SparkSQL TPC-DS Benchmarking

Data Warehouses (INFO-H419)

Cools Arnaud (499956)
Dubois Alexandre (501050)
Rocca Valerio (589084)
Salazar Maria (587449)

November 2, 2023

Contents

Abstract

This project focuses on evaluating the performance and scalability of Apache Spark's SQL engine using the TPC-DS (Transaction Processing Performance Council - Decision Support) benchmark. Leveraging SparkSQL, an efficient and distributed query engine, we conduct a comprehensive performance assessment to measure the system's capabilities in handling complex decision support workloads.

1 Introduction

1.1 Introduction to TPC-DS

The Transaction Processing Performance Council (TPC)[3], is a non-profit organization founded in 1988 that defines and develops industry-standard benchmarks for evaluating the performance of various database and transaction processing systems.

TPC defines the Transaction Processing Performance Council Decision Support (TPC-DS) as "the de-facto industry standard benchmark for measuring the performance of decision support solutions including, but not limited to, Big Data systems." More specifically, TPC-DS has been designed for DSSs that:

- examine large volumes of data;
- give answers to real-world business questions;
- execute queries of various operational requirements and complexities;
- are characterized by high CPU and IO load;
- are periodically synchronized with source OLTP databases through database maintenance functions;
- run on Big Data solutions, such as RDBMS and Hadoop/Spark-based systems.

1.2 Introduction to Spark SQL

Spark SQL [1] is an Apache Spark module for working with structured data. The main advantages of this technology are:

- a native integration with Spark, which allows to easily combine it with other Spark components such as Spark Streaming and MLib;
- a uniform way to access various data sources, including Hive, Avro, Parquet, and JSON.
- the possibility to exploit parallel computing by leveraging the Spark engine, which allows Spark SQL to handle large-scale data with speed;
- a sophisticated query optimiser, Catalyst, which can optimize SQL queries with different techniques, including predicate pushdown, constant folding, and join reordering.

It is essential to underline that while it allows the integration of SQL in the Spark environment, Spark SQL is not a Database Management System (DBMS).

1.3 Objective and scope

This project aims to implement the TPC-DS benchmark on Spark SQL and analyse the results. The chosen programming language is Python.

The scope of this project includes (but is not limited to) the setup of Spark SQL on Python, the generation of the data and the generation, the optimization, and the execution of the queries. This report is not intended for industry use, as it performs computations on a smaller scale than the full-scale TPC-DS benchmarks (see section 3.2 for more details). Moreover, our benchmark uses only QphDS@SF metrics (see section 2.4 for more information). Finally, for limitations regarding Spark SQL, we did not run data maintenance functions (see section 3.4 for more information).

1.4 Tools

In Table 1, we summarize the main tools and technologies utilized.

Tool	**Version**	**Description**
TPC-DS tools	3.2.0	.zip file by TPC containing the official data and query generation tools.
Apache Spark	3.5	Unified engine for large-scale data analytics. It contains Spark SQL as a built-in library.
Hadoop	2.8	Collection of open-source software utilities. Necessary for some utilities.
Python	3.11	High-level, general-purpose programming language used to run the scripts.
Visual Studio Code	1.83.1	Visual Studio Code was used to generate the data and queries through dsdgen and dsqgen, into tool
Jupyter Notebook	7.0.6	Environment used to produce Python notebooks.
GitHub	-	GitHub Desktop was used to share code files as well as images conveniently with the team members.
Desktop PC	-	Computer used to perform the analysis. It is equipped with: **processor**: Intel(R) Core(TM) i5-2400 GPU 3.10GHz; **RAM**: 8045MB; **operating system**: Debian GNU/Linux 12

Table 1: Tool Versions and Descriptions

2 TPC-DS Benchmark

As said before, the TPC-DS benchmark [4] is a recognized and standardized method for evaluating the performance of DBMSs and data warehouses, specifically emphasizing Online Analytical Processing (OLAP) tasks. This section aims to provide a brief and practical overview of TPC-DS.

2.1 Database schema

By employing a snowflake schema, the TPC-DS schema models the sales and sales returns process for an organization that employs three primary sales channels: stores, catalogues, and the Internet. The schema includes seven fact tables:

- a pair of fact tables focused on the product sales and returns for each of the three channels;
- a single fact table that models inventory for the catalogue and internet sales channels.

In addition, the schema includes 17 dimension tables that are associated with all sales channels. Those tables can be classified into the following types.

- Static. The contents remain fixed and unchanging after the initial load during the database creation. An example is the date dimension.
- Historical. They record changes to dimension data over time by creating multiple rows for a single business key value, such that each row includes time-based validity information. An example is the item dimension.
- Non-Historical, They do not maintain a history of changes, in the sense that when dimension rows are updated, the previous values are overwritten, resulting in the loss of historical information. An example is the customer dimension.

2.2 Data generation and scale factor

TPC-DS includes a tool to generate data to populate the tables. The size is determined by discrete scaling points called "scale factors"). A scale factor equal to 1 corresponds to 1 GB of generated data.

The set of scale factors defined for the benchmark ranges from 1 TB to 100 TB. However, considering the project's limited scope, we are performing the analyses using the smaller scale factor (see section 4.2 for more details).

2.3 Workload

To be realistic compared to a real-life DSS, the TPC-DS workload comprises both a query workload and a data maintenance workload.

The first one is accomplished through a set of 99 SQL:1999 queries, which can be classified into:

- reporting queries, which include queries that are executed periodically to answer well-known, pre-defined questions about the financial and operational health of a business;
- ad hoc queries, to capture those queries constructed to answer immediate and specific business questions;
- iterative OLAP queries, which allow for the exploration and analysis of business data to discover relationships and trends;

- data mining queries, which typically consists of joins and large aggregations that return large data result sets for possible extraction.

On the other hand, the data maintenance workload models the data transformation and load parts of ETL (Extraction-Transformation-Load). Data transformation is the process through which the extracted data is cleansed and massaged into a standard format suitable for assimilation. In contrast, data load is the actual insertion, modification, and deletion of data.

2.4 Execution rules and performance measure

TPC-DS benchmarking is performed by executing the following sequence of tests:

1. Database Load Test;
2. Power Test;
3. 1st Throughput Test;
4. 1st Data Maintenance Test;
5. 2nd Throughput Test;
6. 2nd Data Maintenance Test.

Now we briefly explain what each test measures.

- The **Database Load Test** measures the time required to set up the database for the other tests. This includes the creation of the tables and their population with generated data.
- The **Power Test** measures the time required to process a single query stream, i.e. to execute sequentially a permutation of the 99 queries.
- The **Throughput Tests** study the ability of the system to deal with requests from multiple users by measuring the time required to process multiple query streams. The number of query streams depends on the chosen scale factor, but it has to be even and greater than 3.
- The **Data Maintenance Tests** measure the ability to perform desired data changes to the TPC-DS data set by executing the data maintenance functions.

TPC-DS defines three primary metrics. However, our benchmark deals only with QphDS@SF, as the others are not applicable. QphDS@SF - an acronym of Queries per hour for Decision Support at Scale Factor - can be calculated as follows:

$$QphDS@SF = \lfloor \frac{SF \cdot Q}{\sqrt[4]{T_{PT} \cdot T_{TT} \cdot T_{DM} \cdot T_{LD}}} \rfloor$$

Where:

- SF is the scale factor;
- $Q = S_q \cdot 99$ is the total number of weighted queries, as S_q is the number of streams executed in a Throughput Test;
- $T_{PT} = T_{Power} \cdot S_q$, where T_{PT} is the total elapsed time to complete the Power Test;

- $T_{TT} = T_{TT1} + T_{TT2}$, where T_{TT1} and T_{TT2} are the total elapsed times of respectively Throughput Test 1 and Throughput Test 2;
- $T_{DM} = T_{DM1} + T_{DM2}$, where T_{DM1} and T_{DM2} are the total elapsed times of respectively Data Maintenance Test 1 and Data Maintenance Test 2;
- T_{LD} is the load factor computed as $T_{LD} = 0.01 \cdot S_q \cdot T_{Load}$, where T_{Load} is the time to finish the load.

Moreover, we also employ time as a metric to draw conclusions.

3 Implementation

3.1 Setting

Considering the small scale of our project, we decided to create four query streams, for a total of eight refresh datasets.

3.2 Data generation

The scale factors chosen for this project are:

- 1 GB
- 3 GB
- 5 GB
- 8 GB
- 10 GB

Data at each scale factor was generated by the following command in **Visual Studio Code**, over the toolkit folder:

```
./dsdgen -DIR ../data$SC  -SCALE $SC
```

Listing 1: dsdgen

where *SC* was changed for each scale factor and saved in the main folder.

3.2.1 Databases

The following table summarizes the number of records of the database tables at different scale factors.

Table Name	SF=1	SF=3	SF=5	SF=8	SF=10
call center	6	10	14	20	24
catalog page	11718	11718	11718	11718	12000
catalog returns	144067	432000	720174	1151949	1439749
catalog sales	1441548	4319367	7199490	11522895	14401261
customer	100000	188000	277000	411000	500000
customer address	50000	94000	138000	205000	250000
customer demographics	1920800	1920800	1920800	1920800	1920800
date dim	73049	73049	73049	73049	73049
household demographics	7200	7200	7200	7200	7200
income band	20	20	20	20	20
inventory	11745000	28188000	49329000	85610088	133110000
item	18000	36000	54000	82000	102000
promotion	300	344	388	455	500
reason	35	37	39	42	45
ship mode	20	20	20	20	20
store	12	32	52	82	102
store returns	287514	862834	1437911	2300664	2875432
store sales	2880404	8639377	14400052	23038469	28800991
time dim	86400	86400	86400	86400	86400
warehouse	5	6	7	8	10
web page	60	90	122	168	200
web returns	71763	215477	359991	575522	719217
web sales	719384	2160165	3599503	5757840	7197566
web site	30	32	34	38	42

Table 2: Number of rows for tables in each scale

3.3 Query generation, modification and execution

The main idea of the TPC-DS benchmarking is to obtain results that allow making decisions in the industry about the use of tools. Therefore, it uses queries of queries that allow answering business questions. For example, the following query (**query 7**) gets the average of some variables from the store sales table under some demographical variables like gender, marital status, education status, and some other business conditions like the promotion channel for a particular year.

```
select  i_item_id,
        avg(ss_quantity) agg1,
        avg(ss_list_price) agg2,
        avg(ss_coupon_amt) agg3,
        avg(ss_sales_price) agg4
 from store_sales, customer_demographics, date_dim, item, promotion
 where ss_sold_date_sk = d_date_sk and
       ss_item_sk = i_item_sk and
       ss_cdemo_sk = cd_demo_sk and
       ss_promo_sk = p_promo_sk and
       cd_gender = 'M' and
       cd_marital_status = 'M' and
       cd_education_status = '4 yr Degree' and
       (p_channel_email = 'N' or p_channel_event = 'N') and
       d_year = 2001
```

```
group by i_item_id
order by i_item_id
limit 100;
```

Listing 2: Query7

3.3.1 Query generation

TPC-DC toolkit allows the generation of 99 queries from different templates. For this project, we wrote the following code using the Netezza template due to its similarity with Spark SQL.

In the template, you can add the next lines to create the start and end points for each query. Some of them have more than one **";"** inside and can make errors if you want to separate them "automatically".

```
define _END = "";
define _BEGIN = "-- start query " + [_QUERY] + " in stream " + [_STREAM] + "
    using template " + [_TEMPLATE];
define _END = "-- end query " + [_QUERY] + " in stream " + [_STREAM] + " using
    template " + [_TEMPLATE];
```

Listing 3: Additional information for query templates

Then, you finally can run the next command in **Visual Studio Code** over the toolkit folder:

```
./dsqgen -DIRECTORY ../query_templates -INPUT ../query_templates/templates.lst
    -OUTPUT_DIR ../Result -DIALECT netezza -STREAMS 9
```

Listing 4: Query generation

To separate the queries, we use a program created in Python called **Separate queries**.

3.3.2 Query modifications

During the process of executing the queries, some errors arose due to Spark SQL syntax. Thus, the following small modifications were performed:

- Dates: It does not need "days";

```
   and (cast('1998-08-04' as date) +  14 days)
   and (cast('1998-08-04' as date) +  14 )
```

- Alias: Problem with names with spaces;

```
  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"
  count(distinct ws_order_number) order_count
 ,sum(ws_ext_ship_cost) total_shipping_cost
 ,sum(ws_net_profit) total_net_profit
```

```
-                (ws_ship_date_sk - ws_sold_date_sk <= 120) then 1 else 0 end)  as "91-120 days"
-   ,sum(case when (ws_ship_date_sk - ws_sold_date_sk  > 120) then 1 else 0 end)  as ">120 days"
+                (ws_ship_date_sk - ws_sold_date_sk <= 120) then 1 else 0 end) days91_to_120
+   ,sum(case when (ws_ship_date_sk - ws_sold_date_sk  > 120) then 1 else 0 end) days_bigger_than_120
```

- Columns: in query 30, there was a selection on a column of a table that does not exist. We simply removed it.

3.3.3 Query execution

3.4 Data maintenance

During the coding phase, attempts were made to implement maintenance functions. However, deployment was not completely feasible due to a fundamental constraint within Spark. Specifically, Spark does not allow DELETE or UPDATE operations as it adheres to an immutable paradigm. Modifications to data are restricted, necessitating the creation of new objects rather than direct alterations.

We tried to work with Spark's immutability and we coded it like this to simulate an UPDATE-like action:

- Select the relevant rows from the table.
- Overwrite the initial table by replacing it with the selected rows.

Unfortunately, when dealing with substantial data volumes, such as 3.5 GB, this process becomes exceptionally time-consuming. The query execution time, compounded by the extensive duration required for overwriting the entire table (as it necessitates rewriting the entire dataset), significantly hampers efficiency.

3.5 Code Availability

The implementation and code used for this project are available at this GitHub link for further reference and examination. This code includes the scripts, notebooks, and any additional resources utilized in the project.

4 Results

In this benchmark analysis, output times are consistently presented in seconds across all graphs and were chosen due to their widespread usage in performance measurements.

4.1 Linear Growth of QphDS and Scalability

Our primary inquiry delved into understanding the correlation between the QphDS metric and the scale factor. Figure 1 presents a clear depiction indicating a linear relationship between QphDS and the scale factor. This finding implies a consistent performance trend as the dataset size increases, signifying the system's proportional efficiency in managing larger data volumes.

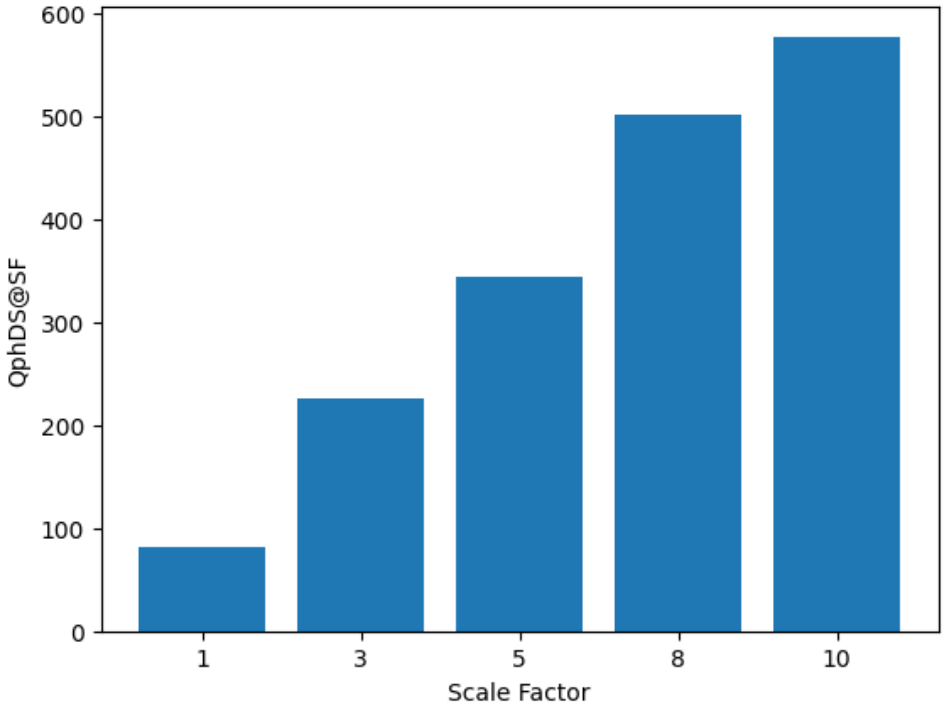


Figure 1: QphDS@S value by scale factor

4.2 Execution Time Measurements

Figure 2 illustrates the execution times by scale factor, broken up by type of test. Notably, the Load test emerged as the most time-intensive process across all scale factors. The extended duration of this phase could be attributed to the complexities involved in loading substantial data volumes into the system. As the scale factor increases, the time required for data ingestion and preparation proportionally rises.

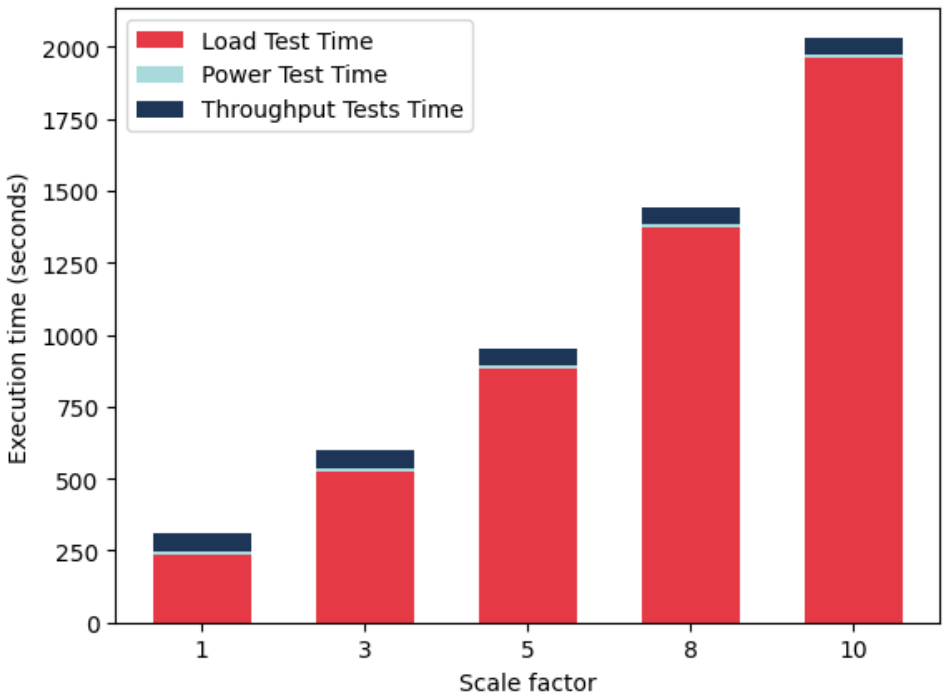


Figure 2: Execution time by scale factor and type of test

4.3 Query Execution Time Analysis

Figure 5 illustrating individual query execution times in the Power Test shows variations in execution times for each query across different scale factors. Despite these variations, the execution times of individual queries did not exhibit significant changes in response to the scale factor alterations. This finding suggests that the performance of individual queries remained relatively consistent, irrespective of the dataset's size.

Figure 4 aims to discern any potential exponential trends in query execution times. The graph confirms the aforementioned hypothesis: all the queries have a constant performance for scale factors 1, 3, 5, and 8, while only a few of them show an increase in the execution time from SF 8 to SF 10.

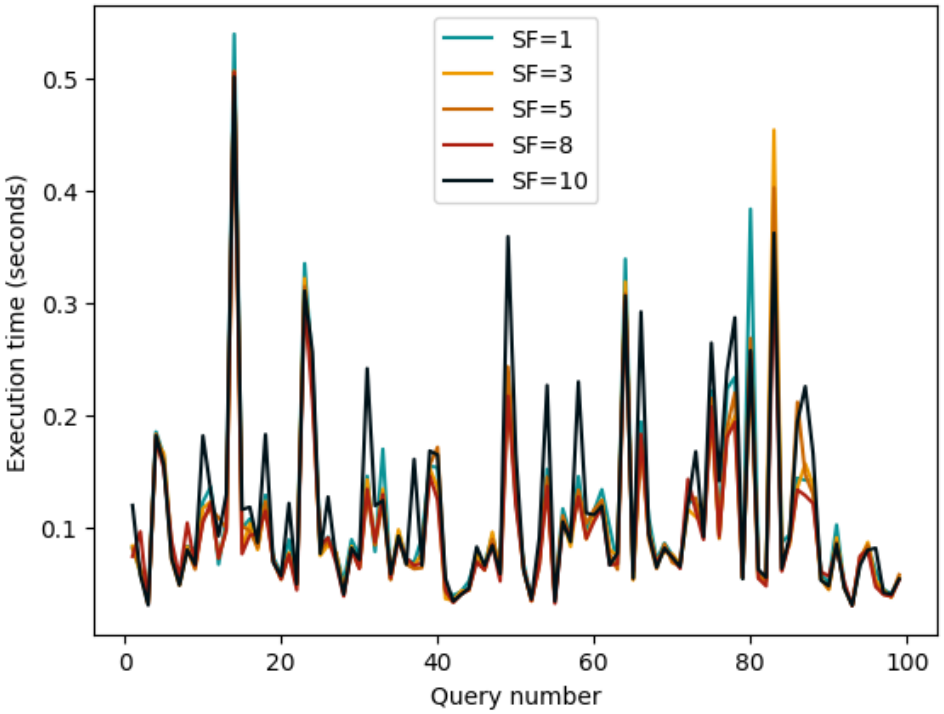


Figure 3: Individual query execution time in the Power Test

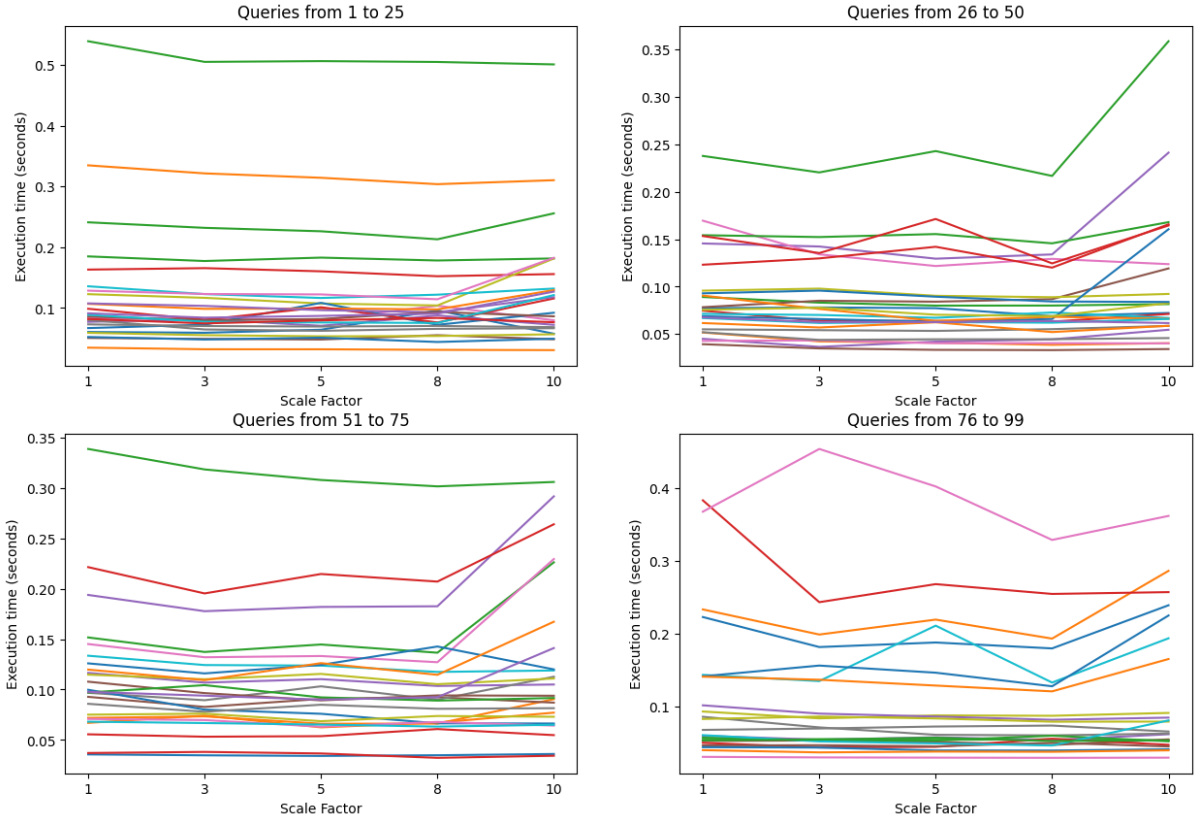


Figure 4: Queries behaviour in the Power Test

5 Discussion

5.1 Discussion on the Results

Spark SQL was able to maintain efficiency and scalability within the range of scales tested. This holds true not only in aggregate analyses but also for individual queries. If on one side this consistency in performance as the dataset size increases are indicative of the system's ability to handle larger data volumes effectively, on the other extrapolating this trend to significantly larger scales necessitates caution, as potential constraints or bottlenecks might be encountered beyond the observed range.

The evident time intensity of the Load test phase, particularly with increased scale factors, suggests a need for optimization in data loading processes to enhance overall system efficiency. For example, this may be achieved by using more efficient file formats for data, such as Parquet. Contrarily, the relatively stable performance of the Power test and Throughput test phases across varied scales implies their insensitivity to changes in the dataset's size, potentially requiring less immediate optimization efforts.

5.2 Optimization

We discussed that Spark SQL showed good results in terms of execution time. Nevertheless, query optimization may still be useful, especially if the user works on less powerful machines. Spark SQL uses the Catalyst optimizer, which creates a logical query tree based on the SQL code and then applies a series of rules and optimizations to the logical query tree. Those are the main peculiarities of Spark SQL:

- instead of executing the query directly, Catalyst generates optimized Java or Scala code that will execute more efficiently;
- Catalyst is able to optimize rows and columns, such as "deleting" unused rows and columns and re-sorting to improve performance, among other ways to reduce redundancy and improve processing.

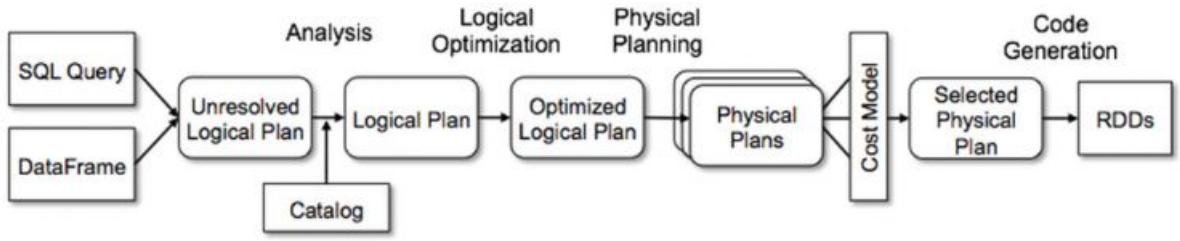


Figure 5: Catalyst optimizer [2]

Follow, a set of optimizations made on some of the most time-consuming queries that took more time, alongside some considerations about them.

```
1
2 SELECT AVG(ss_quantity) AS avg_ss_quantity,
3        AVG(ss_ext_sales_price) AS avg_ss_ext_sales_price,
4        AVG(ss_ext_wholesale_cost) AS avg_ss_ext_wholesale_cost,
5        SUM(ss_ext_wholesale_cost) AS sum_ss_ext_wholesale_cost
6 FROM store_sales
```

```
JOIN store ON s_store_sk = ss_store_sk
JOIN date_dim ON ss_sold_date_sk = d_date_sk
JOIN customer_demographics ON ss_hdemo_sk = hd_demo_sk
JOIN household_demographics ON cd_demo_sk = ss_cdemo_sk
JOIN customer_address ON ss_addr_sk = ca_address_sk
WHERE d_year = 2001
AND (
  (
    (cd_marital_status = 'U' AND cd_education_status = '4 yr Degree' AND
    ss_sales_price BETWEEN 100.00 AND 150.00 AND hd_dep_count = 3)
    OR
    (cd_marital_status = 'S' AND cd_education_status = 'Unknown' AND
    ss_sales_price BETWEEN 50.00 AND 100.00 AND hd_dep_count = 1)
    OR
    (cd_marital_status = 'D' AND cd_education_status = '2 yr Degree' AND
    ss_sales_price BETWEEN 150.00 AND 200.00 AND hd_dep_count = 1)
  )
  AND
  (
    (ca_country = 'United States' AND ca_state IN ('CO', 'MI', 'MN') AND
    ss_net_profit BETWEEN 100 AND 200)
    OR
    (ca_country = 'United States' AND ca_state IN ('NC', 'NY', 'TX') AND
    ss_net_profit BETWEEN 150 AND 300)
    OR
    (ca_country = 'United States' AND ca_state IN ('CA', 'NE', 'TN') AND
    ss_net_profit BETWEEN 50 AND 250)
  )
);
```

Listing 5: Query 13 optimized

```
SELECT i_product_name,
       i_brand,
       i_class,
       i_category,
       AVG(inv_quantity_on_hand) AS qoh
FROM inventory
JOIN date_dim ON inv_date_sk = d_date_sk
JOIN item ON inv_item_sk = i_item_sk
WHERE d_month_seq BETWEEN 1201 AND 1201 + 11
GROUP BY ROLLUP(i_product_name, i_brand, i_class, i_category)
ORDER BY qoh, i_product_name, i_brand, i_class, i_category
LIMIT 100;
```

Listing 6: Query 22 optimized

```

WITH frequent_ss_items AS (
    SELECT SUBSTRING(i_item_desc, 1, 30) AS itemdesc, i_item_sk AS item_sk,
    d_date AS solddate, COUNT(*) AS cnt
    FROM store_sales
    JOIN date_dim ON ss_sold_date_sk = d_date_sk
    JOIN item ON ss_item_sk = i_item_sk
    WHERE d_year IN (2000, 2000 + 1, 2000 + 2, 2000 + 3)
    GROUP BY SUBSTRING(i_item_desc, 1, 30), i_item_sk, d_date
    HAVING COUNT(*) > 4
),

max_store_sales AS (
    SELECT MAX(csales) AS tpcds_cmax
```

```
    FROM (
        SELECT c_customer_sk, SUM(ss_quantity * ss_sales_price) AS csales
        FROM store_sales
        JOIN customer ON ss_customer_sk = c_customer_sk
        JOIN date_dim ON ss_sold_date_sk = d_date_sk
        WHERE d_year IN (2000, 2000 + 1, 2000 + 2, 2000 + 3)
        GROUP BY c_customer_sk
    )
),

best_ss_customer AS (
    SELECT c_customer_sk, SUM(ss_quantity * ss_sales_price) AS ssales
    FROM store_sales
    JOIN customer ON ss_customer_sk = c_customer_sk
    GROUP BY c_customer_sk
    HAVING SUM(ss_quantity * ss_sales_price) > (95 / 100.0) * (SELECT * FROM
    max_store_sales)
)

SELECT c_last_name, c_first_name, sales
FROM (
    SELECT c_last_name, c_first_name, SUM(cs_quantity * cs_list_price) AS sales
    FROM catalog_sales
    JOIN customer ON cs_bill_customer_sk = c_customer_sk
    JOIN date_dim ON cs_sold_date_sk = d_date_sk
    WHERE d_year = 2000 AND d_moy = 3 AND cs_item_sk IN (SELECT item_sk FROM
    frequent_ss_items)
    AND cs_bill_customer_sk IN (SELECT c_customer_sk FROM best_ss_customer)
    AND cs_bill_customer_sk = c_customer_sk
    GROUP BY c_last_name, c_first_name

    UNION ALL

    SELECT c_last_name, c_first_name, SUM(ws_quantity * ws_list_price) AS sales
    FROM web_sales
    JOIN customer ON ws_bill_customer_sk = c_customer_sk
    JOIN date_dim ON ws_sold_date_sk = d_date_sk
    WHERE d_year = 2000 AND d_moy = 3 AND ws_item_sk IN (SELECT item_sk FROM
    frequent_ss_items)
    AND ws_bill_customer_sk IN (SELECT c_customer_sk FROM best_ss_customer)
    AND ws_bill_customer_sk = c_customer_sk
    GROUP BY c_last_name, c_first_name
)
ORDER BY c_last_name, c_first_name, sales
LIMIT 100;
```

Listing 7: Query 23 optimized

```
SELECT SUM(ss_quantity) AS sum_ss_quantity
FROM store_sales
JOIN store ON s_store_sk = ss_store_sk
JOIN date_dim ON ss_sold_date_sk = d_date_sk
JOIN customer_demographics ON cd_demo_sk = ss_cdemo_sk
JOIN customer_address ON ss_addr_sk = ca_address_sk
WHERE d_year = 2001
AND (
  (
    cd_marital_status = 'W' AND cd_education_status = '2 yr Degree' AND
    ss_sales_price BETWEEN 100.00 AND 150.00
  )
  OR
```

```
  (
    cd_marital_status = 'S' AND cd_education_status = 'Advanced Degree' AND
    ss_sales_price BETWEEN 50.00 AND 100.00
  )
  OR
  (
    cd_marital_status = 'D' AND cd_education_status = 'Primary' AND
    ss_sales_price BETWEEN 150.00 AND 200.00
  )
)
AND (
  (
    ca_country = 'United States' AND ca_state IN ('IL', 'KY', 'OR') AND
    ss_net_profit BETWEEN 0 AND 2000
  )
  OR
  (
    ca_country = 'United States' AND ca_state IN ('VA', 'FL', 'AL') AND
    ss_net_profit BETWEEN 150 AND 3000
  )
  OR
  (
    ca_country = 'United States' AND ca_state IN ('OK', 'IA', 'TX') AND
    ss_net_profit BETWEEN 50 AND 25000
  )
);
```

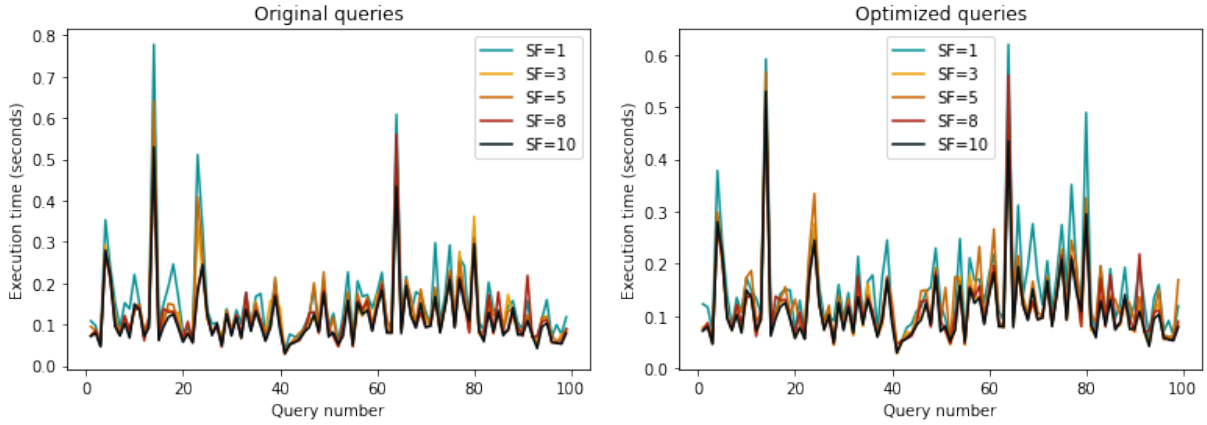
Listing 8: Query 48 optimized

```
SELECT i_item_id, i_item_desc, i_current_price
FROM item
JOIN inventory ON inv_item_sk = i_item_sk
JOIN date_dim ON d_date_sk = inv_date_sk
JOIN store_sales ON ss_item_sk = i_item_sk
WHERE i_current_price BETWEEN 69 AND 69 + 30
AND d_date BETWEEN DATE '1998-06-06' AND (DATE '1998-06-06' + 60)
AND i_manufact_id IN (105, 513, 180, 137)
AND inv_quantity_on_hand BETWEEN 100 AND 500
GROUP BY i_item_id, i_item_desc, i_current_price
ORDER BY i_item_id
LIMIT 100;
```

Listing 9: Query 82 optimized

Most of them are optimized through JOINS instead of CROSS PRODUCTS between tables. Those optimization does not improve significantly the results in Spark.

SCALE	ORIGINAL	OPTIMIZED
1	15.28	15.36
3	12.30	11.50
5	13.09	13.40
8	12.65	11.95
10	11.42	11.25



(a) Original (b) Optimized

Figure 6: Original vs optimized

Those results were generated in a different computer from the main analysis, the results could vary. The times are almost the same, the only query which improve over the scales was query 23, with almost 50% of time consumption.

SCALE 1		
Query	ORIGINAL	OPTIMIZED
13	0.1028	0.1040
22	0.0726	0.0680
23	0.5109	0.2162
48	0.0818	0.1551
82	0.0844	0.0693

SCALE 3		
Query	ORIGINAL	OPTIMIZED
13	0.1149	0.0952
22	0.0571	0.0566
23	0.3408	0.1746
48	0.0855	0.0903
82	0.0639	0.0652

SCALE 5		
Query	ORIGINAL	OPTIMIZED
13	0.1162	0.1065
22	0.0638	0.0789
23	0.4086	0.2286
48	0.0966	0.0842
82	0.0790	0.0898

SCALE 8		
Query	ORIGINAL	OPTIMIZED
13	0.1053	0.1144
22	0.0807	0.0570
23	0.3587	0.1868
48	0.0870	0.0786
82	0.0637	0.0647

SCALE 10		
Query	ORIGINAL	OPTIMIZED
13	0.0889	0.0924
22	0.0651	0.0557
23	0.3786	0.1754
48	0.0855	0.0833
82	0.0805	0.0584

References

[1] Apache Spark. *Apache Spark SQL Documentation*. Accessed on October 07, 2023. URL: https://spark.apache.org/sql/.

[2] Databricks. *Catalyst Optimizer*. Accessed on October 31, 2023. URL: https://www.databricks.com/glossary/catalyst-optimizer#:~:text=At%20the%20core%20of%20Spark,build%20an%20extensible%20query%20optimizer.

[3] TPC. *TPC - Transaction Processing Performance Council*. Accessed on October 05, 2023. URL: https://www.tpc.org/.

[4] TPC. *TPC-DS Benchmark Version 3.2.0*. Accessed on October 27, 2023. URL: https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DS_v3.2.0.pdf.