Q1:
In order to calculate the active user retention, we need to identify users who were active in both the current month and the previous month. We can approach this in two steps:

Step 1: Find users who were active in the previous month

To accomplish this, we can use the following code:

```
SELECT last_month.user_id
FROM user_actions AS last_month
WHERE last_month.user_id = curr_month.user_id
  AND EXTRACT(MONTH FROM last_month.event_date) =
  EXTRACT(MONTH FROM curr_month.event_date - interval '1 month'
```

- We alias the `user_actions` table as `last_month` in the FROM clause to indicate that this table contains information from the previous month.
- In the WHERE clause, we match users in the `last_month` table to the `curr_month` table based on user ID.
- `EXTRACT(MONTH FROM last_month.event_date) = EXTRACT(MONTH FROM curr_month.event_date - interval '1 month')` means that a particular user from the last month exists in the current month, indicating that this user was active in both months.
- `- interval '1 month'` subtracts one month from the current month's date to give us last month's date.

Important: Please note that you won't be able to run this query on its own, as it references another table `curr_month` which is not included in this query. We will provide you with the complete query to run later. For now, focus on understanding the logic behind the solution."

Step 2: Find users who are active in the current month and also existed in the previous month

To accomplish this, we use the `EXISTS` operator to check for users in the current month who also exist in the subquery, which represents active users in the previous month (identified in Step 1). Note that the `user_actions` table is aliased as `curr_month` to indicate that this represents the current month's user information.

To extract the month information from the `event_date` column, we use the `EXTRACT` function. Then, we use a `COUNT DISTINCT` over the `user_id` to obtain the count of monthly active users for the current month.

Here's the complete query to run in the editor:

```
SELECT
  EXTRACT(MONTH FROM curr_month.event_date) AS mth,
  COUNT(DISTINCT curr_month.user_id) AS monthly_active_users
FROM user_actions AS curr_month
WHERE EXISTS (
  SELECT last_month.user_id
  FROM user_actions AS last_month
```

```
  WHERE last_month.user_id = curr_month.user_id
    AND EXTRACT(MONTH FROM last_month.event_date) =
    EXTRACT(MONTH FROM curr_month.event_date - interval '1 month')
)
  AND EXTRACT(MONTH FROM curr_month.event_date) = 7
  AND EXTRACT(YEAR FROM curr_month.event_date) = 2022
GROUP BY EXTRACT(MONTH FROM curr_month.event_date);
```

Results:

month	monthly_active_users
7	2

In July 2022, there are only 2 distinct users who are active in both the previous month and the current month.

Q2:
Our goal for this question is to find the year-on-year (y-o-y) growth rate for Wayfair's user spend.

Our multi-step approach for solving the question:

1. Summarising user_transactions table into a table containing the yearly spend information.
2. Find the prior year's spend and keep the information parallel with current year's spend row.
3. Get the variance between current year and prior year's spend and apply the y-o-y growth rate formula.

Step 1

First, we need to obtain the year by using `EXTRACT` on the transaction date as written in the code below.

```
SELECT
  EXTRACT(YEAR FROM transaction_date) AS year,
  product_id,
  spend AS curr_year_spend
FROM user_transactions
```

In this output table, you can see that the spend for product id 234412 is summarised by year. Take note that you would have to query your output for all the product ids.

year	product_id	curr_year_spend
2019	234412	1800.00
2020	234412	1234.00
2021	234412	889.50
2022	234412	2900.00

Step 2

Next, we convert the query in step 1 into a CTE called `yearly_spend` (you can name the CTE as you wish). With this CTE, we then calculate the prior year's spend for each product. We can do so by applying `LAG` function onto each year and partitioning by product id to calculate the prior year's spend for the given product.

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

SELECT
  *,
  LAG(curr_year_spend, 1) OVER (
    PARTITION BY product_id
    ORDER BY product_id, year) AS prev_year_spend
FROM yearly_spend;
```

Showing the output only for product id 134412:

year	product_id	curr_year_spend	prev_year_spend
2019	234412	1800.00	
2020	234412	1234.00	1800.00
2021	234412	889.50	1234.00
2022	234412	2900.00	889.50

In 2020, the previous year's spend is 1800.00 which is the current year's spend for year 2019. Subsequently, in the year 2021, 1234.00 which is the previous year's spend is the current year's spend in 2020.

Step 3

Finally, we wrap the query above in another CTE called `yearly_variance`.

Year-on-Year Growth Rate = (Current Year's Spend - Prior Year's Spend) / Prior Year's Spend x 100

In the final query, we apply the y-o-y growth rate formula and round the results to 2 nearest decimal places.

```
WITH yearly_spend
AS (
-- Insert above query
),
yearly_variance
AS (
-- Insert above query
)

SELECT
  year,
  product_id,
  curr_year_spend,
  prev_year_spend,
  ROUND(100 * (curr_year_spend - prev_year_spend)/ prev_year_spend, 2)
AS yoy_rate
FROM yearly_variance;
```

Results for product id 134412:

year	product_id	curr_year_spend	prev_year_spend	yoy_rate
2019	234412	1800.00		
2020	234412	1234.00	1800.00	-31.44
2021	234412	889.50	1234.00	-27.92
2022	234412	2900.00	889.50	226.03

Solution:

```
WITH yearly_spend AS (
  SELECT
    EXTRACT(YEAR FROM transaction_date) AS year,
    product_id,
    spend AS curr_year_spend
  FROM user_transactions
),
yearly_variance AS (
```

```
  SELECT
    *,
    LAG(curr_year_spend, 1) OVER (
      PARTITION BY product_id
      ORDER BY product_id, year) AS prev_year_spend
  FROM yearly_spend)

SELECT
  year,
  product_id,
  curr_year_spend,
  prev_year_spend,
  ROUND(100 * (curr_year_spend - prev_year_spend)/ prev_year_spend,2)
AS yoy_rate
FROM yearly_variance;
```

Q3:
Goal: Find the maximum number of prime and non-prime items that can be stocked in a 500,000 sq ft warehouse, with prime items being prioritised.

Step 1: Calculate the total square footage and the number of items for both prime and non-prime items as separate groups

To start, we use the following query to summarise the relevant information:

```
SELECT
  item_type,
  SUM(square_footage) AS total_sqft,
  COUNT(*) AS item_count
FROM inventory
GROUP BY item_type;
```

This query will return the total square footage and the item count for each item type (prime and non-prime).

item_type	total_sqft	item_count
prime_eligible	555.20	6
not_prime	128.50	4

Step 2: Determine the maximum number of prime items that can be stored in the 500,000 sq ft warehouse

Next, we need to figure out the maximum number of prime items that can be stored in the warehouse by dividing the warehouse's area by the total area of prime items and rounding down to the nearest integer.

```
WITH summary
AS (
  SELECT
    item_type,
    SUM(square_footage) AS total_sqft,
    COUNT(*) AS item_count
  FROM inventory
  GROUP BY item_type
)

SELECT
  item_type,
  total_sqft
  FLOOR(500000/total_sqft) AS prime_item_combination_count,
  (FLOOR(500000/total_sqft) * item_count) AS prime_item_count
FROM summary
WHERE item_type = 'prime_eligible';
```

item_type	total_sqft	prime_item_combination_count	prime_item_count
prime_eligible	555.20	900	5400

The result shows that the warehouse can stock 900 times of prime items which can be mathematically expressed as:

900 times x 555.20 sq ft = 499,680 sq ft area occupied by prime items

The remaining warehouse space to stock non-prime items is **500,000 sq ft - 499,680 sq ft = 320 sq ft**.

Step 3: Calculate the maximum number of non-prime items that can be stored

Finally, we need to calculate the maximum number of non-prime items that can be stored in the warehouse.

(1) Calculate the maximum number of prime items

We calculated this in the previous step and the maximum number of prime items is 5,400 prime items.

(2) Calculate the maximum number of non-prime items

Storage area - (total area occupied by prime items) / area occupied by non-prime items * number of non-prime items per area

= FLOOR(500,000 sq ft - (900 x 555.20 prime sq ft)) / 128.50 non-prime sq ft* 4 items = 8 non-prime items

Output the results with prime items followed by non-prime items.

Here is the final query to accomplish this:

```
WITH summary AS (
  SELECT
    item_type,
    SUM(square_footage) AS total_sqft,
    COUNT(*) AS item_count
  FROM inventory
  GROUP BY item_type
),
prime_occupied_area AS (
  SELECT
    item_type,
    total_sqft,
    FLOOR(500000/total_sqft) AS prime_item_combination_count,
    (FLOOR(500000/total_sqft) * item_count) AS prime_item_count
  FROM summary
  WHERE item_type = 'prime_eligible'
)

SELECT
  item_type,
  CASE
    WHEN item_type = 'prime_eligible'
      THEN (FLOOR(500000/total_sqft) * item_count)
    WHEN item_type = 'not_prime'
      THEN FLOOR((500000 -
        (SELECT FLOOR(500000/total_sqft) * total_sqft FROM
prime_occupied_area))
        / total_sqft)
        * item_count
  END AS item_count
FROM summary
ORDER BY item_type DESC;
```

Final results:

item_type	item_count
prime_eligible	5400
not_prime	8

Method #2: Using `FILTER` and `UNION ALL` operator

```
WITH summary AS (
SELECT
  SUM(square_footage) FILTER (WHERE item_type = 'prime_eligible') AS
prime_sq_ft,
  COUNT(item_id) FILTER (WHERE item_type = 'prime_eligible') AS
prime_item_count,
  SUM(square_footage) FILTER (WHERE item_type = 'not_prime') AS
not_prime_sq_ft,
  COUNT(item_id) FILTER (WHERE item_type = 'not_prime') AS
not_prime_item_count
FROM inventory
),
prime_occupied_area AS (
SELECT
  FLOOR(500000/prime_sq_ft)*prime_sq_ft AS max_prime_area
FROM summary
)

SELECT
  'prime_eligible' AS item_type,
  FLOOR(500000/prime_sq_ft)*prime_item_count AS item_count
FROM summary

UNION ALL

SELECT
  'not_prime' AS item_type,
  FLOOR((500000-(SELECT max_prime_area FROM prime_occupied_area))
    / not_prime_sq_ft) * not_prime_item_count AS item_count
FROM summary;
```

Q4:
Start by forming a common table expression (CTE) or subquery that will expand each search by the number of users' values.

Using the `GENERATE_SERIES()` function, we can pass in 1 as the start value and the num_users value as the stop value. This will give us a set of numbers, `[1, 1, 2, 2, 3, 3, 3, 4]`, for example.

Since PostgreSQL does not provide a function to help us find the median value, we can use the `PERCENTILE_CONT()` function. This function takes the percentile required as an argument, in this case it is 0.5, i.e. the 50th percentile (MEDIAN). The `WITHIN GROUP` clause creates an ordered subset of search values returned by the CTE/Subquery mentioned above that can be used to perform aggregations. Finally, to make sure our output is rounded to one decimal place,

you can use the `ROUND()` function. Converting the value returned by the `PERCENTILE_CONT` to a decimal using `::decimal` is necessary since it may be an integer value which won't allow the `ROUND` function to work properly.

```
WITH searches_expanded AS (
  SELECT searches
  FROM search_frequency
  GROUP BY
    searches,
    GENERATE_SERIES(1, num_users))

SELECT
  ROUND(PERCENTILE_CONT(0.50) WITHIN GROUP (
    ORDER BY searches)::DECIMAL, 1) AS  median
FROM searches_expanded;
```

Q5:

Step 1:

First, we merge the `advertiser` and `daily_pay` tables with `LEFT JOIN`.

```
SELECT
  advertiser.user_id,
  advertiser.status,
  payment.paid
FROM advertiser
LEFT JOIN daily_pay AS payment
  ON advertiser.user_id = payment.user_id;
```

Showing first 5 rows out of 8 rows:

user_id	status	paid
bing	NEW	
yahoo	NEW	45.00
alibaba	EXISTING	100.00
baidu	EXISTING	
target	CHURN	13.00

Do you know why we're joining the tables using `LEFT JOIN` instead of (INNER) `JOIN`?

Say, if we use `JOIN` instead, this is the output we'll get:

user_id	status	paid
yahoo	NEW	45.00
alibaba	EXISTING	100.00
target	CHURN	13.00

Bing and Baidu are no longer in the output because they have not made any payment yet and do not exist in the `daily_pay` table. Since the question wants us to tag their payment status regardless of payment made, we have to ensure that we gather all the advertisers' payment information.

Step 2

Did you realise that one of the advertisers is missing from the table? If you query `SELECT * FROM daily_pay`, you will notice that Fitdata is missing. That's because Fitdata is a new advertiser and its data is not reflected in the advertiser table yet.

To ensure that we have a complete table with all the existing and new advertisers, we have to merge both tables vertically.

After this, we `LEFT JOIN daily_pay` with the advertiser. This will let us know which users are new. At last, we join the results produced by these two left joins using `UNION`. If paid is null, then that user have not paid at day t and if status is null then the user is a new user(who's information is not present in `advertiser` table but that user paid on day t, so this particular user is called as new user)

A graphical presentation of how the UNION looks:

```
SELECT ...
FROM advertiser
LEFT JOIN daily_pay -- Find out who pay and did not pay
UNION
SELECT ...
FROM daily_pay
LEFT JOIN advertiser -- Find out if there's new paid user
```

And, how the complete query looks:

```
SELECT
  advertiser.user_id,
  advertiser.status,
  payment.paid
FROM advertiser
LEFT JOIN daily_pay AS payment
  ON advertiser.user_id = payment.user_id
UNION
SELECT
```

```
  payment.user_id,
  advertiser.status,
  payment.paid
FROM daily_pay AS payment
LEFT JOIN advertiser
  ON advertiser.user_id = payment.user_id
```

Bear in mind that the second query has the `daily_pay` table being the left table as the Fitdata advertiser is sitting in this table.

Now you should have the complete table with 9 rows:

user_id	status	paid
fitdata		25.00
baidu	EXISTING	
morgan	RESURRECT	600.00
target	CHURN	13.00
alibaba	EXISTING	100.00

Interpreting this table based on the transitions table:

- Fit Data is a new advertiser who has paid $25 with its new status being NEW.
- Baidu and Alibaba are existing advertisers, but only Alibaba has made a payment and remains as EXISTING whereas Baidu is CHURN.
- Since Morgan has made a payment, its new status will be changed to EXISTING.
- Target's CHURN status will be changed to RESURRECT.

Step 3:

Our final step is to assign each advertiser to its new status using a conditional CASE statement based on the payment conditions set out:

- **New**: users registered and made their first payment.
- **Existing**: users who paid previously and recently made a current payment.
- **Churn**: users who paid previously, but have yet to make any recent payment.
- **Resurrect**: users who did not pay recently but may have made a previous payment and have made payment again recently.

```
WITH payment_status
AS (
-- Insert query in step 2)

SELECT
  user_id,
  CASE WHEN paid IS NULL THEN 'CHURN'
    WHEN status != 'CHURN' AND paid IS NOT NULL THEN 'EXISTING'
    WHEN status = 'CHURN' AND paid IS NOT NULL THEN 'RESURRECT'
    WHEN status IS NULL THEN 'NEW'
  END AS new_status
FROM payment_status
ORDER BY user_id;
```

Results:

user_id	new_status
fitdata	NEW
baidu	CHURN
morgan	EXISTING
target	RESURRECT
alibaba	EXISTING

Solution:

```
WITH payment_status AS (
SELECT
  advertiser.user_id,
  advertiser.status,
  payment.paid
FROM advertiser
LEFT JOIN daily_pay AS payment
  ON advertiser.user_id = payment.user_id

UNION

SELECT
  payment.user_id,
  advertiser.status,
  payment.paid
FROM daily_pay AS payment
LEFT JOIN advertiser
  ON advertiser.user_id = payment.user_id
)

SELECT
  user_id,
```

```
  CASE WHEN paid IS NULL THEN 'CHURN'
      WHEN status != 'CHURN' AND paid IS NOT NULL THEN 'EXISTING'
      WHEN status = 'CHURN' AND paid IS NOT NULL THEN 'RESURRECT'
      WHEN status IS NULL THEN 'NEW'
  END AS new_status
FROM payment_status
ORDER BY user_id;
```

Q6: We have developed 4 solutions to solve this question:

1. Using INNER JOIN
2. Using CROSS JOIN
3. Using RECURSIVE CTE
4. USING Array function

Solution #1: Using INNER JOIN

Step 1: Join the tables thrice horizontally

First, we create a table where the ingredients are displayed in horizontal style 3 times. That will make the concatenation easier.

```
SELECT *
FROM pizza_toppings AS p1
INNER JOIN pizza_toppings AS p2
  ON p1.topping_name < p2.topping_name
INNER JOIN pizza_toppings AS p3
  ON p2.topping_name < p3.topping_name;
```

Important tip: Instead of matching the tables on columns using the equal sign `=`, we use the **less than** sign `<` to ensure that:

1. There is no duplication of the ingredients across each row.
2. The ingredients are arranged in an alphabetical manner from left to right.

Showing the output of the first 3 rows:

topping_name	ingredient_cost	topping_name	ingredient_cost	topping_name	ingredient_cost
Pepperoni	0.50	Sausage	0.70	Spinach	0.30
Pepperoni	0.50	Pineapple	0.25	Sausage	0.70
Pepperoni	0.50	Pineapple	0.25	Spinach	0.30

Step 2: Combine pizza toppings and add the ingredients

Next, we will design the table based on the required output, which is a 3-topping combination and total ingredient cost.

To combine the 3 toppings into a single string, we can either use the `CONCAT()` function or pipe `||`.

```
CONCAT(expression1, expression2, expression3, ...)
```

```
expression1 || expression2 || expression3
```

We'll use the `CONCAT()` function for this question. As for the total ingredient costs, just add them up using the plus sign `+` as you would use a calculator.

```
SELECT
  CONCAT(p1.topping_name, ',', p2.topping_name, ',', p3.topping_name)
AS pizza,
  p1.ingredient_cost + p2.ingredient_cost + p3.ingredient_cost AS
total_cost
FROM pizza_toppings AS p1
INNER JOIN pizza_toppings AS p2
  ON p1.topping_name < p2.topping_name
INNER JOIN pizza_toppings AS p3
  ON p2.topping_name < p3.topping_name;
```

pizza	total_cost
Pepperoni,Sausage,Spinach	1.50
Pepperoni,Pineapple,Sausage	1.45
Pepperoni,Pineapple,Spinach	1.05

Step 3: Order the output accordingly

Finally, order the output by the highest cost at the top and to break ties, sort the pizza ingredients in ascending order (it also means sort in alphabetical order.

```
SELECT
  CONCAT(p1.topping_name, ',', p2.topping_name, ',', p3.topping_name)
AS pizza,
  p1.ingredient_cost + p2.ingredient_cost + p3.ingredient_cost AS
total_cost
FROM pizza_toppings AS p1
INNER JOIN pizza_toppings AS p2
  ON p1.topping_name < p2.topping_name
INNER JOIN pizza_toppings AS p3
  ON p2.topping_name < p3.topping_name
ORDER BY total_cost DESC, pizza;
```

Solution #2: Using CROSS JOIN

Let's split the problem into 3 slices.

1. Get every possible combination of 3-topping pizzas
2. Discard rows with duplicate toppings and arrange them according to alphabetical order
3. Order output by lowest to highest cost

Step 1: 3-Topping Combinations

Using a neat trick with `CROSS JOIN`, we're joining the `pizza_toppings` table horizontally 3 times to obtain an output with 3 columns of the `topping_name`.

```
SELECT *
FROM pizza_toppings AS p1
CROSS JOIN
  pizza_toppings AS p2,
  pizza_toppings AS p3;
```

Output of randomly selected 5 rows:

topping_name	ingredient_cost	topping_name	ingredient_cost	topping_name	ingredient_cost
Pepperoni	0.50	Pepperoni	0.50	Pepperoni	0.50
Pepperoni	0.50	Pepperoni	0.50	Sausage	0.70
Pepperoni	0.50	Pepperoni	0.50	Chicken	0.55
Pepperoni	0.50	Pepperoni	0.50	Extra Cheese	0.40
Pepperoni	0.50	Pepperoni	0.50	Mushrooms	0.25

Step 2: Discard duplicates and arrange them alphabetically

Our idea is to concatenate the topping names across a single row, however, in the above output, we have rows where the topping names are duplicates of each other (i.e. [Pepperoni, Pepperoni, Pepperoni], [Pepperoni, Pepperoni, Sausage]).

To ensure that the topping names across a row are different, we use comparison operators in the `WHERE` clause.

```
SELECT *
FROM pizza_toppings AS p1
CROSS JOIN
  pizza_toppings AS p2,
  pizza_toppings AS p3
WHERE p1.topping_name < p2.topping_name
  AND p2.topping_name < p3.topping_name;
```

Using `<` for string values is a handy trick to prevent duplication. Not only that, remember the question's requirement of **"break ties by listing the ingredients in alphabetical order"**?

By writing `p1.topping_name < p2.topping_name`, it ensures that the second topping's name (`p2.topping_name`) comes after the first topping's name (`p1.topping_name`). Isn't this cool?

Step 3: Return desired output

To retrieve the combination of 3-topping values in a single cell, we can use the CONCAT function and then add up all the ingredient's cost.

```
SELECT
  CONCAT(p1.topping_name, ',', p2.topping_name, ',', p3.topping_name)
AS pizza,
  p1.ingredient_cost + p2.ingredient_cost + p3.ingredient_cost AS
total_cost
FROM pizza_toppings AS p1
CROSS JOIN
  pizza_toppings AS p2,
  pizza_toppings AS p3
WHERE p1.topping_name < p2.topping_name
  AND p2.topping_name < p3.topping_name
ORDER BY total_cost DESC, pizza;
```

Solution #3: Using RECURSIVE CTE

Using the recursive Common Table Expression (also known as **recursive CTE**), we will find every possible combination of 3-toppings pizzas.

The term **recursive CTE** refers to a subquery that references itself. There are essentially 3 parts to it:

1. Non-recursive query (AKA the base query or anchor),
2. Recursive query, and
3. Termination condition

Note that strictly speaking, this process is iteration not recursion, but RECURSIVE is the terminology chosen by the SQL standards committee; see PostgreSQL Documentation.

Here is how it is structured in PostgreSQL:

```
WITH RECURSIVE [cte_name] AS (
-- Write the non-recursive query

UNION ALL

-- Write the recursive query
```

```
WHERE -- Write a termination condition(s) to stop the recursive loop
);
```

For our problem, let's start by enclosing the non-recursive query in the recursive CTE.

```
WITH RECURSIVE all_toppings AS (
-- Let's start with the 1st element: Non-recursive query
SELECT
  topping_name,
  ingredient_cost,
  1 AS topping_numbers
FROM pizza_toppings
)

-- Querying the non-recursive query
SELECT
  topping_name,
  ingredient_cost,
  topping_numbers
FROM all_toppings;
```

Displaying randomly selected 4 records in the output:

topping_name	ingredient_cost	topping_numbers
Pepperoni	0.50	1
Sausage	0.70	1
Chicken	0.55	1
Extra Cheese	0.40	1

A newly created column `topping_numbers` is introduced which serves as a numerical representation of the number of toppings each pizza has. Since each of these toppings in the output is a separate item, the topping number would equal to 1. As more toppings are placed onto the pizza, this `topping_numbers` will increase incrementally.

We will use this non-recursive query (or the **anchor**) and iterate through to create different combinations of toppings using a recursive CTE.

Let's incorporate the recursive query inside this CTE – this is the part that makes it recursive.

```
WITH RECURSIVE all_toppings AS (
-- 1st element: Non-recursive query
SELECT
  topping_name::VARCHAR,
  ingredient_cost::DECIMAL AS total_cost,
  1 AS topping_numbers
FROM pizza_toppings

-- 2nd element: UNION ALL
```

```
UNION ALL

-- 3rd element: Recursive query
SELECT
  CONCAT(addon.topping_name, ',', anchor.topping_name) AS topping_name,
  addon.ingredient_cost + anchor.total_cost AS total_cost,
  topping_numbers + 1
FROM
  pizza_toppings AS addon,
  all_toppings AS anchor
WHERE anchor.topping_name < addon.topping_name
)

-- Querying the output
SELECT *
FROM all_toppings;
```

Note that a recursive CTE must have all 3 of the elements in place for it to work.

The first requirement of `UNION ALL` is:

- Each `SELECT` operation within the `UNION ALL` must produce result sets with an identical number of fields and data types.
- Due to syntax limitations in PostgreSQL, we have to explicitly cast the columns `topping_name` and `ingredient_cost` AS `VARCHAR` and `DECIMAL` datatypes in the non-recursive query, respectively.
    - PostgreSQL considers (before casting) `VARCHAR(255)` and (after casting) `VARCHAR` to be two different entities.
    - The `topping_name` field in the non-recursive query is of the `VARCHAR(255)` datatype as defined in the table.
    - After casting, the `topping_name` in the recursive query becomes a `VARCHAR` data type.
    - Similarly, we cast`ingredient_cost` to `DECIMAL` as well.

Note the **termination condition** in the recursive query: `WHERE anchor.topping_name < addon.topping_name`. It compares the topping names between the non-recursive and recursive query and prevents the same topping names from showing up multiple times in the 3-topping combination. This recursive query will continue to run until the `WHERE` clause condition is met.

Let's interpret the `SELECT` clause of the **recursive query**:

- The `CONCAT` function is used to combine two or more toppings.
- The cost of the new topping will be added to the expenses of the prior toppings.
- When a topping is put on a pizza, the topping counter (`topping_numbers`) will be incremented.

The recursive CTE will continue to run as more toppings are introduced until all possible combinations have been found.

Each new result is added to the preceding result set using the `UNION ALL` operator.

Showing 5 records for toppings **Pepperoni, Extra Cheese, Sausage and Chicken**.

topping_name	total_cost	topping_numbers
Pepperoni	0.50	1
Sausage,Pepperoni	1.25	2
Extra Cheese,Chicken	0.95	2
Pepperoni,Extra Cheese,Chicken	1.45	3
Sausage,Extra Cheese,Chicken	1.65	3

We can filter for pizzas with 3 toppings, or `WHERE topping_numbers = 3`:

```
WITH RECURSIVE all_toppings AS (
-- Insert the previous query here
)

SELECT *
FROM all_toppings
WHERE topping_numbers = 3;
```

Output:

topping_name	total_cost	topping_numbers
Sausage,Pepperoni,Chicken	1.75	3
Pepperoni,Extra Cheese,Chicken	1.45	3
Sausage,Extra Cheese,Chicken	1.65	3
Sausage,Pepperoni,Extra Cheese	1.60	3

Step 2: Arrange toppings alphabetically

We'll use the following steps to arrange each set of pizza toppings alphabetically:

1. Divide the toppings up into multiple rows.
2. Sort them alphabetically and combine the toppings from the same group.

We will split the toppings using the function `REGEXP_SPLIT_TO_TABLE`. It splits the values in the form of rows based on the specified delimiter. Let's call this column `single_topping`.

```
WITH RECURSIVE all_toppings AS (
-- Insert the previous query here
)

SELECT
  topping_name,
  total_cost,
  single_topping
FROM
  all_toppings,
  REGEXP_SPLIT_TO_TABLE(topping_name, ',') AS single_topping
WHERE topping_numbers = 3;
```

Showing how toppings are split for a pizza with **Sausage, Pepperoni, Chicken** toppings.

topping_name	total_cost	single_topping
Sausage,Pepperoni,Chicken	1.75	Pepperoni
Sausage,Pepperoni,Chicken	1.75	Sausage
Sausage,Pepperoni,Chicken	1.75	Chicken

The values in the `single_topping` column will then be sorted and combined to restore their previous groupings.

One function, `STRING_AGG`, can do both of these things! `STRING_AGG` joins sets of strings into a single string. The inner `ORDER BY` clause then sets the order of the concatenated results.

```
STRING_AGG (single_topping, ',' , ORDER BY single_topping )
```

The output is then arranged in descending order of total cost, followed by the names of the toppings.

We've reached the end of this long road, and our final query is ready :)

```
WITH RECURSIVE all_toppings AS (
  SELECT
  topping_name::VARCHAR,
  ingredient_cost::DECIMAL AS total_cost,
  1 AS topping_numbers
FROM pizza_toppings

UNION ALL

SELECT
  CONCAT(addon.topping_name, ',', anchor.topping_name) AS topping_name,
  addon.ingredient_cost + anchor.total_cost AS total_cost,
  topping_numbers + 1
FROM
  pizza_toppings AS addon,
  all_toppings AS anchor
```

```
WHERE anchor.topping_name < addon.topping_name
)

SELECT
  STRING_AGG (single_topping, ',' ORDER BY single_topping) AS pizza,
  total_cost
FROM
  all_toppings,
  REGEXP_SPLIT_TO_TABLE(topping_name, ',') AS single_topping
WHERE topping_numbers = 3
GROUP BY topping_name, total_cost
ORDER BY total_cost DESC, pizza;
```

Solution #4: Using Array Function

```
WITH RECURSIVE all_toppings AS (
SELECT
  topping_name::VARCHAR,
  ingredient_cost::DECIMAL AS total_cost,
  1 AS topping_numbers
FROM pizza_toppings

UNION ALL

SELECT
  CONCAT(addon.topping_name, ',', anchor.topping_name) AS topping_name,
  addon.ingredient_cost + anchor.total_cost AS total_cost,
  topping_numbers + 1
FROM
  pizza_toppings AS addon,
  all_toppings AS anchor
WHERE anchor.topping_name < addon.topping_name
),
arrange AS (
SELECT
  topping_name,
  UNNEST(STRING_TO_ARRAY(topping_name, ',')) AS single_topping,
  total_cost
FROM all_toppings
WHERE topping_numbers = 3
)

SELECT
  STRING_AGG(single_topping, ',' ORDER BY single_topping) AS pizza,
  total_cost
FROM arrange
GROUP BY topping_name, total_cost
```

```
ORDER BY total_cost DESC, pizza;
```

Q7:

The following steps will be used to find patients who made calls within 7 days of their previous call.

1. Get the previous call for each logged call.
2. Calculate the difference between a patient's current and immediate previous call.
3. Convert the time difference to days.
4. Count the number of unique patients.

Step 1: Find the most recent call for each logged call

For each logged call, we have to retrieve the previous call as well using the `call_received` column.

To get data from the previous row, the row before the previous row, and so on, we can use the `LAG()` window function. Take a quick look here to read about the `LAG()` function.

```
SELECT
  policy_holder_id,
  call_received,
  LAG(call_received) OVER (
      PARTITION BY policy_holder_id
    ORDER BY call_received) AS previous_call
FROM callers;
```

Displaying a few randomly selected records for patients 51435044 and 50986511:

policy_holder_id	call_received	previous_call
51435044	01/18/2022 02:46:00	
51435044	04/11/2022 02:41:00	01/18/2022 02:46:00
51435044	04/18/2022 21:58:00	04/11/2022 02:41:00
50986511	01/28/2022 09:46:00	
50986511	01/31/2022 04:44:00	01/28/2022 09:46:00
50986511	01/31/2022 16:36:00	01/31/2022 04:44:00

Let's interpret the `LAG()` function:

- `PARTITION BY` clause separates the rows by unique patients i.e. `policy_holder_id`
- Each patient-partition is applied with a separate application of the `LAG()` function and is restarted for each patient-partition.

- Rows in each partition are sorted based on the `call_received` in the `ORDER BY` clause.

Step 2: Difference between a patient's current and previous logged calls

For each record, subtract the `previous_call` column's values from the `call_received` column's values.

```
SELECT
  policy_holder_id,
  call_received AS current_call,
  LAG(call_received) OVER (
      PARTITION BY policy_holder_id
    ORDER BY call_received
  ) AS previous_call, -- Maintaining for demo purposes. It will not be part of the final solution
  call_received - LAG(call_received) OVER (
   PARTITION BY policy_holder_id
   ORDER BY call_received) AS time_difference
FROM callers;
```

Output:

policy_holder_id	current_call	previous_call	time_difference
51435044	01/18/2022 02:46:00		
51435044	04/11/2022 02:41:00	01/18/2022 02:46:00	"days":82,"hours":23,"minutes":55
51435044	04/18/2022 21:58:00	04/11/2022 02:41:00	"days":7,"hours":19,"minutes":17
50986511	01/28/2022 09:46:00		
50986511	01/31/2022 04:44:00	01/28/2022 09:46:00	"days":2,"hours":18,"minutes":58
50986511	01/31/2022 16:36:00	01/31/2022 04:44:00	"hours":11,"minutes":52

The `time_difference` column displays the precise time difference in `days, hours and minutes` format between the `current_call` and `previous_call` columns.

For example,

- Patient **51435044** made their second call after **82 days, 23 hours, and 55 minutes** from their first call. Hence, this is not considered a call within 7 days.
- Patient **50986511** made their second call after **2 days, 18 hours, and 58 minutes** from their first call. This is considered a call within 7 days.

Step 3: Get the time difference in days

Since the time difference contains multiple units of time, we're only taking the `DAY` portion using the `EXTRACT()` function.

But there is a catch!

If only the `DAY` argument is used, we might also wrongly include calls made within 7 hours and xx minutes.

For example:

- User 51435044 made their third call after 7 days, 19 hours, and 17 minutes of their second call. The time difference is outside of the 7-day window.

To solve this, the `EPOCH` function can be used with the `EXTRACT()` function to return in the format of **seconds**.

```
SELECT
  policy_holder_id,
  call_received,
  LAG(call_received) OVER (
      PARTITION BY policy_holder_id
    ORDER BY call_received) AS previous_call,
  EXTRACT(EPOCH FROM
  call_received - LAG(call_received) OVER (
      PARTITION BY policy_holder_id
    ORDER BY call_received)
  ) AS time_difference
FROM callers;
```

policy_holder_id	call_received	previous_call	time_difference
51435044	01/18/2022 02:46:00		
51435044	04/11/2022 02:41:00	01/18/2022 02:46:00	7170900
51435044	04/18/2022 21:58:00	04/11/2022 02:41:00	674220
50986511	01/28/2022 09:46:00		

50986511	01/31/2022 04:44:00	01/28/2022 09:46:00	241080
50986511	01/31/2022 16:36:00	01/31/2022 04:44:00	42720

The `time_difference` column now displays the difference between two calls in **seconds**. To convert it into days, we multiply by the following:

1 day = 24 hours x 60 minutes x 60 seconds

```
SELECT
  policy_holder_id,
  call_received,
  LAG(call_received) OVER (
      PARTITION BY policy_holder_id
    ORDER BY call_received) AS previous_call,
  EXTRACT(EPOCH FROM call_received
    - LAG(call_received) OVER (
        PARTITION BY policy_holder_id
      ORDER BY call_received)
  )/(24*60*60) AS day_difference
FROM callers;
```

policy_holder_id	call_received	previous_call	day_difference
51435044	01/18/2022 02:46:00		
51435044	04/11/2022 02:41:00	01/18/2022 02:46:00	82.99
51435044	04/18/2022 21:58:00	04/11/2022 02:41:00	7.80
50986511	01/28/2022 09:46:00		
50986511	01/31/2022 04:44:00	01/28/2022 09:46:00	2.79
50986511	01/31/2022 16:36:00	01/31/2022 04:44:00	0.49

Step 4: Find patients who made calls within 7 days

Finally, we filter the records to only include calls with no more than seven days difference.

Create a common table expression (CTE) around our previous query and add a `WHERE` clause to it.

Patients will be counted using the `COUNT()` function.

However, as we can see from the preceding output, **patient 50986511** made all three calls within a 7-day period; hence, the patient should only be taken into account once. When counting the patients, the `DISTINCT` keyword is the best to use.

```
WITH call_history AS (
 -- Enter the previous query here
)

SELECT COUNT(DISTINCT policy_holder_id) AS patient_count
FROM call_history
WHERE day_difference <= 7;
```

Result:

patient_count
1

Remove the unnecessary column which was used for demonstration purposes in the CTE. And there you have it - our final query!

Solution #1: Using EPOCH()

```
WITH call_history AS (
SELECT
  policy_holder_id,
  call_received,
   EXTRACT(EPOCH FROM call_received
     - LAG(call_received) OVER (
         PARTITION BY policy_holder_id ORDER BY call_received)
    )/(24*60*60) AS day_difference
FROM callers
)

SELECT COUNT(DISTINCT policy_holder_id) AS patient_count
FROM call_history
WHERE day_difference <= 7;
```

Solution #2: Using INTERVAL

```
WITH calls AS (
SELECT
  policy_holder_id,
  call_received AS current_call,
  LEAD(call_received) OVER (
    PARTITION BY policy_holder_id ORDER BY call_received) AS next_call
FROM callers
)
```

```
SELECT COUNT(DISTINCT policy_holder_id) AS patient_count
FROM calls
WHERE current_call + INTERVAL '168 hours' >= next_call;
```

Q8:
Goal: Find the month-on-month (m-o-m) growth rate for UHG's long calls.

Our multi-step approach for solving the question:

1. Filter for long calls (more than 300 seconds).
2. Summarise data into a table containing monthly long call information.
3. Find the monthly call counts.
4. Calculate the variance between the current and previous month's calls.
5. Apply the m-o-m growth rate formula.

Step 1: Filter for long calls

Long calls are defined to be calls of more than 300 seconds. We can filter for long calls on the `call_duration_secs` field.

```
SELECT *
FROM callers
WHERE call_duration_secs > 300;
```

Step 2: Summarise data into a table containing monthly long call information

Next, we obtain the year and month information in numerical format using the `EXTRACT()` function on the `call_received` field.

Bear in mind not to alias the columns as `year` and `month` as these are SQL's date parts and may confuse SQL whether you're calling the fields or SQL's date parts.

```
SELECT
  EXTRACT(YEAR FROM call_received) AS yr,
  EXTRACT(MONTH FROM call_received) AS mth
FROM callers
WHERE call_duration_secs > 300;
```

Step 3: Find the monthly call counts for the current and previous month

Next, we find the count of calls by year and month using the `COUNT()` aggregate function. Additionally, we are also introducing a `LAG()` window function to generate the previous month's calls.

```
LAG(COUNT(case_id)) OVER (
    ORDER BY EXTRACT(MONTH FROM call_received)) AS prev_mth_call
```

See how we incorporate the `COUNT()` and `EXTRACT()` functions directly into the `LAG()` function? It keeps the query short and sweet.

And, here's how we include it in our query:

```
SELECT
  EXTRACT(YEAR FROM call_received) AS yr,
  EXTRACT(MONTH FROM call_received) AS mth,
  COUNT(case_id) AS curr_mth_call,
  LAG(COUNT(case_id)) OVER (
    ORDER BY EXTRACT(MONTH FROM call_received)) AS prev_mth_call
FROM callers
WHERE call_duration_secs > 300
GROUP BY
  EXTRACT(YEAR FROM call_received),
  EXTRACT(MONTH FROM call_received)
```

Showing the output for the first 5 months:

yr	mth	curr_mth_call	prev_mth_call
2022	1	2	
2022	2	2	2
2022	3	10	2
2022	4	5	10
2022	5	19	5

Let's interpret the output together:

- 2022/1 - January is the first month of the year, hence it doesn't have the previous month's call count in the `prev_mth_call` field.
- 2022/2, 3 - In February, the previous month's 2 calls are pulled from January. Similarly, in March.
- 2022/4 - In April, the previous month's 10 calls are pulled from March.

Step 4: Calculate the variance between the current and previous month's calls

Here's the formula that we're using:

M-on-M Growth Rate = (Current Month's Calls - Previous Month's Calls) / Previous Month's Calls x 100

We have the current and previous month's calls so let's put them into the query.

```
WITH calls AS (
SELECT
  EXTRACT(YEAR FROM call_received) AS yr,
  EXTRACT(MONTH FROM call_received) AS mth,
  COUNT(case_id) AS curr_mth_call,
  LAG(COUNT(case_id)) OVER (
    ORDER BY EXTRACT(MONTH FROM call_received)) AS prev_mth_call
FROM callers
WHERE call_duration_secs > 300
GROUP BY
  EXTRACT(YEAR FROM call_received),
  EXTRACT(MONTH FROM call_received)
)

SELECT
  yr,
  mth,
  100.0 * (curr_mth_call - prev_mth_call)/prev_mth_call AS growth_pct
FROM calls
ORDER BY mth;
```

Showing the output for the first 5 months:

yr	mth	growth_pct
2022	1	
2022	2	0.00000000000000000000
2022	3	400.00000000000000000
2022	4	-50.00000000000000000
2022	5	280.00000000000000000

We want the percentages to be rounded to 1 decimal place, so use a `ROUND()` function on the `growth_pct` field. The result should be returned in chronological sequence so order the months accordingly.

```
WITH calls AS (
SELECT
```

```
  EXTRACT(YEAR FROM call_received) AS yr,
  EXTRACT(MONTH FROM call_received) AS mth,
  COUNT(case_id) AS curr_mth_call,
  LAG(COUNT(case_id)) OVER (
    ORDER BY EXTRACT(MONTH FROM call_received)) AS prev_mth_call
FROM callers
WHERE call_duration_secs > 300
GROUP BY
  EXTRACT(YEAR FROM call_received),
  EXTRACT(MONTH FROM call_received)
)

SELECT
  yr,
  mth,
  ROUND(100.0 *
    (curr_mth_call - prev_mth_call)/prev_mth_call,1) AS growth_pct
FROM calls
ORDER BY yr, mth;
```

Q9:
Here are the steps to solve this question:

1. Bring together the transactions that are a part of the same group.
2. Work out the time difference between the groups' successive transactions.
3. Identify the identical transactions that took place within ten minutes of one another.

Step 1

For each transaction present in the `transactions` table, we will obtain the time of the most recent identical transaction. With the `LAG window function`, it is feasible. This function accesses the specified field's values from the **previous** rows.

Run this query and we will explain more below.

```
SELECT
  transaction_id
  merchant_id,
  credit_card_id,
  amount,
  transaction_timestamp,
  LAG(transaction_timestamp) OVER (
    PARTITION BY merchant_id, credit_card_id, amount
    ORDER BY transaction_timestamp
  ) AS previous_transaction
```

```
FROM transactions;
```

Let's interpret the `LAG` function:

- `PARTITION BY` clause separates the result's rows by the unique merchant ID, credit card and payment.
- Each partition is applied with a separate application of the `LAG` function, and the computation is restarted for each partition (merchant ID, credit card and payment).
- Rows in each partition are sorted based on `transaction_timestamp` in the `ORDER BY` clause.

Showing 4 transactions worth $100 performed at Merchant ID 101 with credit card ID 1:

transaction_id	merchant_id	credit_card_id	amount	transaction_timestamp	previous_transaction
1	101	1	100	09/25/2022 12:00:00	
2	101	1	100	09/25/2022 12:08:00	09/25/2022 12:00:00
3	101	1	100	09/25/2022 12:17:00	09/25/2022 12:28:00
5	101	1	100	09/25/2022 12:27:00	09/25/2022 13:17:00

- Transaction ID 1 was the first of the series of payments, therefore it doesn't have a `previous_transaction` value from earlier transactions.
- Transaction ID 2 has a previous_transaction of 09/25/2022 12:00:00 which is pulled from the transaction in transaction ID 1.

Can you follow the pattern of the records in the `previous_transaction`?

Step 2

Next, we should evaluate the difference in time between two consecutive identical transactions. We can simply subtract the `previous_transaction` values from the `transaction_timestamp` values as we now have the previous transaction time for each completed payment.

The following statement can be easily incorporated into the `SELECT` clause of the previous query.

```
transaction_timestamp - previous_transaction -- (Obtained from the LAG function)
AS time_difference
```

transaction_id	merchant_id	credit_card_id	amount	transaction_timestamp	previous_transaction	time_difference
1	101	1	100	09/25/2022 12:00:00		
2	101	1	100	09/25/2022 12:08:00	09/25/2022 12:00:00	"minutes":8
3	101	1	100	09/25/2022 12:28:00	09/25/2022 12:08:00	"minutes":20
5	101	1	100	09/25/2022 13:37:00	09/25/2022 12:28:00	"hours":1,"minutes":9

The `time_difference` field makes it clear that there is an 8-minute lag between the first and second transactions. We will now convert this field into minutes format so that we can easily filter them.

One of the best methods to achieve that is to use the `EXTRACT` function.

```
SELECT
  merchant_id,
  credit_card_id,
  amount,
  transaction_timestamp,
  EXTRACT(EPOCH FROM transaction_timestamp -
    LAG(transaction_timestamp) OVER(
      PARTITION BY merchant_id, credit_card_id, amount
      ORDER BY transaction_timestamp)
  )/60 AS minute_difference
FROM transactions;
```

- `EPOCH` calculates the total number of seconds in a given interval.
- To calculate the difference in minutes, we divide these seconds by 60 (1 minute = 60 seconds). For example, the time interval for transaction id 5 is 1 hour and 9 minutes. `EPOCH` calculates its value as 4140 seconds. By dividing it by 60, we arrive at 69 minutes.

This is how the outcome looks:

merchant_id	credit_card_id	amount	transaction_timestamp	minute_difference
101	1	100	09/25/2022 12:00:00	
101	1	100	09/25/2022 12:08:00	8
101	1	100	09/25/2022 12:28:00	20

101	1	100	09/25/2022 13:37:00	69

Step 3

The last thing we need to do is to gather all the identical transactions which occurred within a **10-minute window**.

To do that, we must first convert the query into a common table expression (CTE). Then, we will filter the records using the `WHERE` clause for transactions in a 10-minute or lesser window.

Additionally, the `COUNT` function allows us to count the filtered records.

```
WITH payments AS (
  -- enter the previous query here
)

SELECT COUNT(merchant_id) AS payment_count
FROM payments
WHERE minute_difference <= 10;
```

Output for merchant ID 1:

payment_count
1

Now, we will eliminate the unnecessary columns from the `SELECT` clause since they were only there for exploration purposes.

That's it! We have our final query :)

Solution:

```
WITH payments AS (
  SELECT
    merchant_id,
    EXTRACT(EPOCH FROM transaction_timestamp -
      LAG(transaction_timestamp) OVER(
        PARTITION BY merchant_id, credit_card_id, amount
        ORDER BY transaction_timestamp)
    )/60 AS minute_difference
  FROM transactions)

SELECT COUNT(merchant_id) AS payment_count
FROM payments
WHERE minute_difference <= 10;
```