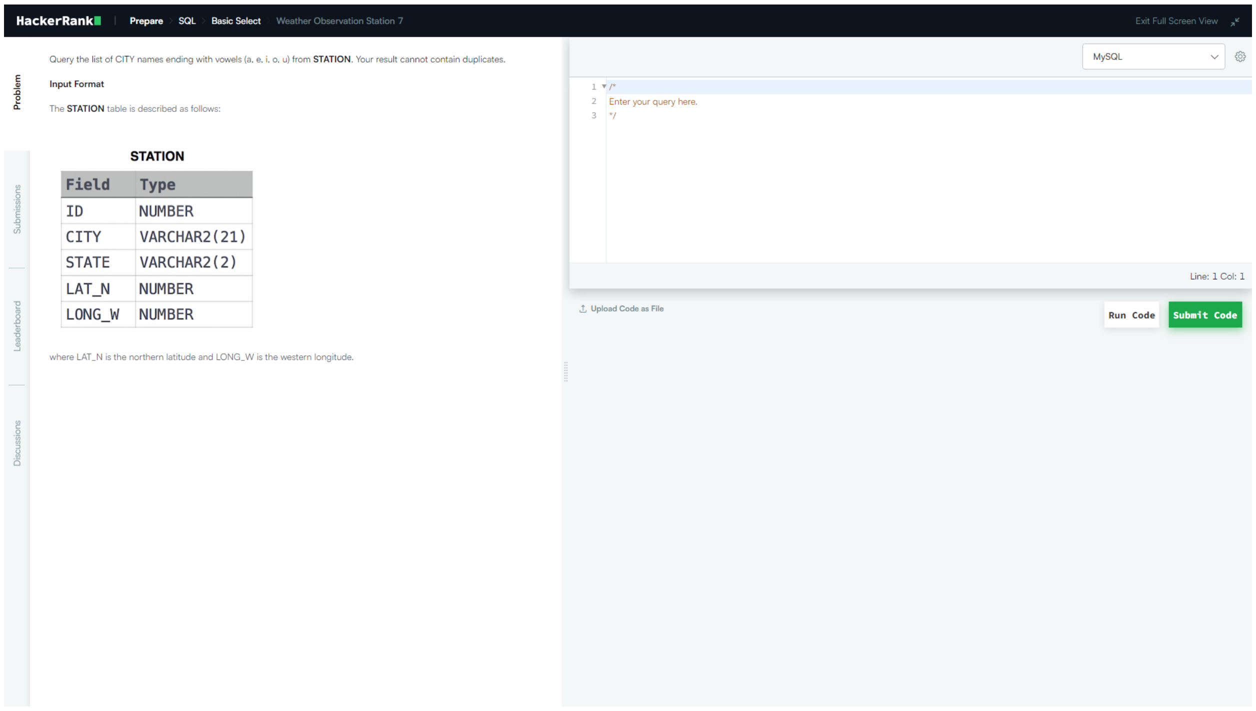
SQL queries

Q1



Query the list of CITY names ending with vowels (a, e, i, o, u) from **STATION**. Your result cannot contain duplicates.

Input Format

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

STATION

Field	Type
ID	NUMBER
CITY	VARCHAR2(21)
STATE	VARCHAR2(2)
LAT_N	NUMBER
LONG_W	NUMBER

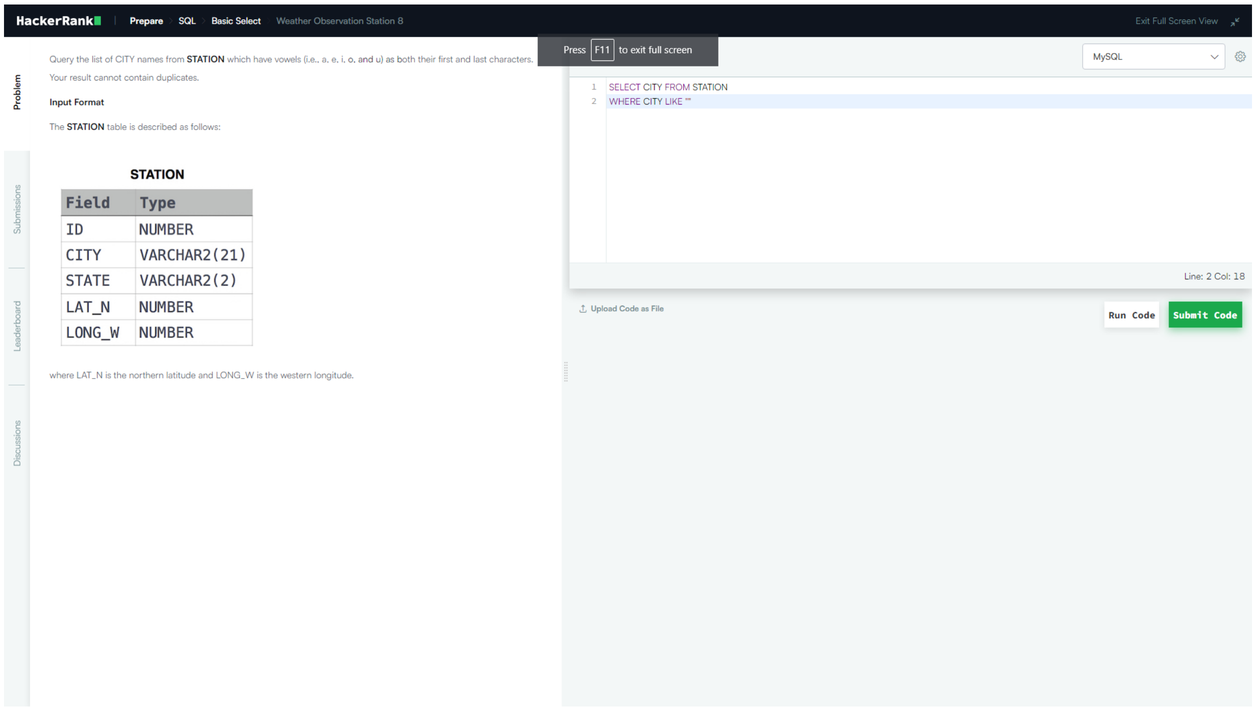
where LAT_N is the northern latitude and LONG_W is the western longitude.

```
/*
Enter your query here.
*/
```

answer:

SELECT DISTINCT CITY
FROM STATION
WHERE CITY LIKE "%a" OR CITY LIKE "%e" OR CITY LIKE "%i" OR CITY LIKE "%o" OR CITY LIKE "%u";

Q2



Query the list of CITY names from **STATION** which have vowels (i.e., a, e, i, o, and u) as both their first and last characters. Your result cannot contain duplicates.

Input Format

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

STATION

Field	Type
ID	NUMBER
CITY	VARCHAR2(21)
STATE	VARCHAR2(2)
LAT_N	NUMBER
LONG_W	NUMBER

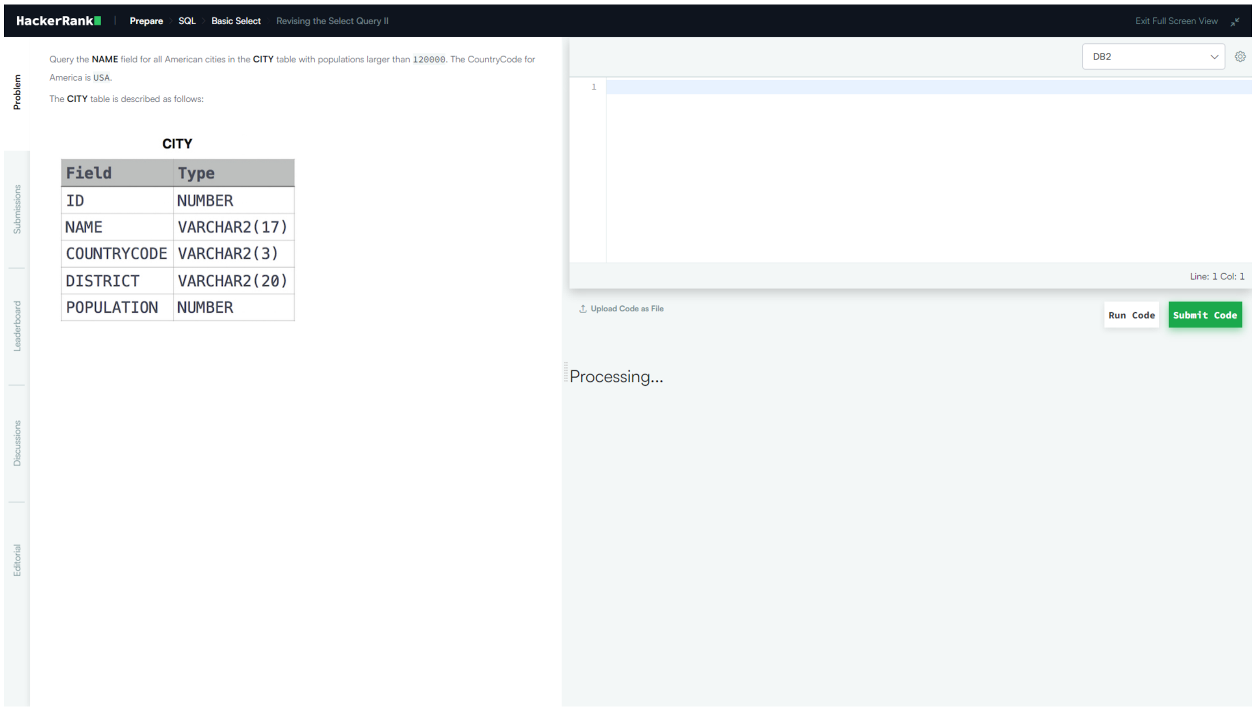
where LAT_N is the northern latitude and LONG_W is the western longitude.

```
SELECT CITY FROM STATION
WHERE CITY LIKE ""
```

answer:

```
SELECT DISTINCT city
FROM   station
WHERE  city RLIKE '^[aeiouAEIOU].*[aeiouAEIOU]$'
```

Q3:



Query the **NAME** field for all American cities in the **CITY** table with populations larger than `120000`. The CountryCode for America is `USA`.

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

CITY

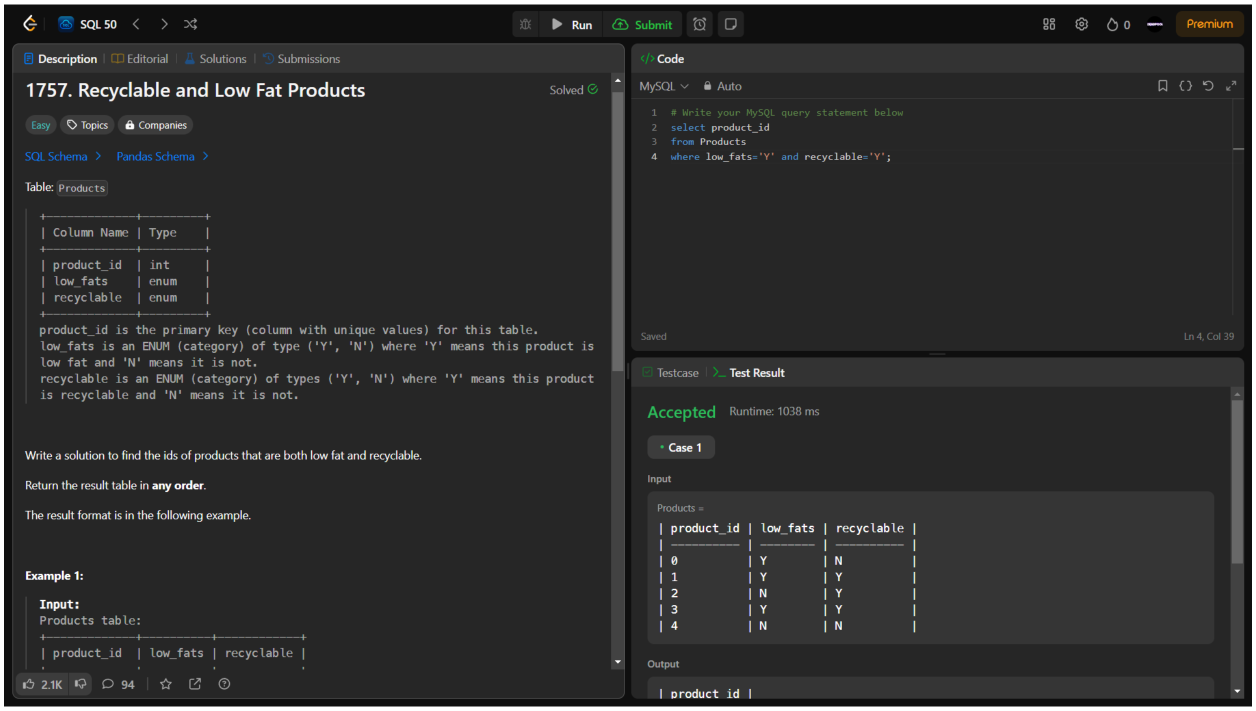
Field	Type
ID	NUMBER
NAME	VARCHAR2(17)
COUNTRYCODE	VARCHAR2(3)
DISTRICT	VARCHAR2(20)
POPULATION	NUMBER

answer:

SELECT NAME
FROM CITY
WHERE COUNTRYCODE= "USA" AND POPULATION >= 120000

Leetcode sql challenge:

Q1:



1757. Recyclable and Low Fat Products

Table: `Products`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| product_id  | int     |
| low_fats    | enum    |
| recyclable  | enum    |
+-------------+---------+
product_id is the primary key (column with unique values) for this table.
low_fats is an ENUM (category) of type ('Y', 'N') where 'Y' means this product is
low fat and 'N' means it is not.
recyclable is an ENUM (category) of types ('Y', 'N') where 'Y' means this product
is recyclable and 'N' means it is not.
```

Write a solution to find the ids of products that are both low fat and recyclable.

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

The result format is in the following example.

Example 1:

```
Input:
Products table:
+-------------+----------+------------+
| product_id  | low_fats | recyclable |
```

```sql
# Write your MySQL query statement below
select product_id
from Products
where low_fats='Y' and recyclable='Y';
```

Accepted Runtime: 1038 ms

Input

```
Products =
| product_id | low_fats | recyclable |
| ---------- | -------- | ---------- |
| 0          | Y        | N          |
| 1          | Y        | Y          |
| 2          | N        | Y          |
| 3          | Y        | Y          |
| 4          | N        | N          |
```

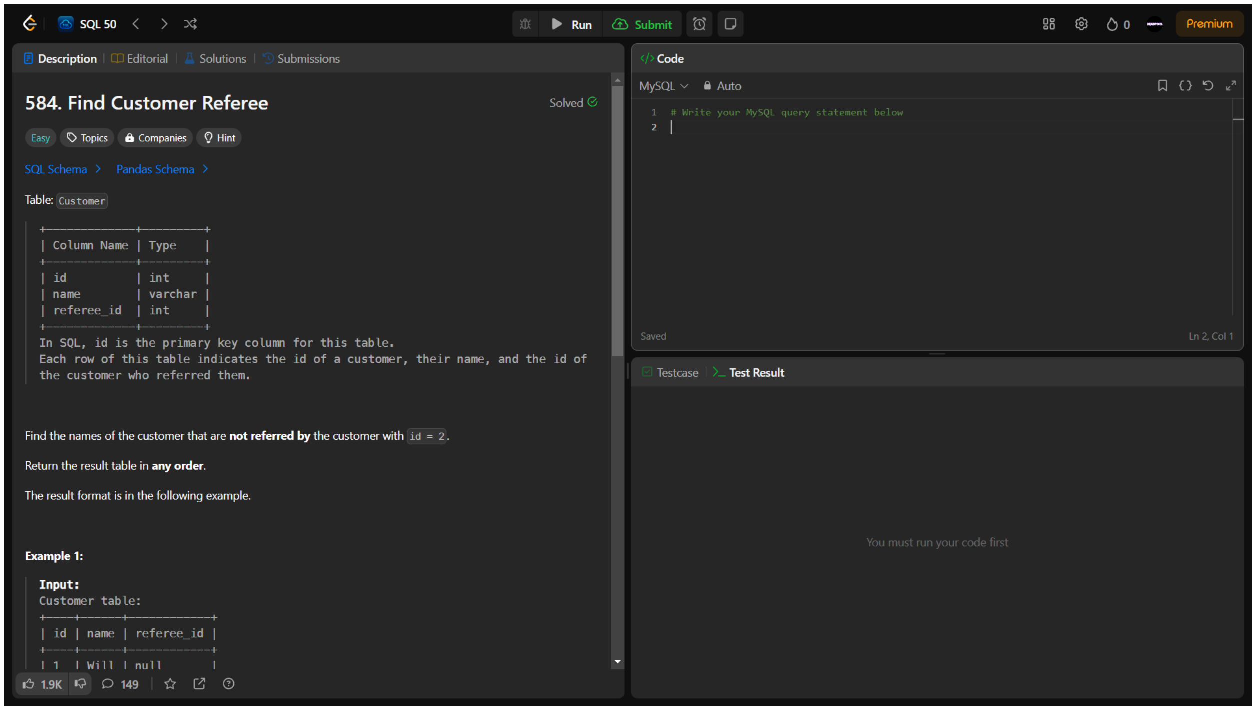
Output

```
| product_id |
```

answer:

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

q2:



584. Find Customer Referee

Table: `Customer`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| name        | varchar |
| referee_id  | int     |
+-------------+---------+
In SQL, id is the primary key column for this table.
Each row of this table indicates the id of a customer, their name, and the id of
the customer who referred them.
```

Find the names of the customer that are **not referred by** the customer with `id = 2`.

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

The result format is in the following example.

Example 1:

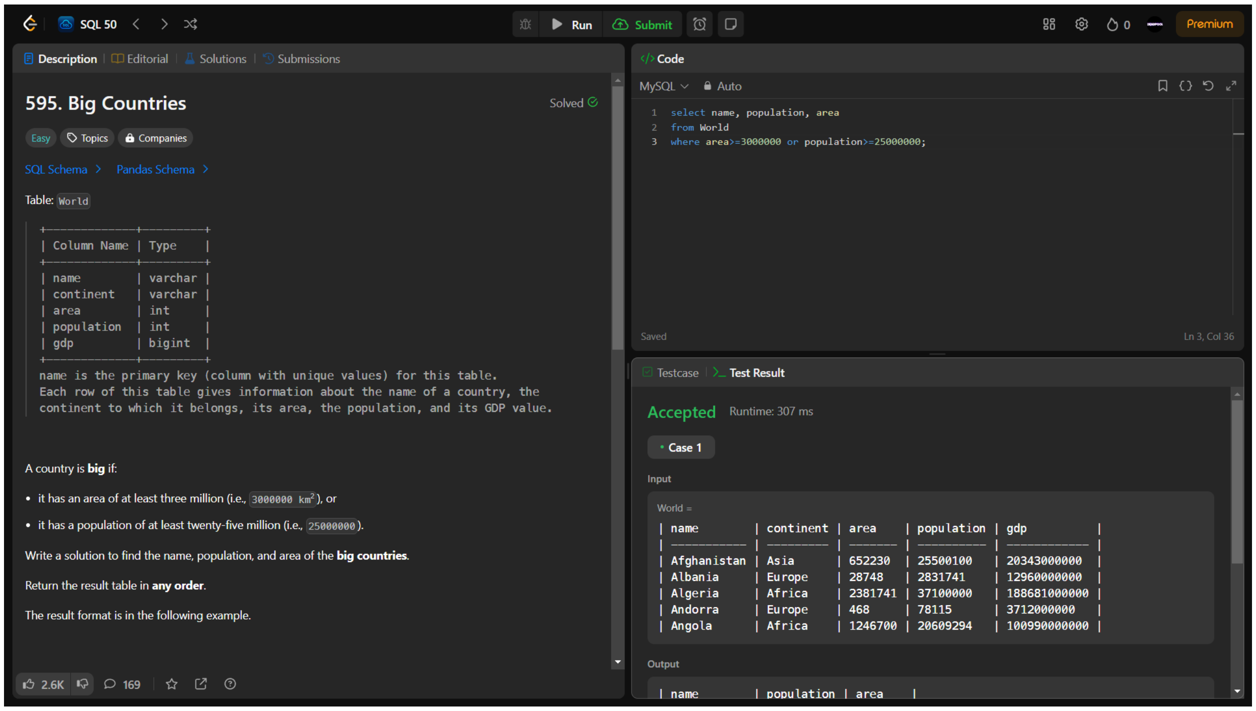
```
Input:
Customer table:
+----+------+------------+
| id | name | referee_id |
+----+------+------------+
| 1  | Will | null       |
```

```
# Write your MySQL query statement below
```

answer:

```
SELECT name
FROM Customer
WHERE referee_id IS NULL OR referee_id <> 2;
```

q3:



595. Big Countries

Solved

Table: `World`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| name        | varchar |
| continent   | varchar |
| area        | int     |
| population  | int     |
| gdp         | bigint  |
+-------------+---------+
name is the primary key (column with unique values) for this table.
Each row of this table gives information about the name of a country, the
continent to which it belongs, its area, the population, and its GDP value.
```

A country is **big** if:

- it has an area of at least three million (i.e., `3000000 km²`), or
- it has a population of at least twenty-five million (i.e., `25000000`).

Write a solution to find the name, population, and area of the **big countries**.

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

The result format is in the following example.

```
select name, population, area
from World
where area>=3000000 or population>=25000000;
```

Saved Ln 3, Col 36

Accepted Runtime: 307 ms

Case 1

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 |
```

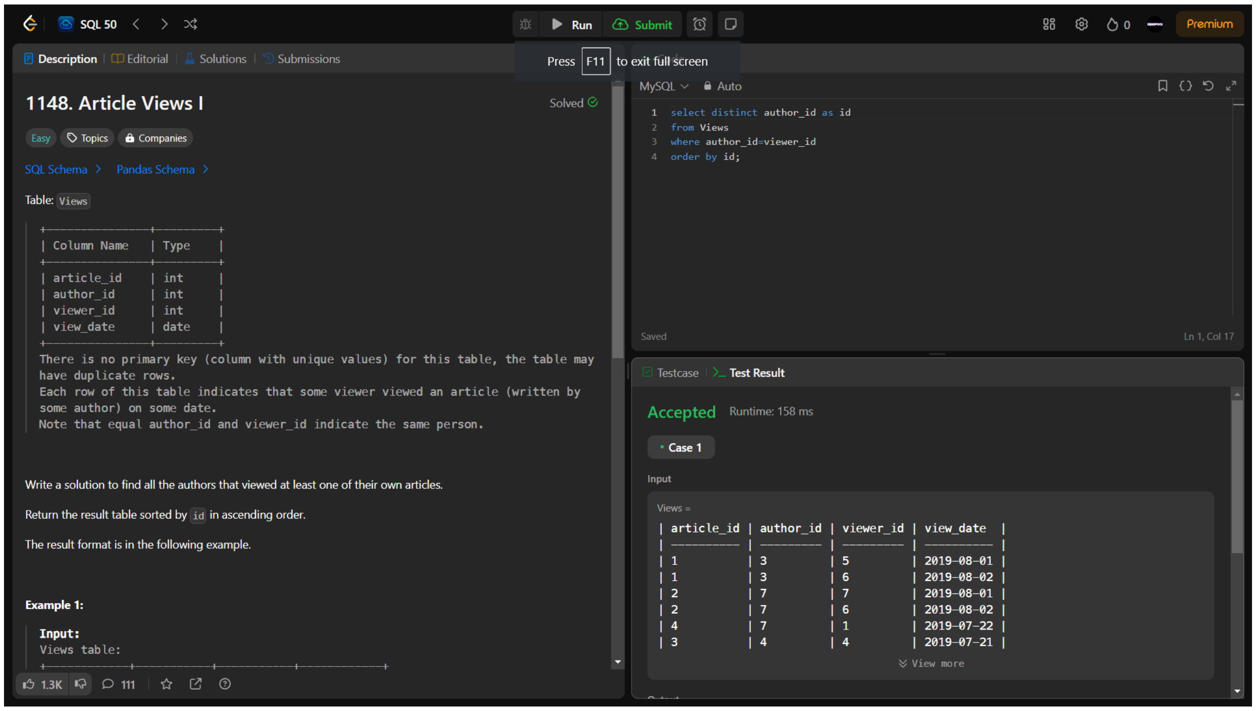
Output

```
| name        | population | area    |
```

answer:

```
select name, population, area
from World
where area>=3000000 or population>=25000000;
```

question 4:



1148. Article Views I

Table: Views

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| article_id    | int     |
| author_id     | int     |
| viewer_id     | int     |
| view_date     | date    |
+---------------+---------+
There is no primary key (column with unique values) for this table, the table 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 a solution to find all the authors that viewed at least one of their own articles.

Return the result table sorted by `id` in ascending order.

The result format is in the following example.

Example 1:

```
Input:
Views table:
```

```
select distinct author_id as id
from Views
where author_id=viewer_id
order by id;
```

Accepted Runtime: 158 ms

Case 1

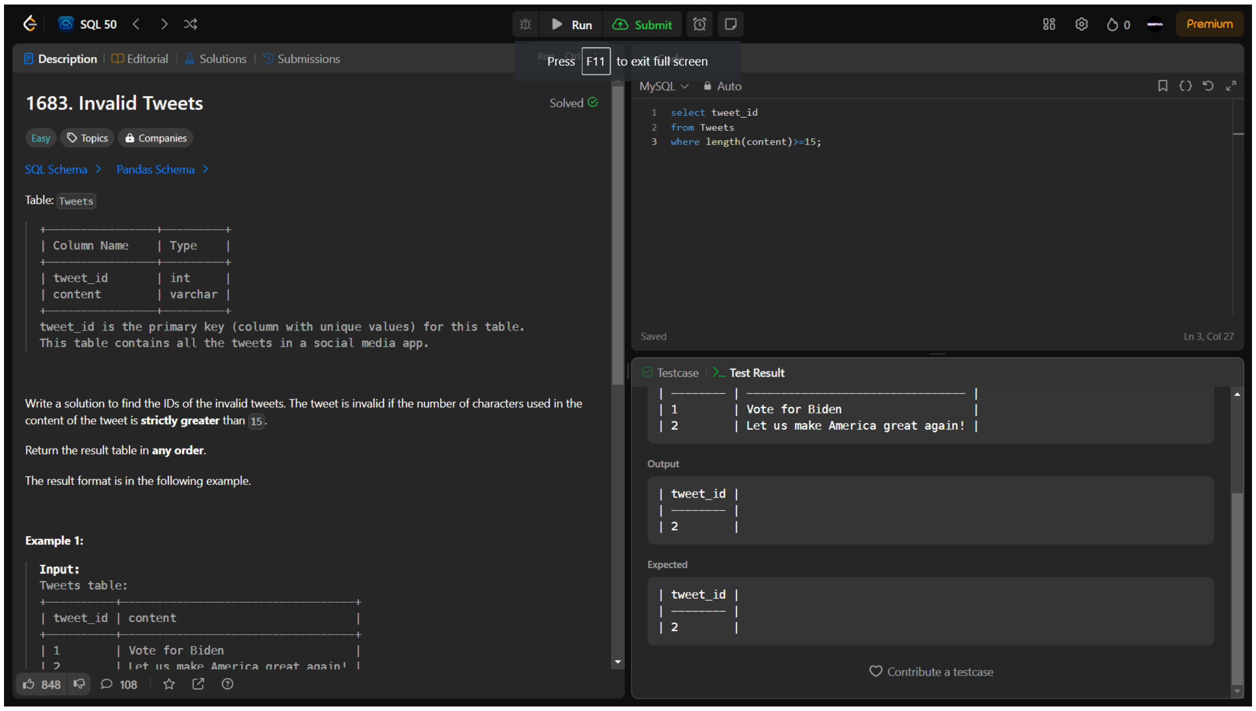
Input

```
Views =
| article_id | author_id | viewer_id | view_date  |
| ---------- | --------- | --------- | ---------- |
| 1          | 3         | 5         | 2019-08-01 |
| 1          | 3         | 6         | 2019-08-02 |
| 2          | 7         | 7         | 2019-08-01 |
| 2          | 7         | 6         | 2019-08-02 |
| 4          | 7         | 1         | 2019-07-22 |
| 3          | 4         | 4         | 2019-07-21 |
```

answer:

```
select distinct author_id as id
from Views
where author_id=viewer_id
order by id;
```

question 5:



1683. Invalid Tweets

Table: `Tweets`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| tweet_id       | int     |
| content        | varchar |
+----------------+---------+
tweet_id is the primary key (column with unique values) for this table.
This table contains all the tweets in a social media app.
```

Write a solution to find the IDs of the invalid tweets. The tweet is invalid if the number of characters used in the content of the tweet is **strictly greater** than `15`.

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

The result format is in the following example.

Example 1:

```
Input:
Tweets table:
+----------+----------------------------------+
| tweet_id | content                          |
+----------+----------------------------------+
| 1        | Vote for Biden                   |
| 2        | Let us make America great again! |
```

```sql
select tweet_id
from Tweets
where length(content)>=15;
```

```
| -------- | -------------------------------- |
| 1        | Vote for Biden                   |
| 2        | Let us make America great again! |
```

Output

```
| tweet_id |
| -------- |
| 2        |
```

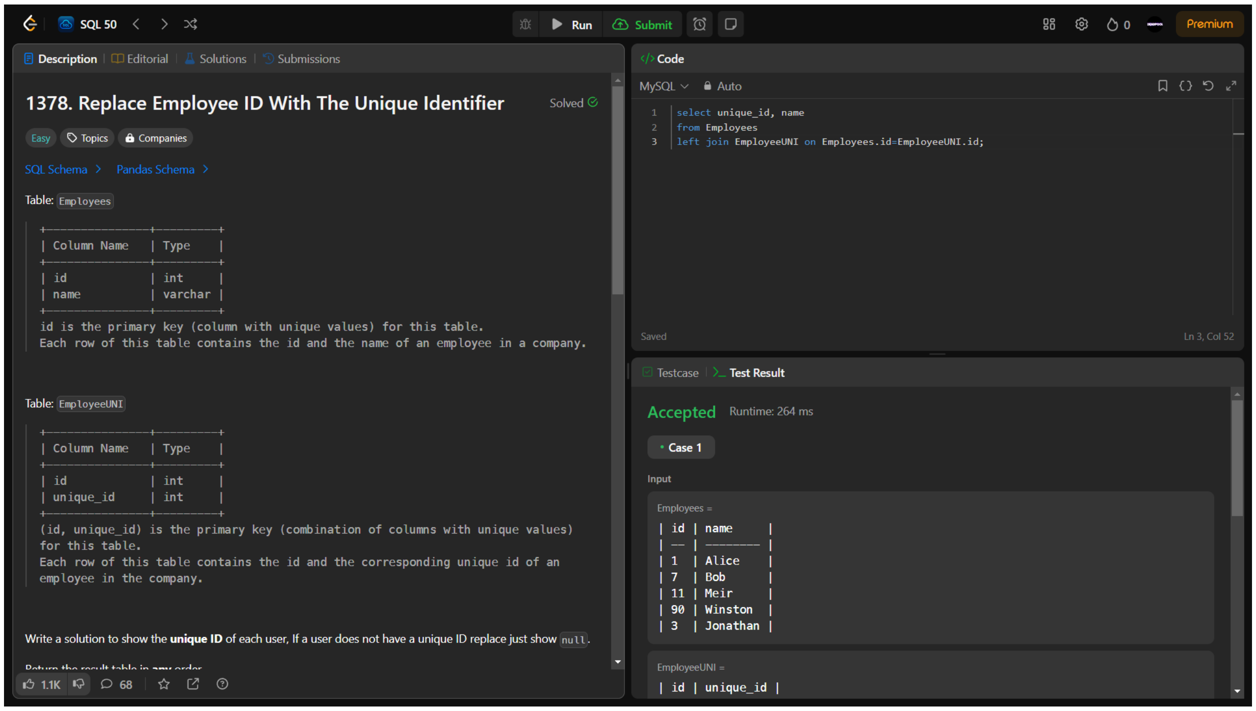
Expected

```
| tweet_id |
| -------- |
| 2        |
```

answer:

```
select tweet_id
from Tweets
where length(content)>15;
```

question 6:



1378. Replace Employee ID With The Unique Identifier

Solved

Table: `Employees`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| name          | varchar |
+---------------+---------+
id is the primary key (column with unique values) for this table.
Each row of this table contains the id and the name of an employee in a company.
```

Table: `EmployeeUNI`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| unique_id     | int     |
+---------------+---------+
(id, unique_id) is the primary key (combination of columns with unique values)
for this table.
Each row of this table contains the id and the corresponding unique id of an
employee in the company.
```

Write a solution to show the **unique ID** of each user, If a user does not have a unique ID replace just show `null`.

```
select unique_id, name
from Employees
left join EmployeeUNI on Employees.id=EmployeeUNI.id;
```

Saved   Ln 3, Col 52

Accepted   Runtime: 264 ms

Case 1

Input

Employees =

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

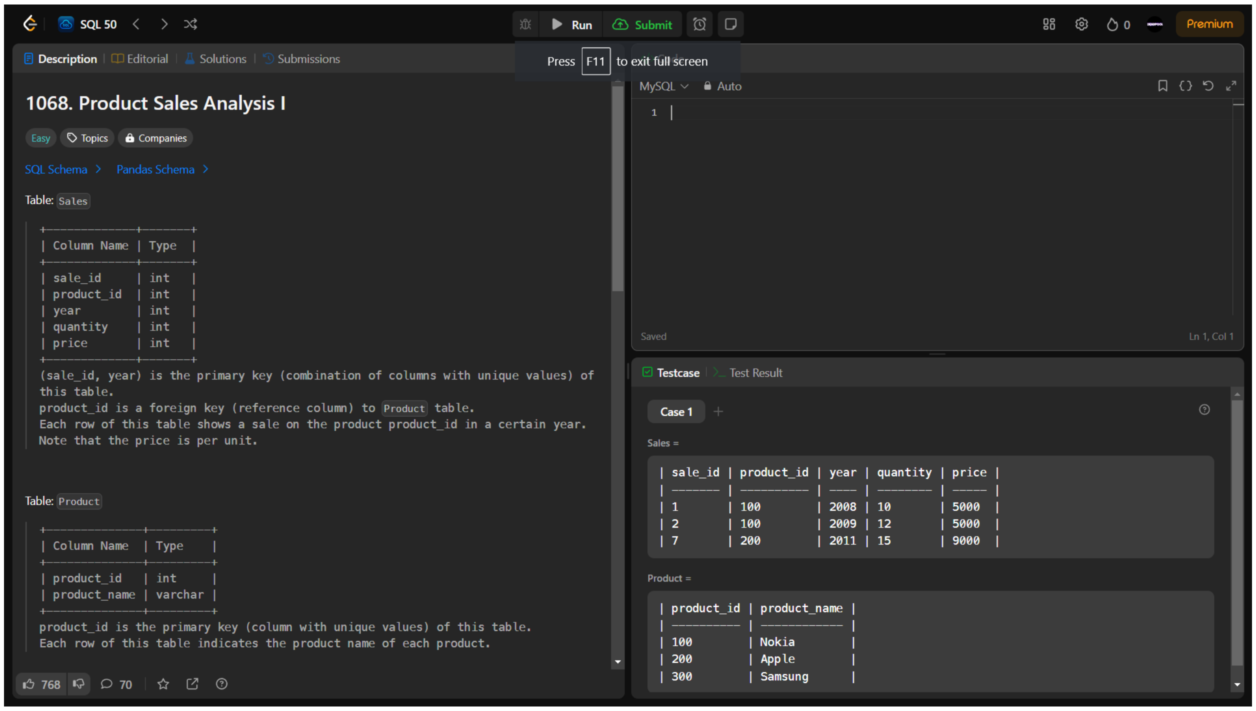
EmployeeUNI =

```
| id | unique_id |
```

answer:

```
select unique_id, name
from Employees
left join EmployeeUNI on Employees.id=EmployeeUNI.id;
```

q7:



1068. Product Sales Analysis I

Easy · Topics · Companies

SQL Schema > Pandas Schema >

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.
```

Table: `Product`

```
+--------------+---------+
| Column Name  | Type    |
+--------------+---------+
| product_id   | int     |
| product_name | varchar |
+--------------+---------+
product_id is the primary key (column with unique values) of this table.
Each row of this table indicates the product name of each product.
```

Testcase · Test Result

Case 1

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        |
| 200        | Apple        |
| 300        | Samsung      |
```

answer:

```
select product_name, year, price
from Sales inner join Product on Sales.product_id=Product.produc
```

q8:

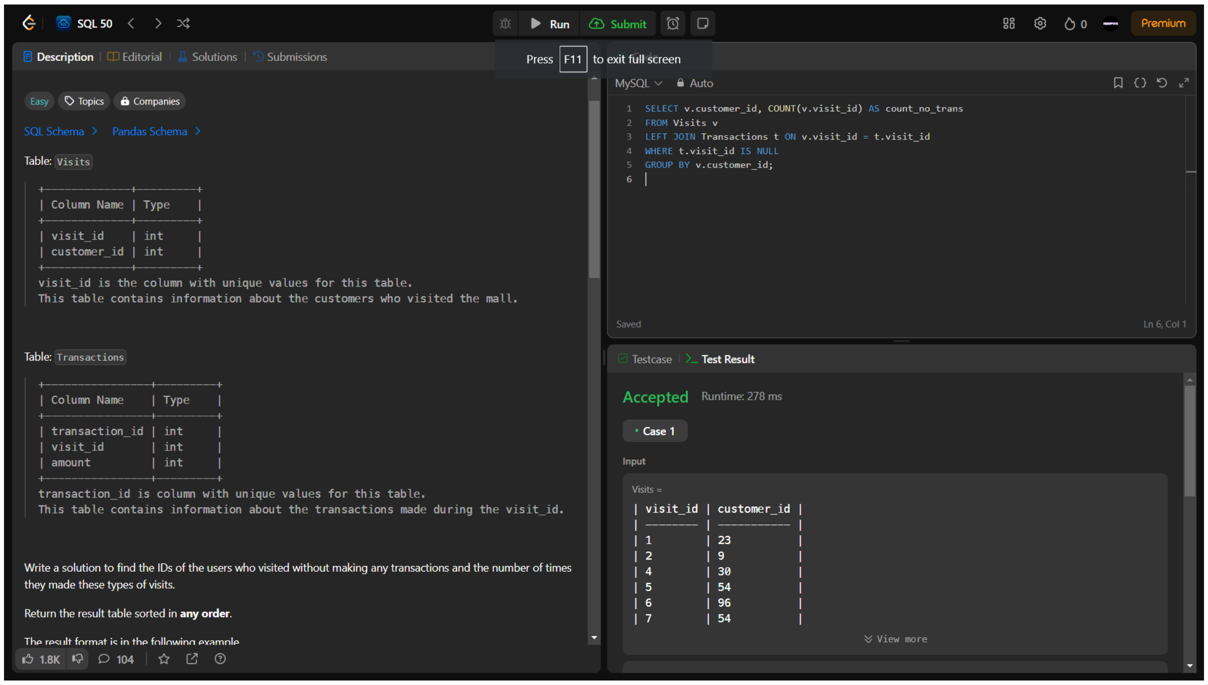


Table: `Visits`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| visit_id    | int     |
| customer_id | int     |
+-------------+---------+
visit_id is the column with unique values for this table.
This table contains information about the customers who visited the mall.
```

Table: `Transactions`

```
+----------------+---------+
| Column Name    | Type    |
+----------------+---------+
| transaction_id | int     |
| visit_id       | int     |
| amount         | int     |
+----------------+---------+
transaction_id is column with unique values for this table.
This table contains information about the transactions made during the visit_id.
```

Write a solution to find the IDs of the users who visited without making any transactions and the number of times they made these types of visits.

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

The result format is in the following example.

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

Accepted Runtime: 278 ms

Case 1

Input

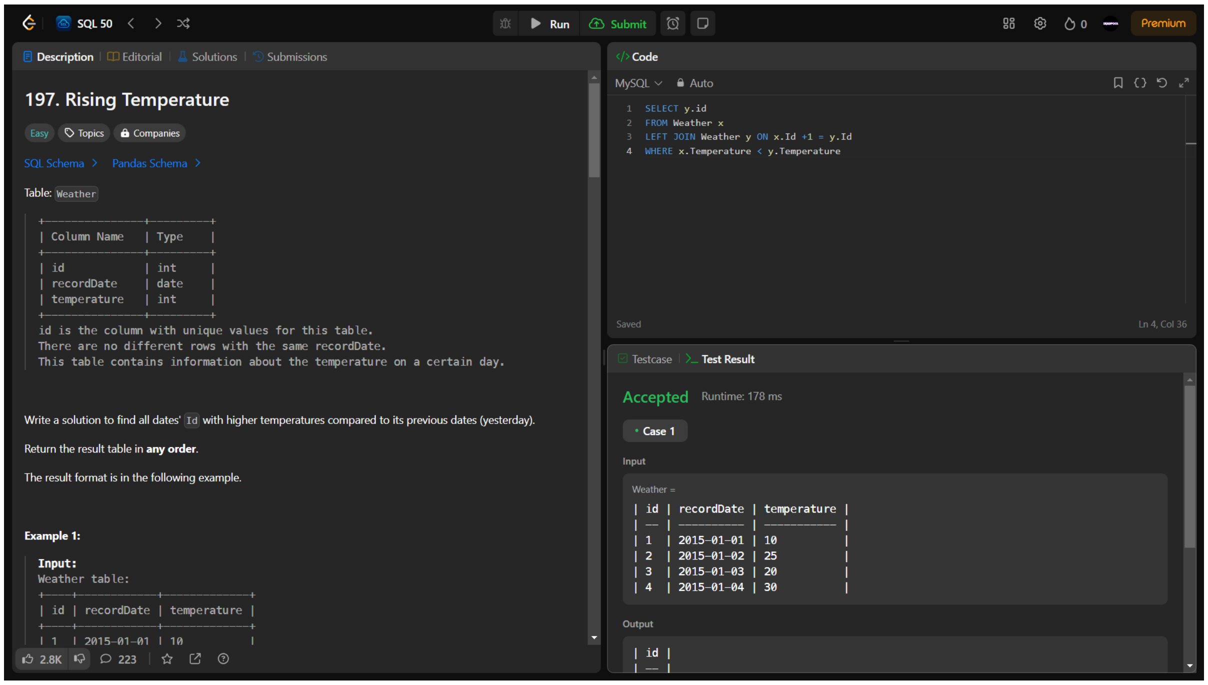
```
Visits =
| visit_id | customer_id |
| -------- | ----------- |
| 1        | 23          |
| 2        | 9           |
| 4        | 30          |
| 5        | 54          |
| 6        | 96          |
| 7        | 54          |
```

View more

answer:

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

q9:



197. Rising Temperature

Table: `Weather`

```
+---------------+---------+
| Column Name   | Type    |
+---------------+---------+
| id            | int     |
| recordDate    | date    |
| temperature   | int     |
+---------------+---------+
id is the column with unique values for this table.
There are no different rows with the same recordDate.
This table contains information about the temperature on a certain day.
```

Write a solution to find all dates' `Id` with higher temperatures compared to its previous dates (yesterday).

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

The result format is in the following example.

Example 1:

```
Input:
Weather table:
+----+------------+-------------+
| id | recordDate | temperature |
+----+------------+-------------+
| 1  | 2015-01-01 | 10          |
```

Code

MySQL Auto

```
SELECT y.id
FROM Weather x
LEFT JOIN Weather y ON x.Id +1 = y.Id
WHERE x.Temperature < y.Temperature
```

Saved Ln 4, Col 36

Testcase | Test Result

Accepted Runtime: 178 ms

Case 1

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          |
```

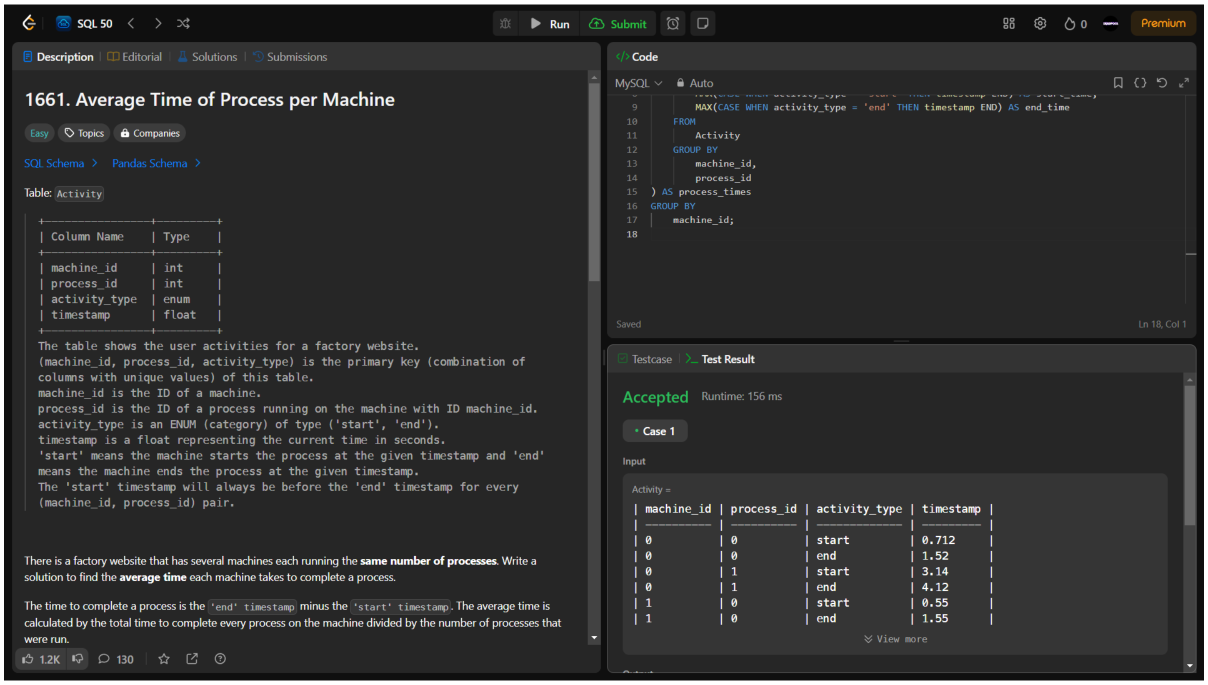
Output

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

answer:

```
SELECT w.id
FROM Weather w
JOIN Weather w_prev ON w.recordDate = DATE_ADD(w_prev.recordD
WHERE w.temperature > w_prev.temperature;
```

q10:


1661. Average Time of Process per Machine

Easy

SQL Schema   Pandas Schema

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.
```

There is a factory website that has several machines each running the **same number of processes**. Write a solution to find the **average time** each machine takes to complete a process.

The time to complete a process is the `'end' timestamp` minus the `'start' timestamp`. The average time is calculated by the total time to complete every process on the machine divided by the number of processes that were run.

```
        MAX(CASE WHEN activity_type = 'end' THEN timestamp END) AS end_time
    FROM
        Activity
    GROUP BY
        machine_id,
        process_id
) AS process_times
GROUP BY
    machine_id;
```

Accepted Runtime: 156 ms

Case 1

Input

Activity =

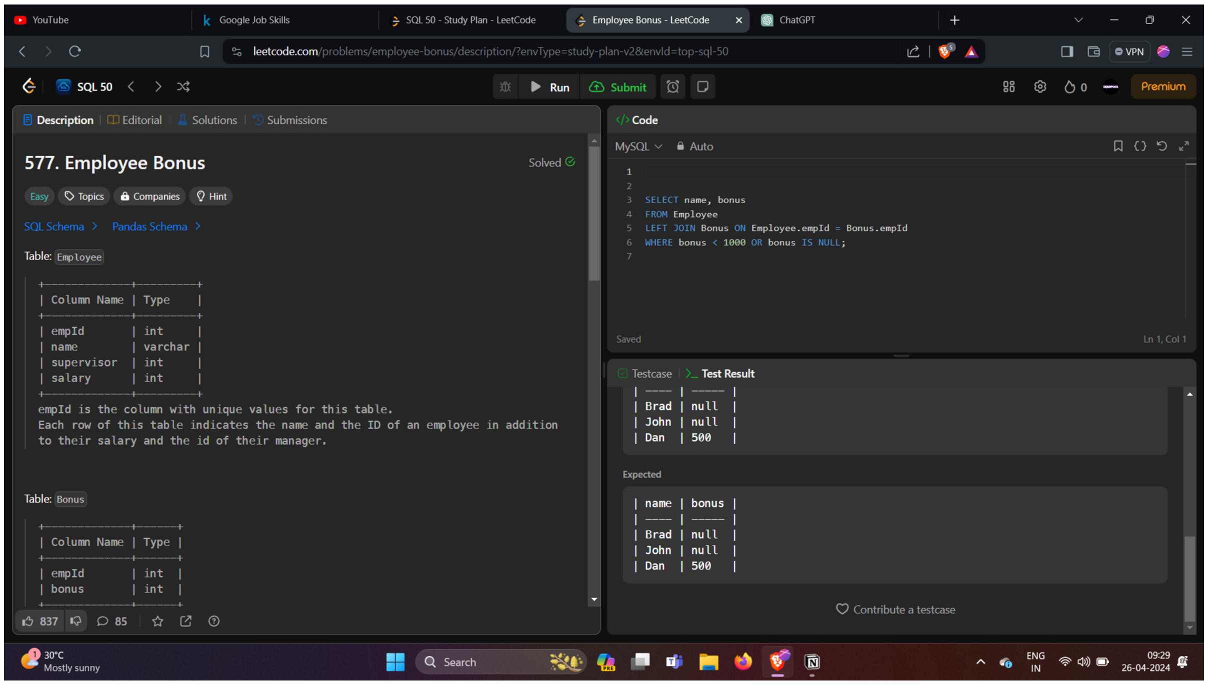
```
| machine_id | process_id | activity_type | timestamp |
| ---------- | ---------- | ------------- | --------- |
| 0          | 0          | start         | 0.712     |
| 0          | 0          | end           | 1.52      |
| 0          | 1          | start         | 3.14      |
| 0          | 1          | end           | 4.12      |
| 1          | 0          | start         | 0.55      |
| 1          | 0          | end           | 1.55      |
```

answer:

```
SELECT
    machine_id,
    ROUND(AVG(end_time - start_time), 3) AS processing_time
FROM (
    SELECT
        machine_id,
        process_id,
        MAX(CASE WHEN activity_type = 'start' THEN timestamp
        MAX(CASE WHEN activity_type = 'end' THEN timestamp EN
    FROM
        Activity
    GROUP BY
        machine_id,
        process_id
) AS process_times
```

```
GROUP BY
    machine_id;
```

q11:



577. Employee Bonus

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

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

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

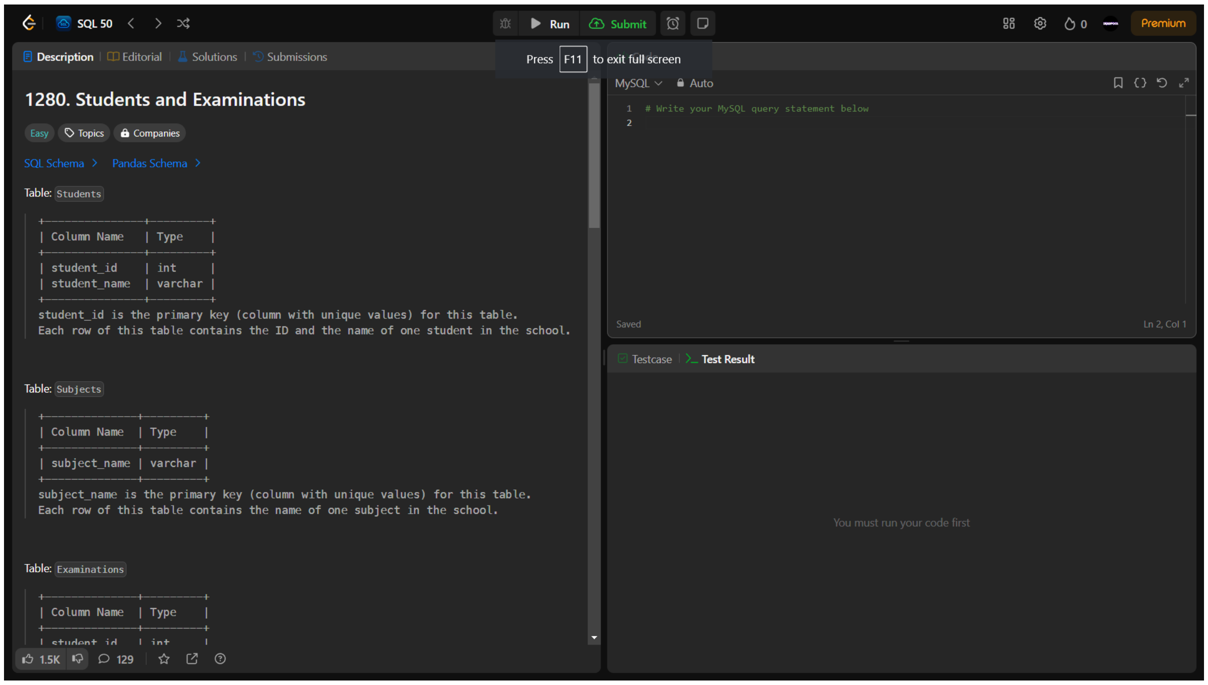
Expected

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

answer:

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

q12:



1280. Students and Examinations

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.
```

Table: `Examinations`

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

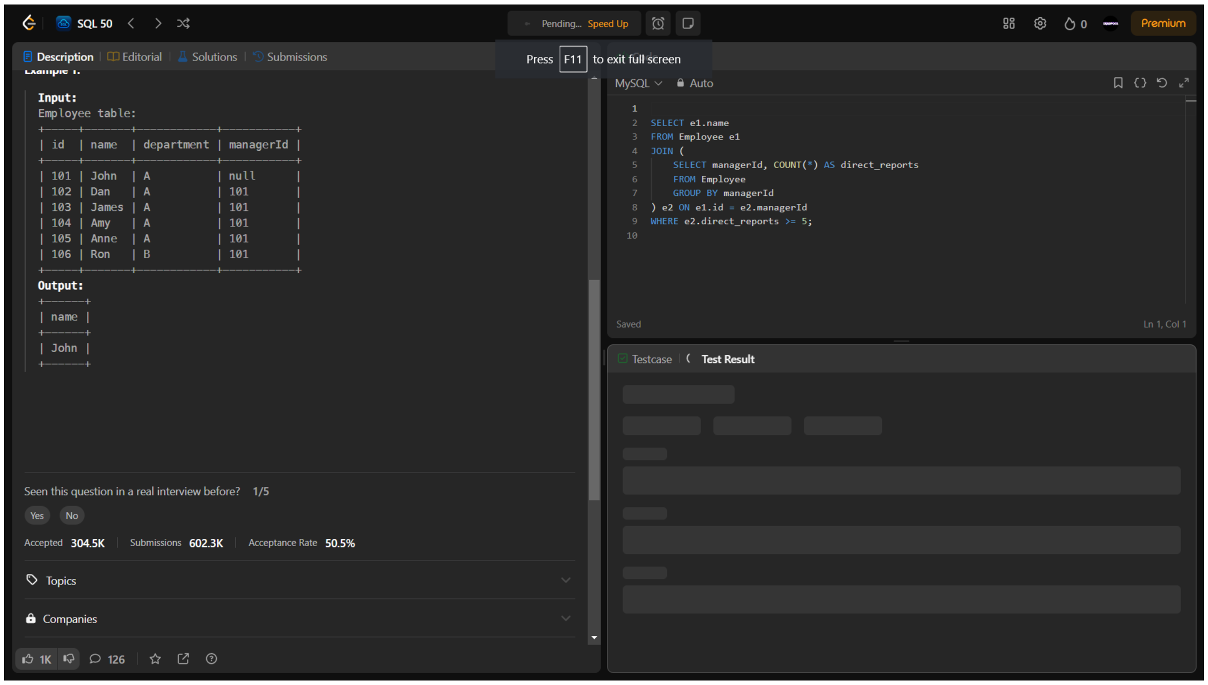
```
# Write your MySQL query statement below
```

answer:

SELECT
Students.student_id,
Students.student_name,
Subjects.subject_name,
COUNT(Examinations.subject_name) AS attended_exams
FROM
Students
CROSS JOIN
Subjects
LEFT JOIN
Examinations ON Students.student_id = Examinations.student_id
AND Subjects.subject_name = Examinations.subject_name
GROUP BY
Students.student_id,
Students.student_name,
Subjects.subject_name
ORDER BY

Students.student_id,
Subjects.subject_name;

q12:



```
Input:
Employee table:
+-----+-------+------------+-----------+
| 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       |
+-----+-------+------------+-----------+
Output:
+------+
| name |
+------+
| John |
+------+
```

Seen this question in a real interview before? 1/5

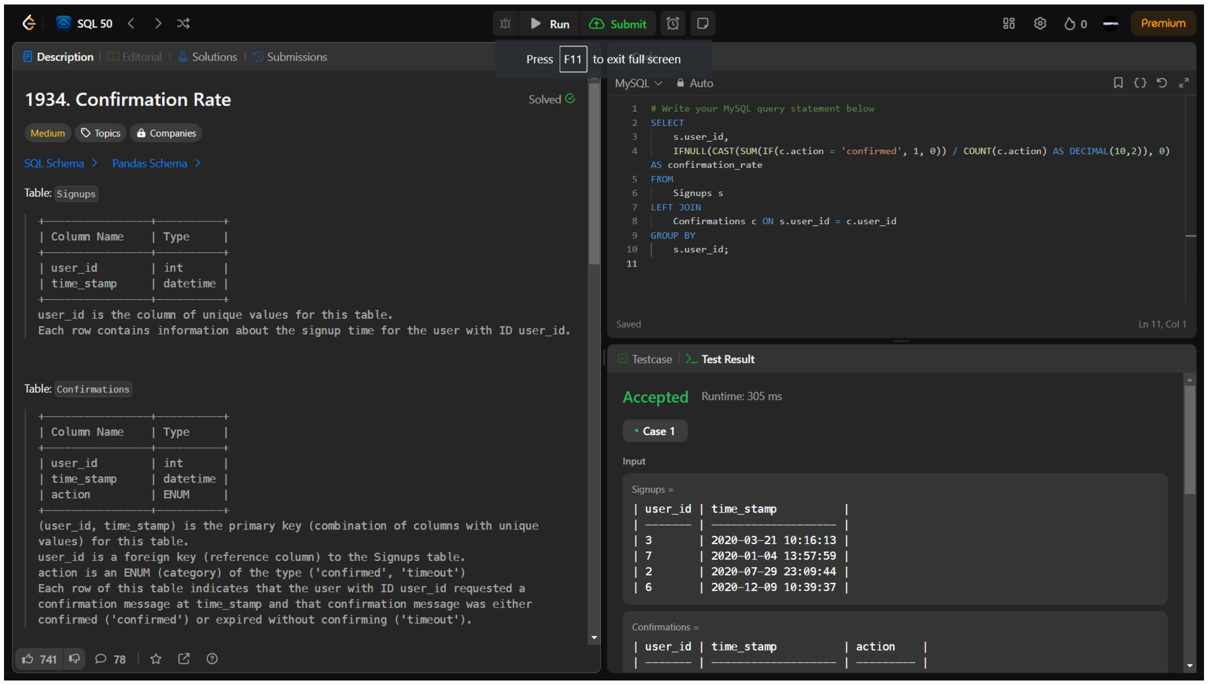
Accepted 304.5K | Submissions 602.3K | Acceptance Rate 50.5%

```
SELECT e1.name
FROM Employee e1
JOIN (
    SELECT managerId, COUNT(*) AS direct_reports
    FROM Employee
    GROUP BY managerId
) e2 ON e1.id = e2.managerId
WHERE e2.direct_reports >= 5;
```

answer:

```
SELECT e1.name
FROM Employee e1
JOIN (
    SELECT managerId, COUNT(*) AS direct_reports
    FROM Employee
    GROUP BY managerId
) e2 ON e1.id = e2.managerId
WHERE e2.direct_reports >= 5;
```

q13:


1934. Confirmation Rate

Solved

Medium · Topics · Companies

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 |
| action         | ENUM     |
+----------------+----------+
(user_id, time_stamp) is the primary key (combination of columns with unique
values) for this table.
user_id is a foreign key (reference column) to the Signups table.
action is an ENUM (category) of the type ('confirmed', 'timeout')
Each row of this table indicates that the user with ID user_id requested a
confirmation message at time_stamp and that confirmation message was either
confirmed ('confirmed') or expired without confirming ('timeout').
```

MySQL · Auto

```sql
# Write your MySQL query statement below
SELECT
    s.user_id,
    IFNULL(CAST(SUM(IF(c.action = 'confirmed', 1, 0)) / COUNT(c.action) AS DECIMAL(10,2)), 0)
AS confirmation_rate
FROM
    Signups s
LEFT JOIN
    Confirmations c ON s.user_id = c.user_id
GROUP BY
    s.user_id;
```

Saved · Ln 11, Col 1

Test Result

Accepted Runtime: 305 ms

Case 1

Input

Signups =

```
| user_id | time_stamp          |
| ------- | ------------------- |
| 3       | 2020-03-21 10:16:13 |
| 7       | 2020-01-04 13:57:59 |
| 2       | 2020-07-29 23:09:44 |
| 6       | 2020-12-09 10:39:37 |
```

Confirmations =

```
| user_id | time_stamp          | action    |
| ------- | ------------------- | --------- |
```

answer:

```
# Write your MySQL query statement below
SELECT
    s.user_id,
    IFNULL(CAST(SUM(IF(c.action = 'confirmed', 1, 0)) / COUNT
FROM
    Signups s
LEFT JOIN
    Confirmations c ON s.user_id = c.user_id
GROUP BY
    s.user_id;
```

q14:

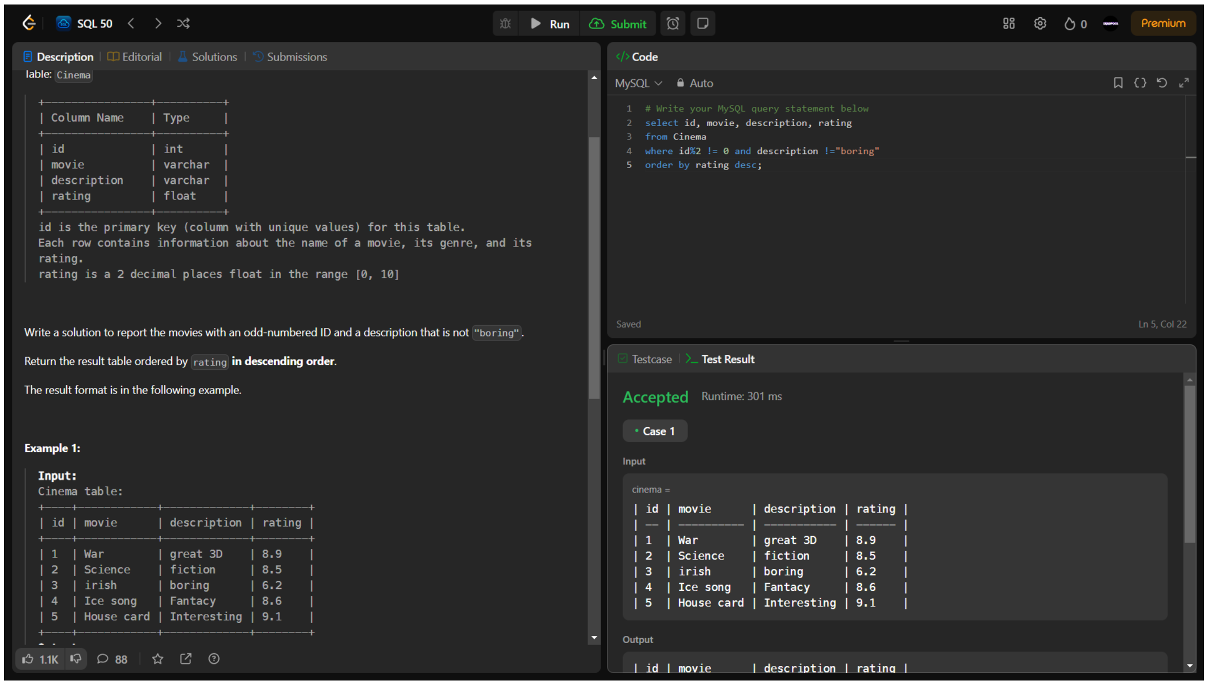


Table: `Cinema`

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

```
Input:
Cinema table:
+----+------------+-------------+--------+
| id | movie      | description | rating |
+----+------------+-------------+--------+
| 1  | War        | great 3D    | 8.9    |
| 2  | Science    | fiction     | 8.5    |
| 3  | irish      | boring      | 6.2    |
| 4  | Ice song   | Fantacy     | 8.6    |
| 5  | House card | Interesting | 9.1    |
+----+------------+-------------+--------+
```

```
# Write your MySQL query statement below
select id, movie, description, rating
from Cinema
where id%2 != 0 and description !="boring"
order by rating desc;
```

Saved   Ln 5, Col 22

Testcase | Test Result

Accepted Runtime: 301 ms

Case 1

Input

```
cinema =
| id | movie      | description | rating |
| -- | ---------- | ----------- | ------ |
| 1  | War        | great 3D    | 8.9    |
| 2  | Science    | fiction     | 8.5    |
| 3  | irish      | boring      | 6.2    |
| 4  | Ice song   | Fantacy     | 8.6    |
| 5  | House card | Interesting | 9.1    |
```

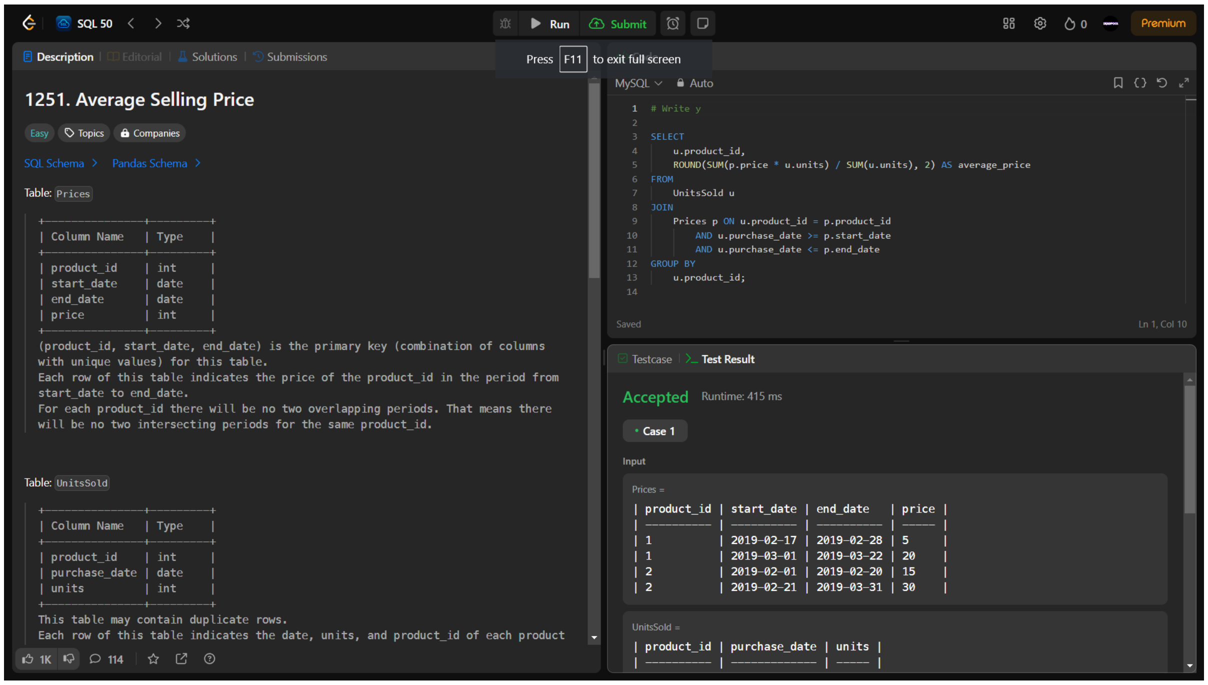
Output

```
| id | movie      | description | rating |
```

answer:

```
# Write your MySQL query statement below
select id, movie, description, rating
from Cinema
where id%2 != 0 and description !="boring"
order by rating desc;
```

q15:



1251. Average Selling Price

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    |
+---------------+---------+
| product_id    | int     |
| purchase_date | date    |
| units         | int     |
+---------------+---------+
This table may contain duplicate rows.
Each row of this table indicates the date, units, and product_id of each product
```

```
# Write y

SELECT
    u.product_id,
    ROUND(SUM(p.price * u.units) / SUM(u.units), 2) AS average_price
FROM
    UnitsSold u
JOIN
    Prices p ON u.product_id = p.product_id
        AND u.purchase_date >= p.start_date
        AND u.purchase_date <= p.end_date
GROUP BY
    u.product_id;
```

Accepted Runtime: 415 ms

Case 1

Input

Prices =

```
| product_id | start_date | end_date   | price |
| ---------- | ---------- | ---------- | ----- |
| 1          | 2019-02-17 | 2019-02-28 | 5     |
| 1          | 2019-03-01 | 2019-03-22 | 20    |
| 2          | 2019-02-01 | 2019-02-20 | 15    |
| 2          | 2019-02-21 | 2019-03-31 | 30    |
```

UnitsSold =

```
| product_id | purchase_date | units |
| ---------- | ------------- | ----- |
```

answer:

```
SELECT
    u.product_id,
    ROUND(SUM(p.price * u.units) / SUM(u.units), 2) AS averag
FROM
    UnitsSold u
JOIN
    Prices p ON u.product_id = p.product_id
        AND u.purchase_date >= p.start_date
        AND u.purchase_date <= p.end_date
GROUP BY
    u.product_id;
```

Q 16: