# Query Optimization

Since I first tested the performance of the original query and then applied various optimization strategies, the following text summarizes all three requests.

I created a sample dataset with data on 10 products and 35 product orders in the Orders and Product tables. To track changes in performance as well as the execution time of the query, I used the EXPLAIN ANALYZE function and obtained the following results for the original query:

A screenshot of a computer

Description automatically generated

***Note:*** *The execution time varies between 0.22 ms and 0.44 ms.*

Based on the results, I identified that it is necessary to optimize the join between the Orders and Products tables, as well as the aggregation of data by product category, since these are the most “expensive” operations in the query.

* **Adding an Index**

The first thing I did to optimize the join of the two tables was to cluster the index which PostgreSQL created by default for the Product\_Id primary key in the Products table, and to add a non-clustered index on the Product\_Id column in the Orders table.

* CLUSTER optimization.Product USING product\_pkey;
* CREATE INDEX idx\_orders\_product\_id ON optimization.Orders(product\_id);

This did not significantly change the performance, as it is a small dataset, and therefore PostgreSQL favors a sequential load of the entire table over indexing. However, considering that the Orders table is expected to have millions of rows and that the join of these two tables is a common operation, the index is certainly something that can improve performance in the long run.

* **Common Table Expression (CTE)**

I then decided to break the query into smaller parts to first perform aggregation by product in the Orders table and then join the aggregated table with the dimensional table and execute aggregation by product category using a CTE (Common Table Expression).

The performance of the query improved, and the results are as follows:

A screenshot of a computer

Description automatically generated

I also tested the performance of the subquery, which was slightly worse compared to the CTE. However, I chose the CTE because it is more readable and understandable than the subquery (which can be significant if additional subqueries are needed) and because a CTE can be referenced multiple times, reducing code redundancy.

* **Data Filtering**

Additionally, since the original query loads the entire Orders table and there is an Order\_Date column, I decided to include a condition to consider data for the year 2024, as I assume that the Orders table may contain data from previous years that may not be relevant for the current analysis.

The final performance after adding the date condition is:

A screenshot of a computer

Description automatically generated