Contents

[Nth action 1](#_Toc20886198)

[Consecutive Number 4](#_Toc20886199)

[Consecutive Days 5](#_Toc20886200)

[All the products 7](#_Toc20886201)

[Top N analysis 9](#_Toc20886202)

[Department top 3 salaries 11](#_Toc20886203)

[Find Cumulative Salary of an Employee 14](#_Toc20886204)

[3 consecutive days 16](#_Toc20886205)

[Trans SQL 18](#_Toc20886206)

[Pivot 19](#_Toc20886207)

[Self Join to get information 24](#_Toc20886208)

[Dense Rank Case 28](#_Toc20886209)

[Date \_sub 29](#_Toc20886210)

[Date\_sub 31](#_Toc20886211)

[Union all 33](#_Toc20886212)

[Comparison with sub query 36](#_Toc20886213)

[Group by on more than 1 column 40](#_Toc20886214)

[adding flag for common rows between two tables 43](#_Toc20886215)

[Where and having 43](#_Toc20886216)

[Tree 47](#_Toc20886217)

[Self Join Employee Manager 51](#_Toc20886218)

# Nth action

**User table**

**Order table**

Table: Users

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

| Column Name | Type |

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

| user\_id | int |

| join\_date | date |

| favorite\_brand | varchar |

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

user\_id is the primary key of this table.

This table has the info of the users of an online shopping website where users can sell and buy items.

Table: Orders

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

| Column Name | Type |

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

| order\_id | int |

| order\_date | date |

| item\_id | int |

| buyer\_id | int |

| seller\_id | int |

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

order\_id is the primary key of this table.

item\_id is a foreign key to the Items table.

buyer\_id and seller\_id are foreign keys to the Users table.

Table: Items

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

| Column Name | Type |

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

| item\_id | int |

| item\_brand | varchar |

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

item\_id is the primary key of this table.

Write an SQL query to find for each user, whether the brand of the second item (by date) they sold is their favorite brand. If a user sold less than two items, report the answer for that user as no.

It is guaranteed that no seller sold more than one item on a day.

The answer for the user with id 1 is no because they sold nothing.

The answer for the users with id 2 and 3 is yes because the brands of their second sold items are their favorite brands.

The answer for the user with id 4 is no because the brand of their second sold item is not their favorite brand.

|  |
| --- |
| **Mudit Solution**  select a.user\_id as seller\_id  , case when a.favorite\_brand=b.item\_brand then 'yes' else 'no' end as "2nd\_item\_fav\_brand"  from Users a  left join (  select seller\_id , item\_brand, rank() over (partition by seller\_id order by order\_date) as rnk    from Orders a  Left join  Items b  on  a.item\_id=b.item\_id  ) b on (a.user\_id=b.seller\_id and b.rnk=2)  order by seller\_id; |

Table: Sales

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

| Column Name | Type |

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

| sale\_id | int |

| product\_id | int |

| year | int |

| quantity | int |

| price | int |

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

sale\_id is the primary key of this table.

product\_id is a foreign key to Product table.

Note that the price is per unit.

Table: Product

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

| Column Name | Type |

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

| product\_id | int |

| product\_name | varchar |

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

product\_id is the primary key of this table.

Write an SQL query that selects the **product id**, **year**, **quantity**, and **price** for the **first year** of every product sold.

|  |
| --- |
| SELECT product\_id, year AS first\_year, quantity, price FROM (SELECT sale\_id, product\_id, year, quantity, price, RANK() OVER(PARTITION BY product\_id ORDER BY year ASC) AS rnk FROM sales) temp WHERE rnk = 1 |

# Consecutive Number

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

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

| Id | Num |

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

| 1 | 1 |

| 2 | 1 |

| 3 | 1 |

| 4 | 2 |

| 5 | 1 |

| 6 | 2 |

| 7 | 2 |

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

For example, given the above Logs table, 1 is the only number that appears consecutively for at least three times.

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

| ConsecutiveNums |

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

| 1 |

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

|  |
| --- |
| select distinct Num as ConsecutiveNums  from  (  select \*,  lag(Num,1,NULL) over(order by Id) as lag1,  lag(Num,2,NULL) over(order by Id) as lag2  from Logs  ) as t  where Num = lag1 and Num = lag2; |

# Consecutive Days

Table: Activity

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

| Column Name | Type |

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

| player\_id | int |

| device id | int |

| event\_date | date |

| games\_played | int |

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

(player\_id, event\_date) is the primary key of this table.

This table shows the activity of players of some game.

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

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

The query result format is in the following example:

Activity table:

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

| player\_id | device\_id | event\_date | games\_played |

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

| 1 | 2 | 2016-03-01 | 5 |

| 1 | 2 | 2016-03-02 | 6 |

| 2 | 3 | 2017-06-25 | 1 |

| 3 | 1 | 2016-03-02 | 0 |

| 3 | 4 | 2018-07-03 | 5 |

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

Result table:

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

| fraction |

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

| 0.33 |

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

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

|  |
| --- |
| select round(count(distinct a2.player\_id)/count(distinct a1.player\_id),2) as fraction  from Activity a1  left join Activity a2  on a1.player\_id = a2.player\_id  and  a1.event\_date = a2.event\_date - 1  where  (a1.player\_id, a1.event\_date)  in  (select player\_id, min(event\_date)  from Activity  group by 1) |

# All the products

Table: Customer

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

| Column Name | Type |

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

| customer\_id | int |

| product\_key | int |

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

product\_key is a foreign key to Product table.

Table: Product

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

| Column Name | Type |

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

| product\_key | int |

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

product\_key is the primary key column for this table.

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

For example:

Customer table:

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

| customer\_id | product\_key |

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

| 1 | 5 |

| 2 | 6 |

| 3 | 5 |

| 3 | 6 |

| 1 | 6 |

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

Product table:

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

| product\_key |

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

| 5 |

| 6 |

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

Result table:

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

| customer\_id |

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

| 1 |

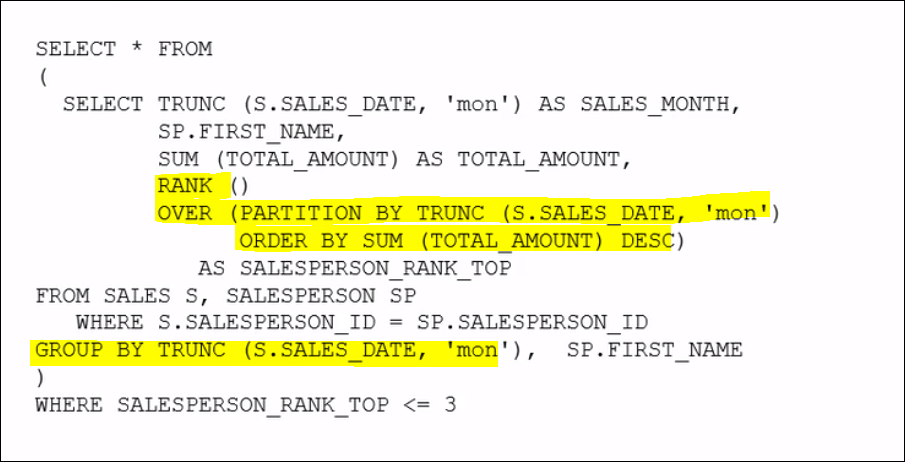
| 3 |

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

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

|  |
| --- |
| select customer\_id from Customer group by customer\_id having count(distinct product\_key) = (select count(1) from Product) order by customer\_id; |

# Top N analysis



**Find the best customer in every product**

|  |
| --- |
| **With temp as (**  Select  **product\_id**  , customer\_id  ,rank() over(partition by **product\_id** order by sum(total\_amount)) as rn  from sales  group by **product\_id**,customer\_id  **)**  with temp as (  )  Select \*  From temp  Where rn =1  ; |

# Department top 3 salaries

SQL Schema

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

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

| Id | Name | Salary | DepartmentId |

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

| 1 | Joe | 85000 | 1 |

| 2 | Henry | 80000 | 2 |

| 3 | Sam | 60000 | 2 |

| 4 | Max | 90000 | 1 |

| 5 | Janet | 69000 | 1 |

| 6 | Randy | 85000 | 1 |

| 7 | Will | 70000 | 1 |

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

The Department table holds all departments of the company.

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

| Id | Name |

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

| 1 | IT |

| 2 | Sales |

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

Write a SQL query to find employees who earn the top three salaries in each of the department. For the above tables, your SQL query should return the following rows (order of rows does not matter).

**Explanation:**

In IT department, Max earns the highest salary, both Randy and Joe earn the second highest salary, and Will earns the third highest salary. There are only two employees in the Sales department, Henry earns the highest salary while Sam earns the second highest salary.

|  |
| --- |
| /\* Write your PL/SQL query statement below \*/  with toppers as (  select  e.Id,  e.Name as Employee,  e.Salary,  e.DepartmentId,  d.Name as Department\_Name,  dense\_rank() over(partition by DepartmentId order by Salary DESC) as rank  from Employee e  join Department d on d.Id = e.DepartmentId  )  select  Department\_Name as Department,  Employee,  Salary  from toppers  where rank in (1,2,3)  ; |

Table: Project

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

| Column Name | Type |

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

| project\_id | int |

| employee\_id | int |

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

(project\_id, employee\_id) is the primary key of this table.

employee\_id is a foreign key to Employee table.

Table: Employee

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

| Column Name | Type |

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

| employee\_id | int |

| name | varchar |

| experience\_years | int |

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

employee\_id is the primary key of this table.

Write an SQL query that reports the **most experienced** employees in each project. In case of a tie, report all employees with the maximum number of experience years.

|  |
| --- |
| with employee\_experience as (  select p.project\_id, p.employee\_id,  rank() over(partition by p.project\_id order by experience\_years desc) as rank  from Project p join Employee e  on p.employee\_id = e.employee\_id)  select project\_id, employee\_id  from employee\_experience  where rank = 1; |

# Find Cumulative Salary of an Employee

SQL Schema

The **Employee** table holds the salary information in a year.

Write a SQL to get the cumulative sum of an employee's salary over a period of 3 months but exclude the most recent month.

The result should be displayed by 'Id' ascending, and then by 'Month' descending.

**Example**  
**Input**

| Id | Month | Salary |

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

| 1 | 1 | 20 |

| 2 | 1 | 20 |

| 1 | 2 | 30 |

| 2 | 2 | 30 |

| 3 | 2 | 40 |

| 1 | 3 | 40 |

| 3 | 3 | 60 |

| 1 | 4 | 60 |

| 3 | 4 | 70 |

**Output**

| Id | Month | Salary |

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

| 1 | 3 | 90 |

| 1 | 2 | 50 |

| 1 | 1 | 20 |

| 2 | 1 | 20 |

| 3 | 3 | 100 |

| 3 | 2 | 40 |

**Explanation**

Employee '1' has 3 salary records for the following 3 months except the most recent month '4': salary 40 for month '3', 30 for month '2' and 20 for month '1'  
So the cumulative sum of salary of this employee over 3 months is 90(40+30+20), 50(30+20) and 20 respectively.

| Id | Month | Salary |

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

| 1 | 3 | 90 |

| 1 | 2 | 50 |

| 1 | 1 | 20 |

Employee '2' only has one salary record (month '1') except its most recent month '2'.

| Id | Month | Salary |

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

| 2 | 1 | 20 |

Employ '3' has two salary records except its most recent pay month '4': month '3' with 60 and month '2' with 40. So the cumulative salary is as following.

| Id | Month | Salary |

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

| 3 | 3 | 100 |

| 3 | 2 | 40 |

|  |
| --- |
| SELECT Id, Month,  Salary + ISNULL(Lag\_1\_salary,0) + ISNULL(Lag\_2\_salary,0) AS Salary  FROM (  SELECT \*, MAX(Month) OVER(PARTITION BY Id) AS most\_recent,  LAG(Salary,1) OVER(PARTITION BY ID ORDER By Month ASC) AS Lag\_1\_salary, LAG(Salary,2) OVER(PARTITION BY ID ORDER By Month ASC) AS Lag\_2\_salary  FROM Employee) AS temp  WHERE Month != most\_recent  ORDER BY Id, Month DESC; |

# 3 consecutive days

X city built a new stadium, each day many people visit it and the stats are saved as these columns: **id**, **visit\_date**, **people**

Please write a query to display the records which have 3 or more consecutive rows and the amount of people more than 100(inclusive).

For example, the table stadium:

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

| id | visit\_date | people |

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

| 1 | 2017-01-01 | 10 |

| 2 | 2017-01-02 | 109 |

| 3 | 2017-01-03 | 150 |

| 4 | 2017-01-04 | 99 |

| 5 | 2017-01-05 | 145 |

| 6 | 2017-01-06 | 1455 |

| 7 | 2017-01-07 | 199 |

| 8 | 2017-01-08 | 188 |

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

For the sample data above, the output is:

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

| id | visit\_date | people |

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

| 5 | 2017-01-05 | 145 |

| 6 | 2017-01-06 | 1455 |

| 7 | 2017-01-07 | 199 |

| 8 | 2017-01-08 | 188 |

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

|  |
| --- |
| with laggers\_n\_leaders as (  select  id,  visit\_date,  people,  lag(people, 1) over(order by id) as lag\_1,  lag(people, 2) over(order by id) as lag\_2,  lead(people,1) over(order by id) as next\_1,  lead(people,2) over(order by id) as next\_2  from stadium  )  select  id,  visit\_date,  people  from laggers\_n\_leaders  where -- where it's the \*end\* of the 3 consecutive  (people >= 100  and lag\_1 >= 100  and lag\_2 >= 100)  or -- where it's the \*beginning\* of the 3 consecutive  (people >= 100  and next\_1 >= 100  and next\_2 >= 100)  or -- where it's the \*middle\* of the 3 consecutive  (people >= 100  and lag\_1 >= 100  and next\_1 >= 100  ); |

# Trans SQL

SQL Schema

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

| name | continent |

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

| Jack | America |

| Pascal | Europe |

| Xi | Asia |

| Jane | America |

[Pivot](https://en.wikipedia.org/wiki/Pivot_table) the continent column in this table so that each name is sorted alphabetically and displayed underneath its corresponding continent. The output headers should be America, Asia and Europe respectively. It is guaranteed that the student number from America is no less than either Asia or Europe.

For the sample input, the output is:

| America | Asia | Europe |

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

| Jack | Xi | Pascal |

| Jane | | |

|  |
| --- |
| select min(America) as america ,min(Asia) as asia,min(Europe) as europe  from(  select  iif(Continent='America',Name,NULL)AS America,  iif(Continent='Asia',Name,NULL)AS Asia,  iif(Continent='Europe',Name,NULL) AS Europe,  row\_number() over(partition by continent order by name) As rn  from student  ) t  group by rn |

# Pivot

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 an SQL query 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.

The query result format is in the following example:

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 |

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

Result table:

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

| 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 |

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

On 2019-07-01, user 1 purchased using **both** desktop and mobile, user 2 purchased using mobile **only** and user 3 purchased using desktop **only**.

On 2019-07-02, user 2 purchased using mobile **only**, user 3 purchased using desktop **only** and no one purchased using **both** platforms.

Answer

Let's start with a simple preprocess:

SELECT

spend\_date,

user\_id,

SUM(CASE platform WHEN 'mobile' THEN amount ELSE 0 END) mobile\_amount,

SUM(CASE platform WHEN 'desktop' THEN amount ELSE 0 END) desktop\_amount

FROM Spending

GROUP BY spend\_date, user\_id

For each user in each day, we fetch its mobile\_amount and desktop\_amount respectively and output them into a single row. In this form, we can see a user belongs to which platform very clearly:

| **spend\_date** | **user\_id** | **mobile\_amount** | **desktop\_amount** | **->*(platform)*** |
| --- | --- | --- | --- | --- |
| 2019-07-01 | 1 | 100 | 100 | -> *(both)* |
| 2019-07-01 | 2 | 100 | 0 | -> *(mobile)* |
| 2019-07-01 | 3 | 0 | 100 | -> *(desktop)* |
| 2019-07-02 | 2 | 100 | 0 | -> *(mobile)* |
| 2019-07-02 | 3 | 0 | 100 | ->*(desktop)* |

Based on the above table, we use the following SQL to bind users to their platforms and calculate the amounts spent:

SELECT

spend\_date,

user\_id,

IF(mobile\_amount > 0, IF(desktop\_amount > 0, 'both', 'mobile'), 'desktop') platform,

(mobile\_amount + desktop\_amount) amount

FROM (

...

) o

Result table:

| **spend\_date** | **user\_id** | **platform** | **amount** |
| --- | --- | --- | --- |
| 2019-07-01 | 1 | both | 200 |
| 2019-07-01 | 2 | mobile | 100 |
| 2019-07-01 | 3 | desktop | 100 |
| 2019-07-02 | 2 | mobile | 100 |
| 2019-07-02 | 3 | desktop | 100 |

We don't wanna miss any record which has ZERO total\_amount and total\_users. So we need to get all combinations of spend\_date and platform:

SELECT DISTINCT(spend\_date), 'desktop' platform FROM Spending

UNION

SELECT DISTINCT(spend\_date), 'mobile' platform FROM Spending

UNION

SELECT DISTINCT(spend\_date), 'both' platform FROM Spending

The output:

| **spend\_date** | **platform** |
| --- | --- |
| 2019-07-01 | desktop |
| 2019-07-01 | mobile |
| 2019-07-01 | both |
| 2019-07-02 | desktop |
| 2019-07-02 | mobile |
| 2019-07-02 | both |

After joinning this table to the previous one, we have our **final answer**:

SELECT

p.spend\_date,

p.platform,

IFNULL(SUM(amount), 0) total\_amount,

COUNT(user\_id) total\_users

FROM

(

SELECT DISTINCT(spend\_date), 'desktop' platform FROM Spending

UNION

SELECT DISTINCT(spend\_date), 'mobile' platform FROM Spending

UNION

SELECT DISTINCT(spend\_date), 'both' platform FROM Spending

) p

LEFT JOIN (

SELECT

spend\_date,

user\_id,

IF(mobile\_amount > 0, IF(desktop\_amount > 0, 'both', 'mobile'), 'desktop') platform,

(mobile\_amount + desktop\_amount) amount

FROM (

SELECT

spend\_date,

user\_id,

SUM(CASE platform WHEN 'mobile' THEN amount ELSE 0 END) mobile\_amount,

SUM(CASE platform WHEN 'desktop' THEN amount ELSE 0 END) desktop\_amount

FROM Spending

GROUP BY spend\_date, user\_id

) o

) t

ON p.platform=t.platform AND p.spend\_date=t.spend\_date

GROUP BY spend\_date, platform

# Self Join to get information

SQL Schema

In facebook, there is a follow table with two columns: **followee**, **follower**.

Please write a sql query to get the amount of each follower’s follower if he/she has one.

For example:

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

| followee | follower |

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

| A | B |

| B | C |

| B | D |

| D | E |

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

should output:

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

| follower | num |

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

| B | 2 |

| D | 1 |

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

**Explaination:**  
Both B and D exist in the follower list, when as a followee, B's follower is C and D, and D's follower is E. A does not exist in follower list.

**Method 1 (left Join)**

SELECT follower, COUNT(DISTINCT follower\_fow) num

FROM

( SELECT f1.follower, f2.follower follower\_fow

FROM

follow f1 LEFT JOIN follow f2

ON f1.follower = f2.followee) AS new\_follow

GROUP BY 1

HAVING num != 0

**Method 2 (Cartesian Product)**

/\* Write your PL/SQL query statement below \*/

SELECT t1.follower, COUNT(DISTINCT t2.follower) AS num

FROM follow t1, follow t2

WHERE t1.follower = t2.followee

GROUP BY t1.follower

Get the highest answer rate question from a table survey\_log with these columns: **uid**, **action**, **question\_id**, **answer\_id**, **q\_num**, **timestamp**.

uid means user id; action has these kind of values: "show", "answer", "skip"; answer\_id is not null when action column is "answer", while is null for "show" and "skip"; q\_num is the numeral order of the question in current session.

Write a sql query to identify the question which has the highest answer rate.

**Example:**

**Input:**

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

| uid | action | question\_id | answer\_id | q\_num | timestamp |

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

| 5 | show | 285 | null | 1 | 123 |

| 5 | answer | 285 | 124124 | 1 | 124 |

| 5 | show | 369 | null | 2 | 125 |

| 5 | skip | 369 | null | 2 | 126 |

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

**Output:**

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

| survey\_log |

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

| 285 |

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

**Explanation:**

question 285 has answer rate 1/1, while question 369 has 0/1 answer rate, so output 285.

**Note:** The highest answer rate meaning is: answer number's ratio in show number in the same question.

|  |
| --- |
| # Write your MySQL query statement below  select question\_id as survey\_log  from  (  select question\_id,  sum(case when answer\_id is NOT NULL then 1 else 0 end)/count(distinct question\_id) as answer\_rate  from survey\_log  group by question\_id  order by answer\_rate desc) a  limit 1 |

Method 2

SELECT question\_id as survey\_log

FROM

(

SELECT question\_id,

SUM(case when action="answer" THEN 1 ELSE 0 END) as num\_answer,

SUM(case when action="show" THEN 1 ELSE 0 END) as num\_show,

FROM survey\_log

GROUP BY question\_id

) as tbl

ORDER BY (num\_answer / num\_show) DESC

LIMIT 1

Method 3

SELECT

question\_id AS 'survey\_log'

FROM

survey\_log

GROUP BY question\_id

ORDER BY COUNT(answer\_id) / COUNT(IF(action = 'show', 1, 0)) DESC

LIMIT 1;

# Dense Rank Case

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

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

| Id | Score |

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

| 1 | 3.50 |

| 2 | 3.65 |

| 3 | 4.00 |

| 4 | 3.85 |

| 5 | 4.00 |

| 6 | 3.65 |

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

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

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

| Score | Rank |

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

| 4.00 | 1 |

| 4.00 | 1 |

| 3.85 | 2 |

| 3.65 | 3 |

| 3.65 | 3 |

| 3.50 | 4 |

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

|  |
| --- |
| with cte as  (  select  Score  , dense\_rank() over (order by score desc) as Rank  from scores  )  select \* from cte |

# Date \_sub

able: Books

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

| Column Name | Type |

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

| book\_id | int |

| name | varchar |

| available\_from | date |

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

book\_id is the primary key of this table.

Table: Orders

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

| Column Name | Type |

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

| order\_id | int |

| book\_id | int |

| quantity | int |

| dispatch\_date | date |

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

order\_id is the primary key of this table.

book\_id is a foreign key to the Books table.

Write an SQL query that reports the **books** that have sold **less than 10** copies in the last year, excluding books that have been available for less than 1 month from today. **Assume today is 2019-06-23**.

The query result format is in the following example:

Books table:

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

| book\_id | name | available\_from |

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

| 1 | "Kalila And Demna" | 2010-01-01 |

| 2 | "28 Letters" | 2012-05-12 |

| 3 | "The Hobbit" | 2019-06-10 |

| 4 | "13 Reasons Why" | 2019-06-01 |

| 5 | "The Hunger Games" | 2008-09-21 |

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

Orders table:

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

| order\_id | book\_id | quantity | dispatch\_date |

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

| 1 | 1 | 2 | 2018-07-26 |

| 2 | 1 | 1 | 2018-11-05 |

| 3 | 3 | 8 | 2019-06-11 |

| 4 | 4 | 6 | 2019-06-05 |

| 5 | 4 | 5 | 2019-06-20 |

| 6 | 5 | 9 | 2009-02-02 |

| 7 | 5 | 8 | 2010-04-13 |

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

Result table:

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

| book\_id | name |

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

| 1 | "Kalila And Demna" |

| 2 | "28 Letters" |

| 5 | "The Hunger Games" |

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

|  |
| --- |
| # Write your MySQL query statement below  SELECT book\_id, name  FROM  (SELECT b.book\_id as book\_id, b.name as name, SUM(CASE WHEN o.dispatch\_date >= DATE\_SUB('2019-06-23', INTERVAL 1 YEAR) THEN o.quantity ELSE 0 END) as q\_sum  FROM Books b LEFT JOIN Orders o ON b.book\_id = o.book\_id  WHERE b.available\_from < DATE\_SUB('2019-06-23', INTERVAL 1 MONTH)  GROUP BY b.book\_id  HAVING q\_sum < 10) a; |

# Date\_sub

Table: Traffic

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

| Column Name | Type |

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

| user\_id | int |

| activity | enum |

| activity\_date | date |

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

There is no primary key for this table, it may have duplicate rows.

The activity column is an ENUM type of ('login', 'logout', 'jobs', 'groups', 'homepage').

Write an SQL query that reports for every date within at most **90 days** from today, the number of users that logged in for the first time on that date. Assume today is **2019-06-30**.

The query result format is in the following example:

Traffic table:

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

| user\_id | activity | activity\_date |

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

| 1 | login | 2019-05-01 |

| 1 | homepage | 2019-05-01 |

| 1 | logout | 2019-05-01 |

| 2 | login | 2019-06-21 |

| 2 | logout | 2019-06-21 |

| 3 | login | 2019-01-01 |

| 3 | jobs | 2019-01-01 |

| 3 | logout | 2019-01-01 |

| 4 | login | 2019-06-21 |

| 4 | groups | 2019-06-21 |

| 4 | logout | 2019-06-21 |

| 5 | login | 2019-03-01 |

| 5 | logout | 2019-03-01 |

| 5 | login | 2019-06-21 |

| 5 | logout | 2019-06-21 |

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

Result table:

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

| login\_date | user\_count |

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

| 2019-05-01 | 1 |

| 2019-06-21 | 2 |

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

Note that we only care about dates with non zero user count.

The user with id 5 first logged in on 2019-03-01 so he's not counted on 2019-06-21.

|  |
| --- |
| With temp as (    select user\_id, min(activity\_date) as first\_login from Traffic  where activity = 'login'  group by user\_id    )    Select first\_login as login\_date, count(\*) as user\_count  from  temp where  first\_login between date\_sub('2019-06-30', interval 90 day) and '2019-06-30'  group by first\_login |

# Union all

In social network like Facebook or Twitter, people send friend requests and accept others' requests as well.

Table request\_accepted holds the data of friend acceptance, while **requester\_id** and **accepter\_id** both are the id of a person.

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

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

| 1 | 2 | 2016\_06-03 |

| 1 | 3 | 2016-06-08 |

| 2 | 3 | 2016-06-08 |

| 3 | 4 | 2016-06-09 |

Write a query to find the the people who has most friends and the most friends number. For the sample data above, the result is:

| id | num |

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

| 3 | 3 |

**Note:**

* It is guaranteed there is only 1 people having the most friends.
* The friend request could only been accepted once, which mean there is no multiple records with the same **requester\_id** and **accepter\_id** value.

**Explanation:**  
The person with id '3' is a friend of people '1', '2' and '4', so he has 3 friends in total, which is the most number than any others.

**Follow-up:**  
In the real world, multiple people could have the same most number of friends, can you find all these people in this case?

|  |
| --- |
| SELECT ids as id , COUNT(\*) as num  FROM  (SELECT requester\_id as ids FROM request\_accepted  UNION ALL  SELECT accepter\_id as ids FROM request\_accepted) AS u  GROUP BY ids  ORDER BY COUNT(\*) DESC  LIMIT 1 |

Table: Views

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

| Column Name | Type |

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

| article\_id | int |

| author\_id | int |

| viewer\_id | int |

| view\_date | date |

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

There is no primary key for this table, it may have duplicate rows.

Each row of this table indicates that some viewer viewed an article (written by some author) on some date.

Note that equal author\_id and viewer\_id indicate the same person.

Write an SQL query to find all the people who viewed more than one article on the same date, sorted in ascending order by their id.

|  |
| --- |
| select distinct Viewer\_id as id  from Views  group by View\_date, Viewer\_id  having count(distinct article\_id) > 1  order by 1 |

# Comparison with sub query

Write a query to print the sum of all total investment values in 2016 (**TIV\_2016**), to a scale of 2 decimal places, for all policy holders who meet the following criteria:

1. Have the same **TIV\_2015** value as one or more other policyholders.
2. Are not located in the same city as any other policyholder (i.e.: the (latitude, longitude) attribute pairs must be unique).

**Input Format:**  
The ***insurance*** table is described as follows:

| Column Name | Type |

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

| PID | INTEGER(11) |

| TIV\_2015 | NUMERIC(15,2) |

| TIV\_2016 | NUMERIC(15,2) |

| LAT | NUMERIC(5,2) |

| LON | NUMERIC(5,2) |

where **PID** is the policyholder's policy ID, **TIV\_2015** is the total investment value in 2015, **TIV\_2016** is the total investment value in 2016, **LAT** is the latitude of the policy holder's city, and **LON** is the longitude of the policy holder's city.

|  |
| --- |
| select sum(TIV\_2016) as TIV\_2016 from insurance i where i.TIV\_2015 in (select i1.TIV\_2015 from insurance i1 where i.PID!=i1.PID) and concat(i.LAT,i.LON)  not in  (select concat(i2.LAT,i2.LON) from insurance i2 where i.PID!=i2.PID) |

Three c in life

Vhoice vhances change

You kmust make a choice to make a change oy you

Table: Transactions

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

| Column Name | Type |

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

| id | int |

| country | varchar |

| state | enum |

| amount | int |

| trans\_date | date |

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

id is the primary key of this table.

The table has information about incoming transactions.

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

Table: Chargebacks

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

| Column Name | Type |

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

| trans\_id | int |

| charge\_date | date |

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

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

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

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

Write an SQL query 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 query, given the month and country, ignore rows with all zeros.

The query result format is in the following example:

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 | approved | 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 |

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

Result table:

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

| month | country | approved\_count | approved\_amount | chargeback\_count | chargeback\_amount |

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

| 2019-05 | US | 1 | 1000 | 1 | 2000 |

| 2019-06 | US | 3 | 12000 | 1 | 1000 |

| 2019-09 | US | 0 | 0 | 1 | 5000 |

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

|  |
| --- |
| select month, country,  sum(case when type='approved' then 1 else 0 end) as approved\_count,  sum(case when type='approved' then amount else 0 end) as approved\_amount,  sum(case when type='chargeback' then 1 else 0 end) as chargeback\_count,  sum(case when type='chargeback' then amount else 0 end) as chargeback\_amount  from (  (  select left(t.trans\_date, 7) as month, t.country, amount,'approved' as type  from Transactions as t  where state='approved'  )  union all (  select left(c.trans\_date, 7) as month, t.country, amount,'chargeback' as type  from Transactions as t join Chargebacks as c  on t.id = c.trans\_id  )  ) as tt  group by tt.month, tt.country |

# Group by on more than 1 column

Table: Users

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

| Column Name | Type |

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

| user\_id | int |

| join\_date | date |

| favorite\_brand | varchar |

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

user\_id is the primary key of this table.

This table has the info of the users of an online shopping website where users can sell and buy items.

Table: Orders

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

| Column Name | Type |

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

| order\_id | int |

| order\_date | date |

| item\_id | int |

| buyer\_id | int |

| seller\_id | int |

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

order\_id is the primary key of this table.

item\_id is a foreign key to the Items table.

buyer\_id and seller\_id are foreign keys to the Users table.

Table: Items

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

| Column Name | Type |

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

| item\_id | int |

| item\_brand | varchar |

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

item\_id is the primary key of this table.

Write an SQL query to find for each user, the join date and the number of orders they made as a buyer in **2019**.

The query result format is in the following example:

Users table:

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

| user\_id | join\_date | favorite\_brand |

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

| 1 | 2018-01-01 | Lenovo |

| 2 | 2018-02-09 | Samsung |

| 3 | 2018-01-19 | LG |

| 4 | 2018-05-21 | HP |

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

Orders table:

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

| order\_id | order\_date | item\_id | buyer\_id | seller\_id |

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

| 1 | 2019-08-01 | 4 | 1 | 2 |

| 2 | 2018-08-02 | 2 | 1 | 3 |

| 3 | 2019-08-03 | 3 | 2 | 3 |

| 4 | 2018-08-04 | 1 | 4 | 2 |

| 5 | 2018-08-04 | 1 | 3 | 4 |

| 6 | 2019-08-05 | 2 | 2 | 4 |

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

Items table:

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

| item\_id | item\_brand |

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

| 1 | Samsung |

| 2 | Lenovo |

| 3 | LG |

| 4 | HP |

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

Result table:

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

| buyer\_id | join\_date | orders\_in\_2019 |

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

| 1 | 2018-01-01 | 1 |

| 2 | 2018-02-09 | 2 |

| 3 | 2018-01-19 | 0 |

| 4 | 2018-05-21 | 0 |

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

SELECT u.user\_id as buyer\_id, u.join\_date, COUNT(o.order\_id) as orders\_in\_2019  
FROM Users u  
LEFT JOIN Orders o  
ON u.user\_id = o.buyer\_id

AND

YEAR(O.order\_date) = 2019  
GROUP BY u.user\_id, u.join\_date  
ORDER BY u.user\_id;

# [adding flag for common rows between two tables](https://stackoverflow.com/questions/11502753/adding-flag-for-common-rows-between-two-tables)

select a.\*

, case when b.column1 is not null then 'YES' else 'NO' end as flag

from a left outer join b

on a.column1 = b.column1

# Where and having

SQL Schema

Table: Events

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

| Column Name | Type |

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

| business\_id | int |

| event\_type | varchar |

| occurences | int |

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

(business\_id, event\_type) is the primary key of this table.

Each row in the table logs the info that an event of some type occured at some business for a number of times.

Write an SQL query to find all *active businesses*.

An active business is a business that has more than one event type with occurences greater than the average occurences of that event type among all businesses.

The query result format is in the following example:

Events table:

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

| business\_id | event\_type | occurences |

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

| 1 | reviews | 7 |

| 3 | reviews | 3 |

| 1 | ads | 11 |

| 2 | ads | 7 |

| 3 | ads | 6 |

| 1 | page views | 3 |

| 2 | page views | 12 |

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

Result table:

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

| business\_id |

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

| 1 |

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

Average for 'reviews', 'ads' and 'page views' are (7+3)/2=5, (11+7+6)/3=8, (3+12)/2=7.5 respectively.

Business with id 1 has 7 'reviews' events (more than 5) and 11 'ads' events (more than 8) so it is an active business.

|  |
| --- |
| # Write your MySQL query statement below  select business\_id from  (select e.business\_id, e.event\_type, e.occurences, temp.avg\_occurences  from events e  join  (select event\_type, avg(occurences) as avg\_occurences  from events  group by event\_type) as temp  on e.event\_type = temp.event\_type) as temp2  where occurences>avg\_occurences  group by business\_id  having  count(event\_type)>=2 |

Table: Queue

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

| Column Name | Type    |

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

| person\_id   | int     |

| person\_name | varchar |

| weight      | int     |

| turn        | int     |

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

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

This table has the information about all people waiting for an elevator.

The person\_id and turn columns will contain all numbers from 1 to n, where n is the number of rows in the table.

The maximum weight the elevator can hold is **1000**.

Write an SQL query to find the person\_name of the last person who will fit in the elevator without exceeding the weight limit. It is guaranteed that the person who is first in the queue can fit in the elevator.

The query result format is in the following example:

Queue table

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

| person\_id | person\_name       | weight | turn |

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

| 5         | George Washington | 250    | 1    |

| 3         | John Adams        | 350    | 2    |

| 6         | Thomas Jefferson  | 400    | 3    |

| 2         | Will Johnliams    | 200    | 4    |

| 4         | Thomas Jefferson  | 175    | 5    |

| 1         | James Elephant    | 500    | 6    |

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

Result table

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

| person\_name       |

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

| Thomas Jefferson  |

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

Queue table is ordered by turn in the example for simplicity.

In the example George Washington(id 5), John Adams(id 3) and Thomas Jefferson(id 6) will enter the elevator as their weight sum is 250 + 350 + 400 = 1000.

Thomas Jefferson(id 6) is the last person to fit in the elevator because he has the last turn in these three people.

|  |
| --- |
| SELECT TOP 1 person\_name  FROM  (SELECT person\_name, weight,      SUM(weight) OVER(ORDER BY turn) as total\_weight      FROM queue) temp  WHERE total\_weight <= 1000  ORDER BY total\_weight DESC; |

# Tree

Given a table tree, **id** is identifier of the tree node and **p\_id** is its parent node's **id**.

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

| id | p\_id |

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

| 1  | null |

| 2  | 1    |

| 3  | 1    |

| 4  | 2    |

| 5  | 2    |

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

Each node in the tree can be one of three types:

* Leaf: if the node is a leaf node.
* Root: if the node is the root of the tree.
* Inner: If the node is neither a leaf node nor a root node.

Write a query to print the node id and the type of the node. Sort your output by the node id. The result for the above sample is:

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

| id | Type |

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

| 1  | Root |

| 2  | Inner|

| 3  | Leaf |

| 4  | Leaf |

| 5  | Leaf |

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

**Explanation**

 Node '1' is root node, because its parent node is NULL and it has child node '2' and '3'.

Node '2' is inner node, because it has parent node '1' and child node '4' and '5'.

Node '3', '4' and '5' is Leaf node, because they have parent node and they don't have child node.

And here is the image of the sample tree as below:

                     1

                   /   \

                      2       3

                    /   \

                  4       5

**Note**

If there is only one node on the tree, you only need to output its root attributes.

|  |
| --- |
| # Write your MySQL query statement below  SELECT id,  CASE WHEN p\_id is null THEN 'Root'       WHEN id in (select distinct p\_id from tree) THEN 'Inner'       ELSE 'Leaf'  END AS Type  FROM tree; |

Table: Products

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

| Column Name   | Type    |

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

| product\_id    | int     |

| new\_price     | int     |

| change\_date   | date    |

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

(product\_id, change\_date) is the primary key of this table.

Each row of this table indicates that the price of some product was changed to a new price at some date.

Write an SQL query to find the prices of all products on **2019-08-16**. Assume the price of all products before any change is **10**.

The query result format is in the following example:

Products table:

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

| product\_id | new\_price | change\_date |

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

| 1          | 20        | 2019-08-14  |

| 2          | 50        | 2019-08-14  |

| 1          | 30        | 2019-08-15  |

| 1          | 35        | 2019-08-16  |

| 2          | 65        | 2019-08-17  |

| 3          | 20        | 2019-08-18  |

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

Result table:

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

| product\_id | price |

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

| 2          | 50    |

| 1          | 35    |

| 3          | 10    |

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

|  |
| --- |
| select distinct a.product\_id, coalesce(b.new\_price, 10) as price from Products as a left join (select product\_id, rank() over(partition by product\_id order by change\_date DESC) as xrank, new\_price from Products where change\_date<='2019-08-16') as b on a.product\_id=b.product\_id and b.xrank=1 order by 2 DESC; ''' |

# Self Join Employee Manager

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

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

|Id    |Name       |Department |ManagerId |

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

|101   |John       |A          |null      |

|102   |Dan        |A          |101       |

|103   |James      |A          |101       |

|104   |Amy        |A          |101       |

|105   |Anne       |A          |101       |

|106   |Ron        |B          |101       |

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

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

+-------+

| Name  |

+-------+

| John  |

+-------+

**Note:**  
No one would report to himself.

|  |
| --- |
| SELECT e2.Name FROM Employee e1, Employee e2  where e1.ManagerId=e2.Id  GROUP BY e1.ManagerId  HAVING Count(Distinct e1.Id)>=5 |

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

The column **id** is continuous increment.

Mary wants to change seats for the adjacent students.

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

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

|    id   | student |

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

|    1    | Abbot   |

|    2    | Doris   |

|    3    | Emerson |

|    4    | Green   |

|    5    | Jeames  |

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

For the sample input, the output is:

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

|    id   | student |

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

|    1    | Doris   |

|    2    | Abbot   |

|    3    | Green   |

|    4    | Emerson |

|    5    | Jeames  |

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

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

|  |
| --- |
| Select case when id%2=0 then id-1 when id%2=1 and id<(select top 1 id from seat order by id desc) then id+1 else id end as id , student from seat order by id asc |

|  |
| --- |
| select [t.id](http://t.id/), case when [t.id](http://t.id/) % 2 = 1 and t.next is not null then t.next when [t.id](http://t.id/) % 2 = 1 and (t.next is null or t.previous is null) then t.student else t.previous end as 'student' from ( select id, student, lead(student,1) over (order by id) as 'next', lag(student,1) over (order by id) as 'previous' from seat ) t |

Table: Transactions

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

| Column Name   | Type    |

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

| id            | int     |

| country       | varchar |

| state         | enum    |

| amount        | int     |

| trans\_date    | date    |

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

id is the primary key of this table.

The table has information about incoming transactions.

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

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

The query result format is in the following example:

Transactions table:

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

| id   | country | state    | amount | trans\_date |

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

| 121  | US      | approved | 1000   | 2018-12-18 |

| 122  | US      | declined | 2000   | 2018-12-19 |

| 123  | US      | approved | 2000   | 2019-01-01 |

| 124  | DE      | approved | 2000   | 2019-01-07 |

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

Result table:

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

| month    | country | trans\_count | approved\_count | trans\_total\_amount | approved\_total\_amount |

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

| 2018-12  | US      | 2           | 1              | 3000               | 1000                  |

| 2019-01  | US      | 1           | 1              | 2000               | 2000                  |

| 2019-01  | DE      | 1           | 1              | 2000               | 2000                  |

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

|  |
| --- |
| SELECT      DATE\_FORMAT(trans\_date, '%Y-%m') AS month,      country,      COUNT(id) AS trans\_count,      COUNT(IF(state = 'approved', 1, NULL)) AS approved\_count,      SUM(amount) AS trans\_total\_amount,      SUM(IF(state='approved', amount, 0)) AS approved\_total\_amount  FROM Transactions  GROUP BY 1, 2 |