# LEETCODE SQL INTERVIEW QUESTIONS & SOLUTIONS

https://www.linkedin.com/in/niranjana405/

## 1)Recyclable and Low Fat Products

**Table: Products**

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

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**.

**Output:**

```
+-------------+
| product_id  |
+-------------+
| 1           |
| 3           |
+-------------+
```

### Solution

```
SELECT product_id FROM Products WHERE low_fats = 'Y' AND recyclable = 'Y';
```

---

## 2) Find Customer Referee

**Customer table:**

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

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**.

**Output:**

+------+
| name |
+------+
| Will |
| Jane |
| Bill |
| Zack |
+------+

## Solution

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

---

## 3) Find Big Countries

**World table:**

| name | continent | area | population | gdp |
|---|---|---|---|---|
| Afghanistan | Asia | 652230 | 25,500,100 | 20,343,000,000 |
| Albania | Europe | 28,748 | 2,831,741 | 12,960,000,000 |
| Algeria | Africa | 2,381,741 | 37,100,000 | 188,681,000,000 |
| Andorra | Europe | 468 | 78,115 | 3,712,000,000 |
| Angola | Africa | 1,246,700 | 20,609,294 | 100,990,000,000 |

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**.

**Output:**

| name | population | area |
|---|---|---|
| Afghanistan | 25,500,100 | 652,230 |
| Algeria | 37,100,000 | 2,381,741 |

## Solution

```
SELECT name, population, area FROM World WHERE area >= 3000000 OR population >= 25000000;
```

---

## 4)Find authors who viewed at least one of their own articles

**Views table:**

| 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 |
| 3 | 4 | 4 | 2019-07-21 |

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.

**Output:**

```
+------+
| id   |
+------+
| 4    |
| 7    |
+------+
```

## Solution

```
SELECT DISTINCT author_id AS id FROM Views WHERE author_id = viewer_id ORDER BY id;
```

---

# 5)Find Invalid Tweets

Tweets table:

```
+----------+----------------------------------+
| tweet_id | content                          |
+----------+----------------------------------+
| 1        | Let us Code                      |
| 2        | More than fifteen chars are here! |
+----------+----------------------------------+
```

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

content consists of characters on an American Keyboard, and no other special characters.

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**.

**Output:**

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

## Solution

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

---

# 6)Replace Emp Id with Unique Identifier

Employees table:

```
+----+----------+
| id | name     |
```

+----+----------+
| 1  | Alice    |
| 7  | Bob      |
| 11 | Meir     |
| 90 | Winston  |
| 3  | Jonathan |
+----+----------+

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.

EmployeeUNI table:
+----+-----------+
| id | unique_id |
+----+-----------+
| 3  | 1         |
| 11 | 2         |
| 90 | 3         |
+----+-----------+

(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.**

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

**Output:**

+-----------+----------+
| unique_id | name     |
+-----------+----------+
| null      | Alice    |
| null      | Bob      |
| 2         | Meir     |
| 3         | Winston  |
| 1         | Jonathan |
+-----------+----------+

## Solution

```
SELECT unique_id, name FROM Employees  LEFT JOIN EmployeeUNI  USING(id)
```

**Explanation:**

- Use LEFT JOIN to ensure that all employees from the Employees table are included in the result, even if they don't have a corresponding unique_id in the EmployeeUNI table.
- If an employee doesn't have a matching id in EmployeeUNI, unique_id will be NULL.

---

## 7)Find the product Sales analysis

**Sales table:**

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

(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.

**Product table:**

| product_id | product_name |
|---|---|
| 100 | Nokia |
| 200 | Apple |
| 300 | Samsung |

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.

**Write a solution to report the product_name, year, and price for each sale_id in the Sales table.**

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

**Output:**

| product_name | year | price |
|---|---|---|
| Nokia | 2008 | 5000 |
| Nokia | 2009 | 5000 |
| Apple | 2011 | 9000 |

## Solution

```
SELECT product_name, year, price FROM Sales  JOIN Product  USING(product_id)
```

## 8)Customers Who Visited mall but Did Not Make Any Transactions

**Table: Visits**

| visit_id | customer_id |
|---|---|
| 1 | 23 |
| 2 | 9 |
| 4 | 30 |
| 5 | 54 |
| 6 | 96 |
| 7 | 54 |
| 8 | 54 |

visit_id is the column with unique values for this table.

This table contains information about the customers who visited the mall.

**Table: Transactions**

| transaction_id | visit_id | amount |
|---|---|---|
| 2 | 5 | 310 |
| 3 | 5 | 300 |
| 9 | 5 | 200 |
| 12 | 1 | 910 |
| 13 | 2 | 970 |

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**.

**Output:**

| customer_id | count_no_trans |
|---|---|
| 54 | 2 |
| 30 | 1 |
| 96 | 1 |

## Solution

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

### *Explanation:*

Customer with id = 23 visited the mall once and made one transaction during the visit with id = 12.

Customer with id = 9 visited the mall once and made one transaction during the visit with id = 13.

Customer with id = 30 visited the mall once and did not make any transactions.

Customer with id = 54 visited the mall three times. During 2 visits they did not make any transactions, and during one visit they made 3 transactions.

Customer with id = 96 visited the mall once and did not make any transactions.

As we can see, users with IDs 30 and 96 visited the mall one time without making any transactions. Also, user 54 visited the mall twice and did not make any transactions.

1. **Use LEFT JOIN**:
   - We join the Visits table with the Transactions table on visit_id.
   - This helps us retrieve all visits, even if no transaction was made.
2. **Filter Out Customers Who Made Transactions**:
   - The condition WHERE t.transaction_id IS NULL ensures that we only select customers who visited but did not make any transactions.
3. **Count Such Visits for Each Customer**:
   - COUNT(v.visit_id) AS count_no_trans counts the number of times a customer visited without making a transaction.
   - We group by v.customer_id to get results per customer.

---

## 9)Find higher temperatures compared to its previous dates

**Weather table:**

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

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**.

**Output:**

```
+----+
| id |
+----+
| 2  |
| 4  |
+----+
```

## Solution

```
SELECT w1.id FROM Weather w1

JOIN Weather w2 ON DATEDIFF(w1.recordDate, w2.recordDate) = 1

WHERE w1.temperature > w2.temperature;
```

***Explanation***:

In 2015-01-02, the temperature was higher than the previous day (10 -> 25).

In 2015-01-04, the temperature was higher than the previous day (20 -> 30).

1. **Self Join on Weather Table**:
   - We join the Weather table (w1) with itself (w2) using the recordDate column.
   - The condition DATEDIFF(w1.recordDate, w2.recordDate) = 1 ensures that we are comparing each day's temperature with the previous day's temperature.
2. **Filter the Higher Temperature**:
   - We use WHERE w1.temperature > w2.temperature to select only those days where the temperature is higher than the previous day.

---

# 10)Find average time of process per machine

**Activity table:**

| machine_id | process_id | activity_type | timestamp |
|---|---|---|---|
| 0 | 0 | start | 0.712 |
| 0 | 0 | end | 1.520 |
| 0 | 1 | start | 3.140 |
| 0 | 1 | end | 4.120 |
| 1 | 0 | start | 0.550 |
| 1 | 0 | end | 1.550 |
| 1 | 1 | start | 0.430 |
| 1 | 1 | end | 1.420 |
| 2 | 0 | start | 4.100 |
| 2 | 0 | end | 4.512 |
| 2 | 1 | start | 2.500 |
| 2 | 1 | end | 5.000 |

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.

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. The resulting table should have the machine_id along with the **average time** as processing_time, which should be **rounded to 3 decimal places**.

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

**Output:**

```
+------------+-----------------+
```

| machine_id | processing_time |

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

| 0 | 0.894 |

| 1 | 0.995 |

| 2 | 1.456 |

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

## Solution

```
WITH ProcessTime AS

 (SELECT machine_id,

        process_id,

        MAX(CASE WHEN activity_type = 'end' THEN timestamp END) -

       MAX(CASE WHEN activity_type = 'start' THEN timestamp END) AS process_time

  FROM Activity

  GROUP BY machine_id, process_id )

SELECT machine_id,

      ROUND(AVG(process_time), 3) AS processing_time FROM ProcessTime

GROUP BY machine_id;
```

***Explanation:***

There are 3 machines running 2 processes each.

Machine 0's average time is ((1.520 - 0.712) + (4.120 - 3.140)) / 2 = 0.894

Machine 1's average time is ((1.550 - 0.550) + (1.420 - 0.430)) / 2 = 0.995

Machine 2's average time is ((4.512 - 4.100) + (5.000 - 2.500)) / 2 = 1.456

**Use a CTE (ProcessTime)**:

- We extract the start and end timestamps for each (machine_id, process_id) pair.
- Using MAX(CASE WHEN activity_type = 'end' THEN timestamp END), we get the end timestamp.
- Using MAX(CASE WHEN activity_type = 'start' THEN timestamp END), we get the start timestamp.
- The time taken for each process is calculated as end timestamp - start timestamp.

**Compute the Average Processing Time**:

- We use AVG(process_time) grouped by machine_id to compute the average processing time per machine.
- The result is rounded to 3 decimal places using ROUND().

---

# 11)Find Employee bonus

**Employee table:**

| empId | name | supervisor | salary |
|---|---|---|---|
| 3 | Brad | null | 4000 |
| 1 | John | 3 | 1000 |
| 2 | Dan | 3 | 2000 |
| 4 | Thomas | 3 | 4000 |

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**

| empId | bonus |
|---|---|
| 2 | 500 |
| 4 | 2000 |

empId is the column of unique values for this table.

empId is a foreign key (reference column) to empId from the Employee table.

Each row of this table contains the id of an employee and their respective bonus.

**Write a solution to report the name and bonus amount of each employee with a bonus less than 1000.**

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

**Output:**

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

## Solution

```
SELECT e.name,  b.bonus FROM Employee e

LEFT JOIN Bonus b ON e.empId = b.empId

WHERE b.bonus < 1000 OR b.bonus IS NULL;
```

***Explanation:***

- **LEFT JOIN**: We join the Employee table with the Bonus table on empId. This ensures that all employees are included, even if they don't have a bonus.

---

## 12)Find  the number of times each student attended each exam.

**Students table:**

| student_id | student_name |
|---|---|
| 1 | Alice |
| 2 | Bob |
| 13 | John |
| 6 | Alex |

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**

| subject_name |
|---|
| Math |
| Physics |
| Programming |

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**

+------------+--------------+
| student_id | subject_name |
+------------+--------------+
| 1          | Math         |
| 1          | Physics      |
| 1          | Programming  |
| 2          | Programming  |
| 1          | Physics      |
| 1          | Math         |
| 13         | Math         |
| 13         | Programming  |

| 13 | Physics |
| 2 | Math |
| 1 | Math |
+------------+--------------+

There is no primary key (column with unique values) for this table. It may contain duplicates.

Each student from the Students table takes every course from the Subjects table.

Each row of this table indicates that a student with ID student_id attended the exam of subject_name.

**Write a solution to find the number of times each student attended each exam.**

**Return the result table ordered by student_id and subject_name.**

The result format is in the following example.

**Output:**

| student_id | student_name | subject_name | attended_exams |
|---|---|---|---|
| 1 | Alice | Math | 3 |
| 1 | Alice | Physics | 2 |
| 1 | Alice | Programming | 1 |
| 2 | Bob | Math | 1 |
| 2 | Bob | Physics | 0 |
| 2 | Bob | Programming | 1 |
| 6 | Alex | Math | 0 |
| 6 | Alex | Physics | 0 |
| 6 | Alex | Programming | 0 |
| 13 | John | Math | 1 |
| 13 | John | Physics | 1 |
| 13 | John | Programming | 1 |

## Solution

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

***Explanation:***

**CROSS JOIN**:

- We first create a combination of all students with all subjects using CROSS JOIN **(no common column id needed for CROSS JOIN)**
- This ensures that even students who haven't attended any exams still appear in the result.

**LEFT JOIN with Examinations**:

- We join the generated student-subject combination with the Examinations table to count the number of times a student attended an exam for a given subject.

**COUNT(e.subject_name)**:

- If a student attended an exam, COUNT counts the occurrences.
- If a student did not attend an exam for that subject, COUNT returns 0 due to LEFT JOIN.

**GROUP BY & ORDER BY**:

- We group by student_id, student_name, and subject_name to get the correct counts per student and subject.
- The final output is sorted by student_id and subject_name to match the required format.

The result table should contain all students and all subjects.

Alice attended the Math exam 3 times, the Physics exam 2 times, and the Programming exam 1 time.

Bob attended the Math exam 1 time, the Programming exam 1 time, and did not attend the Physics exam.

Alex did not attend any exams.

John attended the Math exam 1 time, the Physics exam 1 time, and the Programming exam 1 time.

---

## 13)Find managers with at least five direct reports.

**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 |

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**.

**Output:**

+------+
| name |
+------+
| John |
+------+

## Solution

```
SELECT e.name FROM Employee e JOIN Employee m ON e.id = m.managerId

GROUP BY m.managerID HAVING COUNT(*) >= 5;
```

---

## 14)Find  the confirmation rate of each user.

**Signups table:**

| 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 |

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**

| user_id | time_stamp | action |
|---|---|---|
| 3 | 2021-01-06 03:30:46 | timeout |
| 3 | 2021-07-14 14:00:00 | timeout |
| 7 | 2021-06-12 11:57:29 | confirmed |
| 7 | 2021-06-13 12:58:28 | confirmed |
| 7 | 2021-06-14 13:59:27 | confirmed |
| 2 | 2021-01-22 00:00:00 | confirmed |
| 2 | 2021-02-28 23:59:59 | timeout |

(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').

The **confirmation rate** of a user is the number of 'confirmed' messages divided by the total number of requested confirmation messages. The confirmation rate of a user that did not request any confirmation messages is 0. Round the confirmation rate to **two decimal** places.

**Write a solution to find the confirmation rate of each user.**

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

**Output:**

```
+---------+-------------------+
| user_id | confirmation_rate |
+---------+-------------------+
| 6       | 0.00              |
| 3       | 0.00              |
| 7       | 1.00              |
| 2       | 0.50              |
+---------+-------------------+
```

## Solution

```
SELECT user_id,
    ROUND(COALESCE(SUM(IF(C.action='confirmed', 1, 0))/COUNT(C.action), 0), 2) AS confirmation_rate
```

```
FROM Signups

LEFT JOIN Confirmations C USING(user_id)

GROUP BY user_id;
```

***Explanation:***

- **SUM(IF(C.action='confirmed', 1, 0))/COUNT(C.action), 0), 2)** : This part sums up the results of the CASE statement. It counts the number of times the CASE statement returned 1, which effectively counts the number of 'confirmed' actions for each user.
- **COALESCE(..., 0)**: This is crucial. COALESCE handles the case where a user has *no* entries in the Confirmations table. In that scenario, COUNT(c.action) would be 0, and dividing by zero would result in an error or NULL. COALESCE checks the result of the division.

User 6 did not request any confirmation messages. The confirmation rate is 0.

User 3 made 2 requests and both timed out. The confirmation rate is 0.

User 7 made 3 requests and all were confirmed. The confirmation rate is 1.

User 2 made 2 requests where one was confirmed and the other timed out. The confirmation rate is 1 / 2 = 0.5.

---

## 15)Find movies with odd numbered id and not boring

**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 | Fantasy | 8.6 |
| 5 | House card | Interesting | 9.1 |

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**.

**Output:**

```
+----+------------+-------------+--------+
```

| id | movie | description | rating |
+----+------------+-------------+--------+
| 5 | House card | Interesting | 9.1 |
| 1 | War | great 3D | 8.9 |
+----+------------+-------------+--------+

## Solution

```
SELECT id, movie, description, rating FROM Cinema

WHERE mod(id, 2)!=0 AND description <>'boring' ORDER BY rating DESC;
```

---

## 16)Find average selling prices for each product

**Prices table:**

| 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 |

(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**

| product_id | purchase_date | units |
|---|---|---|
| 1 | 2019-02-25 | 100 |
| 1 | 2019-03-01 | 15 |
| 2 | 2019-02-10 | 200 |
| 2 | 2019-03-22 | 30 |

This table may contain duplicate rows.

Each row of this table indicates the date, units, and product_id of each product sold.

**Write a solution to find the average selling price for each product. average_price should be rounded to 2 decimal places. If a product does not have any sold units, its average selling price is assumed to be 0.**

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

**Output:**

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

## Solution

```
SELECT p.product_id,
    ROUND(COALESCE(SUM(p.price * u.units) / SUM(u.units), 0), 2) AS average_price
FROM Prices p
LEFT JOIN UnitsSold u ON p.product_id = u.product_id
AND u.purchase_date BETWEEN p.start_date AND p.end_date
GROUP BY p.product_id;
```

***Explanation:***

Average selling price = Total Price of Product / Number of products sold.

Average selling price for product 1 = ((100 * 5) + (15 * 20)) / 115 = 6.96

Average selling price for product 2 = ((200 * 15) + (30 * 30)) / 230 = 16.96

---

## 17)Find average experience of all employees per project

**Project table:**

| project_id | employee_id |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 1 | 3 |
| 2 | 1 |
| 2 | 4 |

(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**

| employee_id | name | experience_years |
|---|---|---|
| 1 | Khaled | 3 |
| 2 | Ali | 2 |
| 3 | John | 1 |
| 4 | Doe | 2 |

employee_id is the primary key of this table. It's guaranteed that experience_years is not NULL.

Each row of this table contains information about one employee.

**Write an SQL query that reports the average experience years of all the employees for each project, rounded to 2 digits.**

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

**Output:**

```
+-------------+---------------+
| project_id  | average_years |
+-------------+---------------+
| 1           | 2.00          |
| 2           | 2.50          |
+-------------+---------------+
```

## Solution

```
SELECT p.project_id, ROUND(AVG(e.experience_years), 2) AS average_years FROM Project p

JOIN Employee e ON p.employee_id = e.employee_id GROUP BY p.project_id;
```

***Explanation:***

The average experience years for the first project is (3 + 2 + 1) / 3 = 2.00 and for the second project is (3 + 2) / 2 = 2.50

---

## 18)Find percentage of the users registered in each contest

**Users table:**

Here is your data in table format:

| user_id | user_name |
|---|---|
| 6 | Alice |
| 2 | Bob |
| 7 | Alex |

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

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

**Table: Register**

| contest_id | user_id |
|---|---|
| 215 | 6 |
| 209 | 2 |
| 208 | 2 |
| 210 | 6 |
| 208 | 6 |
| 209 | 7 |
| 209 | 6 |
| 215 | 7 |
| 208 | 7 |
| 210 | 2 |
| 207 | 2 |
| 210 | 7 |

(contest_id, user_id) is the primary key (combination of columns with unique values) for this table.

Each row of this table contains the id of a user and the contest they registered into.

**Write a solution to find the percentage of the users registered in each contest rounded to two decimals.**

Return the result table ordered by percentage in **descending order**. In case of a tie, order it by contest_id in **ascending order**.

**Output:**

| contest_id | percentage |
|---|---|
| 208 | 100.0 |
| 209 | 100.0 |
| 210 | 100.0 |
| 215 | 66.67 |
| 207 | 33.33 |

## Solution

```
SELECT contest_id,
    round(COUNT(USER_ID)*100 /
        (SELECT count(user_id) FROM users), 2) AS percentage
FROM USERS
JOIN REGISTER USING(user_id)
GROUP BY contest_id
ORDER BY percentage DESC, CONTEST_ID ASC ;
```

***Explanation:***

**Finds total users** → (SELECT COUNT(user_id) FROM Users).

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%

Bob registered in contest 207 and the percentage is ((1/3) * 100) = 33.33%

---

## 19)Find queries quality and percentage

**Queries table:**

| query_name | result | position | rating |
|---|---|---|---|
| Dog | Golden Retriever | 1 | 5 |
| Dog | German Shepherd | 2 | 5 |
| Dog | Mule | 200 | 1 |
| Cat | Shirazi | 5 | 2 |
| Cat | Siamese | 3 | 3 |
| Cat | Sphynx | 7 | 4 |

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.

**We also define poor query percentage as:**

The percentage of all queries with rating less than 3.

**Write a solution to find each query_name, the quality and poor_query_percentage.**

Both quality and poor_query_percentage should be **rounded to 2 decimal places**.

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

**Output:**

| query_name | quality | poor_query_percentage |
|---|---|---|
| Dog | 2.50 | 33.33 |
| Cat | 0.66 | 33.33 |

## Solution

```
SELECT query_name,
    ROUND(AVG(rating / POSITION), 2) AS quality,
```

```
    ROUND(SUM ( CASE WHEN rating < 3 THEN 1 ELSE 0 END) *100 /  COUNT(*), 2)  AS poor_query_percentage
FROM Queries
GROUP BY query_name;
```

***Explanation:***

Dog queries quality is ((5 / 1) + (5 / 2) + (1 / 200)) / 3 = 2.50

Dog queries poor_ query_percentage is (1 / 3) * 100 = 33.33

Cat queries quality equals ((2 / 5) + (3 / 3) + (4 / 7)) / 3 = 0.66

Cat queries poor_ query_percentage is (1 / 3) * 100 = 33.33

---

## 20)Find Monthly Transactions Summary by Country

**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 |

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.**

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

**Output:**

| 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 |

## Solution

```
SELECT  DATE_FORMAT(trans_date, '%Y-%m') AS MONTH,
    country,
    COUNT(trans_date) AS trans_count,
    SUM(IF(state = 'approved', 1, 0)) AS approved_count,
    SUM(amount) AS trans_total_amount,
    SUM(IF(state = 'approved', amount, 0)) AS approved_total_amount
FROM transactions
GROUP BY country, MONTH;
```

---

## 21)Find immediate food delivery for the first orders

**Delivery table:**

| delivery_id | customer_id | order_date | customer_pref_delivery_date |
|---|---|---|---|
| 1 | 1 | 2019-08-01 | 2019-08-02 |
| 2 | 2 | 2019-08-02 | 2019-08-02 |
| 3 | 1 | 2019-08-11 | 2019-08-12 |
| 4 | 3 | 2019-08-24 | 2019-08-24 |
| 5 | 3 | 2019-08-21 | 2019-08-22 |
| 6 | 2 | 2019-08-11 | 2019-08-13 |
| 7 | 4 | 2019-08-09 | 2019-08-09 |

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;** otherwise, it is called **scheduled**.

The **first order** of a customer is the order with the earliest order date that the customer made. It is guaranteed that a customer has precisely one first order.

**Write a solution to find the percentage of immediate orders in the first orders of all customers, rounded to 2 decimal places.**

**Output:**

```
+----------------------+
| immediate_percentage |
+----------------------+
| 50.00                |
+----------------------+
```

## Solution

```
WITH FirstOrders AS

 (SELECT *,  ROW_NUMBER() over(PARTITION BY customer_id ORDER BY order_date ASC) AS min_order_date

  FROM Delivery)

SELECT ROUND(count(*)*100 /

            (SELECT count(DISTINCT customer_id) FROM Delivery), 2) AS immediate_percentage

FROM FirstOrders

WHERE order_date=customer_pref_delivery_date AND min_order_date=1
```

***Explanation:***

The customer id 1 has a first order with delivery id 1 and it is scheduled.

The customer id 2 has a first order with delivery id 2 and it is immediate.

The customer id 3 has a first order with delivery id 5 and it is scheduled.

The customer id 4 has a first order with delivery id 7 and it is immediate.

**Hence, half the customers have immediate first orders**

1️⃣ Understanding the WITH Clause (Common Table Expression - CTE)

This **CTE (FirstOrders)** finds the **first order** of each customer using the **ROW_NUMBER()** window function.

PARTITION BY customer_id → Groups data by customer_id, so that each customer's orders are processed separately.

ORDER BY order_date ASC → Assigns a **row number** to each order in ascending order of order_date.

The **first order of each customer** will have min_order_date = 1

### 2️⃣ Filtering for Immediate First Orders

**Filters**:

- min_order_date = 1 → Ensures only the **first orders** of each customer are considered.
- order_date = customer_pref_delivery_date → Selects **only immediate orders** (orders delivered on the same day).

**Calculation**:

- COUNT(*) → Counts the number of **immediate first orders**.
- (SELECT COUNT(DISTINCT customer_id) FROM Delivery) → Counts the **total number of customers**.

---

## 22)Find Game plan analysis

**Activity table:**

Here is your data in table format:

| 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 |

(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.**

**Output:**

```
+-----------+
| fraction  |
+-----------+
| 0.33      |
+-----------+
```

## Solution

```
SELECT ROUND(COUNT(DISTINCT player_id) /
            (SELECT COUNT(DISTINCT player_id)
             FROM Activity), 2) AS fraction
FROM Activity
WHERE (player_id, DATE_SUB(event_date, INTERVAL 1 DAY))
 IN
   (SELECT player_id, MIN(event_date) AS first_login  FROM ACTIVITY GROUP BY player_id)
```

***Explanation:***

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

***1. Subquery (Finding First Login Date)***

```
SELECT player_id, MIN(event_date) AS first_login
FROM Activity
GROUP BY player_id
```

- This finds the **earliest login date (first login)** for each player_id.

***2. Main Query (Checking Next Day Logins)***

```
SELECT ROUND(COUNT(DISTINCT player_id) /
   (SELECT COUNT(DISTINCT player_id) FROM Activity), 2) as fraction
FROM Activity
WHERE (player_id, DATE_SUB(event_date, INTERVAL 1 DAY))
IN (SELECT player_id, MIN(event_date) AS first_login FROM Activity GROUP BY player_id)
```

- The **WHERE condition** checks if (player_id, event_date - 1 day) exists in the first login subquery.
- **DATE_SUB(event_date, INTERVAL 1 DAY)** shifts the event_date back by one day.
- The **IN clause** ensures that this new date matches the first login date of that player.

---

## 23)Find the number of unique subjects taught by each teacher

| teacher_id | subject_id | dept_id |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 2 | 4 |
| 1 | 3 | 3 |
| 2 | 1 | 1 |
| 2 | 2 | 1 |
| 2 | 3 | 1 |
| 2 | 4 | 1 |

(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**.

**Output:**

| teacher_id | cnt |
|---|---|
| 1 | 2 |
| 2 | 4 |

## Solution

```
SELECT teacher_id,  COUNT(DISTINCT subject_id) AS cnt FROM Teacher GROUP BY teacher_id;
```

***Explanation:***

Teacher 1:

- They teach subject 2 in departments 3 and 4.

- They teach subject 3 in department 3.

Teacher 2:

- They teach subject 1 in department 1.

- They teach subject 2 in department 1.

- They teach subject 3 in department 1.

- They teach subject 4 in department 1.

## 24)Find the User activity for the past 30 days

**Activity table:**

Here is your data in table format:

| user_id | session_id | activity_date | activity_type |
|---|---|---|---|
| 1 | 1 | 2019-07-20 | open_session |
| 1 | 1 | 2019-07-20 | scroll_down |
| 1 | 1 | 2019-07-20 | end_session |
| 2 | 4 | 2019-07-20 | open_session |
| 2 | 4 | 2019-07-21 | send_message |
| 2 | 4 | 2019-07-21 | end_session |
| 3 | 2 | 2019-07-21 | open_session |
| 3 | 2 | 2019-07-21 | send_message |
| 3 | 2 | 2019-07-21 | end_session |
| 4 | 3 | 2019-06-25 | open_session |
| 4 | 3 | 2019-06-25 | end_session |

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.

Write a solution to find the daily active user count for a period of 30 days ending 2019-07-27inclusively. A user was active on someday if they made at least one activity on that day.

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

**Output:**

| day | active_users |
|---|---|
| 2019-07-20 | 2 |
| 2019-07-21 | 2 |

**Note** that we do not care about days with zero active users.

## Solution

```
SELECT activity_date AS DAY, COUNT(DISTINCT user_id) AS active_users  FROM Activity

WHERE activity_date BETWEEN DATE_SUB('2019-07-27', INTERVAL 29 DAY) AND '2019-07-27'

GROUP BY activity_date;
```

***Explanation:***

1. **WHERE activity_date BETWEEN DATE_SUB('2019-07-27', INTERVAL 29 DAY) AND '2019-07-27'**:
   - Filters records for the last **30 days**, **ending on 2019-07-27** (from **2019-06-28** to **2019-07-27**).
2. **COUNT(DISTINCT user_id) AS active_users**:
   - Counts the number of unique users (user_id) who performed any activity on each date.
3. **GROUP BY activity_date**:
   - Groups the data by activity_date to get the count of active users per day.

## 25)Find the sales analysis for the first year of every product sold

**Table: Sales**

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

(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.

**Write a solution to select the product id, year, quantity, and price for the first year of every product sold.**

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

**Output:**

| product_id | first_year | quantity | price |
|---|---|---|---|
| 100 | 2008 | 10 | 5000 |
| 200 | 2011 | 15 | 9000 |

## Solution

```
WITH FirstProductSale AS

 ( SELECT product_id,  year,  quantity,  price,  RANK()

OVER (PARTITION BY product_id ORDER BY YEAR ASC) AS sale_rank

  FROM SALES)

SELECT product_id, year AS first_year,  quantity,  price  FROM FirstProductSale WHERE sale_rank = 1;
```

***Explanation:***

1. **CTE FirstProductSale**:
   - This CTE calculates the rank for each product's sales based on the year (ascending order), partitioned by product_id, which helps identify the first sale for each product.
2. **sale_rank**:
   - It represents the ranking of the sales for each product by year. The first sale of each product will have a sale_rank of 1.
3. **Final SELECT Statement**:
   - Filters out the first sale of each product (where sale_rank = 1) and returns the relevant columns (product_id, first_year, quantity, and price).

https://www.linkedin.com/in/niranjana405/