Q1:

There are two ways to go about it. Either `LEFT JOIN` or `RIGHT JOIN` can be established between tables `pages` and `page_likes` or a subquery can be used to identify which pages have not been liked by any user.

The `LEFT JOIN` clause starts selecting data from the left table. For each row in the left table (`pages`), it compares the value in the `page_id` column with the value of each row in the `page_id` column in the right table (`page_likes`).

When page_id are found on both sides, the `LEFT JOIN` clause creates a new row that contains columns that appear in the `SELECT` clause and adds this row to the result set.

In case `page_id` from`pages` table is not available in `page_likes` table, the `LEFT JOIN` clause also creates a new row that contains columns that appear in the `SELECT` clause. In addition, it fills the columns that come from the `page_likes` (right table) with NULL. Rows having NULL values in the result is the set of the solution.

Read about `LEFT JOIN` [1] and `RIGHT JOIN` [2] to get the better understanding.

Solution #1: Using LEFT OUTER JOIN

```
SELECT pages.page_id
FROM pages
LEFT OUTER JOIN page_likes AS likes
  ON pages.page_id = likes.page_id
WHERE likes.page_id IS NULL;
```

Another solution to this problem, since pages with NO LIKES are needed, would be the `NOT EXISTS` clause (refer to Solution #2). It's an appropriate and efficient operator to get this information. Check out here.

Both methods give the same output.

Solution #2: Using `EXCEPT`

```
SELECT page_id
FROM pages
EXCEPT
SELECT page_id
FROM page_likes;
```

Solution #3: Using `NOT IN`

```
SELECT page_id
FROM pages
WHERE page_id NOT IN (
  SELECT DISTINCT page_id
  FROM page_likes
);
```

Solution #4: Using `NOT EXISTS`

```
SELECT page_id
FROM pages
WHERE NOT EXISTS (
  SELECT page_id
  FROM page_likes AS likes
  WHERE likes.page_id = pages.page_id
;)
```

Q2:

Great news! The parts table already includes all parts currently in production, so no additional filtering is necessary to exclude non-production parts.
To extract unfinished parts, we can simply filter for rows where the `finish_date` column contains no data, indicated by a NULL value.

```
SELECT part, assembly_step
FROM parts_assembly
WHERE finish_date IS NULL;
```

Q3:

First, we need to find the number of tweets posted by each user in 2022 by grouping the tweet records by user ID and counting the tweets.

```
SELECT
  user_id,
  COUNT(tweet_id) AS tweet_count_per_user
FROM tweets
WHERE tweet_date BETWEEN '2022-01-01'
  AND '2022-12-31'
GROUP BY user_id;
```

The output shows the number of tweets posted by each user in 2022:

user_id	tweet_count_per_user
111	2
148	1
254	1

Based on the output, we can infer that in the year 2022, user 111 has posted two tweets, while users 148 and 254 have only posted one tweet each.

Next, we use the query above as a subquery, then we use the `tweet_count_per_user` field as the tweet bucket and retrieve the number of users.

```
SELECT
  tweet_count_per_user AS tweet_bucket,
  COUNT(user_id) AS users_num
FROM (
  SELECT
    user_id,
    COUNT(tweet_id) AS tweet_count_per_user
  FROM tweets
  WHERE tweet_date BETWEEN '2022-01-01'
    AND '2022-12-31'
  GROUP BY user_id) AS total_tweets
GROUP BY tweet_count_per_user;
```

This query generates a histogram of the number of tweets per user in 2022. The output shows the tweet count per user as the tweet bucket and the number of Twitter users who fall into that bucket.

tweet_bucket	users_num
1	2
2	1

Alternatively, we can use a Common Table Expression (CTE) instead of a subquery to compute the tweet counts.

A CTE is a data set that is created temporarily and can be used within a query. It is available for use during the entire session of the query execution. On the other hand, a subquery is a query nested within another query and can only be used within that query. A subquery typically acts as a column with a single value in the FROM or WHERE clause.

The benefits of using a CTE are that it is more readable and can be reused throughout the query session, whereas a subquery can only be used within the query in which it is defined.

Solution #2: Using CTE

```
WITH total_tweets AS (
  SELECT
    user_id,
    COUNT(tweet_id) AS tweet_count_per_user
  FROM tweets
  WHERE tweet_date BETWEEN '2022-01-01'
    AND '2022-12-31'
  GROUP BY user_id)

SELECT
  tweet_count_per_user AS tweet_bucket,
  COUNT(user_id) AS users_num
FROM total_tweets
GROUP BY tweet_count_per_user;
```

Q4:
To calculate the viewership on different devices (laptops vs. mobile devices), we can utilise the aggregate function `COUNT()` along with the `FILTER` clause to apply conditional expressions.

```
SELECT
  COUNT(*) FILTER (WHERE conditional_expression)
FROM table_name;
```

In the given example, the device types 'tablet' and 'phone' are considered as 'mobile' devices, while 'laptop' is treated as a separate device type.

The following query can be used to obtain the desired result:

```
SELECT
  COUNT(*) FILTER (WHERE device_type = 'laptop') AS laptop_views,
  COUNT(*) FILTER (WHERE device_type IN ('tablet', 'phone'))  AS
mobile_views
FROM viewership;
```

In the first column `laptop_views`, `COUNT(*) FILTER (WHERE device_type = 'laptop')` calculates the count of rows where the device type is labeled as 'laptop'.

In the second column `mobile_views`, `COUNT(*) FILTER (WHERE device_type IN ('tablet', 'phone'))` counts the number of rows where the device type is a tablet or a phone.

The result would have two columns, `laptop_views` and `mobile_views` displaying the respective counts of views for each device type.

laptop_views	mobile_views
2	3

Solution #2: Using SUM() & CASE statement

```
SELECT
  SUM(CASE WHEN device_type = 'laptop' THEN 1 ELSE 0 END) AS
laptop_views,
  SUM(CASE WHEN device_type IN ('tablet', 'phone') THEN 1 ELSE 0 END)
AS mobile_views
FROM viewership;
```

Q5:
Candidates with a variety of skillsets have applied for this role, but we need candidates who know Python, Tableau, and PostgreSQL.

We'll start by using the `IN` operator to find candidates which have some of the required skills:

```
SELECT candidate_id
FROM candidates
WHERE skill IN ('Python', 'Tableau', 'PostgreSQL');
```

The output should look something like this: (Showing random 5 records)

candidate_id	skill
123	Python
123	Tableau
123	PostgreSQL
345	Python
345	Tableau

We can see from the output that these candidates possess at least one of the necessary skills, but keep in mind, the problem is asking for candidates who have ALL THREE of these skills, so we aren't done quite yet!

It's important to keep in mind that the `candidates` table does not contain any duplicates, so each combination of candidate and skill is a unique row. Therefore, a candidate should have exactly 3 rows for each of the necessary skills in order to be qualified for the job.

Now, we group the candidates table by candidate ID using the `GROUP BY` clause and count the number of skills for each group using the `COUNT` function.

Let's look at the total number of required skills for each candidate:

```
SELECT
  candidate_id,
  COUNT(skill) AS skill_count
FROM candidates
WHERE skill IN ('Python', 'Tableau', 'PostgreSQL')
GROUP BY candidate_id;
```

Output:

candidate_id	skill_count
123	3

345	2

Candidate 123 possesses all three of the required skills in this instance, but Candidate 345 possesses only two of the required skills.

In the last step, we'll use `HAVING` to select only candidates with three skills and `ORDER BY` the candidate ID, as per the task.

Note that the full solution below counts skills inside the `HAVING`, not in the `SELECT` as shown above.

Full Solution:

```
SELECT candidate_id
FROM candidates
WHERE skill IN ('Python', 'Tableau', 'PostgreSQL')
GROUP BY candidate_id
HAVING COUNT(skill) = 3
ORDER BY candidate_id;
```

Q6:

First, we can use `MIN` and `MAX` clauses on the `post_date` column to retrieve the dates for the first and the last post, and then substract one from another accordingly.

As we are asked to find the difference on a user basis for the year 2021, it is important to `GROUP` the results by user_id, and then filter for the year 2021. To do so, we can use `date_part` function, which - as the name suggests - retrieves a part from input date. Thus, in our scenario it is the post_date variable.

Lastly, to filter out the users who have only posted once during the year, we can use `HAVING` clause with the `COUNT` of posts over 1

```
SELECT
      user_id,
    MAX(post_date::DATE) - MIN(post_date::DATE) AS days_between
FROM posts
WHERE DATE_PART('year', post_date::DATE) = 2021
GROUP BY user_id
HAVING COUNT(post_id)>1;
```

Q7:

To find the top 2 Power Users who sent the most messages on Microsoft Teams in August 2022, we need to first determine the count of messages sent by each user, which we'll refer to as "senders".

We start by extracting the month and year from the `sent_date` field and filtering the results to only include messages sent in August 2022. We then use the GROUP BY clause to group the messages by `sender_id` and calculate the count of messages using the `COUNT()` function:

```
SELECT
  sender_id,
  COUNT(message_id) AS count_messages
FROM messages
WHERE EXTRACT(MONTH FROM sent_date) = '8'
  AND EXTRACT(YEAR FROM sent_date) = '2022'
GROUP BY sender_id;
```

Here's the output from the query:

sender_id	count_messages
2520	3
3601	4
4500	1

The output of this query will provide the count of messages for each sender as shown in the example table.

Since we assume that no two users have sent the same number of messages in August 2022, we can simply use an ORDER BY clause in descending order to sort the results based on the count of messages.

Finally, we use a LIMIT clause to restrict the results to only the top 2 senders, giving us the desired outcome.

```
SELECT
  sender_id,
  COUNT(message_id) AS count_messages
FROM messages
WHERE EXTRACT(MONTH FROM sent_date) = '8'
  AND EXTRACT(YEAR FROM sent_date) = '2022'
GROUP BY sender_id
ORDER BY count_messages DESC
LIMIT 2;
```

Q8:

The first step is to find all the companies with job listings that has the same title and description. We can do that by `COUNT`ing the number of `job_id`s grouped by `company_id`, `title` and `description`.

```
SELECT
  company_id,
  title,
  description,
  COUNT(job_id) AS job_count
FROM job_listings
GROUP BY
  company_id,
  title,
  description;
```

Output (showing first 5 rows with total of 7 rows):

company_id	title	description	job_count
827	Data Scientist	Data scientist uses data to understand and explain the phenomena around them, and help organizations make better decisions.	2
244	Data Engineer	Data engineer works in a variety of settings to build systems that collect, manage, and convert raw data into usable information for data scientists and business analysts to interpret.	1
845	Business Analyst	Business analyst evaluates past and current business data with the primary goal of improving decision-making processes within organizations.	1
244	Software Engineer	Software engineers design and create computer systems and applications to solve real-world problems.	2
345	Data Analyst	Data analyst reviews data to identify key insights into a business's customers and ways the data can be used to solve problems.	2

Next, we convert the previous query into a CTE and filter for when `job_count` is more than 1 meaning we only want where there are 2 or more duplicate job listings. Then, we apply a `DISTINCT` on `company_id` to get the unique `company_id` and count them.

```
WITH jobs_grouped AS (
-- Insert above query here
)

SELECT COUNT(DISTINCT company_id) AS co_w_duplicate_jobs
FROM jobs_grouped
WHERE job_count > 1;
```

Results:

co_w_duplicate_jobs
3

Solution #1: Using CTE

```
WITH jobs_grouped AS (
  SELECT
    company_id,
    title,
    description,
    COUNT(job_id) AS job_count
  FROM job_listings
  GROUP BY
    company_id,
    title,
    description)

SELECT COUNT(DISTINCT company_id) AS co_w_duplicate_jobs
FROM jobs_grouped
WHERE job_count > 1;
```

Solution #2: Using Subquery

```
SELECT COUNT(DISTINCT company_id) AS co_w_duplicate_jobs
FROM (
  SELECT
    company_id,
    title,
    description,
    COUNT(job_id) AS job_count
  FROM job_listings
  GROUP BY
    company_id,
    title,
    description) AS jobs_grouped
WHERE job_count > 1;
```

Q9:

We begin by joining the `trades` and `users` tables based on the related column `user_id`. This is because the 'Completed' order `status` is stored in the `trades` table, while the cities are stored in the `users` table.

In the SELECT statement, we pull the `city` field from the `users` table and the `order_id` field from the `trades` table.

```
SELECT users.city, trades.order_id
FROM trades
INNER JOIN users
  ON trades.user_id = users.user_id;
```

Output (showing the first 5 rows only):

city	order_id
San Francisco	100777
San Francisco	100102
San Francisco	100101
Boston	100259
Boston	100264

Next, we filter the 'Completed' orders and retrieve the number of orders for each city using the `COUNT()` function. We group the results by the `city` column using the `GROUP BY` statement.

```
SELECT
  users.city,
  COUNT(trades.order_id) AS total_orders
FROM trades
INNER JOIN users
  ON trades.user_id = users.user_id
WHERE trades.status = 'Completed'
GROUP BY users.city;
```

The GROUP BY statement is commonly employed in conjunction with aggregate functions such as `COUNT`, `MAX`, `MIN`, `SUM`, and `AVG` to group the results based on non-aggregate columns.

Did you notice that our output is grouped by the `city` column?

city	total_orders
Boston	1
New York	2

San Francisco	4

Finally, to arrange the output in descending order based on the highest number of completed orders, we utilize the `ORDER BY` clause and limit the results to the top 3 orders using the `LIMIT` clause.

```
SELECT
  users.city,
  COUNT(trades.order_id) AS total_orders
FROM trades
INNER JOIN users
  ON trades.user_id = users.user_id
WHERE trades.status = 'Completed'
GROUP BY users.city
ORDER BY total_orders DESC
LIMIT 3;
```

city	total_orders
San Francisco	4
Boston	3
Denver	2

Based on the results, San Francisco has the highest number of completed orders with 4 orders. Boston has the second-highest number of completed orders with 3 orders and Denver has the third-highest number of completed orders with 2 orders.

Q10:

As observed, the `reviews` table does not have a separate column for month. Therefore, we need to extract the month from the `submit_date` column using the `EXTRACT(MONTH FROM column_name)` function, which returns the month in numerical format.

To calculate the average star ratings per month for each product, we can use the `AVG()` aggregate function to calculate the mean of the stars column and the `ROUND()` function to round the result to two decimal places for accuracy.

The query would be as follows:

```
SELECT
  EXTRACT(MONTH FROM submit_date) AS mth,
  product_id,
  ROUND(AVG(stars), 2) AS avg_stars
FROM reviews
GROUP BY
  EXTRACT(MONTH FROM submit_date),
  product_id
ORDER BY mth, product_id;
```

In SQL, the order of execution is important to understand. In the given solution's query, the sequence of execution is as follows:

1. FROM clause: The query fetches data from the `reviews` table.
2. GROUP BY clause: SQL performs grouping based on the `EXTRACT(MONTH FROM submit_date)` and `product_id` columns.
3. SELECT clause: The query selects the `EXTRACT(MONTH FROM submit_date)` column and aliases it as `mth`, along with the `product_id` and the average of stars rounded to two decimal places as `avg_stars`.
4. ORDER BY clause: The query sorts the results based on the `mth` column, which is the alias used in the SELECT clause, followed by the `product_id` column.

It's important to note that the GROUP BY clause is executed before the SELECT statement. Therefore, we cannot use the mth alias in the GROUP BY clause, as the mth column is created after the SELECT statement is executed. However, we can use the mth alias in the ORDER BY clause, as it is executed after the SELECT statement, and the mth column has been created by then.

Understanding the order of SQL execution is crucial, as it is a common topic in technical interviews. It's recommended to familiarize yourself with the sequence of execution in SQL for better query writing and debugging.

Q11:

Step 1: Filter for analytics events from year 2022

First, we filter for analytics events from the year 2022 using the `WHERE` clause with appropriate comparison operators:

- `timestamp >= '2022-01-01'`: Events with timestamps on or after January 1, 2022, are selected.
- `timestamp < '2023-01-01'`: Events before January 1, 2023, are selected, but events on January 1, 2023, are excluded from the result.

```
SELECT *
FROM events
WHERE timestamp >= '2022-01-01'
  AND timestamp < '2023-01-01';
```

Step 2: Calculate the number of clicks and number of impressions

Next, find the number of clicks and impressions using the `CASE` statement to assign a value of 1 for 'click' events and 0 for other events:

```
SELECT
  app_id,
  CASE WHEN event_type = 'click' THEN 1 ELSE 0 END AS clicks,
  CASE WHEN event_type = 'impression' THEN 1 ELSE 0 END AS impressions
FROM events
WHERE timestamp >= '2022-01-01'
  AND timestamp < '2023-01-01';
```

Here's the first 5 rows of output:

app_id	clicks	impressions
123	0	1
123	0	1
123	1	0
234	0	1
234	1	0

Then, we add up the clicks and impressions by wrapping the `CASE` statements with a `SUM()` aggregate function and group the results by `app_id`.

```
SELECT
  app_id,
  SUM(CASE WHEN event_type = 'click' THEN 1 ELSE 0 END) AS clicks,
  SUM(CASE WHEN event_type = 'impression' THEN 1 ELSE 0 END) AS
impressions
FROM events
WHERE timestamp >= '2022-01-01'
  AND timestamp < '2023-01-01'
GROUP BY app_id;
```

app_id	clicks	impressions
123	2	3
234	1	3

Step 4: Calculate the percentage of the click-through rate and round to 2 decimal places

Finally, calculate the percentage of click-through rate (CTR) by dividing the number of clicks by the number of impressions and multiplying by 100.0, rounded to 2 decimal places using the `ROUND()` function.

Percentage of click-through rate = 100.0 * Number of clicks / Number of impressions

```
SELECT
```

```
  app_id,
  ROUND(100.0 *
    SUM(CASE WHEN event_type = 'click' THEN 1 ELSE 0 END) /
    SUM(CASE WHEN event_type = 'impression' THEN 1 ELSE 0 END), 2)  AS
ctr_rate
FROM events
WHERE timestamp >= '2022-01-01'
  AND timestamp < '2023-01-01'
GROUP BY app_id;
```

Solution #2: Using COUNT(CASE ...)

```
SELECT
  app_id,
  ROUND(100.0 *
    COUNT(CASE WHEN event_type = 'click' THEN 1 ELSE NULL END) /
    COUNT(CASE WHEN event_type = 'impression' THEN 1 ELSE NULL END), 2)
AS ctr_rate
FROM events
WHERE timestamp >= '2022-01-01'
  AND timestamp < '2023-01-01'
GROUP BY app_id;
```

Solution #3: Using SUM() *FILTER ()*

```
SELECT
  app_id,
  ROUND(100.0 *
    SUM(1) FILTER (WHERE event_type = 'click') /
    SUM(1) FILTER (WHERE event_type = 'impression'), 2) AS ctr_app
FROM events
WHERE timestamp >= '2022-01-01'
  AND timestamp < '2023-01-01'
GROUP BY app_id;
```

Q12:

1. Users who confirmed on the second day.
2. The texts received must say 'Confirmed'.

To begin, we join the `emails` and `texts` tables on the matching `user_id` field. Feel free to skip this step if you wish as our intention is to clarify the definition of condition no. 1 for you.

```
SELECT *
FROM emails
INNER JOIN texts
  ON emails.email_id = texts.email_id;
```

Output with selected rows:

email_id	user_id	signup_date	text_id	email_id	signup_action	action_date
433	1052	07/09/2022 00:00:00	6997	433	Not confirmed	07/09/2022 00:00:00
433	1052	07/09/2022 00:00:00	7000	433	Confirmed	07/10/2022 00:00:00
236	6950	07/01/2022 00:00:00	9841	236	Confirmed	07/01/2022 00:00:00
450	8963	08/02/2022 00:00:00	6800	450	Not confirmed	08/03/2022 00:00:00
555	8963	08/09/2022 00:00:00	1255	555	Not confirmed	08/09/2022 00:00:00
555	8963	08/09/2022 00:00:00	2660	555	Not confirmed	08/10/2022 00:00:00
555	8963	08/09/2022 00:00:00	2800	555	Confirmed	08/11/2022 00:00:00

Next, we interpret the output together:

- Rows 1-2: User 1052 signed up on 07/09/2022 and confirmed their account on the next day, 07/10/2022. This satisfies both conditions.
- Row 3: User 6950 signed up and confirmed their account on the same day, 07/01/2022, so this user fails both conditions.
- Rows 4-7: User 8963 signed up twice, once on 08/02/2022 and another time on 08/09/2022, and only confirmed their account on 08/11/2022, which is 3 days after their signup. So, the first condition is not fulfilled.

Now that you understand how to fulfill these conditions, let's incorporate them into the solution.

Condition #1: Users who confirmed on the second day

```
SELECT *
FROM emails
INNER JOIN texts
  ON emails.email_id = texts.email_id
WHERE texts.action_date = emails.signup_date + INTERVAL '1 day'
```

The condition `texts.action_date = emails.signup_date + INTERVAL '1 day'` in the WHERE clause means we only want users who confirmed on the second day after their signup, as reflected in the `texts.action_date` field. We achieve this by taking `emails.signup_date` and adding an interval of 1 day.

email_id	user_id	signup_date	text_id	signup_action	action_date
433	1052	07/09/2022 00:00:00	7000	Confirmed	07/10/2022 00:00:00
450	8963	08/02/2022 00:00:00	6800	Not confirmed	08/03/2022 00:00:00
555	8963	08/09/2022 00:00:00	2660	Not confirmed	08/10/2022 00:00:00
741	1235	07/25/2022 00:00:00	1568	Confirmed	07/26/2022 00:00:00

As you can see, the `action_date` is 1 day after the `signup_date`, fulfilling the first condition. Now let's move on to the second condition.

Condition #2: The texts received must say 'Confirmed'

```
SELECT *
FROM emails
INNER JOIN texts
  ON emails.email_id = texts.email_id
WHERE texts.action_date = emails.signup_date + INTERVAL '1 day'
  AND texts.signup_action = 'Confirmed';
```

In addition to the first condition, we add the condition `texts.signup_action = 'Confirmed'` in the WHERE clause to ensure that the texts received must say 'Confirmed'.

email_id	user_id	signup_date	text_id	signup_action	action_date
433	1052	07/09/2022 00:00:00	7000	Confirmed	07/10/2022 00:00:00
741	1235	07/25/2022 00:00:00	1568	Confirmed	07/26/2022 00:00:00

Finally, we retrieve the unique user IDs only.

```
SELECT DISTINCT user_id
FROM emails
INNER JOIN texts
  ON emails.email_id = texts.email_id
WHERE texts.action_date = emails.signup_date + INTERVAL '1 day'
  AND texts.signup_action = 'Confirmed';
```

Q13:

To find the difference between the best and worst performing months in card issuance, you can use the `MAX()` and `MIN()` functions.

Apply the functions on the `issued_amount` column, and simply calculate the difference between the two. As we are asked for the difference between both cards, it is important to group the results by the card name.

Don't forget to order the dataset according to the biggest difference!

```
SELECT
  card_name,
  MAX(issued_amount) - MIN(issued_amount) AS difference
FROM monthly_cards_issued
GROUP BY card_name
ORDER BY difference DESC;
```

Q14:

Step 1: Calculate the weighted average of items per order

To calculate the weighted average of items per order, we multiply each `item_count` with the corresponding number of occurrences `order_occurrences`, calculate the sum using `SUM(item_count * order_occurrences)`, and finally divide it by the total number of orders using `SUM(order_occurrences)`.

```
SELECT
  SUM(item_count*order_occurrences)
    /SUM(order_occurrences) AS mean
FROM items_per_order;
```

However, it's important to note that both `item_count` and `order_occurrences` are of integer type by default, which means that division will return an integer result. To ensure that the output is rounded to 1 decimal place, we can cast either column to a decimal type using `::DECIMAL` or `CAST(field AS decimal)`.

```
SELECT
  SUM(item_count::DECIMAL*order_occurrences)
    /SUM(order_occurrences) AS mean
FROM items_per_order;
```

Step 2: Round results to 1 decimal place

To round the result to 1 decimal place, we can use the `ROUND(__,1)` function.

```
SELECT
  ROUND(
    SUM(item_count::DECIMAL*order_occurrences)
    /SUM(order_occurrences)
  ,1) AS mean
FROM items_per_order;
```

Q15:
First, we must establish the formula used to calculate the profits.

Total Profit = Total Sales - Cost of Goods Sold

The profit is calculated by subtracting the cost of goods sold (being the direct cost associated with producing the drug) (`cogs`) from the total sales generated (`total_sales`).

```
SELECT
  drug,
  total_sales, -- Field is not required in the final query
  cogs, -- Field is not required in the final query
  total_sales - cogs AS total_profit
FROM pharmacy_sales;
```

Displaying the result for 4 random drugs.

drug	total_sales	cogs	total_profit
Zyprexa	293452.54	208876.01	84576.53
Surmontil	600997.19	521182.16	79815.03
Varicose Relief	500101.61	419174.97	80926.64
Burkhart	1084258	1006447.73	77810.27

Profit of $84,576.53 has been made from the sale of Zyprexa.

Let's arrange the results in the decreasing order of the total profits generated by the sale of the drugs. ORDER BY clause with DESC will be added to the query for this step.

```
SELECT
  drug,
  total_sales, -- Field is not required in the final query
  cogs, -- Field is not required in the final query
  total_sales - cogs AS total_profit
FROM pharmacy_sales
ORDER BY total_profit DESC;
```

drug	total_sales	cogs	total_profit
Zyprexa	293452.54	208876.01	84576.53
Varicose Relief	500101.61	419174.97	80926.64
Surmontil	600997.19	521182.16	79815.03
Burkhart	1084258	1006447.73	77810.27

The final step is to only keep the rows of drugs with the highest 3 profits. The LIMIT clause keeps the specified number of rows and discards the rest of the table.

Solution:

```
SELECT
  drug,
  total_sales - cogs AS total_profit
FROM pharmacy_sales
ORDER BY total_profit DESC
LIMIT 3;
```

Q16:

Step 1: Calculate total profit or loss for each manufacturer

To determine the total profit or loss for each manufacturer, we can use the formula:

Total Profit/(Total Loss) = Total Sales - Total Cost of Goods Sold

where a positive value indicates profit and a negative value indicates a loss. The query would look like this:

```sql
SELECT
  manufacturer,
  drug,
  total_sales - cogs AS net_value
FROM pharmacy_sales;
```

Showing the output for 4 randomly selected drugs:

manufacturer	drug	net_value
Biogen	Acyclovir	-297324.73
AbbVie	Lamivudine and Zidovudine	-221429.36
Eli Lilly	Dermasorb TA Complete Kit	-221422.17
Biogen	Medi-Chord	672765.95

This query will provide a result with the `net_value` column showing the calculated profit or loss for each drug.

Step 2: Filter for drugs making losses

To filter for drugs that are making losses, we can add a WHERE clause to keep rows where the `total_sales - cogs` is equal to or less than 0, indicating a loss:

```sql
SELECT
  manufacturer,
  drug,
  total_sales - cogs AS net_value
FROM pharmacy_sales
WHERE total_sales - cogs <= 0;
```

This query will return only the rows where the drug is making a loss.

Step 3: Obtain count of unprofitable drugs and total losses for each manufacturer

Next, we can use aggregate functions to obtain the count of drugs associated with each manufacturer using `COUNT()` and the total losses suffered by each manufacturer using `SUM()`:

```sql
SELECT
  manufacturer,
  COUNT(drug) AS drug_count,
  SUM(total_sales - cogs) AS total_loss
FROM pharmacy_sales
WHERE total_sales - cogs <= 0
GROUP BY manufacturer;
```

Step 4: Convert total loss to absolute value and sort output

To convert the total losses to absolute value (i.e., remove the negative sign), we can use the `ABS()` function on the `SUM(net_value)` and order the results with the highest losses at the top:

```
SELECT
  manufacturer,
  COUNT(drug) AS drug_count,
  ABS(SUM(total_sales - cogs)) AS total_loss
FROM pharmacy_sales
WHERE total_sales - cogs <= 0
GROUP BY manufacturer
ORDER BY total_loss DESC;
```

Solution #2: Without ABS()

Alternatively, we can achieve the same result without using the `ABS()` function by switching the cogs and total_sales positions in the `SUM()` function and filtering for rows where `cogs > total_sales` in the WHERE clause:

```
SELECT
  manufacturer,
  COUNT(drug) AS drug_count,
  SUM(cogs - total_sales) AS total_loss
FROM pharmacy_sales
WHERE cogs > total_sales
GROUP BY manufacturer
ORDER BY total_loss DESC;
```

Q17:
Goal: Find the total drug sales in million for each manufacturer.

1. Find the total sales by manufacturer.
2. Convert the total sales to million-dollar format and round to the closest million.
3. Transform total sales to '$xx million' format.
4. Order the results by the highest total sales.

Step 1: Find the total sales by manufacturer

First, we calculate the sum of total sales using the aggregate function `SUM()` and segregate the results by the manufacturer in the `GROUP BY` clause.

```
SELECT
  manufacturer,
  SUM(total_sales) as sales
FROM pharmacy_sales
GROUP BY manufacturer;
```

Output showing the 2 randomly selected records:

manufacturer	sales
Eli Lilly	81641381.27
Biogen	69824472.58

The output above shows that Eli Lilly and Biogen each sold drugs with a total sales value of $81,641,381.27 and $69,824,472.58, respectively.

Although each manufacturer's sales have been calculated, the figures are not in the million-dollar format.

Step 2: Convert total sales to million-dollar format and round to the closest million

Next, we round up the `sales` to the closest million.

To do so, we must first divide the `sales` by one million `/1000000` and round them to the closest million using the ROUND function. If the decimal place is unspecified, its default value is 0.

```
SELECT
  manufacturer,
  ROUND(SUM(total_sales) / 1000000) AS sales_mil
FROM pharmacy_sales
GROUP BY manufacturer;
```

Showing the output for Eli Lilly and Biogen:

manufacturer	sales_mil
Eli Lilly	82
Biogen	70

Eli Lilly's sales of $81,641,381.27 is rounded to the closest million to $82 and Biogen's $69,824,472.58 is rounded to $70.

Step 3: Transform total sales to '$xx million' format

The sales data will be fed into a dashboard, thus it has to be formatted like this: "$xx million".

Using the CONCAT function, we will concatenate the 3 elements: `$` symbol + `sales_mil` in million + `million` string. Remember to keep a space in front of `million`.

P.S. It is not necessary to convert `sales` into `VARCHAR` data type as the `CONCAT()` function accepts both `VARCHAR` and `INT` data types. Bear in mind that the `sales_mil` column is now a `VARCHAR` data type.

```
SELECT
  manufacturer,
  CONCAT('$', ROUND(SUM(total_sales) / 1000000), ' million') AS
sales_mil
FROM pharmacy_sales
GROUP BY manufacturer;
```

Output:

manufacturer	sales_mil
Eli Lilly	$82 million
Biogen	$70 million

Step 4: Order the results by the highest total sales

Finally, sort the results in the descending order of sales.

But hold on — we can't just apply the `ORDER BY` clause to the new `sales_mil` column because this column is a `VARCHAR` data type.

Hence, we will utilize the `ORDER BY` clause on `SUM(total_sales)` to place the highest total sales at the top followed by the least total sales.

Solution 1

```
SELECT
  manufacturer,
  CONCAT( '$', ROUND(SUM(total_sales) / 1000000), ' million') AS
sales_mil
FROM pharmacy_sales
GROUP BY manufacturer
ORDER BY SUM(total_sales) DESC;
```

Solution 2: Using CTE

```
WITH drug_sales AS (
  SELECT
    manufacturer,
    SUM(total_sales) as sales
  FROM pharmacy_sales
  GROUP BY manufacturer
)
```

```
SELECT
  manufacturer,
  ('$' || ROUND(sales / 1000000) || ' million') AS sales_mil
FROM drug_sales
ORDER BY sales DESC;
```

Q18:

First, we identify who called and how frequently.

GROUP BY clause can be used to generate the groups. Since we need the information for members, we group them based on the `policy_holder_id` column.

Note that members are used interchangeably with policy holders but they mean the same.

Next, we apply an aggregate function COUNT(), which counts the number of values in the column `case_id` for each policyholder-group.

```
SELECT
  policy_holder_id,
  COUNT(case_id) AS call_count
FROM callers
GROUP BY policy_holder_id;
```

Displaying records for policy holder IDs 53578035 and 54126242:

policy_holder_id	call_count
53578035	1
54126242	5

In contrast to member 54126242, who has reportedly made five calls, member 51983251 has only made one call.

Then, a conditional clause with the keyword HAVING can be applied to keep rows with members who called 3 or more times. `HAVING` clause is used to filter group rows. This sets it apart from a WHERE clause that filters individual rows.

```
SELECT
  policy_holder_id,
  COUNT(case_id) AS call_count
FROM callers
GROUP BY policy_holder_id
HAVING COUNT(case_id) >= 3;
```

policy_holder_id	call_count
54126242	5

Only member 54126242 is in the result because this member made five calls.

Finally, we obtain the count of members using another `COUNT()` function. Before the `COUNT()` function can be used, the previous query must first be encapsulated in a subquery.

A subquery is a nested query. It’s a query within a query and can be used within that query only.

```
SELECT COUNT(policy_holder_id) AS member_count

FROM (
  SELECT
    policy_holder_id,
    COUNT(case_id) AS call_count
  FROM callers
  GROUP BY policy_holder_id
  HAVING COUNT(case_id) >= 3
) AS call_records;
```

Output based on the table above:

member_count
1

Solution #2: Using CTE

```
WITH call_records AS (
SELECT
  policy_holder_id,
  COUNT(case_id) AS call_count
FROM callers
GROUP BY policy_holder_id
HAVING COUNT(case_id) >= 3
)

SELECT COUNT(policy_holder_id) AS member_count
FROM call_records;
```

Q19:
We'll start by defining the formula.

Percentage of uncategorized calls = (Number of uncategorized calls / Total calls) x 100

Let's break this problem into 4 steps:

1. Filter for uncategorised calls and count them.
2. Count the total calls.
3. Use the percentage formula.
4. Round the output.

Step 1: Filter for uncategorised calls and count them.

First, count the calls that are uncategorised i.e. call records with the `call_category` column having either "n/a" or `NULL` values which looks like an empty space.

Using the `COUNT()` function, we can get the count of uncategorised calls.

```
SELECT COUNT(case_id) AS uncategorised_calls
FROM callers
WHERE call_category IS NULL
  OR call_category = 'n/a';
```

uncategorised_calls
225

225 calls were recorded without being assigned to a category.

Instead of putting this query into a subquery or CTE which can make the solution a bit lengthy, we're using the `FILTER ()` clause with the combination of the `COUNT()` function. Let's use the `FILTER()` clause in our query above.

```
SELECT
  COUNT (case_id) FILTER (
    WHERE call_category IS NULL OR call_category = 'n/a') AS
uncategorised_calls
FROM callers;
```

Have a run in the editor - it produces the same result!

Step 2: Count the total calls

In the following step, we will utilise another `COUNT()` function to get the number of total calls regardless of the category.

```
SELECT
  COUNT (case_id) FILTER (
    WHERE call_category IS NULL OR call_category = 'n/a') AS
uncategorised_calls,
  COUNT(case_id) AS total_calls
FROM callers;
```

uncategorised_calls	total_calls
225	500

Step 3: Use the percentage formula

Let's now modify our query to fit into the percentage formula.

```
SELECT
  100.0 * COUNT (case_id) FILTER (
    WHERE call_category IS NULL OR call_category = 'n/a')
    / COUNT (case_id) AS uncategorised_call_pct
FROM callers;
```

uncategorised_call_pct
45.0000000000000000

Note: It is crucial to multiply by 100.0 instead of 100 since division operations require at least one numeric value to be of the `DECIMAL` data type. Otherwise, the digits after the decimal `.` will be truncated and the results will be incorrect.

Step 4: Round the output

The last step is to round the percentage to one decimal place. ROUND () function can be used to accomplish it.

This brings us to our final solution query. Yay!

Solution #1: Using FILTER clause

```
SELECT
  ROUND (100.0 *
    COUNT (case_id) FILTER (
      WHERE call_category IS NULL OR call_category = 'n/a')
    / COUNT (case_id), 1) AS uncategorised_call_pct
FROM callers;
```

There are numerous methods to solve this question. Below are 2 more suggested solutions for you to try out.

Solution #2: Using WHERE clause

```sql
SELECT
  ROUND(100.0 *
    COUNT(case_id)/
      (SELECT COUNT(*) FROM callers),1) AS uncategorised_call_pct
FROM callers
WHERE call_category IS NULL
  OR call_category = 'n/a';
```

Solution #3: Using CTE

```sql
WITH uncategorised_calls AS (
  SELECT COUNT(case_id) AS call_count
  FROM callers
  WHERE call_category IS NULL
    OR call_category = 'n/a'
)

SELECT
  ROUND(100.0 * call_count
    / (SELECT COUNT(*) FROM callers), 1) AS uncategorised_call_pct
FROM uncategorised_calls
GROUP BY call_count;
```