**Answers 3.9**

**STEP 1**

**QUERY #1**

A screenshot of a computer code

AI-generated content may be incorrect.

**CODE:**

WITH top\_cities AS

(SELECT city.city

FROM customer

INNER JOIN address ON customer.address\_id = address.address\_id

INNER JOIN city ON address.city\_id = city.city\_id

INNER JOIN country ON city.country\_id = country.country\_id

GROUP BY city.city, country.country

ORDER BY COUNT(customer.customer\_id) DESC

LIMIT 10),

top\_customers AS

(SELECT customer.customer\_id, customer.first\_name, customer.last\_name, city.city, country.country, SUM(payment.amount) AS total\_amount\_paid

FROM payment

INNER JOIN customer ON payment.customer\_id = customer.customer\_id

INNER JOIN address ON customer.address\_id = address.address\_id

INNER JOIN city ON address.city\_id = city.city\_id

INNER JOIN country ON city.country\_id = country.country\_id

WHERE city.city IN (SELECT city FROM top\_cities)

GROUP BY customer.customer\_id, customer.first\_name, customer.last\_name, city.city, country.country

ORDER BY total\_amount\_paid DESC

LIMIT 5)

SELECT AVG(top\_customers.total\_amount\_paid) AS average

FROM top\_customers

**EXPLANATION:**

First CTE (top\_cities):

* Identifies the top 10 cities that have the highest number of customers.
* Uses COUNT(customer.customer\_id) to count the number of customers in each city.
* Sorts cities in descending order.
* Uses LIMIT 10 to return only the top 10 cities

Second CTE (top\_customers):

* Identifies the top 5 customers who have made the highest total payments.
* Only considers customers from the top 10 cities.
* Uses SUM(payment.amount) to calculate each customer’s total payments.
* Groups by customer details (customer\_id, first\_name, last\_name, city, country).
* Sorts by total\_amount\_paid DESC.
* Uses LIMIT 5 to keep only the top 5 customers.

Main query:

* Takes the total\_amount\_paid from the top\_customers CTE.
* Computes the average total payment among these top 5 customers using AVG(top\_customers.total\_amount\_paid).

**QUERY #2:**

**A text on a white background

AI-generated content may be incorrect.**

**OUTPUT:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**CODE:**

**WITH top\_cities AS**

**(SELECT city.city**

**FROM customer**

**INNER JOIN address ON customer.address\_id = address.address\_id**

**INNER JOIN city ON address.city\_id = city.city\_id**

**INNER JOIN country ON city.country\_id = country.country\_id**

**GROUP BY city.city, country.country**

**ORDER BY COUNT(customer.customer\_id) DESC**

**LIMIT 10),**

**top\_customers AS**

**(SELECT customer.customer\_id, customer.first\_name, customer.last\_name, city.city, country.country, SUM(payment.amount) AS total\_amount\_paid**

**FROM payment**

**INNER JOIN customer ON payment.customer\_id = customer.customer\_id**

**INNER JOIN address ON customer.address\_id = address.address\_id**

**INNER JOIN city ON address.city\_id = city.city\_id**

**INNER JOIN country ON city.country\_id = country.country\_id**

**WHERE city.city IN (SELECT city FROM top\_cities)**

**GROUP BY customer.customer\_id, customer.first\_name, customer.last\_name, city.city, country.country**

**ORDER BY total\_amount\_paid DESC**

**LIMIT 5)**

**SELECT country.country,**

**COUNT(DISTINCT customer.customer\_id) AS all\_customer\_count,**

**COUNT(DISTINCT top\_customers.customer\_id) AS top\_customer\_count**

**FROM customer**

**INNER JOIN address ON customer.address\_id = address.address\_id**

**INNER JOIN city ON address.city\_id = city.city\_id**

**INNER JOIN country ON city.country\_id = country.country\_id**

**LEFT JOIN top\_customers ON customer.customer\_id = top\_customers.customer\_id**

**GROUP BY country.country**

**ORDER BY top\_customer\_count DESC**

**EXPLANATION:**

First CTE (top\_cities):

* Same as in Query 1.
* Finds the top 10 cities based on customer count.
* Used later to filter customers.

Second CTE (top\_customers):

* Same as in Query 1.
* Identifies top 5 high paying customers from the top 10 cities.

Main Query:

* Counts all customers per country using COUNT(DISTINCT customer.customer\_id).
* Counts top 5 high paying customers per country using COUNT(DISTINCT top\_customers.customer\_id).
* Uses LEFT JOIN top\_customers so that customers who are not in top\_customers will have NULL values.
* Groups results by country.country.
* Orders by top\_customer\_count DESC to see which countries have the most high paying customers.

**STEP 2: Compare the performance of your CTEs and subqueries.**

**QUERY #1**

**CTE ANALYSIS:**

****

**Planning time: 3.920 ms**

**Execution time: 2.368 ms**

**QUERY #1**

**SUBQUERY ANALYSIS:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Planning time: 3.327 ms**

**Execution time 2.438 ms**

**QUERY #2**

**CTE ANALYSIS:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Planning time: 3.615 ms**

**Execution Time: 3.909 ms**

**QUERY #2**

**SUBQUERY ANALYSIS**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Planning time: 4.367 ms**

**Execution time: 4.733 ms**

To evaluate performance, we run both the original subquery-based queries and the CTE based versions using EXPLAIN ANALYZE.

Expected Result for Subqueries:

* PostgreSQL executes the subquery multiple times, increasing execution cost.
* Potential optimization issues if the same subquery is executed repeatedly.

Expected result for CTEs:

* CTEs prevent repeated execution of the same logic, reducing query cost.
* CTE result is stored temporarily, improving efficiency.
* Performance improves in queries that reuse the same data.

**COMPARISON:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **QUERY #1** | | **QUERY #2** | |
|  | **Planning** | **Execution** | **Planning** | **Execution** |
| **CTE** | 3.920 ms | **2.368 ms** | 3.615 ms | **3.909 ms** |
| **SUBQUERY** | 3.327 ms | 2.438 ms | 4.367 ms | 4.733 ms |

Which Approach is Better?

CTEs generally perform better when the same subquery logic is used multiple times, because PostgreSQL can evaluate them once and reuse the result.

Subqueries might be faster in simpler cases, but for complex queries, CTEs improve both readability and performance.

CTEs are the best choice in this case, as they allow reusing query results without recalculating them multiple times.

**STEP 3: Challenges faced when replacing your subqueries with CTEs.**

1. One of the main challenges I faced when replacing subqueries with CTEs was restructuring the query logic while maintaining accuracy. With subqueries, everything was nested, and the dependencies were clear within each section. However, when using CTEs, I had to break the logic into separate, reusable steps while ensuring that each CTE correctly fed into the next part of the query.
2. Another challenge was performance optimization. While CTEs improve readability, they can sometimes lead to performance issues if not handled properly. I had to consider whether the CTEs would be recomputed multiple times and if using WITH would be beneficial. Additionally, adjusting joins, especially ensuring that the final counts in Query 2 remained correct, required extra attention, as changing from subqueries to CTEs sometimes altered how NULL values were handled.