ECOLE POLYTECHNIQUE DE BRUXELLES

BDMA : Big Data Management and Analytics

Data Warehouses

TPC-DS - PostgreSQL DBMS benchmark

Arijit Samal

Dionisius Mayr

Gabriela Kaczmarek

Jakub Kwiatkowski

Professor

Esteban Zimányi

2023

Table des matières

1 Introduction

Benchmarking in the realm of database technologies serves as a crucial tool for assessing, comparing, and improving the performance of different systems. The fundamental rationale behind benchmarking is its capacity to provide a standardized methodology for evaluating and measuring the efficiency, speed, reliability, crash recovery and other critical performance metrics of various database management systems (DBMS). By conducting benchmarks, researchers, developers, and businesses can gain insights into the strengths and weaknesses of different setups, when selecting, optimizing, or designing database systems. Furthermore, benchmarks allow us to gain a deeper understanding of deliberated tools and technologies on specific workloads and scales without delving into project-specific nuance, thus making an informed choice of the most cost-effective configuration easier at the very beginning of a new endeavour.

This report describes an educational journey to the depths of database benchmarking through implementing a widely known standard - TPC-DS. The volumes of data processed in this experiment are far from industry standards and those seen in commercial systems, however, the main aim of this project is to gain topical knowledge and properly perform a benchmark on a tool of choice, which in this case is PostgreSQL.

1.1 TPC-DS

The Transaction Processing Performance Council Decision Support (TPC-DS[21]) benchmark imitates several qualities of a decision support system and serves as a tool to provide an unbiased assessment of the System Under Test (SUT). TPC-DS encompasses a comprehensive set of queries, data maintenance activities, and concurrent user activities, simulating a mix of decision-support workloads that reflect the complexity of the actual business cases. The most important factor in this benchmark is time. The proposed metrics combine load time, query response time for a single user, query throughput with multiple emulated users, time efficiency of database maintenance

queries and more. Some of the metrics defined by the benchmark's creators involve the cost of the setup, which makes it even more business-oriented and useful for developers and administrators in real-life scenarios.

1.2 PostgreSQL

PostgreSQL (commonly known as Postgres) is an open-source, cross-platform, object-relational Database Management System (DBMS). It was created by Michael Stonebraker around 1986 as the successor of the Ingres (thus the name "Post-Ingres") [Pos23b].

It is ACID compliant, and it aims to conform with the SQL standard. Besides, it is also expandable, having many extensions to increase its capacities, such as PostGIS (for storing and querying spatial data) [Pos23a], and hstore (for storing key/value pairs) [Pos23c] among others. PostgreSQL offers a comprehensive set of functionalities, including support for complex queries, advanced SQL features, transaction management, OLAP analytics, and concurrency control, making it suitable for a wide range of applications, including data warehousing and decision-support systems.

According to DB-Engines [DB-23], Postgres is the fourth most popular DBMS, with its popularity and active community growing over time.

2 Project outline and methodology

2.1 Base TPC-DS methodology

In the benchmark specification, many aspects of the process are explained. Some of them are irrelevant to academic activity, such as the report publication guidelines and obligations, results disclosure, formal rules of agreement between the benchmark sponsor and executor etc. However, the general process flow instructions, rules of implementation, benchmark limitations and result calculation and analysis were followed closely by the project team and are described in the below paragraphs.

The core of the benchmark is a consecutive execution of six timed tests on the retail business database that is generated entirely from the electronic components from the TPC-DS downloadable package. There are two main programmatic tools - `dsdgen` and `dsqgen`, which are responsible for dataset generation and query generation, respectively. These tools utilise supplementary files, including ANSI SQL table definitions, and query templates. According to the documentation, the directory with data maintenance functions in ANSI SQL should be included in the downloadable folder, however as of the date of this publication, it is not available[23].

2.1.1 Database design

The database proposed for the benchmark execution is an example of a retail product supplier's administrative DB. It has a logical snowflake schema with multiple fact and dimension tables. According to the authors, the dimensions are divisible into three categories :

— Static : loaded at the beginning and not changing over time ;

— Non-Historical : values changing with time, and information is simply overwritten - all facts refer to the newest dimension table values ;

— Historical : also changing with time, but the historical data is recorded. Multiple rows for a

single business key value exist together, they have different periods of being active indicated through column values and newly recorded facts are assigned to the proper one based on the fact record timestamp.

Overall, the schema depicts sales, returns and inventory of a business with physical stores, catalogues and e-commerce. All three (sales, returns, inventory) are represented as fact tables, with the two first divided by sales channel - seven fact tables total. This is accompanied by seventeen dimension tables describing details, e.g. customer (non-historical), date (static), or item (historical).

2.1.2 Data generation principles

As a part of benchmark provisioning, the data is generated according to the above-mentioned logical schema. The size of the dataset is defined through the scale factor parameter 2.2.2 and can range from Giga- to Terabytes. The data is representative of actual data types that would be set for such a database in real life, however, some values are not believable. Especially the values of VARCHAR fields are automatically generated phrases that simulate actual expressions and cater to word occurrence probability distribution, but make no logical sense. According to « The Making of TPC-DS »[NP06] paper, the DB generator was mostly built based on widely known and researched Normal and Poisson distributions, so the results are reproducible. However, some crucial data has been modelled to mimic real-world behaviour. The rationale is that if that was not the case, a benchmarked scenario could be accused of irrelevance. The creators of the benchmark attempted to create an appropriate mix of the two approaches to generate a standardized yet "interesting" dataset. According to the authors of « Generating Thousand Benchmark Queries in Seconds »[PS04], the omission of skew in this (retail industry) scenario would be a bad choice. The generator is thus equipped with the comparability zones concept. These are *"essentially flat spots in the distribution"* which allow the creation of realistic but comparable data, especially in a typical business decision support system use case of year-to-year comparisons. The crucial column domains have their comparability zones defined and adjusted to meet application requirements (like year-end holiday sales spike). Information within comparability zones follows a uniform distribution pattern. This method assumes that the query design process is informed accordingly and the query construction algorithm and randomizer consider the zones. When substituting values for selectivity, group by, and order by predicates in the TPC-DS query set, the selections consistently derive from the matching comparability zone.

Specific rules from the documentation[21] were adhered to by the project team in the process

of TPC-DS data provisioning :

— *"The data processing system shall be implemented using a generally available and supported system (DBMS)",*

— *"Each table [...] shall be implemented according to the column definitions provided",*

— *"The column names used in benchmark implementation shall match those defined for each column",*

— *"Table names should match those provided in [documentation]".*

No changes were recognised as necessary, therefore nothing was modified by the team in the data generation tool, as well as in the schema, and output flat files. Other regulations from the specification are equally important in commercial TPC-DS executions, however, they are not relevant to this project as they regulate e.g. partitioning data, physical clustering of data, or data transparency requirements (physical location, access mechanisms by generally available layers, etc.)

2.1.3 Query generation principles

Queries generated by the `dsqgen` tool are designed to reflect the real-world challenges of data warehousing systems. They are purposefully written imperfectly, e.g. with unnecessary SQL structures, because not always human specialists will come up with the most optimal query design. The templates provided in the Electronically Available Specification Material are written per SQL1999 core and should not be changed beyond minor syntactic improvements in table names and references, joins, operators, aliases etc. The proposed queries mirror the complexity of business problems and utilise various patterns, styles and constraints. They can be divided into four types :

— reporting : characterised by low variance, designed to model predictable, well-known questions, about the business financial situation among others ;

— ad hoc : unexpected, dynamic queries, that are supposed to answer very specific problems ;

— iterative OLAP : capturing analysis of a more broad business case than ad hoc, like a scenario-based user query session/sequence ;

— data mining : vast, operationally heavy queries aiming to predict future trends, infer from the knowledge from database to find relationships and make assumptions for the future.

There is also another type of query set generated by the `dsqgen` - the data maintenance queries. It is a must for the decision support system to be able to periodically refresh the data and absorb new chunks of information even if the source tables are not 100% in sync with the data warehouse schema. This process therefore might require mapping and usage of views. In the benchmark,

there are tests that measure the system's ability to modify, remove and add data generated with parametrised execution of `dsdgen`.

2.1.4 TPC-DS benchmark execution rules

Six tests need to be run consecutively and successfully to consider a single benchmark execution valid :

— Database Load Test

— Power Test

— Throughput Test 1

— Data Maintenance Test 1

— Throughput Test 2

— Data Maintenance Test 2

Moreover, the SUT cannot be restarted at any point before the entire run ends.

The Database Load Test measures the ability of the DBMS to create and populate the database. There are two options for data loading : in-line and from flat files. In this project the data was loaded from flat files generated using `dsdgen`. The data provisioning process is not part of the benchmark and does not need to be timed. The crucial metric of the test is the Database Load Time and it is defined as the time delta between the timestamp taken at the beginning of DB creation and the one taken upon the population completion.

Immediately after the Database Load Test comes the Power Test. It is a singular run of the 99 queries, one by one, on a completely fresh database with no cached information. The time is measured for the entire query sequence.

The third type of test in the queue is the Throughput Test. It is supposed to emulate multiple users querying the database at the same time. There is an even number of streams run, with a permutation of all 99 queries. Its execution might take much less than roughly four times the Power Test time because the queries are repeated and the DB cache is not cleared after the Power Test. Each emulated client (stream) should have a unique identification number and on that basis - an order of queries assigned to it. The time is measured from the start of the first query execution in the first stream, up until the output for the last query in the last stream is returned.

Data Maintenance Test is the last type of test in the benchmark. It is there to assess the system's ability to alter the database. The formal rule is to execute as many refresh runs per test as the number of streams in the Throughput Test. Refresh run consists of a combination of INSERT clauses

and DELETE clauses. Insertion populates further a selection of fact tables and deletion removes some of the fact table rows based on dimension values. The time measured in this test is the time difference between a timestamp taken at the start of the first insertion and one taken after the last deletion completion.

After all the tests have been executed without errors, and all vital time measurements have been collected, one can proceed to calculate the primary performance metric of the benchmark (there is more on that topic in section 4) and additional price-related metrics if applicable.

2.2 Project assumptions

To complete the project within the time limit and with available hardware, some assumptions needed to be proposed, and efforts have been made to eliminate mistakes. All changes have been designed with the benchmark goals in mind, which resulted in little to no deviation from the original methodology.

2.2.1 Hardware specification

All the tests presented in this work were conducted on the same machine, with the specifications stated in the table 2.1.

CPU	RAM	DISK MEMORY
INTEL Core i5 9300H Base clock speed : 2.1 GHz Boost clock speed : 4.1 GHz Cores : 4 L3 cache : 8 MB	Total memory : 24 GB (8 GB + 16 GB) Dual channel configuration On board RAM (slot 1) : 8GB @ 2667 MHz Added RAM (slot 2) : 16GB @ 3200 MHz	Seagate ST1000LX015 FireCuda HDD Memory : 1 TB, 5400 RPM (Data and Results are stored here) Western Digital®PC SN520 NVMe™ SSD Memory : 256 GB (DB is stored here)

TABLE 2.1: Specifications of the machine on which benchmarks were run

2.2.2 Scale Factors

In order to measure the scalability of the Database Management System being tested, it is necessary to try to execute the benchmark with multiple data sizes. The Scale Factor (SF), also referred to as Benchmark Scaling, is a number used to indicate how big is the generated dataset. It also allows the modification of the generated queries accordingly, and the preparation of accurate maintenance queries.

A Scale Factor of 1 consists of approximately [1] 1GB of data, and it scales linearly (i.e. a Scale Factor of 2 would be around 2GB and so on). It is important to understand that not only the *amount* of data is increased, but also the *domain* of the data, meaning that the set of possible values for a given column can expand on higher SFs.

The complete specification of TPC-DS uses SFs of 1000, 3000, 10000, 30000 and 100000 for the benchmark. However, this is too big for casual computers and thus in this work, the scale factors of 1, 3, 5 and 6 were used. In Section 4 it is possible to see a comparison of its results.

Lastly, it is strongly discouraged to compare results between different scale factors because the challenges posed by various volumes of workload change significantly. For instance, if the growth of the row quantity is not identical in all tables (i.e. one table grows in size and another one expands the domain but size remains somewhat static) the ratio of table sizes changes and the query planner will approach the queries differently. Scale factor changes and the aforementioned ratio will be especially important in columns that participate in `selectivity`, `order by` or `group by` predicates. Their volume influences the anticipated performance of join algorithms, memory and disk usage, or sorting algorithms, and therefore the query planner might have different outcomes for each SF. That said, even though the project team is aiming to meet the specification requirements and mimic actual benchmark execution as closely as possible, the results of this experiment with different SFs are presented together and loosely compared. The reason is that the aim here is not to compare the values directly and to draw conclusions about e.g. the setup choice, but to experiment and observe how the size of DB influences the complexity and, in consequence, time consumption.

2.2.3 Other project assumptions

In the specification, it is determined that the number of streams in the throughput test must be an *"any even number larger or equal to 4"*. For the sake of the project, the number of streams was set to minimal (4) because of the limited hardware capabilities and relatively small dataset.

Moreover, some not purely syntactic changes were introduced to the templates for data maintenance queries. The CREATE TABLE template was not present for `s_web_returns` - one of the source tables. Its contents needed to be inferred from the documentation and the proper template was created by the team. There were also issues with the primary keys, which required creative problem solving as it was not supposed to be a problem to resolve for the DBMS, as per clause 5.1.3 of the specification :

1. According to section 3.1.1 of the TPC-DS specification : “The actual byte count may vary depending on individual hardware and software platforms.”

"Implementers can assume that if all DM operations complete successfully that the PK/FK relationship is preserved. ***Any exceptions are bugs that need to be fixed in the spec.****"*

There were multiple technical issues with every run of the maintenance queries set, so given the informality of the benchmarking and the dataset size, it was reasonable for the project team to deviate from the specification and settle for one refresh run per test instead of the original four.

All guidelines related to hardware were also simplified. Many policies from the spec regulate what pieces of hardware can and cannot be used, on what principles the data can be physically divided, how nothing can happen between tests, there should be no programmatically implemented waiting times, and even how much knowledge the system administrator can have about the TPC-DS data prior to the test execution. All these are in place mostly because usually, the test sponsor and the tested subject are two different entities and their interest could be different. The sponsor expects entirely objective scoring and the DBMS vendor wants to have the highest score, which leaves room for potential malpractice or intentional misinterpretation of too lenient or vague requirements. In the case of this project, the only involved entity is the project team with no interest in falsifying the scores. Thus, although the process was automated, there was no need for additional security and transparency measures. Since the results are also not published as an official TPC-DS document, many clauses of full code disclosure could be omitted and only the most interesting parts were included in this report.

3 Implementation

3.1 Data Generation

In order to populate the benchmark database, it is necessary to generate two sets of data for each Scale Factor :

1. Base data : contains the data used to populate the relations of the TPC-DS benchmark;
2. Maintenance data : contains the data that is inserted during the maintenance phase and also files used to specify delete intervals.

These two datasets are produced through an automated script (`./scripts/data_generation.sh`), that makes use of the `dsdgen` tool to generate data in parallel. This script receives a Scale Factor as an argument and populates the folder specified by the local variable `TARGET_DIR` with both the base data and the maintenance data.

The random seed used to generate the data was fixed to the value 1 so that the results achieved by this work can be reproduced by anyone.

3.2 Query and Code Generation

Similarly to the data generation step, it is also necessary to generate the queries to be used by the benchmark for each scale factor.

The queries are created through another automated script (`./scripts/query_generation.sh`), that makes use of the `dsqgen` tool to generate the queries. This script receives a Scale Factor as an argument and populates the folder specified by the local variable `OUTPUT_DIR` with the generated queries.

3.2.1 COPY statements generation

In order to load the data into the DBMS, the `COPY` command is used. This command receives the name of the relation where to insert the data, a path to the location of the raw data itself and also some parameters regarding the format of the data file. See the example below :

```
1 COPY call_center FROM '<PATH_TO_RAW_FILE>' DELIMITER '|' NULL '' ENCODING '
    LATIN1';
2 COPY catalog_page FROM '<PATH_TO_RAW_FILE>' DELIMITER '|' NULL '' ENCODING '
    LATIN1';
```

To generate these statements programmatically, the script `generate_load_data.sh` was developed.

3.3 Query Preparation

The queries generated by the tool aren't ready to be used yet. This Section describes the required changes that need to be made to the queries.

3.3.1 Dialect

Since there are dozens of different SQL dialects (and more yet to be created), the `dsqgen` tool does not support all dialects that exist. At the moment, only ANSI, DB2, Netezza, Oracle, and SQL-Server are supported. Since Postgres isn't on the list, it was necessary to generate the queries using the most similar dialect (which in this case was Netezza) and make the required corrections.

3.3.2 Syntax corrections

All the corrections performed in this step were only syntactical and didn't affect the meaning nor the reasoning behind the query.

The generated queries from the Netezza dialect had 19 queries with errors that were not in accordance with the PostgreSQL syntax. Those queries were fixed to be consonant with the PostgreSQL syntax. All the errors found in those queries and their fixes are mentioned below :

1. Queries 5, 12, 16, 20, 21, 32, 37, 40, 77, 80, 82, 92, 94, 95 and 98 showed a simple error regarding dates. When a query selected data where certain columns were within a date range, the Netezza dialect did not need an `interval` keyword. PostgreSQL, on the other hand, needs it to perform the date range operations. Hence, the `interval` keyword was added to fix these queries. For example :

Wrong query : `SELECT * FROM table WHERE date BETWEEN cast('1998-08-04' AS date) AND (cast('1998-08-04' AS date) + 14 days).`

Right query : `SELECT * FROM table WHERE date BETWEEN cast('1998-08-04' AS date) AND (cast('1998-08-04' AS date) + interval '14 days');`

2. Query 30 had an error because it tried to select a column from the `customer` table, **c_last_review_date_sk**, which wasn't defined in the table. This was fixed by changing the `SELECT` command to select **c_last_review_date** column, which was part of the customer table;

3. Queries 36, 70, and 86 had errors in finding the column `lochierarchy`, which was referred to while ordering the selected columns using the `ORDER BY` statement. This error occurred in the query because `lochierarchy` was an alias given to a grouping and not an actual column by which the data can be ordered. So a common table expression (CTE) was used and put the `SELECT` commands with the `lochierarchy` grouping in the table. This table now has the `lochierarchy` as one of the columns. Then, all the columns of this CTE are selected and ordered on the `lochierarchy` constraints as it was mentioned in the original queries.

3.3.3 Query Optimizations

Strictly speaking, one is not allowed to optimise the queries of this benchmark, since part of the idea behind it is to assess the capacity of the DBMS query optimizer to identify these optimizations automatically. However, due to hardware constraints, it wouldn't be feasible to run the benchmark for multiple Scale Factors (let alone run it multiple times for each factor).

In order to get an idea about the running time of the queries, a dry run of the test was performed, using initially a Scale Factor of 1. The longer queries were identified, their query plans analysed, the optimizations performed, and the time was measured again to compare it with the original version of the query. Since the performance of the queries can vary depending on the SF used, whenever a query was taking too long to run on a higher scale factor, this procedure was repeated.

Thus, the most time-consuming queries were optimized, and a list of the alterations performed, as well as an explanation of them, can be found in Appendix B.

From our preliminary experiments, it was found that some queries were taking more than an hour to execute, even for the Scale Factor of 1. Those queries were optimized such that their overall execution time decreased, allowing us to run them for bigger scale factors.

Their query plans were plotted using a query plan visualizer called Dalibo [DAL23] and then

analyzed extensively to optimize the queries. The queries optimized and their optimization are as follows :

1. Query 1 used cross products to join tables and this is very costly for tables with many tuples. It also had a subquery used to calculate the average returns and select those customers having more than 20% of usual returns. There was also a CTE in this query whose scan took a lot of time, as shown in figure 6.1 due to the use of the subquery. The query was optimised by creating a CTE for average customer returns (and thus removing the need for the subquery), using an index on the column on which aggregate functions were applied and converting the cross products to inner joins on the referential keys of the tables. This optimization decreased the CTE scan time significantly which can be seen in figure 6.1.
   Disclaimer : The figures of the query plans provided for comparison are not the complete query plan. They only compare the optimized parts.
2. Queries 4, 11, and 74 had the same issues. They were creating a CTE and inside this CTE appending 2 (in query 11 and 74) or 3 (in query 4) tables using a `UNION ALL` operator. This was taking a lot of time as shown in figure 6.3, 6.5 and 6.7. Moreover, they were performing cross-products of the tables and using range conditions of dates. These queries were fixed by creating an index on the columns used to calculate the aggregate functions. The appended tables were disjoined and separate CTEs for all of them were created. The query and the query plan were analyzed and the logic was changed so that there was no need for a union operation anymore. Lastly, the cross-products were converted to inner joins. After the optimization, the CTE scan timings improved drastically, as shown in figure 6.4, 6.6 and 6.8.
3. Query 6 had two subqueries that slowed down the execution of the query as shown in 6.9. The original query used the d_month_seq column being equal to the result of a subquery, but it was redundant. In the revised query this redundancy was removed by directly filtering on the date (e.g. where year = 2000 and month = 2). The other subquery was calculating the average of the sales over a category of products. This issue was fixed by creating a CTE and selecting the average from the CTE when required. The cross products were also converted to inner joins and indices on the columns having aggregate functions were created to speed up the process. The plan after optimization is shown in figure 6.10.
4. Query 72 had one issue. It had a redundant join operation, which was not used anywhere, so the extra join operation was removed and therefore the execution time was enhanced. An index on the columns used for calculating the aggregate functions was also created to further improve the times. Here, the plans remained almost identical but the execution times

decreased after optimization.

5. Query 95 had a subquery performing a cross-product between a table and a CTE defined in the query. The CTE scan took a lot of time in the original query as shown in figure 6.11. The query was fixed and accelerated by separating the subquery, making it as a CTE and then using it for the conditions. Joins were also used instead of cross-products to speed up the query execution time. Moreover, there was a redundant condition in the query. One condition was meant to find order numbers that are in the `ws_sh` table, and the other found order numbers in the subquery, which was made a CTE `wr` after optimization that was formed by joining the `ws_sh` table with the `web_returns` table. The `wr` CTE contained equal or fewer tuples because of the join, and hence, only this condition is necessary to join. The decision was made to remove the condition that searched for order numbers in the `ws_sh` CTE, and only keep the condition where it was searched in a small space of the `wr` CTE. In addition, indices were also utilised to speed up the queries.

3.3.4 Insertions

The Data Maintenance step of the test simulates data insertion in a real-world scenario, where multiple problems can arise. For instance :

— Insertion of (duplicate) data already present in the table ;

— Insertion of rows with the same primary key ;

— Insertion of rows violating a constraint (e.g. `NON-NULL`).

There are multiple approaches to solve these issues, one of which is to perform an `UPSERT`. An `UPSERT` is the combination of `UPDATE` and `INSERT`, where the former is only performed if a tuple already exists on the target relation.

This is achieved in Postgres by using the `ON CONFLICT` clause of an `INSERT` statement [Pos23d]. This clause not only allows the DBMS to avoid errors but also to implement a policy on how to deal with this kind of insertion performed by the users. One example policy could be “always update the existing row”. For example :

```
1 INSERT INTO catalog_sales   SELECT * FROM csv WHERE cs_item_sk IS NOT NULL
    ON CONFLICT (cs_item_sk, cs_order_number) DO UPDATE SET
2    cs_sold_date_sk           = excluded.cs_sold_date_sk,
3    cs_sold_time_sk           = excluded.cs_sold_time_sk,
4    cs_ship_date_sk           = excluded.cs_ship_date_sk,
5    cs_bill_customer_sk       = excluded.cs_bill_customer_sk,
```

```
    cs_bill_cdemo_sk          = excluded.cs_bill_cdemo_sk,
    cs_bill_hdemo_sk          = excluded.cs_bill_hdemo_sk,
    cs_bill_addr_sk           = excluded.cs_bill_addr_sk,
    cs_ship_customer_sk       = excluded.cs_ship_customer_sk,
    cs_ship_cdemo_sk          = excluded.cs_ship_cdemo_sk,
    cs_ship_hdemo_sk          = excluded.cs_ship_hdemo_sk,
    cs_ship_addr_sk           = excluded.cs_ship_addr_sk,
    cs_call_center_sk         = excluded.cs_call_center_sk,
    cs_catalog_page_sk        = excluded.cs_catalog_page_sk,
    cs_ship_mode_sk           = excluded.cs_ship_mode_sk,
    cs_warehouse_sk           = excluded.cs_warehouse_sk,
    cs_promo_sk               = excluded.cs_promo_sk,
    cs_quantity               = excluded.cs_quantity,
    cs_wholesale_cost         = excluded.cs_wholesale_cost,
    cs_list_price             = excluded.cs_list_price,
    cs_sales_price            = excluded.cs_sales_price,
    cs_ext_discount_amt       = excluded.cs_ext_discount_amt,
    cs_ext_sales_price        = excluded.cs_ext_sales_price,
    cs_ext_wholesale_cost     = excluded.cs_ext_wholesale_cost,
    cs_ext_list_price         = excluded.cs_ext_list_price,
    cs_ext_tax                = excluded.cs_ext_tax,
    cs_coupon_amt             = excluded.cs_coupon_amt,
    cs_ext_ship_cost          = excluded.cs_ext_ship_cost,
    cs_net_paid               = excluded.cs_net_paid,
    cs_net_paid_inc_tax       = excluded.cs_net_paid_inc_tax,
    cs_net_paid_inc_ship      = excluded.cs_net_paid_inc_ship,
    cs_net_paid_inc_ship_tax  = excluded.cs_net_paid_inc_ship_tax,
    cs_net_profit             = excluded.cs_net_profit;
```

In the example above, the pair `(cs_item_sk, cs_order_number)` is the primary key of `catalog_sales`, thus avoiding any problematic duplicates being inserted.

We also faced an error due to the insertion of null tuples into the tables where the columns are keys. This error, specifically, occurred in scale factors greater than five and needed to be fixed to measure the maintenance test timings. The keys can not be null, and this violates the entity integrity constraints. Therefore, to tackle this error, we use a not null clause on the column of the view being inserted into the primary key of the table, as mentioned in the SQL code above.

This policy was chosen arbitrarily, and in a real-world application, it would need to be tailor-made for each scenario.

3.4 Instructions for replication

Along with this document, a zipped folder containing all the scripts, code and tools used to perform the benchmark is being provided. There one will find everything needed to replicate the results presented in this section.

3.4.1 Python and run_all.sh scripts

Besides the shell scripts mentioned previously, Python scripts were also created to manage the communication with PostgreSQL, measure the execution times and simulate concurrent users.

The `run_all.sh` script makes use of the above-mentioned Python scripts, running them in the order specified as follows :

1. `load_test.py`
2. `power_test.py`
3. `throughput_test.py`
4. `maintenance_test.py`
5. `throughput_test.py`
6. `maintenance_test.py`
7. `db_postprocess.py` (Dropping the DB)

Regarding the maintenance script, it uses pre-made queries to create temporary tables, load those tables, create views that "point" to those tables in order to correct their schema and make they compatible with the base tables, and finally copy the data to the base tables as explained in subsection 3.3.4.

3.4.2 Usage instructions

Find below a set of instructions to run the benchmark.

1. Set the paths accordingly in the four following scripts `data_generation.sh`, `query_generation.sh`, `generate_load_data.sh` and `run_all.sh`;
2. Generate code and data :
   (a) Generate the data for the specified Scale Factor such as
   `./data_generation.sh <SF>`

(b) Generate the queries for the specified Scale Factor such as

`./query_generation.sh <SF>`

(c) Generate the `COPY` statements to load the data for the specified Scale Factor such as

`./generate_load_data.sh <SF>`

3. Run the benchmark for the specified scale factor and number of repetitions using

`./run_all.sh <SF> <ITERATIONS>`

As a result, the folder `Results_sf_<SF>` will be created, containing .csv files with all the query results and .txt files with all test timings.

3.4.3 Directory structure

```
Copy_data
├── data_sf_<SF>.sql
Data
├── data_sf_<SF>
├── maintenance_data_sf_<SF>
maintenance
├── maintenance_data_sf_<SF>
|   ├── method_1
|   |   ├── 0_tpcds_source.sql
|   |   ├── 1_populate_s_tables.sql
|   |   ├── 2_create_views.sql
|   |   ├── 3_free_tuples_present_on_views.sql
|   |   ├── 4_populate_main_tables_1.sql
|   |   └── 5_drop_s_tables_and_views.sql
|   ├── method_2
|   |   └── 0_delete.sql
|   └── method_3
|       └── 0_inventory_delete.sql
queries
|   ├── queries_sf_<SF>
|   |   ├── 99 sql queries (not optimized)
|   ├── optimized_queries_sf_<SF>
|   |   └── 99 sql queries (optimized)
query_optimization
├── optimized_queries
├── query_plans_before_optimization
├── query_plans_after_optimization
query_templates
├── Templates of 99 queries and dialect
Results_sf_<SF>
├── Contains the result of a particular SF
scripts
├── data_generation.sh
├── db_postprocess.py
├── generate_load_data.sh
├── load_test.py
├── maintenance_test.py
├── power_test.py
├── query_generation.sh
├── run_all.sh
└── throughput_test.py
tools
├── dsdgen (used for data generation)
├── dsqgen (used for query generation)
└── tpcds.sql (used to create tables in the DB)
```

FIGURE 3.1: Directory Structure

4 Results

4.1 TPC-DS and custom metrics

The TPC-DS specification defines a set of metrics for assessing the System Under Test (SUT). This project focuses on one of the main metrics, which is a *Performance Metric*. The Performance Metric evaluates the capability of the system based on the times captured during runs of different tests, as well as the Scale Factor, number of streams and queries. Some of those are listed as secondary TPC-DS metrics, including :

— Load Time T_{Load} - time elapsed for the Load Test execution,

— Power Test Elapsed Time T_{Power},

— Throughput Test 1 and Throughput Test 2 elapsed times.

The Performance Metric is defined as :

$$QphDS@SF = \left\lfloor \frac{SF * Q}{\sqrt[4]{T_{PT} * T_{TT} * T_{DT} * T_{LD}}} \right\rfloor$$

Where :

— SF - Scale Factor used in timed run;

— Q - total number of weighted queries, defined as a product of the number of streams in the Throughput Test and number of queries per stream;

— T_{PT} - value calculated from the formula $T_{PT} = T_{Power} * S_q$, where T_{Power} is Power Test Elapsed Time, and S_q is a number of streams used in Throughput Test;

— T_{TT} - sum of times needed for completion of the two Throughput Tests;

— T_{DT} - sum of times needed for completion of the two Maintenance Tests;

— T_{LD} - load factor, computed as $T_{LD} = 0.01 * S_q * T_{Load}$, where T_{Load} is Load Time, and S_q is a number of streams used in Throughput Test.

The other two main metrics are : *Price-Performance metric* and *System availability date*. They are not applicable in the context of this project, since it was not an execution formally demanded by the benchmark sponsor and the SUT was not prepared exclusively for the benchmarking purposes. Similarly, the secondary TPC-DS energy metric is also not applicable in the case of this project.

To fully understand the TPC-DS process and what is impacting results the most, further analysis was conducted. Apart from the chosen TPC-DS metrics, times for all the queries in each test were measured and studied.

4.2 First power test observations and consequences

The first analysis that was done as a part of the project was a study of the Power Test for Scale Factor 1. The results (times measured for each query) are displayed in the figure 4.1.

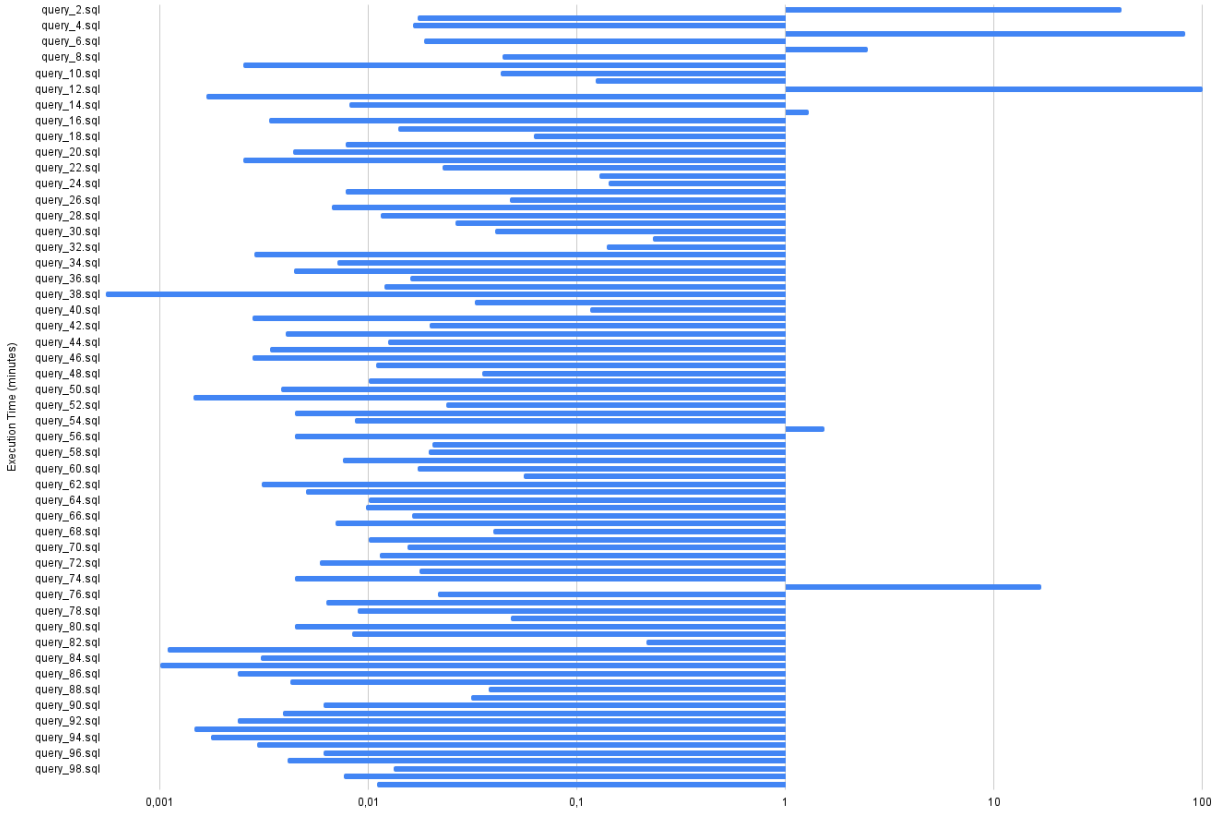

FIGURE 4.1: Power test queries times in minutes (log scale)

It is evident that most of the queries were able to run under 1 minute. There were four outlying queries, all of which took longer than 10 minutes :

— query_1.sql - 41 minutes,

— query_4.sql - 82,5 minutes,

— query_11.sql - 100,1 minutes,

— query_74.sql - 17 minutes.

After those four, the second longest time captured was 148,5 seconds. Apart from that, only four other queries were taking longer than 10 seconds, while around 62% of all queries took less than 1 second.

Other statistics also show a huge impact of the four outlying values from the rest of the query set. The differences can be seen in the tables 4.1 and 4.2 - with and without the outliers, respectively.

Quartile 0	**Quartile 1**	**Quartile 2**	**Quertile 3**	**Quertile 4**	**Mean**	**Variance**
0,033	0,262	0,601	1,905	6006,462	150,355	667893,828

TABLE 4.1: Stats with outliers

Quartile 0	**Quartile 1**	**Quartile 2**	**Quertile 3**	**Quertile 4**	**Mean**	**Variance**
0,033	0,256	0,599	1,379	148,499	4,763	374,437

TABLE 4.2: Stats without outliers

It is apparent that the occurrence of the outlying queries drastically impacts the distribution of elapsed times. For that reason, queries : query_1.sql, query_4.sql, query_11.sql, query_74.sql needed a special approach, which is thoroughly detailed in section 3.3.3. During the optimisation process, fixes included also query_6.sql, query_72.sql and query_95.sql.

4.3 Results of runs by test type

As a consequence of the first power test conclusions, two approaches to fix the problematic queries have been proposed : they could either be optimised, or removed and not considered in the results (there is no need for imputation as the final value of the benchmark's metric is weighed by the number of queries in the run). A series of benchmark runs for different scale factors was executed and the results have been developed according to both strategies. Times were measured and averaged for six runs, to reduce the impact of outliers and potential hardware malfunctions to get the representative measurements.

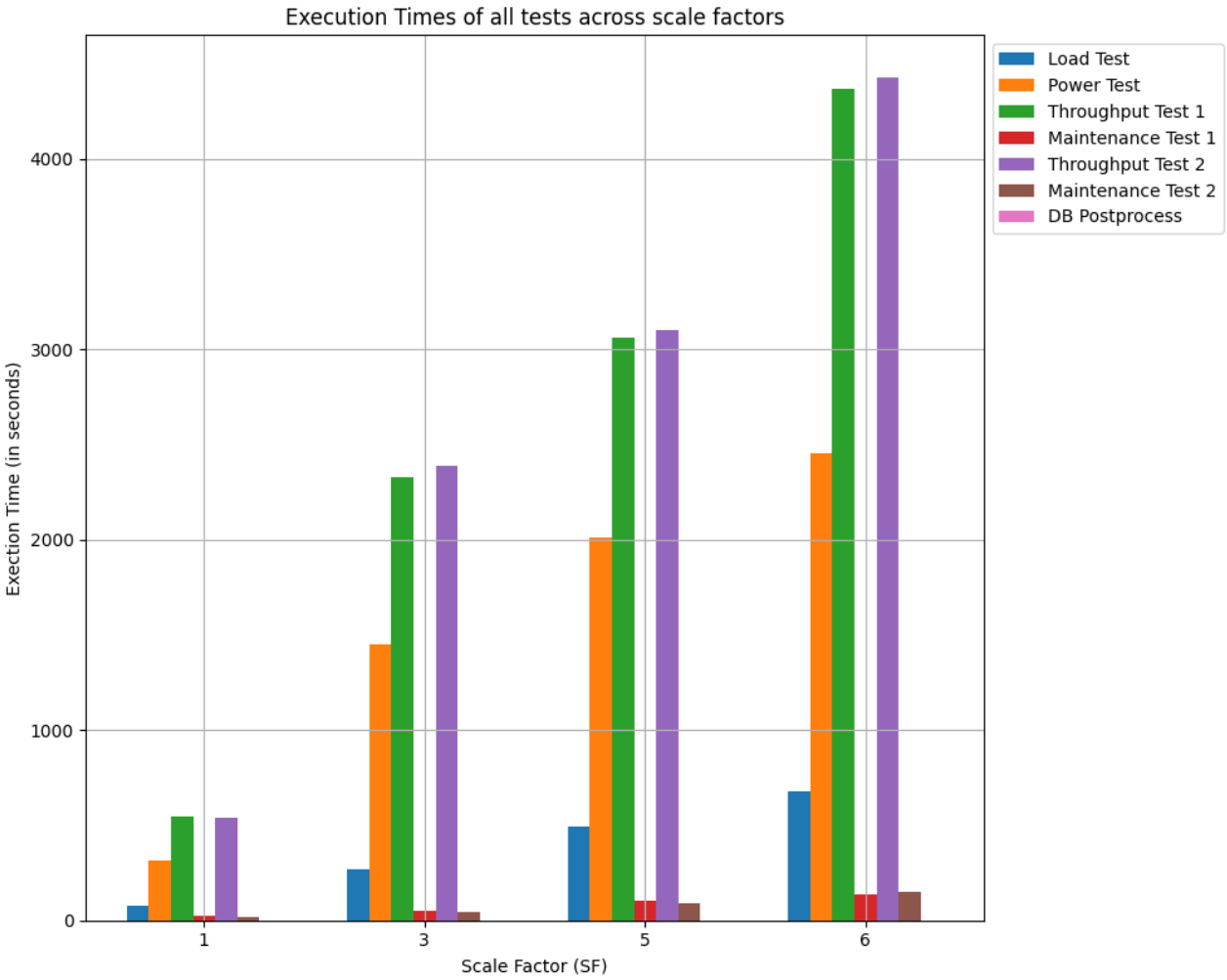

FIGURE 4.2: Comparison of times measured for each test for different SF

4.3.1 Load Test

Figure 4.3 depicts the progression of the time required to create and populate the database on different dataset volumes.

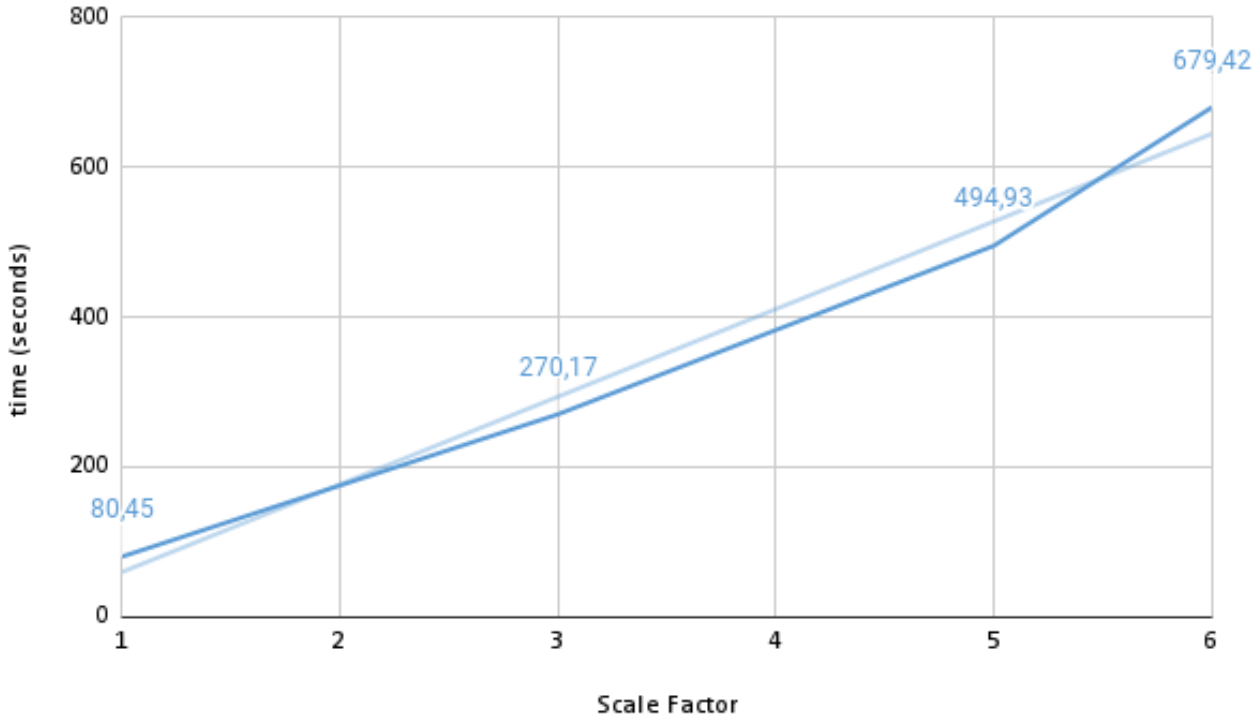

FIGURE 4.3: Load Test times for different SF (in seconds)

The trend on the chart is not very prominent - on one hand, it is almost linear, but on the other - it goes upwards sharper towards the end of the figure. This leaves room for speculation that with higher volumes, which could not be processed by a personal computer in this experiment, the trend could bear resemblance of exponential growth. In the table 4.3 the average, minimal and maximal values of time measurements are compared.

	Average	Minimal	Maximal
SF1	80,445	77,324	83,6
SF3	270,171	259,683	291,735
SF5	494,933	485,382	509,836
SF6	679,42	677,391	681,449

TABLE 4.3: Variability of Load Test times between averaged runs for different SF (in seconds)

It was observed that as expected, the CREATE statements always take approximately the same amount of time, no matter the scale factor. The mean amount for the database creation is 0,071 +-

0,009 s, and for the scale factors 1, 3, 5 and 6 they take respectively 0,088%, 0,026%, 0,014%, 0,01% of the entire load time.

4.3.2 Power Test

After the modifications that followed the initial Power Test run, a significant reduction of times for outlying queries was registered :

— query_1.sql - 41 minutes ⟶ 0,22 seconds ;

— query_4.sql - 82,5 minutes ⟶ 0,19 seconds ;

— query_11.sql - 100,1 minutes ⟶ 69,09 seconds ;

— query_74.sql - 17 minutes ⟶ 59,08 seconds.

In the figure 4.4 average individual query execution times are depicted for the scale factor 1.

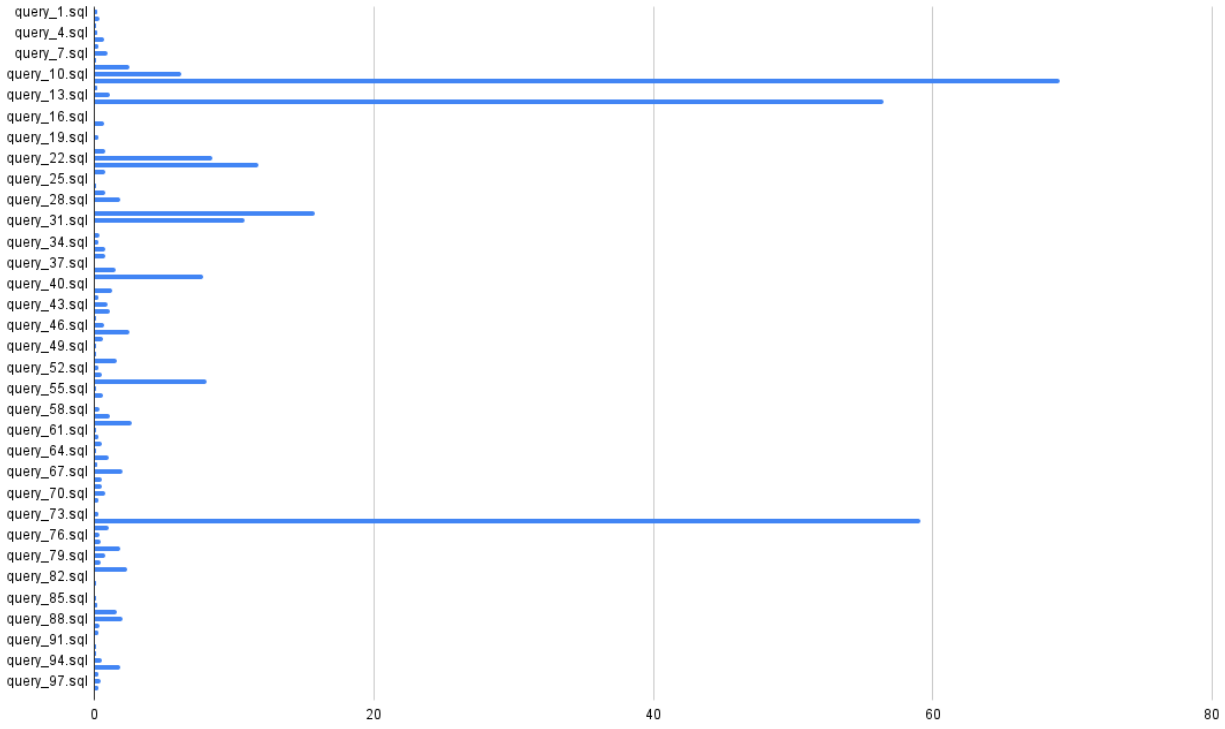

FIGURE 4.4: Query times after optimisation (in seconds) @SF1

Although notable drops in the execution times were observed, two out of four initially problematic queries still are among the most time-consuming operations. In the figure 4.5 the Power Test times for each scale factor are shown.

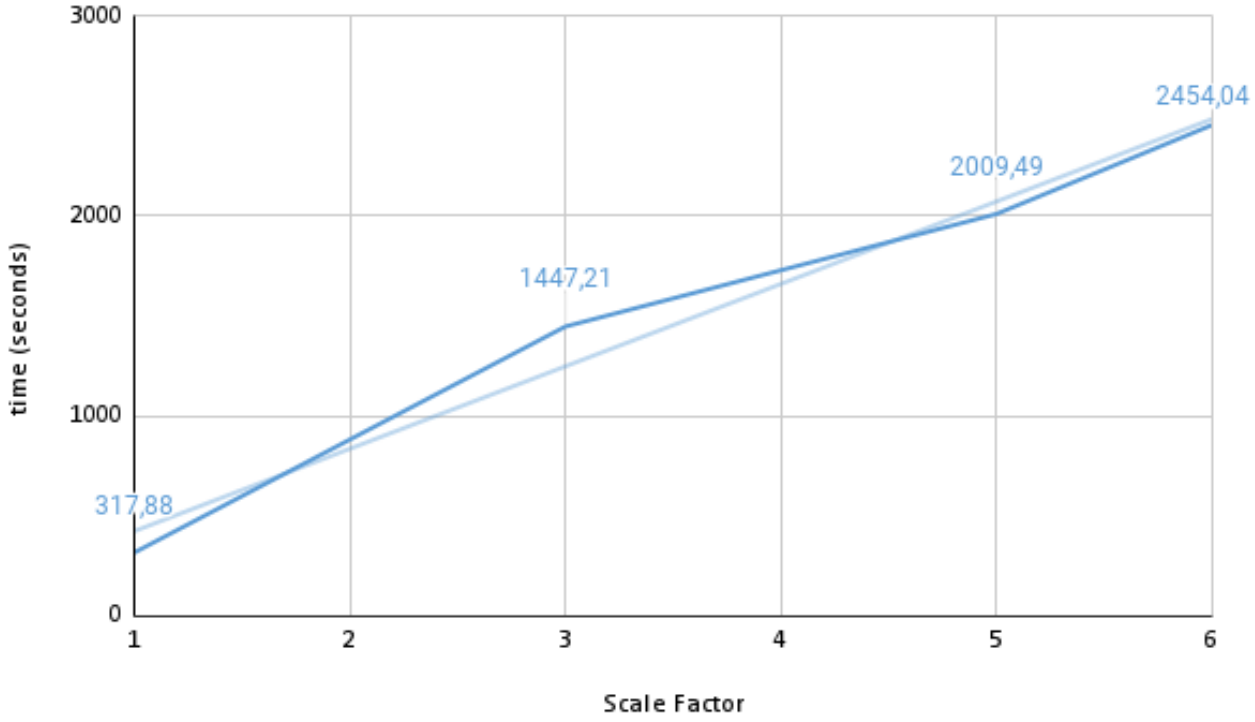

FIGURE 4.5: Power Test times for different SFs (in seconds)

	Average	Minimal	Maximal
SF1	317,880	287,083	357,073
SF3	1447,205	1412,268	1516,126
SF5	2009,486	1983,654	2037,051
SF6	2454,044	2365,986	2543,113

TABLE 4.4: Variability of Power Test times between averaged runs for different SF (in seconds)

Interestingly, the chart appears to be linear and even the slight variance does not suggest what could be expected upon further volume growth.

The growing trend for the whole Power Test is not visible in the analysis of the separate queries. For each query, the proportion of elapsed time for SF6 to elapsed time for SF1 was calculated and it was discovered that the value of this ratio varies from 0,35 to 3560. This essentially means that all queries have their unique progressions, sometimes the time increases (even 3560 times), but in other cases the performance on higher data volumes is, counter-intuitively, better. There are examples of consistent growth and almost constant results, but also of decline or irregular changes. This could potentially be a result of the hybrid (domain and row quantity) approach to the expansion of the dataset, which was discussed in the section 2.2.2.

	Average	Maximal	Minimal
SF1	3,1041	69,0887	0,0036
SF3	14,5588	337,7507	0,0347
SF5	20,5045	343,5151	0,0321
SF6	25,5613	571,7725	0,0397

TABLE 4.5: Variance of single queries in Power Test for different SF (in seconds)

4.3.3 Throughput Tests

Like it was anticipated in the section 2.1.4, the results of Throughput Tests, executed twice per each run of the whole benchmark, show the impact that a caching mechanism has on the database performance. Even though four times as many queries were run as in the Power Test (99 queries per each of 4 streams) the total times for each SF in the Throughput Tests do not exceed twice that of their Power Test counterparts. In the figure 4.7 the results are summarised, and in the table 4.7 the variability can be observed.

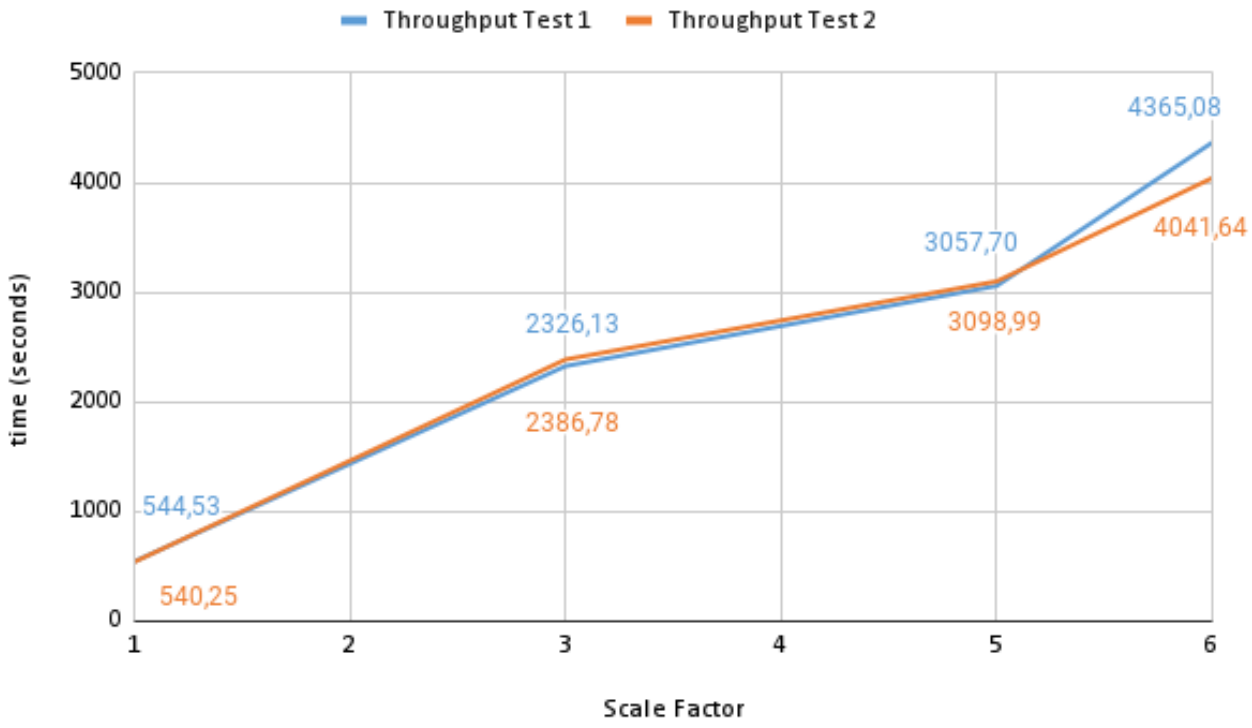

FIGURE 4.6: Throughput Tests times for different SF (in seconds)

	Average	Minimal	Maximal
SF1	1084,782	1033,644	1148,046
SF3	4712,911	4458,213	5097,143
SF5	6156,694	6079,861	6192,401
SF6	8406,720	8102,121	8711,318

TABLE 4.6: Variability of Throughput Tests summed up times for different SF (in seconds)

4.3.4 Maintenance Tests

Maintenance Tests are supposed to be run twice for every benchmark process execution. Results of the tests are shown in Figure 4.7, where average times for Maintenance Tests 1 and 2 are plotted separately.

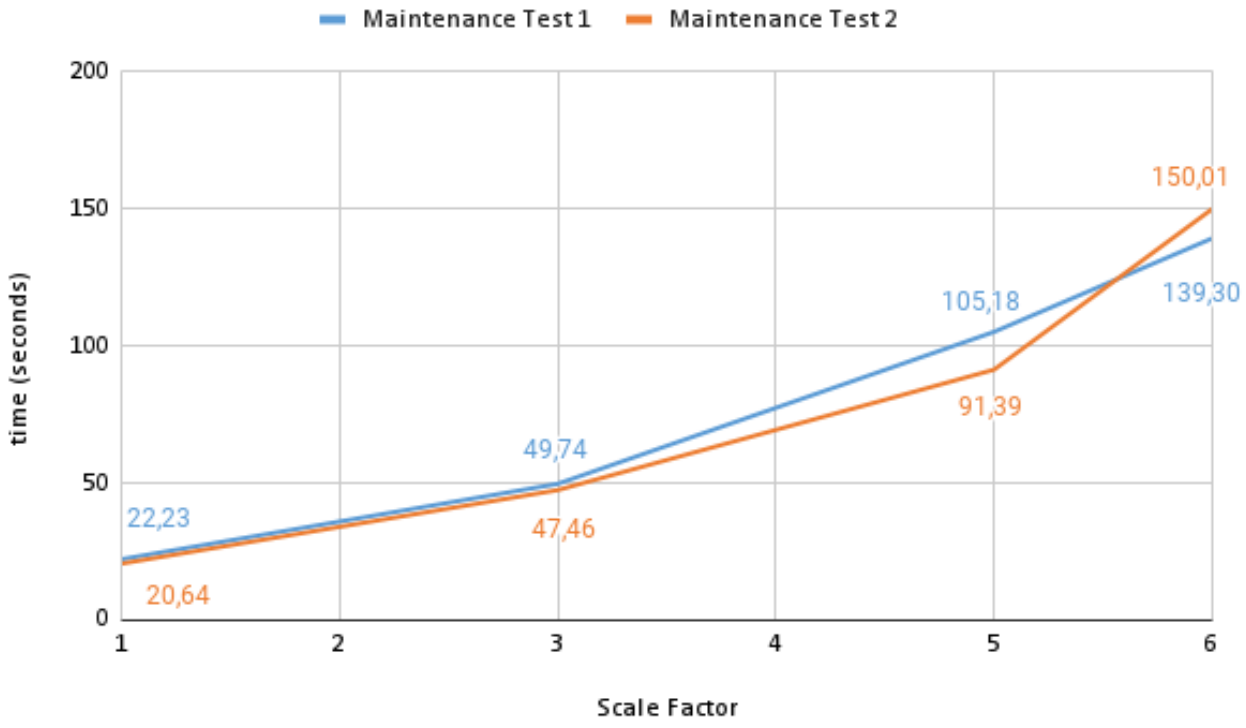

FIGURE 4.7: Maintenance Test times for different SF (in seconds)

The growing tendency is visible for both tests, but there is no obvious correlation between the first and second execution in the benchmark. The times vary more for SF5 and SF6, but to check it properly it would require more repetitions of the studied runs. The maximal variability of elapsed times for each Scale Factor was growing for each scale factor.

	Average	Minimal	Maximal
SF1	42,875	39,859	46,945
SF3	97,204	91,988	105,070
SF5	196,577	170,622	229,111
SF6	289,317	274,360	304,274

TABLE 4.7: Variability of Maintenance Tests summed up times for different SF (in seconds)

Maintenance Test consists of three refresh methods : fact insertions, fact deletion and inventory deletion. The results were studied for the purpose of analysing which part of the refresh actions is contributing the most to the whole Maintenance Test elapsed time :

— Method 1. Fact insert : 84,7% - 86,7% ;

  — Views operation : 30% - 49% of method 1. (decreasing with Scale Factor)

  — Rows insertion : 48% - 66% of method 1. (growing with Scale Factor)

— Method 2. Fact delete : 11,5% - 13% ;

— Method 3. Inventory delete : 1,8% - 2,3% of maintenance test time.

4.4 Benchmark results for the Performance Metric

The main Performance Metric can be calculated differently for the approach with omitting the outlying queries and using the optimised queries. The value of the Q factor differs between the two approaches. For optimised queries $Q = S_q * 99$, where 99 is the number of queries per Throughput Test stream. In the runs with omitted queries, this value is equal to 92, since seven queries were not used. S_q (number of streams in Throughput Test) stays the same and is equal to four.

It is also worth mentioning that since the values of T_{PT}, T_{TT}, T_{DM} and T_{LD} were originally measured in seconds (with a resolution of 1 microsecond), they need to be converted to decimal hours for calculating the final metric value.

Values required to calculate the metric for Scale Factor 1 are shown in the table 4.8 side-by-side for those two approaches.

	Omitted queries	Optimised queries
SF	1	1
S_q	4	4
Q	368	396
T_{Power}	0,049	0,0883
T_{PT}	0,196	0,35320
T_{TT}	0,1766	0,30133
T_{DT}	0,01191	0,01191
T_{Load}	0,02235	0,02235
T_{LD}	0,00089	0,00089
$QphDS@1$	16090	12137

TABLE 4.8: Values connected to Performance Metric for SF1

The main Performance Metric has been calculated for all the Scale Factors using the average values of measured times. The final results of the Performance Metric are presented in the table 4.9.

	Omitted queries	Optimised queries
SF1	16090	12137
SF3	11472	10394
SF5	11322	10759
SF6	9757	9530

TABLE 4.9: Performance Metric values with different SF and approaches

The results of the Performance Metric show that higher test result values outweigh the impact the SF variable has on the final metric value, meaning that the SF in the formula's numerator does not compensate for the denominator's incrementation. This might be yet another reason why direct comparison of results between scale factors is discouraged. For bigger Scale Factors the metric is declining for both approaches. Results for the omitting strategy are better than their corresponding optimised parts, but they are getting closer for bigger Scale Factors. It shows that the problematic queries have more impact on lower Scale Factors and they could be further optimised to even the results for all the queries. It is important to remember that Performance Metric is not formally defined for any number of queries different than 99, so the values in the column with omitted queries should not be considered as formal Performance Metric results.

It is worth mentioning that the TPC-DS Specification states : *"The TPC believes that comparisons of TPC-DS results measured against different database sizes are misleading and discourages such comparisons"* and requires all the results to be accompanied by this disclaimer. It is possible to theorize based on the results for different Scale Factors, but in reality, the justification of the observed patterns may be more intricate than simply stemming from table row number increase.

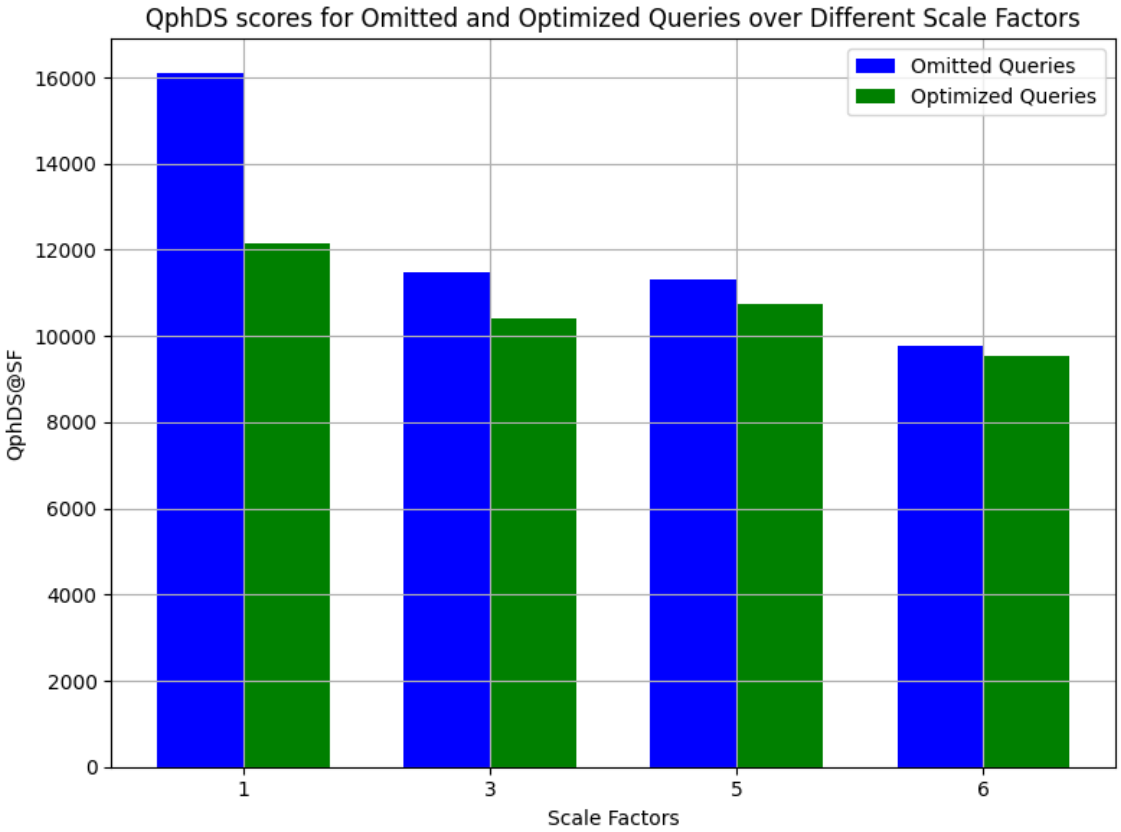

FIGURE 4.8: Performance metric over different scale factors

5 Conclusion

The TPC-DS benchmark for PostgreSQL was performed, where it proved to be an excellent general-purpose Relational Database Management System. Although initially, some queries were taking a long time to run, it was possible to analyse their query plans and easily optimize them. Exceptionally, a speedup ratio of over 25000 was obtained for query 4. Definitely, Postgres query optimizer has margin to improve, for instance detecting where to change cartesian products to inner joins.

The project team was able to reproduce the complete TPC-DS benchmark process, measure the times of all the tests and calculate the Performance Metric. In addition, extensive research on the principles behind the inner workings of the TPC-DS was conducted and summarized. Postgres seemed to scale linearly on the Scale Factors tested, both on the Throughput test and on the Maintenance test.

Not only that, but Postgres also showed pragmatic features regarding handling data maintenance with ease. It has fully functional concurrent control, provides a working `UPSERT` functionality and is also able to ensure integrity constraints strictly. All in all, Postgres appears to be a solid DBMS choice for implementing a Data Warehouse.

6 Appendix A - Query plans

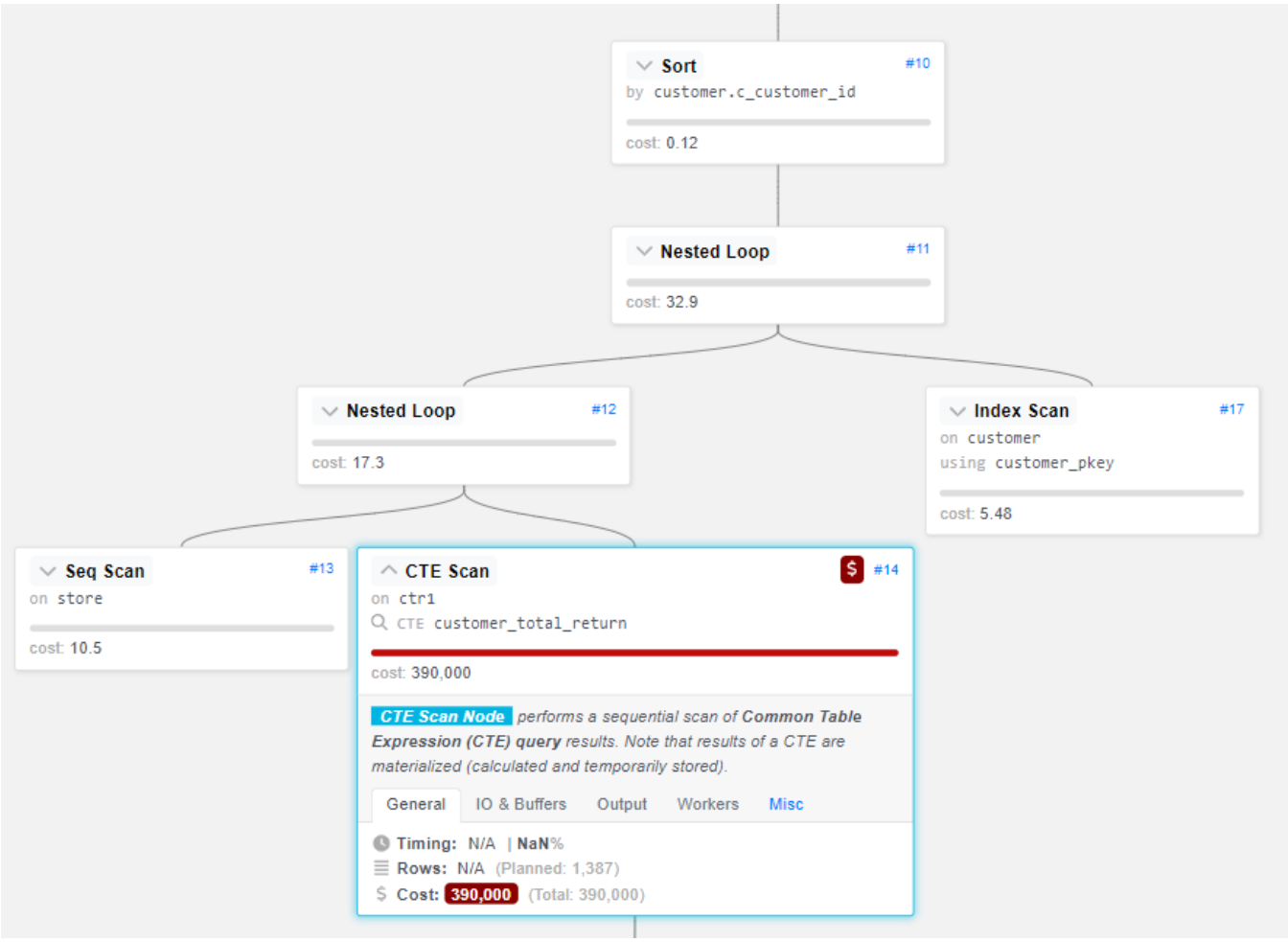

FIGURE 6.1: Query 1 before optimization

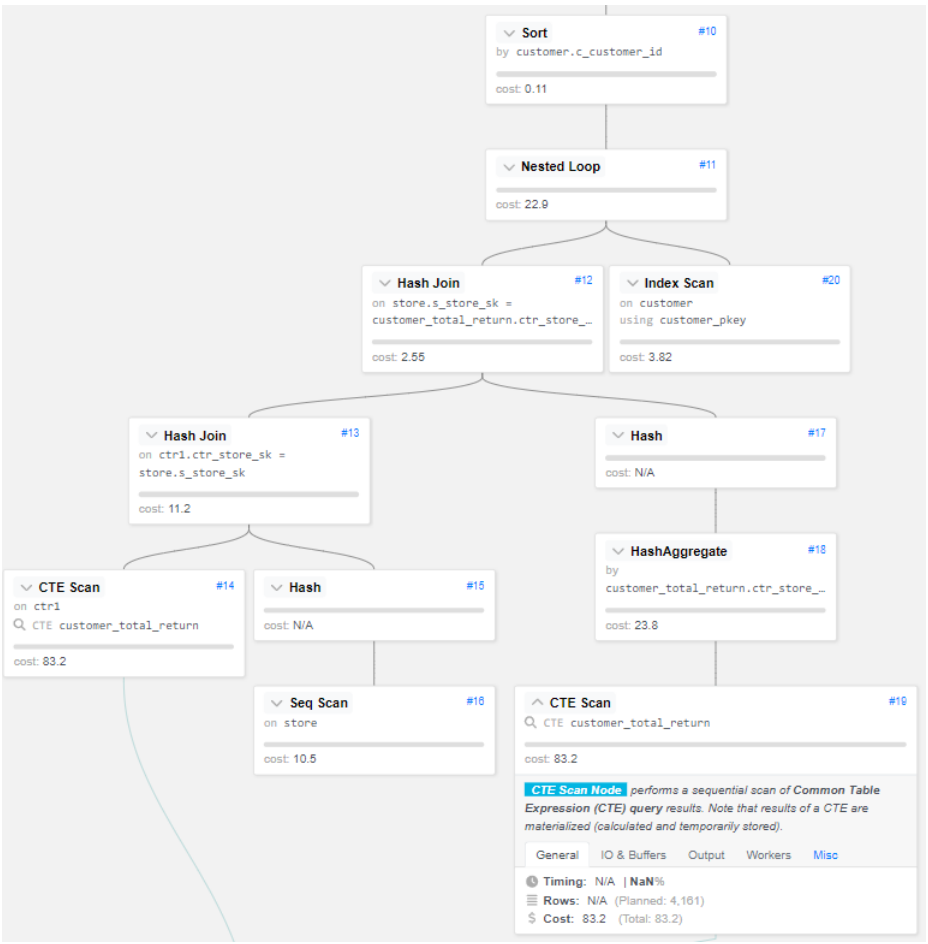

FIGURE 6.2: Query 1 after optimization

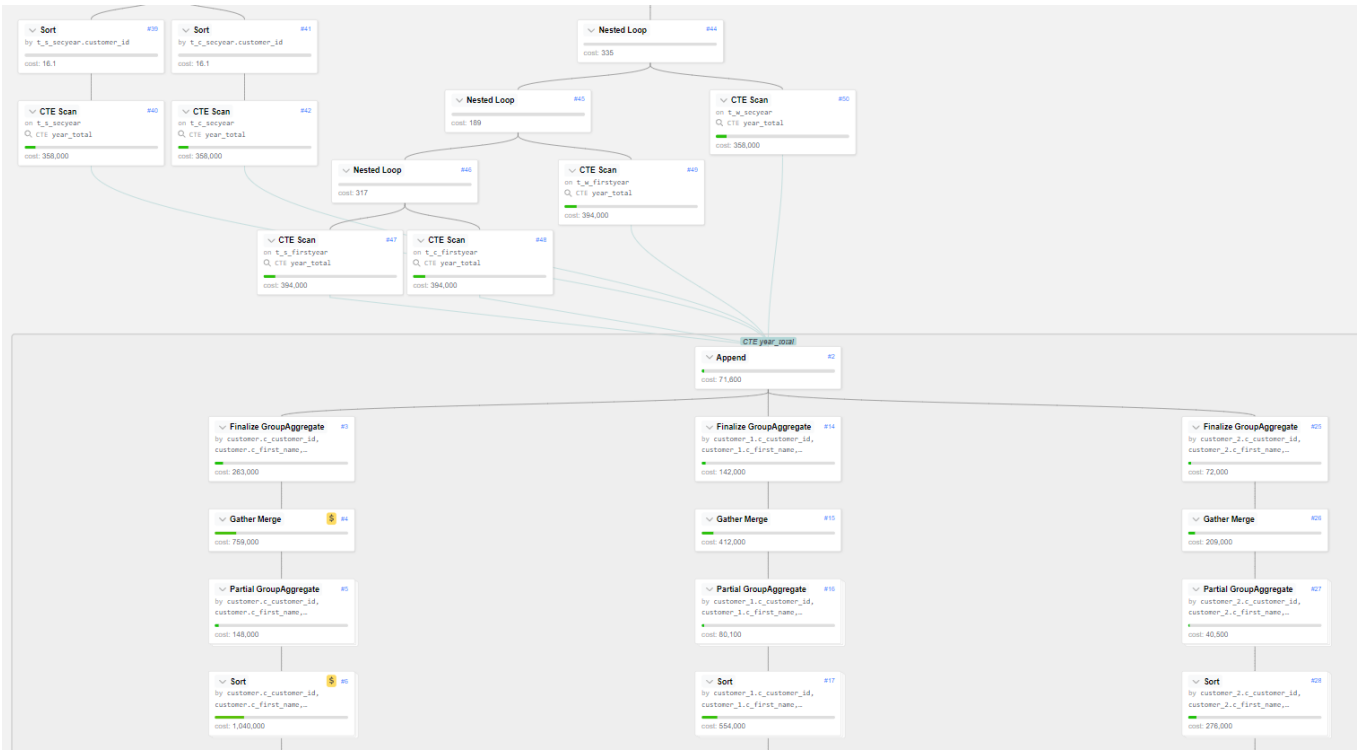

FIGURE 6.3: Query 4 before optimization

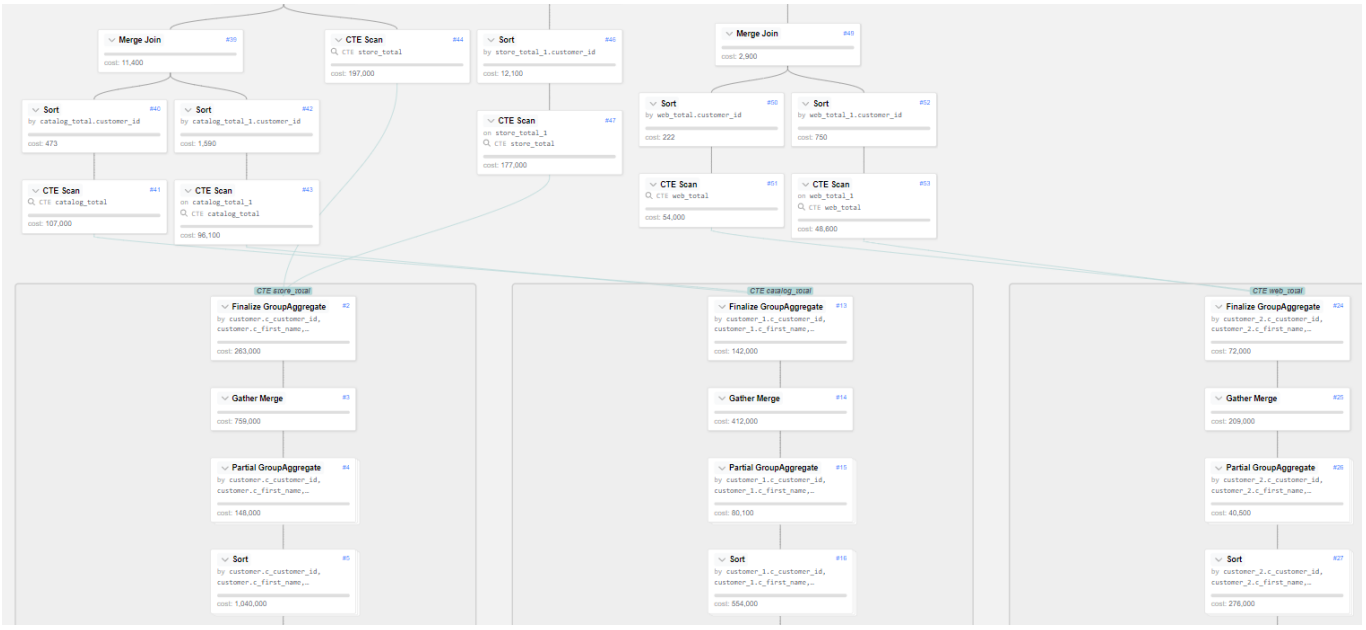

FIGURE 6.4: Query 4 after optimization

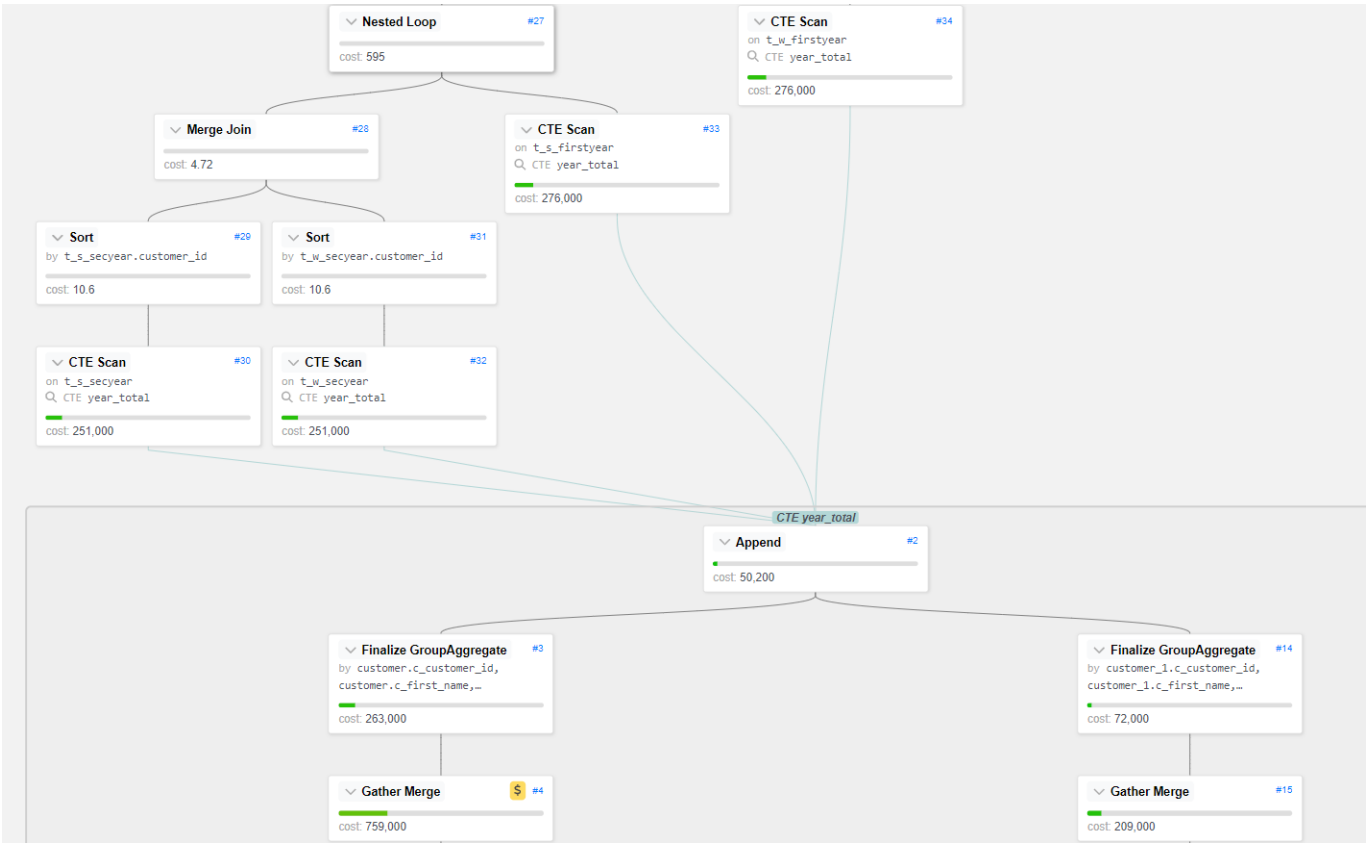

FIGURE 6.5: Query 11 before optimization

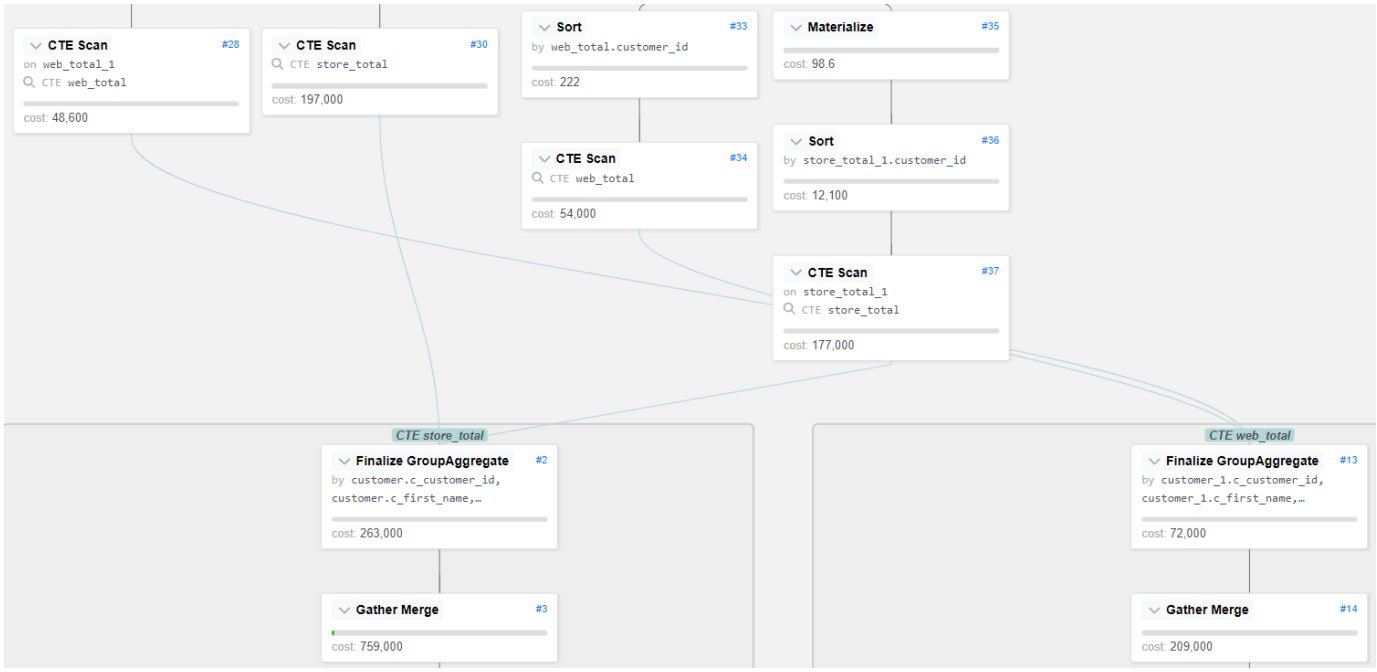

FIGURE 6.6: Query 11 after optimization

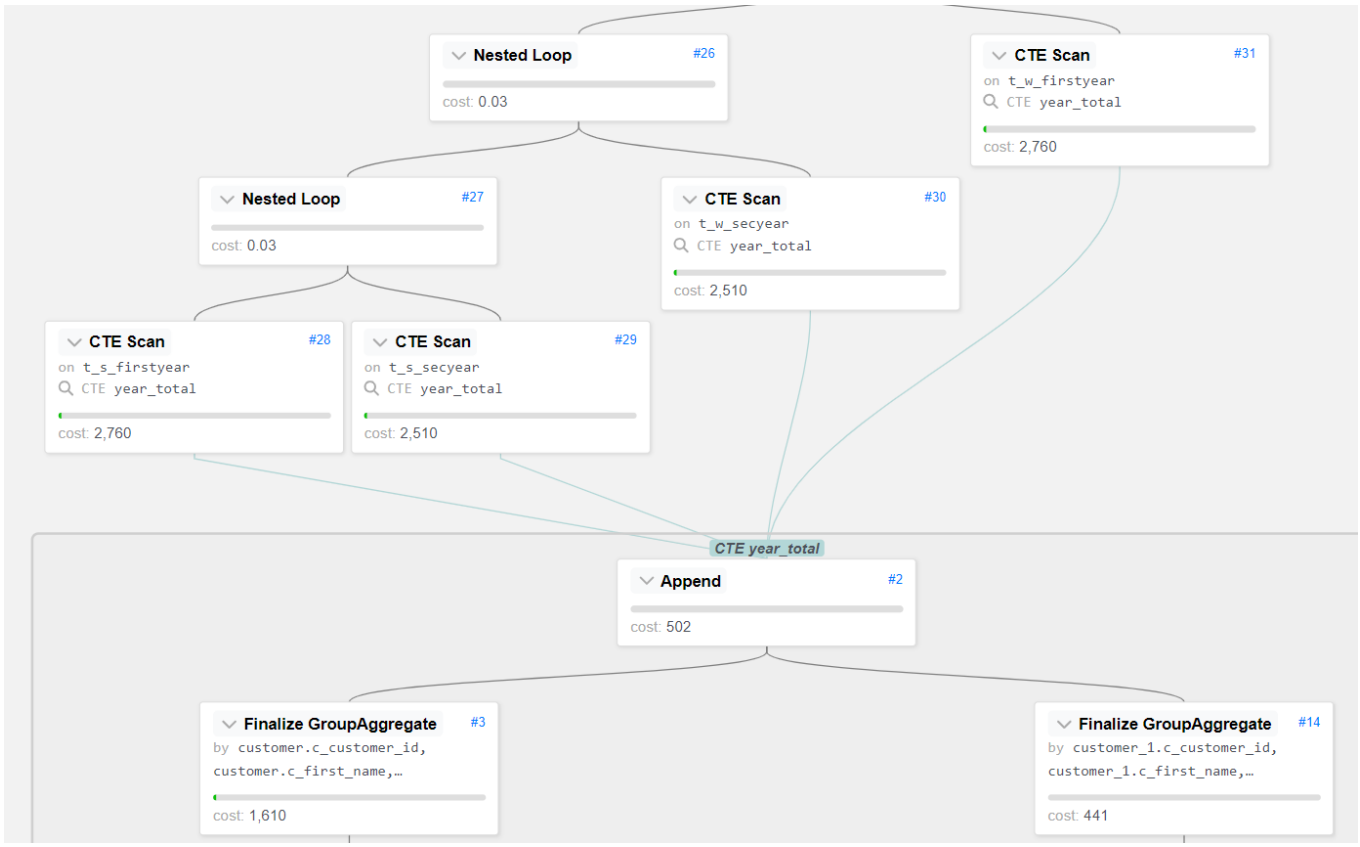

FIGURE 6.7: Query 74 before optimization

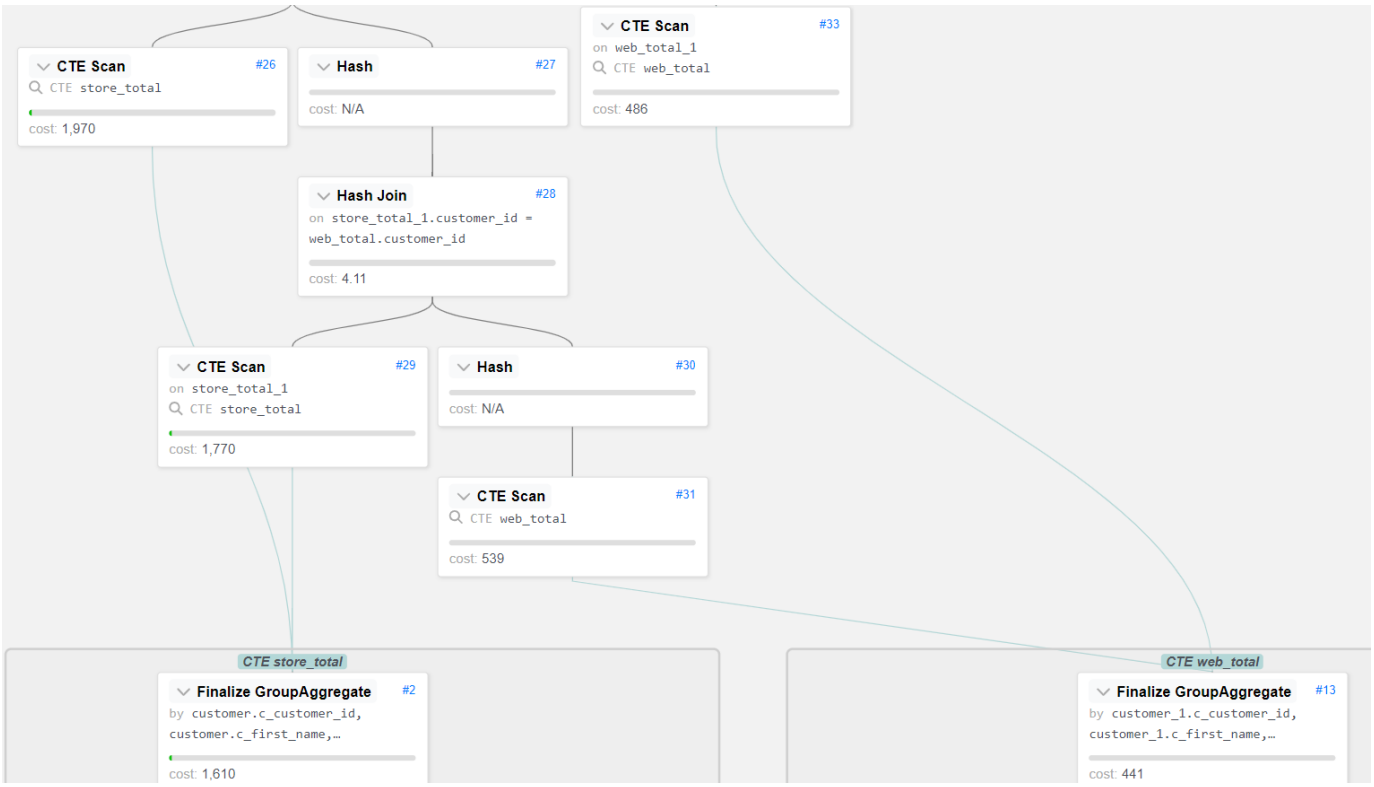

FIGURE 6.8: Query 74 after optimization

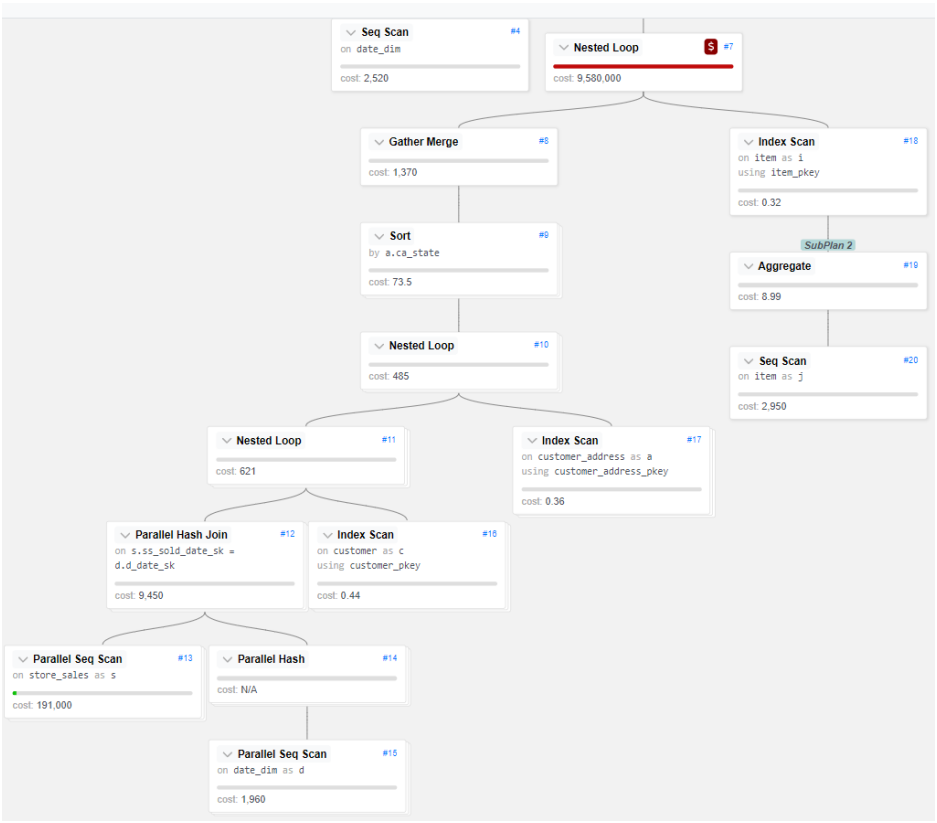

FIGURE 6.9: Query 6 before optimization

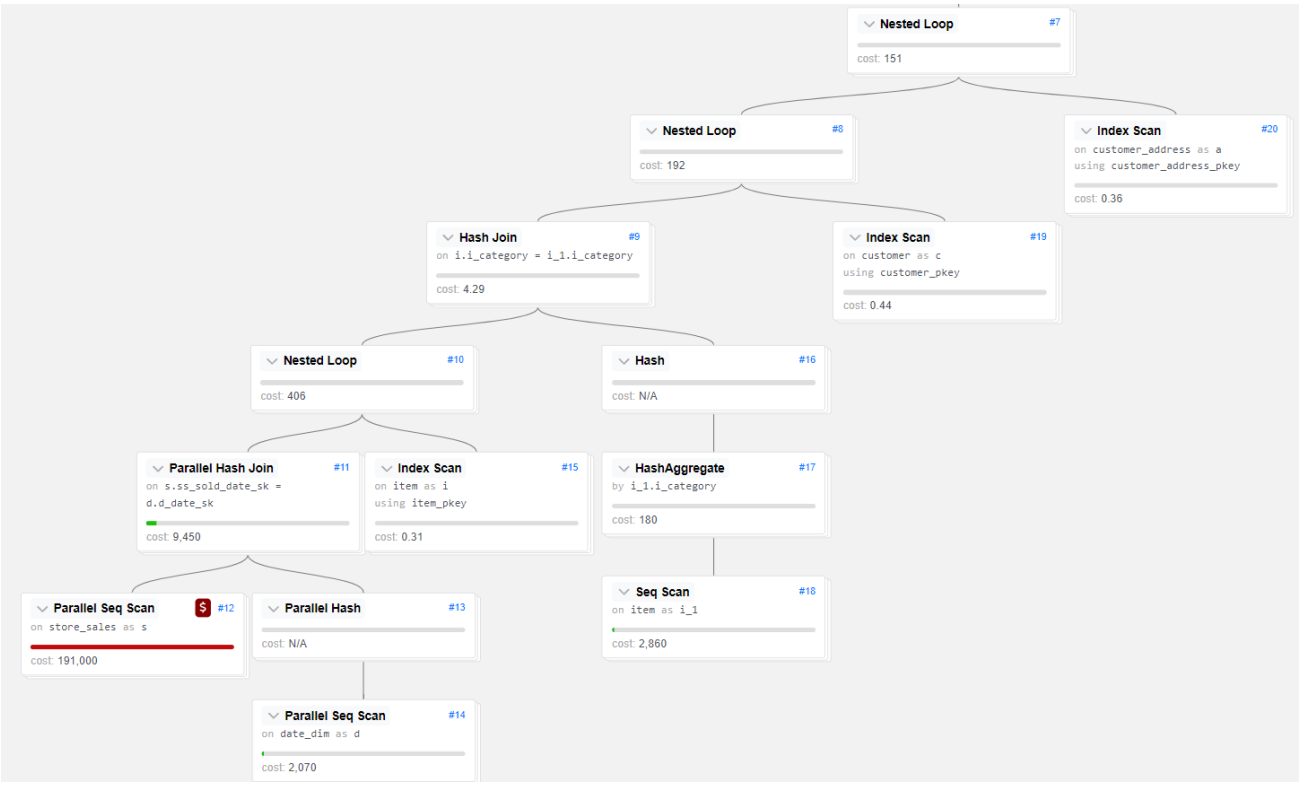

FIGURE 6.10: Query 6 after optimization

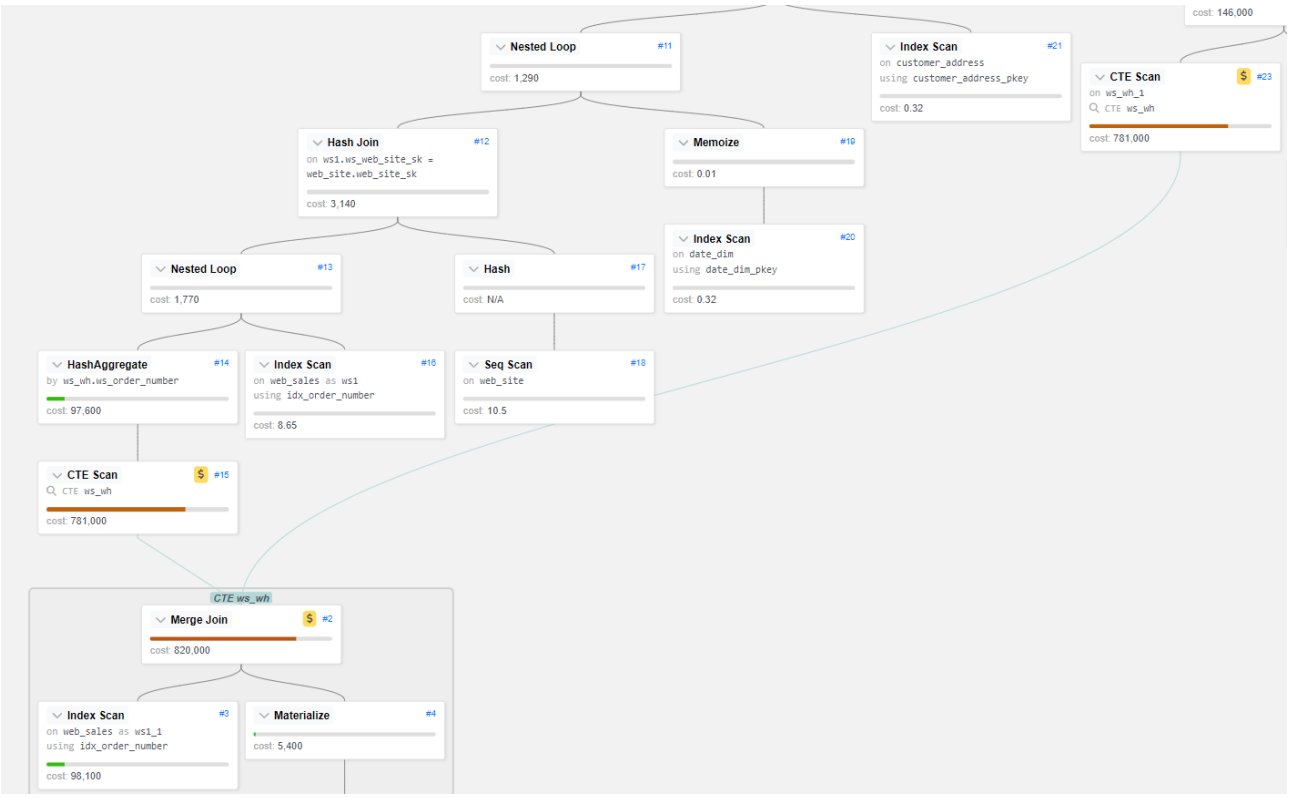

FIGURE 6.11: Query 95 before optimization

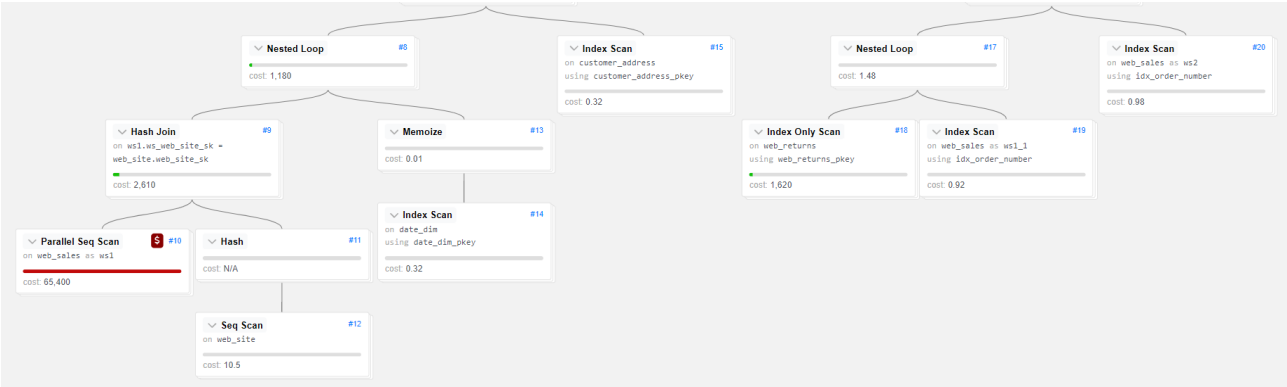

FIGURE 6.12: Query 95 after optimization

7 Appendix B - Optimized version of the queries

OPTIMIZED QUERY 1

```
CREATE INDEX if not exists idx_sr_fee ON store_returns (sr_fee);

with customer_total_return as
(select sr_customer_sk as ctr_customer_sk
,sr_store_sk as ctr_store_sk
,sum(sr_fee) as ctr_total_return
from store_returns
inner join date_dim on sr_returned_date_sk = d_date_sk
where d_year =2000
group by sr_customer_sk
,sr_store_sk)

, avg_cust_returns as
(select ctr_store_sk as store,
avg(ctr_total_return) as avg_returns
from customer_total_return
group by ctr_store_sk)

select  c_customer_id

from customer_total_return ctr1
inner join store on s_store_sk = ctr1.ctr_store_sk
inner join customer on c_customer_sk = ctr1.ctr_customer_sk
inner join avg_cust_returns on store = ctr1.ctr_store_sk

where ctr1.ctr_total_return > 1.2*avg_returns
and s_state = 'TN'

```

```
order by c_customer_id

limit 100;
```

OPTIMIZED QUERY 4

```
-- index on store sales
CREATE INDEX if not exists idx_ss_ext_list_price on store_sales(
    ss_ext_list_price);
CREATE INDEX if not exists idx_ss_ext_wholesale_cost on store_sales(
    ss_ext_wholesale_cost);
CREATE INDEX if not exists idx_ss_ext_discount_amt on store_sales(
    ss_ext_discount_amt);
CREATE INDEX if not exists idx_ss_ext_sales_price on store_sales(
    ss_ext_sales_price);
-- index on catalog sales
CREATE INDEX if not exists idx_cs_ext_list_price on catalog_sales(
    cs_ext_list_price);
CREATE INDEX if not exists idx_cs_ext_wholesale_cost on catalog_sales(
    cs_ext_wholesale_cost);
CREATE INDEX if not exists idx_cs_ext_discount_amt on catalog_sales(
    cs_ext_discount_amt);
CREATE INDEX if not exists idx_cs_ext_sales_price on catalog_sales(
    cs_ext_sales_price);
-- index on web sales
CREATE INDEX if not exists idx_ws_ext_list_price on web_sales(
    ws_ext_list_price);
CREATE INDEX if not exists idx_ws_ext_wholesale_cost on web_sales(
    ws_ext_wholesale_cost);
CREATE INDEX if not exists idx_ws_ext_discount_amt on web_sales(
    ws_ext_discount_amt);
CREATE INDEX if not exists idx_ws_ext_sales_price on web_sales(
    ws_ext_sales_price);

-- explain analyze
with store_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
```

```
25        ,c_email_address customer_email_address
26        ,d_year dyear
27        ,sum(((ss_ext_list_price-ss_ext_wholesale_cost-ss_ext_discount_amt)+
    ss_ext_sales_price)/2) year_total
28        ,'s' sale_type
29  from customer
30      inner join store_sales on c_customer_sk = ss_customer_sk
31      inner join date_dim on ss_sold_date_sk = d_date_sk
32  group by c_customer_id
33          ,c_first_name
34          ,c_last_name
35          ,c_preferred_cust_flag
36          ,c_birth_country
37          ,c_login
38          ,c_email_address
39          ,d_year
40  ),
41
42 catalog_total as(
43  select c_customer_id customer_id
44        ,c_first_name customer_first_name
45        ,c_last_name customer_last_name
46        ,c_preferred_cust_flag customer_preferred_cust_flag
47        ,c_birth_country customer_birth_country
48        ,c_login customer_login
49        ,c_email_address customer_email_address
50        ,d_year dyear
51        ,sum((((cs_ext_list_price-cs_ext_wholesale_cost-cs_ext_discount_amt)+
    cs_ext_sales_price)/2) ) year_total
52        ,'c' sale_type
53  from customer
54      inner join catalog_sales on c_customer_sk = cs_bill_customer_sk
55      inner join date_dim on cs_sold_date_sk = d_date_sk
56  group by c_customer_id
57          ,c_first_name
58          ,c_last_name
59          ,c_preferred_cust_flag
60          ,c_birth_country
61          ,c_login
62          ,c_email_address
```

```
         ,d_year
),

web_total as(
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
       ,d_year dyear
       ,sum((((ws_ext_list_price-ws_ext_wholesale_cost-ws_ext_discount_amt)+
   ws_ext_sales_price)/2) ) year_total
       ,'w' sale_type
 from customer
     inner join web_sales on c_customer_sk = ws_bill_customer_sk
     inner join date_dim on ws_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
),

t_s_firstyear as(
    select customer_id,
           customer_first_name,
           customer_last_name,
           customer_email_address,
           year_total
    from store_total
    where dyear= 2001
    and year_total>0
),

t_s_secyear as(
```

```
    select customer_id,
           customer_first_name,
           customer_last_name,
           customer_email_address,
           year_total
    from store_total
    where dyear= 2001+1
),

t_c_firstyear as(
    select customer_id,
           customer_first_name,
           customer_last_name,
           customer_email_address,
           year_total
    from catalog_total
    where dyear= 2001
    and year_total>0
),

t_c_secyear as(
    select customer_id,
           customer_first_name,
           customer_last_name,
           customer_email_address,
           year_total
    from catalog_total
    where dyear= 2001+1
),

t_w_firstyear as(
    select customer_id,
           customer_first_name,
           customer_last_name,
           customer_email_address,
           year_total
    from web_total
    where dyear= 2001
    and year_total>0
),
```

```
142
143 t_w_secyear as(
144     select customer_id,
145            customer_first_name,
146            customer_last_name,
147            customer_email_address,
148            year_total
149     from web_total
150     where dyear= 2001+1
151 )
152
153 select
154 t_s_secyear.customer_id
155 ,t_s_secyear.customer_first_name
156 ,t_s_secyear.customer_last_name
157 ,t_s_secyear.customer_email_address
158
159 from t_s_firstyear
160 inner join t_s_secyear on t_s_secyear.customer_id = t_s_firstyear.
    customer_id
161 inner join t_c_firstyear on t_s_firstyear.customer_id = t_c_firstyear.
    customer_id
162 inner join t_c_secyear on t_s_firstyear.customer_id = t_c_secyear.
    customer_id
163 inner join t_w_firstyear on t_s_firstyear.customer_id = t_w_firstyear.
    customer_id
164 inner join t_w_secyear on t_s_firstyear.customer_id = t_w_secyear.
    customer_id
165
166 where
167    case when t_c_firstyear.year_total > 0 then t_c_secyear.year_total /
    t_c_firstyear.year_total else null end
168            > case when t_s_firstyear.year_total > 0 then t_s_secyear.
    year_total / t_s_firstyear.year_total else null end
169    and case when t_c_firstyear.year_total > 0 then t_c_secyear.year_total /
    t_c_firstyear.year_total else null end
170            > case when t_w_firstyear.year_total > 0 then t_w_secyear.
    year_total / t_w_firstyear.year_total else null end
171
172 order by t_s_secyear.customer_id
```

```
        ,t_s_secyear.customer_first_name
        ,t_s_secyear.customer_last_name
        ,t_s_secyear.customer_email_address

limit 100;
```

OPTIMIZED QUERY 6

```
CREATE INDEX if not exists idx_item_current_price on item(i_current_price);

with avg_price as(
select i.i_category, avg(i.i_current_price) as avg
from item i
group by i.i_category
)

select  a.ca_state state, count(*) cnt
from customer_address a
inner join customer c on a.ca_address_sk = c.c_current_addr_sk
inner join store_sales s on c.c_customer_sk = s.ss_customer_sk
inner join date_dim d on s.ss_sold_date_sk = d.d_date_sk
inner join item i on s.ss_item_sk = i.i_item_sk
inner join avg_price on i.i_category= avg_price.i_category

where d.d_year= 2000
and d.d_moy= 2
and i.i_current_price > 1.2 * avg_price.avg

group by a.ca_state
having count(*) >= 10

order by cnt, a.ca_state

limit 100;
```

OPTIMIZED QUERY 11

```
-- index on store sales
CREATE INDEX if not exists idx_ss_ext_list_price on store_sales(
   ss_ext_list_price);
CREATE INDEX if not exists idx_ss_ext_wholesale_cost on store_sales(
   ss_ext_wholesale_cost);
CREATE INDEX if not exists idx_ss_ext_discount_amt on store_sales(
```

```
   ss_ext_discount_amt);
CREATE INDEX if not exists idx_ss_ext_sales_price on store_sales(
   ss_ext_sales_price);
-- index on web sales
CREATE INDEX if not exists idx_ws_ext_list_price on web_sales(
   ws_ext_list_price);
CREATE INDEX if not exists idx_ws_ext_wholesale_cost on web_sales(
   ws_ext_wholesale_cost);
CREATE INDEX if not exists idx_ws_ext_discount_amt on web_sales(
   ws_ext_discount_amt);
CREATE INDEX if not exists idx_ws_ext_sales_price on web_sales(
   ws_ext_sales_price);

-- explain analyze
with store_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
       ,d_year dyear
       ,sum(ss_ext_list_price-ss_ext_discount_amt) year_total
       ,'s' sale_type
 from customer
 inner join store_sales on c_customer_sk = ss_customer_sk
 inner join date_dim on ss_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
),

web_total as(
 select c_customer_id customer_id
```

```
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
       ,d_year dyear
       ,sum(ws_ext_list_price-ws_ext_discount_amt) year_total
       ,'w' sale_type
 from customer
 inner join web_sales on c_customer_sk = ws_bill_customer_sk
 inner join date_dim on ws_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
),

t_s_firstyear as(
    select customer_id,
           customer_first_name,
           customer_last_name,
           customer_email_address,
           year_total
    from store_total
    where dyear= 2001
    and year_total>0
),

t_s_secyear as(
    select customer_id,
           customer_first_name,
           customer_last_name,
           customer_email_address,
           year_total
    from store_total
```

```
    where dyear= 2001+1
),

t_w_firstyear as(
    select customer_id,
           customer_first_name,
           customer_last_name,
           customer_email_address,
           year_total
    from web_total
    where dyear= 2001
    and year_total>0
),

t_w_secyear as(
    select customer_id,
           customer_first_name,
           customer_last_name,
           customer_email_address,
           year_total
    from web_total
    where dyear= 2001+1
)

select
t_s_secyear.customer_id
,t_s_secyear.customer_first_name
,t_s_secyear.customer_last_name
,t_s_secyear.customer_email_address

from t_s_firstyear
inner join t_s_secyear on t_s_secyear.customer_id = t_s_firstyear.
   customer_id
inner join t_w_firstyear on t_s_firstyear.customer_id = t_w_firstyear.
   customer_id
inner join t_w_secyear on t_s_firstyear.customer_id = t_w_secyear.
   customer_id

where
     case when t_w_firstyear.year_total > 0 then t_w_secyear.year_total /
```

```
    t_w_firstyear.year_total else 0.0 end
116         > case when t_s_firstyear.year_total > 0 then t_s_secyear.
    year_total / t_s_firstyear.year_total else 0.0 end
117
118 order by t_s_secyear.customer_id
119         ,t_s_secyear.customer_first_name
120         ,t_s_secyear.customer_last_name
121         ,t_s_secyear.customer_email_address
122
123 limit 100;
```

OPTIMIZED QUERY 72

```
1  CREATE INDEX if not exists idx_promo on promotion(p_promo_sk);
2
3  select  i_item_desc
4       ,w_warehouse_name
5       ,d1.d_week_seq
6       ,sum(case when p_promo_sk is null then 1 else 0 end) no_promo
7       ,sum(case when p_promo_sk is not null then 1 else 0 end) promo
8       ,count(p_promo_sk) total_cnt
9  from catalog_sales
10 join inventory on (cs_item_sk = inv_item_sk)
11 join warehouse on (w_warehouse_sk=inv_warehouse_sk)
12 join item on (i_item_sk = cs_item_sk)
13 join customer_demographics on (cs_bill_cdemo_sk = cd_demo_sk)
14 join household_demographics on (cs_bill_hdemo_sk = hd_demo_sk)
15 join date_dim d1 on (cs_sold_date_sk = d1.d_date_sk)
16 join date_dim d2 on (inv_date_sk = d2.d_date_sk)
17 join date_dim d3 on (cs_ship_date_sk = d3.d_date_sk)
18 left outer join promotion on (cs_promo_sk=p_promo_sk)
19 -- left outer join catalog_returns on (cr_item_sk = cs_item_sk and
    cr_order_number = cs_order_number)
20 where d1.d_week_seq = d2.d_week_seq
21   and inv_quantity_on_hand < cs_quantity
22   and d3.d_date > d1.d_date + 5
23   and hd_buy_potential = '1001-5000'
24   and d1.d_year = 2001
25   and cd_marital_status = 'M'
26 group by i_item_desc,w_warehouse_name,d1.d_week_seq
27 order by total_cnt desc, i_item_desc, w_warehouse_name, d_week_seq
28 limit 100;
```

OPTIMIZED QUERY 74

```
-- index on store sales
CREATE INDEX if not exists idx_ss_net_paid on store_sales(ss_net_paid);
-- index on web sales
CREATE INDEX if not exists idx_ws_net_paid on web_sales(ws_net_paid);

-- explain analyze
with store_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,d_year as year
       ,max(ss_net_paid) year_total
       ,'s' sale_type
 from customer
 inner join store_sales on c_customer_sk = ss_customer_sk
 inner join date_dim on ss_sold_date_sk = d_date_sk
 where d_year in (2001,2001+1)
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,d_year
 ),

 web_total as(
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,d_year as year
       ,max(ws_net_paid) year_total
       ,'w' sale_type
 from customer
 inner join web_sales on c_customer_sk = ws_bill_customer_sk
 inner join date_dim on ws_sold_date_sk = d_date_sk
 where d_year in (2001,2001+1)
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,d_year
),
```

```
40
41 t_s_firstyear as(
42     select customer_id,
43            customer_first_name,
44            customer_last_name,
45            year_total
46     from store_total
47     where year= 2001
48     and year_total>0
49 ),
50
51 t_s_secyear as(
52     select customer_id,
53            customer_first_name,
54            customer_last_name,
55            year_total
56     from store_total
57     where year= 2001+1
58 ),
59
60 t_w_firstyear as(
61     select customer_id,
62            customer_first_name,
63            customer_last_name,
64            year_total
65     from web_total
66     where year= 2001
67     and year_total>0
68 ),
69
70 t_w_secyear as(
71     select customer_id,
72            customer_first_name,
73            customer_last_name,
74            year_total
75     from web_total
76     where year= 2001+1
77 )
78  select
79         t_s_secyear.customer_id,
```

```
        t_s_secyear.customer_first_name,
        t_s_secyear.customer_last_name

 from t_s_firstyear
 inner join t_s_secyear on t_s_secyear.customer_id = t_s_firstyear.
   customer_id
 inner join t_w_firstyear on t_s_firstyear.customer_id = t_w_firstyear.
   customer_id
 inner join t_w_secyear on t_s_firstyear.customer_id = t_w_secyear.
   customer_id

 where
     case when t_w_firstyear.year_total > 0 then t_w_secyear.year_total /
   t_w_firstyear.year_total else null end
          > case when t_s_firstyear.year_total > 0 then t_s_secyear.
   year_total / t_s_firstyear.year_total else null end

 order by 2,1,3

limit 100;
```

OPTIMIZED QUERY 95

```
CREATE INDEX if not exists idx_order_number on web_sales(ws_order_number);
CREATE INDEX if not exists idx_ext_ship_cost on web_sales(ws_ext_ship_cost);
CREATE INDEX if not exists idx_net_profit on web_sales(ws_net_profit);

with ws_wh as
(select ws1.ws_order_number,ws1.ws_warehouse_sk wh1,ws2.ws_warehouse_sk wh2
 from web_sales ws1,web_sales ws2
 where ws1.ws_order_number = ws2.ws_order_number
   and ws1.ws_warehouse_sk <> ws2.ws_warehouse_sk
),

wr as(
select wr_order_number
from web_returns
inner join ws_wh on wr_order_number = ws_wh.ws_order_number
)

select
   count(distinct ws_order_number) as "order count"
```

```
  ,sum(ws_ext_ship_cost) as "total shipping cost"
  ,sum(ws_net_profit) as "total net profit"

from
   web_sales ws1
inner join date_dim on ws1.ws_ship_date_sk = d_date_sk
inner join customer_address on ws1.ws_ship_addr_sk = ca_address_sk
inner join web_site on ws1.ws_web_site_sk = web_site_sk

where
    d_date between '1999-5-01' and
           (cast('1999-5-01' as date) + interval '60 days')
and ca_state = 'TX'
and web_company_name = 'pri'
-- and ws1.ws_order_number in (select ws_order_number from ws_wh)
and ws1.ws_order_number in (select wr.wr_order_number from wr)
order by count(distinct ws_order_number)
limit 100;
```

Bibliographie

[DAL23] DALIBO (2023). *Query Plan Visualizer*. https://explain.dalibo.com/.

[DB-23] DB-ENGINES (oct. 2023). URL : https://db-engines.com/en/ranking.

[NP06] NAMBIAR, Raghu et Meikel POESS (jan. 2006). « The Making of TPC-DS ». In : p. 1049-1058.

[PS04] POESS, Meikel et John M. STEPHENS (2004). « Generating Thousand Benchmark Queries in Seconds ». In : *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*. VLDB '04. Toronto, Canada : VLDB Endowment, p. 1045-1053. ISBN : 0120884690.

[Pos23a] POSTGIS (oct. 2023). URL : https://postgis.net/.

[Pos23b] POSTGRESQL (oct. 2023a). URL : https://www.postgresql.org/about/.

[Pos23c] — (sept. 2023b). URL : https://www.postgresql.org/docs/current/hstore.html.

[Pos23d] — (oct. 2023c). URL : https://www.postgresql.org/docs/current/sql-insert.html#SQL-ON-CONFLICT.

[23] *TPC current document versions* (oct. 2023). URL : https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp.

[21] *TPC-DS Standard Specification* (juin 2021). URL : https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf.