COMMON TABLE EXPRESSIONS (CTEs)

Step 1A: Find the average amount paid by the top 5 customers.

Query | Query History

```sql
-- Step 1: Calculate the total amount paid by each customer, along with their name, city, and country
WITH customer_payments AS (
    SELECT
        A.customer_id,
        A.first_name,
        A.last_name,
        D.country,
        C.city,
        SUM(P.amount) AS total_amount_paid  -- total amount paid by the customer
    FROM customer A
    JOIN address B ON A.address_id = B.address_id  -- link customer to their address
    JOIN city C ON B.city_id = C.city_id            -- link address to city
    JOIN country D ON C.country_id = D.country_id   -- link city to country
    JOIN payment P ON A.customer_id = P.customer_id -- link customer to their payments
    GROUP BY A.customer_id, A.first_name, A.last_name, D.country, C.city
),

-- Step 2: Find the top 10 countries with the most customers
top_countries AS (
    SELECT
        D.country                           -- country name
    FROM customer A
    JOIN address B ON A.address_id = B.address_id
    JOIN city C ON B.city_id = C.city_id
    JOIN country D ON C.country_id = D.country_id
    GROUP BY D.country                      -- group by country
    ORDER BY COUNT(A.customer_id) DESC  -- sort by number of customers in descending order
    LIMIT 10                                -- take only the top 10 countries
),

-- Step 3: Find the top 10 cities within those countries, based on customer count
top_cities AS (
    SELECT
        C.city                              -- city name
    FROM customer A
    JOIN address B ON A.address_id = B.address_id
    JOIN city C ON B.city_id = C.city_id
    JOIN country D ON C.country_id = D.country_id
    WHERE D.country IN (SELECT country FROM top_countries)  -- only include cities from top countries
    GROUP BY D.country, C.city              -- group by both country and city
    ORDER BY COUNT(A.customer_id) DESC  -- sort by number of customers per city
    LIMIT 10                                -- take top 10 cities
),

-- Step 4: From the top cities, get the top 5 customers based on total amount paid
top_customers AS (
    SELECT *
    FROM customer_payments
    WHERE city IN (SELECT city FROM top_cities)         -- only include customers from top cities
    ORDER BY total_amount_paid DESC                     -- sort by spending amount (highest first)
    LIMIT 5                                              -- take top 5 paying customers
)

-- Step 5: Calculate the average amount paid by these top 5 customers
SELECT
    AVG(total_amount_paid) AS average_amount_paid       -- final result: average amount paid by top 5
FROM top_customers;
```

Data Output | Messages | Explain | Notifications

Showing rows: 1 to 1 | Page No: 1 of 1

	average_amount_paid numeric
1	105.5540000000000000

Step 1B: Find out how many of the top 5 customers you identified in step 1 are based within each country.

Query | Query History

```sql
-- Step 1: Calculate total amount paid by each customer
WITH customer_totals AS (
    SELECT
        p.customer_id,                    -- customer ID
        SUM(p.amount) AS total_sum        -- total amount paid by this customer
    FROM payment p
    GROUP BY p.customer_id                -- group by customer to get total per customer
),

-- Step 2: Calculate the average total amount paid across all customers
avg_total_payment AS (
    SELECT
        AVG(total_sum) AS avg_payment     -- average total amount paid
    FROM customer_totals
),

-- Step 3: Identify top customers (those who paid more than the average)
top_customers AS (
    SELECT
        ct.customer_id                    -- customer ID
    FROM customer_totals ct
    CROSS JOIN avg_total_payment atp      -- include average value for comparison
    WHERE ct.total_sum > atp.avg_payment -- only keep customers above the average
),

-- Step 4: Combine top customer info with their country
customer_country AS (
    SELECT
        a.customer_id,
        d.country                         -- country of the customer
    FROM customer a
    JOIN address b ON a.address_id = b.address_id
    JOIN city c ON b.city_id = c.city_id
    JOIN country d ON c.country_id = d.country_id
),

-- Step 5: Count total and top customers per country
country_stats AS (
    SELECT
        cc.country,                                    -- country name
        COUNT(DISTINCT cc.customer_id) AS all_customer_count,     -- total customers in country
        COUNT(DISTINCT tc.customer_id) AS top_customer_count      -- top customers from that country
    FROM customer_country cc
    LEFT JOIN top_customers tc
        ON cc.customer_id = tc.customer_id            -- check if customer is a "top customer"
    GROUP BY cc.country                               -- group by country
)

-- Step 6: Return the top 10 countries with the most top customers
SELECT
    country,
    all_customer_count,     -- total number of customers
    top_customer_count      -- number of top customers (paid above average)
FROM country_stats
ORDER BY top_customer_count DESC      -- sort by number of top customers
LIMIT 10;                              -- return top 10 countries
```

Data Output | Messages | Explain | Notifications

Showing rows: 1 to 10 | Page No: 1 of 1

	country character varying (50)	all_customer_count bigint	top_customer_count bigint
1	India	60	26
2	China	53	25
3	United States	36	16
4	Japan	31	14
5	Russian Federation	28	13
6	Brazil	28	12
7	Mexico	30	11
8	Philippines	20	11
9	Taiwan	10	7
10	Turkey	15	7

Step 2:

I initially expected the subquery version to perform similarly or even slightly better than the CTE version, since inline subqueries often allow PostgreSQL to optimize the execution path more freely. However, in this case, the subquery version— which nests a CASE WHEN condition containing two subselects (one of them with a nested AVG() of a grouped subquery)— resulted in significantly higher planning and execution costs. This is because PostgreSQL evaluates those subqueries repeatedly for each row, which adds considerable overhead.

In contrast, the CTE version calculates total payments and the average once, in isolated steps, and then joins the results. This avoids redundant computation and leads to improved performance and efficiency. While PostgreSQL may materialize CTEs (treating them as temporary result sets), which can sometimes limit query planner optimizations, in this scenario, the clarity, modularity, and pre-aggregation provided by CTEs outweighed those drawbacks.

That said, I believe there are cases where the performance of a CTE is nearly identical to that of a subquery, and other cases where CTEs provide a substantial improvement. Therefore, it is important to test and analyze each query individually, using tools like EXPLAIN ANALYZE, rather than assuming one approach is always better than the other.

SUBQUERY STEP1A

Data Output | Messages | Explain | Notifications — Showing rows: 1 to 65 — Page No: 1 of 1

	QUERY PLAN text
1	Aggregate (cost=1078.11..1078.12 rows=1 width=32)

SUBQUERY STEP1B

Data Output | Messages | Explain | Notifications — Showing rows: 1 to 31 — Page No: 1 of 1

	QUERY PLAN text
1	Limit (cost=38213.96..38213.98 rows=10 width=25)

CTE STEP1A

Data Output | Messages | Explain | Notifications — Showing rows: 1 to 66 — Page No: 1 of 1

	QUERY PLAN text
1	Aggregate (cost=1125.02..1125.03 rows=1 width=32)

CTE STEP1B

Data Output | Messages | Explain | Notifications — Showing rows: 1 to 32 — Page No: 1 of 1

	QUERY PLAN text
1	Limit (cost=475.77..475.80 rows=10 width=25)

Step 3:

Replacing subqueries with CTEs was helpful for breaking down the logic into manageable, readable sections. It made the query structure easier to understand, especially when working with multi-step filtering (e.g. top countries → top cities → top customers). Each block could be explained and debugged independently.