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$')

ORDER BY USER_ID;

8. 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;

9. 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:

$$percentage_{maonth}=\frac{\#\ drivers\ that\ accepted\ at\ least\ over\ ride\ during\ the\ month}{\#\ available\ drivers\ during\ the\ month}*100.0$$

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_	chargeback_count	chargeback_

			amount		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),
2) AS RATING , SUM(T.DISTANCE) AS DISTANCE
3) 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;
```