SQL Music Store Analysis

Data Insights and Query Explanations

Using PostgreSQL and pgAdmin 4

Project by Saswat Seth

Question Set 1

-> Who is the senior most employee based on job title?

Ans:

Solution 1:

```
SELECT
   employee_id,
   first_name,
   last_name
FROM
   employee
ORDER BY
   levels DESC
LIMIT 1;
```

Intuition for Solution 1:

- **Identify the Senior-Most Employee**: To determine the senior-most employee, we use the levels column, which reflects the job level of each employee. Higher levels indicate more senior positions.
- **Sorting**: By sorting the levels column in descending order (DESC), the employee with the highest job level will appear first in the result set.
- **Limiting the Results**: The LIMIT 1 ensures that only one record (the topmost, senior-most employee) is returned.

Explanation for Solution 1:

- **ORDER BY levels DESC**: This sorts the employees based on their job level in descending order, meaning the employee with the highest level (i.e., most senior) appears first.
- **LIMIT 1**: This restricts the result to the top-most record after sorting, so only the senior-most employee is returned.
- The query selects and returns the employee_id, first_name, and last_name of the most senior employee.

Solution 2:

```
SELECT
   employee_id,
   first_name
FROM
   employee
   WHERE
   employee_id = '9';
```

Intuition for Solution 2:

- **Direct Employee Lookup**: This query looks up the employee information directly by using their employee_id. The value '9' is assumed to be the specific employee ID in question.
- **Efficient Query**: The query uses a direct search by employee_id, which is usually the primary key, ensuring that the result is retrieved quickly.

Explanation for Solution 2:

- **WHERE employee_id = '9'**: This clause filters the result to the employee whose employee_id is '9'. Since employee_id is typically a primary key, this operation is very efficient.
- The query returns the employee_id and first_name of the employee with the specified employee_id.

-> Which countries have the most Invoices?

Ans:

```
SELECT
   billing_country AS Country,
   COUNT(invoice_id) AS Invoice_count
FROM
   invoice
GROUP BY
   Country
ORDER BY
   Invoice_count DESC
LIMIT 4;
```

Intuition:

- **Group by Country**: Since the query requires us to find out which countries have the most invoices, we need to group the invoices by the billing_country.
- **Count Invoices**: By counting the invoice_id for each country, we can determine how many invoices were generated per country.
- **Sort by Invoice Count**: To get the countries with the most invoices, we order the result set by Invoice_count in descending order.
- **Limit the Result**: The query returns only the top 4 countries with the highest number of invoices using LIMIT 4.

Explanation:

- **COUNT(invoice_id)**: This counts the number of invoices for each country. We use the invoice_id column to tally the number of invoices.
- **GROUP BY billing_country**: This groups the results by country, so the count is aggregated for each distinct billing_country.
- **ORDER BY Invoice_count DESC**: The result set is sorted in descending order, meaning countries with the most invoices will appear first.
- **LIMIT 4**: This restricts the output to the top 4 countries with the highest invoice counts.

-> What are top 3 values of total invoice?

Ans:

Solution 1:

For top 3 values:

```
SELECT
   total
FROM
   invoice
ORDER BY
   total DESC
LIMIT 3;
```

Intuition:

- **Retrieve Total Invoice Amounts**: The goal is to return the total invoice amounts in descending order.
- **Sort by Total**: Sorting by the total column in descending order ensures the highest totals appear at the top.
- **Limit to 3 Rows**: We limit the result to only 3 rows to get the top 3 total amounts, even if some values are repeated.

Explanation:

- **ORDER BY total DESC**: Sorting by the total in descending order ensures that the highest invoice values are prioritized.
- **LIMIT 3**: This limits the result set to the top 3 rows, giving us the highest 3 total values, regardless of whether they are distinct.

Solution 2:

For distinct top 3 values

```
SELECT
   DISTINCT total
FROM
   invoice
ORDER BY
   total DESC
LIMIT 3;
```

Intuition:

- **Remove Duplicates**: To ensure that only unique total values are returned, we use the DISTINCT keyword.
- **Sort by Total**: Sorting the unique totals in descending order ensures the highest distinct totals are shown first.
- **Limit to 3 Rows**: We limit the result to 3 rows, ensuring that only the top 3 distinct invoice totals are returned.

Explanation:

- **DISTINCT total**: This ensures that only unique total invoice values are included in the result, filtering out any duplicate totals.
- **ORDER BY total DESC**: Sorting by the total in descending order ensures that the highest unique totals are prioritized.
- **LIMIT 3**: This limits the result set to the top 3 distinct total values, giving us the highest 3 unique invoice totals.

-> Which city has the best customers? We would like to throw a promotional Music Festival in the city we made the most money. Write a query that returns one city that has the highest sum of invoice totals. Return both the city name & sum of all invoice totals.

Ans:

```
SELECT
   Billing_City AS City,
   SUM(total)
FROM
   invoice
GROUP BY
   City
ORDER BY
   City DESC
LIMIT 1;
```

Intuition:

- **Group by City**: We need to sum up the total invoice amounts for each city. Therefore, we group the data by billing_city.
- **Calculate Total Revenue**: The sum of all invoice totals for each city is calculated using SUM(total).
- **Sort by Revenue**: Sorting by the total revenue in descending order ensures the city with the highest total revenue appears first.
- **Return the Top Result**: We limit the result to one row to return the city with the highest total invoice sum.

Explanation:

- **SUM(total)**: This aggregates the invoice totals for each city, giving us the total revenue for each billing city.
- **GROUP BY billing_city**: Grouping by the billing_city ensures that we calculate the total invoice amount separately for each city.
- **ORDER BY Total_Revenue DESC**: Sorting the total revenue in descending order ensures the city with the highest total invoice amount comes first.
- **LIMIT 1**: We use LIMIT 1 to return only the city with the highest total invoice sum, as we're only interested in the top city.

-> Who is the best customer? The customer who has spent the most money will be declared the best customer. Write a query that returns the person who has spent the most money.

Ans:

Solution 1:

```
SELECT
   c.first_name AS first_name,
   c.last_name AS last_name,
   SUM(i.total) AS total
FROM
   invoice AS i
JOIN
   customer AS c ON i.customer_id = c.customer_id
GROUP BY
   c.customer_id
ORDER BY
   total DESC
LIMIT 1;
```

Intuition:

- **Identify Customer Spending**: We need to sum the total money spent by each customer, which is stored in the invoice table.
- **Join Invoice and Customer Tables**: By joining the invoice table with the customer table, we can map the total invoice amounts to the respective customers.

- **Group by Customer**: We aggregate the total invoice amounts per customer using GROUP BY on customer_id.
- **Sort by Total in Descending Order**: Sorting the results by total spending in descending order helps us find the customer who has spent the most.
- **Return the Best Customer**: By using LIMIT 1, we return the customer with the highest total spending.

Explanation:

- **SUM(i.total)**: Calculates the total spending for each customer by summing up all the invoice totals associated with their customer_id.
- **JOIN customer and invoice**: We join the invoice table with the customer table to associate invoice totals with customer details.
- **GROUP BY c.customer_id**: Groups the sum of invoice totals by each customer to get the total spending per customer.
- **ORDER BY total DESC**: Sorts the customers by their total spending in descending order, so the highest spender appears first.
- **LIMIT 1**: Ensures that only the top customer is returned.

Solution 2:

```
SELECT
   c.first_name AS first_name,
   c.last_name AS last_name,
   ROUND(CAST(SUM(i.total) AS NUMERIC), 2) AS total
FROM
   invoice AS i
JOIN
   customer AS c ON i.customer_id = c.customer_id
GROUP BY
   c.customer_id
ORDER BY
   total DESC
LIMIT 1;
```

Intuition:

- **Same Approach as Solution 1**: We follow the same approach as the first solution, where we join the invoice and customer tables, calculate the sum of invoice totals per customer, and group the results by customer_id.
- **Rounding the Total**: The only difference here is the rounding of the total spending to 2 decimal places using the ROUND function. This is useful for displaying the monetary value in a cleaner format.

Explanation:

- **ROUND(CAST(SUM(i.total) AS NUMERIC), 2)**: In this solution, we round the total invoice amount to 2 decimal places to ensure monetary values are formatted correctly.
- **JOIN, GROUP BY, and ORDER BY**: The logic remains the same as in the first solution, where we join the tables, group by customer, and order by total spending.

- **LIMIT 1**: Returns the top customer based on their total spending.

Question Set 2

-> Write query to return the email, first name, last name, & Genre of all Rock Music listeners. Return your list ordered alphabetically by email starting with A.

Ans:

Solution 1:

```
WITH CTE AS (
   SELECT
      c.email,
      c.first_name,
      c.last_name,
      g.name as genre
   FROM
      customer AS c
   JOIN
      invoice AS i ON c.customer_id = i.customer_id
   JOIN
      invoice_line AS il ON i.invoice_id = il.invoice_id
   JOIN
      track AS t ON t.track_id = il.track_id
   JOIN
      genre AS g ON t.genre_id = g.genre_id
)
SELECT
   DISTINCT email AS Email,
   first_name AS First_name,
   last_name AS Last_name,
   genre
FROM
   CTE
WHERE
   genre = 'Rock'
ORDER BY
   Email;
```

Intuition:

- **Identify Customers Who Listen to Rock**: We need to join various tables such as customer, invoice, invoice_line, track, and genre to identify customers who have purchased tracks in the 'Rock' genre.
- **Filter for the 'Rock' Genre**: After joining the tables, we filter the data by the 'Rock' genre.
- **Remove Duplicates**: Using DISTINCT ensures we don't get duplicate entries for customers who might have purchased multiple Rock tracks.
- **Order by Email**: The result should be sorted alphabetically by the email column.

Explanation:

- **WITH CTE AS**: We use a common table expression (CTE) to simplify the query. This CTE joins the customer, invoice, invoice_line, track, and genre tables to get customer details along with their genre preferences.
- **DISTINCT**: We use the DISTINCT clause in the final SELECT to eliminate any duplicate rows that might occur if a customer purchased multiple tracks in the same genre.
- **Filter by 'Rock'**: The WHERE genre = 'Rock' condition ensures that we only return customers who have listened to Rock music.
- **ORDER BY Email**: Finally, we order the results alphabetically by the email field to match the requirement of sorting the list by email starting from 'A'.

Solution 2:

```
WITH CTE AS (
  SELECT
    DISTINCT c.email,
    c.first_name,
    c.last_name,
    g.name as genre
  FROM
    customer AS c
  JOIN
    invoice AS i ON c.customer_id = i.customer_id
  JOIN
    invoice_line AS il ON i.invoice_id = il.invoice_id
  JOIN
    track AS t ON t.track_id = il.track_id
  JOIN
    genre AS g ON t.genre_id = g.genre_id
)
SELECT
  email AS Email,
  first_name AS First_name,
  last_name AS Last_name,
  genre
FROM
  CTE
WHERE
  genre = 'Rock'
ORDER BY
  Email;
```

Intuition:

- **Pre-filter Duplicates in CTE**: In this optimized version, the DISTINCT keyword is applied inside the CTE itself, reducing the result set early on and minimizing unnecessary processing in the final query.
- **Same Logic for Filtering and Ordering**: The logic for filtering by the 'Rock' genre and ordering by email remains the same.

Explanation:

- **DISTINCT in the CTE**: This query applies DISTINCT in the CTE, meaning duplicates are removed early on, which can improve performance in large datasets.
- **Final Query**: The final SELECT and filtering logic remains the same, with the goal of returning the required customer details and genre, ordered by email.

-> Let's invite the artists who have written the most rock music in our dataset. Write a query that returns the Artist name and total track count of the top 10 rock bands.

Ans:

```
SELECT
   ar.name AS artist_name,
   COUNT(t.track_id) AS rock_track_cnt
FROM
   Track AS t
JOIN
   album AS a ON t.album_id = a.album_id
JOIN
   artist AS ar ON a.artist_id = ar.artist_id
GROUP BY
   ar.name, t.genre_id
HAVING
   t.genre_id = '1'
ORDER BY
   rock_track_cnt DESC
LIMIT 10;
```

Intuition:

- **Identify Rock Tracks**: First, we need to filter for tracks that belong to the 'Rock' genre. We use the genre_id = 1 assumption based on the provided schema, where each genre has a unique ID.
- **Group by Artist**: We count the total number of Rock tracks for each artist by grouping the results by the artist's name and filtering specifically for the 'Rock' genre.
- **Order by Track Count**: To identify the top 10 artists who have composed the most Rock music, we order the results by the total number of Rock tracks in descending order.

Explanation:

- **Table Joins**: We join the track table with the album table using the album_id, and then the album table with the artist table using artist_id. This allows us to link each track with the corresponding artist.
- **Grouping and Filtering**: After joining, we group the tracks by artist name and genre_id, which allows us to aggregate the track count for each artist specifically within the 'Rock' genre.
- **Filtering by Genre**: The HAVING t.genre_id = '1' condition ensures we are only counting tracks that belong to the Rock genre.
- **Ordering and Limiting**: We sort the artists by their Rock track count in descending order and limit the result to the top 10 artists.

-> Return all the track names that have a song length longer than the average song length.Return the Name and Milliseconds for each track. Order by the song length with the longest songs listed first.

Ans:

Solution 1:

```
SELECT
   "name",
   Milliseconds AS Minutes
FROM
   Track
WHERE
   Milliseconds > (SELECT AVG(Milliseconds) FROM Track)
ORDER BY
   Milliseconds DESC;
```

Intuition:

- **Average Song Length**: First, we calculate the average song length in milliseconds using a subquery (SELECT AVG(Milliseconds) FROM Track).
- **Filter for Longer Songs**: We filter tracks whose length in milliseconds is greater than the average.
- **Order by Length**: To show the longest tracks first, we order the results in descending order based on the length in milliseconds.

Explanation:

- **Subquery for Average Length**: The subquery (SELECT AVG(Milliseconds) FROM Track) calculates the average duration of all tracks in the Track table. This result is used in the WHERE clause to filter for songs that are longer than the average.
- **Filtering**: The main query selects the tracks whose Milliseconds value exceeds the calculated average.
- **Ordering**:The results are ordered by Milliseconds DESC to list the longest tracks first.

Solution 2:

```
SELECT
   "name",
   Milliseconds / 60000 AS Minutes
FROM
   Track
WHERE
   Milliseconds > (SELECT AVG(Milliseconds) FROM Track)
ORDER BY
      Milliseconds DESC;
```

Intuition:

- **Convert Milliseconds to Minutes**: In this version of the query, we convert the Milliseconds field to minutes by dividing it by 60,000 (since there are 60,000 milliseconds in a minute).
- **Filter and Order**: The filtering and ordering logic remains the same, based on the comparison to the average song length.

Explanation:

- **Division for Minutes:** The key difference in this query is the calculation Milliseconds / 60000 AS Minutes, which converts the track length from milliseconds to minutes for easier interpretation.
- **Rest of the Logic**: Like the first query, we filter tracks that are longer than the average duration and order them by length in descending order.

Question Set 3

-> Find how much amount spent by each customer on artists? Write a query to return customer name, artist name and total spent.

Ans:

```
SELECT
   c.first_name,
   c.last_name,
   ar.name AS artist_name,
   ROUND(SUM(i.total)::numeric, 2) AS total_spent
FROM
   customer AS c
JOIN
   invoice AS i ON c.customer_id = i.customer_id
JOIN
   invoice_line AS il ON i.invoice_id = il.invoice_id
JOIN
   track AS t ON il.track_id = t.track_id
JOIN
   album AS a ON t.album_id = a.album_id
JOIN
   artist AS ar ON a.artist_id = ar.artist_id
GROUP BY
   c.first_name,
   c.last_name,
       ar.name;
```

Intuition:

- **Customer and Artist Relationship**: The relationship between customers and artists is indirect, going through multiple tables. The customer purchases invoices, which are linked to invoice_lines, and the tracks in these invoice lines are linked to specific albums that belong to artists.
- **Aggregate Spending**: For each customer and artist combination, we need to sum the total amount the customer spent across all purchases related to that artist's tracks.
- **Rounding the Total**: The total spent is rounded to two decimal places for clarity.

Explanation:

- **Table Joins**: We start with the customer table (c) and join it to the invoice table (i) using the customer_id. The invoice table is then joined to invoice_line (il), linking invoices to specific tracks. From the track table (t), we obtain the album_id, which links us to the album table (a), and finally to the artist table (ar) using the artist_id.
- **Group By**: The query groups by the customer's first and last name (c.first_name, c.last_name) and the artist's name (ar.name). This ensures that we calculate the total amount spent by each customer on each artist.

- **Total Calculation**: We use SUM(i.total) to compute the total amount each customer has spent on the tracks by each artist. The result is then rounded to two decimal places using ROUND(SUM(i.total)::numeric, 2).
- **Alternative Approaches**: Instead of using the PostgreSQL-specific ::numeric typecast, you could alternatively use CAST(SUM(i.total) AS numeric) to achieve the same result.

-> We want to find out the most popular music Genre for each country. We determine the most popular genre as the genre with the highest amount of purchases. Write a query that returns each country along with the top Genre. For countries where the maximum number of purchases is shared return all Genres.

Ans:

Solution 1:

```
WITH CTE AS (
  SELECT
    c.country AS country,
    g.name AS genre,
    COUNT(*) AS quantity
  FROM
    customer AS c
  JOIN
    invoice AS i ON c.customer_id = i.customer_id
  JOIN
    invoice_line AS il ON i.invoice_id = il.invoice_id
  JOIN
    track AS t ON il.track_id = t.track_id
  JOIN
    genre AS g ON t.genre_id = g.genre_id
  GROUP BY
    c.country, g.name
)
SELECT
  country,
  genre,
  quantity
FROM
  CTE
WHERE
  (country, quantity) IN (
    SELECT
      country,
      MAX(quantity)
    FROM
      CTE
    GROUP BY
      country
  )
ORDER BY
  country;
```

Intuition:

- **Determine Genre Popularity**: The goal is to calculate the number of purchases for each genre in each country, allowing us to identify the most popular genres based on purchase counts.
- **Create CTE**: The Common Table Expression (CTE) aggregates the data by country and genre, counting the number of purchases for each combination.
- **Filter for Maximum Quantities**: The outer query filters results based on whether a genre's quantity matches the maximum quantity for each country, allowing us to identify the most popular genre(s) for each country.
- **Return Results**: Finally, the results are ordered by country for easy readability.

Explanation:

- **CTE Aggregation**: In the CTE, we join multiple tables (customer, invoice, invoice_line, track, and genre) to derive the quantity of purchases for each genre in each country. We group by both c.country and g.name to get the correct counts.
- **WHERE Clause Filtering**: The filtering occurs through a subquery in the WHERE clause, which retrieves the maximum quantity for each country from the CTE. This ensures that only the most popular genres are returned.
- **DISTINCT Quantity Clause**: Note that the SELECT DISTINCT quantity at the beginning serves to view unique purchase counts, but it is not used in the final output.

Solution 2:

```
WITH CTE AS (
   SELECT
      c.country AS country,
      g.name AS genre,
      COUNT(*) AS quantity
   FROM
      customer AS c
   JOIN
      invoice AS i ON c.customer_id = i.customer_id
   JOIN
      invoice_line AS il ON i.invoice_id = il.invoice_id
   JOIN
      track AS t ON il.track_id = t.track_id
   JOIN
      genre AS g ON t.genre_id = g.genre_id
   GROUP BY
      c.country, g.name
),
MaxQuantityPerCountry AS (
   SELECT
      country,
      MAX(quantity) AS max_quantity
   FROM
      CTE
   GROUP BY
      country
)
SELECT
   CTE.country,
   CTE.genre,
   CTE.quantity
FROM
   CTE
JOIN
   MaxQuantityPerCountry AS M
   ON CTE.country = M.country AND CTE.quantity = M.max_quantity
ORDER BY
   CTE.country;
```

Intuition:

- **Create CTE for Purchases**: Similar to the first solution, we start with a CTE to gather the count of purchases for each genre in each country.
- **Calculate Maximum Quantities**: The second CTE, MaxQuantityPerCountry, captures the maximum quantity of purchases for each country, isolating the most popular genre(s).
- **Efficient Joining**: We then join the original CTE with this maximum quantity CTE, filtering down to those genres that have the maximum purchase count for their respective countries. This allows us to retrieve all genres that are tied for popularity in a single pass.
- **Ordered Results**: Finally, we return the results, ordered by country.

Explanation:

- **CTE Creation**: The first CTE works the same way as in the first solution, counting purchases for each genre and country.
- **Max Quantity CTE**: The second CTE (MaxQuantityPerCountry) simplifies the process of finding the maximum purchase quantity per country, helping to reduce the complexity of the final query.
- **Join for Final Results**: By joining the original CTE with the maximum quantity results, we can easily retrieve all genres tied for the maximum purchase count in each country. This is often more efficient than using a WHERE clause for filtering.
- **Ordering**: The results are ordered by country for a structured output.

-> Write a query that determines the customer that has spent the most on music for each country. Write a query that returns the country along with the top customer and how much they spent. For countries where the top amount spent is shared, provide all customers who spent this amount.

Ans:

Solution 1:

```
WITH CTE AS (
  SELECT
    c.country AS country,
    c.first_name AS first_name,
    c.last_name AS last_name,
    ROUND(SUM(i.total)::numeric, 2) AS total
  FROM
    customer AS c
  JOIN
    invoice AS i ON c.customer_id = i.customer_id
  GROUP BY
    c.country, c.first_name, c.last_name
)
SELECT
  country,
  first_name,
  last_name,
  total
FROM
  CTE
WHERE
  (country, total) IN (
    SELECT
      country,
      MAX(total)
    FROM
      CTE
    GROUP BY
      country
  )
ORDER BY
  country;
```

Intuition:

- **Aggregate Spending**: We need to aggregate the total spending for each customer per country. This involves summing up the invoice totals for each customer and grouping by country, first name, and last name.
- **Create CTE**: A Common Table Expression (CTE) is used to calculate the total amount spent by each customer in their respective countries.
- **Filter for Top Spending**: The outer query filters for the maximum spending amount per country using a subquery that finds the highest total for each country.
- **Return Results**: Finally, the results are ordered by country for clarity.

Explanation:

- **CTE Structure**: In the CTE, we perform a JOIN between the customer and invoice tables to aggregate the total spending. We use ROUND(SUM(i.total)::numeric, 2) to ensure the total is rounded to two decimal places.
- **Subquery Filtering**: The subquery in the WHERE clause retrieves the maximum spending amount for each country, allowing us to filter the CTE results for only those customers who have spent that amount.
- **Ordering**: The results are sorted by country, presenting a clear overview of the top customers.

Solution 2:

```
WITH CTE AS (
    SELECT
        c.country AS country,
        c.first_name AS first_name,
        c.last_name AS last_name,
        ROUND(SUM(i.total)::numeric, 2) AS total
    FROM
        customer AS c
    JOIN
        invoice AS i ON c.customer_id = i.customer_id
    GROUP BY
        c.country, c.first_name, c.last_name
),
MaxSpendingPerCountry AS (
    SELECT
        country,
        MAX(total) AS max_total
    FROM
        CTE
    GROUP BY
        country
)
SELECT
    CTE.country,
    CTE.first_name,
    CTE.last_name,
    CTE.total
FROM
    CTE
JOIN
    MaxSpendingPerCountry AS M
    ON CTE.country = M.country AND CTE.total = M.max_total
ORDER BY
    CTE.country;
```

Intuition:

- **Aggregate and Max Calculation**: The first CTE aggregates total spending per customer and country, while the second CTE calculates the maximum spending per country.
- **Join for Results**: The final query joins the CTE with the maximum spending CTE to filter down to customers who have the maximum spending in their respective countries.
- **Efficiency**: This method is often more efficient for larger datasets, as it clearly separates the aggregation and maximum calculations, leading to fewer rows in the final join.

Explanation:

- **CTE Usage**: The first CTE functions similarly to the first solution, summing the total amounts spent by customers. The second CTE then retrieves the maximum spending for each country.
- **JOINing for Top Customers**: By joining the CTE with the maximum spending results, we efficiently identify all customers with the highest spending amounts without needing a complex filtering method.

- **Ordering**: The results are ordered by country, ensuring that the output is structured for easy analysis.

Verifying Customers with Multiple Purchases:

```
WITH CTEE AS (
  SELECT
    c.country AS country,
    c.first_name AS first_name,
    c.last_name AS last_name,
    COUNT(i.customer_id) AS cnt,
    ROUND(SUM(i.total)::numeric, 2) AS tot
  FROM
    customer AS c
  JOIN
    invoice AS i ON c.customer_id = i.customer_id
  GROUP BY
    c.country, c.first_name, c.last_name
)
SELECT
  country,
  first_name,
  last_name,
  cnt,
  tot
FROM
  CTEE
WHERE
  cnt > 1
ORDER BY
  country;
```

Explanation:

- **Count Purchases**: This query counts how many invoices each customer has and their total spending.
- **Filtering**: It filters to include only those customers who have more than one purchase, providing additional insights into customer behavior in each country.
- **Ordering**: The results are sorted by country, helping to visualize customers with multiple purchases easily.