SQL Reporting Exercises

▼ **10. Given the tables `users` and `rides`, write a query to report the distance traveled by each user in descending order.**

`users` table

Column	Type
`id`	INTEGER
`name`	INTEGER

`rides` table

Column	Type
`id`	INTEGER
`passenger_user_id`	INTEGER
`distance`	FLOAT

```
SELECT users.Id
      ,SUM(rides.distance) AS 'Total Distance'
  FROM users
    LEFT OUTER JOIN rides
      ON users.Id = rides.UserId
  GROUP BY users.Id
  ORDER BY 2 DESC
```

Note: If we just used the rides table, we would be exclude the users who have never taken a ride, which would be excluding very valuable data for us to analyze.

▼ **11. Write a query to find all the users that are currently “Excited” and have never been “Bored” within a campaign.**

Let’s first design the hypothetical table we will work with, called **user_campaigns**

UserId	CampaignId	Date	Status
1	1	4/3/23	Excited
1	1	1/1/23	Neutral
2	3	4/1/23	Excited
3	1	3/31/23	Excited
3	2	2/24/23	Excited
3	2	1/5/23	Bored
4	3	3/26/23	Neutral

UserId	CampaignId	Date	Status
4	1	2/16/23	Excited

We will define “currently excited” as the users latest status by the Date field.

The query should return users “1” and “2”.

We can first get the list of all users that have ever been ‘Bored’ so that we can exclude these users:

```
SELECT DISTINCT UserId
  FROM user_campaigns
  WHERE Status = 'Bored'
```

We also want to only look at the latest status for each user, so we can use this as a subquery called current_status which we will ultimately join to:

```
SELECT UserId
       ,MAX(Date) AS 'LatestDate'
  FROM user_campaigns
  GROUP BY UserId
```

Joining the current_status table to the original user_campaigns table, and using the ‘Bored’ filter, puts everything together to create the final query:

```
SELECT *
  FROM user_campaigns
    INNER JOIN (
        SELECT UserId
              ,MAX(Date) AS 'LatestDate'
          FROM user_campaigns    -- Look only at the latest status for each user
          WHERE UserId NOT IN (SELECT DISTINCT UserId
                               FROM user_campaigns
                               WHERE Status = 'Bored'
                              )  -- Exclude users that were ever 'Bored'
          GROUP BY UserId
        ) current_status
      ON user_campaigns.UserId = current_status.UserId
       AND user_campaigns.Date = current_status.LatestDate -- only look at the current 'Status' for each user
  WHERE user_campaigns.Status = 'Excited' -- only look at users that are 'Excited'
;
```

Here is another solution which uses CTEs and RANK() with a PARTITION

```
-- Another solution which uses RANK() and PARTITION
WITH current_status AS (
    SELECT UserId
          ,Status
          ,Date
          ,RANK() OVER (PARTITION BY UserId ORDER BY Date DESC) AS 'DateRank'
      FROM user_campaigns
),

bored_users AS (
    SELECT DISTINCT UserId
```

```
      FROM user_campaigns
      WHERE Status = 'Bored'
)

SELECT *
  FROM current_status
  WHERE DateRank = 1
    AND Status = 'Excited'
    AND UserId NOT IN (SELECT UserId FROM bored_users)
;
```

▼ **12. Write a SQL query to select the second highest salary in the engineering department.**

Write a SQL query to select the 2nd highest salary in the engineering department.

Note:
If more than one person shares the highest salary, the query should select the next highest salary.

`employees` table

Column	Type
`id`	INTEGER
`first_name`	VARCHAR
`last_name`	VARCHAR
`salary`	INTEGER
`department_id`	INTEGER

`departments` table

Column	Type
`id`	INTEGER
`name`	VARCHAR

We have to join the employees table to the department table so we can get the department name and filter for the engineering department.

```
SELECT employees.id AS 'employee_id'
      ,first_name
      ,last_name
      ,salary
      ,name AS 'department'
      ,DENSE_RANK() OVER (PARTITION BY departments.id ORDER BY salary DESC)
  FROM employees
    LEFT OUTER JOIN departments
      ON employees.department_id = departments.id
    WHERE departments.name = 'engineering'
```

Then, we can use this query as a subquery to get the second highest salary, by filtering on rank = 2

```
SELECT DISTINCT salary
  FROM (SELECT employees.id AS 'employee_id'
        ,first_name
        ,last_name
        ,salary
        ,name AS 'department'
        ,DENSE_RANK() OVER (PARTITION BY departments.id ORDER BY salary DESC) AS 'rank'
  FROM employees
    LEFT OUTER JOIN departments
      ON employees.department_id = departments.id
    WHERE departments.name = 'engineering' ) sub
  WHERE sub.rank = 2
```

▼ **13. Given a table of bank transactions, write a query to get the last transaction for each day.**

Given a table of bank transactions with columns `id`, `transaction_value`, and `created_at` representing the date and time for each transaction, write a query to get the last transaction for each day.

The output should include the id of the transaction, datetime of the transaction, and the transaction amount. Order the transactions by datetime.

`bank_transactions` table

Column	Type
`id`	INTEGER
`created_at`	DATETIME
`transaction_value`	FLOAT

Since we need to get the last transaction for each *day*, we need to convert the DATETIME date which included the time to a DATE which will be in YYYY-MM-DD format.

Then, we can use ROW_NUMBER() and partition by that YYYY-MM-DD date, ordering by the DATETIME date, so that we can order the transaction by hours/minutes etc. ROW_NUMBER() is used because we don't care about ties for transactions that may have occurred at the same time (which should be unlikely). We just want the last transaction.

```
WITH transactions_by_day AS (
    SELECT created_at
          ,transaction_value
          ,id
          ,CAST(created_at AS DATE) AS 'day'
    FROM bank_transactions
)

SELECT created_at
      ,transaction_value
      ,id
  FROM (SELECT day
              ,created_at
              ,transaction_value
              ,id
              ,ROW_NUMBER() OVER (PARTITION BY day ORDER BY created_at DESC) AS 'rank'
```

```
         FROM transactions_by_day) sub
 WHERE sub.rank = 1
```

This is another cool and simple solution!

```
SELECT id
      ,created_at
      ,transaction_value
 FROM bank_transactions
WHERE created_at IN
(
    SELECT MAX(created_at)
      FROM bank_transactions
      GROUP BY DATE(created_at)
)
ORDER BY created_at
```

▼ **14. We want to find the top five most expensive projects by budget to employee count ratio, but we've found a bug. Write a query to account for the error and select the top five most expensive projects by budget to employee count ratio.**

We're given two tables. One is named `projects` and the other maps employees to the projects they're working on.

We want to select the five most expensive projects by budget to employee count ratio. But let's say that we've found a bug where there exist duplicate rows in the `employee_projects` table.

Write a query to account for the error and select the top five most expensive projects by budget to employee count ratio.

`projects` table

column	type
`id`	INTEGER
`title`	VARCHAR
`state_date`	DATETIME
`end_date`	DATETIME
`budget`	INTEGER

`employee_projects` table

Column	Type
`project_id`	INTEGER
`employee_id`	INTEGER

First let's join the two tables together so we can associate each project with the employees that worked on them.

Since we know that there are duplicate rows in the employee_projects table, that means each project will be associated with the same employee multiple times. We will need to account for this when we calculate the budget to employee ratio.

We calculate the employee budget ratio with `budget/COUNT(DISTINCT employee_id)` Using COUNT with DISTINCT will only count the unique number of employee_ids and not the total count.

```
SELECT title
      ,budget/COUNT(DISTINCT employee_id) AS 'budget_per_employee'
  FROM projects
    INNER JOIN employee_projects
      ON projects.id = employee_projects.project_id
  GROUP BY projects.id
  ORDER BY 2 DESC
  LIMIT 5
;
```

▼ 15. You have a table that represents the total number of messages sent between two users by date on Facebook Messenger. Answer these questions:

`messages` table

Column	Type
`id`	INTEGER
`date`	DATETIME
`user1`	INTEGER
`user2`	INTEGER
`msg_count`	INTEGER

1. **What are some insights that could be derived from this table?**
    - Who are the top users that each user messages?

```
SELECT user1
      ,user2
      ,num_conversations
      ,RANK() OVER (PARTITION BY user1 ORDER BY num_conversations DESC)
  FROM (
        SELECT user1
              ,user2
              ,COUNT(*) AS 'num_conversations'
        FROM messages
        GROUP BY user1, user2
       ) sub
  ORDER BY user1, num_conversations DESC
```

    - How much engagement does the messaging functionality have? Is it meeting our goals?
        - What are the total amount of messages sent per day?

```
SELECT DATE(date) AS 'day'
      ,SUM(msg_count) AS 'total_msg_count'
  FROM messages
  GROUP BY 1
  ORDER BY 1 DESC
  LIMIT 5

output:
-- day         total_msg_count
  "2021-12-11"  881
  "2021-12-10"  1336
  "2021-12-09"  769
  "2021-12-08"  642
  "2021-12-07"  943
```

- What is the average number of messages sent per day?

```
SELECT AVG(daily_msg_count) AS 'average daily message count'
  FROM (
        SELECT DATE(date)
              ,sum(msg_count) AS 'daily_msg_count'
        FROM messages
        GROUP BY 1
       ) sub

output: 597.245614
```

- What is the average number of messages per conversation?

```
/* Every row in the messages table represents a conversation,
   so to answer this question, we simply need to
   average the msg_count field.
*/
SELECT AVG(msg_count)
  FROM messages

output: 110.7782228
```

- Can break down by time periods.
  - How many more conversations are being had compared to last year?

```
SELECT YEAR(date) AS 'year'
      ,COUNT(*) AS 'num_conversations'
  FROM messages
  GROUP BY 1
  ORDER by 1

output:
--  year  num_conversations
    2019  158
    2020  2002
    2021  1835
```

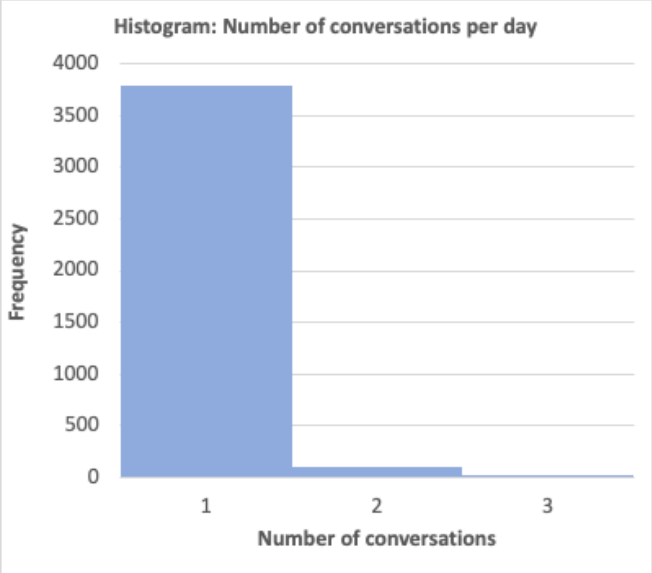
2. **What do you think the distribution of the number of conversations created by each user per day looks like?**
    - We have to think about the probability of a user having new conversations each day.
    - An average user likely doesn't start more than a handful of new conversations per day.
        - This may result in a distribution that is skewed to the right.
    - A smaller number of users may have many more conversations per day if they are using the messenger for work.
        - We could also expect to see a bimodal distribution where the left peak represents the users who start a few new conversations per day and the right peak represents the users who start many new conversations per day.
3. Write a query to get the distribution of the number of conversations created by each user by day in 2020. We can define a new conversation as a set of messages sent between two users.

    We want to write a query that can be used to create a histogram where the x-axis bins represent the number of new conversations started in a day and the y-axis represents the number of people who started that number of conversations (this is known as the frequency).

    The messages table provides us with the number of messages between each pair of users per day, but we need the number of conversations a user has per day. We can get this by finding the unique number of users each user sends messages to each day.

```
SELECT num_conversations
      ,COUNT(*) AS 'frequency'
  FROM (
        SELECT user1
              ,DATE(date)
              ,COUNT(DISTINCT user2) AS 'num_conversations'
        FROM messages
        GROUP BY 1, 2
   ) sub
  GROUP BY sub.num_conversations

output:
-- num_conversations  frequency
      1               3783
      2               103
      3               2
```


Histogram: Number of conversations per day
4000
3500
3000
2500
2000
1500
1000
500
0
Frequency
1
2
3
Number of conversations