**1.Find products with valid serial numbers**

|  |  |
| --- | --- |
| Column Name | Type |
| Product\_id | int |
| Product\_name | varchar |
| description | varchar |

(product\_id) is the unique key for this table.Each row in the table represents a product with its unique id , name , and descriptions

Write a solution to find all products whose description contains a valid serial number pattern . A valid serial number follows these rules:

* It starts with the letters SN(case -sensitive).
* Followed by exactly 4 digits
* It must have a hypen(-) followed by exactly 4 digits
* The serial number must be within the description (it may not necessarily start at the beginning)
* Return the result table ordered by product\_id in ascending order.

**Input:**

**Products Table**

|  |  |  |
| --- | --- | --- |
| **product\_id** | **product\_name** | **description** |
| 1 | Widget A | This is a sample product with SN1234-5678 |
| 2 | Widget B | A product with serial SN9876-1234 in the description |
| 3 | Widget C | Product SN1234-56789 is available now |
| 4 | Widget D | No serial number here |
| 5 | Widget E | Check out SN4321-8765 in this description |

**Output**

|  |  |  |
| --- | --- | --- |
| **product\_id** | **product\_name** | **description** |
| 1 | Widget A | This is a sample product with SN1234-5678 |
| 2 | Widget B | A product with serial SN9876-1234 in the description |
| 5 | Widget E | Check out SN4321-8765 in this description |

**Solution:**

**SELECT \***

**FROM PRODUCTS**

**WHERE DESCRIPTIONS REGREP ‘SN[O-9]{4}\-[0-9]{4}([^0-9]+|$)’**

**ORDER BY PRODUCT\_ID;**

**2**.DNA pattern recognition

|  |  |
| --- | --- |
| Column Name | Type |
| sample\_id | int |
| dna\_sequence | varchar |
| species | varchar |

sample\_id is the unique key for this table.

Each row contains a DNA sequence represented as a string of characters (A,T, G,C) and the species it was collected from.

Biologists are studying basic patterns in DNA sequences. Write a solution to identify sample\_id with the following patterns :

* Sequence that start with ATG(a common start codon)
* Sequence that end with either TAA, TAG, or TGA(stop codons)
* Sequences containing the motif ATAT(a simple repeated pattern)
* Sequences that have at least 3 consecutive G(like GGG or GGGG)

Return the result table ordered by sample\_id in ascending order.

**Input**

**Samples Table**

|  |  |  |
| --- | --- | --- |
| **sample\_id** | **dna\_sequence** | **species** |
| 1 | ATGCTAGCTAGCTAA | Human |
| 2 | GGGTCAATCATC | Human |
| 3 | ATATATCGTAGCTA | Human |
| 4 | ATGGGGTCATCATAA | Mouse |
| 5 | TCAGTCAGTCAG | Mouse |
| 6 | ATATCGCGCTAG | Zebrafish |
| 7 | CGTATGCGTCGTA | Zebrafish |

**Output**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **sample\_id** | **dna\_sequence** | **species** | **has\_start** | **has\_stop** | **has\_atat** | **has\_ggg** |
| 1 | ATGCTAGCTAGCTAA | Human | 1 | 1 | 0 | 0 |
| 2 | GGGTCAATCATC | Human | 0 | 0 | 0 | 1 |
| 3 | ATATATCGTAGCTA | Human | 0 | 0 | 1 | 0 |
| 4 | ATGGGGTCATCATAA | Mouse | 1 | 1 | 0 | 1 |
| 5 | TCAGTCAGTCAG | Mouse | 0 | 0 | 0 | 0 |
| 6 | ATATCGCGCTAG | Zebrafish | 0 | 1 | 1 | 0 |
| 7 | CGTATGCGTCGTA | Zebrafish | 0 | 0 | 0 | 0 |

**Solution**

**SELECT \* ,IF(dna\_sequence REGEXP ’^ATG’,1,0) AS has\_start,**

**IF(dna\_sequence REGEXP ‘(TAA|TAG|TGA)$’,1,0) AS has\_stop,**

**IF(dna\_sequence REGEXP ‘GGG’,1,0) AS has\_999**

**FROM Samples**

**ORDER BY sample\_id;**

**3.Customer purchasing behaviour**

|  |  |
| --- | --- |
| Column\_Name | Type |
| transaction\_id | int |
| customer\_id | int |
| product\_id | int |
| transaction\_date | date |
| amount | decimal |

transaction\_id is the unique identifier for this table.

Each row of this table contains information about a transaction , including the customer ID , product ID, date , and amount spent.

Table:Products

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| category | varchar |
| price | decimal |

product\_id is the unique identifier for this table .

Each row of this table contains information about a product, including its category and price.

Write a solution to analyze customer purchasing behaviour . For each customer , calculate :

* The total amount spent
* The number of transactions
* The number of unique product categories purchased
* The average amount spent
* The mostly frequently purchased product category(if there is a tie , choose the one with the most recent transaction)

A loyalty score defined as:(Number of transactions\*10)+(Total amount spent/100).

Round total\_amount , avg\_transaction\_amount, and loyalty\_score to 2 decimal places

Return the result table ordered\_by loyalty\_score in descending order, then by customer\_id in ascending order.

**Input**

**Transactions Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **transaction\_id** | **customer\_id** | **product\_id** | **transaction\_date** | **amount** |
| 1 | 101 | 1 | 2023-01-01 | 100.00 |
| 2 | 101 | 2 | 2023-01-15 | 150.00 |
| 3 | 102 | 1 | 2023-01-01 | 100.00 |
| 4 | 102 | 3 | 2023-01-22 | 200.00 |
| 5 | 101 | 3 | 2023-02-10 | 200.00 |

**Products Table**

|  |  |  |
| --- | --- | --- |
| **product\_id** | **category** | **price** |
| 1 | A | 100.00 |
| 2 | B | 150.00 |
| 3 | C | 200.00 |

**Output**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **customer\_id** | **total\_amount** | **transaction\_count** | **unique\_categories** | **avg\_transaction\_amount** | **top\_category** | **loyalty\_score** |
| 101 | 450.00 | 3 | 3 | 150.00 | C | 34.50 |
| 102 | 300.00 | 2 | 2 | 150.00 | C | 23.00 |

**Solution**

**SELECT T.\* , P.CATEGORY , COUNT(\*) OVER (PARTITION BY CUSTOMER\_ID , CATEGORY) AS cnt**

**FROM TRANSACTIONS AS t**

**LEFT JOIN PRODUCTS AS p**

**ON T.PRODUCT\_ID =P.PRODUCT\_ID),**

**SELECT \* , ROW\_NUMBER() OVER (PARTITION BY CUSTOMER\_ID ORDER BY cnt DESC, TRANSACTION\_DATE DESC)**

**AS rnk**

**FROM cte)**

**SELECT CUSTOMER\_ID, ROUND(SUM(AMOUNT),2) AS TOTAL\_AMOUNT, COUNT(TRANSACTION\_ID) AS TRANSACTION\_COUNT, COUNT( DISTINCT CATEGORY) AS UNIQUE\_CATEGORIES , ROUND(SUM(AMOUNT)/COUNT(TRANSACTION\_ID),2) AS AVG\_TRANSACTION\_AMOUNT ,MAX(CASE WHEN rnk =1 THEN CATEGORY ELSE NULL END), ROUND((COUNT(TRANSACTION\_ID)\*10) + (SUM(AMOUNT/100),2) AS LOYALTY\_SCORE**

**FOM cte2;**

**GROUP BY CUSTOMER\_ID**

**ORDER BY LOYALTY\_SCORE DESC , CUSTOMER\_ID ;**

**4.Find overlapping shifts II**

|  |  |
| --- | --- |
| Column Name | Type |
| employee\_id | int |
| start\_time | datetime |
| end\_time | datetime |

(employee\_id , start\_time) is the unique key for this table.

This table contains information about the shifts worked by employees , including the start time , and end time.

Write a solution to analyze overlapping shifts for each employee .Two shifts are considered overlapping if they occur on the same date and one shifts end\_time is later than another shifts start\_time.

* For each employees , calculate the following
* The maximum number of shifts that overlap at any given time
* The total duration of all overlaps in minutes

Return the result table ordered by employee\_id in ascending order

**Input:**

|  |  |  |
| --- | --- | --- |
| **employee\_id** | **start\_time** | **end\_time** |
| 1 | 2023-10-01 09:00:00 | 2023-10-01 17:00:00 |
| 1 | 2023-10-01 15:00:00 | 2023-10-01 23:00:00 |
| 1 | 2023-10-01 16:00:00 | 2023-10-02 00:00:00 |
| 2 | 2023-10-01 09:00:00 | 2023-10-01 17:00:00 |
| 2 | 2023-10-01 11:00:00 | 2023-10-01 19:00:00 |
| 3 | 2023-10-01 09:00:00 | 2023-10-01 17:00:00 |

**Output**

|  |  |  |
| --- | --- | --- |
| **employee\_id** | **max\_overlapping\_shifts** | **total\_overlap\_duration** |
| 1 | 3 | 600 |
| 2 | 2 | 360 |
| 3 | 1 | 0 |

**Solution**

**WITH cte AS**

**(SELECT E1.EMPLOYEE\_ID , E2.START\_TIME , COUNT(\*) AS cnt,**

**CASE WHENE1.START\_TIME <> E2.START\_TIME THEN TIMESTAMPDIFF(MINUTE , E1.START\_TIME ,E2.END\_TIME) ELSE 0 END) AS DURATION**

**FROM EMPLOYEESHIFTS AS E1**

**LEFT JOIN EMPLOYEESHIFTS AS E2**

**ON E1.EMPLOYEE\_ID =E2.EMPLOYEE\_ID**

**AND E1.START\_TIME BETWEEN E2.START\_TIME AND E2.END\_TIME**

**GROUP BY E1.EMPLOYEE\_ID , E2.START\_TIME)**

**SELECT EMPLOYEE\_ID , MAX(cnt) AS MAX\_OVERLAPPING\_SHIFTS,**

**SUM(DURATION) AS TOTAL\_OVERLAP\_DURATION**

**FROM cte**

**GROUP BY EMPLOYEE\_ID**

**ORDER BY EMPLOYEE\_ID;**

5.Find overlapping shifts

|  |  |
| --- | --- |
| Column Name | Type |
| employee\_id | int |
| start\_time | time |
| end\_time | time |

(employee\_id , start\_time) is the unique key for this table.

This table contains information about the shifts worked by employees , including the start time , and end times on a specific date.

Write a solution to count the number of overlapping shifts for each employee .Two shifts are considered overlapping if one shifts end\_time is later than another shifts start\_time.

Return the result table ordered by employee\_id in ascending order

**Input:**

|  |  |  |
| --- | --- | --- |
| **employee\_id** | **start\_time** | **end\_time** |
| 1 | 08:00:00 | 12:00:00 |
| 1 | 11:00:00 | 15:00:00 |
| 1 | 14:00:00 | 18:00:00 |
| 2 | 09:00:00 | 17:00:00 |
| 2 | 16:00:00 | 20:00:00 |
| 3 | 10:00:00 | 12:00:00 |
| 3 | 13:00:00 | 15:00:00 |
| 3 | 16:00:00 | 18:00:00 |
| 4 | 08:00:00 | 10:00:00 |
| 4 | 09:00:00 | 11:00:00 |

**Output:**

|  |  |
| --- | --- |
| **employee\_id** | **overlapping\_shifts** |
| 1 | 2 |
| 2 | 1 |
| 4 | 1 |

**Solution**

**SELECT E1.EMPLOYEE\_ID , COUNT(\*) AS OVERLAPPING\_SHIFTS**

**FROM EMPLOYEESHIFTS AS E1**

**INNER JOIN EMPLOYEESHIFTS AS E2**

**ON E1.EMPLOYEE\_ID =E2**

**ON E1.EMPLOYEE\_ID =E2.EMPLOYEE\_ID**

**AND E1.START\_TIME > E2.START\_TIME**

**AND E1.START\_TIME <E2.END\_TIME;**

**6.Find products with three consecutive digits**

|  |  |
| --- | --- |
| Column Name | Type |
| Product\_id | int |
| Name | varchar |

(product\_id) is the unique key for this table.

Each row of this table contains the id and name of a product.

Write a solution to find all products whose names contain a sequence of exactly three digits in a row

Return the result table ordered by product\_id in ascending order.

**Input:**

|  |  |
| --- | --- |
| **product\_id** | **name** |
| 1 | ABC123XYZ |
| 2 | A12B34C |
| 3 | Product56789 |
| 4 | NoDigitsHere |
| 5 | 789Product |
| 6 | Item003Description |
| 7 | Product12X34 |

**Output:**

|  |  |
| --- | --- |
| **product\_id** | **name** |
| 1 | ABC123XYZ |
| 5 | 789Product |
| 6 | Item003Description |

**SELECT \***

**FROM PRODUCTS**

**WHERE REGEXP\_LIKE (NAME ,’[0-9]{3}’)**

**AND NOT REGEXP\_LIKE(NAME , ‘[0-9]{4,}’)**

**ORDER BY PRODUCT\_ID ;**

**7.Find valid emails**

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | int |
| Email | varchar |

(user\_id) is the unique key for this table.

Each row contains a user’s unique id and email address.

Write a solution to find all the valid email addresses , A valid email address meets the following criteria:

* It contains exactly one @ symbol
* It ends with .com
* The part before the @ symbol contains only alphanumeric characters and underscores
* The part after the @ symbol and before .com contains a domain name that contains only letters

Return the result table ordered by user\_id in ascending order.

**Input:**

|  |  |
| --- | --- |
| **user\_id** | **email** |
| 1 | alice@example.com |
| 2 | bob\_at\_example.com |
| 3 | charlie@example.net |
| 4 | david@domain.com |
| 5 | eve@invalid |

**Output:**

|  |  |
| --- | --- |
| **user\_id** | **email** |
| 1 | alice@example.com |
| 4 | david@domain.com |

**SELECT \***

**FROM USERS**

**WHERE REGEXP\_LIKE(EMAIL,** [**‘[a-zA-Z0-9\_]+@[a-zA-Z]+\.com$’)**](mailto:‘[a-zA-Z0-9_]+@[a-zA-Z]+/.com$’))

**ORDER BY USER\_ID;**

1. **Team dominance by pass success**

**Table:Teams**

|  |  |
| --- | --- |
| Column Name | Type |
| player\_id | int |
| Team\_name | varchar |

(player\_id) is the unique key for this table.

Each row contains a user’s unique identifier for player and the name of one of the teams participating in that match

**Table:passes**

|  |  |
| --- | --- |
| Column Name | Type |
| pass\_from | int |
| time\_stamp | varchar |
| Pass\_to | int |

(pass\_from , time\_stamp) is the primary key for this table

Pass\_from is a foreign key to player\_id from teams table

Each row represents a pass made during a match , time\_stamp represents a pass made during a match , time\_stamp represents the time in minutes (00:00-90:00) when the pass was made, pass\_to is the player\_id of the player receiving the pass

Write a solution to calculate the dominance score for each team in both halves of the match .the rule are as follows

A match is divided into two halves :first half (00:00-45:00 minutes) and second half (45:01-90:00 minutes)

The dominance score is calculated base on successful and intercepted passes:

When pass\_to is a player from the same team :+1 point

When pass\_to is a player from the opposing team (interception):-1 point

A higher dominance score indicates better passing performance

Return the result table order by team\_name and half\_number in ascending order.

**Input:**

**Teams Table**

|  |  |
| --- | --- |
| **player\_id** | **team\_name** |
| 1 | Arsenal |
| 2 | Arsenal |
| 3 | Arsenal |
| 4 | Chelsea |
| 5 | Chelsea |
| 6 | Chelsea |

**Passes Table**

|  |  |  |
| --- | --- | --- |
| **pass\_from** | **time\_stamp** | **pass\_to** |
| 1 | 00:15 | 2 |
| 2 | 00:45 | 3 |
| 3 | 01:15 | 1 |
| 4 | 00:30 | 1 |
| 2 | 46:00 | 3 |
| 3 | 46:15 | 4 |
| 1 | 46:45 | 2 |
| 5 | 46:30 | 6 |

**Output:**

|  |  |  |
| --- | --- | --- |
| **team\_name** | **half\_number** | **dominance** |
| Arsenal | 1 | 3 |
| Arsenal | 2 | 1 |
| Chelsea | 1 | -1 |
| Chelsea | 2 | 1 |

**SELECT P.\* , T1.TEAM\_NAME , CASE WHEN T1.TEAM\_NAME =T2.TEAM\_NAME**

**THEN 1 ELSE -1 END AS POINTS , CASE WHEN P.TIME\_STAMP<=’45:00’**

**THEN 1 ELSE 2 END AS HALF\_NUMBER**

**FROM PASSES AS P**

**LEFT JOIM TEAMS AS T1**

**ON P.PASS\_FROM=T1.PLAYER\_ID**

**LEFT JOIN TEAMS AS T2**

**ON PASS\_TO =T2.PLAYER\_ID)**

**SELECT TEAM\_NAME , HALF\_NUMBER,SUM(POINTS) AS DOMINANACE**

**FROM cte**

**GROUP BY TEAM\_NAME , HALF\_NUMBER**

**ORDER BY TEAM\_NAME , HALF\_NAME;**

1. Find students who improved

Table:scores

|  |  |
| --- | --- |
| Column Name | Type |
| student\_id | int |
| Subject | varchar |
| Score | int |
| Exam\_date | varchar |

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

Each row contains information about a studnts score in a specific subject on a particular exam date , score is between 0 and 100(inclusive).

Write a solution to find the students who have shown improvement .A student is considered to have shown improvement if they meet both of these conditions

Have taken exams in the same subject on at least two different dates

Their lates score in that subject is higher that their first score

Return the result table ordered by student\_id , subject in ascending **order**

**Input:**

**Scores Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **student\_id** | **subject** | **score** | **exam\_date** |
| 101 | Math | 70 | 2023-01-15 |
| 101 | Math | 85 | 2023-02-15 |
| 101 | Physics | 65 | 2023-01-15 |
| 101 | Physics | 60 | 2023-02-15 |
| 102 | Math | 80 | 2023-01-15 |
| 102 | Math | 85 | 2023-02-15 |
| 103 | Math | 90 | 2023-01-15 |
| 104 | Physics | 75 | 2023-01-15 |
| 104 | Physics | 85 | 2023-02-15 |

**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **student\_id** | **subject** | **first\_score** | **latest\_score** |
| 101 | Math | 70 | 85 |
| 102 | Math | 80 | 85 |
| 104 | Physics | 75 | 85 |

**SELECT S1.\* ,S2.EXAM\_DATE AS NXT\_DATE , S2.SCORE , ROW\_NUMBER() OVER (PARTITION BY STUDENT\_ID , SUBJECT ORDER BY S1.EXAM\_DATE, S2.EXAM\_DATE DESC ) AS rnk AS NXT\_SCORE**

**FROM SCORES AS S1**

**LEFT JOIN SCORES AS S2**

**ON S1.STUDENT\_ID = S2.STUDENT\_ID**

**AND S1.SUBJECT =S2.SUBJECT**

**AND S1.EXAM\_DATE < S2.EXAM\_DATE**

**WHERE S2.EXAM\_DATE IS NOT NULL)**

**SELECT STUDENT\_D , SUBJECT , SCORE AS FIRST\_SCORE , NXT\_SCORE AS LATEST\_SCORE**

**FROM ct**

**WHERE rnk=1**

**AND NXT\_SCORE >SCORE**

**ORDER BY STUDENT\_ID , SUBJECT;**

**10 Hopper company queries II**

**Table:Drivers**

|  |  |
| --- | --- |
| Column Name | Type |
| driver\_id | int |
| Join\_date | date |

Driver\_id is the column with unique values for this table.

Each row of this table contains the drivers ID and the date they joined the Hopper company.

**Table:Rides**

|  |  |
| --- | --- |
| Column Name | Type |
| ride\_id | int |
| user\_id | int |
| requested\_at | date |

Ride\_id is the column with unique values for this table .Each row of this table contains the ID of a ride , the users ID that requested it , and thae day they requested it.

There may be some ride requests in this table that were not accepted

**Table:AcceptedRides**

|  |  |
| --- | --- |
| Column Name | Type |
| ride\_id | int |
| driver\_id | int |
| ride\_distance | int |
| ride\_duration | int |

ride\_id is the column with unique values for this table.

Each row of this table contains some information about an accepted ride.

It is guaranted that each accepted ride exists in the Rides Table

Write a solution to report the percentage of working drivers (working\_percentage) for each month of 2020 where:

=**\*100.0**

**Note** that if the number of available drivers during a month is zero , we consider the working\_percentage to be 0.

Return the result table ordered by month is ascending order , where month is the months number (january is 1 , february is 2, etc.) Round working\_percentage to the nearest 2 decimal places

The result format is in the following example.

**Input:**

**Drivers Table**

|  |  |
| --- | --- |
| **driver\_id** | **join\_date** |
| 10 | 2019-12-10 |
| 8 | 2020-01-13 |
| 5 | 2020-02-16 |
| 7 | 2020-03-08 |
| 4 | 2020-05-17 |
| 1 | 2020-10-24 |
| 6 | 2021-01-05 |

**Rides Table**

|  |  |  |
| --- | --- | --- |
| **ride\_id** | **user\_id** | **requested\_at** |
| 6 | 75 | 2019-12-09 |
| 1 | 54 | 2020-02-09 |
| 10 | 63 | 2020-03-04 |
| 19 | 39 | 2020-04-06 |
| 3 | 41 | 2020-06-03 |
| 13 | 52 | 2020-06-22 |
| 7 | 69 | 2020-07-16 |
| 17 | 70 | 2020-08-25 |
| 20 | 81 | 2020-11-02 |
| 5 | 57 | 2020-11-09 |
| 2 | 42 | 2020-12-09 |
| 11 | 68 | 2021-01-11 |
| 15 | 32 | 2021-01-17 |
| 12 | 11 | 2021-01-19 |
| 14 | 18 | 2021-01-27 |

**AcceptedRides Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **ride\_id** | **driver\_id** | **ride\_distance** | **ride\_duration** |
| 10 | 10 | 63 | 38 |
| 13 | 10 | 73 | 96 |
| 7 | 8 | 100 | 28 |
| 17 | 7 | 119 | 68 |
| 20 | 1 | 121 | 92 |
| 5 | 7 | 42 | 101 |
| 2 | 4 | 6 | 38 |
| 11 | 8 | 37 | 43 |
| 15 | 8 | 108 | 82 |
| 12 | 8 | 38 | 34 |
| 14 | 1 | 90 | 74 |

**Output:**

|  |  |
| --- | --- |
| **month** | **working\_percentage** |
| 1 | 0.00 |
| 2 | 0.00 |
| 3 | 25.00 |
| 4 | 0.00 |
| 5 | 0.00 |
| 6 | 20.00 |
| 7 | 20.00 |
| 8 | 20.00 |
| 9 | 0.00 |
| 10 | 0.00 |
| 11 | 33.33 |
| 12 | 16.67 |

**WITH RECURSIVE cte AS**

**(SELECT 2020 AS YEAR , 1 AS MONTH )**

**UNION**

**SELECT YEAR , MONTH+1 FROM cte**

**WHERE MONTH <12),**

**cte2 AS**

**(SELECT R.RIDE\_ID , R.REQUESTED\_AT , A.DRIVER\_ID**

**FROM EIDES AS R**

**INNER JOIN ACCEPTRIDES AS A**

**USING (RIDE\_ID))**

**SELECT C.MONTH , ROUND(IFNULL(COUNT(DISTINCT C2.DRIVER\_ ID/ COUNT(DISTINCT D.DRIVER\_ID)\*100 ,0),2) AS WORKING\_PERCENTAGE**

**FROM cte AS c**

**LEFT JOIN DRIVERS AS D**

**ON LAST\_DAY(CONCAT(C.YEAR, ‘-’,C.MONTH,’-01’))**

**>= D.JOIN\_DATE**

**LEFT JOIN cte AS C2**

**ON LAST\_DAY(CONCAT(C.YEAR,’-’,C.MONTH ,’-01’))**

**=LAST\_DAY(C2.REQUESTED\_AT)**

**AND D.DRIVER\_ID =C2.DRIVER\_ID**

**GROUP BY C.MONTH**

**ORDER BY C.MONTH;**

**11 BOOKS WITH NULL RATINGS**

|  |  |
| --- | --- |
| Column Name | Type |
| book\_id | int |
| Title | varchar |
| Author | varchar |
| published\_year | int |
| rating | decimal |

book\_id is the unique key for this table.

Each row of this table contains information about a book including its unique ID, title, author, publication year, and rating.

rating can be NULL, indicating that the book hasn't been rated yet.

Return the result table ordered by book\_id in ascending order.

**Input:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **book\_id** | **Title** | **Author** | **Published Year** | **Rating** |
| 1 | The Great Gatsby | F. Scott | 1925 | 4.5 |
| 2 | To Kill a Mockingbird | Harper Lee | 1960 | NULL |
| 3 | Pride and Prejudice | Jane Austen | 1813 | 4.8 |
| 4 | The Catcher in the Rye | J.D. Salinger | 1951 | NULL |
| 5 | Animal Farm | George Orwell | 1945 | 4.2 |
| 6 | Lord of the Flies | William Golding | 1954 | NULL |

**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **book\_id** | **Title** | **Author** | **Published Year** |
| 2 | To Kill a Mockingbird | Harper Lee | 1960 |
| 4 | The Catcher in the Rye | J.D. Salinger | 1951 |
| 6 | Lord of the Flies | William Golding | 1954 |

**SELECT BOOK\_ID, TITLE, AUTHER, PUBLISHED\_YEAR**

**FROM BOOKS**

**WHERE RATING IS NULL**

**ORDER BY BOOK\_ID;**

**12 Find Candidates for Data Scientist Position II**

**Table:Candidates**

|  |  |
| --- | --- |
| Column Name | Type |
| candidate\_id | int |
| Skill | varchar |
| Proficiency | int |

(candidate\_id, skill) is the primary key for this table.

Each row includes candidate\_id, required skill, and its importance (1-5) for the project.

**Table:Projects**

|  |  |
| --- | --- |
| Column Name | Type |
| project\_id | int |
| Skill | varchar |
| Importance | int |

(project\_id, skill) is the primary key for this table.

Each row includes project\_id, required skill, and its importance (1-5) for the project.

Leetcode is staffing for multiple data science projects. Write a solution to find the **best candidate** for**each project** based on the following criteria:

1. Candidates must have **all** the skills required for a project.
2. Calculate a **score** for each candidate-project pair as follows:
   * **Start** with 100 points
   * **Add** 10 points for each skill where **proficiency > importance**
   * **Subtract** 5 points for each skill where **proficiency < importance**

Include only the top candidate (highest score) for each project. If there’s a **tie**, choose the candidate with the **lower** candidate\_id. If there is **no suitable candidate** for a project, **do not return** that project.

Return a result table ordered by project\_id in ascending order.

**Input:**

**Candidates Table**

|  |  |  |
| --- | --- | --- |
| **candidate\_id** | **skill** | **proficiency** |
| 101 | Python | 5 |
| 101 | Tableau | 3 |
| 101 | PostgreSQL | 4 |
| 101 | TensorFlow | 2 |
| 102 | Python | 4 |
| 102 | Tableau | 5 |
| 102 | PostgreSQL | 4 |
| 102 | R | 4 |
| 103 | Python | 3 |
| 103 | Tableau | 5 |
| 103 | Tableau | 3 |
| 103 | PostgreSQL | 5 |
| 103 | Spark | 4 |

**Projects Table**

|  |  |  |
| --- | --- | --- |
| **project\_id** | **skill** | **importance** |
| 501 | Python | 4 |
| 501 | Tableau | 3 |
| 501 | PostgreSQL | 5 |
| 502 | Python | 3 |
| 502 | Tableau | 4 |
| 502 | R | 2 |

**Output:**

|  |  |  |
| --- | --- | --- |
| **project\_id** | **candidate\_id** | **score** |
| 501 | 101 | 105 |
| 502 | 102 | 130 |

**WITH cte AS**

**(SELECT P.PROJECT\_ID, C.CANDIDATE\_ID,100 + SUM(CASE**

**WHEN C.PROFICIENCY > P,IMPORTANCE THEN 10**

**WHEN C.PROFICIENCY < P,IMPORTANCE THEN 10**

**ELSE 0 END ) AS SCORE**

**COUNT (C.SKILLS) AS SKLL\_CNT**

**FROM PROJECTS AS P**

**LEFT JOIN CANDIDATE AS C**

**USING (SKILLS)**

**GROUP BY P.PROJECT\_ID, C.CANDIDATE\_ID);**

**cte AS**

**(SELECT \*, DENSE\_RANK ( ) OVER ( PARTITION BY PROJECT\_ID**

**ORDER BY SCORE DESC, CANDIDATE\_ID) AS RNK**

**FROM cte**

**WHERE ( PROJECT\_ID, SKLL\_CNT ) IN**

**(SELECT PROJECT\_ID, COUNT(SKILL)**

**FROM PROJECTS GROUP BY PROJECTS\_ID))**

**SELECT PROJECT\_ID, CANDIDATE\_ID, SCORE**

**FROM cte2**

**WHERE RNK=1**

**ORDER BY PROJECT\_ID;**

**13 CEO SUBORDINATE HIERARCHY**

**Table:** **Employees**

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

employee\_id is the unique identifier for this table.

manager\_id is the employee\_id of the employee's manager. The CEO has a NULL manager\_id.

Write a solution to find subordinates of the CEO (both **direct** and **indirect**), along with their **level in the hierarchy** and their **salary difference** from the CEO.

The result should have the following columns:

The query result format is in the following example.

* subordinate\_id: The employee\_id of the subordinate
* subordinate\_name: The name of the subordinate
* hierarchy\_level: The level of the subordinate in the hierarchy (1 for **direct** reports, 2 for **their direct** reports, and **so on**)
* salary\_difference: The difference between the subordinate's salary and the CEO's salary

Return *the result table ordered by* hierarchy\_level ***ascending***, *and then by* subordinate\_id ***ascending***.

**Input :**

**Employees Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **employee\_id** | **employee\_name** | **manager\_id** | **salary** |
| 1 | Alice | NULL | 150000 |
| 2 | Bob | 1 | 120000 |
| 3 | Charlie | 1 | 110000 |
| 4 | David | 2 | 105000 |
| 5 | Eve | 2 | 100000 |
| 6 | Frank | 3 | 95000 |
| 7 | Grace | 3 | 98000 |
| 8 | Helen | 5 | 90000 |

**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **subordinate\_id** | **subordinate\_name** | **hierarchy\_level** | **salary\_difference** |
| 2 | Bob | 1 | -30000 |
| 3 | Charlie | 1 | -40000 |
| 4 | David | 2 | -45000 |
| 5 | Eve | 2 | -50000 |
| 6 | Frank | 2 | -55000 |
| 7 | Grace | 2 | -52000 |
| 8 | Helen | 3 | -60000 |

**WITH RECURSIVE cte AS**

**( ----Anchor Member**

**SELECT \*, 0 AS HIERARCHY\_LEVEL**

**FROM EMPLOYEES**

**WHERE MANAGER\_ID IS NULL**

**UNION ALL**

**----Recursive Member**

**SELECT**

**E.EMPLOYEE\_\*,**

**E.HIERARCHY\_LEVEL + 1 AS HIERARCHY\_LEVEL,**

**FROM EMPLOYEES AS E**

**INNER JOIN cte AS C**

**ON E.MANAGER\_ID=C.EMPLOYEE\_ID**

**)**

**SELECT**

**EMPLOYEE\_ID AS SUBORDINATE\_ID,**

**EMPLOYEE\_NAME AS SUBORDINATE\_NAME,**

**HIERARCHY\_LEVEL,**

**SALARY – (SELECT SALARY FROM EMPLOYEES WHERE MANAGER\_ID IS NULL) AS SALARY DIFFERENCE**

**FROM cte**

**WHERE HIERARCHY\_LEVEL> 0**

**ORDER BY HIERARCHY \_LEVEL, SUBORADINATE\_ID;**

**14 FIND TOP SCORING STUDENTS**

**Table: students**

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

student\_id is the primary key (combination of columns with unique values) for this table.

Each row of this table contains the student ID, student name, and their major.

**Table: courses**

|  |  |
| --- | --- |
| Column Name | Type |
| course\_id | int |
| Name | varchar |
| Credits | int |
| Major | varchar |

course\_id is the primary key (combination of columns with unique values) for this table.

Each row of this table contains the course ID, course name, the number of credits for the course, and the major it belongs to.

**Table: enrollments**

|  |  |
| --- | --- |
| Column Name | Type |
| student\_id | int |
| course\_id | int |
| Semester | varchar |
| Grade | varchar |

(student\_id, course\_id, semester) is the primary key (combination of columns with unique values) for this table.

Each row of this table contains the student ID, course ID, semester, and grade received.

Write a solution to find the students who have **taken** **all courses** offered in their major and have achieved a **grade of A** **in all these courses**.

Return *the result table ordered by* student\_id *in****ascending****order*.

**Input:**

**Students Table**

|  |  |  |
| --- | --- | --- |
| **student\_id** | **name** | **major** |
| 1 | Alice | Computer Science |
| 2 | Bob | Computer Science |
| 3 | Charlie | Mathematics |
| 4 | David | Mathematics |

**Courses Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **course\_id** | **name** | **credits** | **major** |
| 101 | Algorithms | 3 | Computer Science |
| 102 | Data Structures | 3 | Computer Science |
| 103 | Calculus | 4 | Mathematics |
| 104 | Linear Algebra | 4 | Mathematics |

**Enrollments Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **student\_id** | **course\_id** | **semester** | **grade** |
| 1 | 101 | Fall 2023 | A |
| 1 | 102 | Fall 2023 | A |
| 2 | 101 | Fall 2023 | B |
| 2 | 102 | Fall 2023 | A |
| 3 | 103 | Fall 2023 | A |
| 3 | 104 | Fall 2023 | A |
| 4 | 103 | Fall 2023 | A |
| 4 | 104 | Fall 2023 | B |

**Output:**

|  |
| --- |
| **student\_id** |
| 1 |
| 3 |

**SELECT S.STUDENT\_ID**

**FROM STUDENTS AS S**

**LEFT JOIN COURSES AS C**

**USING (MAJOR)**

**LEFT JOIN ENROLLMENTS AS E**

**ON S.STUDENT\_ID = E.STUDENT\_ID**

**AND C.COURSE\_ID = E.COURSE\_ID**

**GROUP BY S.STUDENT\_ID**

**HAVING COUNT (DISTINCT C.COURSE\_ID) = SUM(IF(E.GRADE = ‘A’ , 1,0))**

**ORDER BY S.STUDENT\_ID;**

**15 FIND TOP SCORING STUDENTS II**

**Table: students**

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

student\_id is the primary key for this table.

Each row contains the student ID, student name, and their major.

**Table: courses**

|  |  |
| --- | --- |
| Column Name | Type |
| course\_id | int |
| Name | varchar |
| Credits | int |
| Major | varchar |
| mandatory | enum |

course\_id is the primary key for this table. mandatory is an enum type of ('Yes', 'No').

Each row contains the course ID, course name, credits, major it belongs to, and whether the course is mandatory.

**Table: enrollments**

|  |  |
| --- | --- |
| Column Name | Type |
| student\_id | int |
| course\_id | int |
| Semester | varchar |
| Grade | varchar |
| GPA | decimal |

(student\_id, course\_id, semester) is the primary key (combination of columns with unique values) for this table.

Each row contains the student ID, course ID, semester, and grade received.

Write a solution to find the students who meet the following criteria:

* Have**taken all mandatory courses** and **at least two** elective courses offered in **their major.**
* Achieved a grade of **A** in **all mandatory courses** and at least **B** in**elective courses**.
* Maintained an average GPA of at least 2.5 across all their courses (including those outside their major).

Return *the result table ordered by* student\_id *in****ascending****order*.

**Input:**

**Students Table**

|  |  |  |
| --- | --- | --- |
| **student\_id** | **name** | **major** |
| 1 | Alice | Computer Science |
| 2 | Bob | Computer Science |
| 3 | Charlie | Mathematics |
| 4 | David | Mathematics |

**Courses Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **course\_id** | **name** | **credits** | **major** | **mandatory** |
| 101 | Algorithms | 3 | Computer Science | yes |
| 102 | Data Structures | 3 | Computer Science | yes |
| 103 | Calculus | 4 | Mathematics | yes |
| 104 | Linear Algebra | 4 | Mathematics | yes |
| 105 | Machine Learning | 3 | Computer Science | no |
| 106 | Probability | 3 | Mathematics | no |
| 107 | Operating Systems | 3 | Computer Science | no |
| 108 | Statistics | 3 | Mathematics | no |

**Enrollments Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **student\_id** | **course\_id** | **semester** | **grade** | **GPA** |
| 1 | 101 | Fall 2023 | A | 4.0 |
| 1 | 102 | Spring 2023 | A | 4.0 |
| 1 | 105 | Spring 2023 | A | 4.0 |
| 1 | 107 | Fall 2023 | B | 3.5 |
| 2 | 101 | Fall 2023 | A | 4.0 |
| 2 | 102 | Spring 2023 | B | 3.0 |
| 3 | 103 | Fall 2023 | A | 4.0 |
| 3 | 104 | Spring 2023 | A | 4.0 |
| 3 | 106 | Spring 2023 | A | 4.0 |
| 3 | 108 | Fall 2023 | B | 3.5 |
| 4 | 103 | Fall 2023 | B | 3.0 |
| 4 | 104 | Spring 2023 | B | 3.0 |

**Output:**

|  |
| --- |
| **student\_id** |
| 1 |
| 3 |

**WITH cte AS**

**(SELECT S.STUDENT\_ID**

**FROM STUDENTS AS S**

**LEFT JOIN COURSES AS C**

**USING (MAJOR)**

**LEFT JOIN ENROLLMENTS AS E**

**ON S.STUDENT\_ID = E.STUDENT\_ID**

**AND C.COURSE\_ID =E.COURSE\_ID**

**GROUP BY S.STUDENT**

**HAVING SUM (IF(C.MANDATORY = ‘YES’,1 ,0) \* IF (E.GRADE= ‘A’,1,0))**

**AND SUM (IF(C.MANDATORY = ‘NO’,1 ,0) \* IF (E.GRADE IN( ‘A’,’B’),1,0))>=2)**

**SELECT STUDENT\_ID**

**FROM cte**

**WHERE STUDENT\_ID IN(SELECT STUDENT\_ID FROM ENROLLMENTS GROUP BY STUDENT\_ID**

**HAVING AVG(GPA)>=2.5)**

**ORDER BY STUDENT\_ID;**

**16 FIND MEDIAN GIVEN FREQUENCY OF NUMBERS**

**Table: Numbers**

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

num is the primary key (column with unique values) for this table.

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

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

Write a solution to report the median of all the numbers in the database after decompressing the Numbers table. Round the median to one decimal point.

**Input:**

**Numbers Table**

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

**Output:**

|  |
| --- |
| **median** |
| 0.0 |

**WITH RECURSIVE CTE AS(**

**---Anchor  
 SELECT 1 AS NUM  
 UNION ALL**

**--Recursive  
 SELECT NUM+1 FROM cte**

**--Termination**

**WHERE NUM< (SELECT MAX(FREQUENCY) FROM NUMBERS)),**

**cte2 AS**

**(SELECT N.NUM, ROW\_NUMBER ( ) OVER(ORDER BY N.NUM)**

**AS RNK, COUNT(\*) OVER ( ) AS TOTAL\_NUM**

**FROM NUMBERS AS N**

**LEFT JOIN cte AS C**

**ON N.FREQUENCY>= C.NUM),**

**Cte3 AS**

**(SELECT \*, CASE WHEN TOTAL\_NUM %2=0 THEN RNK IN (TOTAL\_NUM/2,(TOTAL\_NUM/2)+1)**

**ELSE RNK=(TOTAL\_NUM+1)/2 END AS CONSIDER**

**FROM cte2)**

**SELECT ROUND(AVG(NUM),1) AS MEDIAN**

**FROM cte3**

**WHERE CONSIDER=1;**

**17 USER PURCHASE PLATFORM**

**Table: Spending**

|  |  |
| --- | --- |
| Column Name | Type |
| User\_id | int |
| Spend\_date | date |
| platform | Enum |
| amount | int |

The table logs the spendings history of users that make purchases from an online shopping website which has a desktop and a mobile application.

(user\_id, spend\_date, platform) is the primary key of this table.

The platform column is an ENUM type of ('desktop', 'mobile').

Write a solution to find the total number of users and the total amount spent using mobile **only**, **desktop only** and **both** mobile and desktop together for each date.

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

**Input:**

**Spending Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **user\_id** | **spend\_date** | **platform** | **amount** |
| 1 | 2019-07-01 | mobile | 100 |
| 1 | 2019-07-01 | desktop | 100 |
| 2 | 2019-07-01 | mobile | 100 |
| 2 | 2019-07-02 | mobile | 100 |
| 3 | 2019-07-01 | desktop | 100 |
| 3 | 2019-07-02 | desktop | 100 |

**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **spend\_date** | **platform** | **total\_amount** | **total\_users** |
| 2019-07-01 | desktop | 100 | 1 |
| 2019-07-01 | mobile | 100 | 1 |
| 2019-07-01 | both | 200 | 1 |
| 2019-07-02 | desktop | 100 | 1 |
| 2019-07-02 | mobile | 100 | 1 |
| 2019-07-02 | both | 0 | 0 |

**WITH cte AS**

**(SELECT SPEND\_DATE, USER\_ID, GROUP\_CONCAT(**

**DISTINCT PLATFORM ORDER BY PLATFORM**

**SEPARATOR ‘,’) AS P, SUM(AMOUNT) AS TOTAL**

**FROM SPENDING**

**GROUP BY SPENDA\_DATE, USER\_ID,**

**cte2 AS**

**(SELECT SPEMD\_DATE,IF(P=’DESKTOP , MOBILE’, ‘BOTH’,P) AS PLATFORM,**

**SUM(TOTAL) AS TOTAL\_AMOUNT,**

**COUNT(DISTINCT USER\_ID) AS TOTAL\_USERS**

**FROM cte**

**GROUP BY SPEND\_DATE, IF(P=’DESKTOP, MOBILE’, ‘BOTH’,P));**

**cte3 AS**

**(SELECT DISTINCT S.SPEND\_DATE,J.PLATFORM**

**FROM SPENDING AS S**

**CROSS JOIN(SELECT ‘MOBILE’ AS PLATFORM**

**UNION SELECT ‘DESKTOP’**

**UNION SELECT ‘BOTH’ ) AS J)**

**SELECT \***

**FROM cte2**

**UNION**

**SELECT \*,0,0**

**FROM cte3**

**WHERE CONCAT(SPEND\_DATE,PLATFORM) NOT IN (SELECT CONCAT( SPEND\_DATE,PLATFORM) FROM cte2 );**

**18 MONTHLY TRANSACTIONS**

**Table: Transactions**

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| country | varchar |
| state | Enum |
| amount | int |
| Trans\_date | date |

id is the column of unique values of this table.

The table has information about incoming transactions.

The state column is an ENUM (category) of type ["approved", "declined"].

**Table: Chargebacks**

|  |  |
| --- | --- |
| Column Name | Type |
| trans\_id | int |
| Trans\_date | date |

Chargebacks contains basic information regarding incoming chargebacks from some transactions placed in Transactions table.

trans\_id is a foreign key (reference column) to the id column of Transactions table.

Each chargeback corresponds to a transaction made previously even if they were not approved.

Write a solution to find for each month and country: the number of approved transactions and their total amount, the number of chargebacks, and their total amount.

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

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

**Input:**

**Transactions Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **id** | **country** | **state** | **amount** | **trans\_date** |
| 101 | US | approved | 1000 | 2019-05-18 |
| 102 | US | declined | 2000 | 2019-05-19 |
| 103 | US | approved | 3000 | 2019-06-10 |
| 104 | US | declined | 4000 | 2019-06-13 |
| 105 | US | approved | 5000 | 2019-06-15 |

**Chargebacks Table**

|  |  |
| --- | --- |
| **trans\_id** | **trans\_date** |
| 102 | 2019-05-29 |
| 101 | 2019-06-30 |
| 105 | 2019-09-18 |

**Output:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **month** | **country** | **approved\_count** | **approved\_amount** | **chargeback\_count** | **chargeback\_amount** |
| 2019-05 | US | 1 | 1000 | 1 | 2000 |
| 2019-06 | US | 2 | 8000 | 1 | 1000 |
| 2019-09 | US | 0 | 0 | 1 | 5000 |

**WITH trans AS**

**(SELECT DATE\_FORMAT(TRANS\_DATE,’%Y - %M’) AS MONTH,**

**COUNTRY, SUM(IF(STATE=’APPROVED’,1,0) AS APPROVED\_COUNT,**

**SUM(IF(STATE= ‘APPROVED’,AMOUNT,0) ) AS APPROVED\_AMOUNT**

**FROM TRANSACTIONS**

**GROUP BY DATE\_FORMAT(TRANS\_DATE, ‘%Y - %M’),COUNTRY),**

**cgbks AS**

**( SELECT DATE\_FORAMT(C.TRANS\_DATE, ‘%Y - %M’) AS MONTH,**

**COUNTRY, COUNT(\*) AS CHARGEBACK\_COUNT , SUM(AMOUNT)**

**AS CHARGEBACK\_AMOUNT**

**FROM CHARGEBACKS AS C**

**LEFT JOIN TRANSACTIONS AS T**

**ON C.TRANS\_ID =T.ID**

**GROUP BY DATE\_FORMAT(C.TRANS\_DATE, ‘%Y - %M’),COUNTRY),**

**result AS**

**(SELECT T.\*,IFNULL(C.CHARGEBACK\_COUNT,0) AS**

**CHARGEBACK\_COUNT,IFNULL(C.CHARGEBACK\_AMOUNT,0) AS**

**CHARGEBACK\_AMOUNT**

**FROM TRANS AS T**

**LEFT JOIN cgbs AS C**

**ON T.MONTH =C.MONTH**

**AND T.COUNTRY = C.COUNTRY**

**UNION**

**SELECT C.MONTH ,C.COUNTRY,IFNULL(T.APPROVED\_COUNT,0)**

**AS APPROVED\_COUNT, IFNULL(T.APPROVED\_AMOUNT,0) AS APPROVED\_AMOUNT, C.CHARGEBACK\_COUNT, C.CHARGEBACK\_AMOUNT**

**FROM trans AS T**

**RIGHT JOIN CGBKS AS C**

**ON T.MONTH =C.MONTH**

**AND T.COUNTRY =C.COUNTRY)**

**SELECT \***

**FROM RESULT**

**WHERE APPROVED\_COUNT+APPROVED\_AMOUNT+CHARGEBACK-COUNT + CHARGEBACK\_AMOUNT>0);**

**19 NUMBER OF TRANSACTIONS PER VISIT**

**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 |

This table may contain duplicates rows.

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.

**Input:**

**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 |

**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 |

**Output:**

|  |  |
| --- | --- |
| **transactions\_count** | **visits\_count** |
| 0 | 4 |
| 1 | 5 |
| 2 | 0 |
| 3 | 1 |

**WITH RECURSIVE cte AS**

**(SELECT V.VISIT\_DATE,V.USER\_ID, SUM(CASE WHEN T.AMOUNT IS NULL THEN 0 ELSE 1 END) AS NUM\_TRAN**

**FROM VISITS AS V**

**LEFT JOIN TRANSACTIONS AS T**

**ON V.USER\_ID= T.USER\_ID**

**AND V.VISIT\_DATE = T.TRANSACTION\_DATE**

**GROUP BY V.VISIT\_DATE, V.USER\_ID),**

**cte2 AS**

**(SELECT 0 AS NUM\_TRAN**

**UNION**

**SELECT NUM\_TRAN+1**

**FROM cte2**

**WHERE NUM\_TRAN < (SELECT MAX(NUM\_TRAN) FROM cte))**

**SELECT cte2.NUM\_TRANS AS TRANSACTION\_COUNT,**

**COUNT(cte.USER\_ID) AS VISITS\_COUNT**

**FROM cte2**

**LEFT JOIN cte**

**ON cte2.NUM\_TRAN=cte.NUM\_TRAN**

**GROUP BY cte2.NUM\_TRAN**

**ORDER BY TRANSACTION\_COUNT;**

**20  HOPPER COMPANY QUERIES I**

**Table: Drivers**

|  |  |
| --- | --- |
| Column Name | Type |
| driver\_id | int |
| join\_date | date |

driver\_id is the primary key for this table.

Each row of this table contains the driver's ID and the date they joined the Hopper company.

**Table: Rides**

|  |  |
| --- | --- |
| Column Name | Type |
| ride\_id | int |
| user\_id | int |
| Requested\_at | date |

ride\_id is the primary key for this table.

Each row of this table contains the ID of a ride, the user's ID that requested it, and the day they requested it.

There may be some ride requests in this table that were not accepted.

**Table: AcceptedRides**

|  |  |
| --- | --- |
| Column Name | Type |
| ride\_id | int |
| driver\_id | int |
| Ride\_distance | int |
| Ride\_duration | int |

ride\_id is the primary key for this table.

Each row of this table contains some information about an accepted ride.

It is guaranteed that each accepted ride exists in the Rides table.

Write an SQL query to report the following statistics for each month of **2020**:

* The number of drivers currently with the Hopper company by the end of the month (active\_drivers).
* The number of accepted rides in that month (accepted\_rides).

Return the result table ordered by month in ascending order, where month is the month's number (January is 1, February is 2, etc.).

**Input:**

**Drivers Table**

|  |  |
| --- | --- |
| **driver\_id** | **join\_date** |
| 10 | 2019-12-10 |
| 8 | 2020-01-13 |
| 5 | 2020-02-16 |
| 7 | 2020-03-08 |
| 4 | 2020-05-17 |
| 1 | 2020-10-24 |
| 6 | 2021-01-05 |

**Rides Table**

|  |  |  |
| --- | --- | --- |
| **ride\_id** | **user\_id** | **requested\_at** |
| 6 | 75 | 2019-12-09 |
| 1 | 54 | 2020-02-09 |
| 10 | 63 | 2020-03-04 |
| 19 | 39 | 2020-04-06 |
| 3 | 41 | 2020-06-03 |
| 13 | 52 | 2020-06-22 |
| 7 | 69 | 2020-07-16 |
| 17 | 70 | 2020-08-25 |
| 20 | 81 | 2020-11-02 |
| 5 | 57 | 2020-11-09 |
| 2 | 42 | 2020-12-09 |
| 11 | 68 | 2021-01-11 |
| 15 | 32 | 2021-01-17 |
| 12 | 11 | 2021-01-19 |
| 14 | 18 | 2021-01-27 |

**AcceptedRides Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **ride\_id** | **driver\_id** | **ride\_distance** | **ride\_duration** |
| 10 | 10 | 63 | 38 |
| 13 | 10 | 73 | 96 |
| 7 | 8 | 100 | 28 |
| 17 | 7 | 119 | 68 |
| 20 | 1 | 121 | 92 |
| 5 | 7 | 42 | 101 |
| 2 | 4 | 6 | 38 |
| 11 | 8 | 37 | 43 |
| 15 | 8 | 108 | 82 |
| 12 | 8 | 38 | 34 |
| 14 | 1 | 90 | 74 |

**Output:**

|  |  |  |
| --- | --- | --- |
| **month** | **active\_drivers** | **accepted\_rides** |
| 1 | 2 | 0 |
| 2 | 3 | 0 |
| 3 | 4 | 1 |
| 4 | 4 | 0 |
| 5 | 5 | 0 |
| 6 | 5 | 1 |
| 7 | 5 | 1 |
| 8 | 5 | 1 |
| 9 | 5 | 0 |
| 10 | 6 | 0 |
| 11 | 6 | 2 |
| 12 | 6 | 1 |

**WITH RECURSIVE cte AS**

**(**

**SELECT 2020 AS YEAR, 1 AS MONTH**

**UNION SELECT YEAR, MONTH+1 FROM cte**

**WHERE MONTH<12**

**),**

**cte2 AS**

**(SELECT C.MONTH,COUNT(D.DRIVER\_ID) AS**

**ACTIVE\_DRIVERS**

**FROM cte AS C**

**LEFT JOIN DRIVERS AS D**

**ON LAST\_DAY(CONCAT(C.YEAR, ‘-‘,C.MONTH, -‘01’))**

**>= D.JOIN\_DATE**

**GROUP BY C.MONTH),**

**Cte3 AS**

**( SELECT MONTH(R.REQUESTED\_AT) AS MONTH,**

**COUNT (R.RIDE\_ID) AS ACCEPTED\_RIDES**

**FROM RIDES AS R**

**INNER JOIN ACCEPTEDRIDES AS A**

**USING (RIDE\_ID)**

**WHERE YEAR(R.REQUESTED\_AT) =2020**

**GROUP BY MONTH(R.REQUESTED-AT))**

**21 Friends with no mutual Friends**

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id1 | int |
| User\_id2 | int |

(user\_id1 , user\_id2 ) is te primary key

(combination of columns with the unique values ) for this table

Each row contains user\_id1 , user\_id2 , both of whom are friends with each other

Write a solution to find all pairs of users who are friends with each other and have no mutual friends

Return the result table ordered by user\_id1 , user\_id2 in ascending order

**Input:**

**Friends Table**

|  |  |
| --- | --- |
| **user\_id1** | **user\_id2** |
| 1 | 2 |
| 2 | 3 |
| 2 | 4 |
| 1 | 5 |
| 6 | 7 |
| 3 | 4 |
| 2 | 5 |
| 8 | 9 |

**Output:**

|  |  |
| --- | --- |
| **user\_id1** | **user\_id2** |
| 6 | 7 |
| 8 | 9 |

**WITH cte AS**

**SELECT \* FROM FRIENDS**

**UNION**

**SELECT USER\_ID2 , USER\_ID1 FROM FRIENDS ),**

**cte2 AS**

**(SELECT C1.\* ,C2.USER\_ID2 AS F1 , C3.USER\_ID2 AS F2**

**FROM cte AS C1**

**LEFT JOIN AS C2**

**ON C1..USER\_ID1=C2.USER\_ID1**

**LEFT JOIN cte AS C3**

**ON C1.USER\_ID2=C3.USER\_ID1 WHERE C2.USER\_ID2=C3.USER\_ID2)**

**SELECT \***

**FROM FRIENDS**

**WHERE (USER\_ID1 , USER\_ID2) NOT IN**

**(SELECT USER\_ID1 , USER\_ID2 FROM cte2)**

**ORDER BY USER\_ID1 , USER\_ID2;**

**22 Viewers Turned Streamers**

**Table:Sessions**

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | int |
| session\_start | datetime |
| session\_end | datetime |
| session\_id | int |
| session\_type | enum |

session\_id is column of unique values for this table

session\_type is an ENUM(category) type of (Viewer , Streamer).

This table contains user id , session start , session end , session id and session type

Write a solution to find the number of streaming sessions for users whose first session was a viewer .

Return the result table ordered by count of streaming sessions , user\_id in descending order.

**Input:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **user\_id** | **session\_start** | **session\_end** | **session\_id** | **session\_type** |
| 101 | 2023-11-06 13:53:42 | 2023-11-06 14:05:42 | 375 | Viewer |
| 101 | 2023-11-22 16:45:21 | 2023-11-22 20:39:21 | 594 | Streamer |
| 102 | 2023-11-16 13:23:09 | 2023-11-16 16:10:09 | 777 | Streamer |
| 102 | 2023-11-17 13:23:09 | 2023-11-17 16:10:09 | 778 | Streamer |
| 101 | 2023-11-20 07:16:06 | 2023-11-20 08:33:06 | 315 | Streamer |
| 104 | 2023-11-27 03:10:49 | 2023-11-27 03:30:49 | 797 | Viewer |
| 103 | 2023-11-27 03:10:49 | 2023-11-27 03:30:49 | 798 | Streamer |

**Output:**

|  |  |
| --- | --- |
| **user\_id** | **sessions\_count** |
| 101 | 2 |

**SELECT USER\_ID , SESSION\_TYPE , ROW\_NUMBER()**

**OVER(PARTTION BY USER\_ID ORDER BY SESSION\_START) AS rnk**

**FROM SESSIONS)**

**SELECT USER\_ID ,COUNT(\*) AS SESSIONS\_COUNT**

**FROM cte**

**WHERE USER\_ID IN**

**(SELECT USER\_ID**

**FROM cte**

**WHERE rnk=1**

**AND SESSION\_TYPE =’VIEWER’) AND STREAM\_TYPE=’STREAMER’**

**GROUP BY USER\_ID**

**ORDER BY SESSIONS\_COUNT DESC, USER\_ID DESC;**

**23 Find top performing driver**

**Table:drivers**

|  |  |
| --- | --- |
| Column Name | Type |
| driver\_id | int |
| Name | varchar |
| Age | int |
| Experience | int |
| Accidents | int |

(driver\_id) is the unique key for this table .

Each row includes a driver’s ID , their name , age , years of driving experience , and the nymber of accidents they have had.

**Table:vehicles**

|  |  |
| --- | --- |
| Column Name | Type |
| vehicle\_id | int |
| driver\_id | int |
| Model | varchar |
| fuel\_type | varchar |
| Mileage | int |

(vehicle\_id, driver\_id , fuel\_type ) is the unique key for this table

Each row includes the vehicle’s ID , the driver who operates it , the model , fuel type , and mileage.

**Table:Trips**

|  |  |
| --- | --- |
| Column Name | Type |
| trip\_id | int |
| vehicle\_id | int |
| Distance | int |
| Duration | int |
| Rating | int |

(trip\_id) is the unique key for this table.

Each row includes a trips ID , the vehicle used , the distance covered (in miles), the trip duration (in minutes), and the passengers rating(1-5).

Uber is analyzing drivers based on their trips .Write a sloution to find the top-performing driver for each fuel type based on the following criteria.

1. A drivers performance is calculated as the average rating across all their trips . Average rating should be rounded to 2 decimal places
2. If two drivers have the same average rating , the driver with the longer toatal distance traveled shd be ranked higher.
3. If there is still s tie , choose the driver with the fewest accidents
4. Return the result table ordered by fuel\_type in ascending order

**Input:**

**Drivers Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **driver\_id** | **name** | **age** | **experience** | **accidents** |
| 1 | Alice | 34 | 10 | 1 |
| 2 | Bob | 45 | 20 | 3 |
| 3 | Charlie | 28 | 5 | 0 |

**Vehicles Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **vehicle\_id** | **driver\_id** | **model** | **fuel\_type** | **mileage** |
| 100 | 1 | Sedan | Gasoline | 20000 |
| 101 | 2 | SUV | Electric | 30000 |
| 102 | 3 | Coupe | Gasoline | 15000 |

**Trips Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **trip\_id** | **vehicle\_id** | **distance** | **duration** | **rating** | **trip\_id** |
| 201 | 100 | 50 | 30 | 5 | 201 |
| 202 | 100 | 30 | 20 | 4 | 202 |
| 203 | 101 | 100 | 60 | 4 | 203 |
| 204 | 101 | 80 | 50 | 5 | 204 |

**Output:**

|  |  |  |  |
| --- | --- | --- | --- |
| **fuel\_type** | **driver\_id** | **rating** | **distance** |
| Electric | 2 | 4.50 | 180 |
| Gasoline | 3 | 5.00 | 100 |

**SELECT V.FUEL\_TYPE , V.DRIVER\_ID ,ROUND( AVG(T.TRIPS),**

1. **AS RATING , SUM(T.DISTANCE) AS DISTANCE**
2. **AVG(D.ACCIDENTS) AS ACCIDENT**

**FROM TRIPS AS T**

**LEFT JOIN VEHICLES AS V**

**ON T.VEHICLE\_ID =V.VEHICLE\_ID**

**LEFT JOIN DRIVERS AS D**

**ON V.DRIVER\_ID=D.DRIVER\_ID**

**GROUP BY V.FUEL\_TYPE , V.DRIVER\_ID),**

**Cte2 AS**

**SELECT \*, ROW\_NUMBER() OVER(PARTITION BY FUEL\_TYPE ORDER BY**

**RATING DESC,DISTANCE DESC , ACCIDENT) AS rnk**

**FROM cte)**

**SELECT FUEL\_TYPE , DRIVER\_ID , RATING , DISTANCE**

**FROM cte2**

**WHERE rnk=1**

**ORDER BY FUEL\_TYPE;**

24 Find cities in each state II

**Table:cities**

|  |  |
| --- | --- |
| Column Name | Type |
| State | varchar |
| City | varchar |

(state, city) is the combination of columns with unique values for this table

Each row of this table contains the state name and the city name within the state.

Write a solution to find all the cities in each state and analyze them based on the following requirements :

* Combine all cities into a comma-separated string for each state
* Only include states that have at least 3 cities
* Only include states where at least one city starts with the same letter as the state name

Return the result table ordered by the count of matching-letter cities in descending order by the count of matching-letter cities in descending order and thenby state name in ascending order.

**Input:**

**Cities Table**

|  |  |
| --- | --- |
| **state** | **city** |
| New York | New York City |
| New York | Newark |
| New York | Buffalo |
| New York | Rochester |
| California | San Francisco |
| California | Sacramento |
| California | San Diego |
| California | Los Angeles |
| Texas | Tyler |
| Texas | Temple |
| Texas | Taylor |
| Texas | Dallas |
| Pennsylvania | Philadelphia |
| Pennsylvania | Pittsburgh |

**Output:**

|  |  |  |
| --- | --- | --- |
| **state** | **cities** | **matching\_letter\_count** |
| Pennsylvania | Philadelphia, Pittsburgh, Pottstown | 3 |
| Texas | Dallas, Taylor, Temple, Tyler | 3 |
| New York | Buffalo, Newark, New York City, Rochester | 2 |

**WITH cte AS**

**(SELECT \* , CASE WHEN LEFT (LOWER(STATE),1)=**

**LEFT(LOWER(CITY),1) THEN 1 ELSE 0 END AS SAME**

**FROM CITIES)**

**SELECT SATE , GROUP\_CONCAT(CITY ORDER BY CITY**

**SEPARATOR ‘,’) AS CITIES , SUM(SAME) AS**

**MATCHING\_LETTER\_COUNT**

**FROM cte**

**GROUP BY STATE**

**HAVING COUNT(CITY)>=3)**

**SELECT \***

**FROM cte2**

**WHERE MATCHING\_LETTER\_COUNT>=1**

**ORDER BY MATCHING\_LETTER\_COUNTDESC , STATE;**

25 Secondary highest salary ||

**Table:employees**

|  |  |
| --- | --- |
| Column Name | Type |
| emp\_id | int |
| Salary | int |
| Dept | varchar |

Emp\_id is the unique key for this table .

Each row of this table contains information abouut an employee including their ID , salary, and department.

Write a solution to find the employees who earn the second\_highestsalary in each department . If multiple employees have the second-highest salary , include all employees with that salary

Return the result table ordered by emp\_id in ascending order.

**Input:**

**Employees Table**

|  |  |  |
| --- | --- | --- |
| **emp\_id** | **salary** | **dept** |
| 1 | 70000 | Sales |
| 2 | 80000 | Sales |
| 3 | 80000 | Sales |
| 4 | 90000 | Sales |
| 5 | 55000 | IT |
| 6 | 65000 | IT |
| 7 | 65000 | IT |
| 8 | 50000 | Marketing |
| 9 | 55000 | Marketing |
| 10 | 55000 | HR |

**Output:**

|  |  |
| --- | --- |
| **emp\_id** | **dept** |
| 2 | Sales |
| 3 | Sales |
| 5 | IT |
| 8 | Marketing |

**WITH cte AS**

**SELECT \* , DENSE\_RANK() OVER(PARTITION BY DEPT ORDER BY SALARY DESC) AS rnk**

**FROM EMPLOYEES**

**SELECT EMP\_ID , DEPT**

**FROM cte**

**WHERE rnk=2**

**ORDER BY EMP\_ID;**

26.Premier league table ranking ||

Table:SeasonStats

|  |  |
| --- | --- |
| Column Name | Type |
| season\_id | int |
| team\_id | int |
| team\_name | varchar |
| matches\_played | int |
| Wins | int |
| Draws | int |
| Losses | int |
| goals\_for | int |
| goals\_against | int |

(season\_id , team\_id) is the unique key for this table.

This table contains season id , team id, team name , matches played , wins , draws , losses , goals scored (goals\_for), and goals conceded (goals\_against ) for each team in each season.

Write a solution to calculate the points , goal difference , and rankfor each team in each season .The ranking should be determined as follows:

* Teams are first ranked by their total points (highest to lowest)
* If points are tied , teams are then ranked by their goal difference(highest to lowest)
* If goal difference is also tied , teams are then ranked alphabetically by team name

Points are calculated as follows :

* 3 points for a win
* 1 point for a draw
* 0 points for a loss

Goal difference is calculated as:goal\_for-goal\_against

Return the result table ordered by season\_id in ascending order, then by rank in ascending order , and finally by team name in ascending order

**Input:**

**SeasonStats Table**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **season\_id** | **team\_id** | **team\_name** | **matches\_played** | **wins** | **draws** | **losses** | **goals\_for** | **goals\_against** |
| 2021 | 1 | Manchester City | 38 | 29 | 6 | 3 | 99 | 26 |
| 2021 | 2 | Liverpool | 38 | 28 | 8 | 2 | 94 | 26 |
| 2021 | 3 | Chelsea | 38 | 21 | 11 | 6 | 76 | 33 |
| 2021 | 4 | Tottenham | 38 | 22 | 5 | 11 | 69 | 40 |
| 2021 | 5 | Arsenal | 38 | 22 | 3 | 13 | 61 | 48 |
| 2022 | 1 | Manchester City | 38 | 28 | 5 | 5 | 94 | 33 |
| 2022 | 2 | Arsenal | 38 | 26 | 6 | 6 | 88 | 43 |
| 2022 | 3 | Manchester United | 38 | 23 | 6 | 9 | 58 | 43 |
| 2022 | 4 | Newcastle | 38 | 19 | 14 | 5 | 68 | 33 |
| 2022 | 5 | Liverpool | 38 | 19 | 10 | 9 | 75 | 47 |

**Output:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **team\_id** | **team\_name** | **points** | **goal\_difference** | **position** |
| 2021 | 1 | Manchester City | 93 | 73 |
| 2021 | 2 | Liverpool | 92 | 68 |
| 2021 | 3 | Chelsea | 74 | 43 |
| 2021 | 4 | Tottenham | 71 | 29 |
| 2021 | 5 | Arsenal | 69 | 13 |
| 2022 | 1 | Manchester City | 89 | 61 |
| 2022 | 2 | Arsenal | 84 | 45 |
| 2022 | 3 | Manchester United | 75 | 15 |
| 2022 | 4 | Newcastle | 71 | 35 |

**WITH cte AS**

**SELECT SEASON\_ID , TEAM\_ID , TEAM\_NAME ,(WINS\*3) +**

**DRAWS \*1+ LOSSES \* 0) AS POINTS , (GOALS\_FOR -GOALS\_AGAINST) AS GOAL\_DIFFERENCE**

**FROM SEASONSTATS)**

**SELECT \* , ROW\_NUMBER() OVER(PARTITION BY SEASON\_ID ORDER\_BY POINTS DESC,GOAL\_DIFFERENCE DESC, TEAM\_NAME)**

**AS POSITION**

**FROM cte**

**ORDER BY SEASON\_ID , POSITION , TEAM\_NAME;**

**27** Calculate product final

**Table:products**

|  |  |
| --- | --- |
| Column Name | Type |
| product\_id | int |
| Category | varchar |
| Price | decimal |

product\_id is the unique key for this table .

Each row includes the products ID , its category ,and its price.

**Table :Discounts**

|  |  |
| --- | --- |
| Column Name | Type |
| Category | varchar |
| Discount | int |

category is the primary key for this table.

Each row contains a product category and the percentage discount applied to the category (values range from 0 to 100).

Write a solution to find the final price of each product after applying the category discount , its price remains unchanged

Return the result table ordered by product\_id in ascending order.

**Input:**

|  |  |  |
| --- | --- | --- |
| **product\_id** | **category** | **price** |
| 1 | Electronics | 1000 |
| 2 | Clothing | 50 |
| 3 | Electronics | 1200 |
| 4 | Home | 500 |

**Discounts Table**

|  |  |
| --- | --- |
| **category** | **discount** |
| Electronics | 10 |
| Clothing | 20 |

**Output:**

|  |  |  |
| --- | --- | --- |
| **product\_id** | **final\_price** | **category** |
| 1 | 900 | Electronics |
| 2 | 40 | Clothing |
| 3 | 1080 | Electronics |
| 4 | 500 | Home |

**SELECT P.PRODUCT\_ID, P.PRICE(1-IFNULL( D.DISCOUNT ,0)/100) AS FINAL\_PRICE**

**FROM PRODUCTS AS P**

**LEFT JOIN DISCOUNTS AS P**

**USING (CATEGORY)**

**ORDER BY P.PRODUCT\_ID;**

28.Market Analysis |||

|  |  |
| --- | --- |
| Column Name | Type |
| seller\_id | int |
| join\_date | date |
| favorite\_brand | varchar |

Seller\_id is column of unique values for this table.

This table contains seller id , join date , and favorite brand of sellers

Table:Items

|  |  |
| --- | --- |
| Column Name | Type |
| Item\_id | int |
| Item\_brand | varchar |

Item\_id is the column of unique values for this table

This table contains item id and item brand.

Table:Orders

|  |  |
| --- | --- |
| Column Name | Type |
| order\_id | int |
| order\_date | date |
| item\_id | int |
| seller\_id | int |

Order\_id is the column of unique values for this table.

Item\_id is a foreign key to the items table.

seller\_id is a foreign key to the users table .

This table contains order id ,order date , item id and seller id.

Write a solution to find the top seller who has sold the highest number of unique items with a different brand than their favorite brand.If there are multiple sellers with the same highest count , return all of them

Return the result table ordered by seller\_id in ascending order.

**Input:**

**Users Table**

|  |  |  |
| --- | --- | --- |
| **seller\_id** | **join\_date** | **favorite\_brand** |
| 1 | 2019-01-01 | Lenovo |
| 2 | 2019-02-09 | Samsung |
| 3 | 2019-01-19 | LG |

**Orders Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **order\_id** | **order\_date** | **item\_id** | **seller\_id** |
| 1 | 2019-08-01 | 4 | 2 |
| 2 | 2019-08-02 | 2 | 3 |
| 3 | 2019-08-03 | 3 | 3 |
| 4 | 2019-08-04 | 1 | 2 |
| 5 | 2019-08-04 | 4 | 2 |

**Items Table**

|  |  |
| --- | --- |
| **item\_id** | **item\_brand** |
| 1 | Samsung |
| 2 | Lenovo |
| 3 | LG |
| 4 | HP |

**Output:**

|  |  |
| --- | --- |
| **seller\_id** | **num\_items** |
| 2 | 1 |
| 3 | 1 |

**WITH cte AS**

**SELECT O.SELLER\_ID , COUNT(DISTINCT O.ITEM\_ID) AS NUM\_ITEMS**

**FROM ORDERS AS O**

**LEFT JOIN USERS AS U**

**USING (SELLER\_ID)**

**LEFT JOIN ITEMS AS I**

**USING (ITEM\_ID)**

**WHERE I.ITEM\_BRAND<>U.FAVORITE\_BRAND**

**GROUP BY O.SELLER\_ID),**

**cte2 AS**

**(SELECT \* , DENSE\_RANK() OVER(ORDER BY NUM\_ITEMS DESC) AS rnk**

**FROM cte)**

**SELECT SELLER\_ID , NUM\_ITEMS**

**FROM cte2**

**WHERE rnk=1;**

**ORDER BY SELLER\_ID ;**

**29 CALCULATE ORDERS WITHIN EACH INTERVAL**

|  |  |
| --- | --- |
| Column Name | Type |
| Minute | int |
| order\_count | int |

Minute is the primary key for this table.

Each row of this table contains the minute and number of orders received during that specific minute .The total number of rows will be a multiple of 6 .

Write a query to calculate total orders within each interval . Each interval is defined as a combination of 6 minutes.

Minutes 1 to 6 fall within interval 1 , while minutes 7 to 12 belong to interval 2 , and so forth .

Return the result table ordered by interval\_no in ascending order.

**Input:**

**Orders Table**

|  |  |
| --- | --- |
| **minute** | **order\_count** |
| 1 | 0 |
| 2 | 2 |
| 3 | 4 |
| 4 | 6 |
| 5 | 1 |
| 6 | 4 |
| 7 | 1 |
| 8 | 2 |
| 9 | 4 |
| 10 | 1 |
| 11 | 4 |
| 12 | 6 |

**Output:**

|  |  |
| --- | --- |
| **interval\_no** | **total\_orders** |
| 1 | 17 |
| 2 | 18 |

**WITH cte AS**

**SELECT \* , CEIL(MINUTE /6 ) AS INTERVAL\_NO**

**FROM ORDERS)**

**SELECT INTERVAL\_NO , SUM(ORDER\_COUNT) AS**

**TOTAL\_ORDERS**

**FROM cte**

**GROUP BY INTERVAL\_NO**

**ORDER BY INTERVAL\_NO;**

**30 ROLLING AVERAGE STEPS**

**Table:Steps**

|  |  |
| --- | --- |
| Column Name | Type |
| user\_id | int |
| steps\_count | int |
| steps\_date | date |

(user\_id , steps\_date) is the primary key for this table

Each row of this table contains user\_id , steps\_count , and steps\_date**.**

Write a solution to calculate 3-day rolling averages of steps for each user

We calculate the n-day rolling average this way

* For each day , we calculate the average of n consecutive days of step counts ending on that day if available , otherwise , n-day rolling average is not defined for it .

Output the user\_id , steps\_date , and rolling average , Round the rolling average to two decimal places

Return the result table ordered by user\_id , steps\_date in ascending order.

**Input:**

**Steps Table**

|  |  |  |
| --- | --- | --- |
| **user\_id** | **steps\_count** | **steps\_date** |
| 1 | 687 | 2021-09-02 |
| 1 | 395 | 2021-09-04 |
| 1 | 499 | 2021-09-05 |
| 1 | 712 | 2021-09-06 |
| 1 | 576 | 2021-09-07 |
| 2 | 153 | 2021-09-06 |
| 2 | 171 | 2021-09-07 |
| 2 | 530 | 2021-09-08 |
| 3 | 945 | 2021-09-04 |
| 3 | 120 | 2021-09-07 |
| 3 | 557 | 2021-09-08 |
| 3 | 840 | 2021-09-09 |
| 3 | 627 | 2021-09-10 |
| 5 | 382 | 2021-09-05 |
| 6 | 480 | 2021-09-01 |
| 6 | 191 | 2021-09-02 |
| 6 | 303 | 2021-09-05 |

**Output:**

|  |  |  |
| --- | --- | --- |
| **user\_id** | **steps\_date** | **rolling\_average** |
| 1 | 2021-09-06 | 535.33 |
| 1 | 2021-09-07 | 595.67 |
| 2 | 2021-09-08 | 284.67 |
| 3 | 2021-09-09 | 505.67 |
| 3 | 2021-09-10 | 674.67 |

**SELECT \* , AVG(STEPS\_COUNT) OVER (PARTITION BY USER\_ID**

**ORDER BY STEPS\_DATE ROWS BETWEEN 2 PRECEDING AND**

**CURRENT ROW ) AS ROLLING\_AVERAGE, LAG(STEPS\_DATE , 2) OVER (**

**PARTITION BY USER\_ID ORDER BY STEPS\_DATE ) AS TWO\_BEFORE**

**FROM STEPS)**

**SELECT USER\_ID , STEPS\_DATE , ROLLING\_AVERAGE**

**FROM cte**

**WHERE DATEDIFF(STEPS\_DATE, TW0\_BEFORE)=2**

**ORDER BY USER\_ID , STEPS\_DATE;**