



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

I	Introduction	4
II	Database and Schema Creation	5
1	Database and Schema Creation	6
1.1	Database Creation and Connection	6
1.2	SQL Definition of the Tables and Constraints	6
1.2.1	Creating the REGION Table	6
1.2.2	Creating the NATION Table	7
1.2.3	Creating the PART Table	8
1.2.4	Creating the SUPPLIER Table	9
1.2.5	Creating the PARTSUPP Table	10
1.2.6	Creating the CUSTOMER Table	11
1.2.7	Creating the ORDERS Table	12
1.2.8	Creating the LINEITEM Table	13
III	Data Population and Statistical Analysis	15
2	Data Population	16
2.1	Data Generation	16
2.2	Data Conversion from .tbl to .csv	16
2.3	Populating the DB using data generated by dbgen CSV files	17
3	Statistics of the Data	18
3.1	Size of the Database	19
3.2	Number of Rows in Each Table	19
3.3	Table Sizes in Bytes	19
3.4	Distinct Values for Relevant Attributes	21
3.5	Minimum and Maximum Values for Relevant Attributes	22
IV	Query Schema and Analysis Before Optimization	23
4	the Query Schema 1: Export/Import Revenue Value	24
4.1	Objective	24
4.2	Revenue Calculation	24
4.3	Aggregations And Slicing :	24
4.4	Queries :	25
4.4.1	Time-Based Aggregation	25
4.4.2	Type-Based Aggregation	26
4.4.3	Geographical Aggregation	27

5 Query Cost before Optimization	29
5.1 Query Cost of Time-Based Aggregation	29
5.2 Query Cost of Type Aggregation	29
5.3 Query Cost of Region and Nation Aggregation	30
V Index Design and Query Cost with Indexes	31
6 Design of indexes	32
6.1 Creation of Indexes on Tables	32
6.2 Space Cost of Indexes	32
6.3 Space Constraint Verification	33
7 Query Cost of Queries with Indexes	34
7.1 Query Cost of Time-Based Aggregation Query with Indexing	34
7.2 Query Cost of Type-Based Aggregation Query with Indexing	34
7.3 Query Cost of Region and Nation Aggregation Query with Indexing	36
VI Design of Materialization Query Cost with Materialized Views Without Indexing	39
8 Design of Materialization	40
8.1 Initial Materialized View (Discarded due to lack of rows)	40
8.2 Corrected Materialized View	41
8.3 Re Execution of the Queries Using the Materialized View	41
8.3.1 Time Base Aggregation Query Using the Materialized View	41
8.3.2 Type Base Aggregation Query Using the Materialized View	42
8.4 Second View : Region and Nation-Based	42
8.4.1 Re-Execution of the Third Query Using the Region and Nation Materialized View	43
9 Query Cost of Queries with Materialized Views	44
9.1 Query Cost of Time-Based Aggregation using Materialized View	44
9.2 Query Cost of Type-Based Aggregation using Materialized View	44
9.3 Query Cost of Region and Nation Aggregation using Materialized View	45
VII Indexing on Materialized Views and The Query Cost with Indexed Materialized Views	46
10 Indexing on Materialized Views	47
10.1 Creation of Indexes	47
10.1.1 Indexing on The Materialized View <code>mv_export_import_revenue</code>	47
10.1.2 Indexing on The Materialized View <code>mv_export_import_revenue_region</code>	47
11 Query Cost of Queries with Materialized Views and Indexing	48
11.1 Query Cost of Time-Based Aggregation Using Materialized View and Indexing	48

11.2 Query Cost of Type-Based Aggregation Using Materialized View and Indexing	48
11.3 Query Cost of Nation and Region Aggregation Using Materialized View and Indexing	49
VIII Recap of the Final Optimization Strategy Compared with the Initial Query Cost	51
12 Query Cost and Execution Time Comparison Overall Stages	52
12.1 Query 1: Time-Based Aggregation	52
12.2 Query 2: Type-Based Aggregation	52
12.3 Query 3: Nation and Region-Based Aggregation	53
13 Space Constraint Verification	53
14 Summary	55
IX Conclusion	56

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. **The 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:

```
wafaabajjouk@Computeortatile ~ % psql -U wafaabajjouk -d TPCH-Project
psql (14.12 (Homebrew))
Type "help" for help.

TPCH-Project=#
```

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)
);

```

The screenshot shows a PostgreSQL terminal window titled 'tpch/postgres@Bajjouk'. The query tab contains the SQL code for creating the REGION table, which includes constraints for the primary key and a non-negative check. The messages tab shows the execution results, indicating that the table was created successfully and the query returned in 225 msec.

```

-- Create REGION Table with constraints
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)
);

CREATE TABLE

Query returned successfully in 225 msec.

```

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)
);

```

The screenshot shows a database query interface with the following details:

- Query History:** The user has run a query to create the NATION table.
- Code:**

```

1 v | CREATE TABLE NATION (
2     N_NATIONKEY INTEGER NOT NULL PRIMARY KEY,
3     N_NAME VARCHAR(25) NOT NULL,
4     N_REGIONKEY INTEGER NOT NULL,
5     N_COMMENT VARCHAR(152),
6     FOREIGN KEY (N_REGIONKEY) REFERENCES REGION(R_REGIONKEY),
7     CONSTRAINT check_pk_nation CHECK (N_NATIONKEY >= 0)
8 );
9

```
- Messages:** The message indicates that the query was successful.
- Output:**

```

CREATE TABLE

Query returned successfully in 232 msec.

```

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)
);

```

```

Query  Query History

1 ✓ CREATE TABLE PART (
2     P_PARTKEY INTEGER NOT NULL PRIMARY KEY,
3     P_NAME VARCHAR(55),
4     P_MFGR CHAR(25),
5     P_BRAND CHAR(10),
6     P_TYPE VARCHAR(25),
7     P_SIZE INTEGER NOT NULL CHECK (P_SIZE >= 0),
8     P_CONTAINER CHAR(10),
9     P_RETAILPRICE DECIMAL NOT NULL CHECK (P_RETAILPRICE >= 0),
10    P_COMMENT VARCHAR(23),
11    CONSTRAINT check_pk_part CHECK (P_PARTKEY >= 0)
12 );
Data Output  Messages  Notifications

CREATE TABLE

Query returned successfully in 86 msec.

```

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)

);

```

The screenshot shows a PostgreSQL client window with the following details:

- Connection:** tpch/postgres@Bajjouk
- Toolbar:** Includes icons for file operations, search, and various database management functions.
- Query Tab:** Active tab, showing the SQL code for creating the SUPPLIER table.
- Code:**

```

1  CREATE TABLE SUPPLIER (
2      S_SUPPKEY INTEGER NOT NULL PRIMARY KEY,
3      S_NAME CHAR(25),
4      S_ADDRESS VARCHAR(40),
5      S_NATIONKEY INTEGER NOT NULL,
6      S_PHONE CHAR(15),
7      S_ACCTBAL DECIMAL,
8      S_COMMENT VARCHAR(101),
9      FOREIGN KEY (S_NATIONKEY) REFERENCES NATION(N_NATIONKEY),
10     CONSTRAINT check_pk_supplier CHECK ( S_SUPPKEY >= 0)
11 );

```
- Data Output Tab:** Shows the results of the query execution.
- Messages Tab:** Active tab, showing the message: "CREATE TABLE".
- Notifications Tab:**
- Message Content:** "Query returned successfully in 100 msec."

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)
);

```

The screenshot shows the pgAdmin 4 interface with the following details:

- Connection:** tpch/postgres@Bajjouk
- Toolbar:** Includes icons for file operations, search, and various database management functions.
- Query History:** Shows the SQL command used to create the table.
- Code Block (Line 13):**

```

1 CREATE TABLE PARTSUPP (
2     PS_PARTKEY INTEGER NOT NULL,
3     PS_SUPPKEY INTEGER NOT NULL,
4     PS_AVAILQTY INTEGER NOT NULL CHECK (PS_AVAILQTY >= 0),
5     PS_SUPPLYCOST DECIMAL NOT NULL CHECK (PS_SUPPLYCOST >= 0),
6     PS_COMMENT VARCHAR(199),
7     PRIMARY KEY (PS_PARTKEY, PS_SUPPKEY),
8     FOREIGN KEY (PS_PARTKEY) REFERENCES PART(P_PARTKEY),
9     FOREIGN KEY (PS_SUPPKEY) REFERENCES SUPPLIER(S_SUPPKEY),
10    CONSTRAINT check_pk_ps_partkey CHECK (PS_PARTKEY >= 0),
11    CONSTRAINT check_pk_ps_suppkey CHECK (PS_SUPPKEY >= 0)
12 );
13

```
- Data Output:** Shows the command executed: "CREATE TABLE".
- Messages:** Shows the result: "Query returned successfully in 228 msec."

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) );

```

The screenshot shows the Oracle SQL Developer interface. The 'Query' tab is selected, displaying the SQL code for creating the CUSTOMER table. The code includes columns for CUSTKEY, NAME, ADDRESS, NATIONKEY, PHONE, ACCTBAL, MKTSEGMENT, and COMMENT, with constraints like NOT NULL, PRIMARY KEY, FOREIGN KEY, and a CHECK constraint for ACCTBAL. Below the code, the 'Messages' tab is selected, showing the successful execution of the command and a completion message.

```

1 v CREATE TABLE CUSTOMER (
2     C_CUSTKEY INTEGER NOT NULL PRIMARY KEY,
3     C_NAME VARCHAR(25),
4     C_ADDRESS VARCHAR(40),
5     C_NATIONKEY INTEGER NOT NULL,
6     C_PHONE CHAR(15),
7     C_ACCTBAL DECIMAL,
8     C_MKTSEGMENT CHAR(10),
9     C_COMMENT VARCHAR(117),
10    FOREIGN KEY (C_NATIONKEY) REFERENCES NATION(N_NATIONKEY),
11    CONSTRAINT check_positive_custkey CHECK (C_CUSTKEY >= 0)
12 );

```

Data Output Messages Notifications

CREATE TABLE

Query returned successfully in 60 msec.

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)
);

```

The screenshot shows a PostgreSQL database client interface. The connection is set to 'tpch/postgres@Bajjouk'. The toolbar includes various icons for file operations, search, and navigation. The main area displays a SQL query for creating the 'ORDERS' table:

```

1 < CREATE TABLE ORDERS (
2   O_ORDERKEY INTEGER NOT NULL PRIMARY KEY,
3   O_CUSTKEY INTEGER NOT NULL,
4   O_ORDERSTATUS CHAR(1),
5   O_TOTALPRICE DECIMAL NOT NULL CHECK (O_TOTALPRICE >= 0),
6   O_ORDERDATE DATE,
7   O_ORDERPRIORITY CHAR(15),
8   O_CLERK CHAR(15),
9   O_SHIPPRIORITY INTEGER,
10  O_COMMENT VARCHAR(79),
11  FOREIGN KEY (O_CUSTKEY) REFERENCES CUSTOMER(C_CUSTKEY),
12  CONSTRAINT check_pk_orderkey CHECK (O_ORDERKEY >= 0)
13 );
14

```

Below the query, the status bar shows 'CREATE TABLE' and 'Query returned successfully in 64 msec.'

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
);

```

Query History

```

1 ✓ CREATE TABLE LINEITEM (
2     L_ORDERKEY INTEGER NOT NULL,
3     L_PARTKEY INTEGER NOT NULL,
4     L_SUPPKEY INTEGER NOT NULL,
5     L_LINENUMBER INTEGER NOT NULL,
6     L_QUANTITY DECIMAL NOT NULL CHECK (L_QUANTITY >= 0),
7     L_EXTENDEDPRICE DECIMAL NOT NULL CHECK (L_EXTENDEDPRICE >= 0),
8     L_DISCOUNT DECIMAL NOT NULL CHECK (L_DISCOUNT BETWEEN 0 AND 1),
9     L_TAX DECIMAL NOT NULL CHECK (L_TAX >= 0),
10    L_RETURNFLAG CHAR(1),
11    L_LINESTATUS CHAR(1),
12    L_SHIPDATE DATE,
13    L_COMMITDATE DATE,
14    L_RECEIPTDATE DATE CHECK (L_SHIPDATE <= L_RECEIPTDATE),
15    L_SHIPINSTRUCT CHAR(25),
16    L_SHIPMODE CHAR(10),
17    L_COMMENT VARCHAR(44),
18    PRIMARY KEY (L_ORDERKEY, L_LINENUMBER),
19    FOREIGN KEY (L_ORDERKEY) REFERENCES ORDERS(O_ORDERKEY),
20    FOREIGN KEY (L_PARTKEY, L_SUPPKEY) REFERENCES PARTSUPP(PS_PARTKEY, PS_SUPPKEY)
21 );
22

```

Data Output Messages Notifications

CREATE TABLE

Query returned successfully in 73 msec.

Figure 9: Creation of LINEITEM Table

Query History

```

1 ✓ ALTER TABLE LINEITEM
2   ADD CONSTRAINT fk_lineitem_part FOREIGN KEY (L_PARTKEY) REFERENCES PART(P_PARTKEY);
3
4 ✓ ALTER TABLE LINEITEM
5   ADD CONSTRAINT fk_lineitem_supp FOREIGN KEY (L_SUPPKEY) REFERENCES SUPPLIER(S_SUPPKEY);
6

```

Data Output Messages Notifications

ALTER TABLE

Query returned successfully in 321 msec.

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. **The 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.

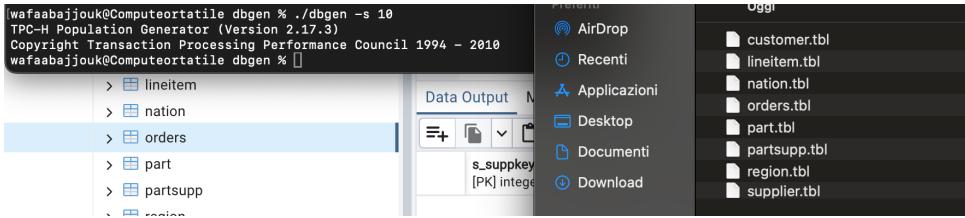


Figure 11: Data generation using the TPC-H tool

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.

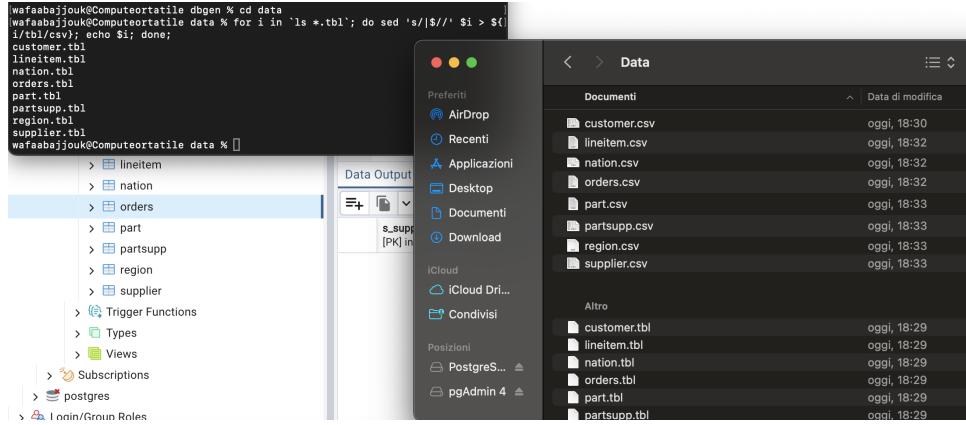


Figure 12: Conversion of data from .tbl to .csv format

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

See the screenshots of the executions below.

```

data — psql -U wafaabajjouk -d TPCH-Project — 131x26
TPCH-Project=# COPY REGION FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/region.csv' DELIMITER '|' CSV;
COPY 5
TPCH-Project=# COPY NATION FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/nation.csv' DELIMITER '|' CSV;
COPY 25
TPCH-Project=# COPY PARTSUPP FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/partsupp.csv' DELIMITER '|' CSV;
ERROR: insert or update on table "partsupp" violates foreign key constraint "partsupp_ps_partkey_fkey"
DETAIL: Key (ps_partkey)=(1) is not present in table "part".
TPCH-Project=# COPY REGION FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/region.csv' DELIMITER '|' CSV;
COPY 5
TPCH-Project=# COPY NATION FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/nation.csv' DELIMITER '|' CSV;
COPY 25
TPCH-Project=# COPY PART FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/part.csv' DELIMITER '|' CSV;
COPY 2000000
TPCH-Project=# COPY SUPPLIER FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/supplier.csv' DELIMITER '|' CSV;
COPY 100000
TPCH-Project=# COPY PARTSUPP FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/partsupp.csv' DELIMITER '|' CSV;
COPY 8000000
TPCH-Project=# COPY CUSTOMER FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/customer.csv' DELIMITER '|' CSV;
COPY 1500000

```

Figure 13: Initial Data Load Execution

```

8 COPY ORDERS FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/orders.csv' DELIMITER '|' CSV;
9 COPY LINEITEM FROM '/Users/wafaabajjouk/BigData/TPC-H/dbgen/data/lineitem.csv' DELIMITER '|' CSV;

```

Data Output Messages Notifications

COPY 59986052

Query returned successfully in 54 min 22 secs.

Figure 14: Successful Data Load Execution

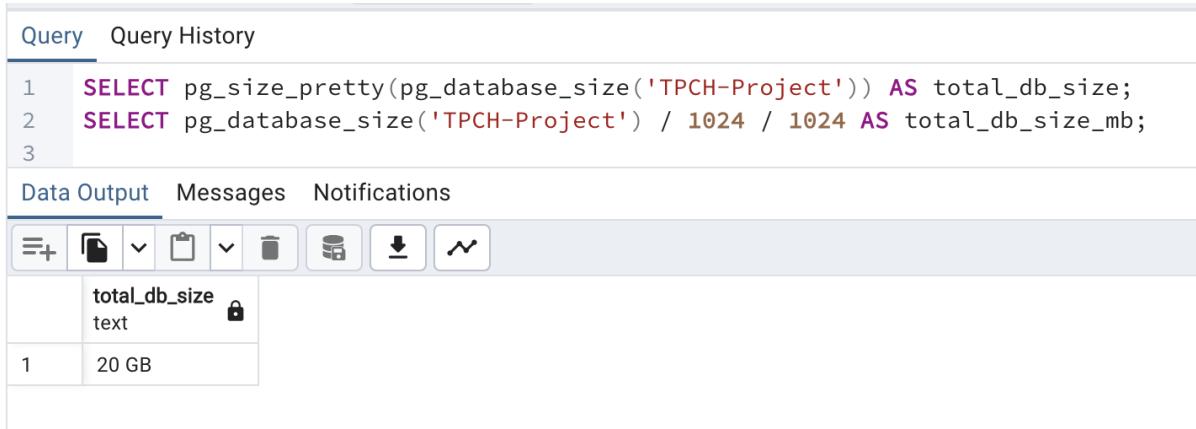
With the database now fully populated with data, the next steps involve implementing the Query Schema 1: Export/Import Revenue Value.

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.



The screenshot shows a PostgreSQL query tool interface. The top bar has tabs for 'Query' (which is selected) and 'Query History'. Below the tabs is a code editor containing the following SQL query:

```
1  SELECT pg_size.pretty(pg_database_size('TPCH-Project')) AS total_db_size;
2  SELECT pg_database_size('TPCH-Project') / 1024 / 1024 AS total_db_size_mb;
3
```

The 'Data Output' tab is selected, showing a single row of results:

	total_db_size
1	20 GB

Below the table are several icons for file operations (e.g., save, open, copy, delete).

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

```

1 -- Number of rows in each table
2 ▼ SELECT 'REGION' AS TableName, COUNT(*) AS RowCount FROM REGION
3 UNION ALL
4 SELECT 'NATION', COUNT(*) FROM NATION
5 UNION ALL
6 SELECT 'PART', COUNT(*) FROM PART
7 UNION ALL
8 SELECT 'SUPPLIER', COUNT(*) FROM SUPPLIER
9 UNION ALL
10 SELECT 'PARTSUPP', COUNT(*) FROM PARTSUPP
11 UNION ALL
12 SELECT 'CUSTOMER', COUNT(*) FROM CUSTOMER
13 UNION ALL
14 SELECT 'ORDERS', COUNT(*) FROM ORDERS
15 UNION ALL
16 SELECT 'LINEITEM', COUNT(*) FROM LINEITEM;
17
18 -- Table sizes in bytes

```

Data Output Messages Notifications

The screenshot shows a PostgreSQL query results window. At the top, there are tabs for 'Data Output', 'Messages', and 'Notifications'. Below the tabs is a toolbar with various icons for file operations like new, open, save, and export. The main area displays a table with the following data:

	tablename	rowcount
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

Query Query History

```

12 SELECT 'CUSTOMER', COUNT(*) FROM CUSTOMER
13 UNION ALL
14 SELECT 'ORDERS', COUNT(*) FROM ORDERS
15 UNION ALL
16 SELECT 'LINEITEM', COUNT(*) FROM LINEITEM;
17
18 -- Table sizes in bytes
19 v SELECT
20   relname AS TableName,
21   pg_size.pretty(pg_total_relation_size(relid)) AS TotalSize
22 FROM
23   pg_catalog.pg_statistic_user_tables
24 ORDER BY
25   pg_total_relation_size(relid) DESC;
26
27 -- Distinct values for relevant attributes
28 v SELECT 'REGION' AS TableName, 'R_REGIONKEY' AS ColumnName, COUNT(DISTINCT R_REGIONKEY) AS DistinctCount FROM REGION
29 UNION ALL
30 SELECT 'NATION', 'N_NATIONKEY', COUNT(DISTINCT N_NATIONKEY) FROM NATION

```

Data Output Messages Notifications

	tablename	totalsize
	name	text
1	lineitem	10074 MB
2	orders	2360 MB
3	partsupp	1535 MB
4	part	363 MB
5	customer	312 MB
6	supplier	20 MB
7	region	24 kB
8	nation	24 kB

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
25      pg_total_retraction_size(recln) DESC,
26
27  -- Distinct values for relevant attributes
28 v SELECT 'REGION' AS TableName, 'R_REGIONKEY' AS ColumnName, COUNT(DISTINCT R_REGIONKEY) AS DistinctCount FROM REGION
29 UNION ALL
30 SELECT 'NATION', 'N_NATIONKEY', COUNT(DISTINCT N_NATIONKEY) FROM NATION
31 UNION ALL
32 SELECT 'PART', 'P_PARTKEY', COUNT(DISTINCT P_PARTKEY) FROM PART
33 UNION ALL
34 SELECT 'SUPPLIER', 'S_SUPPKEY', COUNT(DISTINCT S_SUPPKEY) FROM SUPPLIER
35 UNION ALL
36 SELECT 'PARTSUPP', 'PS_PARTKEY', COUNT(DISTINCT PS_PARTKEY) FROM PARTSUPP
37 UNION ALL
38 SELECT 'CUSTOMER', 'C_CUSTKEY', COUNT(DISTINCT C_CUSTKEY) FROM CUSTOMER
39 UNION ALL
40 SELECT 'ORDERS', 'O_ORDERKEY', COUNT(DISTINCT O_ORDERKEY) FROM ORDERS
41 UNION ALL
42 SELECT 'LINEITEM', 'L_ORDERKEY', COUNT(DISTINCT L_ORDERKEY) FROM LINEITEM;
43

```

Data Output Messages Notifications

	tablename	columnname	distinctcount
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.

```

44 -- Min and max values for relevant attributes
45 ✓ SELECT 'REGION' AS TableName, 'R_REGIONKEY' AS ColumnName, MIN(R_REGIONKEY) AS MinValue, MAX(R_REGIONKEY) AS MaxValue FROM REGION
46 UNION ALL
47 SELECT 'NATION', 'N_NATIONKEY', MIN(N_NATIONKEY), MAX(N_NATIONKEY) FROM NATION
48 UNION ALL
49 SELECT 'PART', 'P_PARTKEY', MIN(P_PARTKEY), MAX(P_PARTKEY) FROM PART
50 UNION ALL
51 SELECT 'SUPPLIER', 'S_SUPPKEY', MIN(S_SUPPKEY), MAX(S_SUPPKEY) FROM SUPPLIER
52 UNION ALL
53 SELECT 'PARTSUPP', 'PS_PARTKEY', MIN(PS_PARTKEY), MAX(PS_PARTKEY) FROM PARTSUPP
54 UNION ALL
55 SELECT 'CUSTOMER', 'C_CUSTKEY', MIN(C_CUSTKEY), MAX(C_CUSTKEY) FROM CUSTOMER
56 UNION ALL
57 SELECT 'ORDERS', 'O_ORDERKEY', MIN(O_ORDERKEY), MAX(O_ORDERKEY) FROM ORDERS
58 UNION ALL
59 SELECT 'LINEITEM', 'L_ORDERKEY', MIN(L_ORDERKEY), MAX(L_ORDERKEY) FROM LINEITEM;
60

```

Data Output Messages Notifications

The screenshot shows a database interface with a table titled 'Data Output'. The table has columns: tablename, columnname, minvalue, and maxvalue. The data is as follows:

	tablename	columnname	minvalue	maxvalue
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, I am 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:

$$\text{Revenue} = \text{l_extendedprice} \times (1 - \text{l_discount}) \quad (1)$$

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.

The screenshot shows a database query interface with the following details:

Query History:

```

1 -- the parameters using a WITH clause
2 WITH
3 param_type AS (SELECT 'STANDARD POLISHED STEEL'::text AS type),
4 param_exportingNation AS (SELECT 'MOROCO'::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

```

Data Output:

	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	STANDARD POLISHED STEEL	MOROCO	ALGERIA	97910.8592
2	1992		1	STANDARD POLISHED STEEL	MOROCO	ARGENTINA	52450.7292
3	1992		1	STANDARD POLISHED STEEL	MOROCO	CANADA	61350.5682
4	1992		1	STANDARD POLISHED STEEL	MOROCO	EGYPT	52038.7200
5	1992		1	STANDARD POLISHED STEEL	MOROCO	FRANCE	29366.4000
6	1992		1	STANDARD POLISHED STEEL	MOROCO	INDIA	140712.4426

Total rows: 1000 of 2249 Query complete 00:00:27.336

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 '**MOROCO**'.

- 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

```

1 WITH params AS (
2     SELECT 'ECONOMY BRUSHED STEEL'::text AS type,
3            'ARGENTINA'::text AS exportingNation
4 )
5 SELECT
6     N1.N_NAME AS "Exporting nation",
7     R1.R_NAME AS "Exporting region",
8     N2.N_NAME AS "Importing nation",
9     R2.R_NAME AS "Importing region",
10    P.P_TYPE AS Type,
11    SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
12 FROM LINEITEM L
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 PART P ON L.L_PARTKEY = P.P_PARTKEY
17 JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
18 JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
19 JOIN REGION R1 ON N1.N_REGIONKEY = R1.R_REGIONKEY
20 JOIN REGION R2 ON N2.N_REGIONKEY = R2.R_REGIONKEY
21 JOIN params ON TRUE
22 WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
23 AND P.P_TYPE = params.type
24 AND N1.N_NAME = params.exportingNation
25 GROUP BY ROLLUP(N1.N_NAME, R1.R_NAME, N2.N_NAME, R2.R_NAME, P.P_TYPE)
26 ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
27

```

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 :

```
Query - Query History
1 WITH
2   param_type AS (SELECT 'STANDARD POLISHED STEEL'::text AS type),
3   param_exportingNation AS (SELECT 'MOROCCO'::text AS exportingNation)
4
5   SELECT
6     EXTRACT(YEAR FROM L.L_SHIPDATE) AS Year,
7     EXTRACT(QUARTER FROM L.L_SHIPDATE) AS Quarter,
8     EXTRACT(MONTH FROM L.L_SHIPDATE) AS Month,
9     P_P_TYPE AS Type,
10    N1.N_NAME AS "Exporting nation",
11    N2.N_NAME AS "Importing nation",
12    SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
13  FROM TBLPARTS L
14  JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
15  JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
16  JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
17  JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
18  JOIN PARTSUPPLY PS ON P.P_PARTKEY = PS.PS_PARTKEY
19  JOIN NATION N1 ON C.C_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  AND N2.N_NAME = (SELECT importingnation FROM param_importingNation)
25  GROUP BY ROLLUP(EXTRACT(YEAR 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";
```

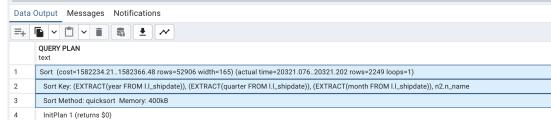


Figure 23: Query Cost of Time-Based Aggregation: COST and ROWS

```
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
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";
```

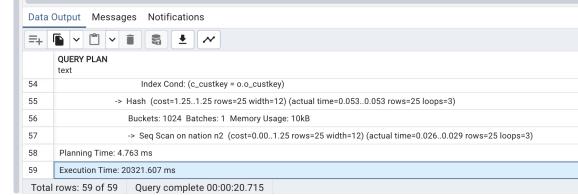


Figure 24: Query Cost of Time-Based Aggregation: Execution Time

- 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**.

```

1 v EXPLAIN ANALYZE
2 WITH params AS (
3   SELECT 'ECONOMY ANODIZED BRASS'::text AS type,
4         'CANADA'::text AS exportingNation
5 )
6
7 SELECT 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 SUPPLIERS S ON L_L_SUPPLKEY = S_S_SUPPLKEY
14 JOIN CUSTOMERS C ON L_L_CUSTKEY = C_C_CUSTKEY
15 JOIN ORDERS O ON L_L_ORDERKEY = O_O_ORDERKEY
16 JOIN REGION R ON O_O_REGIONKEY = R_R_REGIONKEY
17 JOIN NATION N1 ON S_S_NATIONKEY = N1_N_NATIONKEY
18 JOIN NATION N2 ON C_C_NATIONKEY = N2_N_NATIONKEY
19 JOIN PARTS P ON P_P_TYPE = S_S_SUPPLKEY
20 WHERE NL_N_NATIONKEY != N2_N_NATIONKEY
21 AND NL_N_NAME = params.type
22 AND NL_N_NAME = params.exportingNation
23 GROUP BY ROLLUP(P_P_TYPE, NL_N_NAME, N2_N_NAME)
24 ORDER BY Type, "Exporting nation", "Importing nation";
25

```

Group Key: p.p_type, n1_n_name, n2_n_name

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

```

1 v EXPLAIN ANALYZE
2 WITH params AS (
3   SELECT 'ECONOMY ANODIZED BRASS'::text AS type,
4         'CANADA'::text AS exportingNation
5 )
6
7 SELECT 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 SUPPLIERS S ON L_L_SUPPLKEY = S_S_SUPPLKEY
14 JOIN CUSTOMERS C ON L_L_CUSTKEY = C_C_CUSTKEY
15 JOIN ORDERS O ON L_L_ORDERKEY = O_O_ORDERKEY
16 JOIN REGION R ON O_O_REGIONKEY = R_R_REGIONKEY
17 JOIN NATION N1 ON S_S_NATIONKEY = NL_N_NATIONKEY
18 JOIN NATION N2 ON C_C_NATIONKEY = N2_N_NATIONKEY
19 JOIN PARTS P ON P_P_TYPE = S_S_SUPPLKEY
20 WHERE NL_N_NATIONKEY != N2_N_NATIONKEY
21 AND NL_N_NAME = params.type
22 AND NL_N_NAME = params.exportingNation
23 GROUP BY ROLLUP(P_P_TYPE, NL_N_NAME, N2_N_NAME)
24 ORDER BY Type, "Exporting nation", "Importing nation";
25

```

Group Key: p.p_type, n1_n_name, n2_n_name

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.

```

1 v EXPLAIN ANALYZE
2 WITH params AS (
3   SELECT 'ECONOMY BRUSHED STEEL'::text AS type,
4         'ARGENTINA'::text AS exportingNation
5 )
6
7 SELECT P_P_TYPE AS Type,
8       N1_N_NAME AS "Exporting nation",
9       R1_R_NAME AS "Exporting region",
10      N2_N_NAME AS "Importing nation",
11      R2_R_NAME AS "Importing region",
12      SUM(L_L_EXTENDEDPRICE * (1 - L_L_DISCOUNT)) AS Revenue
13 FROM LINEITEM L
14 JOIN ORDERS O ON L_L_ORDERKEY = O_O_ORDERKEY
15 JOIN CUSTOMER C ON O_O_CUSTKEY = C_C_CUSTKEY
16 JOIN SUPPLIERS S ON L_L_SUPPLKEY = S_S_SUPPLKEY
17 JOIN PARTS P ON L_L_PARTKEY = P_P_PARTKEY
18 JOIN NATION N1 ON S_S_NATIONKEY = NL_N_NATIONKEY
19 JOIN NATION N2 ON C_C_NATIONKEY = N2_N_NATIONKEY
20 JOIN REGION R1 ON NL_N_REGIONKEY = R1_R_REGIONKEY
21 JOIN REGION R2 ON N2_N_REGIONKEY = R2_R_REGIONKEY
22 JOIN PARTS P ON P_P_TYPE = S_S_SUPPLKEY
23 WHERE NL_N_NATIONKEY != N2_N_NATIONKEY
24 AND NL_N_TYPE = params.type
25 AND NL_N_NAME = params.exportingNation
26 GROUP BY ROLLUP(N1_N_NAME, R1_R_NAME, N2_N_NAME, R2_R_NAME, P_P_TYPE)
27 ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
28

```

Group Key: n1_n_name, r1_r_name, n2_n_name, r2_r_name, p.p_type

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

```

1 v EXPLAIN ANALYZE
2 WITH params AS (
3   SELECT 'ECONOMY BRUSHED STEEL'::text AS type,
4         'ARGENTINA'::text AS exportingNation
5 )
6
7 SELECT P_P_TYPE AS Type,
8       N1_N_NAME AS "Exporting nation",
9       R1_R_NAME AS "Exporting region",
10      N2_N_NAME AS "Importing nation",
11      R2_R_NAME AS "Importing region",
12      SUM(L_L_EXTENDEDPRICE * (1 - L_L_DISCOUNT)) AS Revenue
13 FROM LINEITEM L
14 JOIN ORDERS O ON L_L_ORDERKEY = O_O_ORDERKEY
15 JOIN CUSTOMER C ON O_O_CUSTKEY = C_C_CUSTKEY
16 JOIN SUPPLIERS S ON L_L_SUPPLKEY = S_S_SUPPLKEY
17 JOIN PARTS P ON L_L_PARTKEY = P_P_PARTKEY
18 JOIN NATION N1 ON S_S_NATIONKEY = NL_N_NATIONKEY
19 JOIN NATION N2 ON C_C_NATIONKEY = N2_N_NATIONKEY
20 JOIN REGION R1 ON NL_N_REGIONKEY = R1_R_REGIONKEY
21 JOIN REGION R2 ON N2_N_REGIONKEY = R2_R_REGIONKEY
22 JOIN PARTS P ON P_P_TYPE = S_S_SUPPLKEY
23 WHERE NL_N_NATIONKEY != N2_N_NATIONKEY
24 AND NL_N_TYPE = params.type
25 AND NL_N_NAME = params.exportingNation
26 GROUP BY ROLLUP(N1_N_NAME, R1_R_NAME, N2_N_NAME, R2_R_NAME, P_P_TYPE)
27 ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
28

```

Group Key: n1_n_name, r1_r_name, n2_n_name, r2_r_name

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:

The screenshot shows a PostgreSQL query history window. The top tab is 'Query History'. The main content area displays the following SQL code:

```
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_D]
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 t
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
22
```

Below the code, there are tabs for 'Data Output', 'Messages' (which is selected), and 'Notifications'. The 'Messages' tab shows the command 'CREATE INDEX' and the message '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

```

1 v SELECT pg_size.pretty(SUM(size_bytes)) AS total_size
2 FROM (
3     SELECT pg_total_relation_size('idx_lineitem_orderkey_shipdate') AS size_bytes
4     UNION ALL
5     SELECT pg_total_relation_size('idx_lineitem_partkey_suppkey')
6     UNION ALL
7     SELECT pg_total_relation_size('idx_lineitem_extendedprice_discount')
8     UNION ALL
9     SELECT pg_total_relation_size('idx_lineitem_partkey_suppkey_orderkey')
10    UNION ALL
11    SELECT pg_total_relation_size('idx_orders_custkey_orderdate')
12    UNION ALL
13    SELECT pg_total_relation_size('idx_customer_nationkey_custkey')
14    UNION ALL
15    SELECT pg_total_relation_size('idx_supplier_nationkey_suppkey')
16    UNION ALL
17    SELECT pg_total_relation_size('idx_part_type')
18    UNION ALL
19    SELECT pg_total_relation_size('idx_part_type_partkey')
20    UNION ALL
21    SELECT pg_total_relation_size('idx_part_partkey_type')
22    UNION ALL
23    SELECT pg_total_relation_size('idx_nation_regionkey_nationkey')
24    UNION ALL
25    SELECT pg_total_relation_size('idx_nation_name')
26 ) AS total;
27

```

Data Output Messages Notifications

The screenshot shows a PostgreSQL query result window. At the top, there are tabs for 'Query' (which is selected) and 'Query History'. Below the tabs is a multi-line code block representing a SQL query. The query calculates the total size of various indexes in a database. It uses the 'pg_size.pretty' function to format the output. The result is displayed in a table with two columns: 'total_size' and 'text'. There is one row with the value '6021 MB'. Below the table is a toolbar with several icons, including a plus sign, a file icon, a dropdown arrow, a clipboard icon, a trash can, a download icon, and a refresh icon.

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) : $6021 \text{ MB} < 30720 \text{ MB}$, **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
3 param_exportingNation AS (SELECT 'MOROCCO'::text AS exportingNation)
4
5 SELECT
6     EXTRACT(YEAR FROM L.L_SHIPDATE) AS Year,
7     EXTRACT(QUARTER FROM L.L_SHIPDATE) AS Quarter,
8     EXTRACT(MONTH FROM L.L_SHIPDATE) AS Month,
9     P.P_TYPE AS Type,
10    N1.N_NAME AS "Exporting nation",
11    N2.N_NAME AS "Importing nation",
12    SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
13  FROM LINEITEM L
14 JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
15 JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
16 JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
17 JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
18 JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
19 JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
20 WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
21 AND P.P_TYPE = (SELECT type FROM param_type)
22 AND N1.N_NAME = (SELECT exportingNation FROM param_exportingNation)
23 --aggregate the data first by month, then roll up to the quarter, and finally to the year,
24 GROUP BY ROLLUP(EXTRACT(MONTH FROM L.L_SHIPDATE), EXTRACT(QUARTER FROM L.L_SHIPDATE),
25                 EXTRACT(YEAR FROM L.L_SHIPDATE), P.P_TYPE, N1.N_NAME, N2.N_NAME)
26 ORDER BY Year, Quarter, Month, Type, "Exporting nation", "Importing nation";
27 --before indexing :27sec 336 msec
28 --after indexing : 15 secs 181 msec.
29

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.I_shipdate)), (EXTRACT(quarter FROM L.I_shipdate)), (EXTRACT(month FROM L.I_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.

The screenshot shows a database query editor with the following details:

- Query History:** The tab is selected.
- Query Text:**

```

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      --before indexing    48 secs 560 msec.
25      -- after indexing : 45 secs 591 msec.
26
27

```
- Data Output:** The tab is selected.
- Messages:** Successfully run. Total query runtime: 45 secs 591 msec.
- Notifications:** 27 rows affected.

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

```

2  WITH params AS (
3      SELECT 'ECONOMY ANODIZED BRASS'::text AS type,
4             'CANADA'::text AS exportingNation
5  )
6
7  SELECT
8      P.P_TYPE AS Type,
9      N1.N_NAME AS "Exporting nation",
10     N2.N_NAME AS "Importing nation",
11     SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
12  FROM LINEITEM L
13  JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
14  JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
15  JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
16  JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
17  JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
18  JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
19  JOIN params ON TRUE
20  WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
21  AND P.P_TYPE = params.type
22  AND N1.N_NAME = params.exportingNation
23  GROUP BY ROLLUP(P.P_TYPE, N1.N_NAME, N2.N_NAME)
24  ORDER BY Type, "Exporting nation", "Importing nation";
25  -- cost before indexes: (cost= 3228022.28..3228246.04 rows=2987 width=69) (actual time=49142.324..50016.339 rows=27 loops=1)
26  -- 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
3   Group Key: p.p_type, n1.n_name

```

Figure 34: Cost Query Result

Query Query History

```

2  SELECT 'ECONOMY BRUSHED STEEL'::text AS type,
3        'ARGENTINA'::text AS exportingNation
4  )
5  SELECT
6      N1.N_NAME AS "Exporting nation",
7      R1.R_NAME AS "Exporting region",
8      N2.N_NAME AS "Importing nation",
9      R2.R_NAME AS "Importing region",
10     P.P_TYPE AS Type,
11     SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
12  FROM LINEITEM L
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 PART P ON L.L_PARTKEY = P.P_PARTKEY
17  JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
18  JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
19  JOIN REGION R1 ON N1.N_REGIONKEY = R1.R_REGIONKEY
20  JOIN REGION R2 ON N2.N_REGIONKEY = R2.R_REGIONKEY
21  JOIN params ON TRUE
22  WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
23  AND P.P_TYPE = params.type
24  AND N1.N_NAME = params.exportingNation
25  GROUP BY ROLLUP(N1.N_NAME, R1.R_NAME, N2.N_NAME, R2.R_NAME, P.P_TYPE)
26  ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
27  -- before indexing : 56 secs 745 msec.
28  -- after indexing : 48 secs 363 msec.
29

```

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.

```
10      R2.R_NAME AS "Importing region",
11      P.P_TYPE AS Type,
12      SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
13  FROM LINEITEM L
14  JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
15  JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
16  JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
17  JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
18  JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
19  JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
20  JOIN REGION R1 ON N1.N_REGIONKEY = R1.R_REGIONKEY
21  JOIN REGION R2 ON N2.N_REGIONKEY = R2.R_REGIONKEY
22  JOIN params ON TRUE
23  WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
24  AND P.P_TYPE = params.type
25  AND N1.N_NAME = params.exportingNation
26  GROUP BY ROLLUP(N1.N_NAME, R1.R_NAME, N2.N_NAME, R2.R_NAME, P.P_TYPE)
27  ORDER BY "Exporting nation", "Exporting region", "Importing nation", Type;
28  -- cost before indexing : cost=341818.21..3418366.78 rows=4405 width=205) (actual time=54157.594..55233.941 rows=75 loops=1)
29  -- 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



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 aggregation 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.

The screenshot shows a database query interface with two main sections: 'Query' and 'Data Output'.

Query:

```
1 ✓ CREATE MATERIALIZED VIEW MV_time_agg AS
2   SELECT
3     EXTRACT(YEAR FROM L.L_SHIPDATE) AS Year,
4     EXTRACT(QUARTER FROM L.L_SHIPDATE) AS Quarter,
5     EXTRACT(MONTH FROM L.L_SHIPDATE) AS Month,
6     P.P_TYPE AS Type,
7     N1.N_NAME AS ExportingNation,
8     N2.N_NAME AS ImportingNation,
9     SUM(L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT)) AS Revenue
10    FROM LINEITEM L
11    JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
12    JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
13    JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
14    JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
15    JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
16    JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
17    WHERE N1.N_NATIONKEY != N2.N_NATIONKEY
18    GROUP BY
19      ROLLUP(
20        EXTRACT(MONTH FROM L.L_SHIPDATE),
21        EXTRACT(QUARTER FROM L.L_SHIPDATE),
22        EXTRACT(YEAR FROM L.L_SHIPDATE),
23        P.P_TYPE,
24        N1.N_NAME,
25        N2.N_NAME
26      )
27    ORDER BY Year, Quarter, Month, Type, ExportingNation, ImportingNation;
28    -- the query using the view
29 ✓ SELECT *
30   FROM MV_time_agg
31   WHERE Type = 'STANDARD POLISHED STEEL'
32   AND ExportingNation = 'MOROCCO'
33   ORDER BY Year, Quarter, Month, Type, ExportingNation, ImportingNation;
```

Data Output:

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

```

Query  Query History
40
41  --CORRECTED QUERY :
42 v CREATE MATERIALIZED VIEW mv_export_import_revenue AS
43  SELECT
44    EXTRACT(YEAR FROM L.L_SHIPDATE) AS Year,
45    EXTRACT(QUARTER FROM L.L_SHIPDATE) AS Quarter,
46    EXTRACT(MONTH FROM L.L_SHIPDATE) AS Month,
47    P.P_TYPE AS Type,
48    N1.N_NAME AS ExportingNation,
49    N2.N_NAME AS ImportingNation,
50    L.L_EXTENDEDPRIICE * (1 - L.L_DISCOUNT) AS Revenue
51  FROM LINEITEM L
52  JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
53  JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
54  JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
55  JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
56  JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
57  JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
58 WHERE N1.N_NATIONKEY != N2.N_NATIONKEY;
59
Data Output  Messages  Notifications

```

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

```

60  --time based aggregation query |
61
62 v SELECT month, quarter ,year, TYPE, ExportingNation,ImportingNation,sum(Revenue) As Revenue
63  FROM mv_export_import_revenue
64  WHERE TYPE = 'STANDARD POLISHED STEEL'
65  AND ExportingNation = 'MOROCCO'
66  GROUP BY ROLLUP(month,quarter ,year, TYPE, ExportingNation, ImportingNation)
67  ORDER BY Year, Quarter, Month, Type, ExportingNation, ImportingNation;
68
Data Output  Messages  Notifications

```

month	quarter	year	TYPE	ExportingNation	ImportingNation	Revenue
1	1	1	1992	STANDARD POLISHED STEEL	MOROCCO	ALGERIA 97910.8592
2	1	1	1992	STANDARD POLISHED STEEL	MOROCCO	ARGENTINA 52450.7292
3	1	1	1992	STANDARD POLISHED STEEL	MOROCCO	CANADA 61350.5682

Total rows: 1000 of 2249 Query complete 00:00:12.237

Figure 39: Time Base Aggregation Query Using the Materialized View

8.3.2 Type Base Aggregation Query Using the Materialized View

```

69 --TYPE BASED aggregation QUERY
70
71 ✓ SELECT type, ExportingNation, ImportingNation,sum(Revenue) AS Revenue
72   FROM mv_export_import_revenue
73   WHERE type = 'ECONOMY ANODIZED BRASS'
74   AND ExportingNation = 'CANADA'
75   GROUP BY ROLLUP(
76     TYPE,
77     ExportingNation,
78     ImportingNation)
79   ORDER BY Type, ExportingNation, ImportingNation;
80

```

Data Output Messages Notifications

type	ExportingNation	ImportingNation	revenue
ECONOMY ANODIZED BRASS	CANADA	ALGERIA	21063762.1642
ECONOMY ANODIZED BRASS	CANADA	ARGENTINA	23305153.3875
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
1 ✓ CREATE MATERIALIZED VIEW mv_export_import_revenue_region AS
2   SELECT
3     N1.N_NAME AS ExportingNation,
4     R1.R_NAME AS ExportingRegion,
5     N2.N_NAME AS ImportingNation,
6     R2.R_NAME AS ImportingRegion,
7     P.P_TYPE AS Type,
8     L.L_EXTENDEDPRICE * (1 - L.L_DISCOUNT) AS Revenue
9   FROM LINEITEM L
10  JOIN ORDERS O ON L.L_ORDERKEY = O.O_ORDERKEY
11  JOIN CUSTOMER C ON O.O_CUSTKEY = C.C_CUSTKEY
12  JOIN SUPPLIER S ON L.L_SUPPKEY = S.S_SUPPKEY
13  JOIN PART P ON L.L_PARTKEY = P.P_PARTKEY
14  JOIN NATION N1 ON S.S_NATIONKEY = N1.N_NATIONKEY
15  JOIN NATION N2 ON C.C_NATIONKEY = N2.N_NATIONKEY
16  JOIN REGION R1 ON N1.N_REGIONKEY = R1.R_REGIONKEY
17  JOIN REGION R2 ON N2.N_REGIONKEY = R2.R_REGIONKEY
18  WHERE N1.N_NATIONKEY != N2.N_NATIONKEY;
19

```

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:

```
19  --> QUERY
20  <-- SELECT
21      ExportingNation AS "Exporting nation",
22      ExportingRegion AS "Exporting region",
23      ImportingNation AS "Importing nation",
24      ImportingRegion AS "Importing region",
25      Type,
26      SUM(Revenue) AS Revenue
27  FROM mv_export_import_revenue_region
28
29  WHERE Type = 'ECONOMY BRUSHED STEEL'
30  AND ExportingNation = 'ARGENTINA'
31  GROUP BY ROLLUP(ExportingNation, ExportingRegion, ImportingNation, ImportingRegion, Type)
32  ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
33
```

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

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

```
Query Query History

1 ✓ EXPLAIN ANALYSE
2 SELECT month,quarter,_year,TYPE, ExportingNation,ImportingNation,sum(Revenue) AS Revenue
3 FROM mv_export_1_import_revenue
4 WHERE TYPE = 'STANDARD POLISHED STEEL'
5 AND ExportingNation = 'MOROCCO'
6 GROUP BY ROLLUP(month,quarter,_year,TYPE, ExportingNation, ImportingNation)
7 ORDER BY Year,Quarter,Month,Type,ExportingNation,ImportingNation;
8

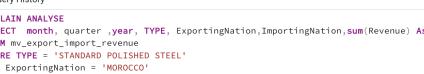
Data Output Messages Notifications

```

Figure 43: Query Cost using Materialized View

```
Query Query History

1 --time based aggregation query
2
3 ✓ SELECT month, quarter ,year ,TYPE,
4   ExportingNation,ImportingNation,sum(Revenue) As Revenue
5 FROM mv_export_import_revenue
6 WHERE TYPE = 'STANDARD POLISHED STEEL'
7 AND ExportingNation = 'MOROCCO'
8 GROUP BY ROLLUP(month,quarter ,year ,TYPE, ExportingNation, ImportingNation)
9 ORDER BY Year, Quarter, Month, Type, ExportingNation, ImportingNation;
10 --before indexing :27sec 336 msec
11 --after indexing : 15 secs 181 msec.
12 --after Mat View : 12 secs 144 msec.

Data Output Messages Notifications

Successfully run. Total query runtime: 12 secs 144 msec.
2249 rows affected.
```

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

```

2 ✓ EXPLAIN ANALYZE
3   SELECT type, ExportingNation, ImportingNation,sum(Revenue) AS Revenue
4   FROM mv_export_import_revenue
5   WHERE type = 'ECONOMY ANODIZED BRASS'
6   AND ExportingNation = 'CANADA'
7   GROUP BY ROLLUP(
8     TYPE,
9     ExportingNation,
10    ImportingNation)
11  ORDER BY Type, ExportingNation, ImportingNation;
12
Data Output Messages Notifications

```

QUERY PLAN

```

1 GroupAggregate (cost=1021368.81..1023553.87 rows=18118 width=69) (actual time=12584.592..12590.968 rows=27 loops=1)
2  Group Key:type,exportingnation,importingnation
3  Group Key:type,exportingnation

```

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

9.3 Query Cost of Region and Nation Aggregation using Materialized View

```

1 ✓ EXPLAIN ANALYZE
2   SELECT
3     ExportingNation AS "Exporting nation",
4     ExportingRegion AS "Exporting region",
5     ImportingNation AS "Importing nation",
6     ImportingRegion AS "Importing region",
7     Type,
8     SUM(Revenue) AS Revenue
9   FROM mv_export_import_revenue_region
10  WHERE TYPE = 'ECONOMY BRUSHED STEEL'
11  AND ExportingNation = 'ARGENTINA'
12  GROUP BY ROLLUP(ExportingNation, ExportingRegion, ImportingNation, ImportingRegion, Type)
13  ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
14
Data Output Messages Notifications

```

QUERY PLAN

```

1 GroupAggregate (cost=1024647.27..1027052.85 rows=28639 width=83) (actual time=14011.081..14020.501 rows=75 loops=1)
2  Group Key:exportingnation,exportingregion,importingnation,importingregion,type
3  Group Key:exportingnation,exportingregion,importingnation,importingregion

```

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

```

1 ✓ SELECT type, ExportingNation, ImportingNation,sum(Revenue) AS Revenue
2   FROM mv_export_import_revenue
3   WHERE type = 'ECONOMY ANODIZED BRASS'
4   AND ExportingNation = 'CANADA'
5   GROUP BY ROLLUP(
6     TYPE,
7     ExportingNation,
8     ImportingNation)
9   ORDER BY Type, ExportingNation, ImportingNation;
10  --before indexing : 48 secs 560 msec.
11  -- after indexing : 45 secs 591 msec.
12  -- after Mat View : 14 secs 618 msec.
13
Data Output Messages Notifications

```

Successfully run. Total query runtime: 14 secs 618 msec.
27 rows affected.

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

```

-----+-----+
1 ✓ SELECT
2   ExportingNation AS "Exporting nation",
3   ExportingRegion AS "Exporting region",
4   ImportingNation AS "Importing nation",
5   ImportingRegion AS "Importing region",
6   Type,
7   SUM(Revenue) AS Revenue
8   FROM mv_export_import_revenue_region
9
10 WHERE Type = 'ECONOMY BRUSHED STEEL'
11 AND ExportingNation = 'ARGENTINA'
12 GROUP BY ROLLUP(ExportingNation, ExportingRegion, ImportingNation, ImportingRegion, Type)
13 ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
14
15  -- before indexing : 56 secs 745 msec.
16  -- after indexing : 48 secs 363 msec.
17  -- after Mat View : 14 secs 108 msec.
18
Data Output Messages Notifications

```

Successfully run. Total query runtime: 14 secs 108 msec.
75 rows affected.

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.

The screenshot shows the Oracle SQL Developer interface. On the left, the Object Navigator pane displays 'Materialized views (2)' with 'mv_export_import_revenue' selected. Under 'Indexes (2)', two indexes are listed: 'idx_mv_revenue_type_exporting' and 'idx_mv_revenue_type_exporting'. On the right, the main area shows a query window with the following SQL code:

```
1 CREATE INDEX idx_mv_revenue_type_exportingnation_year_quarter_month
2 ON mv_export_import_revenue (Type, ExportingNation, Year, Quarter, Month);
3
4 CREATE INDEX idx_mv_revenue_type_exporting_importing_nation
5 ON mv_export_import_revenue (Type, ExportingNation, ImportingNation);
```

The 'Messages' tab at the bottom indicates 'Query returned successfully in 6 min 6 secs.'

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.

The screenshot shows the Oracle SQL Developer interface. On the left, the Object Navigator pane displays 'Materialized views (2)' with 'mv_export_import_revenue_region' selected. Under 'Indexes (1)', one index is listed: 'idx_mv_revenue_region_type_exporting_importing'. On the right, the main area shows a query window with the following SQL code:

```
1 CREATE INDEX idx_mv_revenue_region_type_exporting_importing
2 ON mv_export_import_revenue_region (Type, ExportingNation, ExportingRegion, ImportingNation, ImportingRegion);
```

The 'Messages' tab at the bottom indicates 'Query returned successfully in 6 min 6 secs.'

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.

```
1 v EXPLAIN ANALYSE
2
3 SELECT month, quarter, year, TYPE, ExportingNation,ImportingNation,sum(Revenue) As Revenue
4 FROM mv_export_import_revenue
5 WHERE TYPE = 'STANDARD POLISHED STEEL'
6 AND ExportingNation = 'MOROCCO'
7 GROUP BY ROLLUP(month,quarter ,year, TYPE, ExportingNation, ImportingNation)
8 ORDER BY Year, Quarter, Month, Type, ExportingNation, ImportingNation;
9
10 Data Output Messages Notifications
```

QUERY PLAN
text

```
1 Sort (cost=60968.71..61077.94 rows=43693 width=84) (actual time=1195.733..1195.775 rows=2249 loops=1)
2  Sort Key: year, quarter, month, importnation
3  Sort Method: quicksort Memory: 400KB
4  -> GroupAggregate (cost=54658.76..55508.05 rows=43693 width=84) (actual time=1186.243..1192.366 rows=2249 loops=1)
5    Group Key: month, quarter, year, type, exportnation, importnation
```

Figure 51: Query Cost Using Materialized View and Indexing

```
Query Query History
1 --time based aggregation query
2
3 v SELECT month, quarter ,year, TYPE,
4   ExportingNation,ImportingNation,sum(Revenue) As Revenue
5   FROM mv_export_import_revenue
6   WHERE TYPE = 'STANDARD POLISHED STEEL'
7   AND ExportingNation = 'MOROCCO'
8   GROUP BY ROLLUP(month,quarter ,year, TYPE, ExportingNation, ImportingNation)
9   ORDER BY Year, Quarter, Month, Type, ExportingNation, ImportingNation;
10  --before indexing :27sec 336 msec
11  --after indexing : 15 secs 181 msec.
12  --after Mat View : 12 secs 144 msec.
13  --after mat view and indexng : 116 msec.

Data Output Messages Notifications
```

Successfully run. Total query runtime: 116 msec.
2249 rows affected.

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.

```

2 ✓ EXPLAIN ANALYZE
3   SELECT type, ExportingNation, ImportingNation,sum(Revenue) As Revenue
4   FROM mv_export_import_revenue
5   WHERE type = 'ECONOMY ANODIZED BRASS'
6   AND ExportingNation = 'CANADA'
7   GROUP BY ROLLUP(
8     TYPE,
9     ExportingNation,
10    ImportingNation)
11   ORDER BY Type, ExportingNation, ImportingNation;
12
Data Output Messages Notifications
+-----+
| QUERY PLAN |
+-----+
1 | GroupAggregate (cost=55767.66..56187.72 rows=18118 width=69) (actual time=1584.825..1588.337 rows=27 loops=1)
2 |   Group Key:type,exportingnation,importingnation
3 |   Group Key:type,exportingnation

```

Figure 53: Query Cost 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.

```

1 ✓ EXPLAIN ANALYZE
2   SELECT
3     ExportingNation AS "Exporting nation",
4     ExportingRegion AS "Exporting region",
5     ImportingNation AS "Importing nation",
6     ImportingRegion AS "Importing region",
7     Type,
8     SUM(Revenue) AS Revenue
9   FROM mv_export_import_revenue_region
10  WHERE Type = 'ECONOMY BRUSHED STEEL'
11  AND ExportingNation = 'ARGENTINA'
12  GROUP BY ROLLUP(ExportingNation, ExportingRegion, ImportingNation, ImportingRegion, Type)
13  ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
14
Data Output Messages Notifications
+-----+
| QUERY PLAN |
+-----+
1 | GroupAggregate (cost=56078.78..56709.30 rows=28638 width=83) (actual time=2685.680..2689.364 rows=75 loops=1)
2 |   Group Key:exportingnation,exportingregion,importingnation,importingregion,type

```

Figure 55: Query Cost 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:

```

1 ✓ SELECT type, ExportingNation, ImportingNation,sum(Revenue) As Revenue
2   FROM mv_export_import_revenue
3   WHERE type = 'ECONOMY ANODIZED BRASS'
4   AND ExportingNation = 'CANADA'
5   GROUP BY ROLLUP(
6     TYPE,
7     ExportingNation,
8     ImportingNation)
9   ORDER BY Type, ExportingNation, ImportingNation;
10  -- before indexing : 48 secs 560 msec.
11  -- after indexing : 45 secs 591 msec.
12  -- after Mat View : 14 secs 618 msec.
13  -- after Mat View and indexing : 49 msec.
14
Data Output Messages Notifications
+-----+
| Data Output |
+-----+
| Successfully run. Total query runtime: 49 msec.
| 27 rows affected.

```

Figure 54: Execution Time Using Materialized View and Indexing

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

```

1 ✓ SELECT
2   ExportingNation AS "Exporting nation",
3   ExportingRegion AS "Exporting region",
4   ImportingNation AS "Importing nation",
5   ImportingRegion AS "Importing region",
6   Type,
7   SUM(Revenue) AS Revenue
8   FROM mv_export_import_revenue_region
9
10 WHERE Type = 'ECONOMY BRUSHED STEEL'
11 AND ExportingNation = 'ARGENTINA'
12 GROUP BY ROLLUP(ExportingNation, ExportingRegion, ImportingNation, ImportingRegion, Type)
13 ORDER BY "Exporting nation", "Exporting region", "Importing nation", "Importing region", Type;
14
15 -- before indexing : 56 secs 745 msec.
16 -- after indexing : 49 secs 303 msec.
17 -- after Mat View : 14 secs 108 msec.
18 -- after Mat View and indexing : 61 msec.
19
Data Output Messages Notifications
+-----+
| Data Output |
+-----+
| Successfully run. Total query runtime: 61 msec.
| 75 rows affected.

```

Figure 56: Execution Time Using Materialized View and Indexing

- 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 ($12 \text{ GB} + 15 \text{ GB} = 27 \text{ GB}$) is less than 1.5 times the size of the database (52 GB). Therefore, **the space constraint is satisfied.**

Query Query History

```

1  SELECT pg_size.pretty(pg_database_size('TPCH-Project')) AS total_db_size;
2  ✓ WITH index_size AS (
3      SELECT sum(pg_relation_size(indexrelid::regclass)) AS total_index_size
4      FROM pg_index
5  ),
6  matview_size AS (
7      SELECT sum(pg_total_relation_size(c.oid)) AS total_matview_size
8      FROM pg_class c
9      JOIN pg_namespace n ON n.oid = c.relnamespace
10     WHERE c.relkind = 'm'
11  ),
12 db_size AS (
13     SELECT pg_database_size('TPCH-Project') AS total_db_size
14 )
15 SELECT
16     pg_size.pretty(i.total_index_size) AS total_index_size,
17     pg_size.pretty(m.total_matview_size) AS total_matview_size,
18     pg_size.pretty(d.total_db_size) AS total_db_size,
19     pg_size.pretty(d.total_db_size * 1.5) AS max_allowed_size,
20     pg_size.pretty(i.total_index_size + m.total_matview_size) AS total_used_size,
21     CASE
22         WHEN i.total_index_size + m.total_matview_size < d.total_db_size * 1.5
23             THEN 'Space constraint satisfied'
24         ELSE 'Space constraint not satisfied'
25     END AS constraint_status
26 FROM index_size i, matview_size m, db_size d;
27

```

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 ($12\text{ GB} + 15\text{ GB} = 27\text{ GB}$) 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 ($12\text{ GB} + 15\text{ GB} = 27\text{ GB}$) is less than 1.5 times the size of the database (52 GB). the space constraint is satisfied.