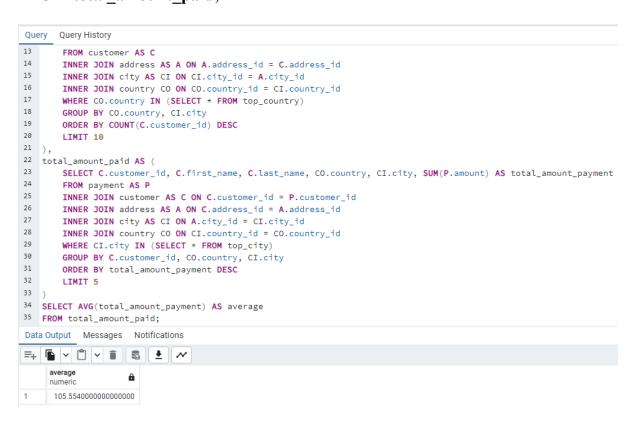
Data Immersion 3.9

```
Step 1
Query 1
WITH top country AS (
  SELECT CO.country
  FROM customer AS C
  INNER JOIN address AS A ON A.address id = C.address id
  INNER JOIN city AS CI ON CI.city id = A.city id
  INNER JOIN country CO ON CO.country id = CI.country id
  GROUP BY CO.country
  ORDER BY COUNT(C.customer id) DESC
  LIMIT 10
),
top city AS (
  SELECT CL.city
  FROM customer AS C
  INNER JOIN address AS A ON A.address_id = C.address_id
  INNER JOIN city AS CI ON CI.city id = A.city id
  INNER JOIN country CO ON CO.country id = CI.country id
  WHERE CO.country IN (SELECT * FROM top country)
  GROUP BY CO.country, CI.city
  ORDER BY COUNT(C.customer_id) DESC
  LIMIT 10
),
total amount paid AS (
  SELECT C.customer id, C.first name, C.last name, CO.country, CI.city,
SUM(P.amount) AS total amount payment
  FROM payment AS P
```

```
INNER JOIN customer AS C ON C.customer_id = P.customer_id
INNER JOIN address AS A ON C.address_id = A.address_id
INNER JOIN city AS CI ON A.city_id = CI.city_id
INNER JOIN country CO ON CI.country_id = CO.country_id
WHERE CI.city IN (SELECT * FROM top_city)
GROUP BY C.customer_id, CO.country, CI.city
ORDER BY total_amount_payment DESC
LIMIT 5
```

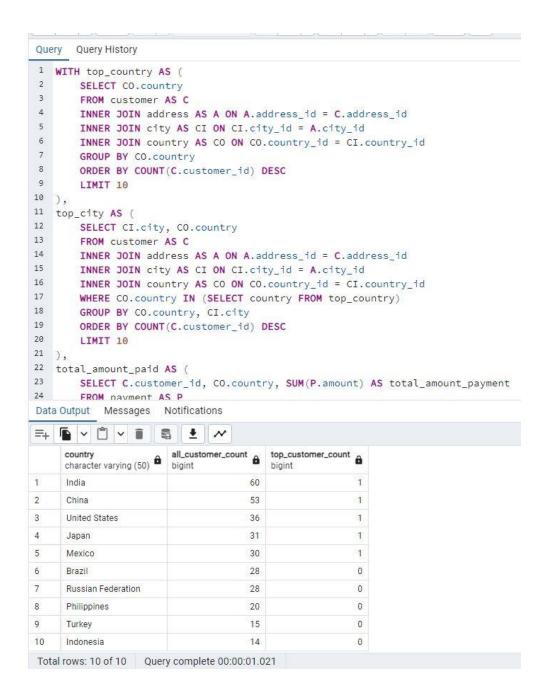
SELECT AVG(total_amount_payment) AS average FROM total amount paid;



)

```
Query 2
WITH top_country AS (
  SELECT CO.country
 FROM customer AS C
 INNER JOIN address AS A ON A.address_id = C.address_id
 INNER JOIN city AS CI ON CI.city_id = A.city_id
 INNER JOIN country AS CO ON CO.country_id = CI.country_id
  GROUP BY CO.country
  ORDER BY COUNT(C.customer_id) DESC
 LIMIT 10
),
top_city AS (
 SELECT CI.city, CO.country
 FROM customer AS C
 INNER JOIN address AS A ON A.address_id = C.address_id
 INNER JOIN city AS CI ON CI.city_id = A.city_id
 INNER JOIN country AS CO ON CO.country_id = CI.country_id
  WHERE CO.country IN (SELECT country FROM top_country)
  GROUP BY CO.country, CI.city
  ORDER BY COUNT(C.customer_id) DESC
 LIMIT 10
),
total_amount_paid AS (
  SELECT C.customer_id, CO.country, SUM(P.amount) AS
total amount payment
```

```
FROM payment AS P
 INNER JOIN customer AS C ON C.customer id = P.customer id
 INNER JOIN address AS A ON A.address id = C.address id
 INNER JOIN city AS CI ON CI.city_id = A.city_id
 INNER JOIN country AS CO ON CO.country_id = CI.country_id
  WHERE CL.city IN (SELECT city FROM top_city)
  GROUP BY C.customer_id, CO.country
  ORDER BY SUM(P.amount) DESC
 LIMIT 5
)
SELECT CO.country,
   COUNT(DISTINCT C.customer_id) AS all_customer_count,
   COUNT(DISTINCT T.customer_id) AS top_customer_count
FROM customer AS C
INNER JOIN address AS A ON A.address id = C.address id
INNER JOIN city AS CI ON CI.city_id = A.city_id
INNER JOIN country AS CO ON CO.country_id = CI.country_id
LEFT JOIN total_amount_paid AS T ON T.customer_id = C.customer_id AND
T.country = CO.country
GROUP BY CO.country
ORDER BY all customer count DESC
LIMIT 10;
```



3. Approach

I initially dissected the queries from our complex example, color-coding the outer sections for clarity before transforming them into CTEs. Each section was tackled by first creating a CTE using the **WITH** clause, then inserting the subquery previously designed. Following this, I incorporated the outer sections from the earlier task, ensuring they correctly referenced the CTE established at the start. For instance, in the second query, this involved adjusting the left join to connect properly with the CTE, streamlining the query structure for better readability and efficiency. This structure made my SQL queries more readable

and allowed easier maintenance and the potential to reuse the CTE within the same query, enhancing overall query organisation.

Step 2

1. CTEs generally offer better readability and maintainability, making complex queries more organised and easier to understand. They can also be referenced multiple times within the same query, which adds a layer of flexibility. While CTEs might not inherently perform faster than subqueries, their clarity and structured approach can lead to more reliable and easier to optimise SQL code.

2.

Comparing costs:

Query 1 task 3.8: "Aggregate (cost=166.06..166.07 rows=1 width=32)" 66msec Query 2 task 3.8: "Limit (cost=268.77..268.80 rows=10 width=25)" 69msec Query 1 task 3.9: "Aggregate (cost=166.06..166.07 rows=1 width=32)" 65msec Query 2 task 3.9: "Limit (cost=268.77..268.80 rows=10 width=25)" 66msec

Step 3

When I set out to replace subqueries with Common Table Expressions (CTEs) in my SQL queries, I encountered a variety of challenges. First and foremost, CTEs require a top-down approach, meaning you need to start from the innermost subquery and work your way outwards, maintaining the original hierarchical relationships of the query logic. This often involves a detailed understanding of each subquery's role within the larger query, which can add complexity and potential for confusion if not carefully managed.

Another challenge arises in the way CTEs are utilised within the main query. Unlike subqueries, which are often contained within parentheses and can be embedded directly where needed, CTEs are defined at the beginning of the query and are then referenced in the FROM or WHERE clauses as if they were actual tables. This shift in syntax and conceptual approach requires careful planning to ensure that all references are correctly aligned with the defined CTEs, enhancing the need for meticulous attention to detail to prevent errors and ensure the integrity of the query's logic.