Analysis of SQL Queries

and

Database Design used to Optimize

and

Improve Query Execution Time

Data Ingestion

Optimization of the database started from data ingestion. In ingesting data into the database, the products table, regions table, and countries table were all parent tables to the sales table which contained their ids as foreign keys. For that simple fact, to ingest data into the sales table, we need to scan through the parent table for the product, region, and country names that are coming from the CSV file in order for the ids associated with the names to be placed inside the sales table. An example of the SELECTION QUERY to get the product_id from the products table is shown below:

```
SELECT product_id FROM products WHERE product_name = 'product name'.
```

Since the WHERE CLAUSE is case sensitive, to prevent errors during such query, the product_name, region_name, and country_name had to be converted to lower case characters during query execution. To speed things up a bit, each of the column had to be indexed. The type of indexed used here is the EXPRESSION INDEX since the function LOWER() is an expression. The query had to change to this:

```
SELECT product_id FROM products WHERE LOWER(product_name) =
LOWER('product_name').
```

With the product_name column having an expression index, the aforementioned query was optimized for execution.

Executing queries

First and foremost, I did an analysis of the questions that the result of the queries will answer to check for possible columns that some optimization techniques will be applied on.

- Monthly sales summaries with optimized query execution.
- Filtering by product, region, and date ranges, ensuring queries scale well with large datasets.
- A ranking of the top 5 products by revenue,

An analysis of the above questions gave me some level of intuition into what I am most likely to do during database design and query writing. First, looking at the three questions, it was clear that the product table will likely be the most hit from the queries. I made sure to apply index (btree) on the products name column and also apply the same index algorithm on the product_id column on the sales table. This will enable the database query optimizer to perform a quick scan if the query planner decides to use the btree index scan instead of the sequence scan.

In question 2, the term “Filtering” gave me an impression that the listed columns will most likely be in the WHERE CLAUSE, hence the need to index those columns for quick search. Filtering by date ranges made me to partition the table for easy searching of the specified date range. This gives the database a quick reference page to search for a particular date range instead of always running through the heap file, constantly searching through sequentially.

The old table without partitioning was temporarily left in the database to enable me to analyze and choose the right SQL query by comparing both the planning time and execution time in the old and new table to use in the application.

Caching

Before carrying out any query, I had to install the buffer cache extension to enable me check what’s in the current cache.

CREATE EXTENSION pg_bufferecache;

Then I went further to increase slightly the amount of dedicated memory by increasing the shared memory in the **postgresq.conf** file as increasing it too much may bloat the cache and slow the down the rate of getting query results due to too much time being wasted scanning the cache

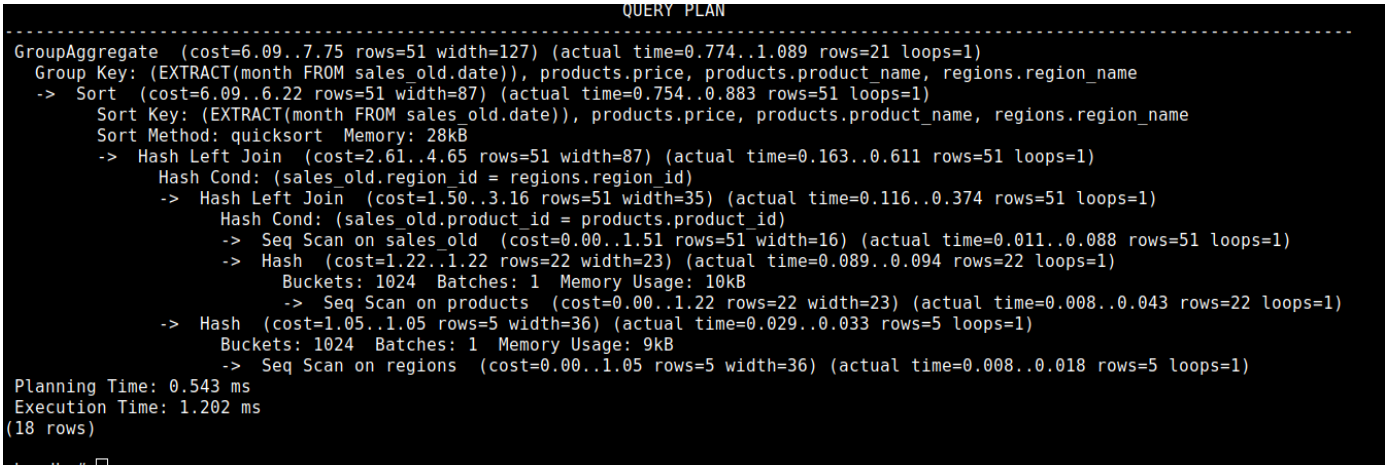
ANALYZING THE QUERIES

There are cases where I need to check for the performance of tables with partitions and the one without partition using the same query and tweaking it a little bit. To differentiate the tables ‘sales_old’ represents the unpartitioned table while table ‘sales’ represents the partitioned table.

Getting the monthly sales summaries

```
EXPLAIN ANALYZE SELECT EXTRACT (MONTH FROM date) AS month,
region_name, COUNT(product_id) AS quantity_sold, COUNT(product_id) *
products.price AS total_sales,
 20 product_name FROM sales_old LEFT JOIN products USING(product_id)
LEFT JOIN regions USING (region_id) GROUP BY (month, products.price,
product_name, region_name) ORDER BY month ASC;
```

The result of using postgresql EXPLAIN ANALYZE to analyze the above query on a non-partitioned table is shown below



From the diagram above, it shows that it took the query planner 0.543 milliseconds to plan for the query execution and 1.202 milliseconds to execute the actual query. This was carried out using LEFT JOIN after INNER JOIN gave a whooping 2.235 milliseconds. So I avoided the use of INNER JOIN since it takes more time to compare the existence of the same data on the two joining tables. The startup cost of execution of the above query during the group aggregation

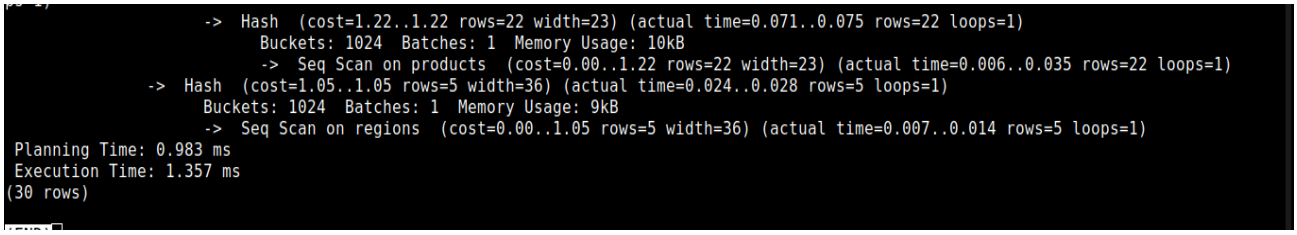
was 6.609 which means it indicates it took some time out to perform pre-execution tasks like scanning the indexes. The planner used a Hash scan in joining the common column elements of both left and right tables. The starting cost of sequentially scanning the products and regions table was 0.00 which means it had to take everything into consideration and just start immediately moving from one row to another. This could be a very big problem when working on large tables.

On the actual query which was implemented on the partitioned table used, the performance was hindered as a result of various partition tables that were created. Each of the tables had to be scanned sequentially to get the product_id. So, issuing the same command on the sales table with a partition wasn't very fast as expected.

```
EXPLAIN ANALYZE SELECT EXTRACT (MONTH FROM date) AS month,
region_name, COUNT(product_id) AS quantity_sold, COUNT(product_id) *
products.price AS total_sales,
 20 product_name FROM sales LEFT JOIN products USING(product_id) LEFT
JOIN regions USING (region_id) GROUP BY (month, products.price,
product_name, region_name) ORDER BY month ASC;
```

Gave this output

```
---------
 GroupAggregate  (cost=105.69..117.09 rows=351 width=127) (actual time=0.866..1.226 rows=21 loops=1)
   Group Key: (EXTRACT(month FROM sales.date)), products.price, products.product_name, regions.region_name
   ->  Sort  (cost=105.69..106.56 rows=351 width=87) (actual time=0.850..1.032 rows=51 loops=1)
         Sort Key: (EXTRACT(month FROM sales.date)), products.price, products.product_name, regions.region_name
         Sort Method: quicksort  Memory: 28kB
         ->  Hash Left Join  (cost=2.61..90.85 rows=351 width=87) (actual time=0.147..0.839 rows=51 loops=1)
               Hash Cond: (sales.region_id = regions.region_id)
               ->  Hash Join  (cost=1.50..87.91 rows=351 width=35) (actual time=0.107..0.588 rows=51 loops=1)
                     Hash Cond: (sales.product_id = products.product_id)
                     ->  Append  (cost=0.00..77.87 rows=3191 width=16) (actual time=0.022..0.358 rows=51 loops=1)
                           ->  Seq Scan on sales202401 sales_1  (cost=0.00..1.05 rows=5 width=16) (actual time=0.019..0.027 rows=5 loop
s=1)
                           ->  Seq Scan on sales202402 sales_2  (cost=0.00..1.07 rows=7 width=16) (actual time=0.007..0.017 rows=7 loop
s=1)
                           ->  Seq Scan on sales202403 sales_3  (cost=0.00..1.03 rows=3 width=16) (actual time=0.006..0.011 rows=3 loop
s=1)
                           ->  Seq Scan on sales202404 sales_4  (cost=0.00..1.03 rows=3 width=16) (actual time=0.007..0.013 rows=3 loop
s=1)
                           ->  Seq Scan on sales202405 sales_5  (cost=0.00..1.10 rows=10 width=16) (actual time=0.007..0.022 rows=10 lo
ops=1)
                           ->  Seq Scan on sales202406 sales_6  (cost=0.00..25.70 rows=1570 width=16) (actual time=0.003..0.005 rows=0
loops=1)
                           ->  Seq Scan on sales202407 sales_7  (cost=0.00..1.03 rows=3 width=16) (actual time=0.005..0.010 rows=3 loop
s=1)
                           ->  Seq Scan on sales202408 sales_8  (cost=0.00..1.07 rows=7 width=16) (actual time=0.007..0.018 rows=7 loop
s=1)
                           ->  Seq Scan on sales202409 sales_9  (cost=0.00..25.70 rows=1570 width=16) (actual time=0.003..0.004 rows=0
loops=1)
                           ->  Seq Scan on sales202410 sales_10  (cost=0.00..1.04 rows=4 width=16) (actual time=0.006..0.013 rows=4 loo
ps=1)
```

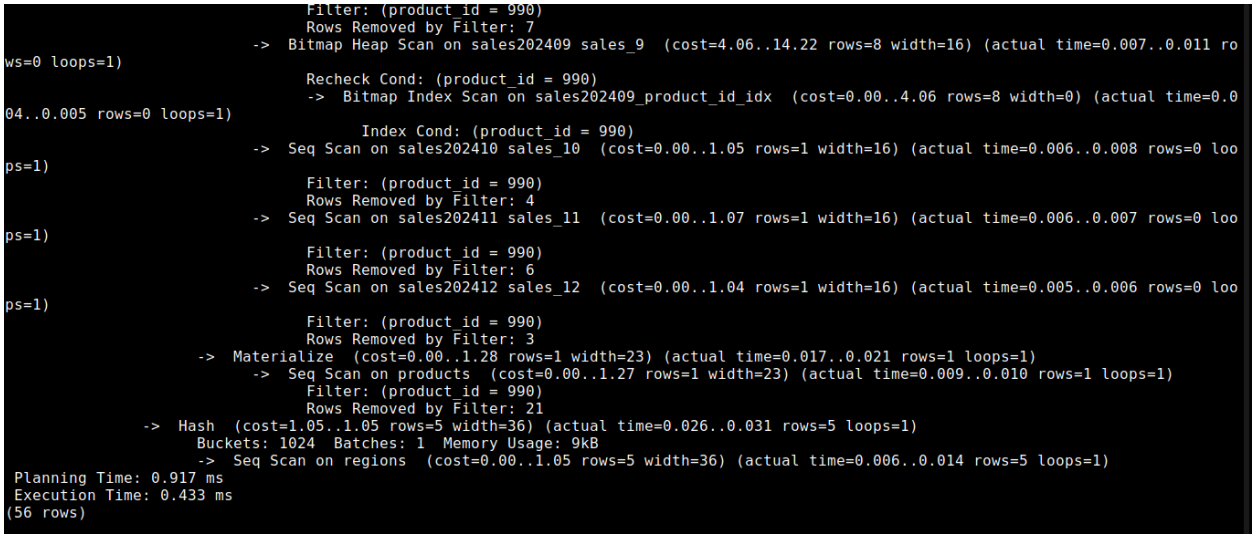


As seen in the two diagrams above, a sequence scan was performed on each and every partition table to the right data to join the contents from both tables. This greatly affected its performance even with the product_id indexed. Subsequent issuance of the query reduced the execution time as a result of the database caching. Altering the index to hash index never really improved the performance since we are probably not performing a WHERE CLAUSE that precisely requires to find one product or region.

Getting the monthly sales summaries based on a criterial

1. Filtering by product

Adding a WHERE CLAUSE to the previous query after the JOIN helped in filtering out the unwanted product ids. This time, because it involved finding a particular set of values, I used a Hash Index scan on the product id as the hashing algorithm performs better in terms of locating specific values. There was quite an improvement on the query execution time. The output is shown below:



The Execution time became quite encouraging this time with a time of 0.433 milliseconds and also the planning time improved with a 0.917 milliseconds planning time.. Looking at the total rows it removed by filtering, it filtered out a few numbers of tables compared to the unpartitioned table. It performed a hash scan when estimating the cost hence the starting value of 1.05 and an ending value of 1.05 which is quite good. It didn't have to start from a cost of 0.00 which requires starting from the beginning of the list to the last value which may slow down the process as it will end with a much larger cost. But on each partitioned table, it performed a sequence scan and also on the regions table which is perfect since it's a small table and will remain constant as it contains the 5 major continents in the world.

2. **Filtering by date range**

The partitioning of the sales table had a very positive effect on this query. During the process of query planning, since the column in which the partition took place is placed on the WHERE CLAUSE, instead of scanning through the entire table, postgres database goes straight to the partitioned table that falls within the range of what is being searched for in the WHERE CLAUSE. This increases the performance of the table and overall query execution. Below is a screenshot of the query execution result in filtering the monthly sales summaries by date range. The first time running the query using the date in the WHERE CLAUSE resulted in an execution time of 0.576 milliseconds as shown below.

```
        Sort Method: quicksort  Memory: 25kB
        ->  Hash Left Join  (cost=2.61..34.97 rows=10 width=87) (actual time=0.171..0.343 rows=10 loops=1)
              Hash Cond: (sales.region_id = regions.region_id)
              ->  Hash Left Join  (cost=1.50..33.80 rows=10 width=35) (actual time=0.125..0.252 rows=10 loops=1)
                    Hash Cond: (sales.product_id = products.product_id)
                    ->  Append  (cost=0.00..32.28 rows=10 width=16) (actual time=0.020..0.104 rows=10 loops=1)
                          ->  Seq Scan on sales202405 sales_1  (cost=0.00..1.15 rows=1 width=16) (actual time=0.016..0.036 rows=10 loo
ps=1)
                                Filter: ((date >= '2024-05-01 00:00:00+01'::timestamp with time zone) AND (date <= '2024-07-03 00:00:0
0+01'::timestamp with time zone))
                          ->  Bitmap Heap Scan on sales202406 sales_2  (cost=19.85..30.03 rows=8 width=16) (actual time=0.012..0.017 r
ows=0 loops=1)
                                Recheck Cond: ((date >= '2024-05-01 00:00:00+01'::timestamp with time zone) AND (date <= '2024-07-03 0
0:00:00+01'::timestamp with time zone))
                                ->  Bitmap Index Scan on sales202406_id_date_key  (cost=0.00..19.85 rows=8 width=0) (actual time=0.007
..0.008 rows=0 loops=1)
                                      Index Cond: ((date >= '2024-05-01 00:00:00+01'::timestamp with time zone) AND (date <= '2024-07-
03 00:00:00+01'::timestamp with time zone))
                          ->  Seq Scan on sales202407 sales_3  (cost=0.00..1.04 rows=1 width=16) (actual time=0.009..0.010 rows=0 loop
s=1)
                                Filter: ((date >= '2024-05-01 00:00:00+01'::timestamp with time zone) AND (date <= '2024-07-03 00:00:0
0+01'::timestamp with time zone))
                                Rows Removed by Filter: 3
                    ->  Hash  (cost=1.22..1.22 rows=22 width=23) (actual time=0.090..0.095 rows=22 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 10kB
                          ->  Seq Scan on products  (cost=0.00..1.22 rows=22 width=23) (actual time=0.008..0.044 rows=22 loops=1)
              ->  Hash  (cost=1.05..1.05 rows=5 width=36) (actual time=0.028..0.032 rows=5 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    ->  Seq Scan on regions  (cost=0.00..1.05 rows=5 width=36) (actual time=0.007..0.016 rows=5 loops=1)
 Planning Time: 0.773 ms
 Execution Time: 0.576 ms
```

The type of partitioning used here is the range partitioning since the query requires fetching of data within a specific date range at a time. The query planning time before execution was 0.773 ms. Both the planning and execution time was a significant improvement from the previous WHERE CLAUSES which required the product_id. Running the query again leads to slightly better performance metrics which in a huge dataset will definitely have a significant effect. Below is a screenshot of rerunning the query. This time, there was an improvement as the initial query was cached. The planner only had to go to the cached memory to execute the query again, instead of the heap file.

```
              Hash Cond: (sales.region_id = regions.region_id)
              ->  Hash Left Join  (cost=1.50..33.80 rows=10 width=35) (actual time=0.094..0.195 rows=10 loops=1)
                    Hash Cond: (sales.product_id = products.product_id)
                    ->  Append  (cost=0.00..32.28 rows=10 width=16) (actual time=0.014..0.081 rows=10 loops=1)
                          ->  Seq Scan on sales202405 sales_1  (cost=0.00..1.15 rows=1 width=16) (actual time=0.011..0.026 rows=10 loo
ps=1)
                                Filter: ((date >= '2024-05-01 00:00:00+01'::timestamp with time zone) AND (date <= '2024-07-03 00:00:0
0+01'::timestamp with time zone))
                          ->  Bitmap Heap Scan on sales202406 sales_2  (cost=19.85..30.03 rows=8 width=16) (actual time=0.009..0.012 r
ows=0 loops=1)
                                Recheck Cond: ((date >= '2024-05-01 00:00:00+01'::timestamp with time zone) AND (date <= '2024-07-03 0
0:00:00+01'::timestamp with time zone))
                                ->  Bitmap Index Scan on sales202406_id_date_key  (cost=0.00..19.85 rows=8 width=0) (actual time=0.005
..0.007 rows=0 loops=1)
                                      Index Cond: ((date >= '2024-05-01 00:00:00+01'::timestamp with time zone) AND (date <= '2024-07-
03 00:00:00+01'::timestamp with time zone))
                          ->  Seq Scan on sales202407 sales_3  (cost=0.00..1.04 rows=1 width=16) (actual time=0.007..0.008 rows=0 loop
s=1)
                                Filter: ((date >= '2024-05-01 00:00:00+01'::timestamp with time zone) AND (date <= '2024-07-03 00:00:0
0+01'::timestamp with time zone))
                                Rows Removed by Filter: 3
                    ->  Hash  (cost=1.22..1.22 rows=22 width=23) (actual time=0.069..0.073 rows=22 loops=1)
                          Buckets: 1024  Batches: 1  Memory Usage: 10kB
                          ->  Seq Scan on products  (cost=0.00..1.22 rows=22 width=23) (actual time=0.005..0.035 rows=22 loops=1)
              ->  Hash  (cost=1.05..1.05 rows=5 width=36) (actual time=0.021..0.024 rows=5 loops=1)
                    Buckets: 1024  Batches: 1  Memory Usage: 9kB
                    ->  Seq Scan on regions  (cost=0.00..1.05 rows=5 width=36) (actual time=0.005..0.012 rows=5 loops=1)
 Planning Time: 0.521 ms
 Execution Time: 0.430 ms
(27 rows)
```

This time planning and execution time became 0.521 ms and 0.430 ms respectively. Again, the number of rows that was filtered by the query planner was just 3 and it did use sequence scan

on a few rows (mostly on the partitioned tables), It rather opted for the more efficient Bitmap Heap Scan.

Checking the query execution time in seconds using Python's unittesting module was also used. This was used to analyze various JOINS and table types (partitioned and unpartitioned). In summary, the output of the two execution times are shown below.

```
(.venv) enoma@enels-pc:~/optimized_data_aggregation$ python query_per*.py
Monthly Sales Summaries Execution time: 0.0077
FMonthly Summaries by Range Execution time: 0.0079seconds
FMonthly Summaries by Product Execution time: 0.0072 seconds
FMonthly Summaries by Region Execution time: 0.0158seconds
FMonthly Summaries by Top 5 Revenue Products Execution time: 0.0080seconds
F
```

With the partitioned table with LEFT JOINED used as the last one which was retrieving the top 5 revenue generated products. The execution time is 0.0080 seconds.

```
Monthly Sales Summaries Execution time: 0.0071
FMonthly Summaries by Range Execution time: 0.0061seconds
FMonthly Summaries by Product Execution time: 0.0077 seconds
FMonthly Summaries by Region Execution time: 0.0073seconds
FMonthly Summaries by Top 5 Revenue Products Execution time: 0.0063seconds
F
```

While the one with INNER JOIN had an execution time of 0.0063 seconds. Even the other query result shows that the query has been optimized slightly but that slight increment will have a huge effect on performance when dealing with a large dataset.

OPTIMIZATION TECHNIQUES THAT WAS IMPLEMENTED BOTH ON THE DATABASE AND ON QUERY DESIGN

1.) I enabled pruning using the command:

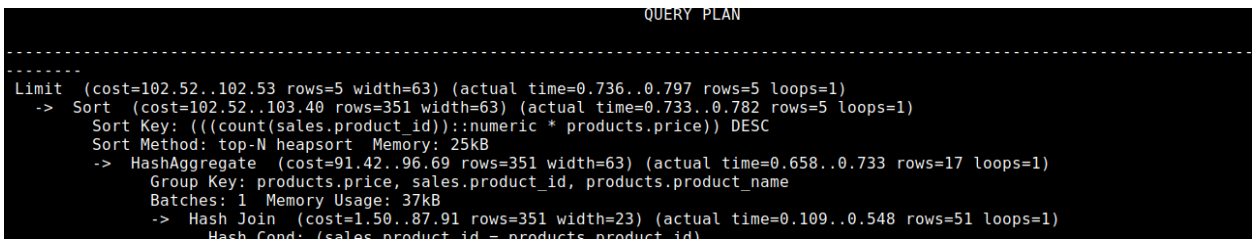
```
SET enable_partition_prunning = on;
```

Even though that is the default setting, I just had to do it to make sure it is actually on.

2.) Secondly, I avoided the use of subqueries as it can slow down the fetching time due to too many SELECT statements. Each SELECT statement in a subquery has its own execution time and summing them up will become large. I channelled my thought process towards using the more effective JOINS. The only problem I encountered was making the right choice between LEFT, RIGHT, AND INNER JOIN. I had to perform different queries using all three (with and without caching) to determine the best to implement on a particular query.
For example, during the process of writing the query to get the top five revenue generated products which is:

```
SELECT product_id, product_name, COUNT(product_id) AS
quantity_sold, COUNT(product_id) * price AS total_revenue
FROM sales INNER JOIN products USING(product_id)
GROUP BY (price,product_id,product_name)
ORDER BY total_revenue DESC LIMIT 5;
```

I used both INNER JOIN and LEFT JOIN. The result of the INNER JOIN is showing the overall cost of operation is shown below:



The sort cost and limit is 102.52 and the execution time is 0.910 as shown below

```
                         ->  Seq Scan on sales202401 sales_1  (cost=0.00..1.05 rows=5 width=4) (actual time=0.018..0.026 rows=5 loops
=1)
                         ->  Seq Scan on sales202402 sales_2  (cost=0.00..1.07 rows=7 width=4) (actual time=0.007..0.017 rows=7 loops
=1)
                         ->  Seq Scan on sales202403 sales_3  (cost=0.00..1.03 rows=3 width=4) (actual time=0.007..0.013 rows=3 loops
=1)
                         ->  Seq Scan on sales202404 sales_4  (cost=0.00..1.03 rows=3 width=4) (actual time=0.006..0.011 rows=3 loops
=1)
                         ->  Seq Scan on sales202405 sales_5  (cost=0.00..1.10 rows=10 width=4) (actual time=0.007..0.020 rows=10 loo
ps=1)
                         ->  Seq Scan on sales202406 sales_6  (cost=0.00..25.70 rows=1570 width=4) (actual time=0.004..0.005 rows=0 l
oops=1)
                         ->  Seq Scan on sales202407 sales_7  (cost=0.00..1.03 rows=3 width=4) (actual time=0.006..0.012 rows=3 loops
=1)
                         ->  Seq Scan on sales202408 sales_8  (cost=0.00..1.07 rows=7 width=4) (actual time=0.007..0.018 rows=7 loops
=1)
                         ->  Seq Scan on sales202409 sales_9  (cost=0.00..25.70 rows=1570 width=4) (actual time=0.004..0.005 rows=0 l
oops=1)
                         ->  Seq Scan on sales202410 sales_10  (cost=0.00..1.04 rows=4 width=4) (actual time=0.005..0.012 rows=4 loop
s=1)
                         ->  Seq Scan on sales202411 sales_11  (cost=0.00..1.06 rows=6 width=4) (actual time=0.007..0.017 rows=6 loop
s=1)
                         ->  Seq Scan on sales202412 sales_12  (cost=0.00..1.03 rows=3 width=4) (actual time=0.006..0.011 rows=3 loop
s=1)
                   ->  Hash  (cost=1.22..1.22 rows=22 width=23) (actual time=0.076..0.080 rows=22 loops=1)
                         Buckets: 1024  Batches: 1  Memory Usage: 10kB
                         ->  Seq Scan on products  (cost=0.00..1.22 rows=22 width=23) (actual time=0.008..0.038 rows=22 loops=1)
 Planning Time: 0.680 ms
 Execution Time: 0.910 ms
(27 rows)

(END)
```

The same query, on executing it on LEFT JOIN, and the result is shown below:

```
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------
--------
 Limit  (cost=220.69..220.70 rows=5 width=63) (actual time=0.778..0.840 rows=5 loops=1)
   ->  Sort  (cost=220.69..228.67 rows=3191 width=63) (actual time=0.774..0.822 rows=5 loops=1)
         Sort Key: (((count(sales.product_id))::numeric * products.price)) DESC
         Sort Method: top-N heapsort  Memory: 25kB
         ->  HashAggregate  (cost=119.82..167.69 rows=3191 width=63) (actual time=0.663..0.771 rows=17 loops=1)
               Group Key: products.price, sales.product_id, products.product_name
               Batches: 1  Memory Usage: 121kB
               ->  Hash Left Join  (cost=1.50..87.91 rows=3191 width=23) (actual time=0.149..0.588 rows=51 loops=1)
                     Hash Cond: (sales.product_id = products.product_id)
                     ->  Append  (cost=0.00..77.87 rows=3191 width=4) (actual time=0.060..0.356 rows=51 loops=1)
                           ->  Seq Scan on sales202401 sales_1  (cost=0.00..1.05 rows=5 width=4) (actual time=0.057..0.066 rows=5 loops
```

The cost is more than twice the cost of the INNER JOIN and the execution time is shown below:

```
                           ->  Seq Scan on sales202410 sales_10  (cost=0.00..1.04 rows=4 width=4) (actual time=0.009..0.019 rows=4 loop
s=1)
                           ->  Seq Scan on sales202411 sales_11  (cost=0.00..1.06 rows=6 width=4) (actual time=0.010..0.026 rows=6 loop
s=1)
                           ->  Seq Scan on sales202412 sales_12  (cost=0.00..1.03 rows=3 width=4) (actual time=0.009..0.018 rows=3 loop
s=1)
                     ->  Hash  (cost=1.22..1.22 rows=22 width=23) (actual time=0.118..0.124 rows=22 loops=1)
                           Buckets: 1024  Batches: 1  Memory Usage: 10kB
                           ->  Seq Scan on products  (cost=0.00..1.22 rows=22 width=23) (actual time=0.012..0.062 rows=22 loops=1)
 Planning Time: 0.995 ms
 Execution Time: 1.529 ms
(27 rows)

(END)
```

The planning and execution time which is 0.995 ms and 1.529 ms respectively are poor compared to the INNER JOIN which will definitely have a bad effect on large datasets.

3.) My analysis of the information to spoof out of the database served as a guideline towards implementing various table enhancement features such as index, partitioning, and caching. From my observation, most of the question required queries that needed to perform some form of search on the product_id column. That gave me the impetus that I will need to index that column since indexing prevents the database from making a full sequential scan on the entire dataset from the heap file as that reduces the stress on the resources and speeds up the search process. The type of indexing to use, depends on the type of queries.

4.) I made sure that column used in WHERE CLAUSES constraint was indexed using HASH INDEX as it will greatly enhance the database search capabilities on large datasets. Same goes for JOINS. Columns that are involved in JOINS are always indexed, most times using Btree which is the default.

5.) An expression index was effected on columns that require some form of expression. Like during the process of comparing two string data, using a lower case makes the comparison independent of the type of cases that was used in storing it or used in searching for it. Using the expression index in columns that require the LOWER() function enhances the performance of the database by preventing it from always performing the expression for every hit.

6.) For filtering out the monthly sales summaries based on the data range, I made sure that the sales table was partitioned based on the date column to narrow down the search for the data to a particular partition table instead of searching the entire sales table, which greatly enhanced the database performance.

7.) Lastly, implementation of cashing to reduce query execution time query especially during reruns was done by slightly increasing the physical memory allocated to the shared from 128MB to 256MB in the *postgres.config* file.