SQL Leet Code:

**) Replace Employee ID with Unique Identifier:**

**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, display null. Return the result table in any order.

**Example 1:**

**Input:**

Employees table:

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

| id | name |

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

| 1 | Alice |

| 7 | Bob |

| 11 | Meir |

| 90 | Winston |

| 3 | Jonathan |

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

EmployeeUNI table:

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

| id | unique\_id |

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

| 3 | 1 |

| 11 | 2 |

| 90 | 3 |

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

**Output:**

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

| unique\_id | name |

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

| null | Alice |

| null | Bob |

| 2 | Meir |

| 3 | Winston |

| 1 | Jonathan |

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

**Explanation:**

* Alice and Bob do not have a unique\_id, so we display null.
* The unique\_id of Meir is 2.
* The unique\_id of Winston is 3.
* The unique\_id of Jonathan is 1.

**Solution:**

SELECT eu.unique\_id,e.name

FROM Employees e

LEFT JOIN EmployeeUNI eu

ON e.id=eu.id;

**2) Top Travellers:**

**Table:** Users

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

| Column Name | Type |

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

| id | int |

| name | varchar |

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

id is the column with unique values for this table. name is the name of the user.

**Table:** Rides

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

| Column Name | Type |

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

| id | int |

| user\_id | int |

| distance | int |

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

id is the column with unique values for this table. user\_id is the id of the user who traveled the distance "distance".

Write a solution to report the distance traveled by each user. Return the result table ordered by travelled\_distance in descending order. If two or more users traveled the same distance, order them by their name in ascending order. The result format is in the following example.

**Example 1:**

**Input:**

Users table:

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

| id | name |

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

| 1 | Alice |

| 2 | Bob |

| 3 | Alex |

| 4 | Donald |

| 7 | Lee |

| 13 | Jonathan |

| 19 | Elvis |

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

Rides table:

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

| id | user\_id | distance |

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

| 1 | 1 | 120 |

| 2 | 2 | 317 |

| 3 | 3 | 222 |

| 4 | 7 | 100 |

| 5 | 13 | 312 |

| 6 | 19 | 50 |

| 7 | 7 | 120 |

| 8 | 19 | 400 |

| 9 | 7 | 230 |

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

**Output:**

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

| name | travelled\_distance |

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

| Elvis | 450 |

| Lee | 450 |

| Bob | 317 |

| Jonathan | 312 |

| Alex | 222 |

| Alice | 120 |

| Donald | 0 |

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

**Explanation:**

Elvis and Lee traveled 450 miles. Elvis is the top traveler as his name is alphabetically smaller than Lee. Bob, Jonathan, Alex, and Alice have only one ride and we just order them by the total distances of the ride. Donald did not have any rides, the distance traveled by him is 0.

**Solution:**

SELECT u.name,

NVL(SUM(r.distance), 0) AS travelled\_distance

FROM Users u

LEFT JOIN Rides r ON u.id = r.user\_id

GROUP BY u.name

ORDER BY travelled\_distance DESC, u.name ASC;

**3) 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. The sold products names for each date should be sorted lexicographically. Return the result table ordered by sell\_date. The result format is in the following example.

**Example 1:**

**Input:**

Activities table:

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

| sell\_date | product |

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

| 2020-05-30 | Headphone |

| 2020-06-01 | Pencil |

| 2020-06-02 | Mask |

| 2020-05-30 | Basketball |

| 2020-06-01 | Bible |

| 2020-06-02 | Mask |

| 2020-05-30 | T-Shirt |

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

**Output:**

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

| sell\_date | num\_sold | products |

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

| 2020-05-30 | 3 | Basketball,Headphone,T-Shirt |

| 2020-06-01 | 2 | Bible,Pencil |

| 2020-06-02 | 1 | Mask |

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

**Explanation:**

* For 2020-05-30, sold items were 1 (Basketball, Headphone, T-Shirt). We sort them lexicographically and separate them by a comma.
* For 2020-06-01, sold items were (Bible, Pencil). We sort them lexicographically and separate them by a comma.
* For 2020-06-02, the sold item is (Mask). We just return it.

Solution:

SELECT

sell\_date,

COUNT(DISTINCT product) AS num\_sold,

LISTAGG(DISTINCT product, ',') WITHIN GROUP (ORDER BY product) AS products

FROM Activities

GROUP BY sell\_date

ORDER BYsell\_date;

**4)**

**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 the 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.

**Example 1:**

**Input:**

Visits table:

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

| visit\_id | customer\_id |

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

| 1 | 23 |

| 2 | 9 |

| 4 | 30 |

| 5 | 54 |

| 6 | 96 |

| 7 | 54 |

| 8 | 54 |

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

Transactions table:

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

| transaction\_id | visit\_id | amount |

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

| 2 | 5 | 310 |

| 3 | 5 | 300 |

| 9 | 5 | 200 |

| 12 | 1 | 910 |

| 13 | 2 | 970 |

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

**Output:**

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

| customer\_id | count\_no\_trans |

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

| 54 | 2 |

| 30 | 1 |

| 96 | 1 |

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

**Explanation:**

* Customer with 1 customer\_id = 23 visited the mall once and made one transaction during the visit with visit\_id = 12.
* Customer with customer\_id = 9 visited the mall once and made one transaction during the visit with visit\_id = 13.
* Customer with customer\_id = 30 visited the mall once and did not make any transactions.
* Customer with customer\_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 customer\_id = 96 visited the mall once and did not make any transactions.

Solution:

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.transaction\_id IS NULL

GROUP BY v.customer\_id;

**5)**

**Table:** Users

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

| Column Name | Type |

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

| account | int |

| name | varchar |

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

account is the primary key (column with unique values) for this table. Each row of this table contains the account number of each user in the bank. There will be no two users having the same name in the table.

**Table:** Transactions

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

| Column Name | Type |

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

| trans\_id | int |

| account | int |

| amount | int |

| transacted\_on | date |

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

trans\_id is the primary key (column with unique values) for this table. Each row of this table contains all changes made to all accounts. amount is positive if the user received money and negative if they transferred money. All accounts start with a balance of 0.

Write a solution to report the name and balance of users with a balance higher than 10000. The balance of an account is equal to the sum of the amounts of all transactions involving that account. Return the result table in any order. The result format is in the following example.

**Example 1:**

**Input:**

Users table:

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

| account | name |

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

| 900001 | Alice |

| 900002 | Bob |

| 900003 | Charlie |

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

Transactions table:

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

| trans\_id | account | amount | transacted\_on |

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

| 1 | 900001 | 7000 | 2020-08-01 |

| 2 | 900001 | 7000 | 2020-09-01 |

| 3 | 900001 | -3000 | 2020-09-02 |

| 4 | 900002 | 1000 | 2020-09-12 |

| 5 | 900003 | 6000 | 2020-08-07 |

| 6 | 900003 | 6000 | 2020-09-07 |

| 7 | 900003 | -4000 | 2020-09-11 |

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

**Output:**

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

| name | balance |

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

| Alice | 11000 |

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

**Explanation:**

* Alice's balance is (7000 + 7000 - 3000) = 11000.
* Bob's balance is 1000.
* Charlie's balance is (6000 + 6000 - 4000) = 8000.

Solution:

SELECT u.name, SUM amount) AS balance

FROM Transactions t

INNER JOIN users u

ON t.account = u.account

GROUP BY u.name

HAVING SUM amount > 10000;

**6)**

**Table:** Tweets

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

| Column Name | Type |

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

| tweet\_id | int |

| content | varchar |

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

tweet\_id is the primary key (column with unique values) for this table. 1 content consists of alphanumeric characters, '!', or ' ' 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. The result format is in the following example.

**Example 1:**

**Input:**

Tweets table:

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

| tweet\_id | content |

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

| 1 | Let us Code |

| 2 | More than fifteen chars are here! |

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

**Output:**

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

| tweet\_id |

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

| 2 |

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

**Explanation:**

* Tweet 1 has length = 11. It is a valid tweet.
* Tweet 2 has length = 33. It is an invalid tweet.

Solution:

SELECT tweet\_id

FROM Tweets

WHERE LENGTH content > 15;

**7)**

**Table:** Employees

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

| Column Name | Type |

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

| emp\_id | int |

| event\_day | date |

| in\_time | int |

| out\_time | int |

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

(emp\_id, event\_day, in\_time) is the primary key 1 (combinations of columns with unique values) of this table. The table shows the employees' entries and exits in an office. event\_day is the day at which this event happened, in\_time is the minute at which the employee entered the office, and out\_time is the minute at which they left the office. in\_time and out\_time are between 1 and 1440. It is guaranteed that no two events on the same day intersect in time, and in\_time < out\_time.

Write a solution to calculate the total time (in minutes) spent by each employee on each day at the office. Note that within one day, an employee can enter and leave more than once. The time spent in the office for a single entry is out\_time - in\_time. Return the result table in any order. The result format is in the following example.

**Example 1:**

**Input:**

Employees table:

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

| emp\_id | event\_day | in\_time | out\_time |

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

| 1 | 2020-11-28 | 4 | 32 |

| 1 | 2020-11-28 | 55 | 200 |

| 1 | 2020-12-03 | 1 | 42 |

| 2 | 2020-11-28 | 3 | 33 |

| 2 | 2020-12-09 | 47 | 74 |

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

**Output:**

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

| day | emp\_id | total\_time |

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

| 2020-11-28 | 1 | 173 |

| 2020-11-28 | 2 | 30 |

| 2020-12-03 | 1 | 41 |

| 2020-12-09 | 2 | 27 |

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

**Explanation:**

* Employee 1 has three events: two on day 2020-11-28 with a total of (32 - 4) + (200 - 55) = 173, and one on day 2020-12-03 with a total of (42 - 1) = 41.
* Employee 2 has two events: one on day 2020-11-28 with a total of (33 - 3) = 30, and one on day 2020-12-09 with a total of (74 - 47) = 27.

Solution:

SELECT event\_day AS day, emp\_id, SUM out\_time\_in\_time AS total\_time

FROM Employees

GROUP BY event\_day, emp\_id;

**8)**

**Table:** Products

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

| Column Name | Type |

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

| product\_id | int |

| low\_fats | enum |

| recyclable | enum |

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

product\_id is the primary key (column with unique 1 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 |

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

| 0 | Y | N |

| 1 | Y | Y |

| 2 | N | Y |

| 3 | Y | Y |

| 4 | N | N |

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

**Output:**

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

| product\_id |

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

| 1 |

| 3 |

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

**Explanation:**

Only products 1 and 3 are both low fat and recyclable.

Solution:

SELECT product\_id

FROM Products

WHERE low\_fats = 'Y' AND recyclable = 'Y';

**10)**

**Table:** Employees

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

| Column Name | Type |

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

| employee\_id | int |

| name | varchar |

| salary | int |

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

employee\_id is the primary key (column with unique values) for this table. Each row of this table indicates the employee ID, employee name, and salary.

Write a solution to calculate the bonus of each employee. The bonus of an employee is 100% of their salary if the ID of the employee is an odd number and the employee's name does not start with the character 'M'. The bonus of an employee is 0 otherwise. Return the result table ordered by employee\_id. The result format is in the following example.

**Example 1:**

**Input:**

Employees table:

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

| employee\_id | name | salary |

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

| 2 | Meir | 3000 |

| 3 | Michael | 3800 |

| 7 | Addilyn | 7400 |

| 8 | Juan | 6100 |

| 9 | Kannon | 7700 |

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

**Output:**

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

| employee\_id | bonus |

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

| 2 | 0 |

| 3 | 0 |

| 7 | 7400 |

| 8 | 0 |

| 9 | 7700 |

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

**Explanation:**

* The employees with IDs 2 and 8 get 0 bonus because they have an even employee\_id.
* The employee with ID 3 1 gets 0 bonus because their name starts with 'M'.
* The rest of the employees get a 100% bonus.

Solution:

SELECT employee\_id, CASE WHEN employee\_id % 2 <> 0 AND name NOT LIKE 'M%' THEN salary

ELSE O END AS bonus

FROM Employees

ORDER BY employee\_id;

**11)**

**Table:** Logins

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

| Column Name | Type |

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

| user\_id | int |

| time\_stamp | datetime |

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

(user\_id, time\_stamp) is the primary key (combination of columns with unique values) for this table. Each row contains information about the login time for the user with ID user\_id.

Write a solution to report the latest login for all users in the year 2020. Do not include the users who did not login in 2020. Return the result table in any order. The result format is in the following example.

**Example 1:**

**Input:**

Logins table:

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

| user\_id | time\_stamp |

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

| 6 | 2020-06-30 15:06:07 |

| 6 | 2021-04-21 14:06:06 |

| 6 | 2019-03-07 00:18:15 |

| 8 | 2020-02-01 05:10:53 |

| 8 | 2020-12-30 00:46:50 |

| 2 | 2020-01-16 02:49:50 |

| 2 | 2019-08-25 07:59:08 |

| 14 | 2019-07-14 09:00:00 |

| 14 | 2021-01-06 11:59:59 |

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

**Output:**

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

| user\_id | last\_stamp |

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

| 6 | 2020-06-30 15:06:07 |

| 8 | 2020-12-30 00:46:50 |

| 2 | 2020-01-16 02:49:50 |

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

**Explanation:**

* User 6 logged into their account 3 times but only once in 2020, so we include this login in the result table.
* User 8 logged into their account 2 times in 2020, once in February and once in December. We include only the latest one (December) in the result table.
* User 2 logged into their account 2 times but only once in 2020, so we include this login in the result table.
* User 14 did not login in 2020, so we do not include them in the result table.

Solution:

SELECT DISTINCT user\_id, FIRST\_VALUE time\_stamp OVER PARTITION BY user\_id ORDER BY time\_stamp DESC ASlast\_stamp

FROM Logins

WHERE YEAR time\_stamp = ‘2020';

**12)**

**Table:** Employees

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

| Column Name | Type |

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

| employee\_id | int |

| name | varchar |

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

employee\_id is the column with unique values for this table. Each row of this table indicates the name of the employee whose ID is employee\_id.

**Table:** Salaries

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

| Column Name | Type |

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

| employee\_id | int |

| salary | int |

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

employee\_id is the column with unique values for this table. Each row of this table indicates the salary of the employee whose ID is employee\_id.

Write a solution to report the IDs of all the employees with missing information. The information of an employee is missing if:

* The employee's name is missing, or
* The employee's salary is missing.

Return the result table ordered by employee\_id in ascending order. The result format is in the following example.

**Example 1:**

**Input:**

Employees table:

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

| employee\_id | name |

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

| 2 | Crew |

| 4 | Haven |

| 5 | Kristian |

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

Salaries table:

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

| employee\_id | salary |

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

| 5 | 76071 |

| 1 | 22517 |

| 4 | 63539 |

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

**Output:**

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

| employee\_id |

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

| 1 |

| 2 |

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

**Explanation:**

Employees 1, 2, 4, and 5 are working at this company.

* The name of employee 1 is missing.
* The salary of employee 2 1 is missing.

Solution:

SELECT e.employee\_id

FROM Employees e

LEFT JOIN Salaries s

ON e.employee\_id = s.employee\_id

WHERE s.salary IS NULL

UNION

SELECT s.employee\_idFROM Salaries s

LEFT JOIN Employees e

ON s.employee\_id = e.employee\_id

WHERE e.name IS NULL

ORDER BY employee\_id;

**13)**

**Table:** Stocks

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

| Column Name | Type |

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

| stock\_name | varchar |

| operation | enum |

| operation\_day | int |

| price | int |

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

(stock\_name, operation\_day) is the primary key (combination of columns with unique values) for this table. The operation column is an ENUM (category) of type ('Sell', 'Buy'). Each row of this table indicates that the stock which has stock\_name had an operation on the day operation\_day with the price. It is guaranteed that each 'Sell' operation for a stock has a corresponding 'Buy' operation in a previous day. It is also guaranteed that each 'Buy' operation for a stock has a corresponding 'Sell' operation in an upcoming day.

Write a solution to report the Capital gain/loss for each stock. The Capital gain/loss of a stock is the total gain or loss after buying and selling the stock one or many times. Return the result table in any order. The result format is in the following example.

**Example 1:**

**Input:**

Stocks table:

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

| stock\_name | operation | operation\_day | price |

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

| Leetcode | Buy | 1 | 1000 |

| Corona Masks | Buy | 2 | 10 |

| Leetcode | Sell | 5 | 9000 |

| Handbags | Buy | 17 | 30000 |

| Corona Masks | Sell | 3 | 1010 |

| Corona Masks | Buy | 4 | 1000 |

| Corona Masks | Sell | 5 | 500 |

| Corona Masks | Buy | 6 | 1000 |

| Handbags | Sell | 29 | 7000 |

| Corona Masks | Sell | 10 | 10000 |

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

**Output:**

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

| stock\_name | capital\_gain\_loss |

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

| Corona Masks | 9500 |

| Leetcode | 8000 |

| Handbags | -23000 |

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

**Explanation:**

* Leetcode stock was bought at day 1 for 1000$ and was sold at day 5 for 9000$. Capital gain = 9000 - 1000 = 8000$.
* Handbags stock was bought at day 17 for 30000$ and was sold at day 29 for 7000$. Capital loss = 7000 - 30000 = -23000$.
* Corona Masks stock was bought at day 1 for 10$ and was sold at day 3 for 1010$. It was bought again at day 4 for 1000$ and was sold at day 5 for 500$. At last, it was bought at day 6 for 1000$ and was sold at day 10 for 10000$. Capital gain/loss is the sum of capital gains/losses for each ('Buy' --> 'Sell') operation = (1010 - 10) + (500 - 1000) + (10000 - 1000) = 1000 - 500 + 9000 = 9500$.

Solution:

SELECT stock\_name, SUM CASE WHEN operation = 'Buy' THEN price\*-1 ELSE price END AS capital\_gain\_loss

FROM Stocks

GROUP BY stock\_name;

**14)**

**Table:** Products

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

| Column Name | Type |

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

| product\_id | int |

| store1 | int |

| store2 | int |

| store3 | int |

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

product\_id is the primary key (column with unique values) for this table. Each row in this table indicates the product's price in 3 different stores: store1, store2, and store3. If the product is not available in a store, the price will be null in that store's column.

Write a solution to rearrange the Products table so that each row has (product\_id, store, price). If a product is not available in a store, do not include a row with that product\_id and store combination in the result table. Return the result table in any order. The result format is in the following example.

**Example 1:**

**Input:**

Products table:

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

| product\_id | store1 | store2 | store3 |

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

| 0 | 95 | 100 | 105 |

| 1 | 70 | null | 80 |

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

**Output:**

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

| product\_id | store | price |

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

| 0 | store1 | 95 |

| 0 | store2 | 100 |

| 0 | store3 | 105 |

| 1 | store1 | 70 |

| 1 | store3 | 80 |

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

**Explanation:**

* Product 0 is available in all three stores with prices 95, 100, and 105 respectively.
* Product 1 is available in store1 with price 70 and store3 with price 80. The product is not available in store2

Solution:

SELECT product\_id, 'storel' AS store, store1 AS price FROM Products

WHERE store1 IS NOT NULL

UNION

SELECT product\_id, 'store2' AS store, storez AS price FROM Products

WHERE store2 IS NOT NULL

UNION

SELECT product\_id, 'store3' AS store, store3 AS price FROM Products

WHERE store3 IS NOT NULL;

**15)**

**Table:** NPV

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

| Column Name | Type |

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

| id | int |

| year | int |

| npv | int |

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

(id, year) is the primary key of this table. This table contains the ID and the year of the investment and its net present value.

**Table:** Queries

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

| Column Name | Type |

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

| id | int |

| year | int |

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

(id, year) is the primary key of this table. This table contains the ID and the year of the investment.

Write an SQL query to find the npv of each query. Return the result table in any order. The result format is in the following example.

**Example 1:**

**Input:**

NPV table:

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

| id | year | npv |

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

| 1 | 2018 | 100 |

| 1 | 2020 | 100 |

| 2 | 2020 | 200 |

| 3 | 2019 | 200 |

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

Queries table:

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

| id | year |

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

| 1 | 2018 |

| 2 | 2020 |

| 3 | 2019 |

| 1 | 2021 |

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

**Output:**

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

| id | year | npv |

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

| 1 | 2018 | 100 |

| 2 | 2020 | 200 |

| 3 | 2019 | 200 |

| 1 | 2021 | 0 |

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

**Explanation:**

The npv of the query (1, 2021) is not in the NPV table, so its npv is 0.

Solution:

SELECT q.id, q.year, NVL(n.npv, 0) AS npv

FROM Queries q

LEFT JOIN

NPV n ON q.id = n.id AND q.year = n.year;

**16) Average Selling Price**

**Table:** Products

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

| Column Name | Type |

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

| product\_id | int |

| product\_name | varchar |

| unit\_price | int |

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

product\_id is the primary key of this table. Each row of this table indicates the name and the price of each product.

**Table:** Orders

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

| Column Name | Type |

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

| product\_id | int |

| quantity | int |

| order\_date | date |

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

(product\_id, order\_date) is the primary key of this table. Each row of this table indicates the quantity of product\_id was ordered on order\_date.

Write an SQL query to find the average selling price for each product. The average\_price should be rounded to 2 decimal places. Return the result table in any order.

The result format is in the following example.

**Example 1:**

**Input:**

Products table:

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

| product\_id | product\_name | unit\_price |

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

| 1 | LC Phone | 300 |

| 2 | LC T-Shirt | 10 |

| 3 | LC Book | 20 |

| 4 | LC Mug | 25 |

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

Orders table:

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

| product\_id | quantity | order\_date |

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

| 1 | 2 | 2020-02-05 |

| 1 | 1 | 2020-02-10 |

| 2 | 3 | 2020-02-10 |

| 3 | 1 | 2020-02-15 |

| 1 | 5 | 2020-02-25 |

| 4 | 6 | 2020-02-25 |

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

**Output:**

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

| product\_id | average\_price |

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

| 1 | 300.00 |

| 2 | 10.00 |

| 3 | 20.00 |

| 4 | 25.00 |

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

**Explanation:**

* Average selling price for product 1 = (300 \* 2 + 300 \* 1 + 300 \* 5) / 8 = 300.00
* Average selling price for product 2 = (10 \* 3) / 3 = 10.00
* Average selling price for product 3 = (20 \* 1) / 1 = 20.00
* Average selling price for product 4 = (25 \* 6) / 6 = 25.00

Solution:

SELECT p.product\_id,

ROUND(SUM(p.unit\_price \* o.quantity) / SUM(o.quantity), 2) AS average\_price

FROM Products p

JOIN

Orders o ON p.product\_id = o.product\_id

GROUP BY p.product\_id;

**17) Customer Visits and Purchases**

**Table:** Orders

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

| Column Name | Type |

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

| order\_id | int |

| order\_date | date |

| customer\_id | int |

| product\_id | int |

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

order\_id is the primary key for this table. This table contains 1 the ID of an order, the date of the order, the ID of the customer who ordered it, and the ID of the product which they ordered.

[1. github.com](https://github.com/AbhaySingh5349/Leetcode-Premium" \t "_blank)

[github.com](https://github.com/AbhaySingh5349/Leetcode-Premium" \t "_blank)

Write an SQL query to find the number of times each customer visited the mall and the number of products each customer bought. Return the result table in any order.

The result format is in the following example.

**Example 1:**

**Input:**

Orders table:

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

| order\_id | order\_date | customer\_id | product\_id |

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

| 1 | 2019-01-01 | 1 | 1 |

| 2 | 2019-01-01 | 2 | 2 |

| 3 | 2019-01-01 | 1 | 3 |

| 4 | 2019-01-01 | 1 | 4 |

| 5 | 2019-01-02 | 2 | 5 |

| 6 | 2019-01-02 | 3 | 6 |

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

**Output:**

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

| customer\_id | count\_visits | count\_products |

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

| 1 | 2 | 3 |

| 2 | 2 | 2 |

| 3 | 1 | 1 |

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

**Explanation:**

* Customer 1 visited the mall twice and bought 3 products.
* Customer 2 visited the mall twice and bought 2 products.
* Customer 3 visited the mall once and bought 1 product.

Solution:

SELECT customer\_id, COUNT(DISTINCT order\_date) AS count\_visits,

COUNT(product\_id) AS count\_products

FROM Orders

GROUP BY customer\_id

ORDER BY customer\_id;

**18) Most Recent Orders for Each Product**

**Table:** Products

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

| Column Name | Type |

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

| product\_id | int |

| product\_name | varchar |

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

product\_id is the primary key of this table. Each row of this table indicates the name of each product.

**Table:** Orders

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

| Column Name | Type |

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

| product\_id | int |

| order\_date | date |

| unit | int |

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

(product\_id, order\_date) is the primary key of this table. Each row of this table indicates the quantity of each product was ordered on each date.

Write an SQL query to find the most recent order(s) of each product. Return the result table ordered by product\_name in ascending order and order\_date in ascending order.

The result format is in the following example.

**Example 1:**

**Input:**

Products table:

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

| product\_id | product\_name |

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

| 1 | Avocado |

| 2 | Blueberry |

| 3 | Almond |

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

Orders table:

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

| product\_id | order\_date | unit |

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

| 1 | 2020-01-02 | 800 |

| 1 | 2020-01-15 | 900 |

| 2 | 2020-02-10 | 600 |

| 2 | 2020-03-01 | 700 |

| 3 | 2020-01-07 | 750 |

| 3 | 2020-01-30 | 800 |

| 3 | 2020-03-15 | 1000 |

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

**Output:**

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

| product\_name | order\_date | unit |

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

| Almond | 2020-03-15 | 1000 |

| Avocado | 2020-01-15 | 900 |

| Blueberry | 2020-03-01 | 700 |

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

Solution:

SELECT w.name AS warehouse\_name,

SUM w.units\*p.Width\*p.Length\*p.Height AS volume

FROM Warehouse w

INNER JOIN Products p

ON w.product\_id = p.product\_id

GROUP BY w.name;

OK, here's the formatted and more readable version of the LeetCode problem description:

**19) Machines That Process a Task**

**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 production machine.

* machine\_id is the primary key of this table.
* process\_id is the ID of the process run on the machine.
* activity\_type is an ENUM 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 is always less than the 'end' timestamp for every (machine\_id, process\_id) combination.

There is a factory production machine that might run various number of processes. Each process is assigned a unique process\_id, and each process may be executed many times.

Write an SQL query 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.

The result should be in ascending order by machine\_id. Round the average to 3 decimal places.

The result format is in the following example.

**Example 1:**

**Input:**

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 | 4.520 |

| 2 | 1 | start | 2.500 |

| 2 | 1 | end | 5.000 |

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

**Output:**

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

| machine\_id | processing\_time |

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

| 0 | 0.894 |

| 1 | 0.995 |

| 2 | 2.710 |

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

**Explanation:**

* Machine 0:
  + process 0: 1.520 - 0.712 = 0.808
  + process 1: 4.120 - 3.140 = 0.980
  + average: (0.808 + 0.980) / 2 = 0.894
* Machine 1:
  + process 0: 1.550 - 0.550 = 1.000
  + process 1: 1.420 - 0.430 = 0.990
  + average: (1.000 + 0.990) / 2 = 0.995
* Machine 2:
  + process 0: 4.520 - 4.100 = 0.420
  + process 1: 5.000 - 2.500 = 2.500
  + average: (0.420 + 2.500) / 2 = 2.710

Solution:

SELECT problem\_id

FROM Problems

WHERE likes / likes+dislikes < 0.6

ORDER BY problem\_id;

**20) Number of Products Ordered in the Period**

**Table:** Products

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

| Column Name | Type |

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

| product\_id | int |

| product\_name | varchar |

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

product\_id is the primary key of this table. Each row of this table indicates the name of each product.

**Table:** Orders

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

| Column Name | Type |

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

| product\_id | int |

| order\_date | date |

| unit | int |

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

(product\_id, order\_date) is the primary key of this table. Each row of this table indicates the quantity of each product was ordered on each date.

Write an SQL query to find the number of products that were ordered in the year 2020. Return the result table in any order.

The result format is in the following example.

**Example 1:**

**Input:**

Products table:

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

| product\_id | product\_name |

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

| 1 | Leetcode Mobile|

| 2 | Google Pixel |

| 3 | Samsung Galaxy |

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

Orders table:

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

| product\_id | order\_date | unit |

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

| 1 | 2020-02-05 | 60 |

| 2 | 2020-03-01 | 70 |

| 3 | 2020-04-18 | 80 |

| 1 | 2020-02-09 | 100 |

| 2 | 2020-06-30 | 90 |

| 3 | 2020-12-25 | 200 |

| 1 | 2019-01-01 | 50 |

| 2 | 2019-02-02 | 60 |

| 3 | 2019-03-03 | 70 |

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

**Output:**

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

| products\_count|

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

| 3 |

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

**Explanation:**

All products were ordered in the year 2020.

Solution:

SELECT COUNT(DISTINCT product\_id) AS products\_count

FROM Orders

WHERE EXTRACT(YEAR FROM order\_date) = 2020;

**21) Daily Active Users II**

**Table:** Activities

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

| Column Name | Type |

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

| activity\_date | date |

| user\_id | int |

| activity\_type | enum |

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

There is no primary key for this table; it may have duplicate rows. The activity\_type column is an ENUM of type ('open\_session', 'end\_session', 'scroll\_down', 'send\_message'). The table shows the user activities for a social media website.

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

Return the result table in any order.

The query result format is in the following example.

**Example 1:**

**Input:**

Activities table:

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

| activity\_date | user\_id | activity\_type |

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

| 2019-07-27 | 1 | open\_session |

| 2019-07-27 | 2 | open\_session |

| 2019-07-25 | 1 | end\_session |

| 2019-07-25 | 2 | scroll\_down |

| 2019-07-25 | 1 | send\_message |

| 2019-07-25 | 3 | open\_session |

| 2019-07-25 | 3 | end\_session |

| 2019-07-24 | 1 | open\_session |

| 2019-07-24 | 1 | end\_session |

| 2019-07-24 | 2 | scroll\_down |

| 2019-07-23 | 1 | send\_message |

| 2019-07-23 | 2 | open\_session |

| 2019-07-23 | 3 | scroll\_down |

| 2019-07-23 | 3 | send\_message |

| 2019-06-23 | 4 | open\_session |

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

**Output:**

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

| activity\_date | user\_count |

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

| 2019-07-27 | 2 |

| 2019-07-25 | 3 |

| 2019-07-24 | 2 |

| 2019-07-23 | 3 |

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

**Explanation:**

Note that we only care about dates with activity in the range between 2019-06-28 and 2019-07-27 inclusive.

* On 2019-07-27, users 1 and 2 were active.
* On 2019-07-25, users 1, 2, and 3 were active.
* On 2019-07-24, users 1 and 2 were active.
* On 2019-07-23, users 1, 2, and 3 were active.

Solution:

SELECT activity\_date, COUNT(DISTINCT user\_id) AS user\_count

FROM Activities

WHERE activity\_date BETWEEN ADD\_MONTHS(DATE '2019-07-27', -1) + 1 AND DATE '2019-07-27'

GROUP BY activity\_date;

**22) Customer Who Placed the Largest Number of Orders**

**Table:** Orders

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

| Column Name | Type |

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

| order\_id | int |

| customer\_name | varchar |

| customer\_id | int |

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

order\_id is the primary key for this table. customer\_name is the name of the customer who placed the order. customer\_id is the ID of the customer who placed the order.

**Table:** Customers

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

| Column Name | Type |

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

| customer\_id | int |

| customer\_name | varchar |

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

customer\_id is the primary key for this table. customer\_name is the name of the customer.

Write an SQL query to find the name of the customer who has placed the most orders.

Return the result table in any order.

The result format is in the following example.

**Example 1:**

**Input:**

Orders table:

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

| order\_id | customer\_name | customer\_id |

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

| 1 | Diana | 1 |

| 2 | Diana | 1 |

| 3 | Nicholas | 2 |

| 4 | Diana | 1 |

| 5 | Harrison | 3 |

| 6 | Alice | 4 |

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

Customers table:

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

| customer\_id | customer\_name |

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

| 1 | Diana |

| 2 | Nicholas |

| 3 | Harrison |

| 4 | Alice |

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

**Output:**

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

| customer\_name |

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

| Diana |

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

**Explanation:**

Diana has placed 3 orders, which is the most orders any customer has placed.

Solution:

SELECT customer\_name

FROM ( SELECT customer\_name,

COUNT(\*) AS order\_count

FROM Orders

GROUP BY customer\_name

ORDER BY COUNT(\*) DESC)

WHERE ROWNUM = 1;

**23) Activity With the Most Users**

**Table:** Friends

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

| Column Name | Type |

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

| id | int |

| name | varchar |

| activity | varchar |

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

id is the primary key for this table. name is the name of the user. activity is the name of the activity the user likes to do.

**Table:** Activities

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

| Column Name | Type |

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

| activity | varchar |

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

activity is the primary key for this table. activity is the name of some activity.

Write an SQL query to find the activity name with the maximum number of distinct users that like it.

Return the result table in any order.

The result format is in the following example.

**Example 1:**

**Input:**

Friends table:

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

| id | name | activity |

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

| 1 | Jonathan | Hiking |

| 2 | Amy | Hiking |

| 3 | Drew | Swimming |

| 4 | Alice | Swimming |

| 5 | Daniel | Hiking |

| 6 | Helen | Running |

| 7 | Jonathan | Running |

| 8 | Lisa | Running |

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

Activities table:

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

| activity |

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

| Hiking |

| Swimming |

| Running |

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

**Output:**

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

| activity |

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

| Hiking |

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

**Explanation:**

There are 3 distinct users who like "Hiking", 2 distinct users who like "Swimming", and 3 distinct users who like "Running".

Since "Hiking" and "Running" have the same maximum number of distinct users, we return "Hiking" as it has the smallest lexicographical order.

Solution:

SELECT activity

FROM (

SELECT activity,

COUNT(DISTINCT id) AS user\_count

FROM Friends

GROUP BY activity

ORDER BY COUNT(DISTINCT id) DESC, activity AS)

WHERE ROWNUM = 1;

**24) Department Total Revenue**

**Table:** Departments

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

| Column Name | Type |

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

| id | int |

| name | varchar |

| revenue | int |

| month | varchar |

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

(id, month) is the primary key of this table. The month column has values in the format "YYYY-MM".

Write an SQL query to find the total revenue for each department in each year.

Return the result table in any order.

The result format is in the following example.

**Example 1:**

**Input:**

Departments table:

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

| id | name | revenue | month |

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

| 1 | A | 7000 | 2018-01 |

| 2 | A | 6500 | 2018-02 |

| 3 | A | 8000 | 2018-03 |

| 4 | B | 5000 | 2018-01 |

| 5 | B | 6000 | 2018-02 |

| 6 | C | 4000 | 2018-01 |

| 7 | A | 9000 | 2019-01 |

| 8 | A | 7000 | 2019-02 |

| 9 | B | 6000 | 2019-03 |

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

**Output:**

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

| dept\_name | year | total\_revenue |

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

| A | 2018 | 21500 |

| A | 2019 | 16000 |

| B | 2018 | 11000 |

| B | 2019 | 6000 |

| C | 2018 | 4000 |

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

**Explanation:**

The total revenue for department 'A' in year 2018 is 7000 + 6500 + 8000 = 21500. The total revenue for department 'A' in year 2019 is 9000 + 7000 = 16000. The total revenue for department 'B' in year 2018 is 5000 + 6000 = 11000. The total revenue for department 'B' in year 2019 is 6000. The total revenue for department 'C' in year 2018 is 4000.

Solution:

SELECT name AS dept\_name,

EXTRACT(YEAR FROM TO\_DATE(month, 'YYYY-MM')) AS year,

SUM(revenue) AS total\_revenue

FROM Departments

GROUP BY name, EXTRACT(YEAR FROM TO\_DATE(month, 'YYYY-MM'))

ORDER BY name, year;

**25) Consecutive Numbers**

**Table:** Logs

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

| Column Name | Type |

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

| log\_id | int |

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

log\_id is the primary key for this table. Each row of this table contains the ID of a log from a certain database. Since some of the log\_id's were removed from the database, you have missing log\_id's. Consecutive log\_id's are those log\_id's with the difference of 1 between them.

Write an SQL query to find the start and end number of continuous ranges in the table Logs.

Return the result table ordered by start\_id.

The result format is in the following example.

**Example 1:**

**Input:**

Logs table:

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

| log\_id |

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

| 1 |

| 2 |

| 3 |

| 7 |

| 8 |

| 10 |

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

**Output:**

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

| start\_id | end\_id |

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

| 1 | 3 |

| 7 | 8 |

| 10 | 10 |

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

**Explanation:**

The 1 ranges are:

[1. github.com](https://github.com/Icewell2/Leetcode" \t "_blank)

[github.com](https://github.com/Icewell2/Leetcode" \t "_blank)

* 1, 2, 3
* 7, 8
* 10, 10

Solution:

SELECT MIN(log\_id) AS start\_id, MAX(log\_id) AS end\_id

FROM (

SELECT log\_id, log\_id - ROWNUM AS grp

FROM (

SELECT log\_id

FROM Logs

ORDER BY log\_id

)

)

GROUP BY grp

ORDER BY start\_id;

**26) Customers With Positive Revenue in the Year 2021**

**Table:** Customer

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

| Column Name | Type |

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

| customer\_id | int |

| year | int |

| revenue | int |

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

(customer\_id, year) is the primary key for this table. There are no nulls in any column. The revenue column gives the revenue of the customer with customer\_id in the year year.

Write an SQL query to report the customer\_id from the Customer table that had positive revenue in the year 2021.

**Example 1:**

**Input:**

Customer table:

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

| customer\_id | year | revenue |

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

| 1 | 2021 | 50 |

| 1 | 2022 | 100 |

| 2 | 2021 | 0 |

| 3 | 2021 | -50 |

| 3 | 2022 | 100 |

| 4 | 2021 | 200 |

| 4 | 2022 | -200 |

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

**Output:**

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

| customer\_id |

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

| 1 |

| 4 |

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

**Explanation:**

Customer 1 and 4 had positive revenue in 2021.

Solution:

SELECT customer\_id

FROM Customer

WHERE year = 2021 AND revenue > 0;