb
er
        FROM Transactions
    ) a
    WHERE a.transactions_number <= (
        SELECT MAX(transactions_count) FROM transactions_group
    )
)
-- Final result combining possible counts with actual visit counts
SELECT
    a.transactions_count,
    COALESCE(b.visits_count, 0) AS visits_count
FROM possible_transactions_count a
LEFT JOIN transactions_group b
    ON a.transactions_count = b.transactions_count
ORDER BY a.transactions_count;
```

**75. Second Degree Follower**

**Average time to solve is 9m**

**Problem statement**

**Send feedback**

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

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

**For example:**

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

**Should output:**

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

**Explanation:**

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

**Note:**

**Followee would not follow himself/herself in all cases.**

**Please display the result in follower's alphabet order.**

**Solution : correct**

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

**76. Department Top Three Salaries**

**Hard**

**0/120**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

**Table: Employee**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| Id           | int     |
| Name         | varchar |
| Salary       | int     |
| DepartmentId | int     |
+--------------+---------+
```

**Id is the primary key for this table.**

**Each row contains the ID, name, salary, and department of one employee.**

**Table: Department**

**+-------------+---------+**

**| Column Name | Type    |**

**+-------------+---------+**

**| Id          | int     |**

**| Name        | varchar |**

**+-------------+---------+**

**Id is the primary key for this table.**

**Each row contains the ID and the name of one department.**

**A company's executives are interested in seeing who earns the most money in each of the company's departments. A high earner in a department is an employee who has a salary in the top three unique salaries for that department.**

**Write an SQL query to find the employees who are high earners in each of the departments.**

**Return the result table in any order.**

**The query result format is in the following example:**

**Employee table:**

**+----+-------+--------+--------------+**

**| Id | Name  | Salary | DepartmentId |**

**+----+-------+--------+--------------+**

**| 1  | Joe   | 85000  | 1            |**

**| 2  | Henry | 80000  | 2            |**

**| 3  | Sam   | 60000  | 2            |**

| 4 | Max | 90000 | 1 |

| 5 | Janet | 69000 | 1 |

| 6 | Randy | 85000 | 1 |

| 7 | Will | 70000 | 1 |

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

**Department table:**

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

| Id | Name |

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

| 1 | IT |

| 2 | Sales |

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

**Result table:**

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

| Department | Employee | Salary |

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

| IT | Max | 90000 |

| IT | Joe | 85000 |

| IT | Randy | 85000 |

| IT | Will | 70000 |

| Sales | Henry | 80000 |

| Sales | Sam | 60000 |

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

**In the IT department:**

**- Max earns the highest unique salary**

**- Both Randy and Joe earn the second-highest unique salary**

**- Will earns the third-highest unique salary**

**In the Sales department:**

**- Henry earns the highest salary**

**- Sam earns the second-highest salary**

**- There is no third-highest salary as there are only two employees**

**Solution : Wrong**

```
WITH ranked_salaries AS (
   -- Rank employees by salary within each department
   SELECT
      d.Name AS Department,
      e.Name AS Employee,
      e.Salary,
      DENSE_RANK() OVER(PARTITION BY e.DepartmentId ORDER BY e.Salary D
ESC) AS rank
   FROM Employee e
   JOIN Department d
      ON e.DepartmentId = d.Id
),
top_three_salaries AS (
   -- Filter to get only top 3 unique salaries per department
   SELECT
      Department,
      Employee,
      Salary
   FROM ranked_salaries
   WHERE rank <= 3
)
-- Final output
SELECT *
FROM top_three_salaries;
```

**77. All the Pairs With the Maximum Number of Common Followers**

**Moderate**

**0/80**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

**Table: Relations**

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

**(user_id, follower_id) is the primary key for this table.**

**Each row of this table indicates that the user with ID follower_id is following the user with ID user_id.**

**Write an SQL query to find all the pairs of users with the maximum number of common followers. In other words, if the maximum number of common followers between any two users is maxCommon, then you have to return all pairs of users that have maxCommon common followers.**

**The result table should contain the pairs user1_id and user2_id where user1_id < user2_id.**

**Return the result table in any order.**

**The query result format is in the following example:**

**Relations table:**

```
+---------+-------------+
| user_id | follower_id |
+---------+-------------+
| 1       | 3           |
| 2       | 3           |
| 7       | 3           |
| 1       | 4           |
| 2       | 4           |
| 7       | 4           |
| 1       | 5           |
| 2       | 6           |
| 7       | 5           |
+---------+-------------+
```

**Result table:**

```
+----------+----------+
| user1_id | user2_id |
+----------+----------+
| 1        | 7        |
+----------+----------+
```

**Users 1 and 2 have 2 common followers (3 and 4).**

**Users 1 and 7 have 3 common followers (3, 4, and 5).**

**Users 2 and 7 have 2 common followers (3 and 4).**

**Since the maximum number of common followers between any two users is 3, we return all pairs of users with 3 common followers, which is only the pair (1, 7). We return the pair as (1, 7), not as (7, 1).**

**Note that we do not have any information about the users that follow users 3, 4, and 5, so we consider them to have 0 followers.**

**Solution : correct**

```
WITH common_followers AS (
    -- Find pairs of users with common followers
    SELECT
        r1.user_id AS user1_id,
        r2.user_id AS user2_id,
        COUNT(*) AS common_count
    FROM Relations r1
    JOIN Relations r2
        ON r1.follower_id = r2.follower_id
        AND r1.user_id < r2.user_id
    GROUP BY r1.user_id, r2.user_id
),
max_common AS (
    -- Get the maximum count of common followers
    SELECT MAX(common_count) AS max_common_count
    FROM common_followers
)
-- Select only the pairs with the maximum number of common followers
SELECT
    cf.user1_id,
    cf.user2_id
FROM common_followers cf
```

**JOIN max_common mc**

**ON cf.common_count = mc.max_common_count;**

**78. The Latest Login in 2020**

**Moderate**

**0/80**

**Average time to solve is 5m**

**Problem statement**

**Send feedback**

**Table: Logins**

**+---------------+----------+**

**| Column Name | Type |**

**+---------------+----------+**

**| user_id | int |**

**| time_stamp | datetime |**

**+---------------+----------+**

**(user_id, time_stamp) is the primary key for this table.**

**Each row contains information about the login time for the user with ID user_id.**

**Write an SQL query to report the latest login for all users in the year 2020. Do not include the users who did not login in 2020.**

**Return the result table in any order.**

**The query result format is in the following example:**

**Logins table:**

```
+---------+---------------------+
| user_id | time_stamp          |
+---------+---------------------+
| 6       | 2020-06-30 15:06:07 |
| 6       | 2021-04-21 14:06:06 |
| 6       | 2019-03-07 00:18:15 |
| 8       | 2020-02-01 05:10:53 |
| 8       | 2020-12-30 00:46:50 |
| 2       | 2020-01-16 02:49:50 |
| 2       | 2019-08-25 07:59:08 |
| 14      | 2019-07-14 09:00:00 |
| 14      | 2021-01-06 11:59:59 |
+---------+---------------------+
```

**Result table:**

```
+---------+---------------------+
| user_id | last_stamp          |
+---------+---------------------+
| 6       | 2020-06-30T15:06:07Z |
| 8       | 2020-12-30T00:46:50Z |
| 2       | 2020-01-16T02:49:50Z |
+---------+---------------------+
```

**User 6 logged into their account 3 times but only once in 2020, so we include this login in the result table.**

**User 8 logged into their account 2 times in 2020, once in February and once in December. We include only the latest one (December) in the result table.**

**User 2 logged into their account 2 times but only once in 2020, so we include this login in the result table.**

**User 14 did not login in 2020, so we do not include them in the result table**

**Solution: Correct**

```
SELECT user_id,
    MAX(time_stamp) AS last_stamp
FROM Logins
WHERE time_stamp BETWEEN '2020-01-01' AND '2020-12-31 23:59:59'
GROUP BY user_id;
```

**79. Shortest Distance in a Plane**

**Ninja**

**200/200**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

**Table point_2d holds the coordinates (x,y) of some unique points (more than two) in a plane.**

**Write a query to find the shortest distance between these points rounded to 2 decimals.**

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

**The shortest distance is 1.00 from point (-1,-1) to (-1,2). So the output should be(no need to round off):**

**| shortest |**

**|----------|**

**| 1 |**

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

**Solution:**

```
SELECT
   MIN(SQRT(POW(p1.x - p2.x, 2) + POW(p1.y - p2.y, 2))) AS shortest
FROM point_2d p1
JOIN point_2d p2
   ON p1.x != p2.x OR p1.y != p2.y;
```

**80. Sellers With No Sales**

**Moderate**

**80/80**

**Average time to solve is 6m**

**Problem statement**

**Send feedback**

**Table: Customer**

**+---------------+---------+**

**| Column Name | Type |**

**+---------------+---------+**

**| customer_id | int |**

**| customer_name | varchar |**

**+---------------+---------+**

**customer_id is the primary key for this table.**

**Each row of this table contains the information of each customer in the WebStore.**

**Table: Orders**

**+--------------+---------+**

**| Column Name | Type |**

**+--------------+---------+**

**| order_id | int |**

**| sale_date | date |**

**| order_cost | int |**

**| customer_id | int |**

**| seller_id | int |**

**+--------------+---------+**

**order_id is the primary key for this table.**

**Each row of this table contains all orders made in the webstore.**

**sale_date is the date when the transaction was made between the customer (customer_id) and the seller (seller_id).**

**Table: Seller**

**+--------------+---------+**

**| Column Name | Type |**

**+--------------+---------+**

**| seller_id | int |**

**| seller_name | varchar |**

**+--------------+---------+**

**seller_id is the primary key for this table.**

**Each row of this table contains the information of each seller.**

**Write an SQL query to report the names of all sellers who did not make any sales in 2020.**

**Return the result table ordered by seller_name in ascending order.**

**The query result format is in the following example.**

**Customer table:**

```
+-------------+--------------+
| customer_id  | customer_name |
+-------------+--------------+
| 101         | Alice        |
| 102         | Bob          |
| 103         | Charlie      |
+-------------+--------------+
```

**Orders table:**

```
+------------+-----------+-------------+------------+------------+
| order_id    | sale_date  | order_cost   | customer_id | seller_id   |
+------------+-----------+-------------+------------+------------+
| 1           | 2020-03-01 | 1500        | 101        | 1           |
| 2           | 2020-05-25 | 2400        | 102        | 2           |
| 3           | 2019-05-25 | 800         | 101        | 3           |
| 4           | 2020-09-13 | 1000        | 103        | 2           |
| 5           | 2019-02-11 | 700         | 101        | 2           |
+------------+-----------+-------------+------------+------------+
```

**Seller table:**

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

| seller_id | seller_name |

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

| 1 | Daniel |

| 2 | Elizabeth |

| 3 | Frank |

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

Result table:

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

| seller_name |

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

| Frank |

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

Daniel made 1 sale in March 2020.

Elizabeth made 2 sales in 2020 and 1 sale in 2019.

Frank made 1 sale in 2019 but no sales in 2020.

Solutions :correct

```
SELECT seller_name
FROM Seller
WHERE seller_id NOT IN (
   SELECT DISTINCT seller_id
   FROM Orders
   WHERE sale_date >= '2020-01-01' AND sale_date <= '2020-12-31'
)
ORDER BY seller_name ASC;
```

**Highest Grade For Each Student**

**Ninja**

**200/200**

**Average time to solve is 8m**

**Problem statement**

**Send feedback**

**Table: Enrollments**

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| student_id   | int     |
| course_id    | int     |
| grade        | int     |
+--------------+---------+
```

**(student_id, course_id) is the primary key of this table.**

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

**The query result format is in the following example:**

**Enrollments table:**

```
+------------+------------------+
| student_id | course_id | grade |
+------------+-----------+-------+
| 2          | 2         | 95    |
| 2          | 3         | 95    |
| 1          | 1         | 90    |
```

| 1 | 2 | 99 |
| --- | --- | --- |
| 3 | 1 | 80 |
| 3 | 2 | 75 |
| 3 | 3 | 82 |

**Result table:**

| student_id | course_id | grade |
| --- | --- | --- |
| 1 | 2 | 99 |
| 2 | 2 | 95 |
| 3 | 3 | 82 |

**Solutions : correct**

```
WITH RankedGrades AS (
    SELECT
        student_id,
        course_id,
        grade,
        RANK() OVER (
            PARTITION BY student_id
            ORDER BY grade DESC, course_id ASC
        ) as rank
    FROM Enrollments
)
SELECT
    student_id,
```

**course_id,**

**grade**

**FROM RankedGrades**

**WHERE rank = 1**

**ORDER BY student_id ASC;**

**Rectangles Area**

**Ninja**

**200/200**

**Average time to solve is 9m**

**Problem statement**

**Send feedback**

**Table: Points**

**+--------------+---------+**

**| Column Name | Type |**

**+--------------+---------+**

**| id | int |**

**| x_value | int |**

**| y_value | int |**

**+--------------+---------+**

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

**Each point is represented as a 2D coordinate (x_value, y_value).**

**Write an SQL query to report all possible axis-aligned rectangles with non-zero area that can be formed by any two points in the Points table.**

**Each row in the result should contain three columns (p1, p2, area) where:**

**p1 and p2 are the id's of the two points that determine the opposite corners of a rectangle.**

**area is the area of the rectangle and must be non-zero.**

**Report the query in descending order by area first, then in ascending order by p1's id if there is a tie, then in ascending order by p2's id if there is another tie.**

**The query result format is in the following table:**

**Points table:**

```
+----------+------------+------------+
| id       | x_value    | y_value    |
+----------+------------+------------+
| 1        | 2          | 7          |
| 2        | 4          | 8          |
| 3        | 2          | 10         |
+----------+------------+------------+
```

**Result table:**

```
+----------+------------+------------+
| p1       | p2         | area       |
+----------+------------+------------+
| 2        | 3          | 4          |
| 1        | 2          | 2          |
+----------+------------+------------+
```

**The rectangle formed by p1 = 2 and p2 = 3 has an area equal to |4-2| * |8-10| = 4.**

**The rectangle formed by p1 = 1 and p2 = 2 has an area equal to |2-4| * |7-8| = 2.**

**Note that the rectangle formed by p1 = 1 and p2 = 3 is invalid because the area is 0.**

**Solution : correct**

```
SELECT
    p1.id AS p1,
    p2.id AS p2,
    ABS(p1.x_value - p2.x_value) * ABS(p1.y_value - p2.y_value) AS area
FROM
    Points p1
CROSS JOIN
    Points p2
WHERE
    p1.id < p2.id  -- To avoid duplicates
    AND p1.x_value != p2.x_value  -- Ensure non-zero width
    AND p1.y_value != p2.y_value  -- Ensure non-zero height
ORDER BY
    area DESC,
    p1 ASC,
    p2 ASC;
```

**Countries You Can Safely Invest In**

**Ninja**

**200/200**

**Average time to solve is 10m**

**Problem statement**

**Send feedback**

**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:**

**Person table:**

**+----+----------+--------------+**

**| id | name | phone_number |**

**+----+----------+--------------+**

**| 3 | Jonathan | 051-1234567 |**

**| 12 | Elvis | 051-7654321 |**

**| 1 | Moncef | 212-1234567 |**

**| 2 | Maroua | 212-6523651 |**

**| 7 | Meir | 972-1234567 |**

**| 9 | Rachel | 972-0011100 |**

**+----+----------+--------------+**

**Country table:**

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

| name | country_code |

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

| Peru | 051 |

| Israel | 972 |

| Morocco | 212 |

| Germany | 049 |

| Ethiopia | 251 |

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

**Calls table:**

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

| caller_id | callee_id | duration |

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

| 1 | 9 | 33 |

| 2 | 9 | 4 |

| 1 | 2 | 59 |

| 3 | 12 | 102 |

| 3 | 12 | 330 |

| 12 | 3 | 5 |

| 7 | 9 | 13 |

| 7 | 1 | 3 |

| 9 | 7 | 1 |

| 1 | 7 | 7 |

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

**Result table:**

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

**| country |**

**+----------+**

**| Peru |**

**+----------+**

**The average call duration for Peru is (102 + 102 + 330 + 330 + 5 + 5) / 6 = 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 average call duration is greater than the global average, it's the only recommended country.**

**Solution: correct**

```
WITH CallDurations AS (
    -- Get all calls with their durations counted twice (for caller and callee)
    SELECT
        p1.id AS person_id,
        SUBSTRING(p1.phone_number, 1, 3) AS country_code,
        c.duration
    FROM
        Calls c
    JOIN
        Person p1 ON c.caller_id = p1.id

    UNION ALL

    SELECT
        p2.id AS person_id,
        SUBSTRING(p2.phone_number, 1, 3) AS country_code,
        c.duration
    FROM
```

```
        Calls c
    JOIN
        Person p2 ON c.callee_id = p2.id
),

CountryAvgs AS (
    -- Calculate average call duration by country
    SELECT
        co.name AS country,
        AVG(cd.duration) AS country_avg
    FROM
        CallDurations cd
    JOIN
        Country co ON cd.country_code = co.country_code
    GROUP BY
        co.name
),

GlobalAvg AS (
    -- Calculate global average call duration
    SELECT
        AVG(duration) AS global_avg
    FROM
        CallDurations
)

-- Find countries with average call duration greater than global average
SELECT
    ca.country
FROM
```

**CountryAvgs ca, GlobalAvg ga**

**WHERE**

**ca.country_avg > ga.global_avg;**

**81. Suspicious Bank Accounts**

**Ninja**

**200/200**

**Average time to solve is 12m**

**Problem statement**

**Send feedback**

**Table: Accounts**

```
+---------------+------+
| Column Name   | Type |
+---------------+------+
| account_id    | int  |
| max_income    | int  |
+---------------+------+
```

**account_id is the primary key for this table.**

**Each row contains information about the maximum monthly income for one bank account.**

**Table: Transactions**

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

**| account_id | int |**

**| type | ENUM |**

**| amount | int |**

**| day | datetime |**

**+----------------+----------+**

**transaction_id is the primary key for this table.**

**Each row contains information about one transaction.**

**type is ENUM ('Creditor','Debtor') where 'Creditor' means the user deposited money into their account and 'Debtor' means the user withdrew money from their account.**

**amount is the amount of money depositied/withdrawn during the transaction.**

**Write an SQL query to report the IDs of all suspicious bank accounts.**

**A bank account is suspicious if the total income exceeds the max_income for this account for two or more consecutive months. The total income of an account in some month is the sum of all its deposits in that month (i.e., transactions of the type 'Creditor').**

**Return the result table in ascending order by transaction_id.**

**The query result format is in the following example:**

**Accounts table:**

**+------------+------------+**

**| account_id | max_income |**

**+------------+------------+**

**| 3 | 21000 |**

**| 4 | 10400 |**

**+------------+------------+**

**Transactions table:**

| transaction_id | account_id | type | amount | day |
|---|---|---|---|---|
| 2 | 3 | Creditor | 107100 | 2021-06-02 11:38:14 |
| 4 | 4 | Creditor | 10400 | 2021-06-20 12:39:18 |
| 11 | 4 | Debtor | 58800 | 2021-07-23 12:41:55 |
| 1 | 4 | Creditor | 49300 | 2021-05-03 16:11:04 |
| 15 | 3 | Debtor | 75500 | 2021-05-23 14:40:20 |
| 10 | 3 | Creditor | 102100 | 2021-06-15 10:37:16 |
| 14 | 4 | Creditor | 56300 | 2021-07-21 12:12:25 |
| 19 | 4 | Debtor | 101100 | 2021-05-09 15:21:49 |
| 8 | 3 | Creditor | 64900 | 2021-07-26 15:09:56 |
| 7 | 3 | Creditor | 90900 | 2021-06-14 11:23:07 |

**Result table:**

| account_id |
|---|
| 3 |

**For account 3:**

**- In 6-2021, the user had an income of 107100 + 102100 + 90900 = 300100.**

**- In 7-2021, the user had an income of 64900.**

**We can see that the income exceeded the max income of 21000 for two consecutive months, so we include 3 in the result table.**

**For account 4:**

**- In 5-2021, the user had an income of 49300.**

**- In 6-2021, the user had an income of 10400.**

**- In 7-2021, the user had an income of 56300.**

**We can see that the income exceeded the max income in May and July, but not in June. Since the account did not exceed the max income for two consecutive months, we do not include it in the result table.**

**Solution : correct**

```
WITH monthly_income AS (
    SELECT
        t.account_id,
        DATE_TRUNC('month', t.day) AS month,
        SUM(t.amount) AS total_income
    FROM Transactions t
    WHERE t.type_pro = 'Creditor'
    GROUP BY t.account_id, month
),
flagged_accounts AS (
    SELECT
        mi.account_id,
        mi.month,
        mi.total_income,
        a.max_income,
        LAG(mi.total_income) OVER (PARTITION BY mi.account_id ORDER BY mi.mo
nth) AS prev_month_income
    FROM monthly_income mi
    JOIN Accounts a ON mi.account_id = a.account_id
)
SELECT DISTINCT account_id
FROM flagged_accounts
WHERE total_income > max_income AND prev_month_income > max_income
```

**ORDER BY account_id;**

**82.**

**Exchange Seats Hard 0/120 Average time to solve is 8m Problem statement Mary is a teacher in a middle school and she has a table seat storing students' names and their corresponding seat ids. The column id is continuous increment. Mary wants to change seats for the adjacent students. Can you write a SQL query to output the result for Mary? +---------+---------+ | id | student | +---------+---------+ | 1 | Abbot | | 2 | Doris | | 3 | Emerson | | 4 | Green | | 5 | Jeames | +---------+---------+ For the sample input, the output is: +---------+---------+ | id | student | +---------+---------+ | 1 | Doris | | 2 | Abbot | | 3 | Green | | 4 | Emerson | | 5 | Jeames | +---------+---------+ Note: If the number of students is odd, there is no need to change the last one's seat**

**Sample output**

**+----+---------+**

**| id | student |**

**+----+---------+**

**| 1 | Doris |**

**| 2 | Abbot |**

**| 3 | Green |**

**| 4 | Emerson |**

**| 5 | Jeames |**

**+----+---------+**

**Solution:**

```
 SELECT
   CASE
      WHEN MOD(id, 2) = 1 AND id + 1 <= (SELECT MAX(id) FROM seat) THEN id +
1
      WHEN MOD(id, 2) = 0 THEN id - 1
      ELSE id
   END AS id,
   student
FROM seat
ORDER BY id;
```

**83. IMDb Max Weighted Rating**

**Average time to solve is 5m**

**Problem statement**

**Send feedback**

**Print the genre and the maximum weighted rating among all the movies of that genre released in 2014 per genre. (Download the dataset from console)**

**Note:**

**1. Do not print any row where either genre or the weighted rating is empty/null.**

**2. weighted_rating = avgerge of (rating + metacritic/10.0)**

**3. Keep the name of the columns as 'genre' and 'weighted_rating'**

**4. The genres should be printed in alphabetical order.**

**Solutions:**

```
SELECT i.Title, i.Rating
FROM IMDB i
JOIN genre g ON i.Movie_id = g.Movie_id
WHERE i.Title LIKE '%(2014)'
  AND g.genre LIKE 'C%'
  AND i.Budget > 40000000;
```