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.

**Input: Output**:

Products table:

|  |  |  |
| --- | --- | --- |
| **product\_id** | **low\_fats** | **recyclable** |
| 0 | Y | N |
| 1 | Y | Y |
| 2 | N | Y |
| 3 | Y | Y |
| 4 | N | N |

|  |
| --- |
| product\_id |
| 1 |
| 3 |

**Solution:**

***MYSQL Server***

SELECT product\_id

FROM Products

WHERE low\_fats = ‘Y’ AND recyclable = ‘Y’’

***Pandas***

import pandas as pd

df = pd.read\_csv('path\_to\_your\_file.csv')

*# Filter the DataFrame for products that are both low fat and recyclable*

filtered\_df = df[(df['low\_fats'] == 'Y') & (df['recyclable'] == 'Y')]

*# Select the product\_id column*

result = filtered\_df['product\_id']

print(resu

Find Customer Referee

Table: Customer

|  |  |
| --- | --- |
| Column Name | Type |
| id | int |
| name | varchar |
| referee\_id | int |

* 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.

Input: Output:

Customer table:

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

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

**Solution:**

***MYSQL Server***

SELECT name

FROM Customer

WHERE referee\_id is null or referee\_id <> '2';

***Pandas***

import pandas as pd

df = pd.read\_csv('path\_to\_your\_file.csv')

*# Filter the DataFrame for customers not referred by the customer with id = 2*

filtered\_df = df[(df['referee\_id'].isnull()) | (df['referee\_id'] != 2)]

*# Select the name column*

result = filtered\_df['name']

print(result)

Big Countries

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.

Input:

World table:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Input | | | | | Output | | |
| name | continent | area | population | gdp | name | population | area |
| Afghanistan | Asia | 652230 | 25500100 | 20343000000 | Afghanistan | 25500100 | 652230 |
| Albania | Europe | 28748 | 2831741 | 12960000000 | Algeria | 37100000 | 2381741 |
| Algeria | Africa | 2381741 | 37100000 | 188681000000 |  |  |  |
| Andorra | Europe | 468 | 78115 | 3712000000 |  |  |  |
| Angola | Africa | 1246700 | 20609294 | 1.0099E+11 |  |  |  |

/\* Write your T-SQL query statement below \*/

select name, population, area

from World

where area >= '3000000' or population >= '25000000';

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.
* 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.

The result format is in the following example.

Input:

Tweets table:

|  |  |  |  |
| --- | --- | --- | --- |
| Input | |  | Output |
| tweet\_id | content | tweet\_id |
| 1 | Let us Code | 2 |
| 2 | More than fifteen chars are here! |  |

Explanation:

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

SELECT tweet\_id

FROM Tweets

WHERE LEN(content) > 15;

Article Views

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.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Input: Views table | | | |  | Output |
| article\_id | author\_id | viewer\_id | view\_date | id |
| 1 | 3 | 5 | 2019-08-01 | 4 |
| 1 | 3 | 6 | 2019-08-02 | 7 |
| 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 |

select distinct author\_id as id

from views

where author\_id = viewer\_id

order by author\_id ASC;

Product Sales Analysis

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.

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.

The result format is in the following example.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Input : Sales table | | | | |  | Input : Product table | |
| sale\_id | product\_id | year | quantity | price | product\_id | product\_name |
| 1 | 100 | 2008 | 10 | 5000 | 100 | Nokia |
| 2 | 100 | 2009 | 12 | 5000 | 200 | Apple |
| 7 | 200 | 2011 | 15 | 9000 | 300 | Samsung |

Explanation:

* From sale\_id = 1, we can conclude that Nokia was sold for 5000 in the year 2008.
* From sale\_id = 2, we can conclude that Nokia was sold for 5000 in the year 2009.
* From sale\_id = 7, we can conclude that Apple was sold for 9000 in the year 2011.

|  |  |  |
| --- | --- | --- |
| Output | | |
| product\_name | year | price |
| Nokia | 2008 | 5000 |
| Nokia | 2009 | 5000 |
| Apple | 2011 | 9000 |

/\* Write your T-SQL query statement below \*/

select p.product\_name, s.year, s.price

from Sales s

join Product p

on s.product\_id = p.product\_id;

Replace Employee ID With The 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 replace just show null. Return the result table in any order.

The result format is in the following example.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input: Employees table | |  | Input: EmployeeUNI table | |
| id | name | id | unique\_id |
| 1 | Alice | 3 | 1 |
| 7 | Bob | 11 | 2 |
| 11 | Meir | 90 | 3 |
| 90 | Winston |  |  |
| 3 | Jonathan |  |  |

Explanation:

* Alice and Bob do not have a unique ID, We will show null instead.
* The unique ID of Meir is 2.
* The unique ID of Winston is 3.
* The unique ID of Jonathan is 1.

|  |  |
| --- | --- |
| Output | |
| unique\_id | name |
| null | Alice |
| null | Bob |
| 2 | Meir |
| 3 | Winston |
| 1 | Jonathan |

select eu.unique\_id, e.name

from Employees e

left join EmployeeUNI eu

on e.id = eu.id;

Customer Who Visited but Did Not Make Any Transactions

|  |  |
| --- | --- |
| 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.

|  |  |
| --- | --- |
| Input: Visit | |
| visit\_id | customer\_id |
| 1 | 23 |
| 2 | 9 |
| 4 | 30 |
| 5 | 54 |
| 6 | 96 |
| 7 | 54 |
| 8 | 54 |

|  |  |  |
| --- | --- | --- |
| Input: Transactions | | |
| transaction\_id | visit\_id | amount |
| 2 | 5 | 310 |
| 3 | 5 | 300 |
| 9 | 5 | 200 |
| 12 | 1 | 910 |
| 13 | 2 | 970 |

* 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.

|  |  |
| --- | --- |
| Output: | |
| customer\_id | count\_no\_trans |
| 54 | 2 |
| 30 | 1 |
| 96 | 1 |

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;

Table: Weather

Rising Temperature

|  |  |
| --- | --- |
| 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.

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).

|  |  |  |
| --- | --- | --- |
| Input: 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 |

|  |
| --- |
| Output: |
| id |
|  |
| 2 |
| 4 |

SELECT w1.id

FROM Weather w1

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

WHERE w1.temperature > w2.temperature;

Average Time of Process per Machine

|  |  |
| --- | --- |
| Table: Activity | |
| Column Name | Type |
| machine\_id | int |
| process\_id | int |
| activity\_type | enum |
| timestamp | float |

* The table shows the user activities for a factory website.
* (machine\_id, process\_id, activity\_type) is the primary key (combination of columns with unique values) of this table.
* machine\_id is the ID of a machine.
* process\_id is the ID of a process running on the machine with ID machine\_id.
* activity\_type is an ENUM (category) of type ('start', 'end').
* timestamp is a float representing the current time in seconds.
* 'start' means the machine starts the process at the given timestamp and 'end' means the machine ends the process at the given timestamp.
* The 'start' timestamp will always be before the 'end' timestamp for every (machine\_id, process\_id) pair.
* It is guaranteed that each (machine\_id, process\_id) pair has a 'start' and 'end' timestamp.

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.

The result format is in the following example.

|  |  |  |  |
| --- | --- | --- | --- |
| Input: Activity table | | | |
| 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 |
| 1 | 1 | start | 0.43 |
| 1 | 1 | end | 1.42 |
| 2 | 0 | start | 4.1 |
| 2 | 0 | end | 4.512 |
| 2 | 1 | start | 2.5 |
| 2 | 1 | end | 5 |

|  |  |
| --- | --- |
| Output: | |
| machine\_id | processing\_time |
| 0 | 0.894 |
| 1 | 0.995 |
| 2 | 1.456 |

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

WITH ProcessTimes AS (

    SELECT

        machine\_id,

        process\_id,

        MAX(CASE WHEN activity\_type = 'end' THEN timestamp ELSE NULL END) -

        MIN(CASE WHEN activity\_type = 'start' THEN timestamp ELSE NULL END) AS process\_time

    FROM Activity

    GROUP BY machine\_id, process\_id

)

SELECT

    machine\_id,

    ROUND(AVG(process\_time), 3) AS processing\_time

FROM ProcessTimes

GROUP BY machine\_id;

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 |
| subject\_name | varchar |

* 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.

|  |  |
| --- | --- |
| Input: Examinations table | |
| 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 |

The result format is in the following example.

|  |  |
| --- | --- |
| Input: Students table | |
| student\_id | student\_name |
| 1 | Alice |
| 2 | Bob |
| 13 | John |
| 6 | Alex |

|  |
| --- |
| Input: Subjects table |
| subject\_name |
| Math |
| Physics |
| Programming |

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

Explanation:

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.

SELECT

    s.student\_id,

    s.student\_name,

    sub.subject\_name,

    COUNT(e.student\_id) 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;

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 |

* 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.

The result format is in the following example.

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

|  |  |
| --- | --- |
| Input: Bonus table | |
| empId | bonus |
| 2 | 500 |
| 4 | 2000 |

|  |  |
| --- | --- |
| Output: | |
| name | bonus |
| Brad | null |
| John | null |
| Dan | 500 |

SELECT e.name, b.bonus

FROM Employee e

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

WHERE b.bonus is null or b.bonus < 1000;

Managers with at Least 5 Direct Reports

|  |  |
| --- | --- |
| Table: Employee | |
| Column Name | Type |
| id | int |
| name | varchar |
| department | varchar |
| managerId | int |

* id is the primary key (column with unique values) for this table.
* Each row of this table indicates the name of an employee, their department, and the id of their manager.
* If managerId is null, then the employee does not have a manager.
* No employee will be the manager of themself.

Write a solution to find managers with at least five direct reports. Return the result table in any order.

The result format is in the following example.

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

SELECT e.name

FROM Employee e

JOIN (

    SELECT managerId

    FROM Employee

    WHERE managerId IS NOT NULL

    GROUP BY managerId

    HAVING COUNT(\*) >= 5

) m ON e.id = m.managerId;

Table: Signups

|  |  |
| --- | --- |
| 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').

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.

|  |  |  |
| --- | --- | --- |
| Confirmations table: | | |
| 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 |

The result format is in the following example.

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

Explanation:

* 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.

|  |  |
| --- | --- |
| Output: |  |
| user\_id | confirmation\_rate |
| 6 | 0 |
| 3 | 0 |
| 7 | 1 |
| 2 | 0.5 |

/\* Write your T-SQL query statement below \*/

WITH ConfirmationCounts AS (

    SELECT

        user\_id,

        COUNT(\*) AS total\_requests,

        SUM(CASE WHEN action = 'confirmed' THEN 1 ELSE 0 END) AS confirmed\_requests

    FROM Confirmations

    GROUP BY user\_id

),

UserConfirmationRates AS (

    SELECT

        s.user\_id,

        COALESCE(ROUND(c.confirmed\_requests \* 1.0 / c.total\_requests, 2), 0.00) AS confirmation\_rate

    FROM Signups s

    LEFT JOIN ConfirmationCounts c ON s.user\_id = c.user\_id

)

SELECT

    user\_id,

    confirmation\_rate

FROM UserConfirmationRates

ORDER BY user\_id;