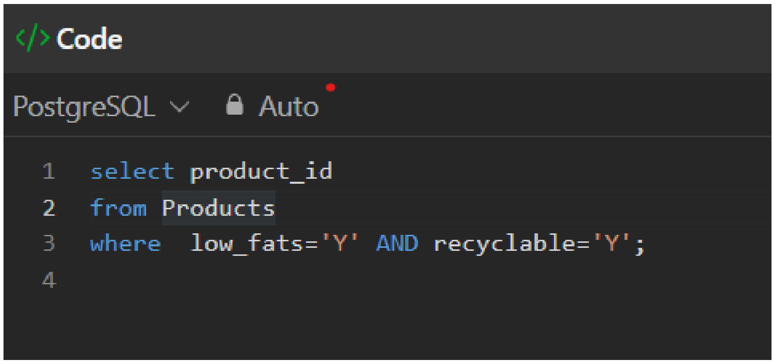
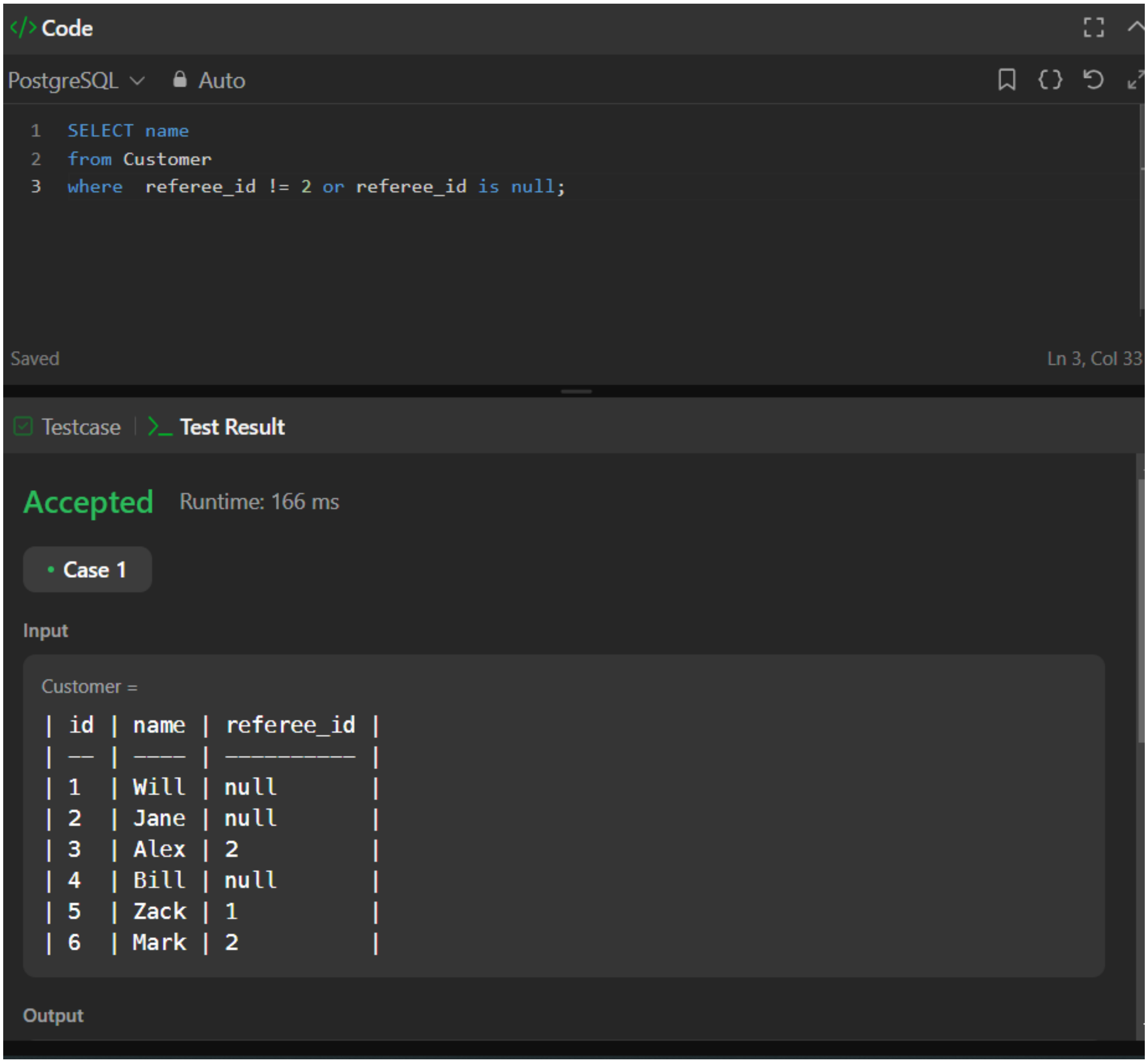
Recyclable and Low Fat Products



```
select product_id
from Products
where  low_fats='Y' AND recyclable='Y';
```

584. Find Customer Referee



```
SELECT name
from Customer
where  referee_id != 2 or referee_id is null;
```

Accepted Runtime: 166 ms

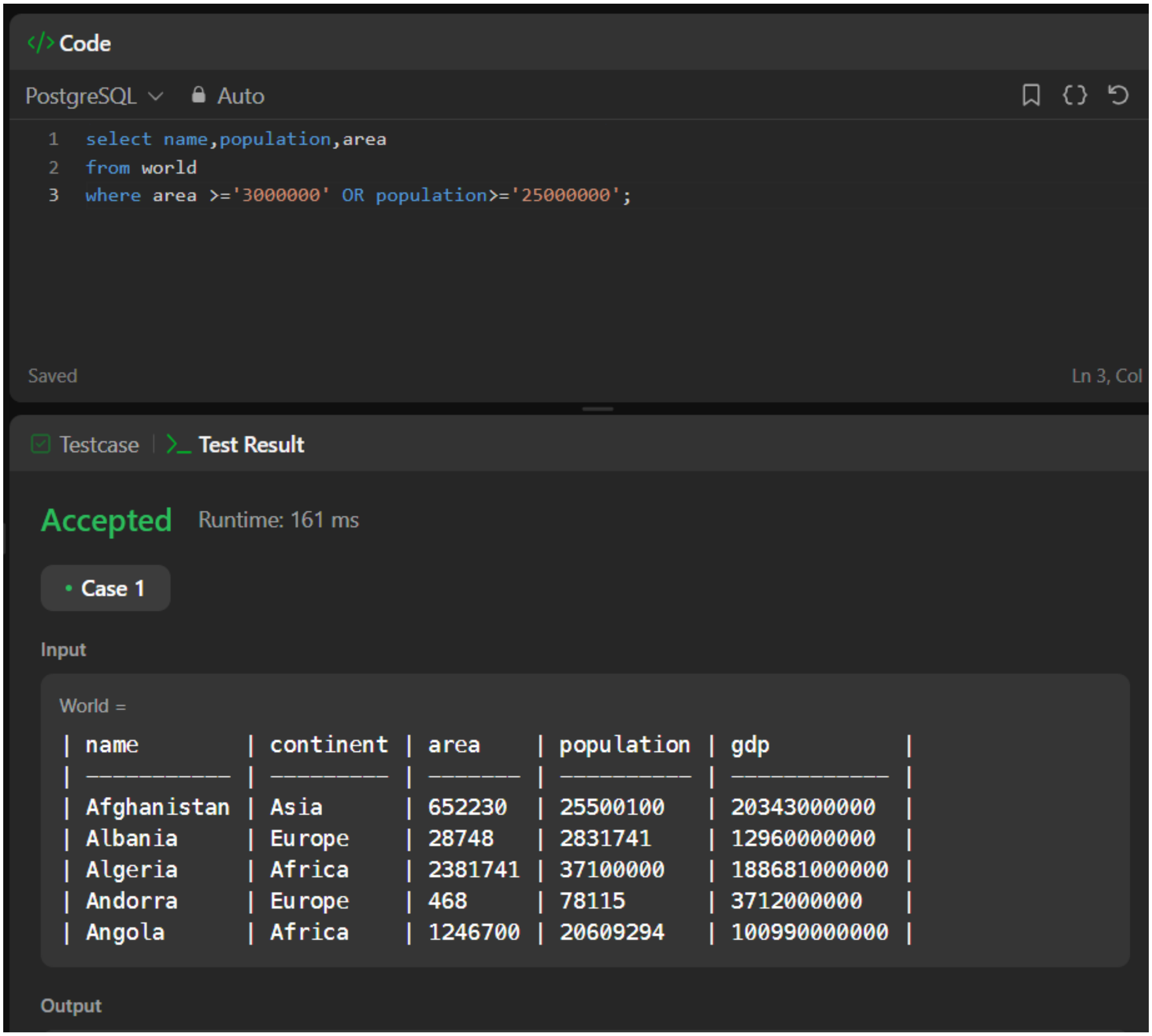
Case 1

Input

```
Customer =
| id | name | referee_id |
| -- | ---- | ---------- |
| 1  | Will | null       |
| 2  | Jane | null       |
| 3  | Alex | 2          |
| 4  | Bill | null       |
| 5  | Zack | 1          |
| 6  | Mark | 2          |
```

Output

Big Countries



```sql
select name,population,area
from world
where area >='3000000' OR population>='25000000';
```

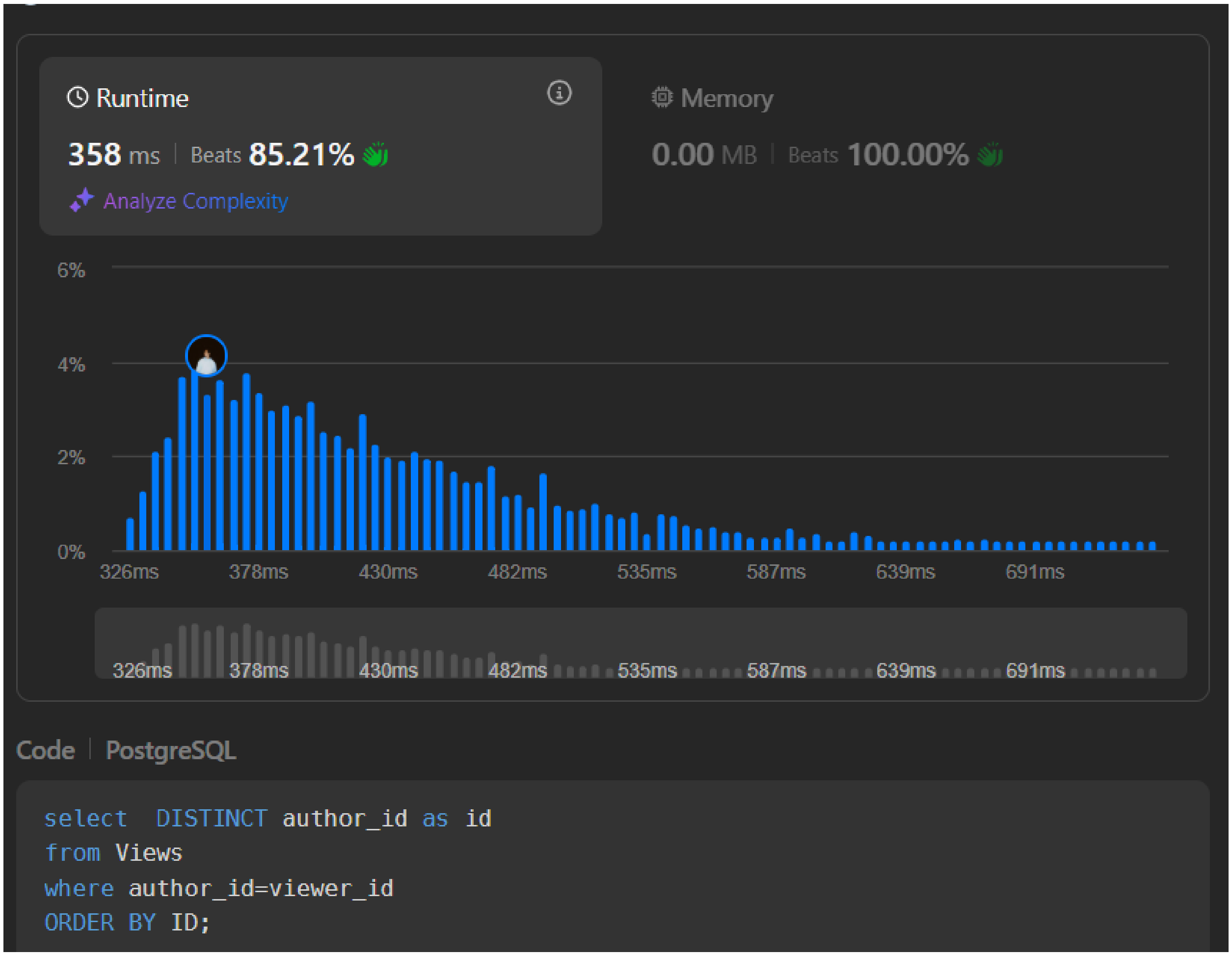
Accepted Runtime: 161 ms

Input

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

Output

Article Views I



Runtime

358 ms | Beats 85.21%

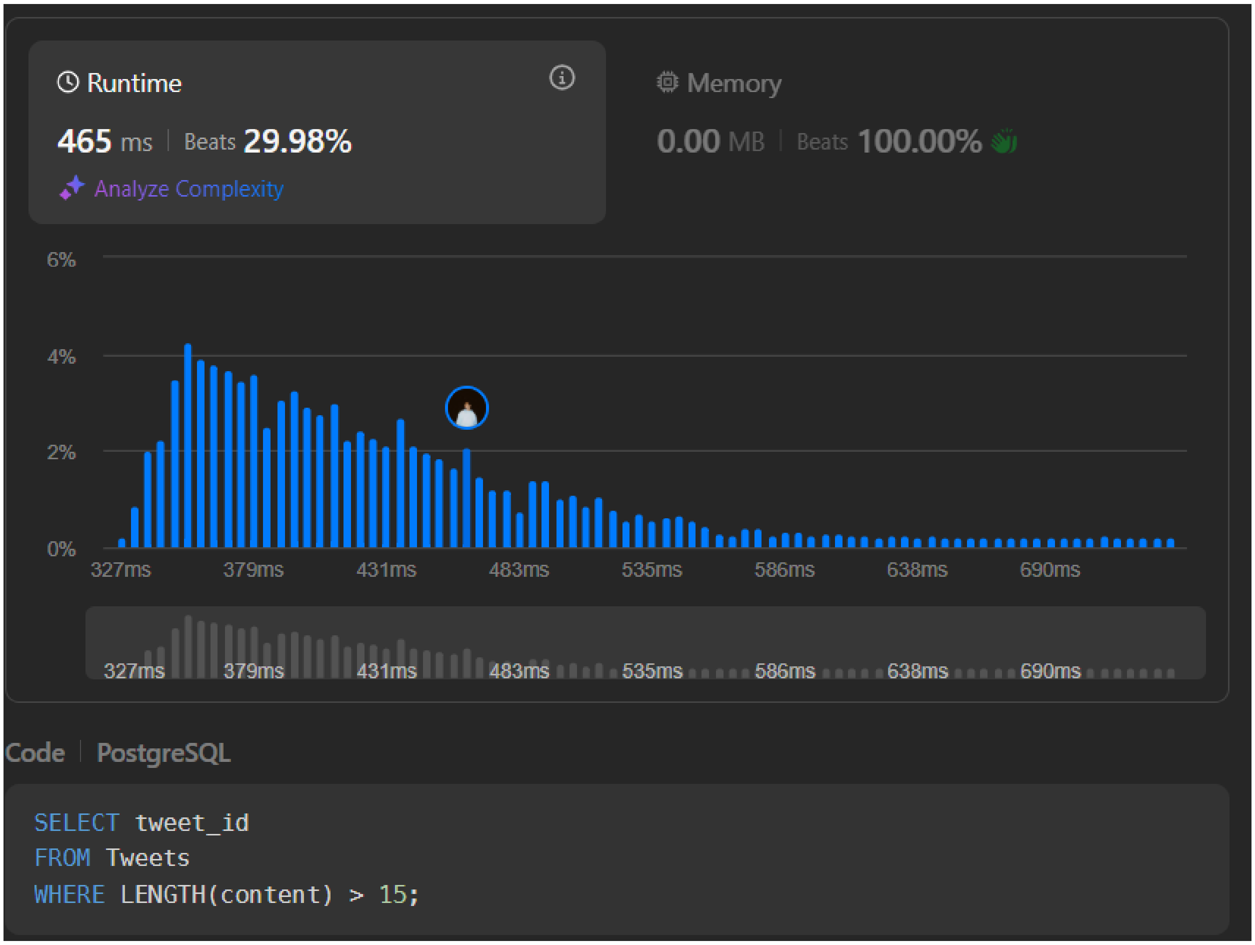
Memory

0.00 MB | Beats 100.00%

Code | PostgreSQL

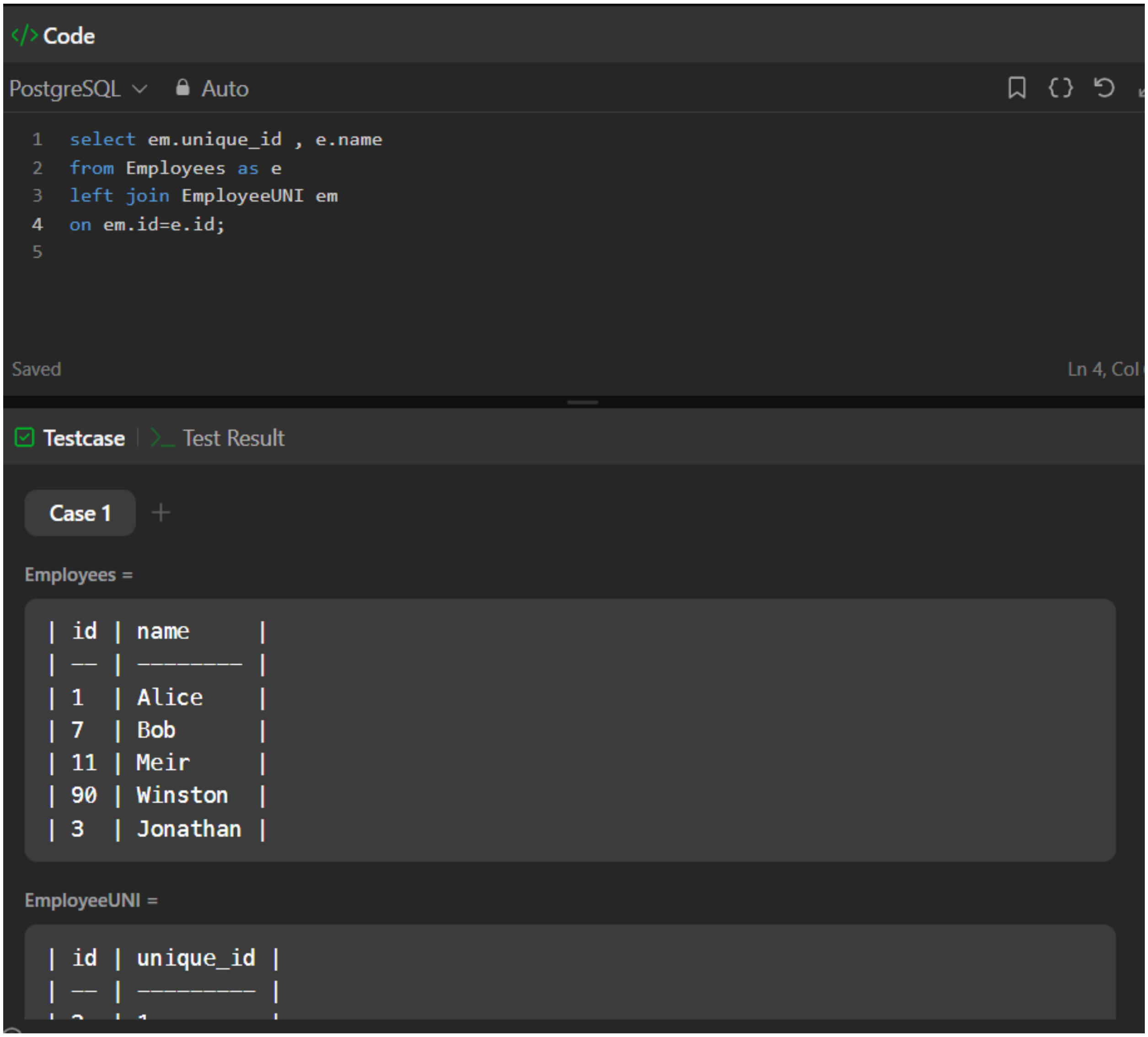
```
select  DISTINCT author_id as id
from Views
where author_id=viewer_id
ORDER BY ID;
```

Invalid Tweets



```
SELECT tweet_id
FROM Tweets
WHERE LENGTH(content) > 15;
```

Replace Employee ID With The Unique Identifier



```sql
select em.unique_id , e.name
from Employees as e
left join EmployeeUNI em
on em.id=e.id;

```

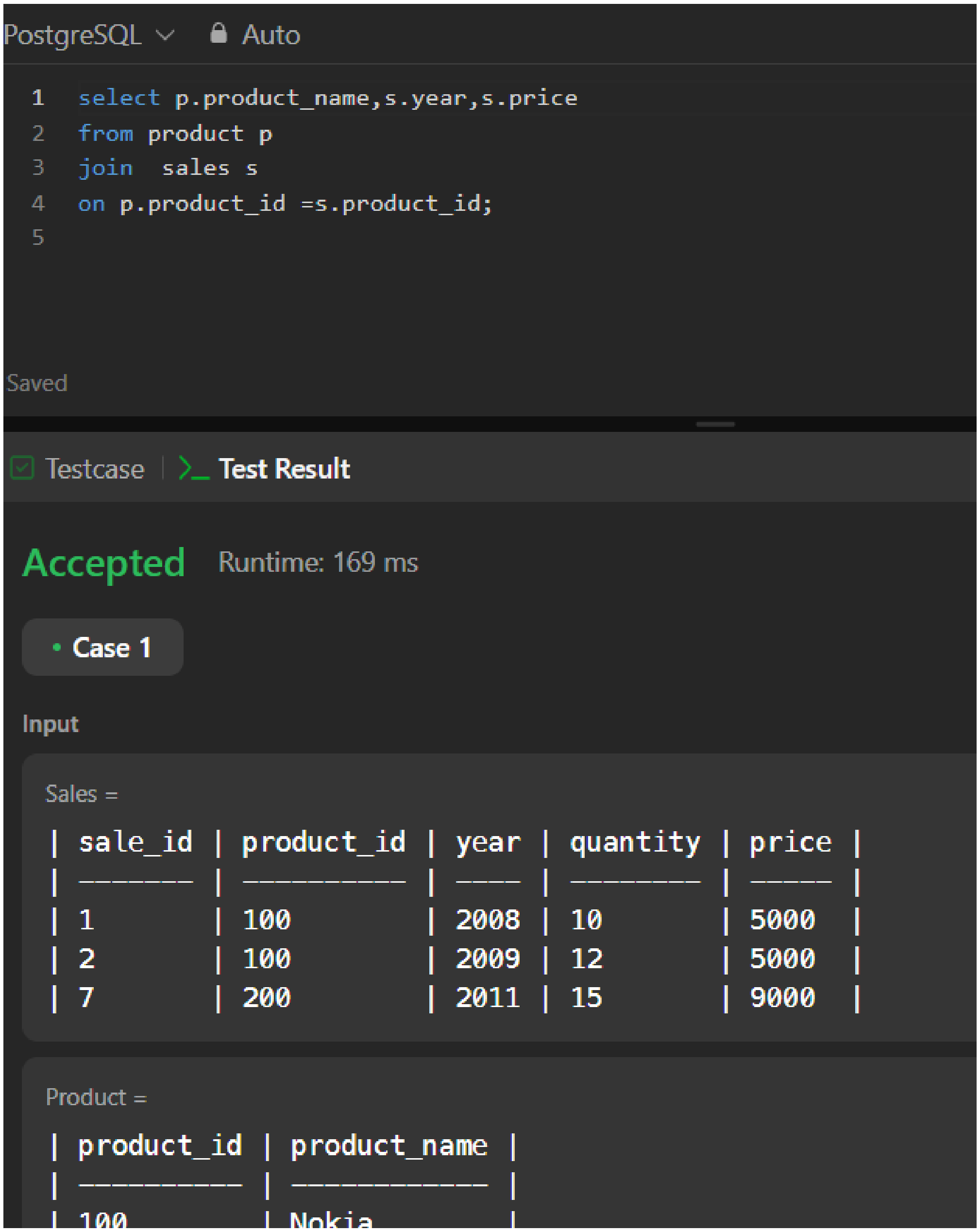
Employees =

```
| id | name     |
| -- | -------- |
| 1  | Alice    |
| 7  | Bob      |
| 11 | Meir     |
| 90 | Winston  |
| 3  | Jonathan |
```

EmployeeUNI =

```
| id | unique_id |
| -- | --------- |
```

[Product Sales Analysis I](#)



```sql
select p.product_name,s.year,s.price
from product p
join  sales s
on p.product_id =s.product_id;
```

Accepted Runtime: 169 ms

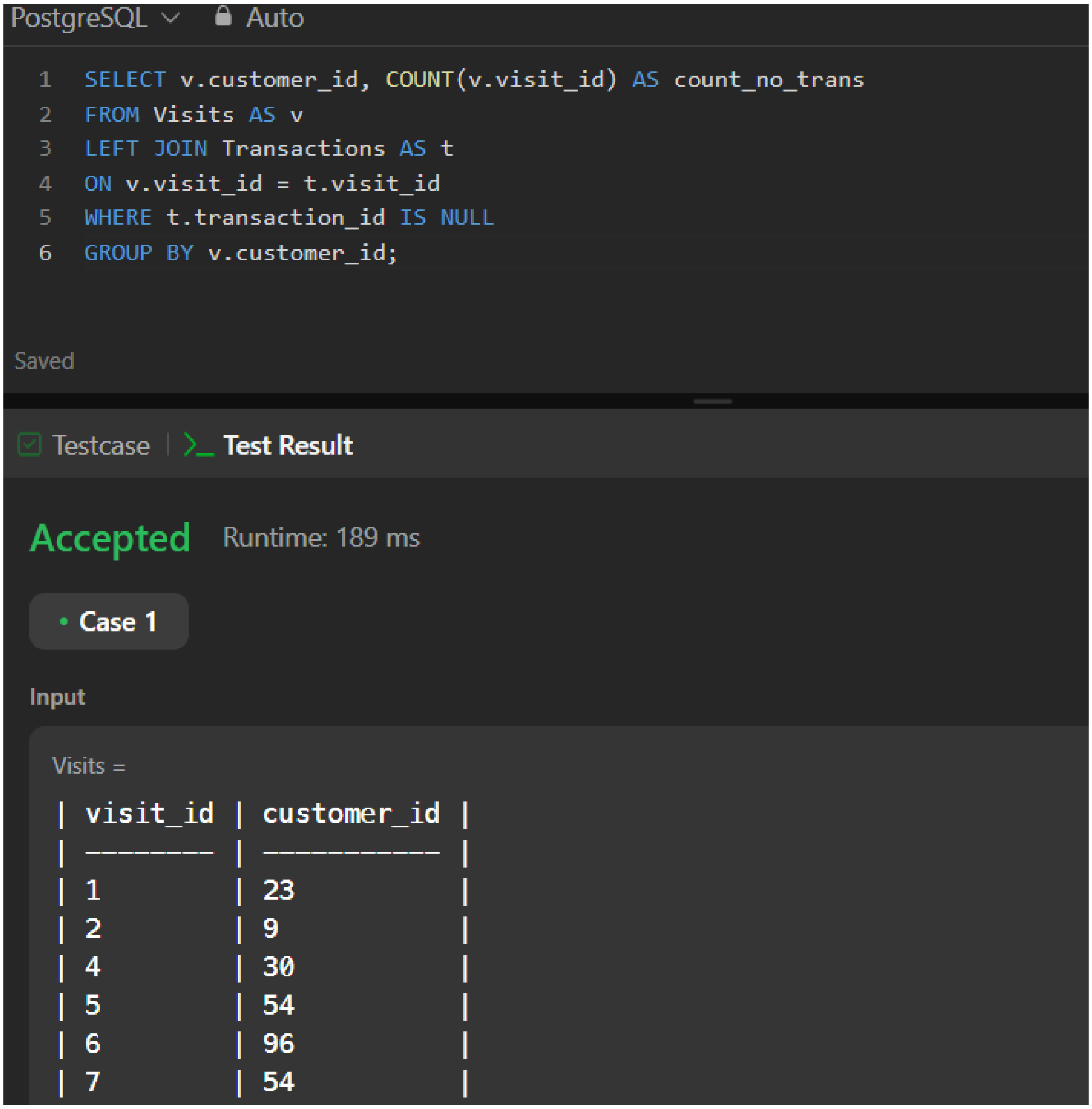
Sales =

```
| sale_id | product_id | year | quantity | price |
| ------- | ---------- | ---- | -------- | ----- |
| 1       | 100        | 2008 | 10       | 5000  |
| 2       | 100        | 2009 | 12       | 5000  |
| 7       | 200        | 2011 | 15       | 9000  |
```

Product =

```
| product_id | product_name |
| ---------- | ------------ |
| 100        | Nokia        |
```

Customer Who Visited but Did Not Make Any Transactions



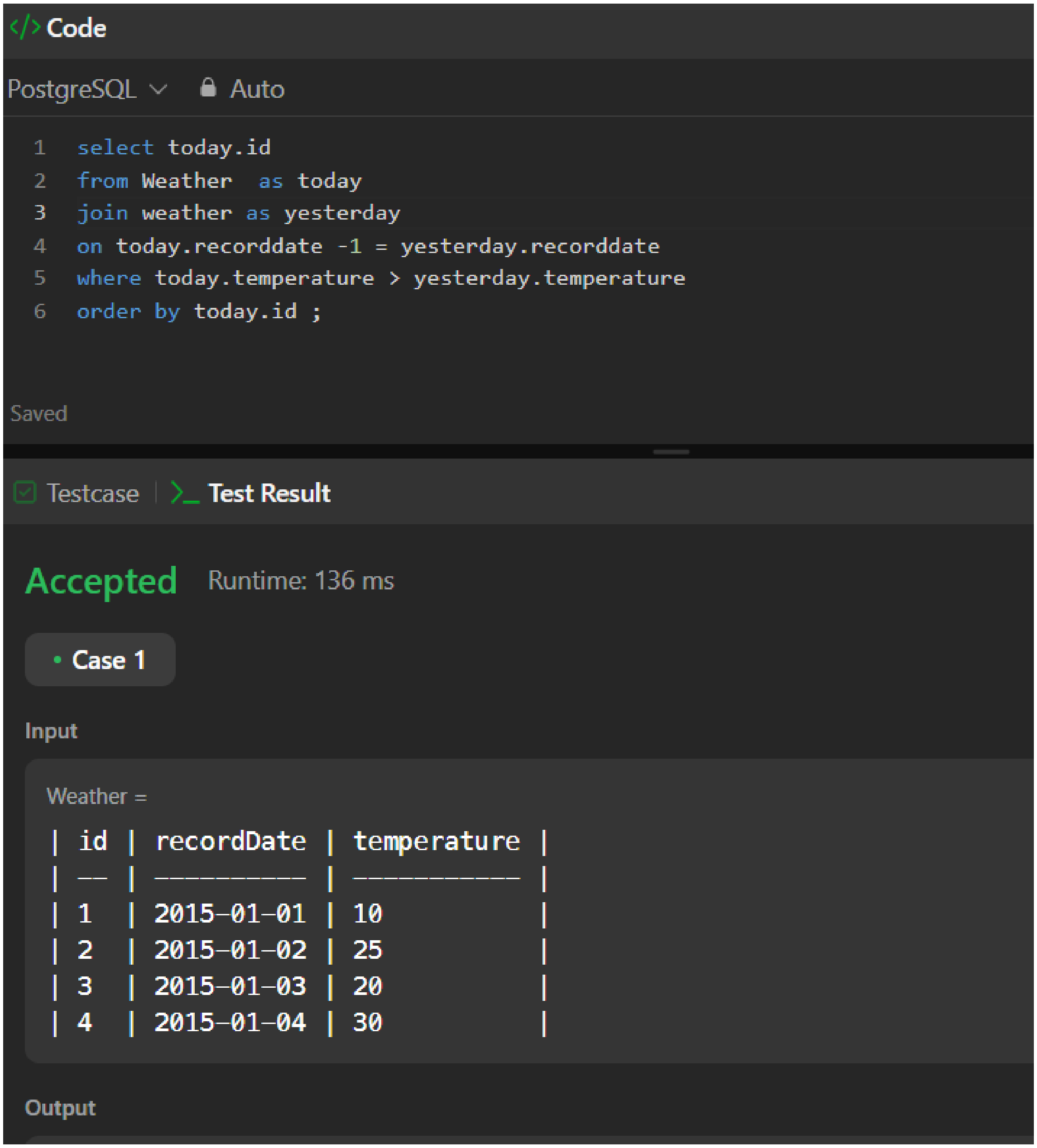
```sql
SELECT v.customer_id, COUNT(v.visit_id) AS count_no_trans
FROM Visits AS v
LEFT JOIN Transactions AS t
ON v.visit_id = t.visit_id
WHERE t.transaction_id IS NULL
GROUP BY v.customer_id;
```

Accepted Runtime: 189 ms

Input

```
Visits =
| visit_id | customer_id |
| -------- | ----------- |
| 1        | 23          |
| 2        | 9           |
| 4        | 30          |
| 5        | 54          |
| 6        | 96          |
| 7        | 54          |
```

Rising Temperature



```sql
select today.id
from Weather  as today
join weather as yesterday
on today.recorddate -1 = yesterday.recorddate
where today.temperature > yesterday.temperature
order by today.id ;
```

Accepted Runtime: 136 ms

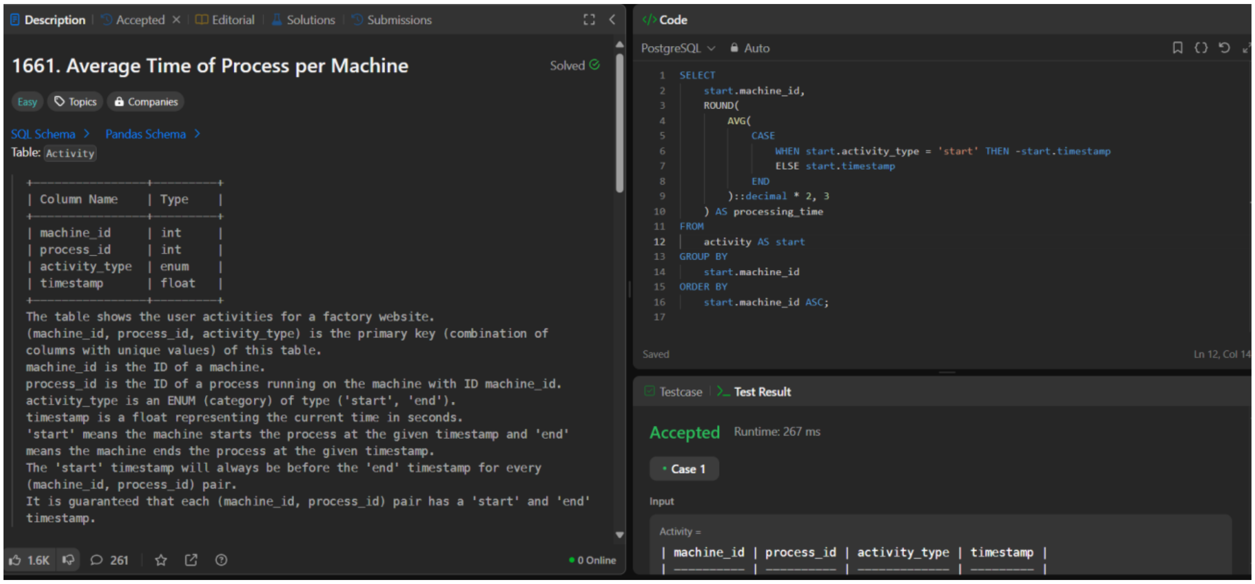
Input

Weather =

```
| id | recordDate | temperature |
| -- | ---------- | ----------- |
| 1  | 2015-01-01 | 10          |
| 2  | 2015-01-02 | 25          |
| 3  | 2015-01-03 | 20          |
| 4  | 2015-01-04 | 30          |
```

Output

Average Time of Process per Machine



1661. Average Time of Process per Machine

Solved

Table: Activity

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| machine_id     | int     |
| process_id     | int     |
| activity_type  | enum    |
| timestamp      | float   |
+----------------+---------+
The table shows the user activities for a factory website.
(machine_id, process_id, activity_type) is the primary key (combination of
columns with unique values) of this table.
machine_id is the ID of a machine.
process_id is the ID of a process running on the machine with ID machine_id.
activity_type is an ENUM (category) of type ('start', 'end').
timestamp is a float representing the current time in seconds.
'start' means the machine starts the process at the given timestamp and 'end'
means the machine ends the process at the given timestamp.
The 'start' timestamp will always be before the 'end' timestamp for every
(machine_id, process_id) pair.
It is guaranteed that each (machine_id, process_id) pair has a 'start' and 'end'
timestamp.
```

PostgreSQL Auto

```sql
SELECT
    start.machine_id,
    ROUND(
        AVG(
            CASE
                WHEN start.activity_type = 'start' THEN -start.timestamp
                ELSE start.timestamp
            END
        )::decimal * 2, 3
    ) AS processing_time
FROM
    activity AS start
GROUP BY
    start.machine_id
ORDER BY
    start.machine_id ASC;

```

Saved Ln 12, Col 14

Accepted Runtime: 267 ms

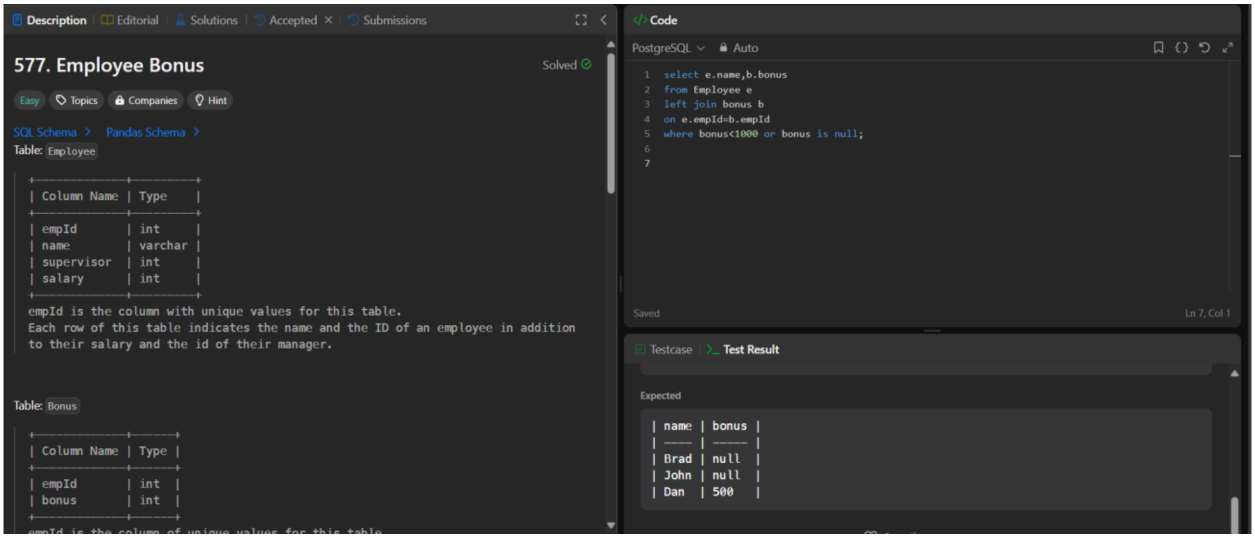
• Case 1

Input

Activity =

```
| machine_id | process_id | activity_type | timestamp |
| ---------- | ---------- | ------------- | --------- |
```

[Employee Bonus]



577. Employee Bonus

Solved

Table: Employee

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| empId       | int     |
| name        | varchar |
| supervisor  | int     |
| salary      | int     |
+-------------+---------+
empId is the column with unique values for this table.
Each row of this table indicates the name and the ID of an employee in addition
to their salary and the id of their manager.
```

Table: Bonus

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| empId       | int  |
| bonus       | int  |
+-------------+------+
empId is the column of unique values for this table.
```

PostgreSQL Auto

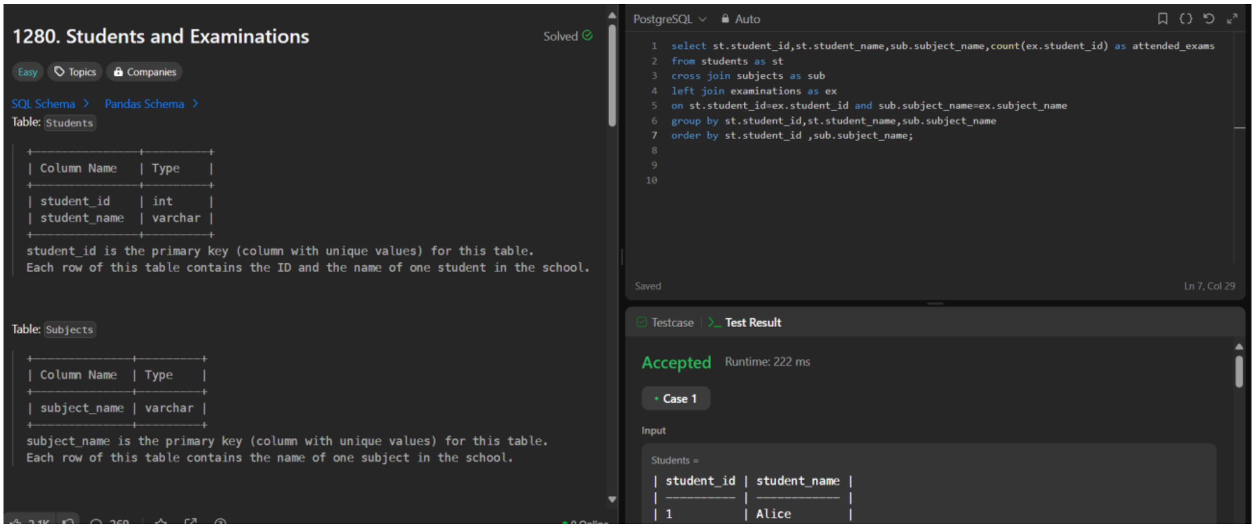
```sql
select e.name,b.bonus
from Employee e
left join bonus b
on e.empId=b.empId
where bonus<1000 or bonus is null;

```

Saved Ln 7, Col 1

Expected

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

[Students and Examinations]



1280. Students and Examinations

Solved

Table: `Students`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| student_id   | int     |
| student_name | varchar |
+--------------+---------+
student_id is the primary key (column with unique values) for this table.
Each row of this table contains the ID and the name of one student in the school.
```

Table: `Subjects`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| subject_name | varchar |
+--------------+---------+
subject_name is the primary key (column with unique values) for this table.
Each row of this table contains the name of one subject in the school.
```

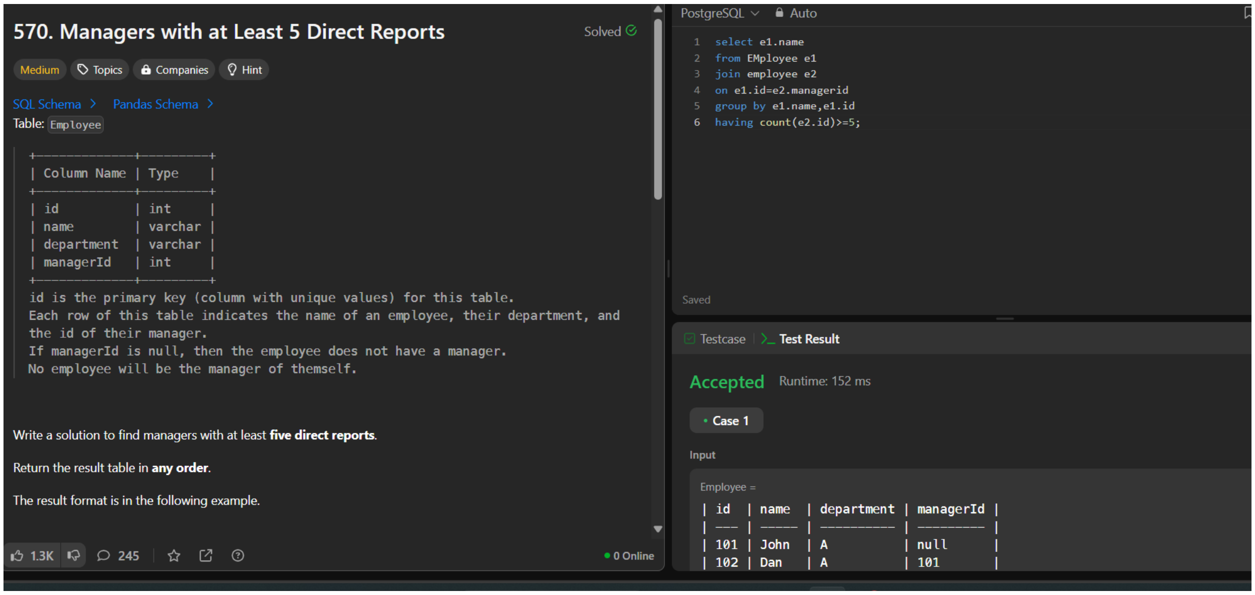
```
select st.student_id,st.student_name,sub.subject_name,count(ex.student_id) as attended_exams
from students as st
cross join subjects as sub
left join examinations as ex
on st.student_id=ex.student_id and sub.subject_name=ex.subject_name
group by st.student_id,st.student_name,sub.subject_name
order by st.student_id ,sub.subject_name;
```

Accepted Runtime: 222 ms

Input

```
Students =
| student_id | student_name |
| ---------- | ------------ |
| 1          | Alice        |
```

[Managers with at Least 5 Direct Reports]



570. Managers with at Least 5 Direct Reports

Solved

Table: `Employee`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| name        | varchar |
| department  | varchar |
| managerId   | int     |
+-------------+---------+
id is the primary key (column with unique values) for this table.
Each row of this table indicates the name of an employee, their department, and
the id of their manager.
If managerId is null, then the employee does not have a manager.
No employee will be the manager of themself.
```

Write a solution to find managers with at least **five direct reports**.

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

The result format is in the following example.

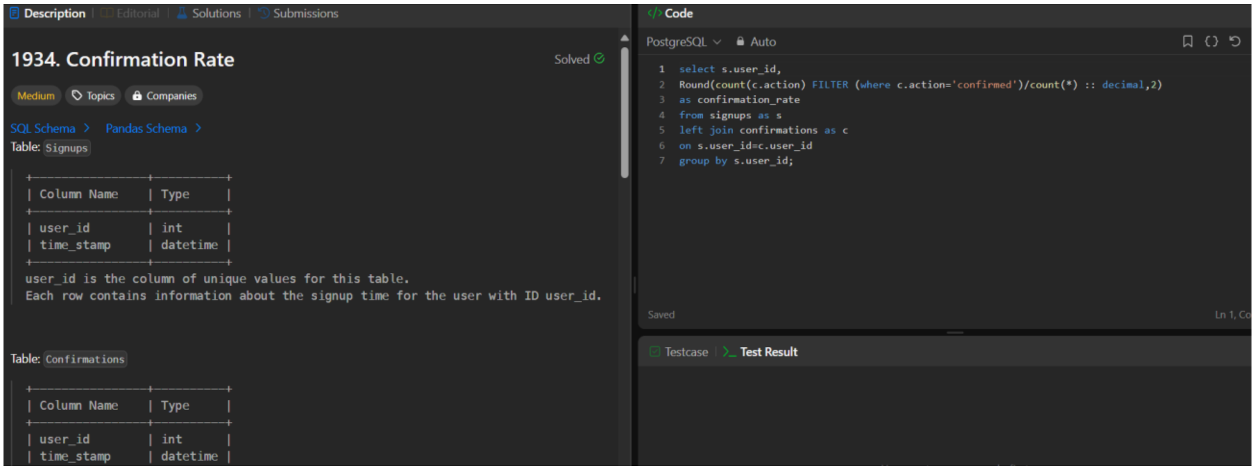
```
select e1.name
from EMployee e1
join employee e2
on e1.id=e2.managerid
group by e1.name,e1.id
having count(e2.id)>=5;
```

Accepted Runtime: 152 ms

Input

```
Employee =
| id  | name | department | managerId |
| --- | ---- | ---------- | --------- |
| 101 | John | A          | null      |
| 102 | Dan  | A          | 101       |
```

[Confirmation Rate]



1934. Confirmation Rate

Solved

SQL Schema > Pandas Schema >

Table: `Signups`

```
+----------------+----------+
| Column Name    | Type     |
+----------------+----------+
| user_id        | int      |
| time_stamp     | datetime |
+----------------+----------+
user_id is the column of unique values for this table.
Each row contains information about the signup time for the user with ID user_id.
```

Table: `Confirmations`

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

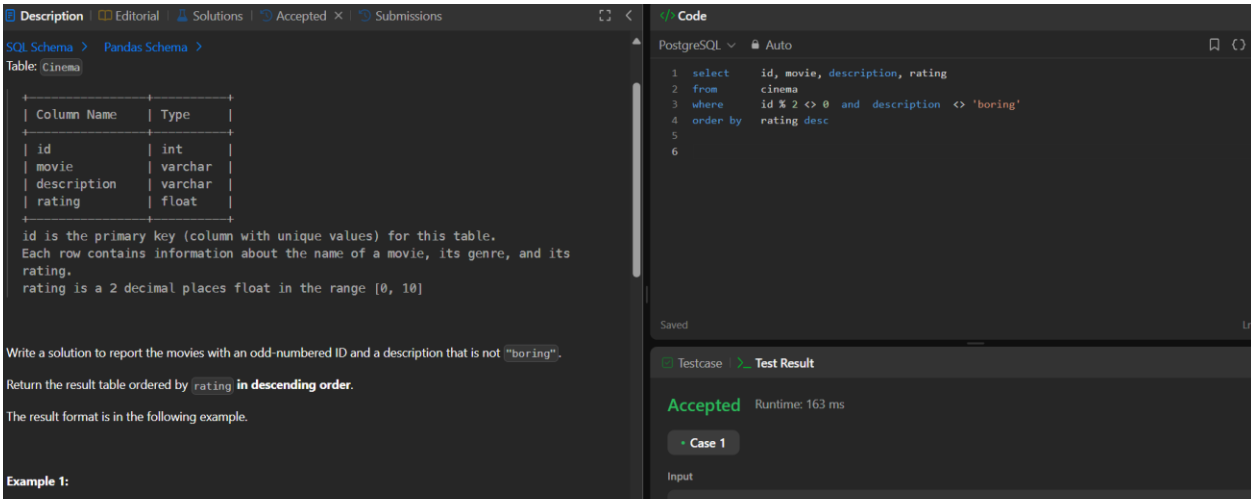
PostgreSQL Auto

```sql
select s.user_id,
Round(count(c.action) FILTER (where c.action='confirmed')/count(*) :: decimal,2)
as confirmation_rate
from signups as s
left join confirmations as c
on s.user_id=c.user_id
group by s.user_id;
```

Saved

Ln 1, Co

Not Boring Movies



SQL Schema > Pandas Schema >

Table: `Cinema`

```
+----------------+----------+
| Column Name    | Type     |
+----------------+----------+
| id             | int      |
| movie          | varchar  |
| description    | varchar  |
| rating         | float    |
+----------------+----------+
id is the primary key (column with unique values) for this table.
Each row contains information about the name of a movie, its genre, and its
rating.
rating is a 2 decimal places float in the range [0, 10]
```

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

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

The result format is in the following example.

Example 1:

PostgreSQL Auto

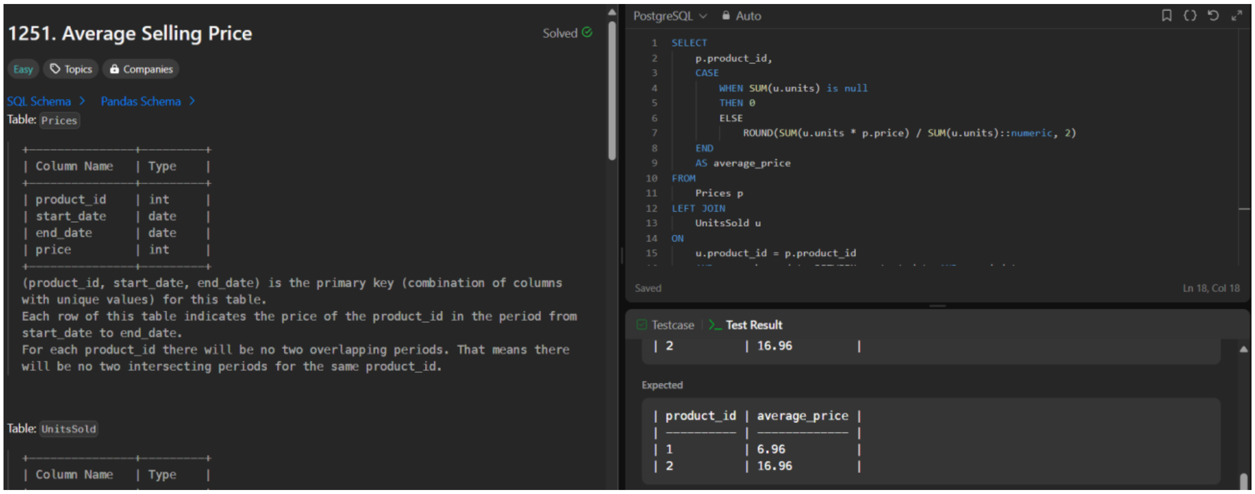
```sql
select     id, movie, description, rating
from       cinema
where      id % 2 <> 0  and  description  <> 'boring'
order by   rating desc


```

Saved

Accepted Runtime: 163 ms

Input

Average Selling Price



1251. Average Selling Price

Solved

Easy · Topics · Companies

SQL Schema > Pandas Schema >

Table: Prices

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| start_date    | date    |
| end_date      | date    |
| price         | int     |
+---------------+---------+
(product_id, start_date, end_date) is the primary key (combination of columns
with unique values) for this table.
Each row of this table indicates the price of the product_id in the period from
start_date to end_date.
For each product_id there will be no two overlapping periods. That means there
will be no two intersecting periods for the same product_id.
```

Table: UnitsSold

```
+---------------+---------+
| Column Name   | Type    |
```

PostgreSQL · Auto

```sql
SELECT
    p.product_id,
    CASE
        WHEN SUM(u.units) is null
        THEN 0
        ELSE
            ROUND(SUM(u.units * p.price) / SUM(u.units)::numeric, 2)
    END
    AS average_price
FROM
    Prices p
LEFT JOIN
    UnitsSold u
ON
    u.product_id = p.product_id
```

Saved · Ln 18, Col 18

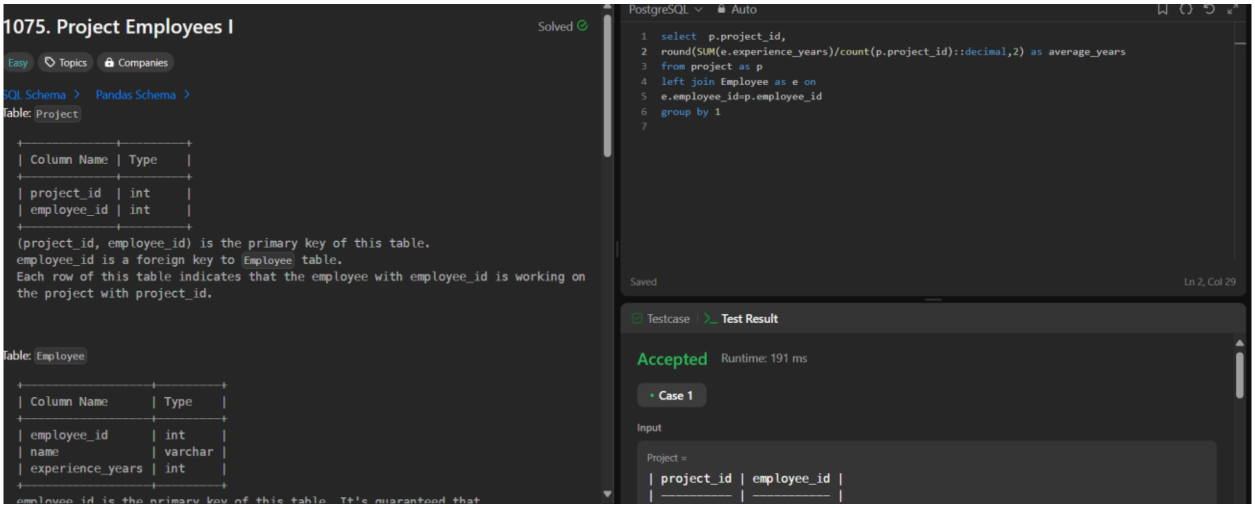
Testcase · Test Result

```
| 2          | 16.96         |
```

Expected

```
| product_id | average_price |
| ---------- | ------------- |
| 1          | 6.96          |
| 2          | 16.96         |
```

Project Employees I



1075. Project Employees I

Solved

Easy · Topics · Companies

SQL Schema > Pandas Schema >

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.
Each row of this table indicates that the employee with employee_id is working on
the project with project_id.
```

Table: Employee

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| employee_id      | int     |
| name             | varchar |
| experience_years | int     |
+------------------+---------+
employee_id is the primary key of this table. It's guaranteed that
```

PostgreSQL · Auto

```sql
select  p.project_id,
round(SUM(e.experience_years)/count(p.project_id)::decimal,2) as average_years
from project as p
left join Employee as e on
e.employee_id=p.employee_id
group by 1
```

Saved · Ln 2, Col 29

Testcase · Test Result

Accepted Runtime: 191 ms

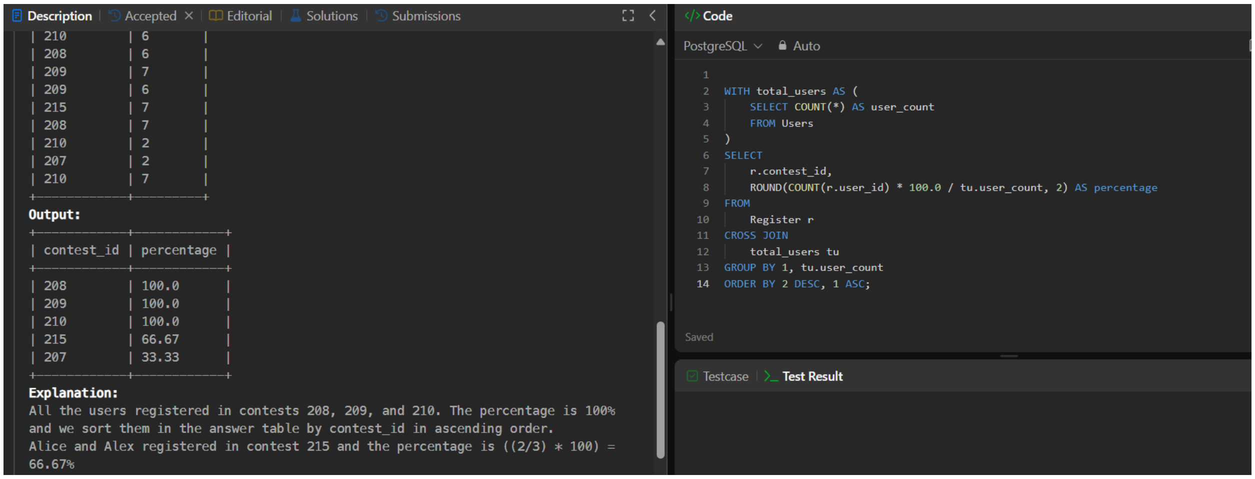
Case 1

Input

Project =

```
| project_id | employee_id |
| ---------- | ----------- |
```

Percentage of Users Attended a Contest



Description | Accepted | Editorial | Solutions | Submissions

```
| 210        | 6       |
| 208        | 6       |
| 209        | 7       |
| 209        | 6       |
| 215        | 7       |
| 208        | 7       |
| 210        | 2       |
| 207        | 2       |
| 210        | 7       |
+------------+---------+
Output:
+------------+------------+
| contest_id | percentage |
+------------+------------+
| 208        | 100.0      |
| 209        | 100.0      |
| 210        | 100.0      |
| 215        | 66.67      |
| 207        | 33.33      |
+------------+------------+
Explanation:
All the users registered in contests 208, 209, and 210. The percentage is 100%
and we sort them in the answer table by contest_id in ascending order.
Alice and Alex registered in contest 215 and the percentage is ((2/3) * 100) =
66.67%
```

Code

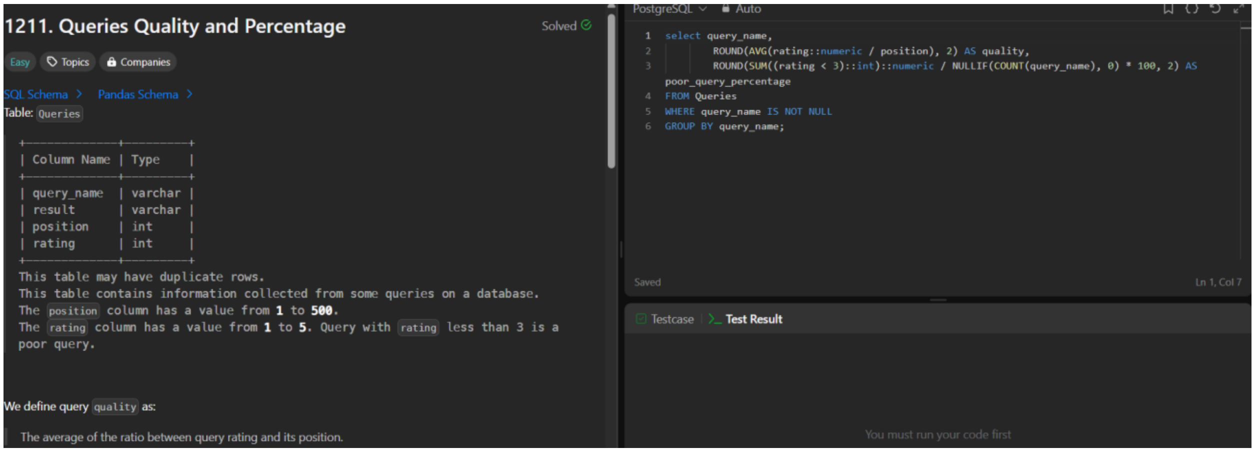
PostgreSQL · Auto

```sql

WITH total_users AS (
    SELECT COUNT(*) AS user_count
    FROM Users
)
SELECT
    r.contest_id,
    ROUND(COUNT(r.user_id) * 100.0 / tu.user_count, 2) AS percentage
FROM
    Register r
CROSS JOIN
    total_users tu
GROUP BY 1, tu.user_count
ORDER BY 2 DESC, 1 ASC;
```

Saved

Testcase · Test Result

Queries Quality and Percentage



1211. Queries Quality and Percentage

Table: `Queries`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| query_name  | varchar |
| result      | varchar |
| position    | int     |
| rating      | int     |
+-------------+---------+
This table may have duplicate rows.
This table contains information collected from some queries on a database.
The position column has a value from 1 to 500.
The rating column has a value from 1 to 5. Query with rating less than 3 is a
poor query.
```

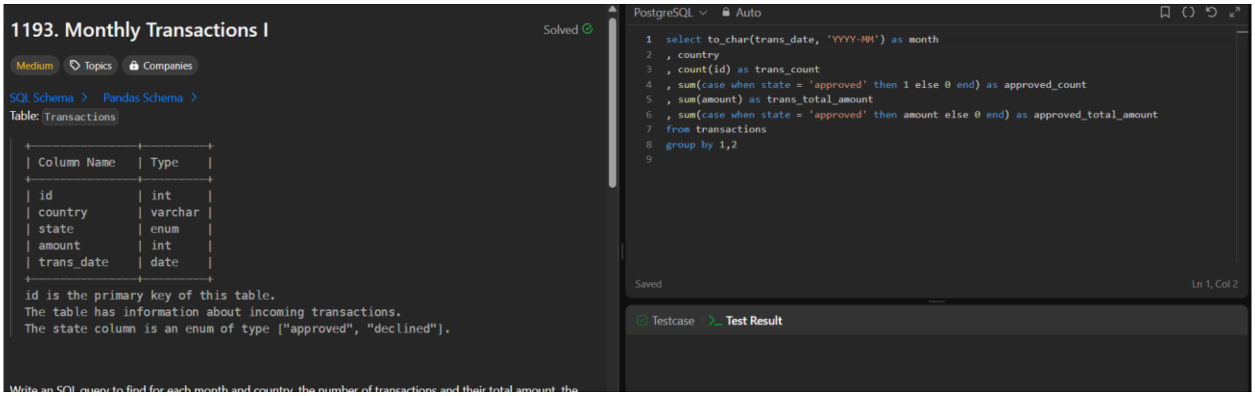
We define query `quality` as:

> The average of the ratio between query rating and its position.

```sql
select query_name,
        ROUND(AVG(rating::numeric / position), 2) AS quality,
        ROUND(SUM((rating < 3)::int)::numeric / NULLIF(COUNT(query_name), 0) * 100, 2) AS
poor_query_percentage
FROM Queries
WHERE query_name IS NOT NULL
GROUP BY query_name;
```

You must run your code first

Monthly Transactions I



1193. Monthly Transactions I

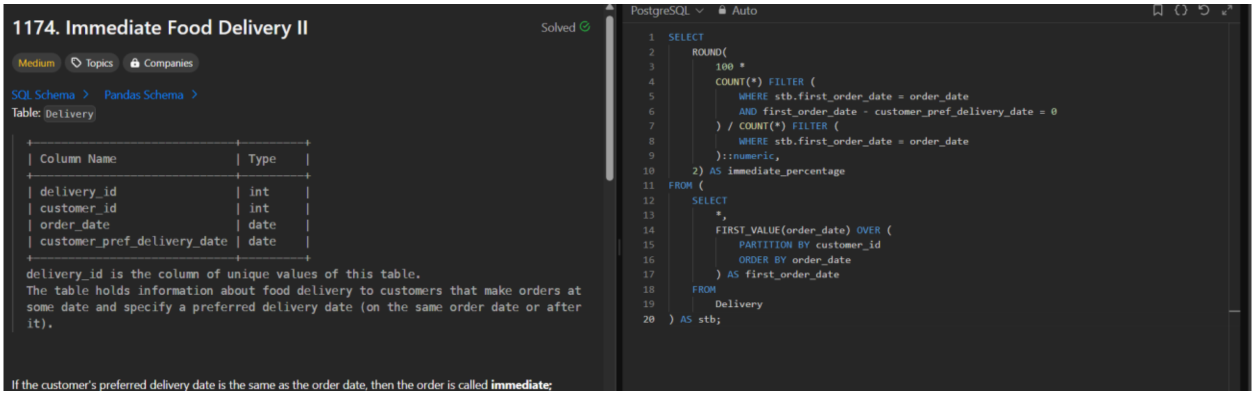
Table: `Transactions`

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

```sql
select to_char(trans_date, 'YYYY-MM') as month
, country
, count(id) as trans_count
, sum(case when state = 'approved' then 1 else 0 end) as approved_count
, sum(amount) as trans_total_amount
, sum(case when state = 'approved' then amount else 0 end) as approved_total_amount
from transactions
group by 1,2

```

Immediate Food Delivery II



1174. Immediate Food Delivery II

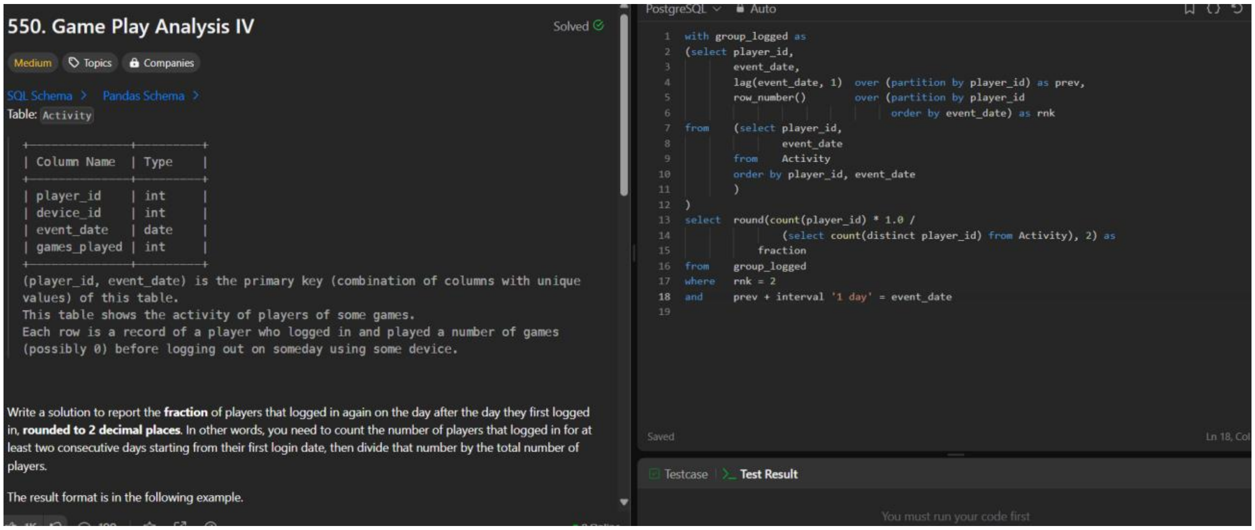
Table: `Delivery`

```
+-----------------------------+---------+
| Column Name                 | Type    |
+-----------------------------+---------+
| delivery_id                 | int     |
| customer_id                 | int     |
| order_date                  | date    |
| customer_pref_delivery_date | date    |
+-----------------------------+---------+
delivery_id is the column of unique values of this table.
The table holds information about food delivery to customers that make orders at
some date and specify a preferred delivery date (on the same order date or after
it).
```

If the customer's preferred delivery date is the same as the order date, then the order is called **immediate;**

```sql
SELECT
    ROUND(
        100 *
        COUNT(*) FILTER (
            WHERE stb.first_order_date = order_date
            AND first_order_date - customer_pref_delivery_date = 0
        ) / COUNT(*) FILTER (
            WHERE stb.first_order_date = order_date
        )::numeric,
    2) AS immediate_percentage
FROM (
    SELECT
        *,
        FIRST_VALUE(order_date) OVER (
            PARTITION BY customer_id
            ORDER BY order_date
        ) AS first_order_date
    FROM
        Delivery
) AS stb;
```

Game Play Analysis IV



550. Game Play Analysis IV

Table: `Activity`

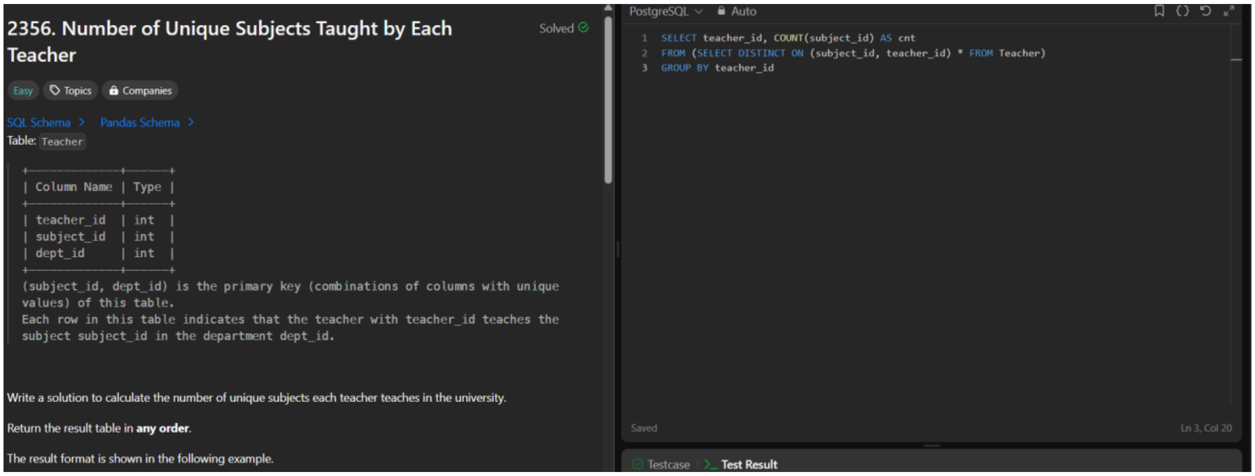
```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| player_id    | int     |
| device_id    | int     |
| event_date   | date    |
| games_played | int     |
+--------------+---------+
(player_id, event_date) is the primary key (combination of columns with unique
values) of this table.
This table shows the activity of players of some games.
Each row is a record of a player who logged in and played a number of games
(possibly 0) before logging out on someday using some device.
```

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

The result format is in the following example.

```sql
with group_logged as
(select player_id,
        event_date,
        lag(event_date, 1)  over (partition by player_id) as prev,
        row_number()        over (partition by player_id
                                  order by event_date) as rnk
from    (select player_id,
                event_date
        from    Activity
        order by player_id, event_date
        )
)
select  round(count(player_id) * 1.0 /
                (select count(distinct player_id) from Activity), 2) as
            fraction
from    group_logged
where   rnk = 2
and     prev + interval '1 day' = event_date
```

Number of Unique Subjects Taught by Each Teacher



2356. Number of Unique Subjects Taught by Each Teacher

Table: `Teacher`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| teacher_id  | int  |
| subject_id  | int  |
| dept_id     | int  |
+-------------+------+
(subject_id, dept_id) is the primary key (combinations of columns with unique
values) of this table.
Each row in this table indicates that the teacher with teacher_id teaches the
subject subject_id in the department dept_id.
```

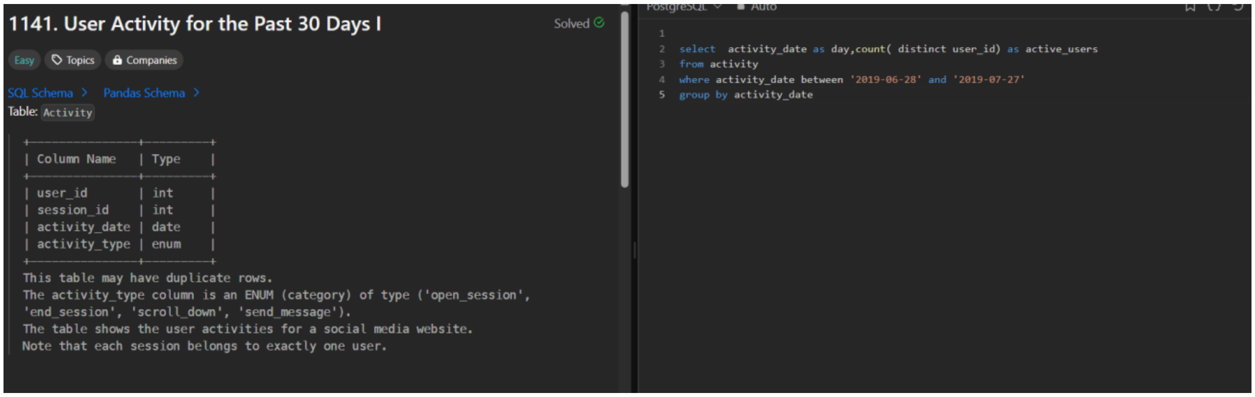
Write a solution to calculate the number of unique subjects each teacher teaches in the university.

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

The result format is shown in the following example.

```sql
SELECT teacher_id, COUNT(subject_id) AS cnt
FROM (SELECT DISTINCT ON (subject_id, teacher_id) * FROM Teacher)
GROUP BY teacher_id
```

User Activity for the Past 30 Days I



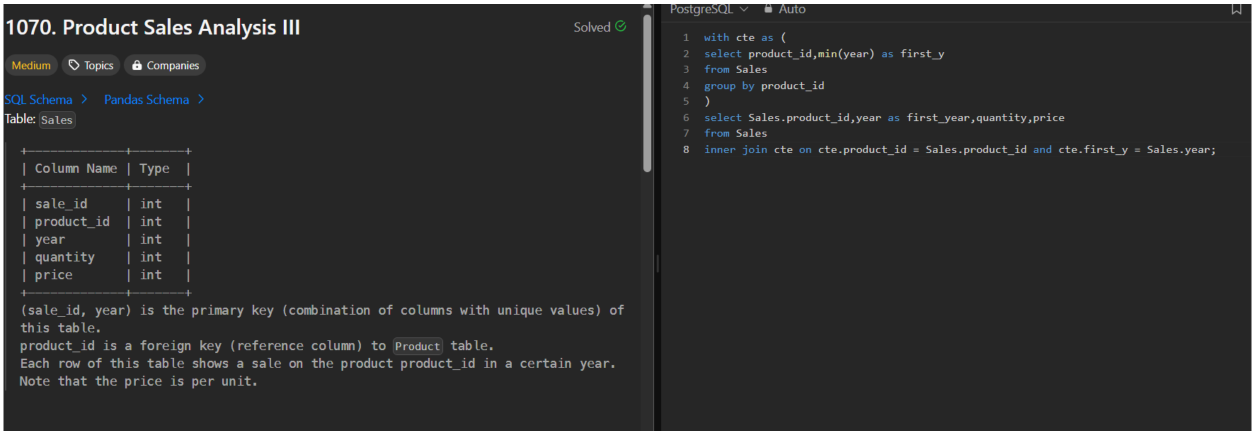
1141. User Activity for the Past 30 Days I

Table: `Activity`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| session_id    | int     |
| activity_date | date    |
| activity_type | enum    |
+---------------+---------+
This table may have duplicate rows.
The activity_type column is an ENUM (category) of type ('open_session',
'end_session', 'scroll_down', 'send_message').
The table shows the user activities for a social media website.
Note that each session belongs to exactly one user.
```

```sql

select  activity_date as day,count( distinct user_id) as active_users
from activity
where activity_date between '2019-06-28' and '2019-07-27'
group by activity_date
```

Product Sales Analysis III



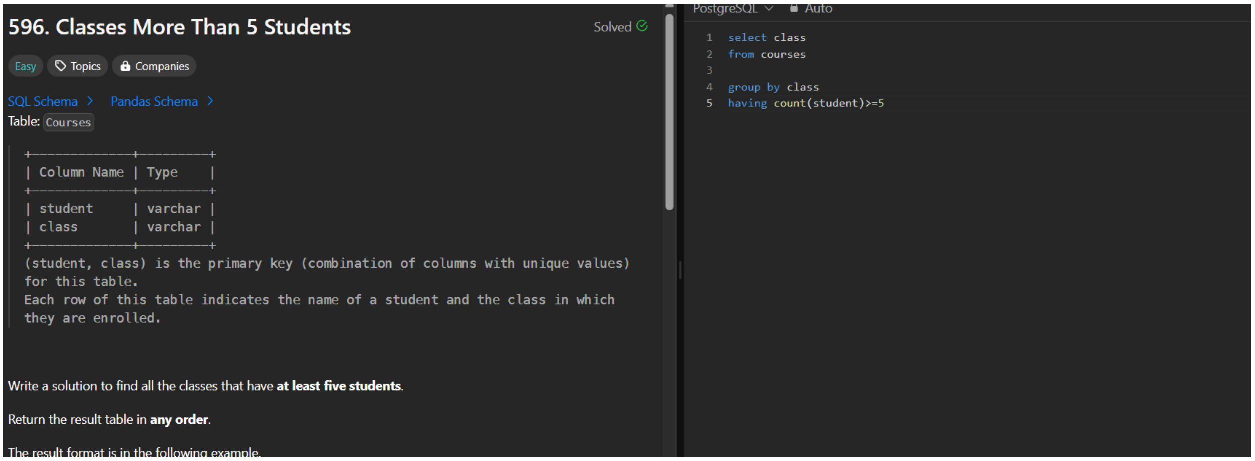
1070. Product Sales Analysis III

Table: `Sales`

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| sale_id     | int   |
| product_id  | int   |
| year        | int   |
| quantity    | int   |
| price       | int   |
+-------------+-------+
(sale_id, year) is the primary key (combination of columns with unique values) of
this table.
product_id is a foreign key (reference column) to Product table.
Each row of this table shows a sale on the product product_id in a certain year.
Note that the price is per unit.
```

```sql
with cte as (
select product_id,min(year) as first_y
from Sales
group by product_id
)
select Sales.product_id,year as first_year,quantity,price
from Sales
inner join cte on cte.product_id = Sales.product_id and cte.first_y = Sales.year;
```

Classes More Than 5 Students



596. Classes More Than 5 Students

Table: `Courses`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| student     | varchar |
| class       | varchar |
+-------------+---------+
(student, class) is the primary key (combination of columns with unique values)
for this table.
Each row of this table indicates the name of a student and the class in which
they are enrolled.
```

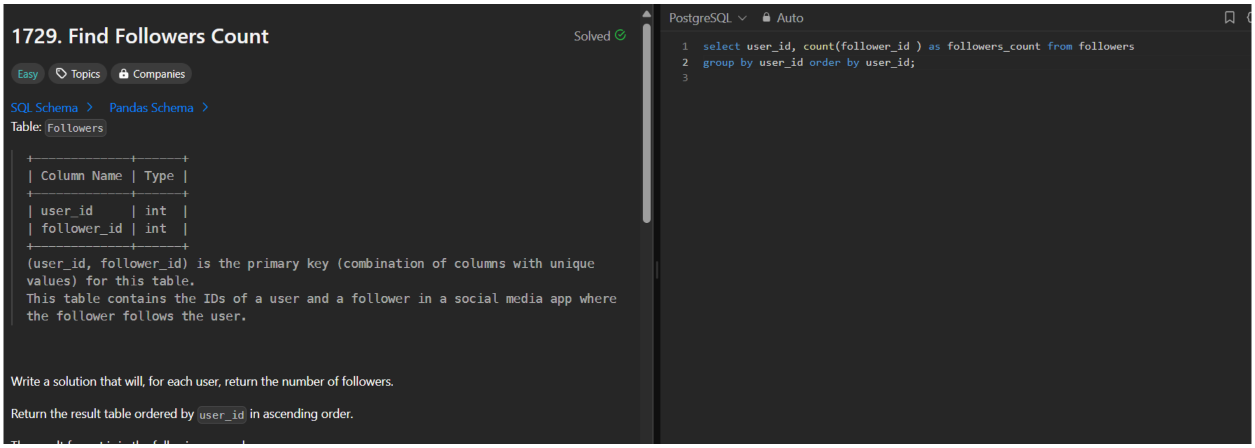
Write a solution to find all the classes that have **at least five students**.

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

The result format is in the following example.

```sql
select class
from courses

group by class
having count(student)>=5
```

Find Followers Count



1729. Find Followers Count

Table: `Followers`

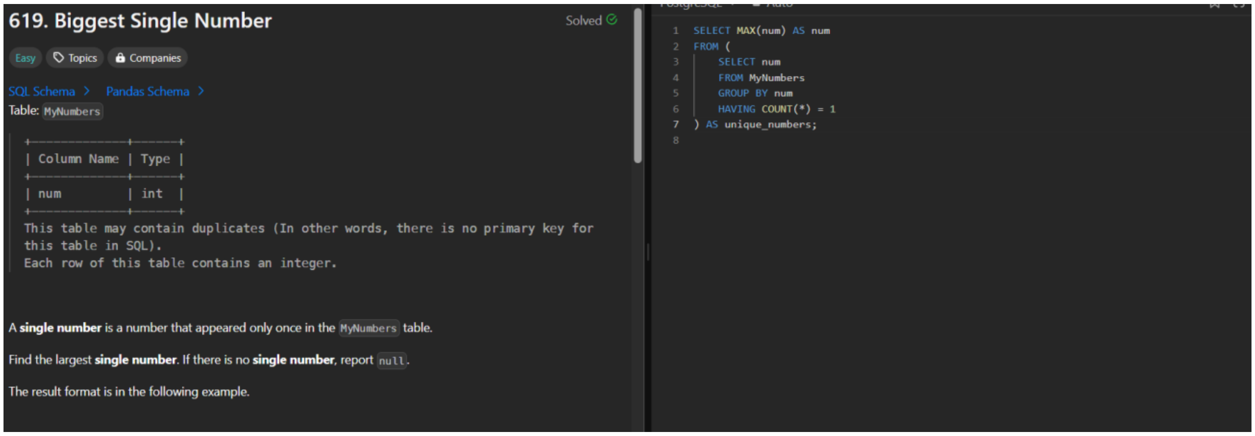
```
+-------------+------+
| Column Name | Type |
+-------------+------+
| user_id     | int  |
| follower_id | int  |
+-------------+------+
(user_id, follower_id) is the primary key (combination of columns with unique
values) for this table.
This table contains the IDs of a user and a follower in a social media app where
the follower follows the user.
```

Write a solution that will, for each user, return the number of followers.

Return the result table ordered by `user_id` in ascending order.

```sql
select user_id, count(follower_id ) as followers_count from followers
group by user_id order by user_id;
```

Biggest Single Number



619. Biggest Single Number

Solved

Table: `MyNumbers`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| num         | int  |
+-------------+------+
This table may contain duplicates (In other words, there is no primary key for
this table in SQL).
Each row of this table contains an integer.
```

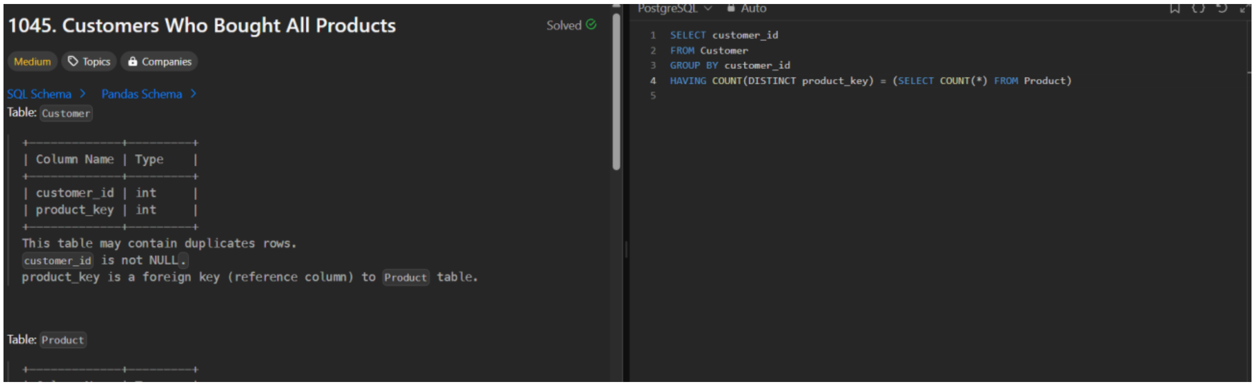
A **single number** is a number that appeared only once in the `MyNumbers` table.

Find the largest **single number**. If there is no **single number**, report `null`.

The result format is in the following example.

```sql
SELECT MAX(num) AS num
FROM (
    SELECT num
    FROM MyNumbers
    GROUP BY num
    HAVING COUNT(*) = 1
) AS unique_numbers;
```

Customers Who Bought All Products



1045. Customers Who Bought All Products

Solved

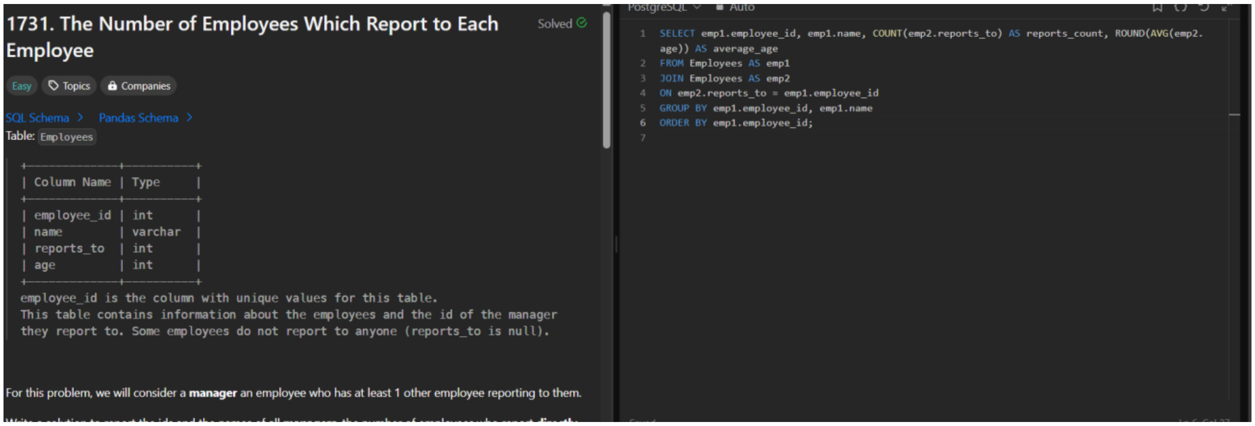
Table: `Customer`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| customer_id | int     |
| product_key | int     |
+-------------+---------+
This table may contain duplicates rows.
customer_id is not NULL.
product_key is a foreign key (reference column) to Product table.
```

Table: `Product`

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

1731. The Number of Employees Which Report to Each Employee



1731. The Number of Employees Which Report to Each Employee

Solved

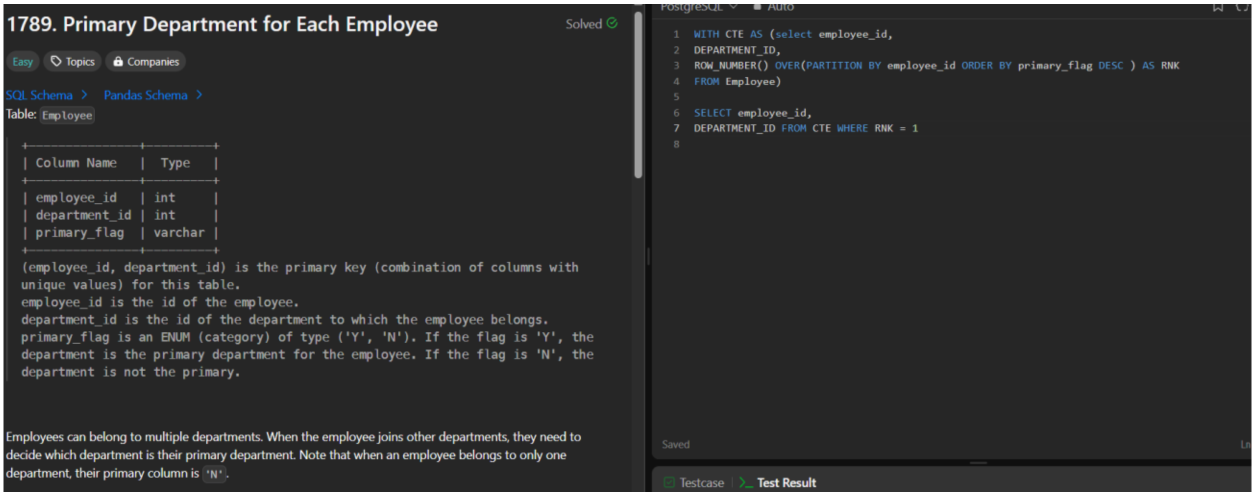
Table: `Employees`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| employee_id | int      |
| name        | varchar  |
| reports_to  | int      |
| age         | int      |
+-------------+----------+
employee_id is the column with unique values for this table.
This table contains information about the employees and the id of the manager
they report to. Some employees do not report to anyone (reports_to is null).
```

For this problem, we will consider a **manager** an employee who has at least 1 other employee reporting to them.

```sql
SELECT emp1.employee_id, emp1.name, COUNT(emp2.reports_to) AS reports_count, ROUND(AVG(emp2.
age)) AS average_age
FROM Employees AS emp1
JOIN Employees AS emp2
ON emp2.reports_to = emp1.employee_id
GROUP BY emp1.employee_id, emp1.name
ORDER BY emp1.employee_id;
```

Primary Department for Each Employee



1789. Primary Department for Each Employee

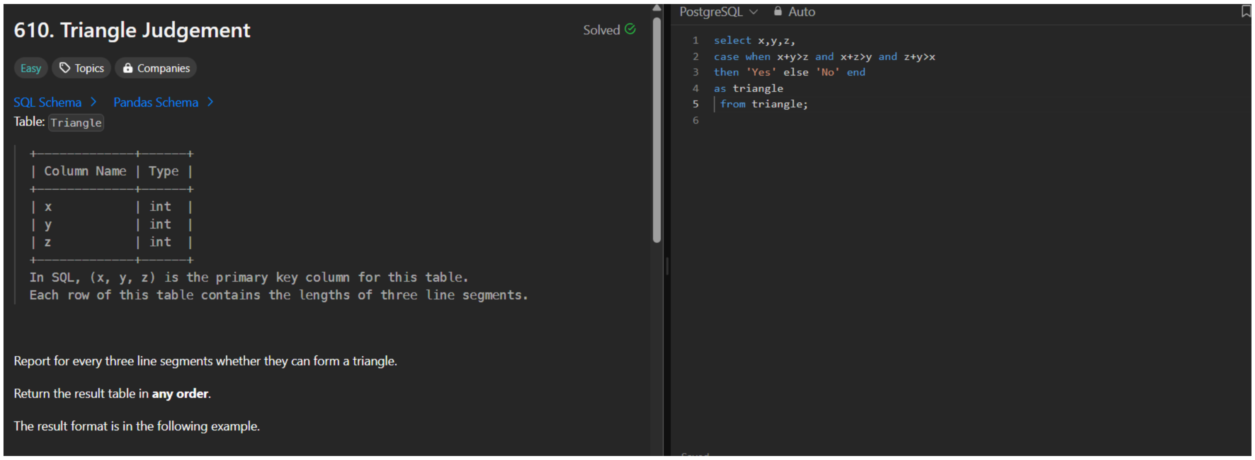
Table: `Employee`

```
+---------------+---------+
| Column Name   |  Type   |
+---------------+---------+
| employee_id   | int     |
| department_id | int     |
| primary_flag  | varchar |
+---------------+---------+
(employee_id, department_id) is the primary key (combination of columns with
unique values) for this table.
employee_id is the id of the employee.
department_id is the id of the department to which the employee belongs.
primary_flag is an ENUM (category) of type ('Y', 'N'). If the flag is 'Y', the
department is the primary department for the employee. If the flag is 'N', the
department is not the primary.
```

Employees can belong to multiple departments. When the employee joins other departments, they need to decide which department is their primary department. Note that when an employee belongs to only one department, their primary column is `'N'`.

```
WITH CTE AS (select employee_id,
DEPARTMENT_ID,
ROW_NUMBER() OVER(PARTITION BY employee_id ORDER BY primary_flag DESC ) AS RNK
FROM Employee)

SELECT employee_id,
DEPARTMENT_ID FROM CTE WHERE RNK = 1

```

Triangle Judgement



610. Triangle Judgement

Table: `Triangle`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| x           | int  |
| y           | int  |
| z           | int  |
+-------------+------+
In SQL, (x, y, z) is the primary key column for this table.
Each row of this table contains the lengths of three line segments.
```

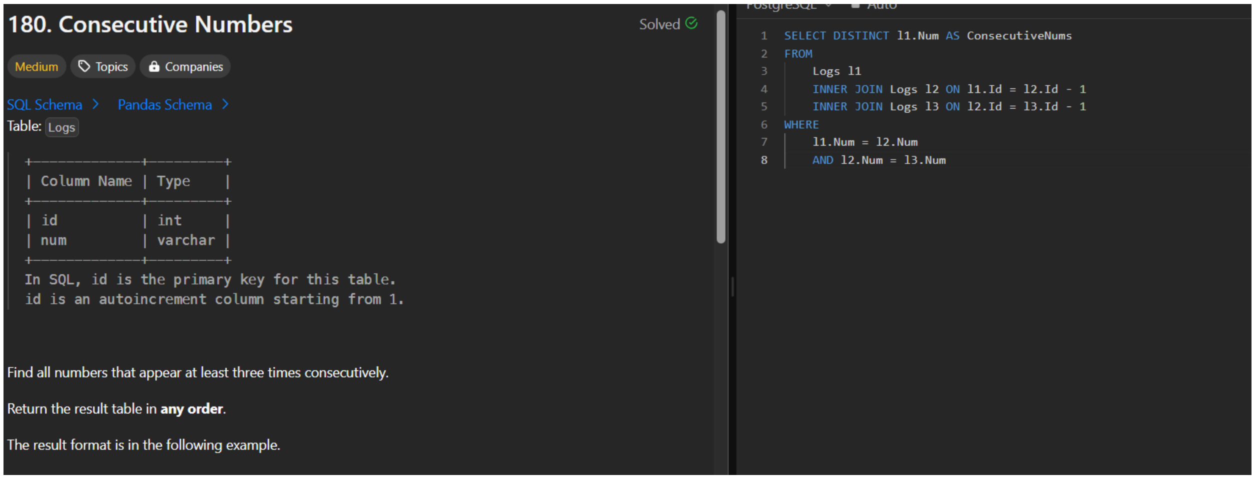
Report for every three line segments whether they can form a triangle.

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

The result format is in the following example.

```
select x,y,z,
case when x+y>z and x+z>y and z+y>x
then 'Yes' else 'No' end
as triangle
 from triangle;

```

Consecutive Numbers



180. Consecutive Numbers

Table: `Logs`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| num         | varchar |
+-------------+---------+
In SQL, id is the primary key for this table.
id is an autoincrement column starting from 1.
```

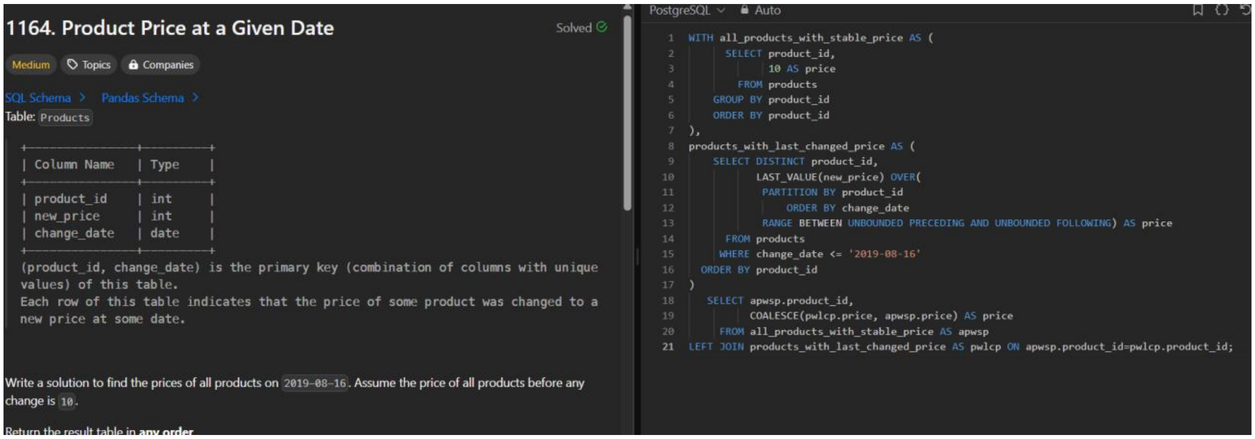
Find all numbers that appear at least three times consecutively.

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

The result format is in the following example.

```
SELECT DISTINCT l1.Num AS ConsecutiveNums
FROM
    Logs l1
    INNER JOIN Logs l2 ON l1.Id = l2.Id - 1
    INNER JOIN Logs l3 ON l2.Id = l3.Id - 1
WHERE
    l1.Num = l2.Num
    AND l2.Num = l3.Num
```

Product Price at a Given Date



1164. Product Price at a Given Date

Table: `Products`

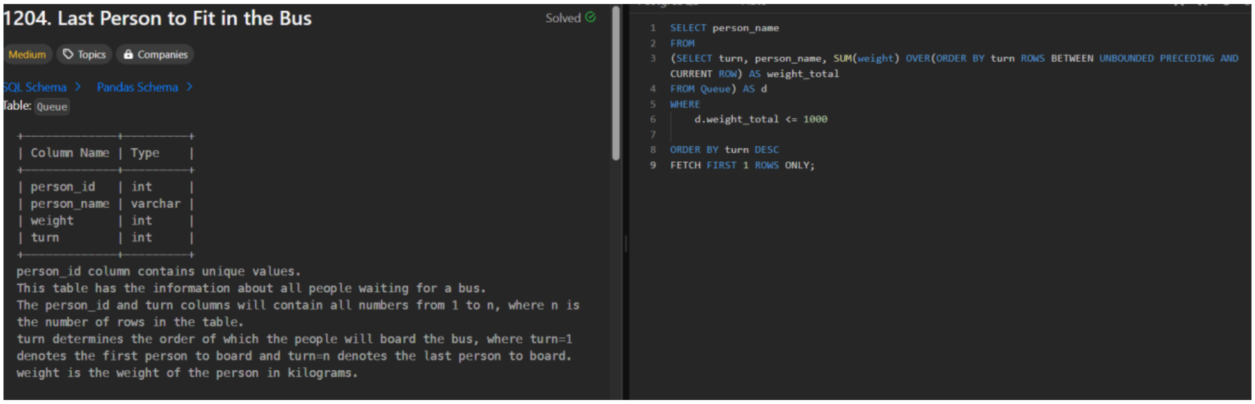
```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| product_id    | int     |
| new_price     | int     |
| change_date   | date    |
+---------------+---------+
(product_id, change_date) is the primary key (combination of columns with unique
values) of this table.
Each row of this table indicates that the price of some product was changed to a
new price at some date.
```

Write a solution to find the prices of all products on `2019-08-16`. Assume the price of all products before any change is `10`.

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

```sql
WITH all_products_with_stable_price AS (
      SELECT product_id,
             10 AS price
        FROM products
    GROUP BY product_id
    ORDER BY product_id
),
products_with_last_changed_price AS (
    SELECT DISTINCT product_id,
           LAST_VALUE(new_price) OVER(
            PARTITION BY product_id
                ORDER BY change_date
            RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS price
      FROM products
     WHERE change_date <= '2019-08-16'
  ORDER BY product_id
)
   SELECT apwsp.product_id,
          COALESCE(pwlcp.price, apwsp.price) AS price
     FROM all_products_with_stable_price AS apwsp
LEFT JOIN products_with_last_changed_price AS pwlcp ON apwsp.product_id=pwlcp.product_id;
```

Last Person to Fit in the Bus



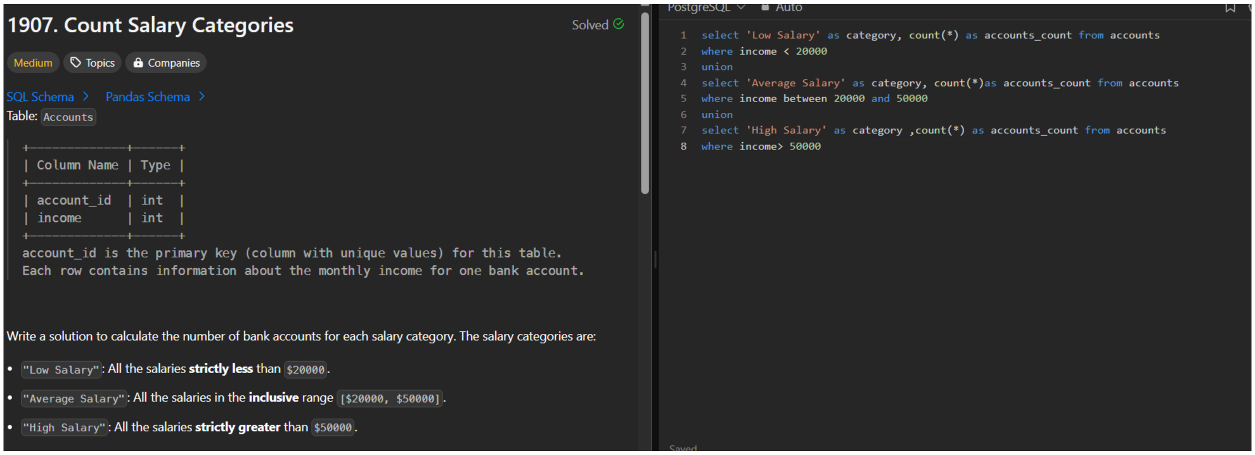
1204. Last Person to Fit in the Bus

Table: `Queue`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| person_id   | int     |
| person_name | varchar |
| weight      | int     |
| turn        | int     |
+-------------+---------+
person_id column contains unique values.
This table has the information about all people waiting for a bus.
The person_id and turn columns will contain all numbers from 1 to n, where n is
the number of rows in the table.
turn determines the order of which the people will board the bus, where turn=1
denotes the first person to board and turn=n denotes the last person to board.
weight is the weight of the person in kilograms.
```

```sql
SELECT person_name
FROM
(SELECT turn, person_name, SUM(weight) OVER(ORDER BY turn ROWS BETWEEN UNBOUNDED PRECEDING AND
CURRENT ROW) AS weight_total
FROM Queue) AS d
WHERE
    d.weight_total <= 1000

ORDER BY turn DESC
FETCH FIRST 1 ROWS ONLY;
```

Count Salary Categories



1907. Count Salary Categories

Table: `Accounts`

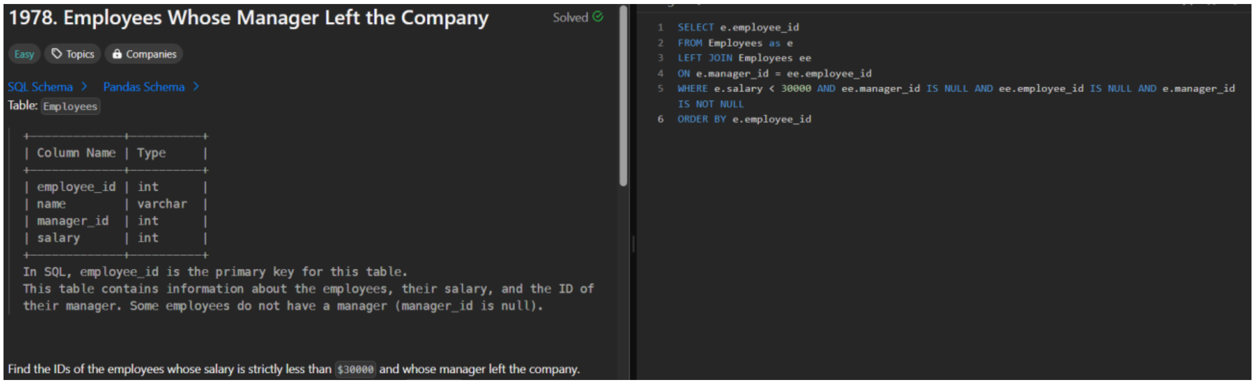
```
+-------------+------+
| Column Name | Type |
+-------------+------+
| account_id  | int  |
| income      | int  |
+-------------+------+
account_id is the primary key (column with unique values) for this table.
Each row contains information about the monthly income for one bank account.
```

Write a solution to calculate the number of bank accounts for each salary category. The salary categories are:

- `"Low Salary"`: All the salaries **strictly less** than `$20000`.
- `"Average Salary"`: All the salaries in the **inclusive** range `[$20000, $50000]`.
- `"High Salary"`: All the salaries **strictly greater** than `$50000`.

```sql
select 'Low Salary' as category, count(*) as accounts_count from accounts
where income < 20000
union
select 'Average Salary' as category, count(*)as accounts_count from accounts
where income between 20000 and 50000
union
select 'High Salary' as category ,count(*) as accounts_count from accounts
where income> 50000
```

employees Whose Manager Left the Company



1978. Employees Whose Manager Left the Company

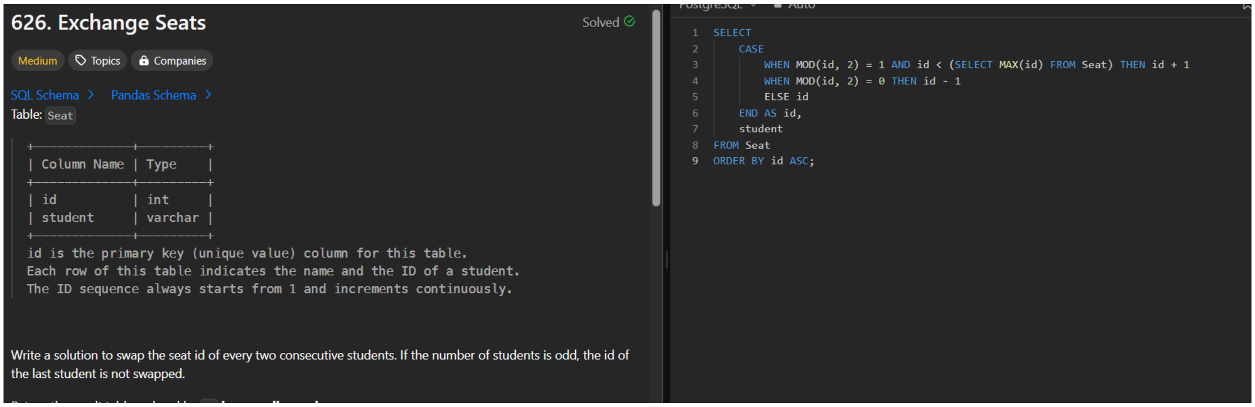
Table: `Employees`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| employee_id | int      |
| name        | varchar  |
| manager_id  | int      |
| salary      | int      |
+-------------+----------+
In SQL, employee_id is the primary key for this table.
This table contains information about the employees, their salary, and the ID of
their manager. Some employees do not have a manager (manager_id is null).
```

Find the IDs of the employees whose salary is strictly less than `$30000` and whose manager left the company.

```sql
SELECT e.employee_id
FROM Employees as e
LEFT JOIN Employees ee
ON e.manager_id = ee.employee_id
WHERE e.salary < 30000 AND ee.manager_id IS NULL AND ee.employee_id IS NULL AND e.manager_id
IS NOT NULL
ORDER BY e.employee_id
```

Exchange Seats



626. Exchange Seats

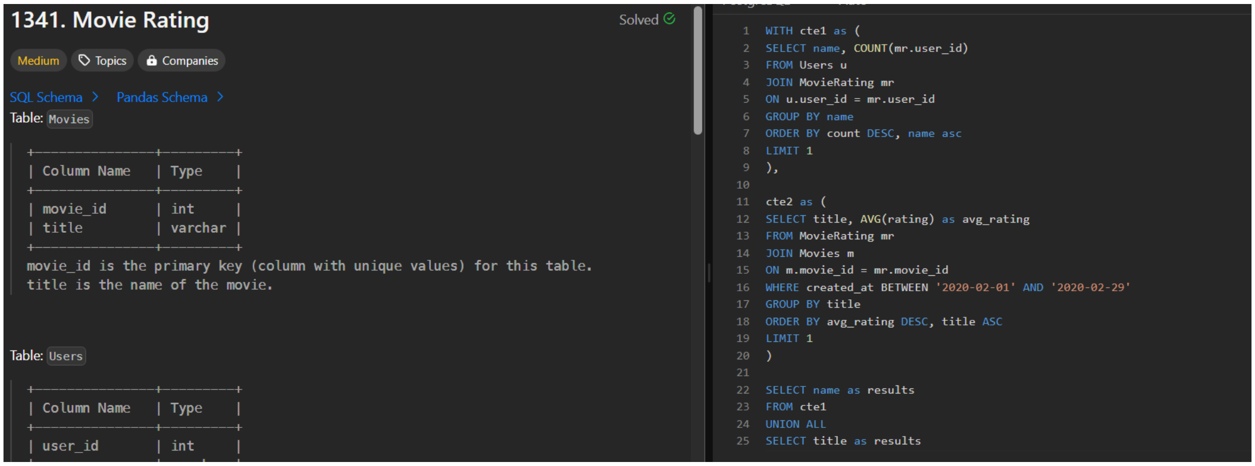
Table: `Seat`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| student     | varchar |
+-------------+---------+
id is the primary key (unique value) column for this table.
Each row of this table indicates the name and the ID of a student.
The ID sequence always starts from 1 and increments continuously.
```

Write a solution to swap the seat id of every two consecutive students. If the number of students is odd, the id of the last student is not swapped.

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

Movie Rating



1341. Movie Rating

Table: `Movies`

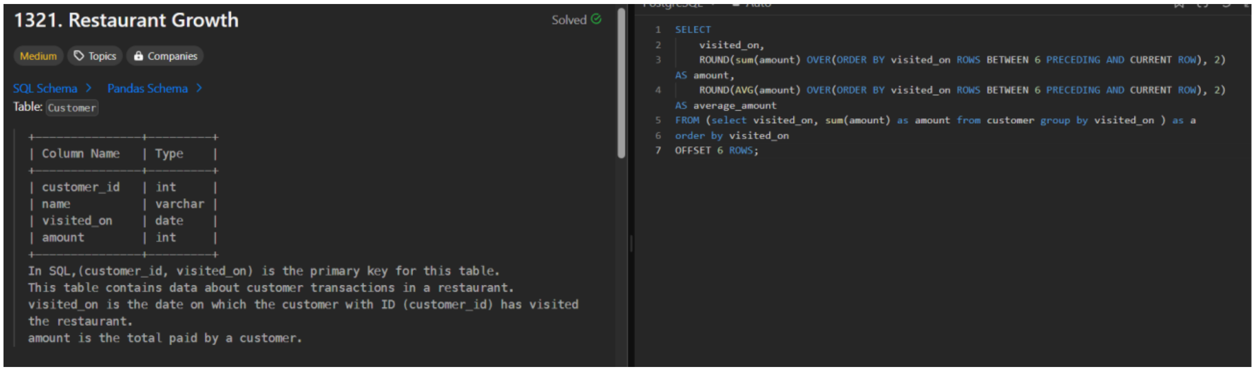
```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| movie_id      | int     |
| title         | varchar |
+---------------+---------+
movie_id is the primary key (column with unique values) for this table.
title is the name of the movie.
```

Table: `Users`

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

```sql
WITH cte1 as (
SELECT name, COUNT(mr.user_id)
FROM Users u
JOIN MovieRating mr
ON u.user_id = mr.user_id
GROUP BY name
ORDER BY count DESC, name asc
LIMIT 1
),

cte2 as (
SELECT title, AVG(rating) as avg_rating
FROM MovieRating mr
JOIN Movies m
ON m.movie_id = mr.movie_id
WHERE created_at BETWEEN '2020-02-01' AND '2020-02-29'
GROUP BY title
ORDER BY avg_rating DESC, title ASC
LIMIT 1
)

SELECT name as results
FROM cte1
UNION ALL
SELECT title as results
```

Restaurant Growth



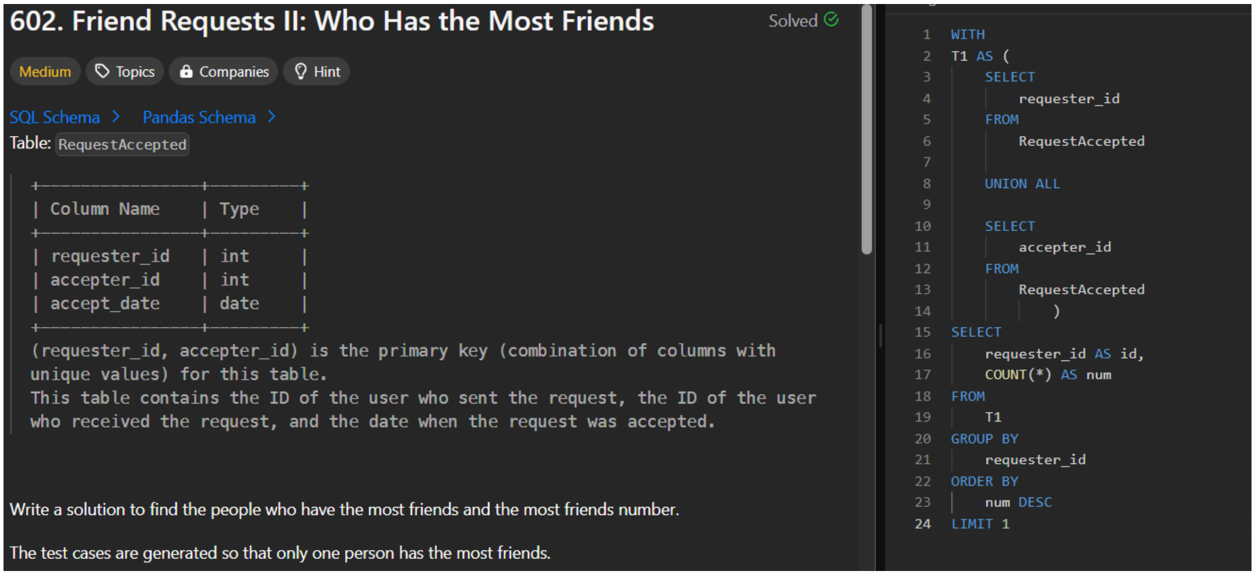
1321. Restaurant Growth

Table: `Customer`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| customer_id   | int     |
| name          | varchar |
| visited_on    | date    |
| amount        | int     |
+---------------+---------+
In SQL,(customer_id, visited_on) is the primary key for this table.
This table contains data about customer transactions in a restaurant.
visited_on is the date on which the customer with ID (customer_id) has visited
the restaurant.
amount is the total paid by a customer.
```

```sql
SELECT
    visited_on,
    ROUND(sum(amount) OVER(ORDER BY visited_on ROWS BETWEEN 6 PRECEDING AND CURRENT ROW), 2)
AS amount,
    ROUND(AVG(amount) OVER(ORDER BY visited_on ROWS BETWEEN 6 PRECEDING AND CURRENT ROW), 2)
AS average_amount
FROM (select visited_on, sum(amount) as amount from customer group by visited_on ) as a
order by visited_on
OFFSET 6 ROWS;
```

Friend Requests II: Who Has the Most Friends



602. Friend Requests II: Who Has the Most Friends

Table: `RequestAccepted`

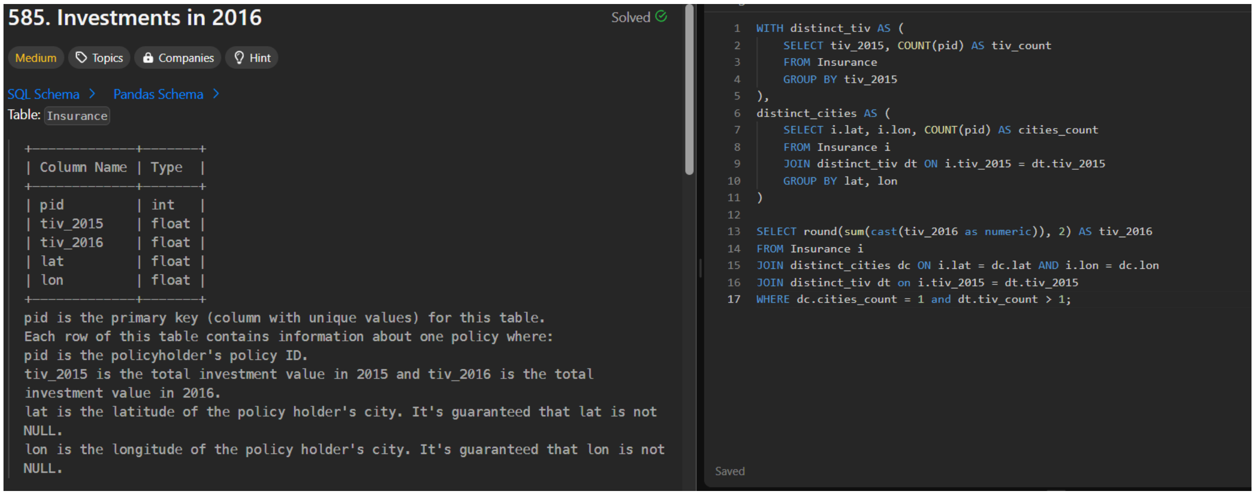
```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| requester_id   | int     |
| accepter_id    | int     |
| accept_date    | date    |
+----------------+---------+
(requester_id, accepter_id) is the primary key (combination of columns with
unique values) for this table.
This table contains the ID of the user who sent the request, the ID of the user
who received the request, and the date when the request was accepted.
```

Write a solution to find the people who have the most friends and the most friends number.

The test cases are generated so that only one person has the most friends.

```sql
WITH
T1 AS (
    SELECT
        requester_id
    FROM
        RequestAccepted

    UNION ALL

    SELECT
        accepter_id
    FROM
        RequestAccepted
           )
SELECT
    requester_id AS id,
    COUNT(*) AS num
FROM
    T1
GROUP BY
    requester_id
ORDER BY
    num DESC
LIMIT 1
```

Investments in 2016



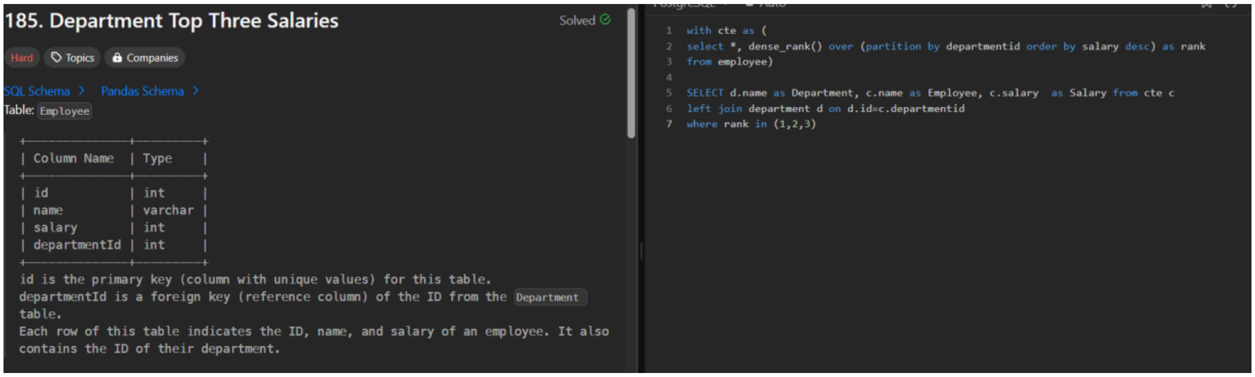
585. Investments in 2016

Table: `Insurance`

```
+-------------+-------+
| Column Name | Type  |
+-------------+-------+
| pid         | int   |
| tiv_2015    | float |
| tiv_2016    | float |
| lat         | float |
| lon         | float |
+-------------+-------+
pid is the primary key (column with unique values) for this table.
Each row of this table contains information about one policy where:
pid is the policyholder's policy ID.
tiv_2015 is the total investment value in 2015 and tiv_2016 is the total
investment value in 2016.
lat is the latitude of the policy holder's city. It's guaranteed that lat is not
NULL.
lon is the longitude of the policy holder's city. It's guaranteed that lon is not
NULL.
```

```sql
WITH distinct_tiv AS (
    SELECT tiv_2015, COUNT(pid) AS tiv_count
    FROM Insurance
    GROUP BY tiv_2015
),
distinct_cities AS (
    SELECT i.lat, i.lon, COUNT(pid) AS cities_count
    FROM Insurance i
    JOIN distinct_tiv dt ON i.tiv_2015 = dt.tiv_2015
    GROUP BY lat, lon
)

SELECT round(sum(cast(tiv_2016 as numeric)), 2) AS tiv_2016
FROM Insurance i
JOIN distinct_cities dc ON i.lat = dc.lat AND i.lon = dc.lon
JOIN distinct_tiv dt on i.tiv_2015 = dt.tiv_2015
WHERE dc.cities_count = 1 and dt.tiv_count > 1;
```

Department Top Three Salaries



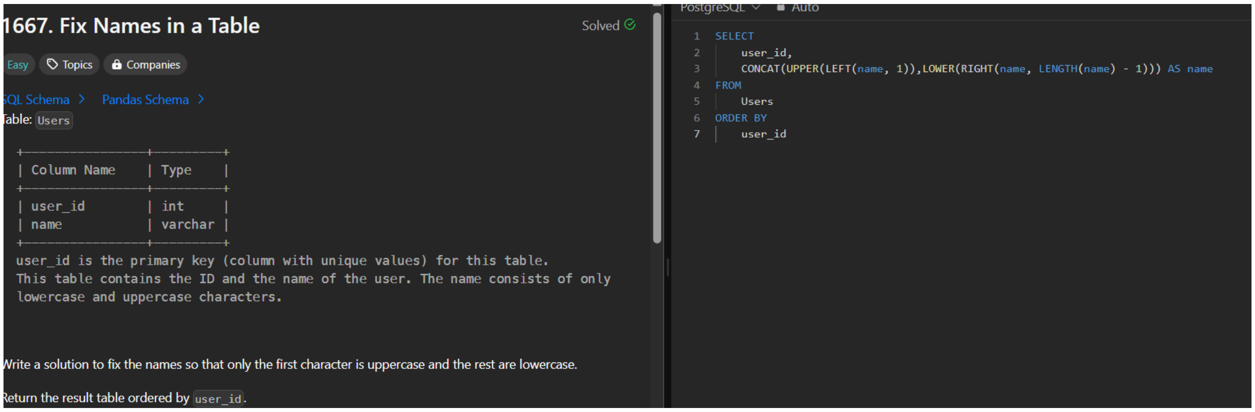
185. Department Top Three Salaries

Table: `Employee`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| id           | int     |
| name         | varchar |
| salary       | int     |
| departmentId | int     |
+--------------+---------+
id is the primary key (column with unique values) for this table.
departmentId is a foreign key (reference column) of the ID from the Department
table.
Each row of this table indicates the ID, name, and salary of an employee. It also
contains the ID of their department.
```

```sql
with cte as (
select *, dense_rank() over (partition by departmentid order by salary desc) as rank
from employee)

SELECT d.name as Department, c.name as Employee, c.salary  as Salary from cte c
left join department d on d.id=c.departmentid
where rank in (1,2,3)
```

Fix Names in a Table



1667. Fix Names in a Table

Table: `Users`

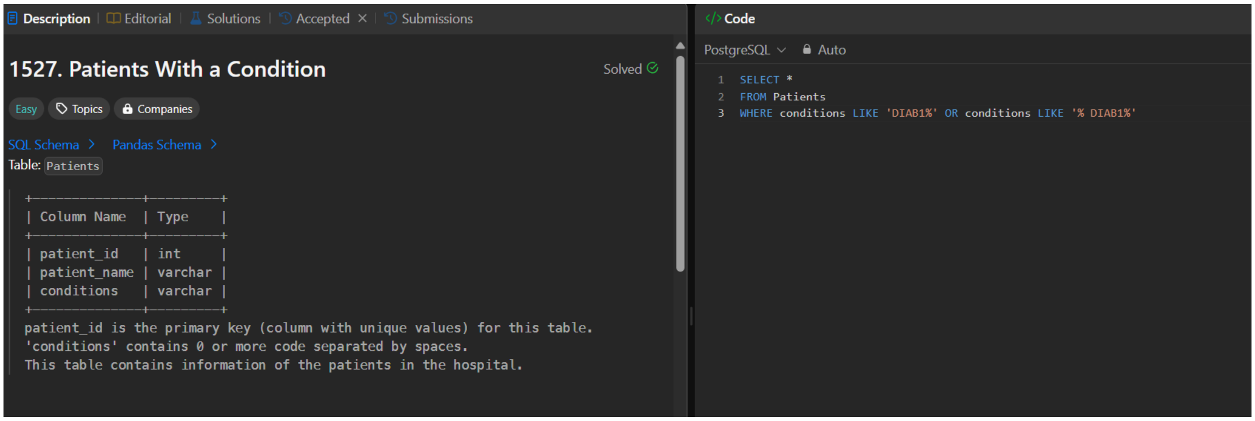
```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| user_id        | int     |
| name           | varchar |
+----------------+---------+
user_id is the primary key (column with unique values) for this table.
This table contains the ID and the name of the user. The name consists of only
lowercase and uppercase characters.
```

Write a solution to fix the names so that only the first character is uppercase and the rest are lowercase.

Return the result table ordered by `user_id`.

```sql
SELECT
    user_id,
    CONCAT(UPPER(LEFT(name, 1)),LOWER(RIGHT(name, LENGTH(name) - 1))) AS name
FROM
    Users
ORDER BY
    user_id
```

Patients with a condition



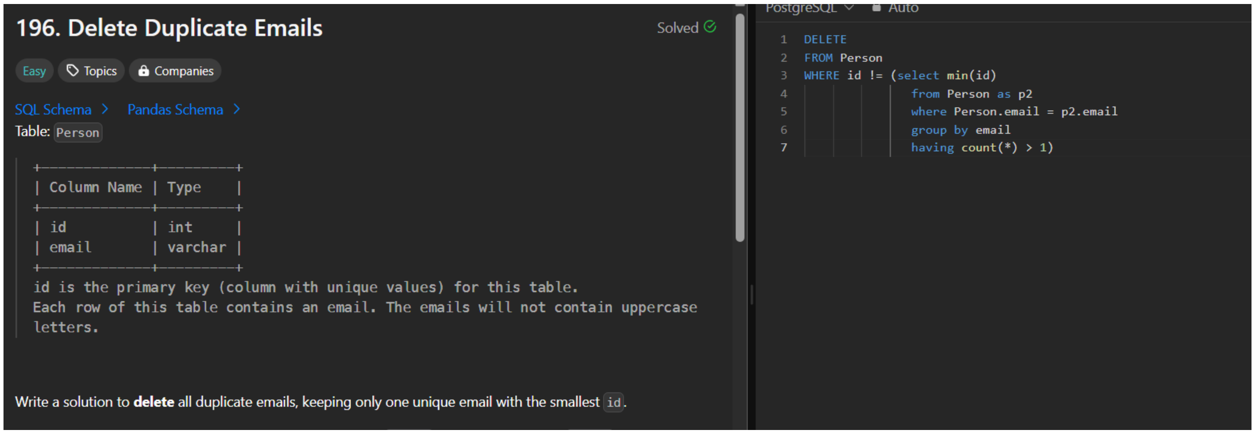
1527. Patients With a Condition

Table: `Patients`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| patient_id   | int     |
| patient_name | varchar |
| conditions   | varchar |
+--------------+---------+
patient_id is the primary key (column with unique values) for this table.
'conditions' contains 0 or more code separated by spaces.
This table contains information of the patients in the hospital.
```

```sql
SELECT *
FROM Patients
WHERE conditions LIKE 'DIAB1%' OR conditions LIKE '% DIAB1%'
```

Delete Duplicate Emails



196. Delete Duplicate Emails

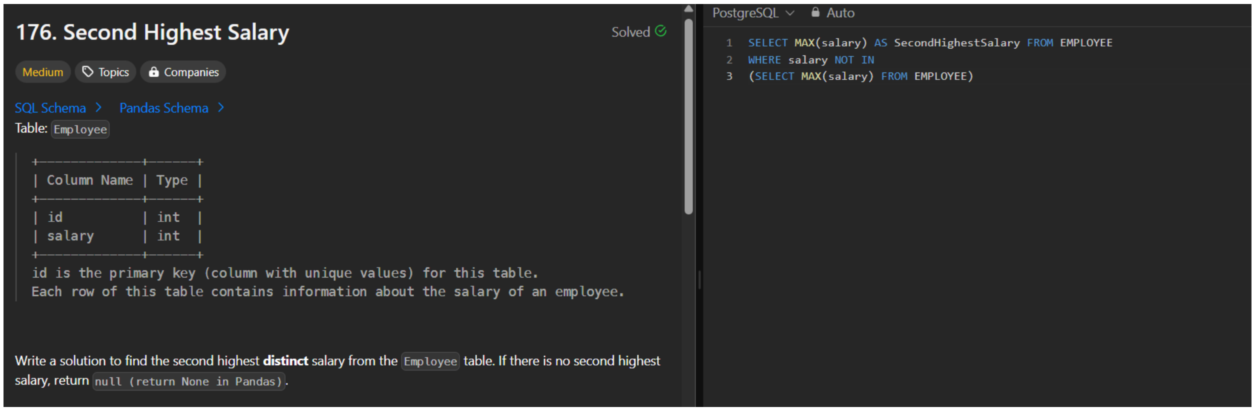
Table: `Person`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| email       | varchar |
+-------------+---------+
id is the primary key (column with unique values) for this table.
Each row of this table contains an email. The emails will not contain uppercase
letters.
```

Write a solution to **delete** all duplicate emails, keeping only one unique email with the smallest `id`.

```
DELETE
FROM Person
WHERE id != (select min(id)
            from Person as p2
            where Person.email = p2.email
            group by email
            having count(*) > 1)
```

Second Highest Salary



176. Second Highest Salary

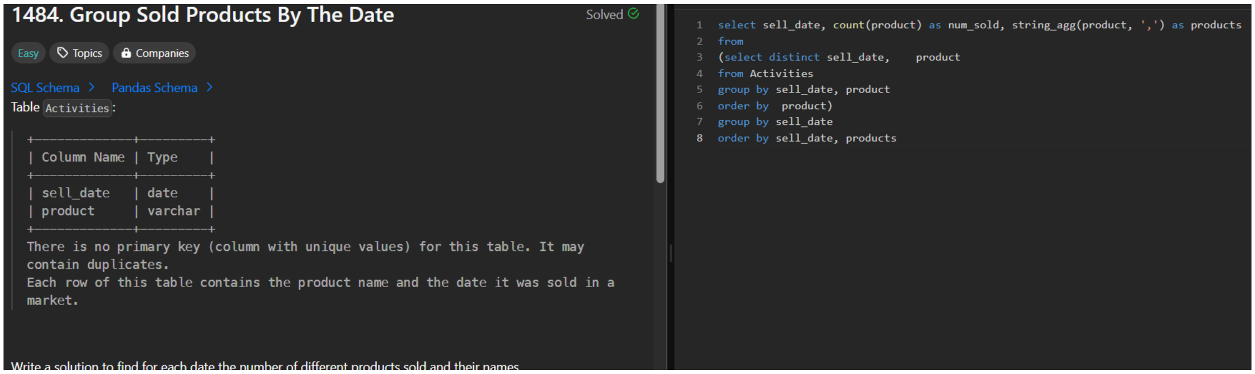
Table: `Employee`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| id          | int  |
| salary      | int  |
+-------------+------+
id is the primary key (column with unique values) for this table.
Each row of this table contains information about the salary of an employee.
```

Write a solution to find the second highest **distinct** salary from the `Employee` table. If there is no second highest salary, return `null (return None in Pandas)`.

```
SELECT MAX(salary) AS SecondHighestSalary FROM EMPLOYEE
WHERE salary NOT IN
(SELECT MAX(salary) FROM EMPLOYEE)
```

Group Sold Products By The Date



1484. Group Sold Products By The Date

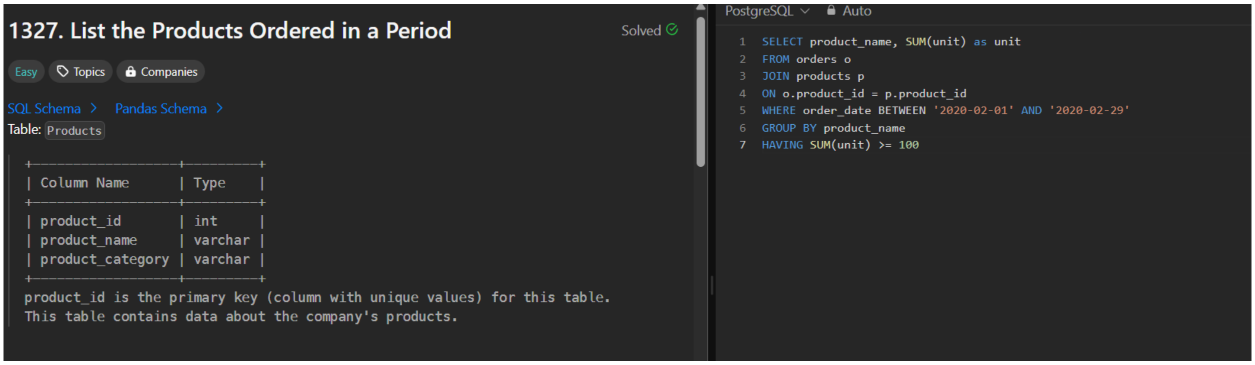
Table `Activities`:

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| sell_date   | date    |
| product     | varchar |
+-------------+---------+
There is no primary key (column with unique values) for this table. It may
contain duplicates.
Each row of this table contains the product name and the date it was sold in a
market.
```

Write a solution to find for each date the number of different products sold and their names.

```
select sell_date, count(product) as num_sold, string_agg(product, ',') as products
from
(select distinct sell_date,    product
from Activities
group by sell_date, product
order by  product)
group by sell_date
order by sell_date, products
```

List the Products Ordered in a Period



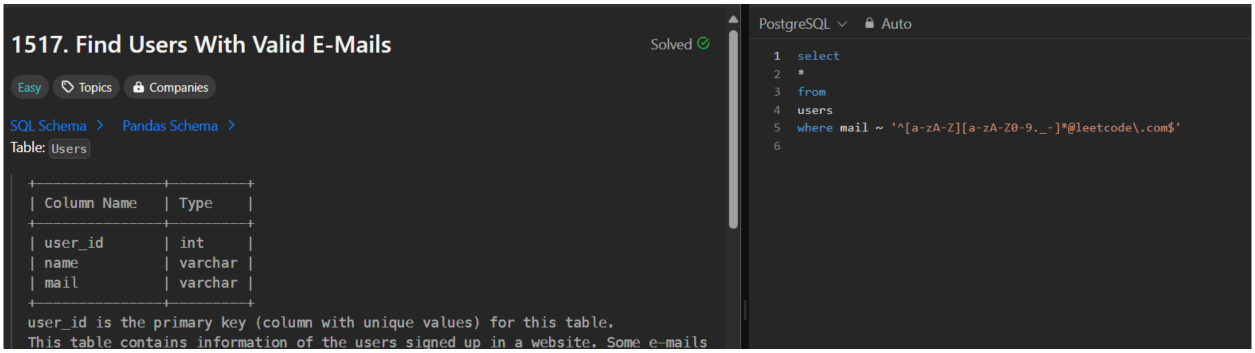
1327. List the Products Ordered in a Period

Table: Products

```
+------------------+---------+
| Column Name      | Type    |
+------------------+---------+
| product_id       | int     |
| product_name     | varchar |
| product_category | varchar |
+------------------+---------+
product_id is the primary key (column with unique values) for this table.
This table contains data about the company's products.
```

```sql
SELECT product_name, SUM(unit) as unit
FROM orders o
JOIN products p
ON o.product_id = p.product_id
WHERE order_date BETWEEN '2020-02-01' AND '2020-02-29'
GROUP BY product_name
HAVING SUM(unit) >= 100
```

Find Users With Valid E-Mails

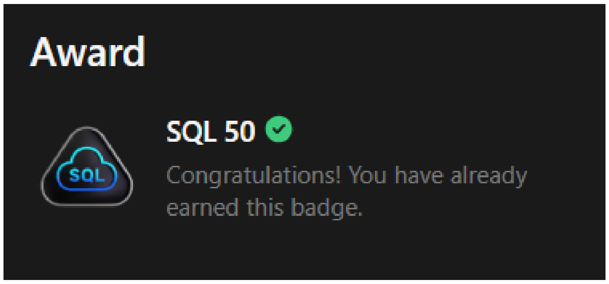


1517. Find Users With Valid E-Mails

Table: Users

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| user_id       | int     |
| name          | varchar |
| mail          | varchar |
+---------------+---------+
user_id is the primary key (column with unique values) for this table.
This table contains information of the users signed up in a website. Some e-mails
```

```sql
select
*
from
users
where mail ~ '^[a-zA-Z][a-zA-Z0-9._-]*@leetcode\.com$'

```



Award

SQL 50 ✓

Congratulations! You have already earned this badge.