University of Trieste
Master in Data Science and Scientific Computing
Curriculum of Artificial Intelligence and Machine Learning

Course of Data Management for Big Data

Benchmark for Decision Support:

TPC-H Implementation and Query Optimization

Author:

Bajjouk Wafaa

Department of Mathematics, Computer Science and Geosciences

May 2024

Contents

Part I
Introduction

The present work focuses on the implementation and optimization of the TPC-H benchmark using PostgreSQL for decision support systems. The TPC-H benchmark is a standard used to evaluate the performance of database systems through a suite of business-oriented ad-hoc queries and concurrent data modifications, designed to simulate decision support systems. T**he primary objective of this project was to optimize the performance of three critical queries—time-based, type-based, and nation and region-based aggregations—by leveraging advanced indexing techniques and materialized views.**

The project commenced with the creation of a detailed database schema as per the TPC-H specifications, followed by data population using the TPC-H `dbgen` tool at a **scale factor of 10**, resulting in a 20GB dataset. An initial statistical analysis was conducted to understand the data distribution and characteristics, laying the groundwork for subsequent optimization.

Three essential queries were implemented to measure initial performance: aggregations by time (month, quarter, year), type of items, and geographical aspects (nation and region). Initial query costs and execution times were recorded, revealing substantial opportunities for optimization.

The optimization phase involved the strategic creation of indexes on key columns to reduce search times and improve join operations. Materialized views were designed to store precomputed query results, significantly enhancing query performance. Space constraints for indexes and materialized views were carefully verified to ensure compliance with specified limits.

Post-optimization analysis demonstrated dramatic improvements in query performance. **The time-based aggregation query execution time reduced from over 27 seconds to just 116 milliseconds, the type-based aggregation query from nearly 49 seconds to 49 milliseconds, and the nation and region-based aggregation query from over 56 seconds to 61 milliseconds.** These results highlight the effectiveness of the optimization strategies employed.

Part II
Database and Schema Creation

The first step in this project involved creating the database schema according to the TPC-H specifications. This schema includes several tables, each with specific structures, constraints, and data types. The process began with establishing the database and successfully connecting to it, followed by the detailed creation of tables such as REGION, NATION, PART, SUPPLIER, PARTSUPP, CUSTOMER, ORDERS, and LINEITEM. Each table was meticulously defined with primary and foreign key constraints, along with integrity checks to ensure data consistency. Screenshots and SQL scripts were used to illustrate the successful creation and population of these tables.

1 Database and Schema Creation

The first step in the project was to create the database schema as specified in the TPC-H documentation. The logical schema includes several tables with specified structures , constraints declarations and data types.

1.1 Database Creation and Connection

Here are the details about the database creation along with a screenshot of the connection.

- **Database Name:** TPCH-Project
- **Database Type:** PostgreSQL
- **Owner:** wafaabajjouk
- **Port:** 5432

To connect to the database, I used the following command:

```
psql -U wafaabajjouk -d TPCH-Project
```

The screenshot below shows the successful connection to the database:

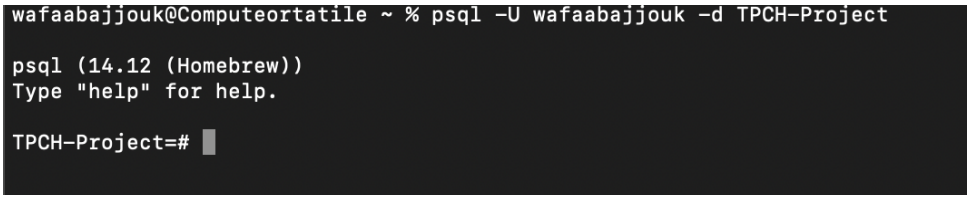


Figure 1: Connecting to TPCH-Project database

1.2 SQL Definition of the Tables and Constraints

The following section details the creation of each table in the PostgreSQL database, including integrity and check constraints. Screenshots of the table creation process are provided for each table. I followed the exact order of creating the tables and also during populating them to avoid integrity constraint errors.

The complete SQL script can be found in the folder associated with this report under the folder `SQL SCRIPTS`, in the file named `schema.sql`.

1.2.1 Creating the REGION Table

The REGION table is created first because it serves as a reference for the NATION table. This table includes a primary key constraint on `R_REGIONKEY` and a check constraint to ensure the key is non-negative.

```
CREATE TABLE REGION (
    R_REGIONKEY INTEGER NOT NULL PRIMARY KEY,
    R_NAME VARCHAR(25) NOT NULL,
    R_COMMENT VARCHAR(152),
    CONSTRAINT check_pk_region CHECK (R_REGIONKEY >= 0)
);
```

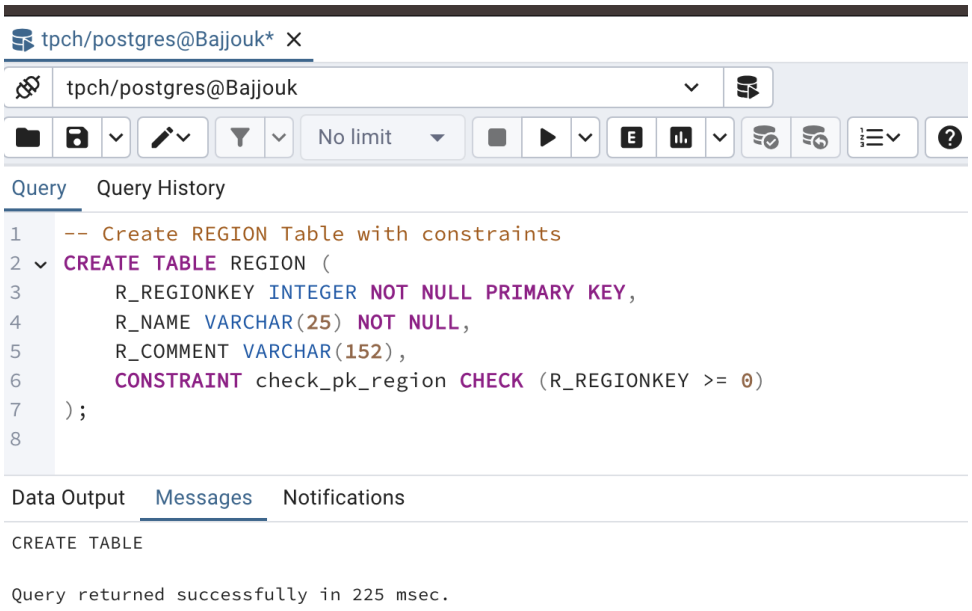


Figure 2: Creation of REGION Table

1.2.2 Creating the NATION Table

The NATION table includes a primary key constraint on `N_NATIONKEY`, a foreign key constraint referencing `REGION.R_REGIONKEY`, and a check constraint to ensure the key is non-negative.

```
CREATE TABLE NATION (
    N_NATIONKEY INTEGER NOT NULL PRIMARY KEY,
    N_NAME VARCHAR(25) NOT NULL,
    N_REGIONKEY INTEGER NOT NULL,
    N_COMMENT VARCHAR(152),
    FOREIGN KEY (N_REGIONKEY) REFERENCES REGION(R_REGIONKEY),
    CONSTRAINT check_pk_nation CHECK (N_NATIONKEY >= 0)
);
```

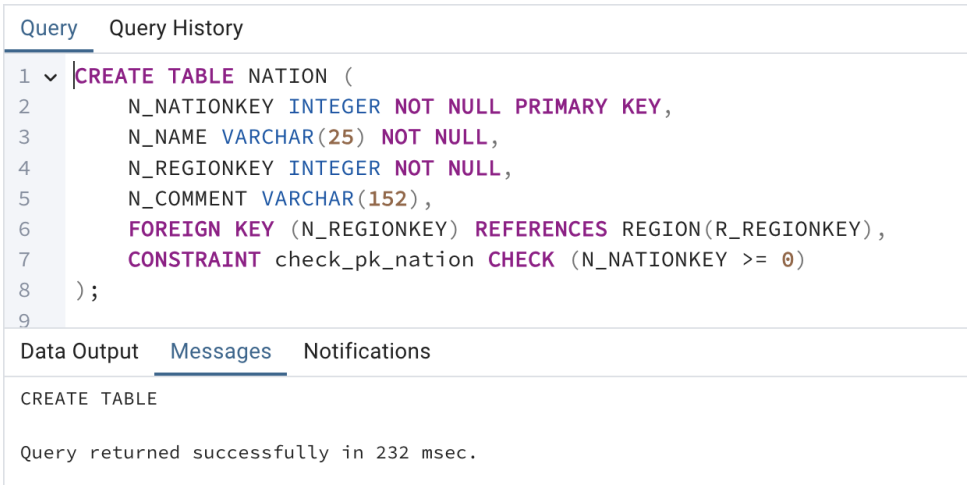


Figure 3: Creation of NATION Table

1.2.3 Creating the PART Table

The PART table includes a primary key (PK) constraint on `P_PARTKEY` and check constraints to ensure `P_SIZE` and `P_RETAILPRICE` are non-negative.

```
CREATE TABLE PART (
    P_PARTKEY INTEGER NOT NULL PRIMARY KEY,
    P_NAME VARCHAR(55),
    P_MFGR CHAR(25),
    P_BRAND CHAR(10),
    P_TYPE VARCHAR(25),
    P_SIZE INTEGER NOT NULL CHECK (P_SIZE >= 0),
    P_CONTAINER CHAR(10),
    P_RETAILPRICE DECIMAL NOT NULL CHECK (P_RETAILPRICE >= 0),
    P_COMMENT VARCHAR(23),
    CONSTRAINT check_pk_part CHECK (P_PARTKEY >= 0)
);
```

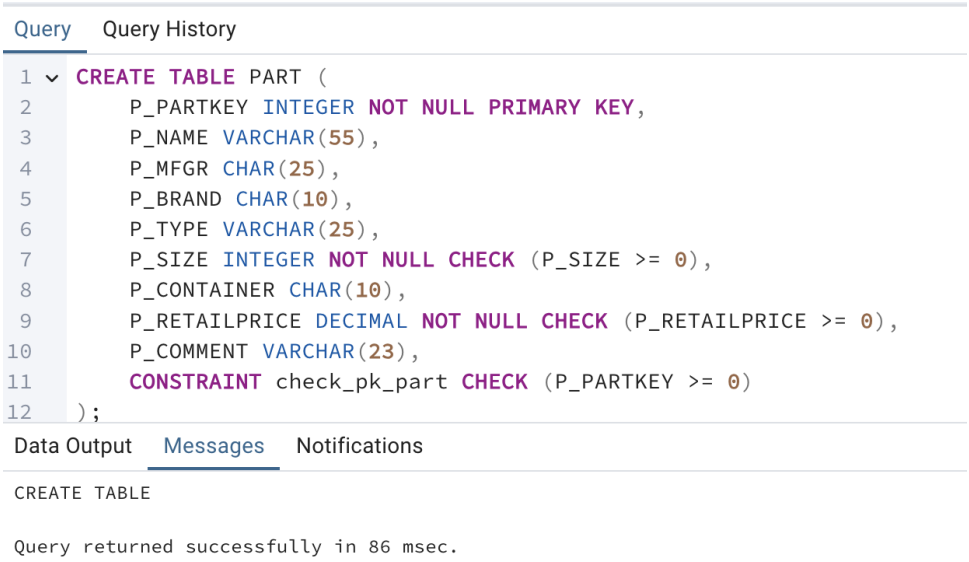


Figure 4: Creation of PART Table

1.2.4 Creating the SUPPLIER Table

The SUPPLIER table includes a primary key (PK) constraint on `S_SUPPKEY`, a foreign key (FK) constraint referencing `NATION.N_NATIONKEY` , and a check constraint to ensure the key is non-negative.

```
CREATE TABLE SUPPLIER (
    S_SUPPKEY INTEGER NOT NULL PRIMARY KEY,
    S_NAME CHAR(25),
    S_ADDRESS VARCHAR(40),
    S_NATIONKEY INTEGER NOT NULL,
    S_PHONE CHAR(15),
    S_ACCTBAL DECIMAL,
    S_COMMENT VARCHAR(101),
    FOREIGN KEY (S_NATIONKEY) REFERENCES NATION(N_NATIONKEY),
    CONSTRAINT check_pk_supplier CHECK (  S_SUPPKEY >= 0)

);
```

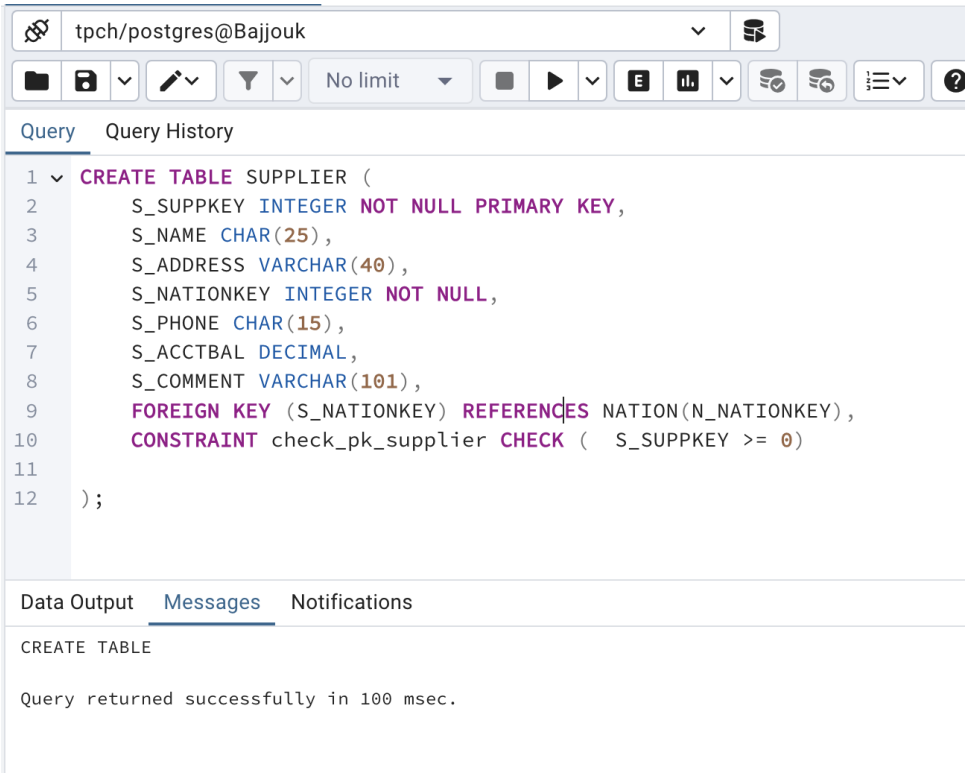


Figure 5: Creation of SUPPLIER Table

1.2.5 Creating the PARTSUPP Table

The PARTSUPP table includes a composite primary key (PK) constraint on `PS_PARTKEY` and `PS_SUPPKEY`, foreign key (FK) constraints referencing `PART.P_PARTKEY` and `SUPPLIER.S_SUPPKEY`, and check constraints to ensure `PS_AVAILQTY` and `PS_SUPPLYCOST` are non-negative.

```
CREATE TABLE PARTSUPP (
    PS_PARTKEY INTEGER NOT NULL,
    PS_SUPPKEY INTEGER NOT NULL,
    PS_AVAILQTY INTEGER NOT NULL CHECK (PS_AVAILQTY >= 0),
    PS_SUPPLYCOST DECIMAL NOT NULL CHECK (PS_SUPPLYCOST >= 0),
    PS_COMMENT VARCHAR(199),
    PRIMARY KEY (PS_PARTKEY, PS_SUPPKEY),
    FOREIGN KEY (PS_PARTKEY) REFERENCES PART(P_PARTKEY),
    FOREIGN KEY (PS_SUPPKEY) REFERENCES SUPPLIER(S_SUPPKEY),
    CONSTRAINT check_pk_ps_partkey CHECK (PS_PARTKEY >= 0),
    CONSTRAINT check_pk_ps_suppkey CHECK (PS_SUPPKEY >= 0)
);
```

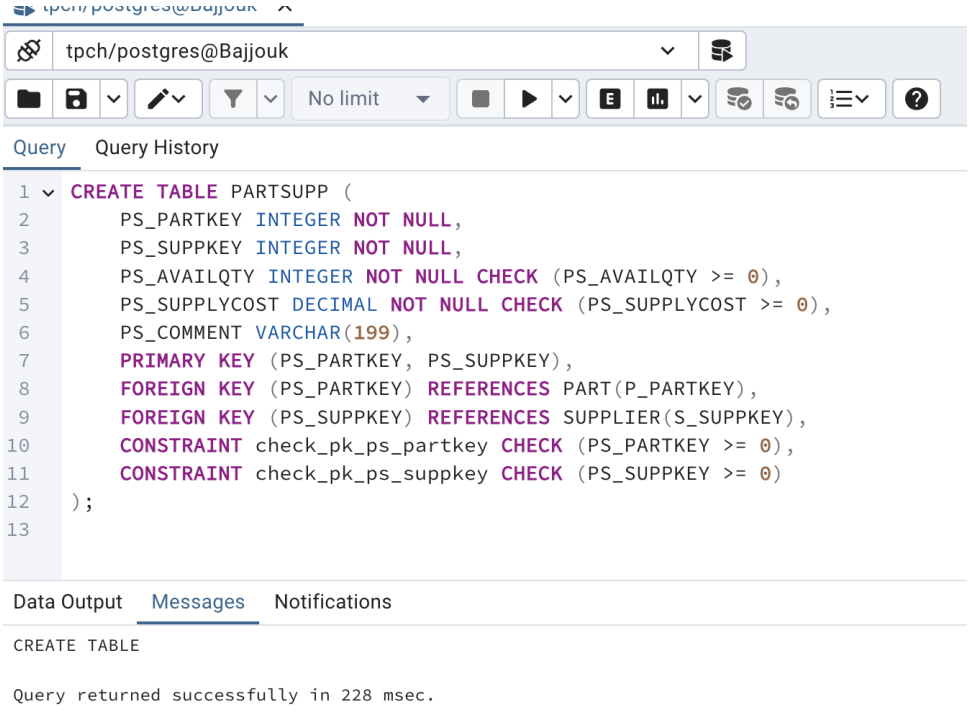


Figure 6: Creation of PARTSUPP Table

1.2.6 Creating the CUSTOMER Table

The CUSTOMER table includes a primary key (PK) constraint on `C_CUSTKEY`, a foreign key (FK) constraint referencing `NATION.N_NATIONKEY`, and a check constraint to ensure the key is non-negative.

```
CREATE TABLE CUSTOMER (
    C_CUSTKEY INTEGER NOT NULL PRIMARY KEY,
    C_NAME VARCHAR(25),
    C_ADDRESS VARCHAR(40),
    C_NATIONKEY INTEGER NOT NULL,
    C_PHONE CHAR(15),
    C_ACCTBAL DECIMAL,
    C_MKTSEGMENT CHAR(10),
    C_COMMENT VARCHAR(117),
    FOREIGN KEY (C_NATIONKEY) REFERENCES NATION(N_NATIONKEY),
    CONSTRAINT check_positive_custkey CHECK (C_CUSTKEY >= 0)  );
```

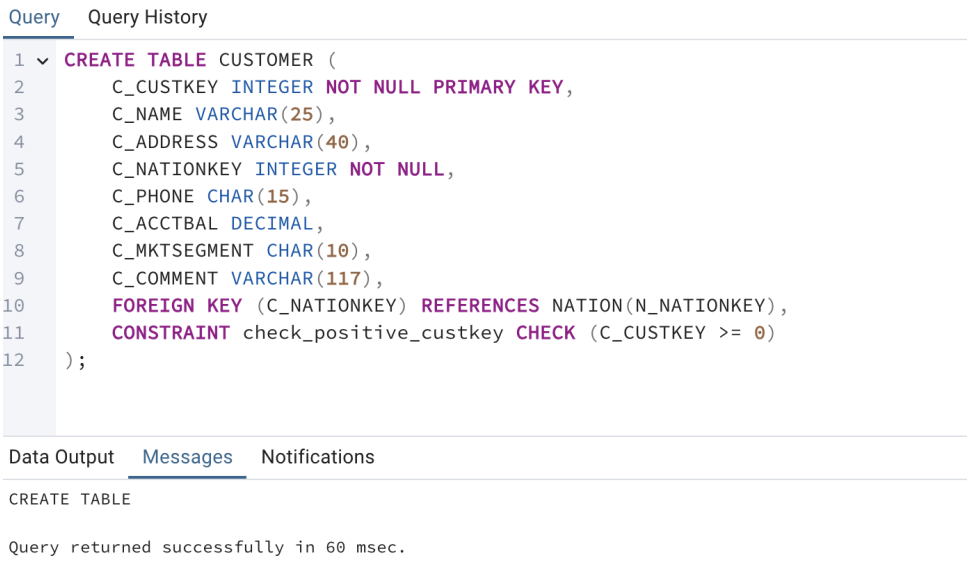


Figure 7: Creation of CUSTOMER Table

1.2.7 Creating the ORDERS Table

The ORDERS table includes a primary key (PK) constraint on `O_ORDERKEY`, a foreign key (FK) constraint referencing `CUSTOMER.C_CUSTKEY`, and a check constraint to ensure `O_TOTALPRICE` is non-negative.

```
CREATE TABLE ORDERS (
    O_ORDERKEY INTEGER NOT NULL PRIMARY KEY,
    O_CUSTKEY INTEGER NOT NULL,
    O_ORDERSTATUS CHAR(1),
    O_TOTALPRICE DECIMAL NOT NULL CHECK (O_TOTALPRICE >= 0),
    O_ORDERDATE DATE,
    O_ORDERPRIORITY CHAR(15),
    O_CLERK CHAR(15),
    O_SHIPPRIORITY INTEGER,
    O_COMMENT VARCHAR(79),
    FOREIGN KEY (O_CUSTKEY) REFERENCES CUSTOMER(C_CUSTKEY),
    CONSTRAINT check_positive_orderkey CHECK (O_ORDERKEY >= 0)
);
```

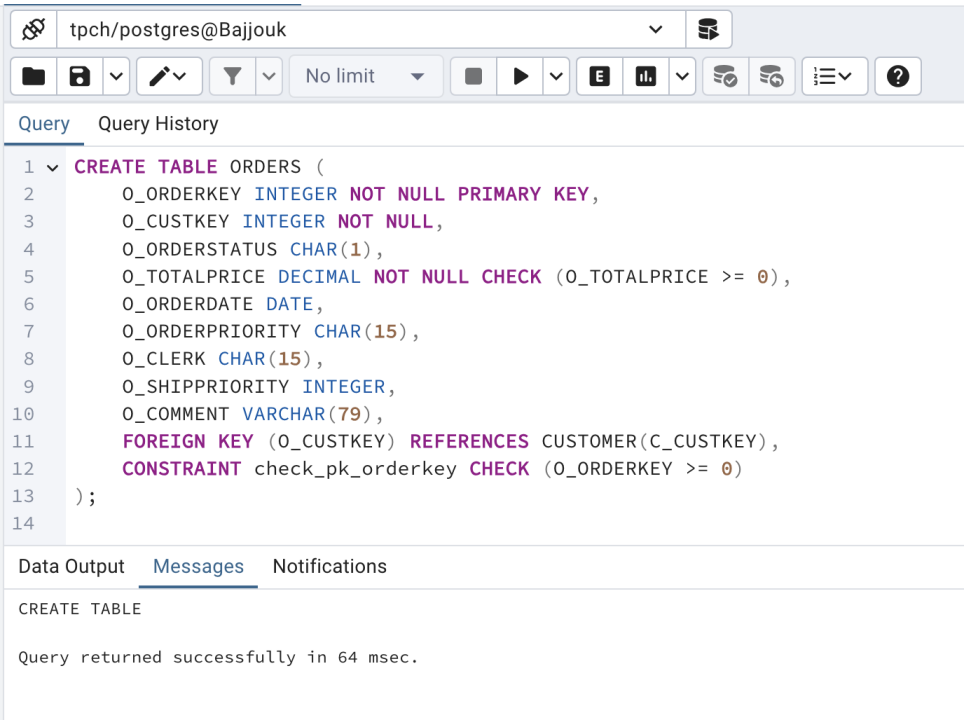


Figure 8: Creation of ORDERS Table

1.2.8 Creating the LINEITEM Table

The LINEITEM table includes a composite primary key (PK) constraint on `L_ORDERKEY` and `L_LINENUMBER`, foreign key (FK) constraints referencing `ORDERS.O_ORDERKEY` and `PARTSUPP(PS_PARTKEY, PS_SUPPKEY)`, and check constraints to ensure `L_QUANTITY`, `L_EXTENDEDPRICE`, and `L_TAX` are non-negative, and `L_DISCOUNT` is between 0 and 1.

```
CREATE TABLE LINEITEM (
    L_ORDERKEY INTEGER NOT NULL,
    L_PARTKEY INTEGER NOT NULL,
    L_SUPPKEY INTEGER NOT NULL,
    L_LINENUMBER INTEGER NOT NULL,
    L_QUANTITY DECIMAL NOT NULL CHECK (L_QUANTITY >= 0),
    L_EXTENDEDPRICE DECIMAL NOT NULL CHECK (L_EXTENDEDPRICE >= 0),
    L_DISCOUNT DECIMAL NOT NULL CHECK (L_DISCOUNT BETWEEN 0 AND 1),
    L_TAX DECIMAL NOT NULL CHECK (L_TAX >= 0),
    L_RETURNFLAG CHAR(1),
    L_LINESTATUS CHAR(1),
    L_SHIPDATE DATE,
    L_COMMITDATE DATE,
    L_RECEIPTDATE DATE CHECK (L_SHIPDATE <= L_RECEIPTDATE),
    L_SHIPINSTRUCT CHAR(25),
    L_SHIPMODE CHAR(10),
    L_COMMENT VARCHAR(44),
    FOREIGN KEY (L_ORDERKEY) REFERENCES ORDERS(O_ORDERKEY),
    FOREIGN KEY (L_PARTKEY, L_SUPPKEY) REFERENCES PARTSUPP(PS_PARTKEY, PS_SUPPKEY),
```

```
    PRIMARY KEY (L_ORDERKEY, L_LINENUMBER)  -- Composite primary key
);
```

Figure 9: Creation of LINEITEM Table

Figure 10: Alter of LINEITEM Table

Part III
Data Population and Statistical Analysis

After defining the schema, the next step involved populating the database with data generated by the TPC-H `dbgen` tool at a Scale Factor (SF) of 10, ensuring a substantial dataset for performance testing and benchmarking. This section details the process of generating the data, converting it into a suitable format for PostgreSQL, and executing the necessary commands to load the data into the database. Screenshots and commands illustrate each step, from data generation to successful data loading, ensuring the database is fully populated and ready for further analysis.

2 Data Population

After defining the schema, I populated the database using the data generated by the dbgen tool. T**he data was generated with a Scale Factor (SF) of 10 as required**. Below are the details of how I implemented the dbgen tool, generated the data, and populated the PostgreSQL database from CSV files.

2.1 Data Generation

To generate data from the TPC-H tool, I executed the following command in the `dbgen` folder:

```
./dbgen -s 10
```

This command generates the data with a Scale Factor (SF) of 10, which means the dataset will be scaled to represent a larger volume of data suitable for performance testing and benchmarking. The generated data files are in the `.tbl` format, as shown in the figure below.

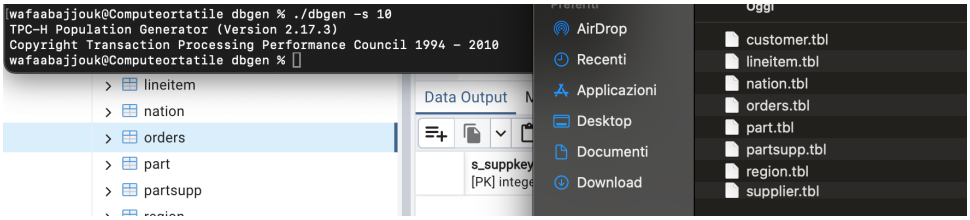


Figure 11: Data generation using the TPC-H tool

The `dbgen` tool generates several `.tbl` files corresponding to different tables in the TPC-H schema, such as `customer.tbl`, `lineitem.tbl`, `nation.tbl`, `orders.tbl`, `part.tbl`, `partsupp.tbl`, `region.tbl`, and `supplier.tbl`. These files contain the raw data that will be loaded into the PostgreSQL database.

The screenshot above shows the execution of the `dbgen` command and the resulting `.tbl` files that were generated. Each file corresponds to a table in the TPC-H schema and will be used to populate the database .

2.2 Data Conversion from `.tbl` to `.csv`

To convert the generated data files from the `.tbl` format to the `.csv` format, I used the following command:

```
for i in ‘ls *.tbl‘; do sed ’s/|$//’ $i > ${i/tbl/csv}; echo $i; done;
```

This command iterates through all the `.tbl` files, removes the trailing pipe character (|), and converts each file to a `.csv` file. The output files are then ready for loading into the PostgreSQL database.

The screenshot below shows the execution of the command and the resulting `.csv` files:

The figure illustrates the terminal command execution and the resulting `.csv` files alongside the original `.tbl` files. Each `.tbl` file, such as `customer.tbl`, `lineitem.tbl`, `nation.tbl`, etc., has been successfully converted to its corresponding `.csv` file.

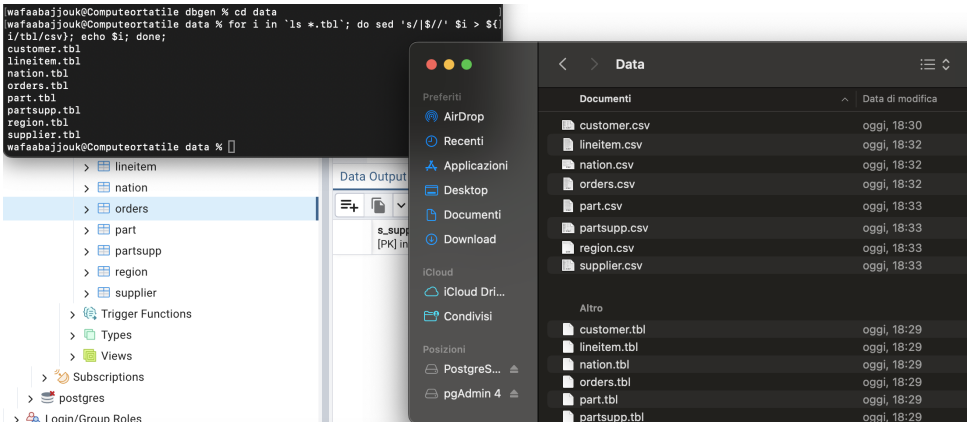


Figure 12: Conversion of data from `.tbl` to `.csv` format

2.3 Populating the DB using data generated by dbgen CSV files

To populate the data into a PostgreSQL database using CSV files, I used the following commands. Ensure to load the data in the specified order to avoid constraint integrity errors:

```
COPY REGION FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/
   data/region.csv' DELIMITER '|' CSV;
COPY NATION FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/
   data/nation.csv' DELIMITER '|' CSV;
COPY PART FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/
   part.csv' DELIMITER '|' CSV;
COPY SUPPLIER FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/
   data/supplier.csv' DELIMITER '|' CSV;
COPY PARTSUPP FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/
   data/partsupp.csv' DELIMITER '|' CSV;
COPY CUSTOMER FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/
   data/customer.csv' DELIMITER '|' CSV;
COPY ORDERS FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/
   data/orders.csv' DELIMITER '|' CSV;
COPY LINEITEM FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/
   data/lineitem.csv' DELIMITER '|' CSV;
```

See the screenshots of the executions below.

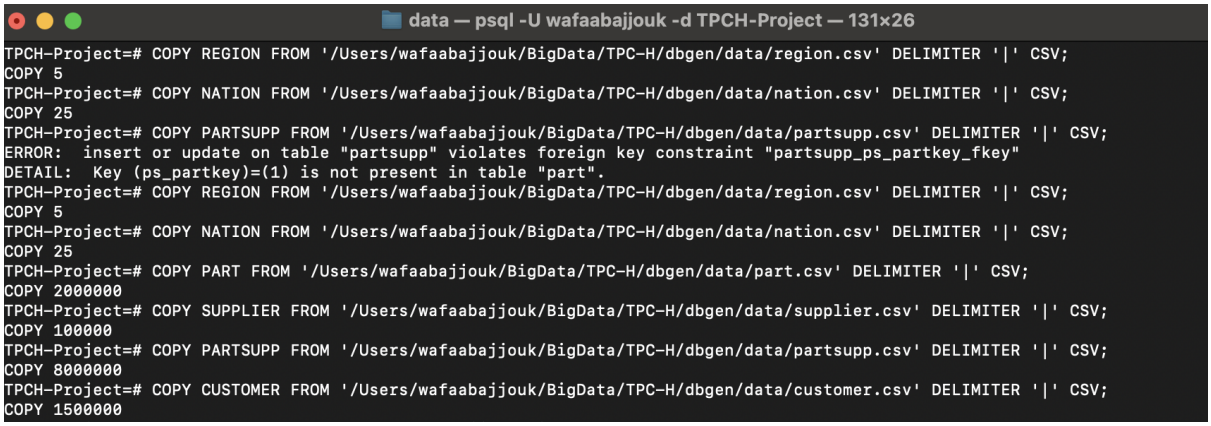


Figure 13: Initial Data Load Execution

Figure 14: Successful Data Load Execution

With the database now fully populated with data, the next steps involve implementing the Query Schema 1: Export/Import Revenue Value.

3 Statistics of the Data

In this section, I will calculate various statistics of the database, including the number of rows in each table, the table sizes in bytes, the number of distinct values for relevant attributes, and the minimum and maximum values for relevant attributes.

3.1 Size of the Database

The total size of the database is: **20GB**. The following figure shows the execution of the query that calculates the database size.

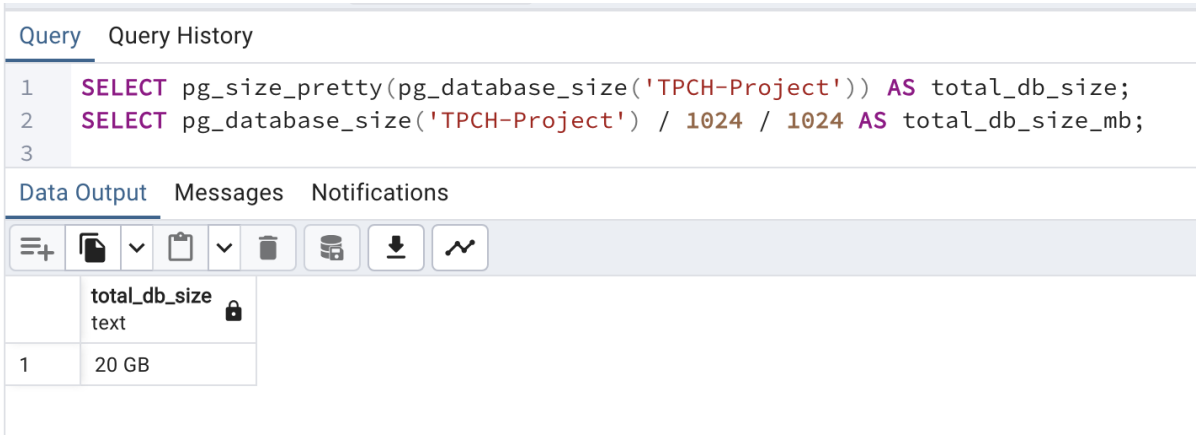


Figure 15: Database Size

3.2 Number of Rows in Each Table

The following query calculates the number of rows in each table:

```
SELECT 'REGION' AS TableName, COUNT(*) AS RowCount FROM REGION
UNION ALL
SELECT 'NATION', COUNT(*) FROM NATION
UNION ALL
SELECT 'PART', COUNT(*) FROM PART
UNION ALL
SELECT 'SUPPLIER', COUNT(*) FROM SUPPLIER
UNION ALL
SELECT 'PARTSUPP', COUNT(*) FROM PARTSUPP
UNION ALL
SELECT 'CUSTOMER', COUNT(*) FROM CUSTOMER
UNION ALL
SELECT 'ORDERS', COUNT(*) FROM ORDERS
UNION ALL
SELECT 'LINEITEM', COUNT(*) FROM LINEITEM;
```

This query uses 'UNION ALL' to combine the row counts from each table into a single result set.

3.3 Table Sizes in Bytes

The following query retrieves the size of each table in bytes:

```
SELECT
    relname AS TableName,
```

Query | Query History

```sql
-- Number of rows in each table
SELECT 'REGION' AS TableName, COUNT(*) AS RowCount FROM REGION
UNION ALL
SELECT 'NATION', COUNT(*) FROM NATION
UNION ALL
SELECT 'PART', COUNT(*) FROM PART
UNION ALL
SELECT 'SUPPLIER', COUNT(*) FROM SUPPLIER
UNION ALL
SELECT 'PARTSUPP', COUNT(*) FROM PARTSUPP
UNION ALL
SELECT 'CUSTOMER', COUNT(*) FROM CUSTOMER
UNION ALL
SELECT 'ORDERS', COUNT(*) FROM ORDERS
UNION ALL
SELECT 'LINEITEM', COUNT(*) FROM LINEITEM;

-- Table sizes in bytes
```

Data Output | Messages | Notifications

	tablename text	rowcount bigint
1	REGION	5
2	NATION	25
3	SUPPLIER	100000
4	CUSTOMER	1500000
5	PART	2000000
6	PARTSUPP	8000000
7	ORDERS	15000000
8	LINEITEM	59986052

Figure 16: Number of rows in each table

```
    pg_size_pretty(pg_total_relation_size(relid)) AS TotalSize
FROM
    pg_catalog.pg_statio_user_tables
ORDER BY
    pg_total_relation_size(relid) DESC;
```

This query uses PostgreSQL `pg_total_relation_size` function to get the total size of each table, and `pg_size_pretty` to format the size in Bytes

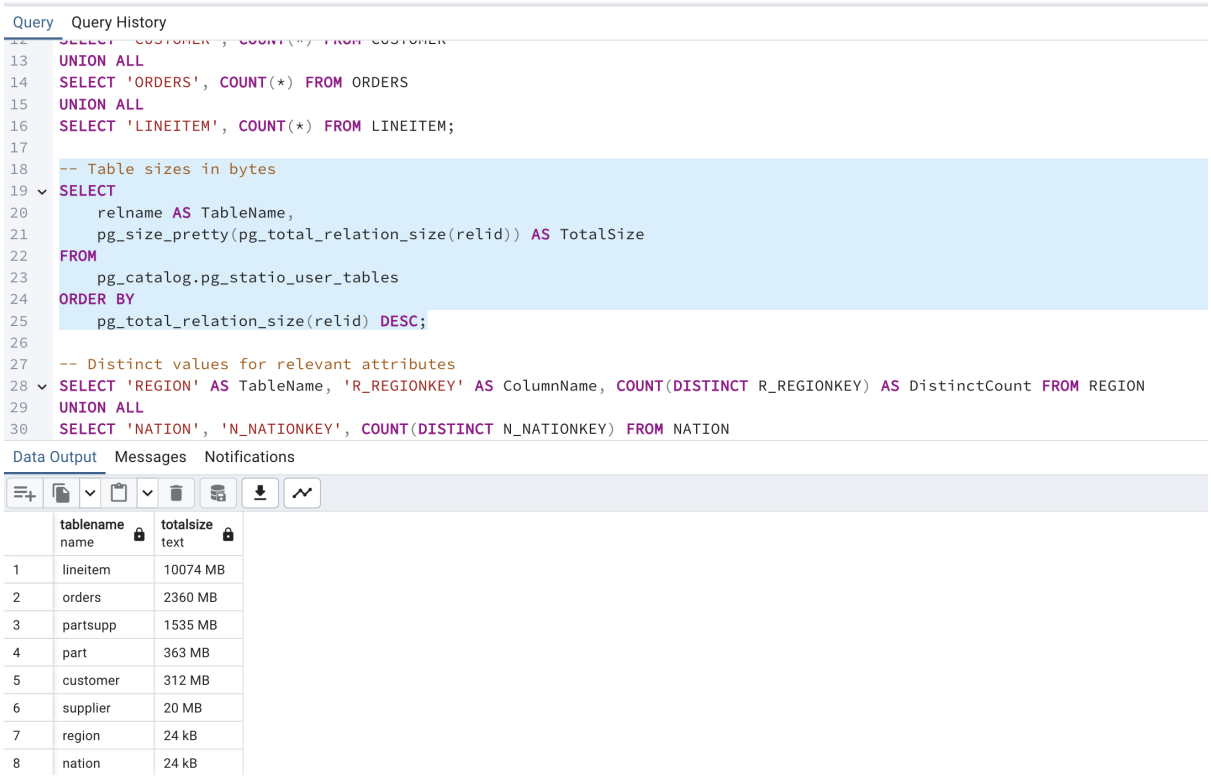


Figure 17: Table sizes in bytes

3.4 Distinct Values for Relevant Attributes

The following query calculates the number of distinct values for relevant attributes in each table:

```
SELECT 'REGION' AS TableName, 'R_REGIONKEY' AS ColumnName, COUNT(DISTINCT R_REGIONKEY
UNION ALL
SELECT 'NATION', 'N_NATIONKEY', COUNT(DISTINCT N_NATIONKEY) FROM NATION
UNION ALL
SELECT 'PART', 'P_PARTKEY', COUNT(DISTINCT P_PARTKEY) FROM PART
UNION ALL
SELECT 'SUPPLIER', 'S_SUPPKEY', COUNT(DISTINCT S_SUPPKEY) FROM SUPPLIER
UNION ALL
SELECT 'PARTSUPP', 'PS_PARTKEY', COUNT(DISTINCT PS_PARTKEY) FROM PARTSUPP
UNION ALL
SELECT 'CUSTOMER', 'C_CUSTKEY', COUNT(DISTINCT C_CUSTKEY) FROM CUSTOMER
UNION ALL
SELECT 'ORDERS', 'O_ORDERKEY', COUNT(DISTINCT O_ORDERKEY) FROM ORDERS
UNION ALL
SELECT 'LINEITEM', 'L_ORDERKEY', COUNT(DISTINCT L_ORDERKEY) FROM LINEITEM;
```

This query uses `COUNT(DISTINCT column_name)` to count the number of unique values in the specified columns.

Query Query History

```
pg_total_relation_size(relid) DESC;

-- Distinct values for relevant attributes
SELECT 'REGION' AS TableName, 'R_REGIONKEY' AS ColumnName, COUNT(DISTINCT R_REGIONKEY) AS DistinctCount FROM REGION
UNION ALL
SELECT 'NATION', 'N_NATIONKEY', COUNT(DISTINCT N_NATIONKEY) FROM NATION
UNION ALL
SELECT 'PART', 'P_PARTKEY', COUNT(DISTINCT P_PARTKEY) FROM PART
UNION ALL
SELECT 'SUPPLIER', 'S_SUPPKEY', COUNT(DISTINCT S_SUPPKEY) FROM SUPPLIER
UNION ALL
SELECT 'PARTSUPP', 'PS_PARTKEY', COUNT(DISTINCT PS_PARTKEY) FROM PARTSUPP
UNION ALL
SELECT 'CUSTOMER', 'C_CUSTKEY', COUNT(DISTINCT C_CUSTKEY) FROM CUSTOMER
UNION ALL
SELECT 'ORDERS', 'O_ORDERKEY', COUNT(DISTINCT O_ORDERKEY) FROM ORDERS
UNION ALL
SELECT 'LINEITEM', 'L_ORDERKEY', COUNT(DISTINCT L_ORDERKEY) FROM LINEITEM;
```

Data Output Messages Notifications

	tablename text	columnname text	distinctcount bigint
1	REGION	R_REGIONKEY	5
2	NATION	N_NATIONKEY	25
3	SUPPLIER	S_SUPPKEY	100000
4	CUSTOMER	C_CUSTKEY	1500000
5	PART	P_PARTKEY	2000000
6	PARTSUPP	PS_PARTKEY	2000000
7	ORDERS	O_ORDERKEY	15000000
8	LINEITEM	L_ORDERKEY	15000000

Figure 18: Distinct values for relevant attributes

3.5 Minimum and Maximum Values for Relevant Attributes

The following query calculates the minimum and maximum values for relevant attributes in each table:

```
SELECT 'REGION' AS TableName, 'R_REGIONKEY' AS ColumnName, MIN(R_REGIONKEY) AS MinVal
UNION ALL
SELECT 'NATION', 'N_NATIONKEY', MIN(N_NATIONKEY), MAX(N_NATIONKEY) FROM NATION
UNION ALL
SELECT 'PART', 'P_PARTKEY', MIN(P_PARTKEY), MAX(P_PARTKEY) FROM PART
UNION ALL
SELECT 'SUPPLIER', 'S_SUPPKEY', MIN(S_SUPPKEY), MAX(S_SUPPKEY) FROM SUPPLIER
UNION ALL
SELECT 'PARTSUPP', 'PS_PARTKEY', MIN(PS_PARTKEY), MAX(PS_PARTKEY) FROM PARTSUPP
UNION ALL
SELECT 'CUSTOMER', 'C_CUSTKEY', MIN(C_CUSTKEY), MAX(C_CUSTKEY) FROM CUSTOMER
UNION ALL
SELECT 'ORDERS', 'O_ORDERKEY', MIN(O_ORDERKEY), MAX(O_ORDERKEY) FROM ORDERS
UNION ALL
SELECT 'LINEITEM', 'L_ORDERKEY', MIN(L_ORDERKEY), MAX(L_ORDERKEY) FROM LINEITEM;
```

This query uses `MIN(column_name)` and `MAX(column_name)` to find the minimum and maximum values in the specified columns.

```sql
-- Min and max values for relevant attributes
SELECT 'REGION' AS TableName, 'R_REGIONKEY' AS ColumnName, MIN(R_REGIONKEY) AS MinValue, MAX(R_REGIONKEY) AS MaxValue FROM REGION
UNION ALL
SELECT 'NATION', 'N_NATIONKEY', MIN(N_NATIONKEY), MAX(N_NATIONKEY) FROM NATION
UNION ALL
SELECT 'PART', 'P_PARTKEY', MIN(P_PARTKEY), MAX(P_PARTKEY) FROM PART
UNION ALL
SELECT 'SUPPLIER', 'S_SUPPKEY', MIN(S_SUPPKEY), MAX(S_SUPPKEY) FROM SUPPLIER
UNION ALL
SELECT 'PARTSUPP', 'PS_PARTKEY', MIN(PS_PARTKEY), MAX(PS_PARTKEY) FROM PARTSUPP
UNION ALL
SELECT 'CUSTOMER', 'C_CUSTKEY', MIN(C_CUSTKEY), MAX(C_CUSTKEY) FROM CUSTOMER
UNION ALL
SELECT 'ORDERS', 'O_ORDERKEY', MIN(O_ORDERKEY), MAX(O_ORDERKEY) FROM ORDERS
UNION ALL
SELECT 'LINEITEM', 'L_ORDERKEY', MIN(L_ORDERKEY), MAX(L_ORDERKEY) FROM LINEITEM;
```

Data Output | Messages | Notifications

	tablename text	columnname text	minvalue integer	maxvalue integer
1	REGION	R_REGIONKEY	0	4
2	NATION	N_NATIONKEY	0	24
3	PART	P_PARTKEY	1	2000000
4	SUPPLIER	S_SUPPKEY	1	100000
5	PARTSUPP	PS_PARTKEY	1	2000000
6	CUSTOMER	C_CUSTKEY	1	1500000
7	ORDERS	O_ORDERKEY	1	60000000
8	LINEITEM	L_ORDERKEY	1	60000000

Figure 19: Minimum and maximum values for relevant attributes

Part IV
Query Schema and Analysis Before Optimization

This section explains the initial query setup and analysis used to calculate the revenue from exporting and importing goods between two countries. The goal is to measure the revenue accurately by using different methods to group and filter data. These methods include time-based, type-based, and geographical aggregations. I also look at the initial performance and costs of these queries before applying any optimization techniques. This analysis gives us a clear picture of the starting point for query performance, helping us understand what needs to be improved.

4 the Query Schema 1: Export/Import Revenue Value

4.1 Objective

The goal is to calculate the revenue from exporting and importing goods between two different countries. Specifically, Iam looking at transactions where:

- **Export**: The supplier is in country E.
- **Import**: The customer is in country I.

4.2 Revenue Calculation

Revenue for each transaction is calculated using the formula:

$$\text{Revenue} = \text{l\_extendedprice} \times (1 - \text{l\_discount}) \tag{1}$$

This formula accounts for the discounted price of the items.

4.3 Aggregations And Slicing :

In this exercise, roll-up and slicing operations are crucial for aggregating and analyzing the data at various hierarchical levels. Here's an explanation of each operation:

Roll-Up Operations

Roll-up operations allow us to aggregate data by climbing up a hierarchy of dimensions. This helps in summarizing data at different levels of granularity. In this scenario, the roll-up operations will be performed as follows:

- Month ->Quarter -> Year
- Type
- Nation -> Region

Slicing Operations

Slicing involves filtering data to focus on specific values within a dimension. This is useful for detailed analysis of particular segments. In my scenario, slicing will be performed as follows:

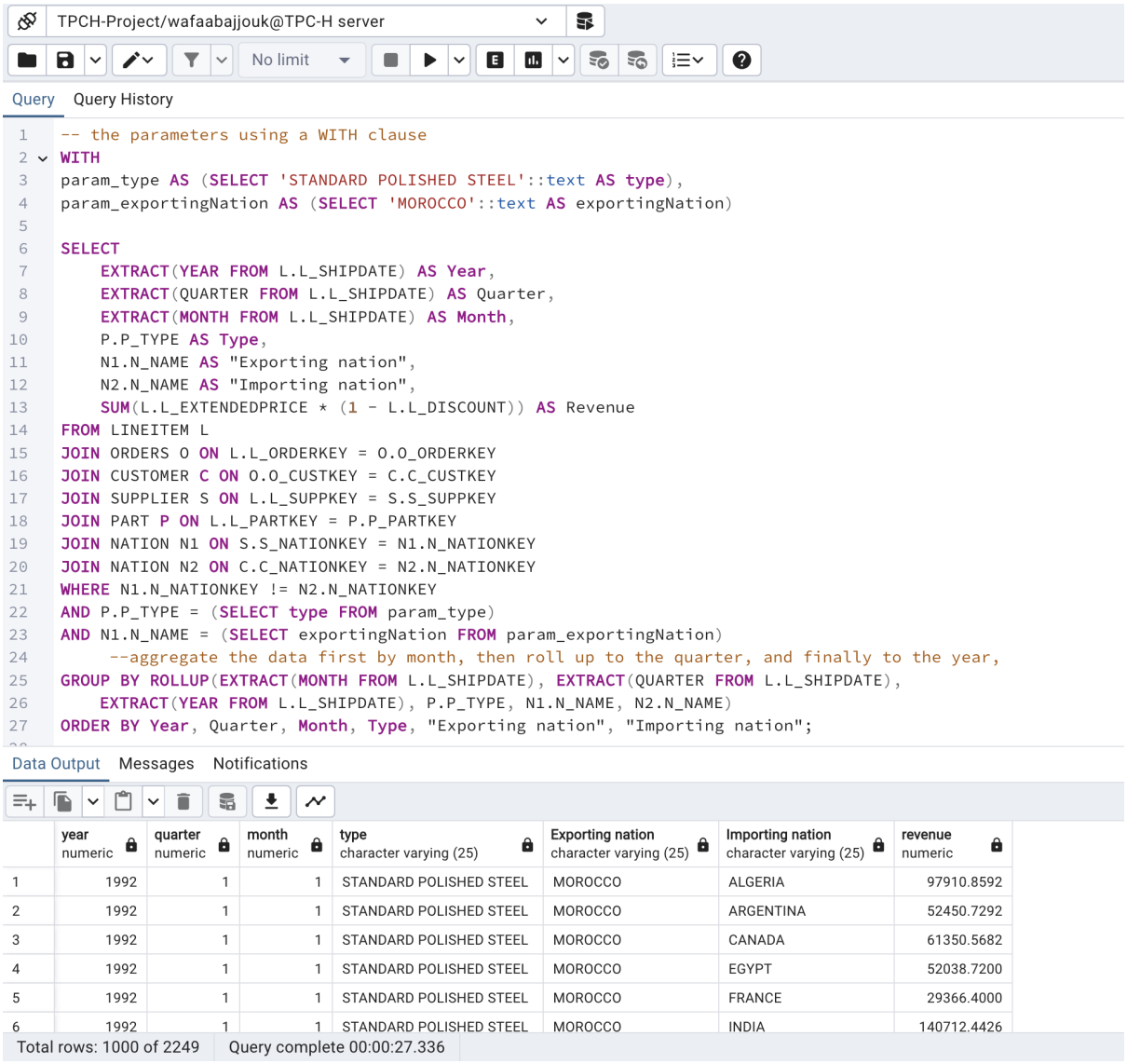
- **Type**: Focus on specific types of lineitems
- **Exporting Nation**: Focus on specific countries that are exporting goods.

4.4 Queries :

4.4.1 Time-Based Aggregation

The order inside the ROLLUP specifies the hierarchy :

- First, **it aggregates by month.**
- Then, it rolls up the monthly aggregates **into quarterly aggregates.**
- Finally, it rolls up the quarterly aggregates **into yearly aggregates.**



	year numeric	quarter numeric	month numeric	type character varying (25)	Exporting nation character varying (25)	Importing nation character varying (25)	revenue numeric
1	1992	1	1	STANDARD POLISHED STEEL	MOROCCO	ALGERIA	97910.8592
2	1992	1	1	STANDARD POLISHED STEEL	MOROCCO	ARGENTINA	52450.7292
3	1992	1	1	STANDARD POLISHED STEEL	MOROCCO	CANADA	61350.5682
4	1992	1	1	STANDARD POLISHED STEEL	MOROCCO	EGYPT	52038.7200
5	1992	1	1	STANDARD POLISHED STEEL	MOROCCO	FRANCE	29366.4000
6	1992	1	1	STANDARD POLISHED STEEL	MOROCCO	INDIA	140712.4426

Figure 20: Time-Based Aggregation

- Parametric Values using the WITH clause. in param_type I Picked the product type as '**STANDARD POLISHED STEEL**'.
- `param_exportingNation`: I Picked the exporting nation as '**MOROCCO**'.

- Extracts the year, quarter, and month from the `L_SHIPDATE` column in the `LINEITEM` table.
- Joins several tables (`ORDERS`, `CUSTOMER`, `SUPPLIER`, `PART`, `NATION`) to gather necessary data.
- The slicing is over Type and Exporting nation .The query Filters the results based on **product type and exporting nation using the defined parameters.**
- Ensures the exporting and importing nations are different.
- Aggregates the revenue by multiplying the extended price by the discount factor, summing these values.
- Uses the `ROLLUP` function to group by month, quarter, and year, allowing for hierarchical aggregation.
- Orders the results by year, quarter, month, type, exporting nation, and importing nation.

4.4.2 Type-Based Aggregation

- Grouping by the type of lineitem

- `params`: I picked the product type as '**ECONOMY ANODIZED BRASS' and the exporting nation as 'CANADA'.**
- Selects the product type, exporting nation, and importing nation.
- Calculates the revenue by multiplying the extended price by the discount factor and summing these values.
- Joins several tables (`PART`, `ORDERS`, `CUSTOMER`, `SUPPLIER`, `NATION`) to gather necessary data.
- Uses the `ROLLUP` function to group by **product type**, exporting nation, and importing nation, allowing for hierarchical aggregation.
- Orders the results by type, exporting nation, and importing nation.
- Filters the results based on product type and exporting nation using the defined parameters.
- Ensures the exporting and importing nations are different.

Query Query History

```
1  WITH params AS (
2      SELECT 'ECONOMY ANODIZED BRASS'::text AS type,
3             'CANADA'::text AS exportingNation
4  )
5
6  SELECT
7      P.P_TYPE AS Type,
8      N1.N_NAME AS "Exporting nation",
9      N2.N_NAME AS "Importing nation",
10     SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
11 FROM LINEITEM L
12 JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
13 JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
14 JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
15 JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
16 JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
17 JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
18 JOIN params ON TRUE
19 WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
20 AND P.P_TYPE = params.type
21 AND N1.N_NAME = params.exportingNation
22 GROUP BY ROLLUP(P.P_TYPE, N1.N_NAME, N2.N_NAME)
23 ORDER BY Type, "Exporting nation", "Importing nation";
24
```

Data Output Messages Notifications

	type character varying (25)	Exporting nation character varying (25)	Importing nation character varying (25)	revenue numeric
1	ECONOMY ANODIZED BRASS	CANADA	ALGERIA	21063762.1642
2	ECONOMY ANODIZED BRASS	CANADA	ARGENTINA	23305153.3875
3	ECONOMY ANODIZED BRASS	CANADA	BRAZIL	22761889.5684
4	ECONOMY ANODIZED BRASS	CANADA	CHINA	22700851.3073
5	ECONOMY ANODIZED BRASS	CANADA	EGYPT	22765921.5024
6	ECONOMY ANODIZED BRASS	CANADA	ETHIOPIA	22263374.0327
7	ECONOMY ANODIZED BRASS	CANADA	FRANCE	22379202.3853
8	ECONOMY ANODIZED BRASS	CANADA	GERMANY	23026963.7384
9	ECONOMY ANODIZED BRASS	CANADA	INDIA	23379976.1618

Total rows: 27 of 27 Query complete 00:00:48.560

Figure 21: Type-Based Aggregation

4.4.3 Geographical Aggregation

- **Nation**: Group by each specific nation.
- **Region**: Group by larger geographical regions that include multiple nations.

- `params`: I picked **the product type as 'ECONOMY BRUSHED STEEL' and the exporting nation as 'ARGENTINA'.**
- Selects the exporting nation, exporting region, importing nation, importing region, and product type.
- Calculates the revenue by multiplying the extended price by the discount factor and summing these values.

Query Query History

```sql
WITH params AS (
    SELECT 'ECONOMY BRUSHED STEEL'::text AS type,
           'ARGENTINA'::text AS exportingNation
)
SELECT
    N1.N_NAME AS "Exporting nation",
    R1.R_NAME AS "Exporting region",
    N2.N_NAME AS "Importing nation",
    R2.R_NAME AS "Importing region",
    P.P_TYPE AS Type,
    SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
FROM LINEITEM L
JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
JOIN REGION R1 ON N1.N_REGIONKEY = R1.R_REGIONKEY
JOIN REGION R2 ON N2.N_REGIONKEY = R2.R_REGIONKEY
JOIN params ON TRUE
WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
AND P.P_TYPE = params.type
AND N1.N_NAME = params.exportingNation
GROUP BY ROLLUP(N1.N_NAME, R1.R_NAME, N2.N_NAME, R2.R_NAME, P.P_TYPE)
ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
```

Data Output Messages Notifications

	Exporting nation character varying (25)	Exporting region character varying (25)	Importing nation character varying (25)	Importing region character varying (25)	type character varying (25)	revenue numeric
1	ARGENTINA	AMERICA	ALGERIA	AFRICA	ECONOMY BRUSHED STEEL	22405660.6732
2	ARGENTINA	AMERICA	ALGERIA	AFRICA	[null]	22405660.6732
3	ARGENTINA	AMERICA	ALGERIA	[null]	[null]	22405660.6732
4	ARGENTINA	AMERICA	BRAZIL	AMERICA	ECONOMY BRUSHED STEEL	23458198.9493
5	ARGENTINA	AMERICA	BRAZIL	AMERICA	[null]	23458198.9493
6	ARGENTINA	AMERICA	BRAZIL	[null]	[null]	23458198.9493

Total rows: 75 of 75 Query complete 00:00:56.745

Figure 22: Geographical Aggregation

- Joins several tables (`ORDERS`, `CUSTOMER`, `SUPPLIER`, `PART`, `NATION`, `REGION`) to gather necessary data.
- Uses the `ROLLUP` function to group by **exporting nation, exporting region, importing nation, importing region, and product type, allowing for hierarchical aggregation.**
- Orders the results by exporting nation, exporting region, importing nation, importing region, and product type.
- Filters the results based on product type and exporting nation using the defined parameters.
- Ensures the exporting and importing nations are different.

5 Query Cost before Optimization

Understanding the cost of queries is essential for optimizing performance within PostgreSQL. By utilizing the `EXPLAIN ANALYZE` command, I will delve into the intricacies of query execution, to be able in the next section to identify the areas for improvement and efficiency gains.

5.1 Query Cost of Time-Based Aggregation

This subsection provides insights into the cost of time-based aggregation queries. By analyzing the bellow figures, I get a deeper understanding of the execution process, including sorting operations, memory utilization, and overall execution time.

The Figures 22 and 23 of the execution Indicates :

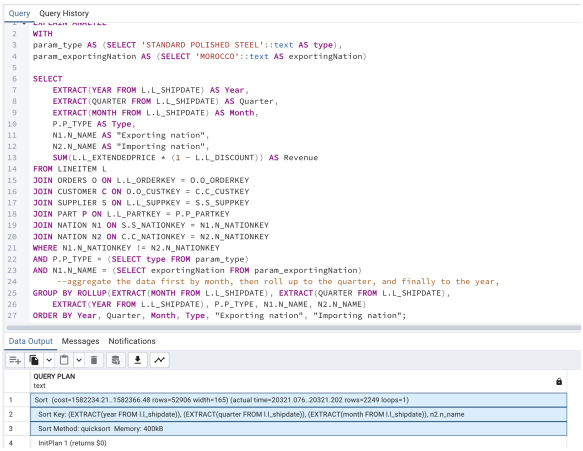


Figure 23: Query Cost of Time-Based Aggregation: COST and ROWS

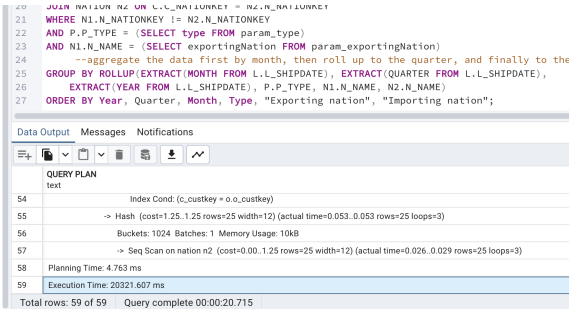


Figure 24: Query Cost of Time-Based Aggregation: Execution Time

- **The cost** of the sorting operation is in the range of **1582234.21 to 1582366.48**. The actual time taken for the sorting process is **20321.076 to 20321.202 milliseconds**, with **2249 rows** processed.
- The sorting algorithm employed and the amount of memory utilized during sorting: **Sort Method: quicksort Memory: 400kB**
- The total time taken to execute the query is **20321.607 milliseconds.**

5.2 Query Cost of Type Aggregation

This subsection delves into the cost analysis of type aggregation queries. By analyzing the provided figures 24 and 25, I can conclude the cost of the type-based aggregation, including execution time, cost range, and row processing details.

In the provided snippet:

- **The cost** of the grouping operation is in the range of **3228022.28 to 3228246.04**. The actual time taken for the grouping process is **49142.324 to 50016.339 milliseconds**, with **27 rows** processed.
- The total time taken to execute the query is **50016.603 milliseconds.**

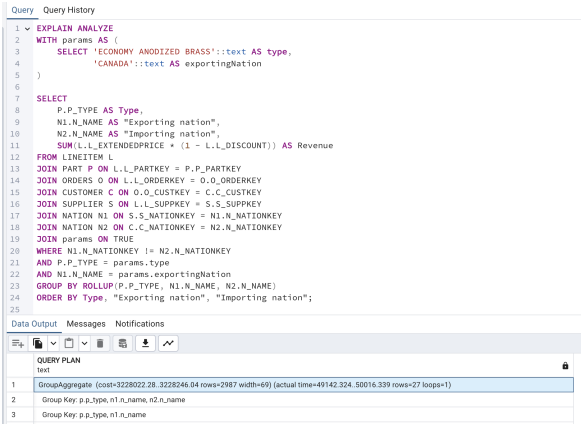


Figure 25: Query Cost of Type Aggregation: COST and ROWS

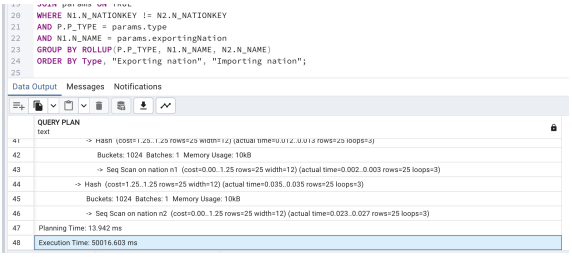


Figure 26: Query Cost of Type Aggregation: Execution Time

5.3 Query Cost of Region and Nation Aggregation

This subsection offers an analysis of the cost associated with region and nation aggregation queries. By examining the provided figures 26 and 27, I conclude the execution time, cost range, and row processing details.

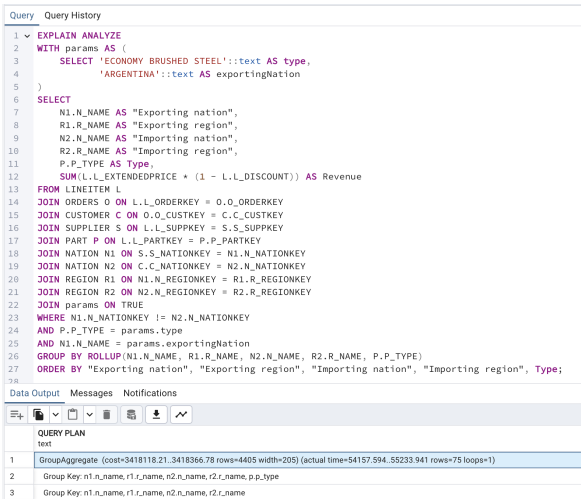


Figure 27: Query Cost of Region and Nation Aggregation: COST and ROWS

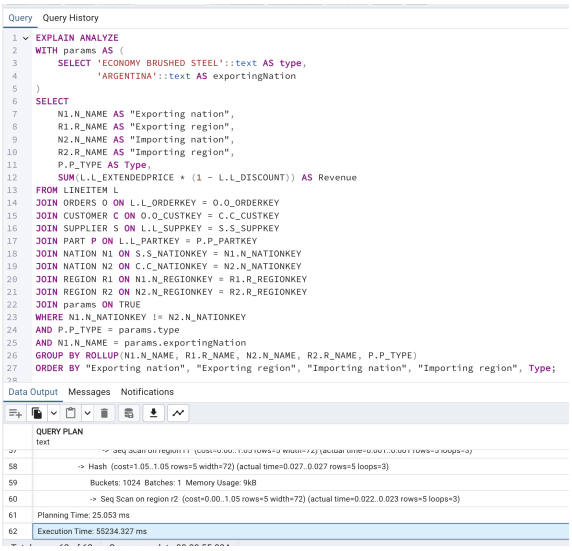


Figure 28: Query Cost of Region and Nation Aggregation: Execution Time

In the Execution of the analysis indicates :

- **The cost** of the grouping operation is in the range of **3418118.21 to 3418366.78**. The actual time taken for the grouping process is **54157.594 to 55233.941 milliseconds**, with **75 rows** processed.
- **Planning Time: 25.053 milliseconds.**
- The total time taken to execute the query is **55234.327 milliseconds.**

Part V
Index Design and Query Cost with Indexes

This chapter covers how indexing was used to improve database performance. Indexing helps reduce search times and makes database operations more efficient, especially for complex queries. The chapter starts with creating indexes on important tables and then looks at the space costs of these indexes. It also checks to ensure that the total size of the indexes stays within acceptable limits. After that, the chapter compares query costs and execution times before and after indexing, showing significant improvements. The results highlight how effective indexing is in reducing execution time and resource use for various aggregation queries.

6 Design of indexes

The optimization of the queries involves several strategies that aim to improve the performance of the database operations. Indexing is an important step in query optimization as it significantly reduces the search time for rows in a table.

6.1 Creation of Indexes on Tables

The following indexes were created:

Query | Query History

```
1   -- LINEITEM
2   CREATE INDEX idx_lineitem_orderkey_shipdate ON LINEITEM (L_ORDERKEY, L_SHIPDATE);
3   CREATE INDEX idx_lineitem_partkey_suppkey ON LINEITEM (L_PARTKEY, L_SUPPKEY);    --
4   CREATE INDEX idx_lineitem_extendedprice_discount ON LINEITEM (L_EXTENDEDPRICE, L_DI
5   CREATE INDEX idx_lineitem_partkey_suppkey_orderkey ON LINEITEM (L_PARTKEY, L_SUPPKE
6   -- ORDERS
7   CREATE INDEX idx_orders_custkey_orderdate ON ORDERS (O_CUSTKEY, O_ORDERDATE);    --
8   -- CUSTOMER
9   CREATE INDEX idx_customer_nationkey_custkey ON CUSTOMER (C_NATIONKEY, C_CUSTKEY);--
10  -- SUPPLIER
11  CREATE INDEX idx_supplier_nationkey_suppkey ON SUPPLIER (S_NATIONKEY, S_SUPPKEY);--
12  -- PART
13  CREATE INDEX idx_part_type ON PART (P_TYPE);  -- For filtering by part type
14  CREATE INDEX idx_part_type_partkey ON PART (P_TYPE, P_PARTKEY);  -- For filtering b
15  CREATE INDEX idx_part_partkey_type ON PART (P_PARTKEY, P_TYPE); -- For joining with
16  -- NATION
17  CREATE INDEX idx_nation_regionkey_nationkey ON NATION (N_REGIONKEY, N_NATIONKEY);--
18  CREATE INDEX idx_nation_name ON NATION (N_NAME);  --For filtering by nation names
19  -- REGION
20  CREATE INDEX idx_region_name ON REGION (R_NAME);  --For filtering by region names
21
```

Data Output | Messages | Notifications

```
CREATE INDEX

Query returned successfully in 64 msec.
```

Figure 29: INDEXES

The creation of these indexes helps to improve the performance of join operations and filtering conditions in our queries.

6.2 Space Cost of Indexes

The SQL query shown in the figure 29 , retrieves the total size of various indexes in a PostgreSQL database. The query calculates the combined size of multiple indexes by summing the individual sizes retrieved using the ‘pg_total_relation_size‘ function.

This visualization in Figure 57 illustrates the aggregate space consumed by these indexes, highlighting the significant storage requirements for indexing in the database system.

The total space cost of these indexes is calculated to be **6021 MB**.

Query Query History

```sql
SELECT pg_size_pretty(SUM(size_bytes)) AS total_size
FROM (
    SELECT pg_total_relation_size('idx_lineitem_orderkey_shipdate') AS size_bytes
    UNION ALL
    SELECT pg_total_relation_size('idx_lineitem_partkey_suppkey')
    UNION ALL
    SELECT pg_total_relation_size('idx_lineitem_extendedprice_discount')
    UNION ALL
    SELECT pg_total_relation_size('idx_lineitem_partkey_suppkey_orderkey')
    UNION ALL
    SELECT pg_total_relation_size('idx_orders_custkey_orderdate')
    UNION ALL
    SELECT pg_total_relation_size('idx_customer_nationkey_custkey')
    UNION ALL
    SELECT pg_total_relation_size('idx_supplier_nationkey_suppkey')
    UNION ALL
    SELECT pg_total_relation_size('idx_part_type')
    UNION ALL
    SELECT pg_total_relation_size('idx_part_type_partkey')
    UNION ALL
    SELECT pg_total_relation_size('idx_part_partkey_type')
    UNION ALL
    SELECT pg_total_relation_size('idx_nation_regionkey_nationkey')
    UNION ALL
    SELECT pg_total_relation_size('idx_nation_name')
) AS total;
```

Data Output Messages Notifications

	total_size text
1	6021 MB

Figure 30: Space Cost of Indexing

6.3 Space Constraint Verification

AS mentioned in the project document , The final optimization must fulfill the space constraint that the amount of space for indexes and materialized views **must be less than 1.5 times** the size of the database.

- **Total Size of Indexes:** 6021 MB
- **Total Size of the Database:** 20 GB (20480 MB)
- **1.5 Times the Size of the Database:** 30720 MB

Since the total size of the indexes (6021 MB) is **less than 1.5 times the size of the database** (30720 MB) : 6021 MB $<$ 30720 MB , **the space constraint is satisfied.**

7 Query Cost of Queries with Indexes

In this section, I will execute the query that calculates the query cost after the creation of indexes.

7.1 Query Cost of Time-Based Aggregation Query with Indexing

The figure 31 shows that the execution time has also been reduced **from 27 seconds 336 milliseconds to 15 seconds 181 milliseconds after indexing.**

Query Query History

```
param_exportingNation AS (SELECT 'MOROCCO'::text AS exportingNation)

SELECT
    EXTRACT(YEAR FROM L.L_SHIPDATE) AS Year,
    EXTRACT(QUARTER FROM L.L_SHIPDATE) AS Quarter,
    EXTRACT(MONTH FROM L.L_SHIPDATE) AS Month,
    P.P_TYPE AS Type,
    N1.N_NAME AS "Exporting nation",
    N2.N_NAME AS "Importing nation",
    SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
FROM LINEITEM L
JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
AND P.P_TYPE = (SELECT type FROM param_type)
AND N1.N_NAME = (SELECT exportingNation FROM param_exportingNation)
     --aggregate the data first by month, then roll up to the quarter, and finally to the year,
GROUP BY ROLLUP(EXTRACT(MONTH FROM L.L_SHIPDATE), EXTRACT(QUARTER FROM L.L_SHIPDATE),
    EXTRACT(YEAR FROM L.L_SHIPDATE), P.P_TYPE, N1.N_NAME, N2.N_NAME)
ORDER BY Year, Quarter, Month, Type, "Exporting nation", "Importing nation";
--before indexing :27sec 336 msec
--after indexing : 15 secs 181 msec.

```

Data Output Messages Notifications

```
Successfully run. Total query runtime: 15 secs 181 msec.
2249 rows affected.
```

Figure 31: Time-Based Aggregation Query Execution Time Reduction

The cost of the Time-Based Aggregation query has been **reduced from 1,582,234.21 to 937,902.40** due to the indexes created.

Additionally, to view the script for calculating the query cost, use the following path: `/BigData/SQL SCRIPTS/COST_QUERY1.sql`

7.2 Query Cost of Type-Based Aggregation Query with Indexing

The following figure shows the execution time has been reduced **from 48 seconds 560 milliseconds to 45 seconds 591 milliseconds after indexing.** The Total query new runtime is 45 seconds 591 milliseconds.

Query Query History

```
3   param_type AS (SELECT 'STANDARD POLISHED STEEL'::text AS type),
4   param_exportingNation AS (SELECT 'MOROCCO'::text AS exportingNation)
5
6   SELECT
7       EXTRACT(YEAR FROM L.L_SHIPDATE) AS Year,
8       EXTRACT(QUARTER FROM L.L_SHIPDATE) AS Quarter,
9       EXTRACT(MONTH FROM L.L_SHIPDATE) AS Month,
10      P.P_TYPE AS Type,
11      N1.N_NAME AS "Exporting nation",
12      N2.N_NAME AS "Importing nation",
13      SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
14  FROM LINEITEM L
15  JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
16  JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
17  JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
18  JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
19  JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
20  JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
21  WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
22  AND P.P_TYPE = (SELECT type FROM param_type)
23  AND N1.N_NAME = (SELECT exportingNation FROM param_exportingNation)
24       --aggregate the data first by month, then roll up to the quarter, and finally to the year,
25  GROUP BY ROLLUP(EXTRACT(MONTH FROM L.L_SHIPDATE), EXTRACT(QUARTER FROM L.L_SHIPDATE),
26      EXTRACT(YEAR FROM L.L_SHIPDATE), P.P_TYPE, N1.N_NAME, N2.N_NAME)
27  ORDER BY Year, Quarter, Month, Type, "Exporting nation", "Importing nation";
28  --cost before indexes : cost=1582234.21..1582366.48 rows=52906 width=165) actual time=20321.076..20321.202 rows=2249 loops=1
29  --cost after indexes :  cost= 937902.40.. 938034.66 rows=52906 width=165) actual time=19017.393..19017.528 rows=2249 loops=1
```

Data Output Messages Notifications

	QUERY PLAN text
1	Sort (cost=937902.40..938034.66 rows=52906 width=165) (actual time=19017.393..19017.528 rows=2249 loops=1)
2	Sort Key: (EXTRACT(year FROM l.l_shipdate)), (EXTRACT(quarter FROM l.l_shipdate)), (EXTRACT(month FROM l.l_shipdate)), n2.n_name
3	Sort Method: quicksort Memory: 400kB

Figure 32: Cost Query Result

The cost of the Type-Based Aggregation query has been **reduced from 3228022.28 to 3215269.89** due to the indexes created.

Additionally, to view the script for calculating the query cost, use the following path: `/BigData/SQL SCRIPTS/COST_QUERY2.sql`

More data is shown in the picture below.

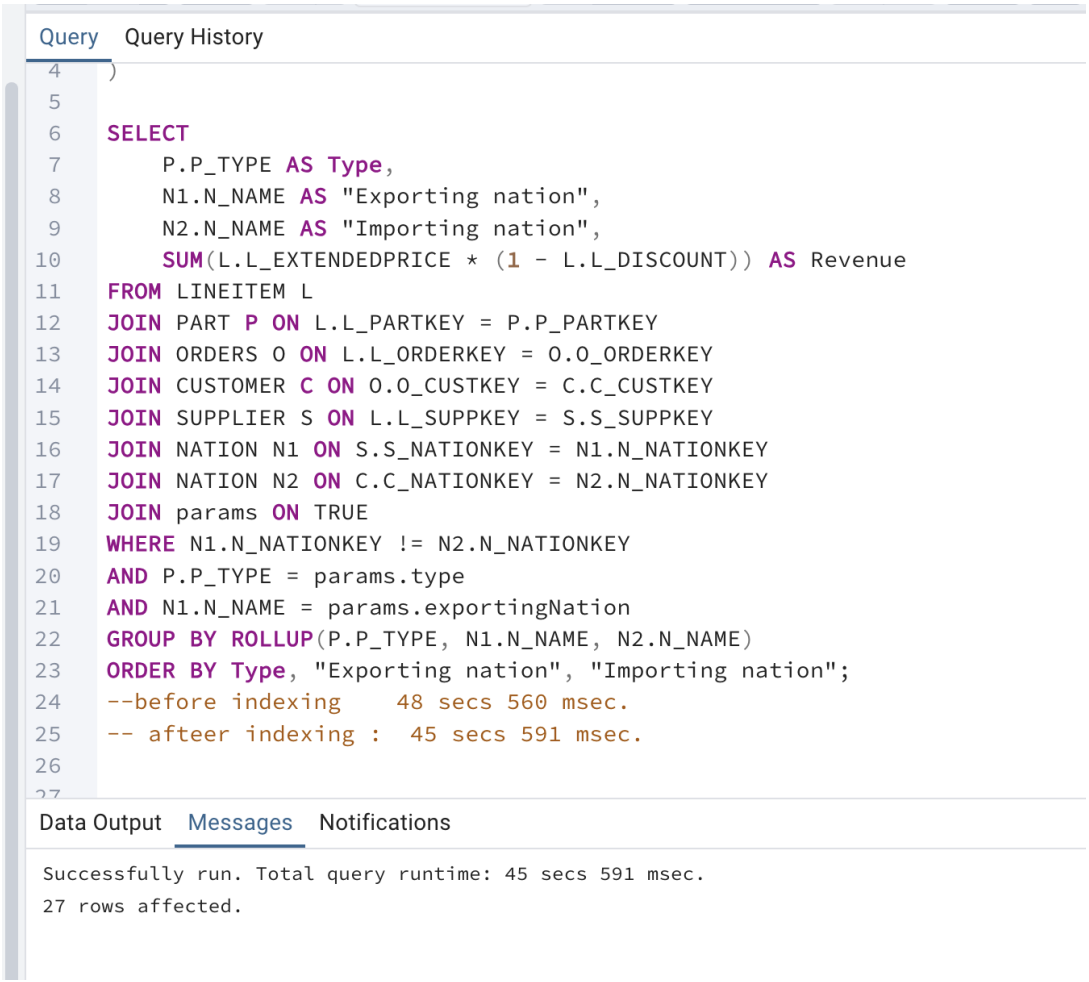


Figure 33: Type-Based Aggregation Query Execution Time Reduction

7.3 Query Cost of Region and Nation Aggregation Query with Indexing

The following figure shows the execution time has been reduced **from 56 seconds 745 milliseconds to 48 seconds 363 milliseconds after indexing.** The total query new runtime is 48 seconds 363 milliseconds.

Query Query History

```sql
WITH params AS (
    SELECT 'ECONOMY ANODIZED BRASS'::text AS type,
           'CANADA'::text AS exportingNation
)

SELECT
    P.P_TYPE AS Type,
    N1.N_NAME AS "Exporting nation",
    N2.N_NAME AS "Importing nation",
    SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
FROM LINEITEM L
JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
JOIN params ON TRUE
WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
AND P.P_TYPE = params.type
AND N1.N_NAME = params.exportingNation
GROUP BY ROLLUP(P.P_TYPE, N1.N_NAME, N2.N_NAME)
ORDER BY Type, "Exporting nation", "Importing nation";
-- cost before indexes  (cost= 3228022.28..3228246.04 rows=2987 width=69) (actual time=49142.324..50016.339 rows=27 loops=1)
-- cost after indexes:  (cost=3215269.89..3215493.65 rows=2987 width=69)  (actual time=44479.155..45363.290 rows=27 loops=1)
```

Data Output Messages Notifications

	QUERY PLAN text
1	GroupAggregate (cost=3215269.89..3215493.65 rows=2987 width=69) (actual time=44479.155..45363.290 rows=27 loops=1)
2	Group Key: p.p_type, n1.n_name, n2.n_name
3	Group Key: p.p_type, n1.n_name

Figure 34: Cost Query Result

Query Query History

```sql
    SELECT 'ECONOMY BRUSHED STEEL'::text AS type,
           'ARGENTINA'::text AS exportingNation
)
SELECT
    N1.N_NAME AS "Exporting nation",
    R1.R_NAME AS "Exporting region",
    N2.N_NAME AS "Importing nation",
    R2.R_NAME AS "Importing region",
    P.P_TYPE AS Type,
    SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
FROM LINEITEM L
JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
JOIN REGION R1 ON N1.N_REGIONKEY = R1.R_REGIONKEY
JOIN REGION R2 ON N2.N_REGIONKEY = R2.R_REGIONKEY
JOIN params ON TRUE
WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
AND P.P_TYPE = params.type
AND N1.N_NAME = params.exportingNation
GROUP BY ROLLUP(N1.N_NAME, R1.R_NAME, N2.N_NAME, R2.R_NAME, P.P_TYPE)
ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
-- before indexing : 56 secs 745 msec.
-- after indexing :  48 secs 363 msec.

```

Data Output Messages Notifications

```
Successfully run. Total query runtime: 48 secs 363 msec.
75 rows affected.
```

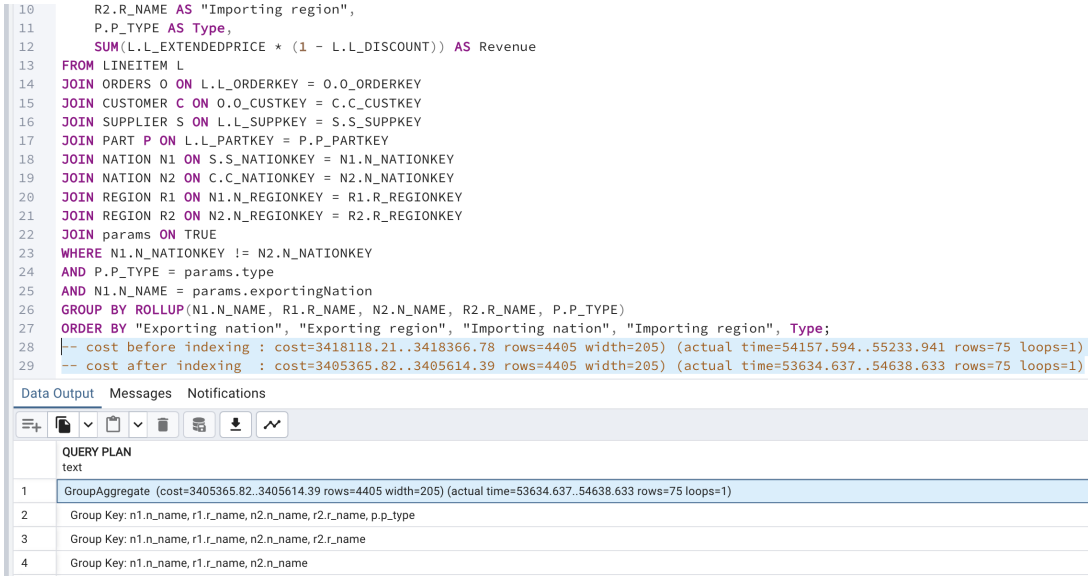
Figure 35: Region and Nation Aggregation Query Execution Time Reduction

The cost of the of Region and Nation Aggregation query has been **reduced from**

3,418,118.21 to 3,405,365.82 due to the indexes created.

Additionally, to view the script for calculating the query cost, use the following path: `/BigData/SQL SCRIPTS/COST_QUERY3.sql`

More data is shown in the picture below.



```
    R2.R_NAME AS "Importing region",
    P.P_TYPE AS Type,
    SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
FROM LINEITEM L
JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
JOIN REGION R1 ON N1.N_REGIONKEY = R1.R_REGIONKEY
JOIN REGION R2 ON N2.N_REGIONKEY = R2.R_REGIONKEY
JOIN params ON TRUE
WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
AND P.P_TYPE = params.type
AND N1.N_NAME = params.exportingNation
GROUP BY ROLLUP(N1.N_NAME, R1.R_NAME, N2.N_NAME, R2.R_NAME, P.P_TYPE)
ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
-- cost before indexing : cost=3418118.21..3418366.78 rows=4405 width=205) (actual time=54157.594..55233.941 rows=75 loops=1)
-- cost after indexing  : cost=3405365.82..3405614.39 rows=4405 width=205) (actual time=53634.637..54638.633 rows=75 loops=1)
```

Data Output | Messages | Notifications

	QUERY PLAN text
1	GroupAggregate (cost=3405365.82..3405614.39 rows=4405 width=205) (actual time=53634.637..54638.633 rows=75 loops=1)
2	Group Key: n1.n_name, r1.r_name, n2.n_name, r2.r_name, p.p_type
3	Group Key: n1.n_name, r1.r_name, n2.n_name, r2.r_name
4	Group Key: n1.n_name, r1.r_name, n2.n_name

Figure 36: f Region and Nation Aggregation Cost Query Result

Part VI
Design of Materialization Query Cost with Materialized Views Without Indexing

In this chapter, I focus on creating and using materialized views to improve query performance. Materialized views store pre-computed complex query results, which speeds up later queries. Initially, I tried a materialized view with the `ROLLUP` function to pre-aggregate data, but it produced fewer rows than expected due to filtering issues. To fix this, I implemented a different materialized view that stores detailed data without pre-aggregation, ensuring accurate filtering and aggregation during queries. This chapter explains how these materialized views were created and applied to various query types (time-based, type-based, and geographical aggregations). It also analyzes query costs and execution times after using materialized views, showing significant performance improvements and demonstrating their effectiveness in speeding up queries and enhancing overall database efficiency.

8 Design of Materialization

In this section, I will create materialized views for each of the queries in Query Schema 1 to optimize their execution times.

I created a materialized view to optimize the queries that aggregates revenue by month, quarter, and year and aggregate by type. Initially, I considered a materialized view using the `ROLLUP` function directly within the view definition to pre-aggregate data. However, **this approach resulted in fewer rows than expected (e. g 2058 Rows instead of 2249 Rows for time aggreagation query)**when filtering by specific parameters (`Type` and `ExportingNation`), **which indicated that the pre-aggregation was causing discrepancies.**

To address this issue, I designed an alternative materialized view that stores the detailed data without pre-aggregating using `ROLLUP`. This approach allows for accurate filtering and aggregation at query time, ensuring that the results match the expected output.

Here is the detailed implementation:

8.1 Initial Materialized View (Discarded due to lack of rows)

This first materialized view, `MV_time_agg`, was created with pre-aggregation using the ROLLUP function. This resulted in fewer rows than expected due to discrepancies in filtering by specific parameters.

Query | Query History

```sql
CREATE MATERIALIZED VIEW MV_time_agg AS
SELECT
    EXTRACT(YEAR FROM L.L_SHIPDATE) AS Year,
    EXTRACT(QUARTER FROM L.L_SHIPDATE) AS Quarter,
    EXTRACT(MONTH FROM L.L_SHIPDATE) AS Month,
    P.P_TYPE AS Type,
    N1.N_NAME AS ExportingNation,
    N2.N_NAME AS ImportingNation,
    SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
FROM LINEITEM L
JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
GROUP BY
    ROLLUP(
        EXTRACT(MONTH FROM L.L_SHIPDATE),
        EXTRACT(QUARTER FROM L.L_SHIPDATE),
        EXTRACT(YEAR FROM L.L_SHIPDATE),
        P.P_TYPE,
        N1.N_NAME,
        N2.N_NAME
    )
ORDER BY Year, Quarter, Month, Type, ExportingNation, ImportingNation;
-- the query using the view
SELECT *
FROM MV_time_agg
WHERE Type = 'STANDARD POLISHED STEEL'
AND ExportingNation = 'MOROCCO'
ORDER BY Year, Quarter, Month, Type, ExportingNation, ImportingNation;
```

Data Output | Messages | Notifications

	year numeric	quarter numeric	month numeric	type character varying (25)	exportingnation character varying (25)	importingnation character varying (25)	revenue numeric
1	1992	1	1	STANDARD POLISHED STEEL	MOROCCO	ALGERIA	97910.8592

Total rows: 1000 of 2058 | Query complete 00:00:00.320

Figure 37: Initial Materialized View (Discarded due to lack of rows)

8.2 Corrected Materialized View

To resolve the issue, I created a materialized view that stores the detailed data and applied aggregation and filtering dynamically at query time.

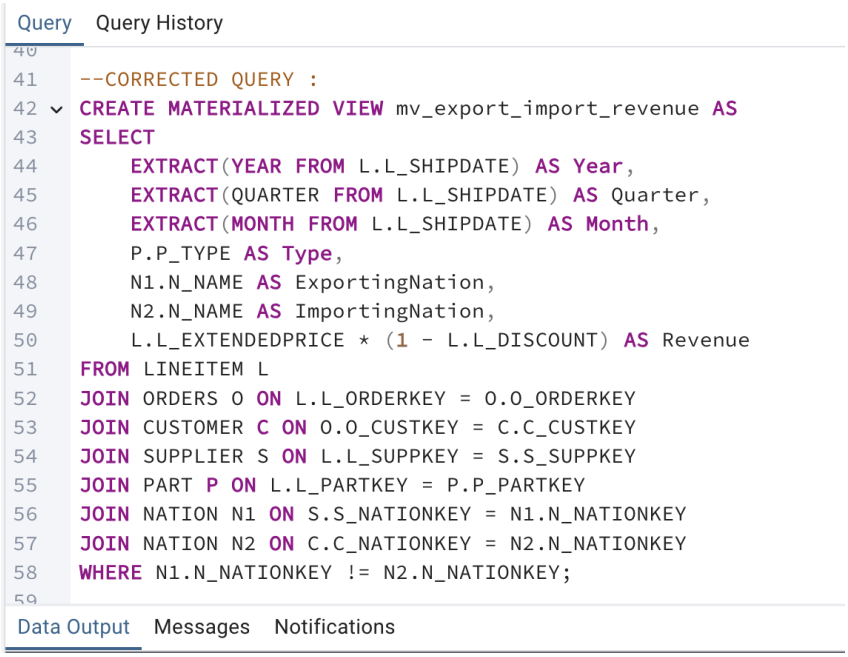


Figure 38: Corrected Materialized View

8.3 Re Execution of the Queries Using the Materialized View

The corrected created previews view can be used for both first aggregations , the time aggregation and type aggregation because they share the same joined table in the view .

The queries will apply the necessary filtering and roll-up aggregation dynamically to ensure accurate results.

8.3.1 Time Base Aggregation Query Using the Materialized View

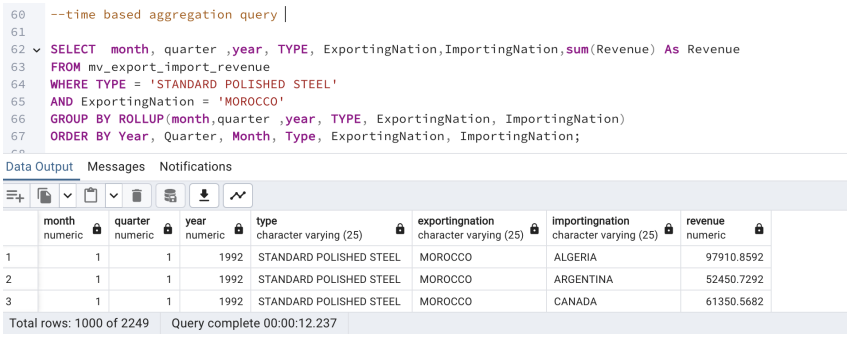


Figure 39: Time Base Aggregation Query Using the Materialized View

8.3.2 Type Base Aggregation Query Using the Materialized View

```
--TYPE BASED aggregation QUERY

SELECT type, ExportingNation, ImportingNation,sum(Revenue) As Revenue
FROM mv_export_import_revenue
WHERE type = 'ECONOMY ANODIZED BRASS'
AND ExportingNation = 'CANADA'
GROUP BY ROLLUP(
            TYPE,
            ExportingNation,
            ImportingNation)
ORDER BY Type, ExportingNation, ImportingNation;
```

Data Output | Messages | Notifications

	type character varying (25)	exportingnation character varying (25)	importingnation character varying (25)	revenue numeric
1	ECONOMY ANODIZED BRASS	CANADA	ALGERIA	21063762.1642
2	ECONOMY ANODIZED BRASS	CANADA	ARGENTINA	23305153.3875
3	ECONOMY ANODIZED BRASS	CANADA	BRAZIL	22761889.5684

Total rows: 27 of 27 | Query complete 00:00:13.017

Figure 40: Type Base Aggregation Query Using the Materialized View

By following this approach, the materialized view contains the detailed data needed to perform accurate and efficient roll-up aggregations and filtering dynamically at query time. **This ensures that the results align with the original query expectations** , optimizing execution times while maintaining accuracy.

8.4 Second View : Region and Nation-Based

To further optimize queries involving Region and Nation Aggregation, I created another materialized view. This view includes the necessary joins and detailed data without pre-aggregation using `ROLLUP`.

Query | Query History

```
CREATE MATERIALIZED VIEW mv_export_import_revenue_region AS
SELECT
    N1.N_NAME AS ExportingNation,
    R1.R_NAME AS ExportingRegion,
    N2.N_NAME AS ImportingNation,
    R2.R_NAME AS ImportingRegion,
    P.P_TYPE AS Type,
    L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT) AS Revenue
FROM LINEITEM L
JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
JOIN REGION R1 ON N1.N_REGIONKEY = R1.R_REGIONKEY
JOIN REGION R2 ON N2.N_REGIONKEY = R2.R_REGIONKEY
WHERE N1.N_NATIONKEY != N2.N_NATIONKEY;
```

Data Output | Messages | Notifications

Figure 41: Second View : Region and Nation-Based

8.4.1 Re-Execution of the Third Query Using the Region and Nation Materialized View

The following query applies the necessary filtering and roll-up aggregation dynamically to ensure accurate results:

```
--QUERY
SELECT
    ExportingNation AS "Exporting nation",
    ExportingRegion AS "Exporting region",
    ImportingNation AS "Importing nation",
    ImportingRegion AS "Importing region",
    Type,
    SUM(Revenue) AS Revenue
FROM mv_export_import_revenue_region

WHERE Type = 'ECONOMY BRUSHED STEEL'
AND ExportingNation = 'ARGENTINA'
GROUP BY ROLLUP(ExportingNation, ExportingRegion, ImportingNation, ImportingRegion, Type)
ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
```

Data Output | Messages | Notifications

	Exporting nation character varying (25)	Exporting region character varying (25)	Importing nation character varying (25)	Importing region character varying (25)	type character varying (25)	revenue numeric
1	ARGENTINA	AMERICA	ALGERIA	AFRICA	ECONOMY BRUSHED STEEL	22405660.6732
2	ARGENTINA	AMERICA	ALGERIA	AFRICA	[null]	22405660.6732

Total rows: 75 of 75 | Query complete 00:00:20.454

Figure 42: Third Query Using the Region and Nation Materialized View

This query returns the correct number of rows, aligning with the expectations and ensuring that the results are accurate and optimized for performance.

9 Query Cost of Queries with Materialized Views

In this section, I will analyze the query costs after creating materialized views for the queries in Query Schema 1. The goal is to demonstrate the performance improvements achieved by using materialized views.

9.1 Query Cost of Time-Based Aggregation using Materialized View

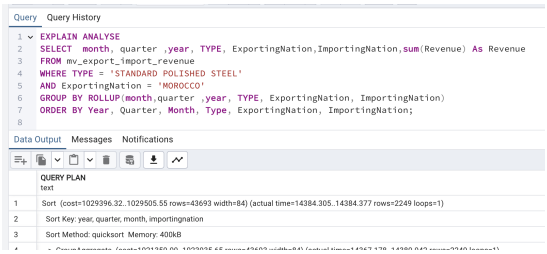


Figure 43: Query Cost using Materialized View

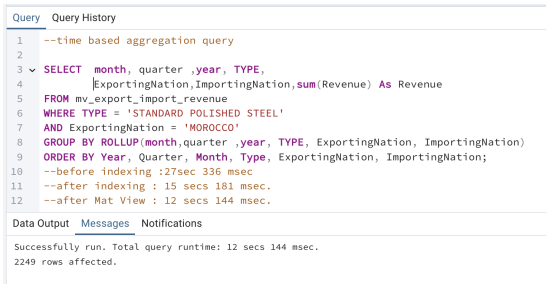


Figure 44: Execution Time using Materialized View

- The cost of sorting ranges from **1,029,396.32 to 1,029,505.55**. Sorting takes **14,384.305 to 14,384.377 milliseconds** and processes **2,249 rows**.
- Total query execution time: **12 seconds 144 milliseconds**.

Execution time improvements:

- **Before indexing:** 27 seconds 336 milliseconds
- **After indexing:** 15 seconds 181 milliseconds
- **After using Materialized View: 12 seconds 144 milliseconds**

9.2 Query Cost of Type-Based Aggregation using Materialized View

- **Cost of grouping operation:** Ranges from **1,021,368.81 to 1,023,553.87**. The actual time taken for grouping is between **12,584.592 and 12,590.968 milliseconds**, processing **27 rows**.
- **Total query execution time: 14 seconds 618 milliseconds**.

Execution time improvements:

- **Before indexing:** 48 seconds 560 milliseconds
- **After indexing:** 45 seconds 591 milliseconds
- **After using Materialized View:** 14 seconds 618 milliseconds

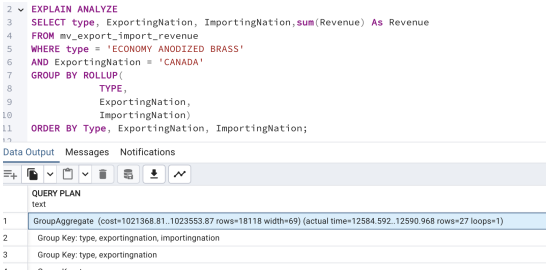


Figure 45: Query Cost of Type-Based Aggregation Query using Materialized View

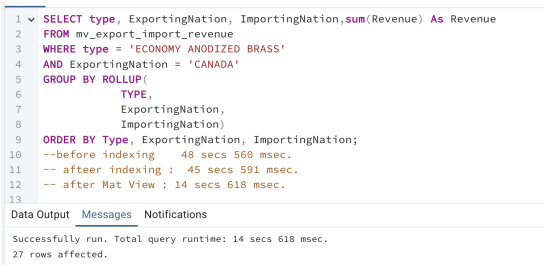


Figure 46: Execution Time of Type-Based Aggregation Query using Materialized View

9.3 Query Cost of Region and Nation Aggregation using Materialized View

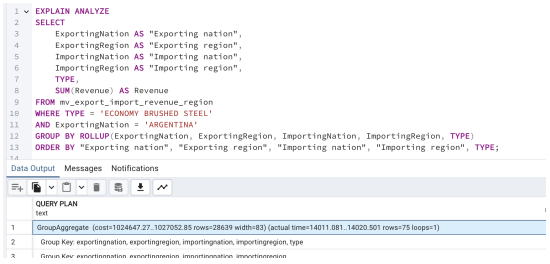


Figure 47: Query Cost of Region and Nation Aggregation using Materialized View

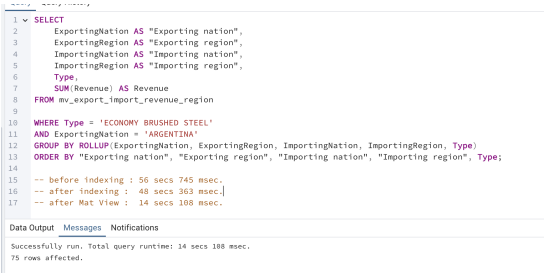


Figure 48: Execution Time of Region and Nation Aggregation using Materialized View

- **Grouping operation cost:** Ranges from **1,024,647.27 to 1,027,052.85**. The actual time taken for grouping is between **14,011.081 and 14,020.501 milliseconds**, processing **75 rows**.
- **Total query execution time: 14 seconds 108 milliseconds**.

Execution time improvements:

- **Before indexing:** 56 seconds 745 milliseconds
- **After indexing:** 48 seconds 363 milliseconds
- **After using Materialized View:** 14 seconds 108 milliseconds

Part VII
Indexing on Materialized Views and The Query Cost with Indexed Materialized Views

In this chapter, I enhance query optimization by adding indexes to materialized views. While materialized views already improve query performance by pre-computing and storing complex query results, indexing them can make them even more efficient. Indexing reduces search times and optimizes data retrieval. This section explains how I created composite indexes on materialized views for different query scenarios, like export/import revenue aggregation by time, type, and geography. By implementing these indexes, I aim to significantly reduce query execution times and costs. After creating the indexes, I analyze performance improvements through detailed cost and time evaluations, demonstrating the effectiveness of combining materialized views with indexing in large-scale data processing.

10 Indexing on Materialized Views

10.1 Creation of Indexes

10.1.1 Indexing on The Materialized View `mv_export_import_revenue`

I created The composite index that cover Type, ExportingNation, and the ROLLUP columns (Year, Quarter, Month).

The second composite index cover Type, ExportingNation, and ImportingNation.

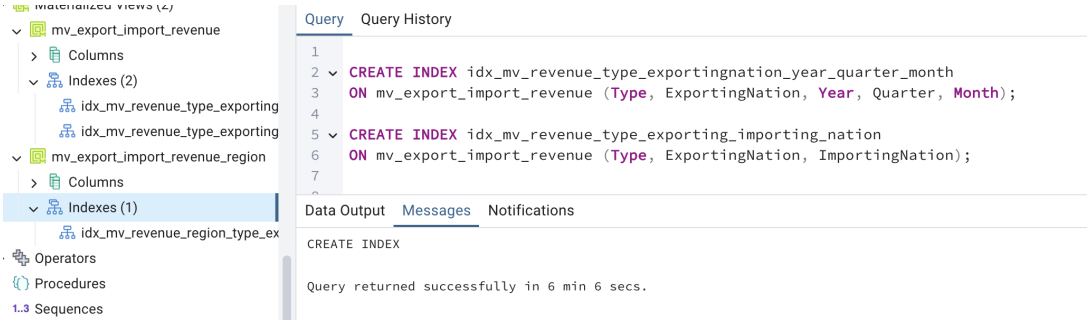


Figure 49: Indexing on The Materialized View `mv_export_import_revenue`

10.1.2 Indexing on The Materialized View `mv_export_import_revenue_region`

I created The composite index that cover Type, ExportingNation, ExportingRegion, ImportingNation, and ImportingRegion.

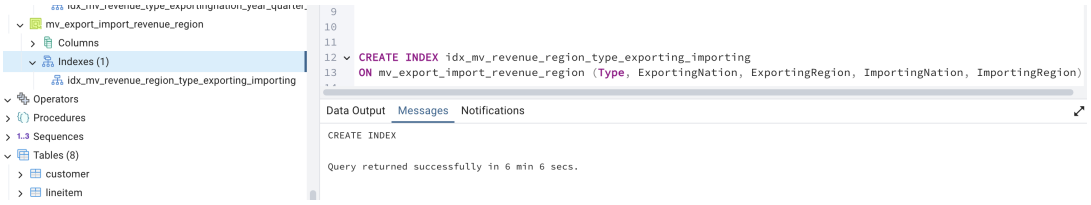


Figure 50: Indexing on The Materialized View `mv_export_import_revenue_region`

11 Query Cost of Queries with Materialized Views and Indexing

After implementing materialized views and applying indexing, I observed significant performance improvements. The execution times of the queries were drastically reduced, demonstrating the effectiveness of these optimization techniques. Below, I provide detailed analyses of the query costs and execution times for various aggregation types.

11.1 Query Cost of Time-Based Aggregation Using Materialized View and Indexing

Combining materialized views with indexing resulted in an impressive reduction to just **116 milliseconds**. This demonstrates the significant impact of these optimization techniques in enhancing query performance.

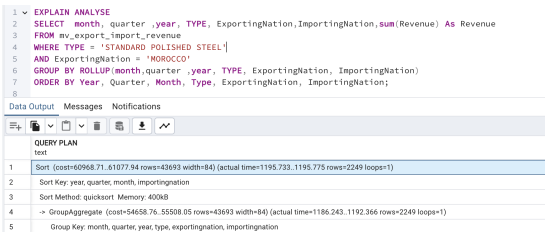


Figure 51: Query Cost Using Materialized View and Indexing

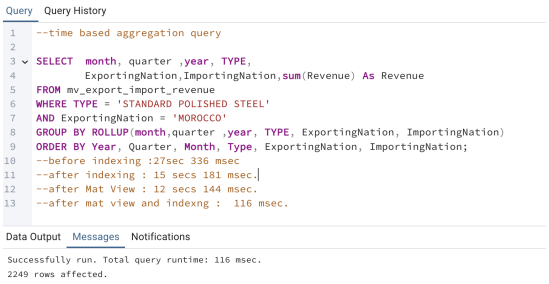


Figure 52: Execution Time Using Materialized View and Indexing

The figure 51 shows the query cost when using the materialized view and indexing. The detailed cost breakdown is as follows:

- The cost of sorting ranges from **60,968.71 to 61,077.94**. Sorting takes **1,195.733 to 1,195.775 milliseconds** and processes **2,249 rows**.
- Total query execution time: **116 milliseconds**.

The figure 52 highlights the execution time improvements achieved at various optimization stages:

- **Before Indexing**: 27 seconds 336 milliseconds
- **After Indexing**: 15 seconds 181 milliseconds
- **After Materialized View**: 12 seconds 144 milliseconds
- **After Materialized View and Indexing**: 116 milliseconds

11.2 Query Cost of Type-Based Aggregation Using Materialized View and Indexing

The application of materialized views and indexing led to a remarkable reduction in execution time, bringing it down to just **49 milliseconds**. This clearly demonstrates the effectiveness of these optimizations.

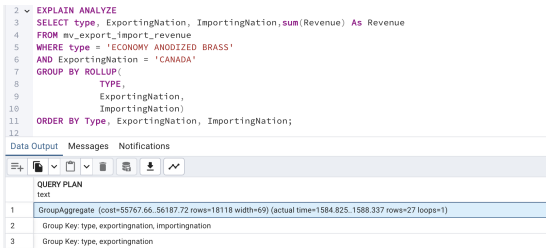


Figure 53: Query Cost Using Materialized View and Indexing

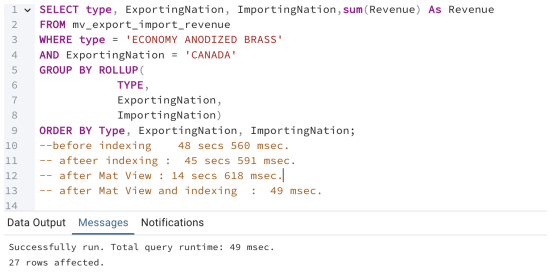


Figure 54: Execution Time Using Materialized View and Indexing

The figure 53 depicts the query cost with the materialized view and indexing. The detailed cost breakdown is as follows:

- The cost of grouping ranges from **55,767.66 to 56,187.72**. Grouping takes **1,584.825 to 1,588.337 milliseconds** and processes **27 rows**.
- Total query execution time: **49 milliseconds**.

The figure 54 illustrates the execution time improvements achieved at various optimization stages:

- **Before Indexing**: 48 seconds 560 milliseconds
- **After Indexing**: 45 seconds 591 milliseconds
- **After Materialized View**: 14 seconds 618 milliseconds
- **After Materialized View and Indexing**: 49 milliseconds

11.3 Query Cost of Nation and Region Aggregation Using Materialized View and Indexing

Applying materialized views along with indexing resulted in an outstanding reduction in execution time, reducing it to just **61 milliseconds**. This highlights the substantial performance improvements gained through these optimizations.

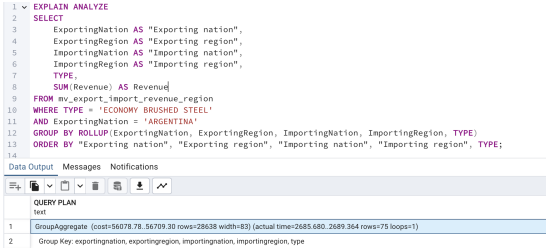


Figure 55: Query Cost Using Materialized View and Indexing

Figure 56: Execution Time Using Materialized View and Indexing

The figure 55 demonstrates the query cost with the materialized view and indexing. The detailed cost breakdown is as follows:

- The cost of grouping ranges from **56,078.78 to 56,709.30**. Grouping takes **2,685.680 to 2,689.364 milliseconds** and processes **75 rows**.
- Total query execution time: **61 milliseconds**.

The figure 56 shows the execution time improvements achieved at various optimization stages:

- **Before Indexing**: 56 seconds 745 milliseconds
- **After Indexing**: 48 seconds 363 milliseconds
- **After Materialized View**: 14 seconds 108 milliseconds
- **After Materialized View and Indexing**: 61 milliseconds

Part VIII
Recap of the Final Optimization Strategy Compared with the Initial Query Cost

This chapter provides a summary of the final optimization strategies and their impact on query performance. It compares the initial and optimized query costs and execution times for three key queries: time-based aggregation, type-based aggregation, and nation and region-based aggregation. The optimizations involved strategic use of indexing and materialized views, greatly improving query efficiency. This section highlights the significant improvements achieved through these techniques, showing how combining indexing with materialized views effectively reduces both execution times and costs, thereby optimizing the performance of decision support systems using PostgreSQL.

12 Query Cost and Execution Time Comparison Overall Stages

This section provides a summary of the final optimization strategy and compares the initial and optimized query costs and execution times for the three queries. The optimizations included the application of indexing and the use of materialized views, which significantly improved query performance.

12.1 Query 1: Time-Based Aggregation

Execution Time:

- **Before Indexing:** 27 seconds 336 milliseconds
- **After Indexing:** 15 seconds 181 milliseconds
- **After Materialized View:** 12 seconds 144 milliseconds
- **After Materialized View and Indexing:** 116 milliseconds

Query Cost:

- **Before Indexing:** 1,582,234.21 to 1,582,366.48
- **After Indexing:** Reduced to 937,902.40
- **After Materialized View:** 1,029,396.32
- **After Materialized View and Indexing:** 60,968.71 to 61,077.94

The optimizations applied to Query 1 show a drastic reduction in both execution time and query cost. The use of materialized views combined with indexing reduced the execution time from over 27 seconds to just 116 milliseconds and the query cost from over 1.5 million to approximately 61,000.

12.2 Query 2: Type-Based Aggregation

Execution Time:

- **Before Indexing:** 48 seconds 560 milliseconds
- **After Indexing:** 45 seconds 591 milliseconds
- **After Materialized View:** 14 seconds 618 milliseconds
- **After Materialized View and Indexing:** 49 milliseconds

Query Cost:

- **Before Indexing:** 3,228,022.28 to 3,228,246.04
- **After Indexing:** Reduced to 3,215,269.89
- **After Materialized View:** 1,021,368.81

- **After Materialized View and Indexing:** 55,767.66 to 56,187.72

For Query 2, the execution time was significantly reduced from nearly 49 seconds to 49 milliseconds after applying the final optimization. The query cost decreased from over 3.2 million to about 56,000.

12.3 Query 3: Nation and Region-Based Aggregation

Execution Time:

- **Before Indexing:** 56 seconds 745 milliseconds
- **After Indexing:** 48 seconds 363 milliseconds
- **After Materialized View:** 14 seconds 108 milliseconds
- **After Materialized View and Indexing:** 61 milliseconds

Query Cost:

- **Before Indexing:** 3,418,118.21 to 3,418,366.78
- **After Indexing:** Reduced to 3,405,365.82
- **After Materialized View:** 1,024,647.27
- **After Materialized View and Indexing:** 56,078.78 to 56,709.30

Query 3 saw a substantial improvement with the execution time dropping from over 56 seconds to just 61 milliseconds. The query cost was reduced from over 3.4 million to about 56,000.

13 Space Constraint Verification

The final optimization must fulfill the space constraint that the amount of space for indexes and materialized views must be less than 1.5 times the size of the database.

To verify this, the following SQL queries were executed:

Explanation:

- **Total Size of Indexes:** 12 GB
- **Total Size of Materialized Views:** 15 GB
- **Total Size of the Database:** 35 GB
- **1.5 Times the Size of the Database:** 52 GB

The total size of indexes and materialized views (12 GB + 15 GB = 27 GB) is less than 1.5 times the size of the database (52 GB). Therefore, **the space constraint is satisfied.**

Query | Query History

```sql
SELECT pg_size_pretty(pg_database_size('TPCH-Project')) AS total_db_size;
WITH index_size AS (
    SELECT sum(pg_relation_size(indexrelid::regclass)) AS total_index_size
    FROM pg_index
),
matview_size AS (
    SELECT sum(pg_total_relation_size(c.oid)) AS total_matview_size
    FROM pg_class c
    JOIN pg_namespace n ON n.oid = c.relnamespace
    WHERE c.relkind = 'm'
),
db_size AS (
    SELECT pg_database_size('TPCH-Project') AS total_db_size
)
SELECT
    pg_size_pretty(i.total_index_size) AS total_index_size,
    pg_size_pretty(m.total_matview_size) AS total_matview_size,
    pg_size_pretty(d.total_db_size) AS total_db_size,
    pg_size_pretty(d.total_db_size * 1.5) AS max_allowed_size,
    pg_size_pretty(i.total_index_size + m.total_matview_size) AS total_used_size,
    CASE
            WHEN i.total_index_size + m.total_matview_size < d.total_db_size * 1.5
            THEN 'Space constraint satisfied'
            ELSE 'Space constraint not satisfied'
    END AS constraint_status
FROM index_size i, matview_size m, db_size d;
```

Data Output | Messages | Notifications

	total_index_size text	total_matview_size text	total_db_size text	max_allowed_size text	total_used_size text	constraint_status text
1	12 GB	15 GB	35 GB	52 GB	27 GB	Space constraint satisfied

Figure 57: Space Constraint Final Verification

14 Summary

The final optimization strategy, which included the use of materialized views and indexing, led to significant improvements in both query execution time and cost. The optimizations reduced the execution times by several orders of magnitude and drastically lowered the query costs. This highlights the importance and effectiveness of these optimization techniques in enhancing database performance.

Overall Execution Time Improvements:

- Query 1: From 27 seconds 336 milliseconds to 116 milliseconds (99.57% reduction)
- Query 2: From 48 seconds 560 milliseconds to 49 milliseconds (99.90% reduction)
- Query 3: From 56 seconds 745 milliseconds to 61 milliseconds (99.89% reduction)

Overall Query Cost Improvements:

- Query 1: From 1,582,234.21 to 60,968.71 (96.15% reduction)
- Query 2: From 3,228,022.28 to 55,767.66 (98.27% reduction)
- Query 3: From 3,418,118.21 to 56,078.78 (98.36% reduction)

These optimizations demonstrate the substantial performance gains that can be achieved through careful indexing and the use of materialized views. The total size of indexes and materialized views (12 GB + 15 GB = 27 GB) is less than 1.5 times the size of the database (52 GB).**the space constraint is satisfied.**

Part IX
Conclusion

This project successfully implemented and optimized the TPC-H benchmark for decision support systems using PostgreSQL, encompassing the creation of a comprehensive database schema, data population, statistical analysis, and advanced query optimization techniques. Initially, a detailed schema was established and populated with data generated by the TPC-H `dbgen` tool, followed by an in-depth statistical analysis to understand the data distribution and characteristics. Three critical queries were implemented—focusing on time-based, type-based, and nation and region-based aggregations—highlighting initial performance metrics. Significant optimizations were achieved through strategic indexing and the use of materialized views, resulting in dramatic reductions in both query execution times and costs. For instance, the time-based aggregation query execution time was reduced from over 27 seconds to just 116 milliseconds, demonstrating a 99.57% improvement. Similarly, the type-based and nation and region-based aggregation queries saw reductions of 99.90% and 99.89%, respectively. The comprehensive approach, which included verifying space constraints for indexes and materialized views, underscores the effectiveness of these optimization strategies in managing large datasets and complex queries efficiently. This project exemplifies the critical role of database design, indexing, and materialized views in enhancing the performance of decision support systems, providing a robust framework that can be adapted for similar optimization tasks in large-scale data environments.

Finnaly , The total size of indexes and materialized views (12 GB + 15 GB = 27 GB) is less than 1.5 times the size of the database (52 GB). the space constraint is satisfied.