28_IBM db2 Product Analytics Easy - Solution

Source - https://datalemur.com/questions/sql-ibm-db2-product-analytics

Running Notes

So I first list down the important components of the question and then build the query up with it

- generate data to populate a histogram that shows the number of unique queries run by employees during the third quarter of 2023 (July to September).

```
SELECT employee_id,count(query_id)
FROM queries
GROUP BY employee_id;
```

- count the number of employees who did not run any queries during this period.

```
WITH no_run_CTE AS (SELECT e.employee_id,q.query_id
FROM employees AS e
LEFT JOIN queries AS q
ON e.employee_id = q.employee_id)

WITH no_run_count_CTE AS (SELECT
0 AS unique_queries,
COUNT(employee_id) AS employee_count
FROM no_run_CTE
WHERE query_id IS NULL)
```

- Display the number of unique queries as histogram categories, along with the count of employees who executed that number of unique queries.

```
WITH queries_run AS (SELECT employee_id,count(query_id) as co
FROM queries
GROUP BY employee_id),

no_run_CTE AS (SELECT e.employee_id,q.query_id
FROM employees AS e
LEFT JOIN queries AS q
ON e.employee_id = q.employee_id),

no_run_count_CTE AS (SELECT
0 AS unique_queries,
COUNT(employee_id) AS employee_count
FROM no_run_CTE
WHERE query_id IS NULL)

SELECT * FROM no_run_count_CTE
UNION
SELECT count_of_queries AS unique_queries,
COUNT(count_of_queries) AS employee_count
FROM queries_run
GROUP BY count_of_queries
ORDER BY unique_queries
```

But the query output didn’t match

So back to square 1

```
WITH unique_queries_run_CTE AS (
SELECT employee_id, COUNT(DISTINCT(query_id))
FROM queries
WHERE query_starttime BETWEEN '2023-07-01T00:00:00Z' AND '2023-1
GROUP BY employee_id),

zero_queries_run_CTE AS (SELECT
e.employee_id,q.count
```

```
FROM employees AS e
LEFT JOIN unique_queries_run_CTE AS q
ON e.employee_id = q.employee_id),

final_0_CTE AS (SELECT
0 AS unique_queries,
count(employee_id) AS employee_count
FROM zero_queries_run_CTE
WHERE count IS NULL),

final_unique_CTE AS (SELECT
count AS unique_queries,
COUNT(count) AS employee_count
FROM unique_queries_run_CTE
GROUP BY count
ORDER BY count)

SELECT * from final_0_CTE
UNION ALL
SELECT * from final_unique_CTE
```

The above is the code that got accepted but there was a slight mistake I had made, that is worth documentation

```
query_starttime
```

is stored in UTC (a common practice for databases):

- `'2023-07-01T00:00:00Z'` matches values in UTC precisely.
- `'2023-07-01'` might shift to the local time zone (e.g., `'2023-07-01 00:00:00'` in a non-UTC time zone like `GMT+5:30`).
- This can exclude or include rows incorrectly based on the time zone difference.

💡**Why Filter Up to October 1st?**

The date range specified in the filter (`query_starttime >= '2023-07-01T00:00:00Z' AND query_starttime < '2023-10-01T00:00:00Z'`) ensures we include all queries executed from the start of July to the end of September. By using `< '2023-10-01T00:00:00Z'`, we capture all timestamps up to, but not including October 1st. This is standard practice to include the entire last day of September without accidentally including any part of October.

```
WITH employee_query_count_CTE AS (
SELECT employee_id, COUNT(DISTINCT(query_id))
FROM queries
WHERE query_starttime >= '2023-07-01T00:00:00Z' AND query_start
GROUP BY employee_id)

SELECT
COALESCE(eqc.count,0) as unique_queries,
COUNT(COALESCE(eqc.count,0)) as employee_count
FROM employees AS e
LEFT JOIN
employee_query_count_CTE as eqc
ON e.employee_id = eqc.employee_id
GROUP BY COALESCE(eqc.count,0)
ORDER BY COALESCE(eqc.count,0)
```

So, I wanted to optimize my code,

- **Query 1** uses multiple CTEs (`unique_queries_run_CTE`, `zero_queries_run_CTE`, `final_0_CTE`, `final_unique_CTE`) to build intermediate results, which increases complexity and can lead to additional overhead in processing and materializing these intermediate tables.
- **Query 2** uses only one CTE (`employee_query_count_CTE`), reducing the amount of work the database engine needs to do to process intermediate results.

- **Query 1** performs multiple joins and uses nested CTEs (`zero_queries_run_CTE` joins `employees` with `unique_queries_run_CTE`, then additional logic is applied on top of this).
- **Query 2** performs a single `LEFT JOIN` between `employees` and `employee_query_count_CTE`, which simplifies the execution plan.
- **Query 1** uses `WHERE count IS NULL` to explicitly filter employees who did not run any queries (in `final_0_CTE`), which adds a separate step.
- **Query 2** directly handles this scenario with `COALESCE(eqc.count, 0)` in the main query, eliminating the need for additional filtering logic.
- **uery 1** splits employees with `0` unique queries and those with non-zero queries into separate CTEs (`final_0_CTE` and `final_unique_CTE`), then combines them using `UNION ALL`.
- **Query 2** calculates both in a single query using `GROUP BY COALESCE(eqc.count, 0)`, which is more efficient as the grouping operation is