Q1: First, we obtain the order of transaction numbers for each user. We can do this by using the `ROW_NUMBER` window function where we `PARTITION` the all transactions by `user_id` and `ORDER BY` the transaction_date.

```
SELECT
  user_id,
  spend,
  transaction_date,
  ROW_NUMBER() OVER (
    PARTITION BY user_id ORDER BY transaction_date) AS row_num
FROM transactions;
```

Here's how the first 5 rows of output looks like:

user_id	spend	transaction_date	row_num
111	100.50	01/08/2022 12:00:00	1
111	55.00	01/10/2022 12:00:00	2
111	89.60	02/05/2022 12:00:00	3
121	36.00	01/18/2022 12:00:00	1
121	22.20	04/01/2022 12:00:00	2

From there on, we can simply convert the query into a subquery and filter for the users' third transaction which is their third transaction sorted by the transaction date (and is denoted as `row_num = 3`).

```
SELECT
  user_id,
  spend,
  transaction_date
FROM (
  -- Insert the above query here
) AS trans_num
WHERE row_num = 3;
```

Results:

user_id	spend	transaction_date
111	89.60	02/05/2022 12:00:00
121	67.90	04/03/2022 12:00:00

Apart from using subquery to solve this question, you can also use a CTE. Do you know the differences between a subquery and a CTE?

A **CTE** is a temporary data set to be used as part of a query and it exists during the entire query session. A **subquery** is a nested query. It's a query within a query and unlike CTE, it can be used within that query only. Both methods give the same output and perform fairly similarly.

Differences are CTE is reusable during the entire session and more readable, whereas subquery can be used in FROM and WHERE clauses and can act as a column with a single value.

Solution #1: Using Subquery

```
SELECT
  user_id,
  spend,
  transaction_date
FROM (
  SELECT
    user_id,
    spend,
    transaction_date,
    ROW_NUMBER() OVER (
      PARTITION BY user_id ORDER BY transaction_date) AS row_num
  FROM transactions) AS trans_num
WHERE row_num = 3;
```

Solution #2: Using CTE

```
WITH trans_num AS (
  SELECT
    user_id,
    spend,
    transaction_date,
    ROW_NUMBER() OVER (
      PARTITION BY user_id ORDER BY transaction_date) AS row_num
  FROM transactions)

SELECT
  user_id,
  spend,
  transaction_date
FROM trans_num
WHERE row_num = 3;
```

Q2:

Step 1: Join tables and filter for 'send' and 'open' snaps

We start by joining the `age_breakdown` table with the `activities` table using the `user_id` field as a common identifier. We then filter the results by 'send' and 'open' `activity_type` because we are interested in focusing on sending and opening snaps. Finally, we group the results by `age_bucket` to aggregate the data.

```
SELECT *
FROM activities
INNER JOIN age_breakdown AS age
  ON activities.user_id = age.user_id
WHERE activities.activity_type IN ('send', 'open') GROUP BY
age.age_bucket;
```

Step 2: Obtain total time spent on sending and opening snaps

Next, we calculate the total time spent on sending and opening snaps using aggregate function `SUM()` along with a FILTER clause for each activity type ('send' and 'open').

```
SELECT
  age.age_bucket,
    SUM(activities.time_spent) FILTER (WHERE activities.activity_type =
'send') AS send_perc,
    SUM(activities.time_spent) FILTER (WHERE activities.activity_type =
'open') AS open_perc
FROM activities
INNER JOIN age_breakdown AS age
  ON activities.user_id = age.user_id
WHERE activities.activity_type IN ('send', 'open')
GROUP BY age.age_bucket;
```

Here's how the output looks like:

age_bucket	send_perc	open_perc
21-25	6.24	5.25
26-30	13.91	3.00
31-35	3.50	5.75

Step 3: Calculate the percentages

Next, we convert the query into a Common Table Expression (CTE) called `snap_statistics`. Within the CTE, we calculate the percentages of time spent on sending and opening snaps using the formula:

Percentage of time spent on sending/opening snaps = Time spent on sending/opening snaps / Total time spent on sending and opening snaps

This formula will help us determine the proportion of time spent on sending or opening snaps relative to the total time spent on both activities.

```
SELECT
  age.age_bucket,
    SUM(activities.time_spent) FILTER (WHERE activities.activity_type =
'send')/
      SUM(activities.time_spent) AS send_perc,
    SUM(activities.time_spent) FILTER (WHERE activities.activity_type =
'open')/
      SUM(activities.time_spent) AS open_perc
FROM activities
INNER JOIN age_breakdown AS age
  ON activities.user_id = age.user_id
WHERE activities.activity_type IN ('send', 'open')
GROUP BY age.age_bucket;
```

age_bucket	send_perc	open_perc
21-25	0.54308093994778067885	0.45691906005221932115
26-30	0.82259018332347723241	0.17740981667652276759
31-35	0.37837837837837837838	0.62162162162162162162

Step 4: Round the percentages

Finally, we round the percentages to 2 decimal places. Ensure that you're wrapping the expression correctly.

Lastly, we ensure that the calculated percentages are rounded to 2 decimal places using the `ROUND()` function with the appropriate precision specified.

```
SELECT
  age.age_bucket,
  ROUND(100.0 *
    SUM(activities.time_spent) FILTER (WHERE activities.activity_type =
'send')/
      SUM(activities.time_spent),2) AS send_perc,
  ROUND(100.0 *
    SUM(activities.time_spent) FILTER (WHERE activities.activity_type =
'open')/
      SUM(activities.time_spent),2) AS open_perc
FROM activities
INNER JOIN age_breakdown AS age
  ON activities.user_id = age.user_id
WHERE activities.activity_type IN ('send', 'open')
GROUP BY age.age_bucket;
```

Here's the results:

age_bucket	send_perc	open_perc
21-25	54.31	45.69
26-30	82.26	17.74
31-35	37.84	62.16

Here's a summary of the results:

- Age group 21-25: Sending snaps is slightly more prevalent (54.31%) than opening snaps (45.69%).
- Age group 26-30: Sending snaps is significantly more dominant (82.26%) than opening snaps (17.74%).
- Age group 31-35: Opening snaps is more prevalent (62.16%) than sending snaps (37.84%).

A useful tip to keep in mind is that when dividing two integers using the division `/` operator in PostgreSQL, only the integer part of the result is considered, disregarding any remainder (e.g., .123, .352). To avoid integer division and ensure that the resulting values are converted into decimal values, it is important to multiply the values by 100.0 before performing the division. This will ensure that the calculated percentages are accurate and expressed as decimal values.

Solution #2: Using CTE and CASE Statement

```
WITH snaps_statistics AS (
  SELECT
    age.age_bucket,
    SUM(CASE WHEN activities.activity_type = 'send'
      THEN activities.time_spent ELSE 0 END) AS send_timespent,
    SUM(CASE WHEN activities.activity_type = 'open'
      THEN activities.time_spent ELSE 0 END) AS open_timespent,
    SUM(activities.time_spent) AS total_timespent
  FROM activities
 INNER JOIN age_breakdown AS age
    ON activities.user_id = age.user_id
  WHERE activities.activity_type IN ('send', 'open')
  GROUP BY age.age_bucket)

SELECT
  age_bucket,
  ROUND(100.0 * send_timespent / total_timespent, 2) AS send_perc,
  ROUND(100.0 * open_timespent / total_timespent, 2) AS open_perc
FROM snaps_statistics;
```

Q3:

Step 1: Calculate the average tweet count for each user

To obtain the average rolling average tweet count for each user, we use the following query which calculates the average tweet count for each user ID and date.

```
SELECT
  user_id,
  tweet_date,
  AVG(tweet_count) OVER (
    PARTITION BY user_id
    ORDER BY tweet_date) AS rolling_avg_3d
FROM tweets;
```

Output showing the first 5 rows:

user_id	tweet_date	tweet_count	rolling_avg
111	06/01/2022 00:00:00	2	2.0000000000000000
111	06/02/2022 00:00:00	1	1.5000000000000000
111	06/03/2022 00:00:00	3	2.0000000000000000
111	06/04/2022 00:00:00	4	2.5000000000000000
111	06/05/2022 00:00:00	5	3.0000000000000000

The output shows the rolling average tweet count for the cumulative number of days.

Calculating the rolling average tweet count:

- 06/01/2022: 2 = 2.0
- 06/02/2022: 2 + 1 = 3 / 2 days = 1.5
- 06/03/2022: 2 + 1 + 3 = 6 / 3 days = 2.0
- 06/04/2022: 2 + 1 + 3 + 4 = 10 / 4 days = 2.5
- 06/05/2022: 2 + 1 + 3 + 4 + 5 = 15 / 5 days = 3.0

However, this is not what we need as we want to calculate the **rolling average over a 3-day period**. To achieve this, we add the expression `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW` to the window function.

```
AVG(tweet_count) OVER (
  PARTITION BY user_id
  ORDER BY tweet_date
  ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) -- This is the additional
expression
```

Step 2: Calculate the rolling average tweet count for each user over a 3-day period

To calculate the rolling average tweet count by 3-day period, we modify the previous query by adding the `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW` expression to the window function.

```
SELECT
  user_id,
  tweet_date,
  tweet_count,
  AVG(tweet_count) OVER (
    PARTITION BY user_id
    ORDER BY tweet_date
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS rolling_avg
FROM tweets;
```

Output showing the first 5 rows:

user_id	tweet_date	tweet_count	rolling_avg
111	06/01/2022 00:00:00	2	2.0000000000000000
111	06/02/2022 00:00:00	1	1.5000000000000000
111	06/03/2022 00:00:00	3	2.0000000000000000
111	06/04/2022 00:00:00	4	2.6666666666666667
111	06/05/2022 00:00:00	5	4.0000000000000000

This query outputs the rolling average tweet count by 3-day period, as required by the question.

Calculating the rolling averages using `ROWS BETWEEN 2 PRECEDING AND CURRENT ROW`

- 06/01/2022: 2 = 2.0
- 06/02/2022: 2 + 1 = 3 / 2 days = 1.5
- 06/03/2022: 2 + 1 + 3 = 6 / 3 days = 2.0
- 06/04/2022: 1 + 3 + 4 = 8 / 3 days = 2.666...
- 06/05/2022: 3 + 4 + 5 = 12 / 3 days = 4.0

Step 3: Round the rolling average tweet count to 2 decimal points

Finally, we round up the rolling averages to the nearest 2 decimal points by incorporating the `ROUND()` function into the previous query.

```
SELECT
  user_id,
  tweet_date,
  ROUND(AVG(tweet_count) OVER (
    PARTITION BY user_id
    ORDER BY tweet_date
    ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
  ,2) AS rolling_avg_3d
FROM tweets;
```

Q4:
To find the highest-grossing products, we must find the total spend by category and product. Note that we must filter by transactions in 2022.

```
SELECT
  category,
  product,
  SUM(spend) AS total_spend
FROM product_spend
WHERE transaction_date >= '2022-01-01'
  AND transaction_date <= '2022-12-31'
GROUP BY category, product;
```

category	product	total_spend
electronics	wireless headset	447.90
appliance	refrigerator	299.99
appliance	washing machine	439.80
electronics	computer mouse	45.00
electronics	vacuum	486.66

The output represents the total spend by category (electronics, appliance) and product.

Then, we reuse the query as a CTE or subquery (in this case, we are using a CTE) and utilize the `RANK` window function to calculate the ranking by total spend, partition by category and order by the total spend in descending order.

```
WITH product_category_spend
AS (
-- Insert query above)

SELECT
  *,
  RANK() OVER (
    PARTITION BY category
    ORDER BY total_spend DESC) AS ranking FROM product_category_spend;
```

category	product	total_spend	ranking
appliance	washing machine	439.80	1
appliance	refrigerator	299.99	2
electronics	vacuum	486.66	1
electronics	wireless headset	447.90	2
electronics	computer mouse	45.00	3

Finally, we use this result and filter for a rank less than or equal to 2 as the question asks for top two highest-grossing products only.

Solution #1: Using CTE

```
WITH product_category_spend AS (
SELECT
  category,
  product,
  SUM(spend) AS total_spend
FROM product_spend
WHERE transaction_date >= '2022-01-01'
  AND transaction_date <= '2022-12-31'
GROUP BY category, product
),
top_spend AS (
SELECT *,
  RANK() OVER (
    PARTITION BY category
    ORDER BY total_spend DESC) AS ranking
FROM product_category_spend)

SELECT category, product, total_spend
FROM top_spend
WHERE ranking <= 2
ORDER BY category, ranking;
```

Solution #2: Using Subquery

```
SELECT
  category,
  product,
  total_spend
FROM (
    SELECT
      *,
      RANK() OVER (
        PARTITION BY category
```

```
        ORDER BY total_spend DESC) AS ranking
    FROM (
        SELECT
          category,
          product,
          SUM(spend) AS total_spend
        FROM product_spend
        WHERE transaction_date >= '2022-01-01'
          AND transaction_date <= '2022-12-31'
        GROUP BY category, product) AS total_spend
  ) AS top_spend
WHERE ranking <= 2
ORDER BY category, ranking;
```

Q5:
The task can be broken down into 3 steps:
Step 1: Find the top 10 artists with the most songs in the global top 10 chart

First, join the `songs` and `global_song_rank` tables to get a table with artists and their song rankings using `INNER JOIN`.

Then, count the number of times an artist's song appears in the top 10 of the chart, grouping by the artist. The resulting output should give us a list of artists with their respective song counts.

Here's the query:

```
SELECT
  songs.artist_id,
  COUNT(songs.song_id) AS song_count
FROM songs
INNER JOIN global_song_rank AS ranking
  ON songs.song_id = ranking.song_id
WHERE ranking.rank <= 10
GROUP BY songs.artist_id;
```

Here's the output for the first 5 rows:

artist_id	song_count
101	5
200	4
125	6
240	3

120	2

Based on the first row, artist ID 101 whose songs have appeared in the Top 10 of the chart 5 times.

Step 2: Rank Artists by Song Appearances in Top 10 Chart

Now that we have the artists and their song counts, we need to rank them according to their number of song appearances in descending order.

We can use a subquery to get the song counts and then apply a `DENSE_RANK` window function to assign a rank to each artist based on their song count. The resulting output should give us a list of artists with their respective ranks.

```
SELECT
  artist_id,
  DENSE_RANK() OVER (
    ORDER BY song_count DESC) AS artist_rank
FROM (
  SELECT
    songs.artist_id,
    COUNT(songs.song_id) AS song_count
  FROM songs
  INNER JOIN global_song_rank AS ranking
    ON songs.song_id = ranking.song_id
  WHERE ranking.rank <= 10
  GROUP BY songs.artist_id
)  AS top_songs;
```

Do you know why we're using `DENSE_RANK` instead of `RANK`?

While both functions assign the same rank to duplicates, `RANK` actually skips the next number in the ranking, which is not what we want for this question. That's why we'll be using `DENSE_RANK` instead, which doesn't skip any rank numbers.

Here's an example to illustrate the difference between DENSE_RANK and RANK:

artist_id	song_count	dense_rank_num	rank_num
125	6	1	1
101	5	2	2
145	4	3	3
200	4	3	3
240	3	4	5

If we compare the `dense_rank_num` and `rank_num` columns in the table, we can see a difference in the ranking of artists with the same number of songs.

For example, artist ID 145 and 200 both have 4 songs, so they should have the same rank. With `DENSE_RANK`, both artists are correctly ranked as 3rd, and the next artist, ID 240, is ranked 4th. However, with `RANK`, the 4th rank is skipped, so artist ID 240 is labeled as the 5th rank. This is not what we want, as it breaks the continuity of the ranking. Therefore, we use `DENSE_RANK` to ensure that all ranks are assigned without skipping any numbers.

It's important that the ranking is continuous (1, 2, 2, 3, 4, 5) and not skipped (1, 2, 2, 4, 5). So in this case, we want to use `DENSE_RANK`. Do you see the difference now? ;)

Step 3: Limit Results to Top 5 Ranked Artists and Join with Artist Names

Finally, we need to filter the results to the top 5 artists by their rank and join the result with the `artists` table to get their names.

Here's the query:

```
WITH top_artists AS (
  SELECT
    artist_id,
    DENSE_RANK() OVER (
      ORDER BY song_count DESC) AS artist_rank
  FROM (
    SELECT
      songs.artist_id,
      COUNT(songs.song_id) AS song_count
    FROM songs
    INNER JOIN global_song_rank AS ranking
      ON songs.song_id = ranking.song_id
    WHERE ranking.rank <= 10
    GROUP BY songs.artist_id) AS top_songs
)

SELECT
  artists.artist_name,
  top_artists.artist_rank
FROM top_artists
INNER JOIN artists
  ON top_artists.artist_id = artists.artist_id
WHERE top_artists.artist_rank <= 5
ORDER BY
  top_artists.artist_rank,
  artists.artist_name;
```

Q6:
The following steps are suggested to solve this question:

1. Join tables for records with successful confirmations.
2. Calculate the signup activation rate of users who have confirmed their accounts.

Step 1: Join tables for records with successful confirmations

To start, we will use a LEFT JOIN to connect the `emails` and `texts` tables. It's important to note that an `INNER JOIN` won't work for this question and we'll explain why shortly.

We'll keep only the users who successfully signed up and received a 'Confirmed' text confirmation in the `signup_action` column located in the `texts` table.

```
SELECT texts.email_id, emails.email_id
FROM emails
LEFT JOIN texts
  ON emails.email_id = texts.email_id
  AND texts.signup_action = 'Confirmed';
```

Your output should look something like this:

email_id	email_id
125	125
236	236
433	
450	

Why `LEFT JOIN` is Necessary in this Query?

Now, it's important to note that not every `email_id` in the `emails` table will have a matching value in the `texts` table, and this is where the `LEFT JOIN` comes into play.

When we perform a `LEFT JOIN`, all the rows from the left table (`emails` in this case) are returned along with matching rows from the right table (`texts` in this case). If there is no match for a particular row in the right table, then the columns from the right table will be NULL.

If we were to use an `INNER JOIN`, only the matching rows between the two tables would be returned, effectively filtering out any `email_id` values from the `emails` table that do not have a corresponding match in the `texts` table.

Output from using an `INNER JOIN`:

email_id	email_id
125	125
236	236

This could result in relevant users being excluded from the calculation completely, which is not what we want.

Step 2: Calculate the signup activation rate of users who have confirmed their accounts

Signup Activation Rate = Number of users who confirmed their accounts / Number of users in the `emails` table

With the given formula, we expressed them in the query below.

```
SELECT
  COUNT(texts.email_id)
    /COUNT(DISTINCT emails.email_id) AS activation_rate
FROM emails
LEFT JOIN texts
  ON emails.email_id = texts.email_id
  AND texts.signup_action = 'Confirmed';
```

But wait, did you get an activation rate of '0' when you ran the query? That's because dividing an integer with another integer would sometimes result in '0'.

To avoid this, we'll need to cast either the denominator or the numerator to DECIMAL type.

Finally, the `ROUND` function is used to truncate the result to 2 decimal places, as specified in the instructions.

```
SELECT
  ROUND(COUNT(texts.email_id)::DECIMAL
    /COUNT(DISTINCT emails.email_id),2) AS activation_rate
FROM emails
LEFT JOIN texts
  ON emails.email_id = texts.email_id
  AND texts.signup_action = 'Confirmed';
```

Q7: As usual, take a step back and identify the key points of solving the task. This is how we see it:

1. In addition to the contracts data, we also need the product category information, thus we need to join the tables.
2. Identify the customers with purchases from all of the product categories (i.e. Supercloud customers).
3. Output the Supercloud customer IDs.

Step 1

Join the two tables with either `LEFT` or `INNER JOIN`.

```
SELECT
  customers.customer_id,
  products.product_category
FROM customer_contracts AS customers
LEFT JOIN products
  ON customers.product_id = products.product_id;
```

Step 2

Now, let's count all of the distinct product categories that a customer has purchased a product from.

```
SELECT
  customers.customer_id,
  COUNT(DISTINCT products.product_category) AS unique_count
FROM customer_contracts AS customers
LEFT JOIN products
  ON customers.product_id = products.product_id
GROUP BY customers.customer_id;
```

customer_id	unique_count
1	3
7	3

Step 3

We are almost there, hang on!

Let's wrap the previous step into a CTE, and select only the users who have products from all of the categories.

One way of doing this would be simply filtering for `unique_count` of 3 (as there are 3 product categories). However, let's be even smarter and make it dynamic by using a subquery referencing the products table instead.

I believe this is a pretty nice solution, don't you?

```
WITH supercloud AS (
SELECT
  customers.customer_id,
  COUNT(DISTINCT products.product_category) as unique_count
FROM customer_contracts AS customers
LEFT JOIN products
  ON customers.product_id = products.product_id
GROUP BY customers.customer_id
)

SELECT customer_id
FROM supercloud
WHERE unique_count = (
  SELECT COUNT(DISTINCT product_category)
  FROM products)
ORDER BY customer_id;
```

Solution #2

```
SELECT customer_id
FROM (
  SELECT customers.customer_id
  FROM customer_contracts AS customers
  LEFT JOIN products
    ON customers.product_id = products.product_id
  GROUP BY customers.customer_id
  HAVING COUNT(DISTINCT products.product_category) = 3
) AS supercloud
ORDER BY customer_id;
```

Q8:

Step 1: Ordering and Partitioning

We first order the measurements based on their measurement time and partition them by day using the `ROW_NUMBER` window function. This helps us establish the order of measurements within each day:

```
SELECT
  CAST(measurement_time AS DATE) AS measurement_day,
  measurement_value,
  ROW_NUMBER() OVER (
    PARTITION BY CAST(measurement_time AS DATE)
    ORDER BY measurement_time) AS measurement_num
```

```
  FROM measurements;
```

It is important to use `measurement_time` in the ORDER BY clause of the window function to ensure that the measurements are ordered within each day based on the actual measurement's time. This will ensure that the row numbering (`measurement_num`) is accurate and reflects the chronological order of the measurements within each day.

Showing the first 5 rows of output:

measurement_day	measurement_value	measurement_num
07/10/2022 00:00:00	1109.51	1
07/10/2022 00:00:00	1662.74	2
07/10/2022 00:00:00	1246.24	3
07/11/2022 00:00:00	1124.50	1
07/11/2022 00:00:00	1234.14	2

Step 2 & 3: Filtering and Summing

To filter for odd and even numbers, we can use the following two methods:

- Modulus operator (`%`): Use `measurement_num % 2 != 0` to check if the result is 1, indicating odd numbers or `measurement_num % 2 = 0` with a result of 1 for even numbers.
- `MOD()`: Use `MOD(measurement_num, 2) != 0` to find odd results and `MOD(measurement_num, 2) = 0` for even results.

Note: The modulus operator `%` returns the remainder of a division. When we divide an even number by 2, the remainder is always 0, whereas dividing an odd number will result in a non-zero value.

Finally, we can apply the modulus concept to the aggregate function `SUM()` along with the `FILTER` clause, summing over the corresponding `measurement_value`.

```
WITH ranked_measurements AS (
  SELECT
    CAST(measurement_time AS DATE) AS measurement_day,
    measurement_value,
    ROW_NUMBER() OVER (
      PARTITION BY CAST(measurement_time AS DATE)
      ORDER BY measurement_time) AS measurement_num
  FROM measurements
)

SELECT
  measurement_day,
  SUM(measurement_value) FILTER (WHERE measurement_num % 2 != 0) AS
odd_sum,
```

```
  SUM(measurement_value) FILTER (WHERE measurement_num % 2 = 0) AS
even_sum
FROM ranked_measurements
GROUP BY measurement_day;
```

Results:

measurement_day	odd_sum	even_sum
07/10/2022 00:00:00	2355.75	1662.74
07/11/2022 00:00:00	2377.12	2480.70
07/12/2022 00:00:00	2903.40	1244.30

Q9:

1. Rank the transaction days to find the latest transaction date by the user.
2. Filter records with the latest transaction date only.
3. Pull the appropriate fields and count the number of products purchased.

First, we obtain the latest transaction date for each user using a `RANK()` window function by partitioning by user and ordering by transaction date in descending order.

```
SELECT
  transaction_date,
  user_id,
  product_id,
  RANK() OVER (
    PARTITION BY user_id
    ORDER BY transaction_date DESC) AS days_rank
  FROM user_transactions;
```

Using the above query, we convert it into a common table expression (CTE) with a `WITH` statement or a subquery. In this example, we are using CTE.

Subsequently, we filter for records where `days_rank` equals 1 indicating the latest transaction date and declare the transaction date and user ID in the `SELECT` statement.

We also include a count of product IDs as the number of products and order the results by transaction date.

Solution #1: Using CTE

```
-- Convert query into a CTE using WITH statement
WITH latest_transaction AS (
  SELECT
    transaction_date,
    user_id,
    product_id,
    RANK() OVER (PARTITION BY user_id
      ORDER BY transaction_date DESC) AS days_rank
  FROM user_transactions)

SELECT
  transaction_date,
  user_id,
  COUNT(product_id) AS purchase_count
FROM latest_transaction
WHERE days_rank = 1
GROUP BY transaction_date, user_id
ORDER BY transaction_date;
```

Solution #2: Using Subquery

```
SELECT
  transaction_date,
  user_id,
  COUNT(product_id) AS purchase_count
FROM (
  SELECT
    transaction_date,
    user_id,
    product_id,
    RANK() OVER (
      PARTITION BY user_id
      ORDER BY transaction_date DESC) AS days_rank
  FROM user_transactions) AS latest_transaction
WHERE days_rank = 1
GROUP BY transaction_date, user_id
ORDER BY transaction_date;
```

Q10:
It is easier to organise our thoughts and code if we break the task apart. To complete the task, we need to

1. Find the most common number of order occurrences.
2. Match the order occurrences to the corresponding item count.
3. Order the results by item count.

Step 1

There are two ways to obtain this: a simple `MAX()` or a built-in `MODE() WITHIN GROUP ()` function. Run the following queries separately. Both queries will yield the same number of order occurrences which is 1000.

```
-- Method #1
SELECT MAX(order_occurrences)
FROM items_per_order;
```

Why are we using `MAX()`? Since the `order_occurrences` field contains the total number of occurrences by `item_count`, simply taking the **maximum value** in this field is the same as taking the **most common** number of order occurrences.

```
-- Method #2
SELECT MODE() WITHIN GROUP (ORDER BY order_occurrences DESC)
FROM items_per_order;
```

Step 2

Now, let's find out which `item_count` corresponds to the `order_occurrences`. To do so, simply declare `item_count` in the `SELECT` statement while filtering the results using either the `MAX()` or `MODE() WITHIN GROUP ()` method.

```
-- Method #1
SELECT item_count
FROM items_per_order
WHERE order_occurrences =
  (SELECT MAX(order_occurrences) FROM items_per_order);

-- Method #2
SELECT item_count
FROM items_per_order
WHERE order_occurrences =
  (SELECT SELECT MODE() WITHIN GROUP (ORDER BY order_occurrences DESC)
  FROM items_per_order);
```

Step 3

Order the result by `item_count` in ascending order.

```
-- Method #1
SELECT item_count
FROM items_per_order
WHERE order_occurrences =
  (SELECT MAX(order_occurrences) FROM items_per_order)
ORDER BY item_count;

-- Method #2
SELECT item_count
FROM items_per_order
WHERE order_occurrences =
  (SELECT MODE() WITHIN GROUP (ORDER BY order_occurrences DESC)
  FROM items_per_order)
ORDER BY item_count;
```

Q11:

Step 1

First, we need to find out the month and year each card was launched. This could be done by using the `MIN()` function.

Hold on.. not so fast! If we would take the minimums for the year and month columns separately, the output wouldn't be accurate.

For example in the table below, the launch month for the Chase Sapphire Reserve card could be wrongly interpreted as 1/2021 although it was launched earlier on 11/2020.

card_name	issued_amount	issue_month	issue_year
Chase Sapphire Reserve	150000	11	2020
Chase Sapphire Reserve	160000	12	2020
Chase Sapphire Reserve	170000	1	2021
Chase Sapphire Reserve	175000	2	2021
Chase Sapphire Reserve	180000	3	2021

Thus, it is important to create a combination of the year and month using the `MAKE_DATE()` or `CONCAT()` functions. The following query shows the creation of the dates using `MAKE_DATE()`. As the table has no date column, we use `1` as the first day of the month.

```
SELECT MAKE_DATE(issue_year, issue_month, 1)
FROM monthly_cards_issued;
```

Step 2

Now we are ready to use the `MIN()` window function to obtain the launch dates. As the launch dates are different for both cards, it is important to partition by the `card_name`.

```
SELECT
  card_name,
  issued_amount,
  MAKE_DATE(issue_year, issue_month, 1) AS issue_date,
  MIN(MAKE_DATE(issue_year, issue_month, 1)) OVER (
    PARTITION BY card_name) AS launch_date
FROM monthly_cards_issued;
```

Step 3

For the next step, simply create a CTE from the previous query and select the records where the `issue_date` is equal to the `launch_date`. Finally, order the results from the biggest to the lowest issued amount.

```
WITH card_launch AS (
SELECT
  card_name,
  issued_amount,
  MAKE_DATE(issue_year, issue_month, 1) AS issue_date,
  MIN(MAKE_DATE(issue_year, issue_month, 1)) OVER (
    PARTITION BY card_name) AS launch_date
FROM monthly_cards_issued
)

SELECT card_name, issued_amount
FROM card_launch
WHERE issue_date = launch_date
ORDER BY issued_amount DESC;
```

Q12:

Here's our algorithm for solving this question:

1. Join the tables to obtain the caller's and receiver's country information.
2. Count the international calls and the total number of calls.
3. Calculate the percentage of international calls.

Step 1

To determine whether a call is international or not, we need `country_id` for both caller and receiver. This can be achieved by joining `phone_info` twice, first for the caller, and second for the receiver.

```
SELECT
  caller.country_id AS caller_country,
  receiver.country_id AS receiver_country
FROM phone_calls AS calls
LEFT JOIN phone_info AS caller
  ON calls.caller_id = caller.caller_id
LEFT JOIN phone_info AS receiver
  ON calls.receiver_id = receiver.caller_id;
```

Step 2

After obtaining the necessary info, we can start with the calculation. To do so, we need 2 metrics:

1. number of total calls
2. number of international calls

Getting the **number of total calls** is easy with `COUNT(*)`.

As for the **number of international calls**, we can use the `CASE` statement to check if the caller's country is different from the receiver's country. If it is, assign the value of 1 for international calls, otherwise `NULL` for non-international calls. This is known as a conditional count function.

Then, we wrap the `CASE` statement with either `SUM()` or `COUNT()` to add up the numbers with a value of 1 to obtain the count of international calls.

```
SELECT
  SUM(CASE
    WHEN caller.country_id <> receiver.country_id THEN 1 ELSE NULL END)
AS international_calls,
  COUNT(*) AS total_calls
FROM phone_calls AS calls
LEFT JOIN phone_info caller
  ON calls.caller_id=caller.caller_id
LEFT JOIN phone_info receiver
  ON calls.receiver_id=receiver.caller_id;
```

caller_country	receiver_country	international
DE	US	1
US	US	0
IN	IN	0

Step 3

For the final step, we calculate the **percentage of international calls**. To obtain this, divide the count of international calls by the total.

However, it is important to multiply this result with 100.0 (instead of 100) due to integer division. To elaborate, as both numbers are integers, the result would be also an integer, truncating the decimals from the result.

Lastly, round the result to 1 decimal.

We are done, good job!

```
SELECT
  ROUND(
    100.0 * SUM(CASE
      WHEN caller.country_id <> receiver.country_id THEN 1 ELSE NULL
END)
    /COUNT(*) ,1) AS international_call_pct
FROM phone_calls AS calls
LEFT JOIN phone_info AS caller
  ON calls.caller_id = caller.caller_id
LEFT JOIN phone_info AS receiver
  ON calls.receiver_id = receiver.caller_id;
```

Solution #2: Using FILTER Calling out user @acejetman12345 for this fantastic solution using `FILTER`, which we have adapted slightly.

```
SELECT
  ROUND(
    100.0 * COUNT(*) FILTER (
      WHERE caller.country_id <> receiver.country_id)
  / COUNT(*), 1) AS international_calls_pct
FROM phone_calls AS calls
LEFT JOIN phone_info AS caller
  ON calls.caller_id = caller.caller_id
LEFT JOIN phone_info AS receiver
  ON calls.receiver_id = receiver.caller_id;
```

Solution #3: Using CTE

Instead of filtering the `caller.country_id <> receiver.country_id` in the `CASE` statement, we're doing it in the `WHERE` clause.

```
WITH international_calls AS (
SELECT
  caller.caller_id,
  caller.country_id,
  receiver.caller_id,
  receiver.country_id
FROM phone_calls AS calls
LEFT JOIN phone_info AS caller
  ON calls.caller_id = caller.caller_id
```

```
LEFT JOIN phone_info AS receiver
  ON calls.receiver_id = receiver.caller_id
WHERE caller.country_id <> receiver.country_id
)

SELECT
  ROUND(
    100.0 * COUNT(*)
  / (SELECT COUNT(*) FROM phone_calls),1) AS international_call_pct
FROM international_calls;
```