SuppressWarning

ECOLE POLYTECHNIQUE DE BRUXELLES

Academic Year:2023

INFOH419 - DATA WAREHOUSES

Evaluating Data Warehousing Performance with Apache Hive: TPC-DS Analysis

Yutao Chen 000585954

Min Zhang 000586970

Ziyong Zhang 000585736

Xianyun Zhuang 000586733

Professor: Zimányi, Esteban

Fall 2023

Contents

List of Figures

List of Tables

Listings

1 Distributed Data Warehouse with Apache Hive

1.1 Overview of Data Warehouses

In the early 1990s, businesses recognized the need for advanced data analysis to enhance decision-making. Traditional databases, designed for daily operations, fell short. They prioritized concurrent user access, data consistency, and recovery. However, they struggled with complex queries and data integration from various systems. In response, data warehousing emerged as a solution to meet evolving decision-making needs[VZ14].

The classic definition of a data warehouse, as established by Inmon, outlines its fundamental characteristics[VZ14]:

- **Subject-Oriented:** Data warehouses are designed with specific subjects or areas of analysis in mind, tailored to meet the requirements of managers at different levels of the decision-making hierarchy. For instance, in a retail company, a data warehouse might be structured to facilitate the analysis of product inventory and sales.
- **Integrated:** Data warehouses are the result of integrating data from a variety of operational and external systems, ensuring a holistic view of the organization's data.
- **Nonvolatile:** Data warehouses are intended to accumulate data from operational systems over an extended period, without allowing data modification or removal, except for the purging of obsolete information.
- **Time-Varying:** Data warehouses retain historical data to track the evolution of data over time. This feature enables in-depth analysis of changes in, for instance, sales figures over months or years.

1.2 Distributed Data Warehouse

Distributed Data Warehouse (DDW) refers to a data storage system where data is distributed across multiple nodes, enabling data intercommunication and sharing. Unlike traditional data warehouses that use a single storage node, leading to issues like unbalanced storage loads, data redundancy, and performance bottlenecks, DDWs effectively address these challenges.

The two main features of DDWs are:

- **Distributed Storage:** Data is stored across multiple nodes, each with its own storage unit and processor.
- **Distributed Processing:** Computational tasks are distributed and processed in parallel across different nodes to enhance efficiency and throughput.

1.3 Data Warehousing with Apache Hive

Apache Hive plays a pivotal role in the field of data warehousing, offering a robust framework for managing, querying, and analyzing large datasets stored in distributed storage environments[Hivssa].

Apache Hive provides several key attributes that make it an effective tool for data warehousing:

- **SQL-like Query Language:** Hive uses HiveQL, a SQL-like query language, making it accessible for users proficient in SQL.
- **Scalability:** Designed to handle large datasets, Hive proves to be an ideal choice for data warehousing, especially in big data scenarios.
- **Integration with Hadoop Ecosystem:** Hive seamlessly integrates with various components of the Hadoop ecosystem, such as HDFS and HBase, making it suitable for big data analytics.
- **Schema Flexibility:** Hive supports both read and write schema methods, offering flexibility in data management.

The core functionalities of Apache Hive include supporting ETL processes, reporting, and data analysis tasks in data warehousing. It imposes structure on various data formats and provides access to data stored in HDFS or other storage systems like HBase. Hive supports query execution through the Apache Hadoop MapReduce or Apache Tez frameworks.

Hive users can choose between three runtimes, using either the Apache Hadoop MapReduce or Apache Tez frameworks for SQL query execution. While MapReduce is mature and scalable, it may

introduce higher latency. In contrast, Tez is designed for interactive query processing, reducing overhead.

Hive is best suited for scenarios with large data volumes requiring a distributed system. It excels in scalability, performance, extensibility, fault tolerance, and loose coupling with input formats. However, Hive is not intended for online transaction processing but is tailored for traditional data warehousing tasks.

In summary, Apache Hive is a powerful tool for managing and querying large datasets in distributed storage environments, making it a valuable asset in the field of data warehousing.

2 Benchmarking with TPC-DS

2.1 Overview on Benchmarking

When designing a data warehouse, the choice of architecture, particularly in the context of Relational On-Line Analytical Processing (ROLAP), involves trade-offs between popular options like star, snowflake, and constellation schemas. Each architecture has its own advantages and drawbacks, directly influencing query response times.

Post-architecture selection, optimization techniques such as indexing and materializing views play a vital role in balancing query performance with maintenance overhead.

To guide users in making informed decisions, assessing data warehouse performance is crucial. However, this evaluation can be complex. Tailored benchmarks, encompassing a database model and a workload model, offer a structured approach to [DBB07]:

1. Compare system performance under specific conditions.
2. Evaluate the impact of architectural choices and optimizations.

These benchmarks facilitate informed decision-making in the data warehousing domain.

2.2 The TPC-DS Benchmark

2.2.1 Overview of Decision Support Benchmark

The evolution of decision support systems (DSS) has been significantly influenced by the benchmarks developed by the Transaction Processing Performance Council (TPC). TPC-DS, the latest benchmark introduced by TPC, stands as a pivotal industry standard for evaluating the performance of decision support systems, including Big Data platforms. It is designed to reflect the

modern challenges and technologies in decision support, offering a comprehensive evaluation of system performance.

TPC-DS simulates a retail product supplier's business model, but its scope extends far beyond this specific industry. The benchmark encompasses a wide array of elements such as a detailed database schema, data generation, diverse query sets, and data maintenance protocols. These components collectively provide a robust platform for benchmarking modern decision support systems.

Key Characteristics of TPC-DS[Couss]:

- **Extensive Data Analysis:** Designed to handle and analyze large volumes of data, TPC-DS mirrors real-world scenarios where decision support systems sift through extensive datasets.
- **Real-World Business Queries:** It includes a variety of queries that reflect real-world business questions, covering a spectrum of complexities and operational requirements.
- **Diverse Query Types:** The benchmark accommodates different types of queries, from ad-hoc analysis and standard reporting to iterative OLAP and data mining.
- **High Resource Intensity:** Characterized by high CPU and IO load, TPC-DS provides a realistic test environment for decision support systems.
- **Database Maintenance:** It involves periodic synchronization with source OLTP databases through database maintenance functions.
- **Compatibility with Big Data Solutions:** Applicable to a variety of Big Data solutions, including RDBMS and Hadoop/Spark-based systems.

The need for TPC-DS arose from the limitations of earlier benchmarks like TPC-D and TPC-H in addressing the evolving requirements of modern decision support systems. TPC-DS, as a testament to the continuous evolution of these systems, serves as a critical tool for evaluating and advancing their performance in the context of modern technologies and methodologies[NP06].

In essence, TPC-DS provides a versatile benchmarking tool that reflects the multifaceted nature of modern decision support systems, making it an invaluable resource for evaluating their performance in various settings.

2.2.2 TPC-DS Benchmark Suite

The TPC-DS benchmark, a cornerstone in decision support system evaluation, offers a multi-faceted approach to assessing the performance of general-purpose DSS, encompassing query exe-

cution and data maintenance. Recognized as a standard in the OLAP and Data Warehouse domain, TPC-DS has been widely adopted across various analytical platforms like RDBMS, Apache Spark, and Apache Flink. Its versatility is highlighted by its array of SQL queries and the capability to generate diverse data sets of varying sizes[Beass].

Schema Composition: At the heart of TPC-DS lies a schema that mirrors the sales and return processes across three sales channels: in-store, catalog, and online. This schema is anchored by seven central fact tables, each capturing unique aspects of sales and returns, alongside a single table dedicated to inventory management for catalog and online channels. Complementing these are 17 dimension tables, which provide a detailed view of the sales ecosystem, encompassing customer details, store information, and sales channels.

Database Tables: The schema's tables, such as store_sales, catalog_sales, and web_sales, each record individual transactions across different channels. Similarly, tables like store_returns and web_returns track product returns. The schema extends to cover various dimensions of the retail environment, including customer demographics, store details, and product information.

Query Set: Central to TPC-DS are its 99 SQL-99 queries, each designed to simulate real-world business scenarios. These queries, templated to allow variability, serve as a benchmark to evaluate the completeness and efficiency of SQL implementations.

Data Generation: The benchmark utilizes the dsdgen tool to create input datasets, offering scalability from 1GB to 1000GB. This flexibility ensures the benchmark's applicability to systems of varied capacities.

Operational Context: The Beam implementation of TPC-DS is optimized for batch processing and has been validated with Spark, Flink, and Dataflow runners.

2.2.3 Scaling Factors and Database Loading in TPC-DS Benchmarking

In our exploration of the TPC-DS benchmark, we focus on the critical aspect of scaling and database population. The benchmark outlines a range of scale factors that directly correspond to the volume of raw data generated for subsequent analysis. This data is primarily produced using the 'dsdgen' tool.

The standard scale factors defined in the TPC-DS benchmark include sizes like 1TB, 3TB, 10TB, 30TB, 100TB. However, due to constraints related to storage capacity and computational limitations of our cloud server setup, we have adapted these scale factors to more manageable sizes for our testing purposes. Our chosen scale factors are 1GB, 0.1GB, 0.3GB, 0.5GB.

The process of loading this data into a database system is a crucial part of the benchmarking

process, known as the Load Test. This test is designed to evaluate the efficiency and effectiveness of the system in handling data ingestion. The guidelines for the Load Test are as follows:

- The entire data loading process must be automated, eliminating the need for manual intervention.
- In cases where the raw data is generated as flat files, only the time taken for loading this data into the system is measured. Conversely, if the data is streamed directly from the generation phase into the system, the entire process duration is considered for measurement.
- The key metrics for this test include:
  - Load Start Time: The timestamp recorded immediately before the data loading phase commences.
  - Load End Time: The timestamp recorded immediately after the data loading phase concludes.
  - Load Time: This is calculated as the difference between the Load Start and End Times.

Further details on our specific implementation of the Load Test are discussed in Section 4.6 of this report.

2.2.4 Optimizing Database Performance: Insights from TPC-DS Power and Throughput Tests

In our study, we delve into the intricacies of the TPC-DS benchmark, a pivotal tool for assessing database systems. We address the common syntactical and semantic issues found in its reference queries and results, offering a refined set of queries and answers in our repository[cwi23]. Our aim is to enhance compatibility and precision across various database systems.

We have made specific adjustments, such as sorting NULL values, resulting in two distinct sets of reference results to accommodate different interpretations by database systems. These results are provided in tab-separated files, corresponding to their respective SQL query files, to facilitate accurate comparisons.

The TPC-DS benchmark comprises 99 unique queries, each posing a different business question and requiring specific substitution parameters for execution. For instance, Query 1 focuses on identifying customers with a high rate of returned items. Tools like `dsqgen` are utilized to convert these queries into executable SQL queries in various dialects.

A critical aspect of our study is the analysis of the benchmark's performance through the Power Test and the Throughput Test. The Power Test evaluates the time efficiency of each query and the aggregate time for all queries when executed sequentially in a single session. This test is crucial for assessing the speed and efficiency of processing individual queries in a database system.

Conversely, the Throughput Test assesses the database system's capability to handle multiple users simultaneously. It involves executing the entire set of 99 queries multiple times in a concurrent, multi-user environment. This test is vital for evaluating the system's performance under load, simulating real-world conditions where multiple users access the database simultaneously.

Both the Power Test and the Throughput Test are essential for a comprehensive evaluation of database systems. They provide valuable insights into the systems' efficiency, scalability, and robustness in managing complex queries and multiple user requests. Our study includes detailed guidelines for conducting these tests, ensuring a standardized and thorough assessment of database performance.

2.2.5 Optimizing Data Maintenance in TPC-DS Benchmark

Data maintenance plays a crucial role in evaluating the performance of systems under test (SUT) within the TPC-DS benchmark framework. This subsection delves into the implementation and execution of data maintenance operations in our project, focusing on their impact on database performance when handling new data influx.

Implementation Requirements

Data maintenance operations are integral to the benchmark, involving a series of refresh runs equal to the number of query streams in a Throughput Test. Each run, distinct in its dataset generated via dsdgen, must be executed in a non-overlapping manner, ensuring operational isolation.

Maintenance Operations

The operations encompass various database functions, including:

- **Fact Insertion:** Introducing new facts into the database.
- **Fact Deletion:** Removing specific facts using time-based filters.
- **Inventory Data Deletion:** Eradicating inventory data, again governed by time filters.

Performance Metrics

Performance during data maintenance is quantified through specific metrics:

- **Start and End Timestamps:** Recorded before and after each function and run, denoted as DS(i, s) and DE(i, s) respectively.
- **Duration Indicators:** Calculated as the difference between start and end timestamps, represented as DI(i, s) for functions and DI(s) for entire runs.

Constraints and Validation

The execution of these functions, while offering flexibility in terms of decomposition and order, must adhere to certain constraints:

- Preservation of primary/foreign key relationships.
- Mandatory time-stamped output upon completion.
- SQL or procedural SQL implementation, subject to auditor validation.

Staging Area and Refresh Data

An optional staging area, comprising various database objects, is utilized exclusively during the data maintenance phase. The refresh data sets, numbered and corresponding to the number of streams in Throughput Tests, are composed of flat files that can populate the source schema[Tra15].

3 Implementation Details

3.1 Setting up Apache Hive

Our project's infrastructure involved configuring Apache Hive within Docker containers, supported by diverse operating systems. Below is a detailed overview of the environments:

Attribute	**Tencent Cloud Ubuntu**	**Arch Linux Desktop**
OS	Ubuntu 22.04 LTS	Arch Linux (Rolling Release)
Kernel Version	5.15.0-76-generic	6.1.60-1-lts
Architecture	x86_64	x86_64
CPUs	4	16
Memory	7.251GiB	15.03GiB

Table 3.1: System Environments for Hive Setup

Docker and Hive Configuration: Our Docker container was based on Debian GNU/Linux 11 (bullseye), providing a stable and isolated environment for Hive. The key configurations for Hive included:

```
Hadoop Version: Hadoop 3.3.1
Hive Version: Hive 4.0.0-beta-1
Hive Configuration:
  - Cost-Based Optimization (CBO) Enabled: hive.cbo.enable=true
  - Execution Engine: Tez
  - Parallel Execution: Set to 4 (Adjustable based on workload)
  - Scratch and Install Directories: Configured for optimal performance
  - Local Mode for Tez: Enabled for efficient resource utilization
```

Listing 3.1: Hive Configuration Details

```
PRETTY_NAME="Debian GNU/Linux 11 (bullseye)"
NAME="Debian GNU/Linux"
```

```
VERSION_ID="11"
VERSION="11 (bullseye)"
ID=debian
```

Listing 3.2: Docker Base OS Configuration

This setup allowed us to leverage the computational power of our Arch Linux desktop for intensive tasks, while maintaining compatibility with our Ubuntu cloud server. The use of Docker ensured a seamless and consistent environment across these platforms.

3.2 Adapting DDL to Hive

In order to create the corresponding tables and load the data from the '*.dat' files correctly, adjustments were made to the table creation statements based on Hive features and the .dat file format. Two main changes were implemented:

1.Removal of the Primary Key:

TPC-DS provided table creation statements for 24 tables, all of which include primary key constraints. Attempting to execute the original Data Definition Language (DDL) statements as-is results in errors. According to the official Hive documentation, while the latest version of Hive does support the "primary key" clause during table creation, it necessitates the concurrent use of "DISABLE NOVALIDATE." This essentially means that genuine primary key constraints are not effectively established, and support for the "unique" constraint is also lacking [Hivssb].

As a result, we have made modifications to the DDL statements by removing the "primary key" clause from the table creation statements.

2.Addition of Row Delimiter '|':

Since the dat file uses '|' as the separator, a constraint specifying the row delimiter '|' was added to the end of the table creation statements.

3.3 Generating and Loading the Data

Data Generation for this project was carried out using the TPC-DS program, *dsdgen*.

In terms of data loading, and in preparation for upcoming performance testing, we have created a Python tool known as *loadtime.py*. This tool has been specifically designed to streamline the bulk loading of all 24 tables while meticulously recording the total loading time, as mandated by the TPC-DS benchmark requirements.

3.4 Generating Queries

3.4.1 Generating Queries from the Templates

We used the official *dsdgen* tool to generate queries. Since the tool does not natively support the Hive dialect, we conducted a comparison of various dialects and ultimately selected the *Netezza* dialect. The reason for this choice was that it generated queries that closely align with Hive's syntax. The queries generated by the tool are consolidated in a single file.

Additionally, to facilitate query execution and analysis, we developed a Python script called *separate.py*. This script divides the previously generated SQL into 99 separate queries, labeled from q_n*.sql*, $n \in 1,\ldots,99$. This division allows for subsequent query modifications, optimizations, result output, and performance testing.

3.4.2 Adapting Queries to Apache Hive

To ensure the generated queries function correctly in Apache Hive, we addressed syntax and logic errors in several of them before implementation. In particular, we made modifications to queries 7, 17, 19, 24, 25, 43, 44, 48, 60, 68, 69, 71, 84, and 94. Table 3.2 describes the specific issues and the corresponding changes we applied.

Queries Number	Issues	Modification
68	The query makes direct references from one subquery to another, which is not supported by Hive.	Use the WITH clause to create Common Table Expressions(CTEs), store the information derived from subqueries, and reference them in the main query.
19, 69	The subquery returns multiple rows, which is not allowed when using the subquery as a scalar expression.	The generated multiple row values are found identical and repetitive. Use the 'LIMIT 1' statement or the 'DISTINCT' statement to ensure that the subquery generates only a single row value.
71	A missing join condition leads to incorrect execution result and long runtime.	Add the missing join condition to avoid redundant rows.
48, 84	Alias is missing in one of the subqueries and the query uses nested WITH clauses, which are not supported in Hive.	Add an alias for the subquery and use a single WITH clause in each step for better readability.
7, 17, 24, 25, 43, 44, 60, 94	Use single quotes or double quotes in column aliases.	Use backticks as the column alias delimiters ot ensure it works correctly in Hive.

Table 3.2: Queries Modification

3.5 Running Queries

For running queries, we developed a Python script named *runall.py*. This script enables us to access Hive via Beeline using Python and execute all 99 queries in a batch process.

The results of each query are stored in corresponding text files, for example placing the result set of 'q1.sql' into 'q1.txt'. Additionally, it records the individual execution times for each query in 'query_execution_time.csv' and compiles the total execution times for all queries in 'total_execution_time.txt'.

3.6 Load Test

The process of loading the generated data into the database system is referred to as load test. Load time (TLOAD) specifically denotes the time taken to load the generated into the data database. The Load Test consists of two main steps: the data generation step and the data loading step.

First, we use dsdgen tool to create data in flat files, and then we store the generated data in the database through a Python script. The load time is calculated automatically. You can access our Load Test implementation `loadtime.py` in our repository, and the logic is showing below:

```
algorithm:
set timer start
load data into database
set timer end
calculate total_time
write to file
```

3.7 Power Test

Power Test refers to the sequential execution time of all statements in a query stream. The power test is conducted immediately after the load test and can only run one query at a time during the power test.

Power Test Time (TPOWER) measures the time elapsed from the start to the end of a power test. We have developed a script that can execute all queries in the specified directory, save the running results in a txt file, and record the execution time of each query in a CSV file.

You can also refer to our implementation for the Power Test in our repository through `Power_test_run_01gb_times.py`. Let's now explore how it works:

```
algorithm:
open query execution time file to write to CSV
extract the number from the query file name
set timer start
load data into database
set timer end
calculate total_time
write to CSV file
write  running results in a txt file
```

3.8 Throughput Test

The Throughput test is used to measure the shortest time it takes for a system to process the most queries in a multi-user query scenario. Throughput testing must be performed immediately after Power test. The order of Throughput test and Data Maintenance testing is as follows:

1. Throughput Test 1 is performed after Data Maintenance Test 1 and then followed by Throughput Test 2.

2.Data Maintenance Test 2 is conducted.

The number of query streams must be an even number greater than or equal to 4. For this test, we have chosen 4 as the number of query streams.

You can find the implementation of the Throughput test in our repository, there are `run_01gb_6times.py`, `run_03gb_6times.py`, `run_05gb_6times.py`, `run_1gb_6times.py`. The algorithm is as follows:

```

algorithm:
open query execution time file to write to CSV
extract the number from the query file name
create 4 threads
set timer start
for each thread:
    run all the queries
wait for the slowest thread
set timer end
write to CSV file
write  running results in a txt file
```

3.9 Maintenance Test

The Data Maintenance Test is used to test, update, and maintain the Refresh Data. It is executed after the Throughput Test. Our TPC-DS data maintenance process involves the following steps:

1. Generate the dataset that needs to be updated.
2. Load the dataset to be updated into the data warehouse for data conversion.
3. Insert new records into the fact table and delete specific records based on a time criterion.

While writing the SQL file for deletion, we discovered that Hive does not support the ACID (Atomicity, Consistency, Isolation, Durability) feature. Therefore, we are using the 'overwrite' approach to implement deletion, as shown below

```
-- Hive does not support delete, so we use 'overwrite' for deletion processing
-- Delete catalog order information
-- Create a new table that only contains the data to be retained
CREATE TABLE temp_Store_Sales AS
SELECT *
FROM store_Sales
WHERE ss_Sold_Date_Sk NOT IN (SELECT d_date_sk FROM date_dim WHERE d_date BETWEEN '
    1920-09-10' AND '1920-09-18');

-- Overwrite the original table
INSERT OVERWRITE TABLE store_Sales
SELECT * FROM temp_Store_Sales;

-- Delete the temporary table
DROP TABLE temp_Store_Sales;
```

Data Maintenance functions are executed through Python scripts `DataMaintenanceFunctionTest.py` and `maintaince_test.py`. The load time is calculated automatically, and the logic is simple as well:

```
algorithm:
set timer start
run 11 functions
set timer end
calculate total_time
write to file
```

4 Results and Discussion

This section presents a comprehensive analysis of the experimental results obtained from our tests. We have conducted a series of evaluations, including Queries, Load Test, Power Test, Throughput Test, and Maintenance Test, to assess the performance and efficiency of our system under various conditions. These tests are crucial for understanding the practical implications of our research and for validating the theoretical concepts that have been discussed earlier in this report.

4.1 Queries Running Time

In our study, we executed all 99 queries of the TPC-DS benchmark on four different scales: 0.1 GB, 0.3 GB, 0.5 GB, and 1.0 GB. The primary objective was to assess the performance of our setup under varying data loads.

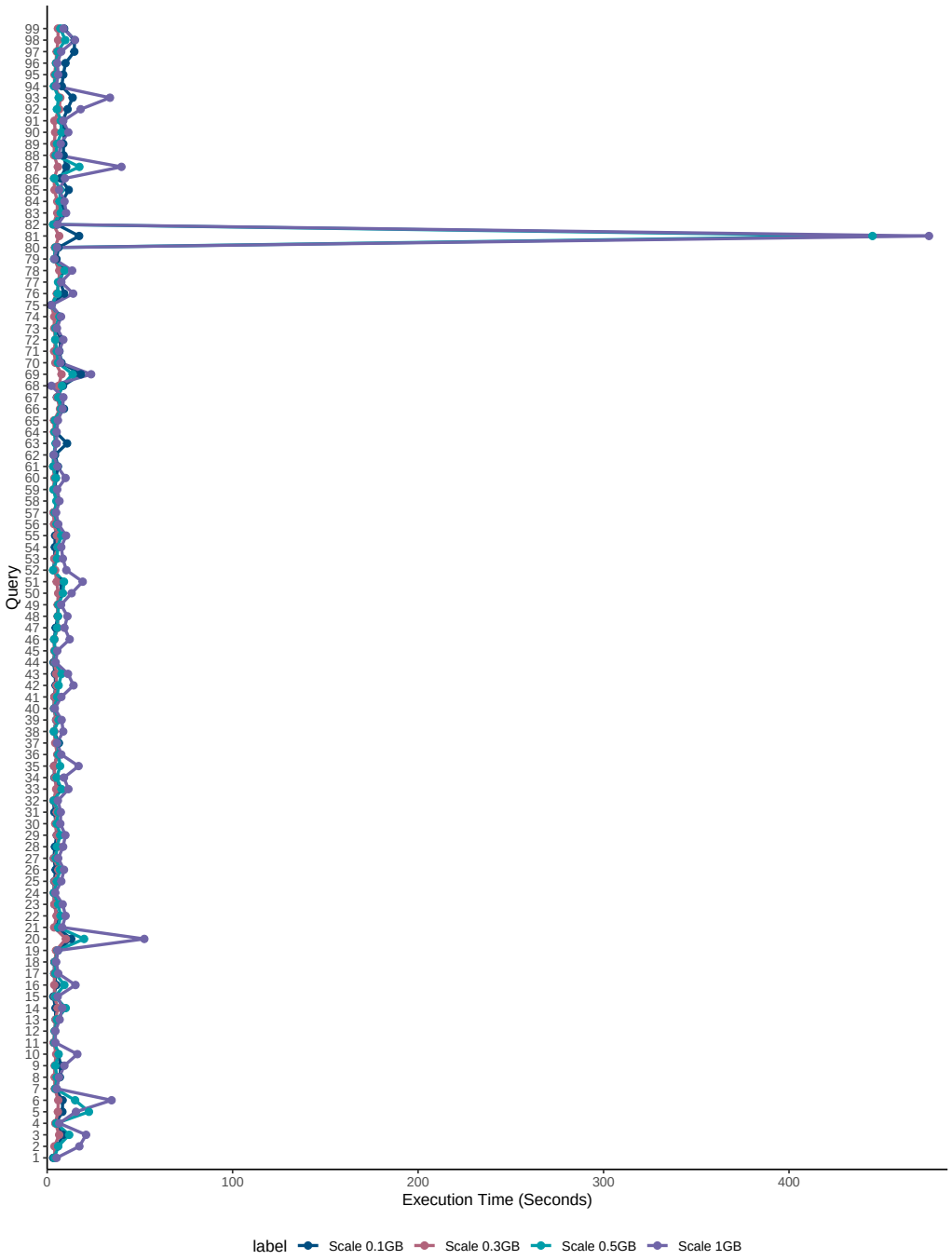


Figure 4.1: Power Test Result from All Scale

We can witness from Figure 4.1 that the execution time of query 81 is too long so we can hardly tell the change of other queries, so let's remove query 81 and get Figure 4.2.

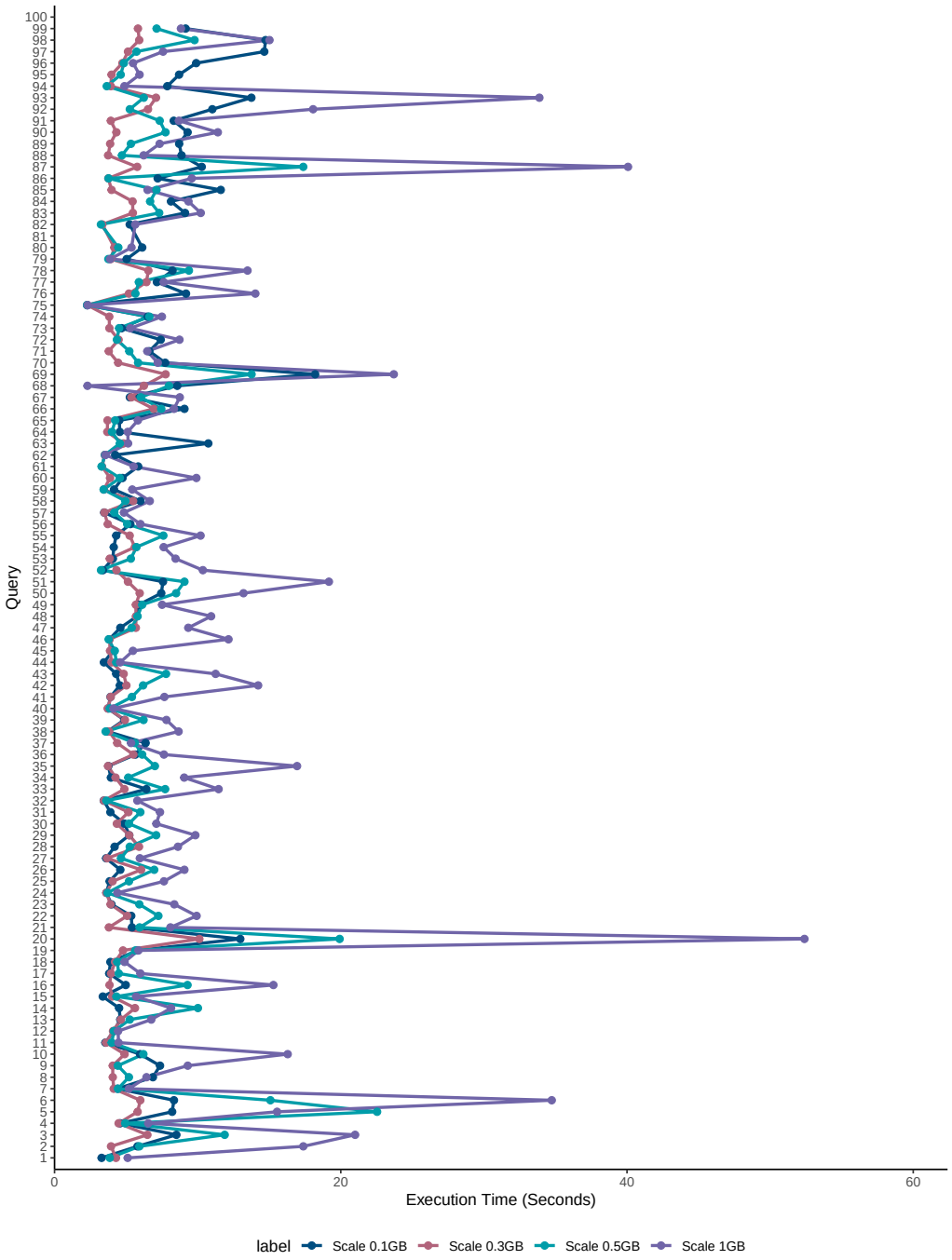
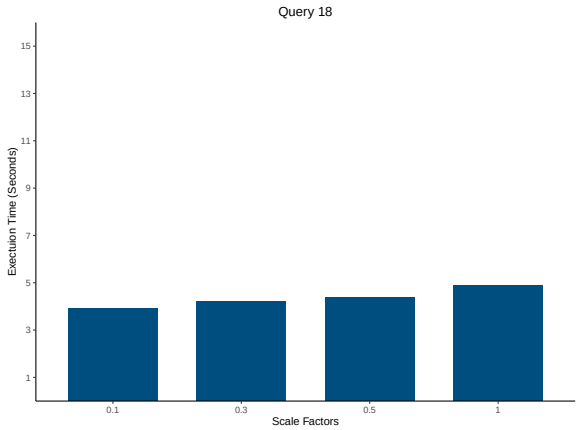
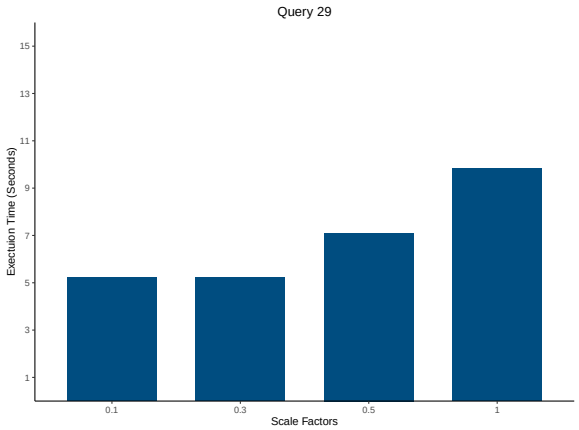


Figure 4.2: Power Test Result from All Scale

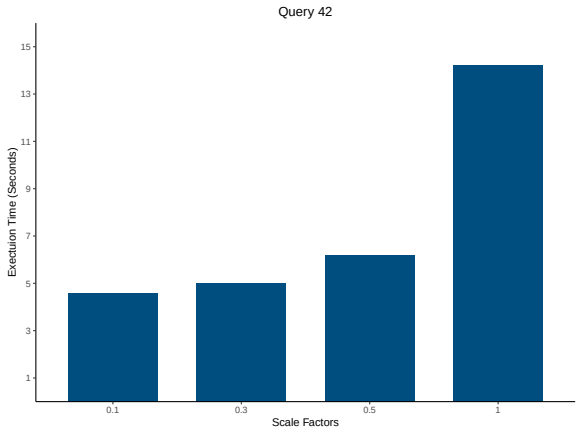
From Figure 4.2, it is evident that some queries demonstrate a linear increase in execution time as the scale increases, as depicted in the 4.3 image. However, we observed that query 81 in the Throughput Test was an outlier. To maintain the integrity and accuracy of our analysis, we made the decision to exclude the results of query 81 from this comparison.



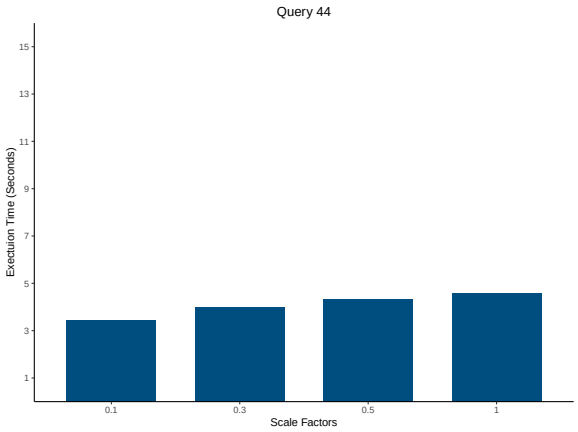
Execution Time Graph of Query 18



Execution Time Graph of Query 29



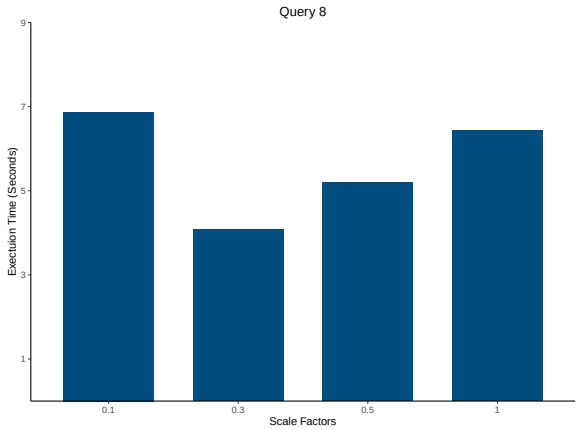
Execution Time Graph of Query 42



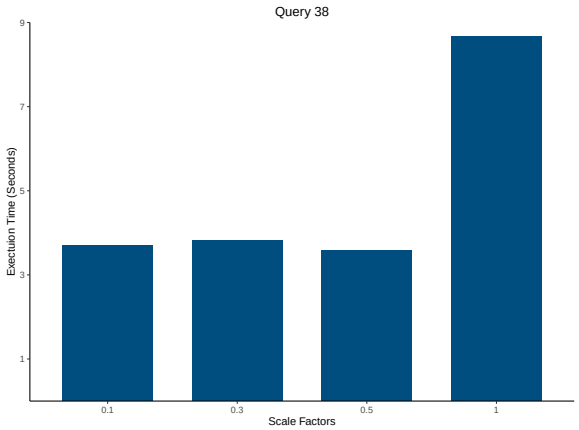
Execution Time Graph of Query 44

Figure 4.3: Execution Time Increase with Scale

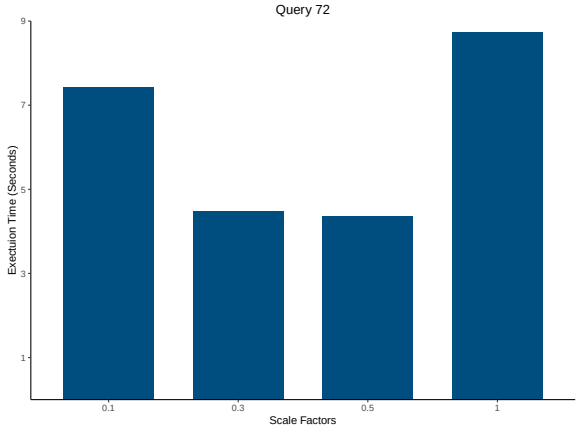
However, intriguingly, not all queries followed this trend. Some showed inconsistent execution time patterns even when the scale increased, as illustrated in Figure 4.4.



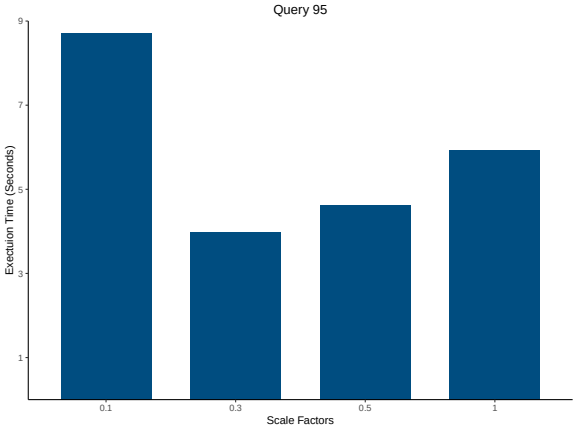
Execution Time Graph of Query 8



Execution Time Graph of Query 38



Execution Time Graph of Query 72



Execution Time Graph of Query 95

Figure 4.4: Execution Time Do Not Increase with Scale

We think two factors might contribute to these observations:

- **Data Skewness**: In some scenarios, data might be skewed in particular scales, which can lead to longer execution times due to uneven distribution of data among reducers.
- **Resource Allocation**: Depending on the cluster's resource allocation, some queries might have access to more resources at particular times, leading to faster execution.

In conclusion, for the general facts that larger scales would generally translate to longer execution times, while various factors inherent to the Hive system and the specific queries can introduce variances in this trend.

4.2 Load Test Results

In the intricate realm of database management, understanding the efficiency of data loading is paramount. Our group delved into this very aspect, focusing on the curious dynamics between

loading times and dataset sizes. The insights drawn from this study not only challenged our initial assumptions but also deepened our appreciation for the underlying intricacies of database operations. As depicted in Figure 4.5, our findings present a puzzling scenario.

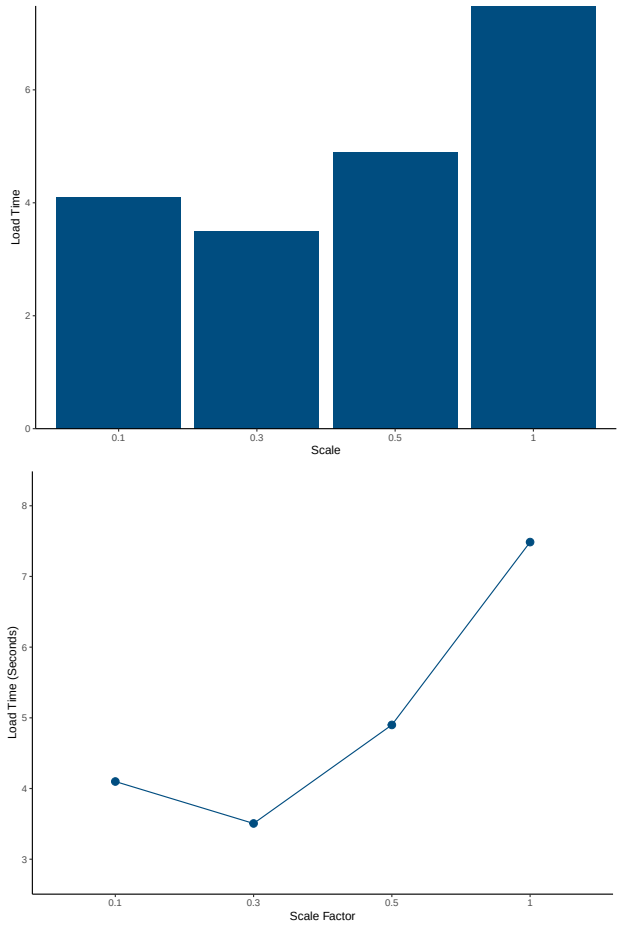


Figure 4.5: Duration of Database Loading

An intriguing observation was made during our load tests, wherein the loading time for 0.1 GB of data was surprisingly greater than that for 0.3 GB. To validate the consistency of this result, we undertook three distinct tests, each of which echoed the same unexpected outcome.

Upon a deeper exploration into this anomaly, we narrowed down the potential contributors to three primary factors:

1. **Hive HDFS Constraints:** Hive primarily uses HDFS files for data storage. In this system, the NameNode manages file metadata such as filenames, directory structures, and block locations. DataNodes, on the other hand, handle the actual data storage. A data deletion in Hive results in the creation of new NameNode files, which can unintentionally lead to an abundance of smaller files. This issue is further magnified in Hive. Managing a vast number of small files during queries notably increases the load on the NameNode, causing heightened I/O overheads. Using TPC-DS's dsdgen tool presented its own challenges. Given its 1GB

minimum data output, we had to partition this to obtain our 0.1GB dataset. This process inevitably produced numerous smaller files, and this fragmentation chiefly accounts for the extended runtime observed for the 0.1GB dataset compared to the 0.3GB one.

2. **Hardware Resource Utilization:** When dealing with smaller datasets, the server's CPU, memory, and disk may not be fully leveraged, resulting in suboptimal performance. Larger datasets, conversely, might tap into these resources more effectively, translating to faster loading times.

3. **Caching Effects:** The benefits of memory caching might not be fully realized with smaller datasets. However, larger datasets have the potential to access cached data more frequently, offering a boost to their load speeds.

4.3 Power Test Results

In our quest to gain a comprehensive understanding of system performance, the Power Test emerges as a critical component. This test, integral to our research, primarily focuses on gauging the efficiency across various scale factors. Figure 4.6 offers a visual representation of the Power Test Time plotted against each scale factor.

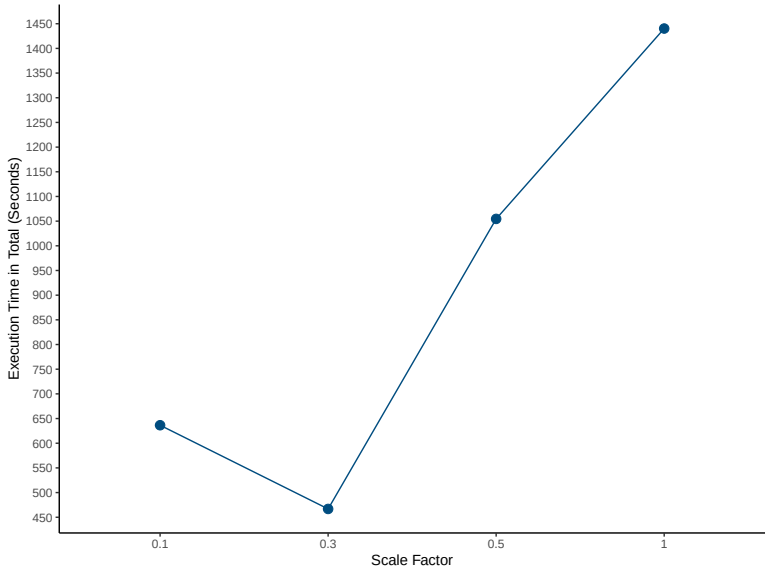


Figure 4.6: Power Test Result from All Scale

A glance at the graph reveals a surprising anomaly. Contrary to our initial hypothesis, which expected a linear correlation between scale factor and Power Test Time, the results depict an irregular pattern of increase. This deviation from our anticipated linear progression prompts a deeper dive into the underlying factors and mechanisms.

Furthermore, since both the Load and Power tests exhibit nearly non-linear growth, we can compare them to check if they share a similar slope. To analyze this, let's refer to Figure 5.3. We've scaled down the time spent on Power testing by a factor of 50 for better visualization. Although both tests show irregularities, they exhibit nearly identical slopes Figure 4.7.

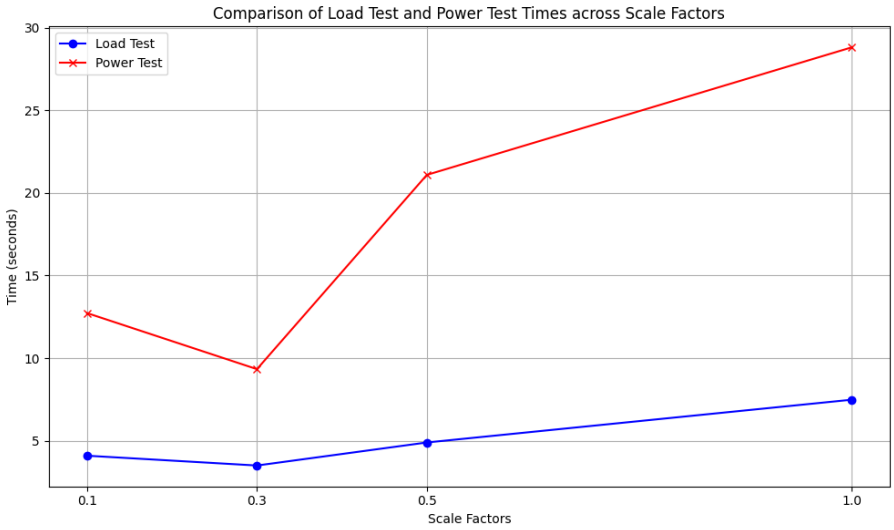


Figure 4.7: Execution Time of Load Test and Power Test.

Drawing a parallel to our prior discussions on load test results in Section **??**, we observed an anomaly in the database loading times, especially when considering the 0.1GB and 0.3GB scale factors. Figure 4.7 further accentuates this discrepancy, revealing another intriguing observation—the execution time for a 0.1GB dataset is noticeably longer than that for a 0.3GB dataset in Hive.

Such behavior, as we mentioned before at Point 1, it is caused mainly by the feature of Hive Distributed File System, every time we cut the dataset into small pieces, more namenode is generated and trigger a long query problem. Apart from this, there are also some factors that can be attributed to several factors inherent to the workings of Hive:

1. **Query Optimization:** Hive tends to optimize queries pertaining to larger tables. Ironically, the same optimizations, when applied to smaller tables, might not be as effective. These optimizations, encompassing aspects like partition pruning, column pruning, and the utilization of statistical information, are particularly beneficial for larger datasets.

2. **Data Loading Duration:** The performance of a query is also contingent upon the time taken for data loading. While data loading might be more time-consuming for larger databases, the query performance subsequently enhances as the data is streamlined through optimization and compression techniques.

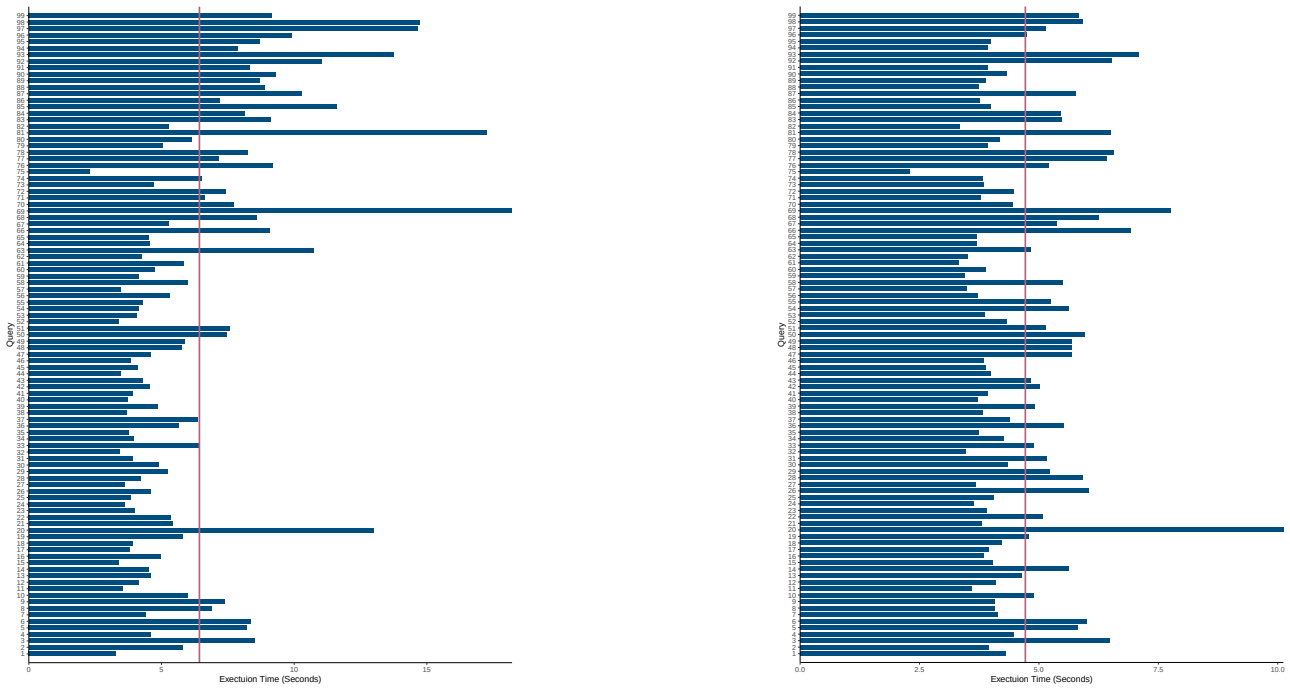
3. **Data Locality:** In the case of smaller databases, there's a possibility that data might not be ideally located near the querying nodes. This necessitates more data transfers, which, in turn, could impede the query performance.

This elucidates a potential inefficiency of the system when grappling with extremely small databases, especially those sized 0.1GB or even lesser. Regardless, a comprehensive analysis of execution times for both smaller datasets (0.1GB, 0.3GB) and larger ones (0.5GB, 1GB) across individual queries and scale factors is warranted.

In the subsequent sections, we shall delve deeper, dissecting the performance metrics for each scale factor, aiming to derive insightful correlations and conclusions.

4.3.1 Scale 0.1, 0.3 Results of Power Test

Figure 4.8 display the execution times of the queries at scale factors of 0.1 and 0.3. It's noticeable that for very small datasets, the running times of all queries are quite similar, with the longest execution time being only 18 seconds.



Power Test Result from 0.1 scale

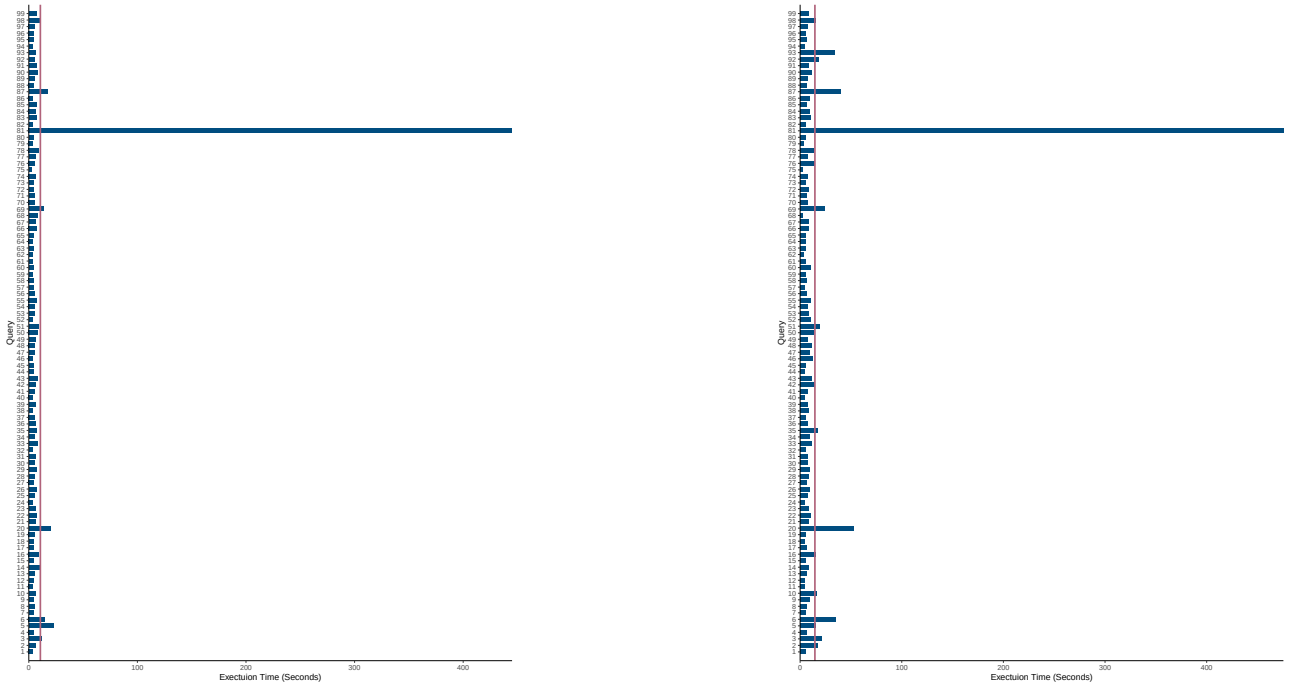
Power Test Result from 0.3 Scale

Figure 4.8: Power Test Results for Scales 0.1 and 0.3

4.3.2 Scale 0.5, 1 Results of Power Test

In Figure 4.9, we observe the execution times for the 99 queries at scale factors of 0.5 and 1. Notably, Query 81 stands out as an outlier, with execution times of 444.84 seconds for Scale 0.5 and

476.45 seconds for Scale 1, considerably deviating from the mean execution time of the other queries.



Power Test Result from 0.5 Scale

Power Test Result from 1 Scale

Figure 4.9: Power Test Result for Scales 0.5 and 1

4.4 Throughput Test Results

Diving further into our investigative study, we now pivot our attention to the throughput test results, a pivotal metric that offers insights into system performance under different user loads and query complexities. Key to our analysis is the evaluation of the results for the 0.1GB dataset. To provide a comprehensive understanding, we'll dissect the data presented in Figure 4.10 and Figure 4.11, each offering a unique perspective on throughput performance. Let's embark on this analytical journey.

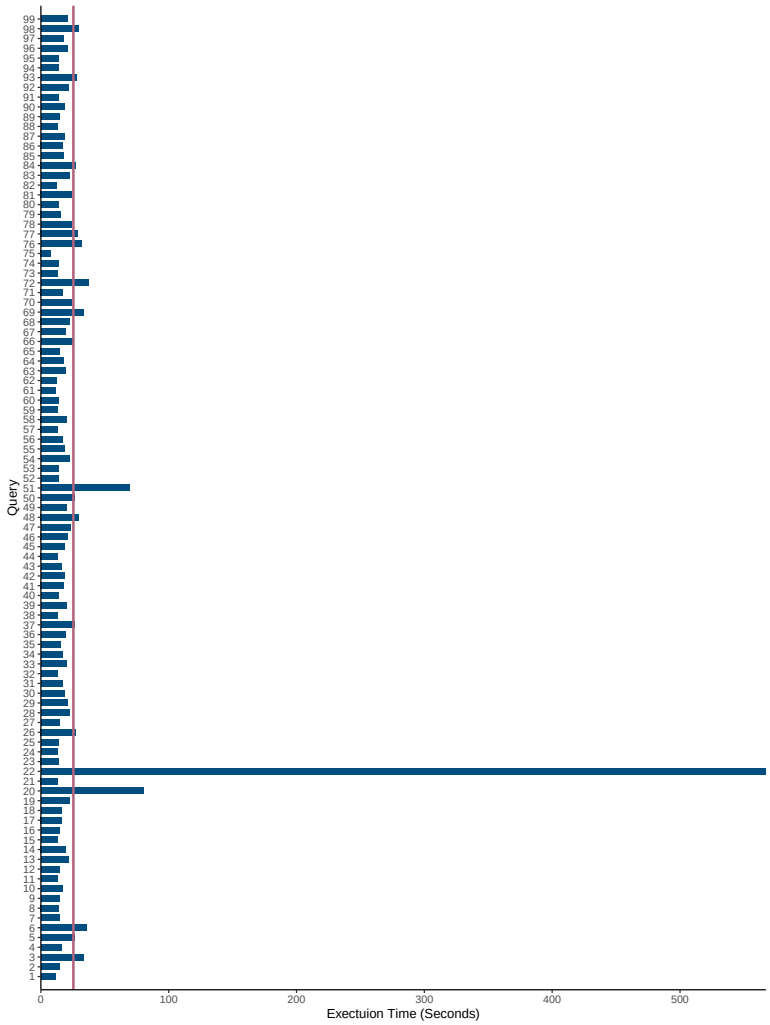


Figure 4.10: Throughput Result by queries

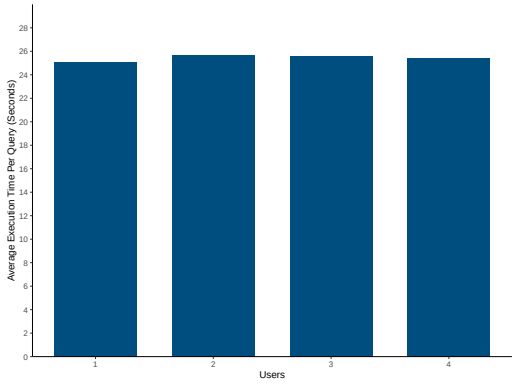


Figure 4.11: Throughput Result by Users

Building upon this foundation, we'll also compare the execution times of the Throughput Test against those of the Power Test and its three variants. With our system running on 4 streams and being configured to accommodate 4 cores, we initially hypothesized that in an optimal setting, the Throughput Test time would mirror the Power Test time. Let's delve deeper to validate or debunk

our assumption.

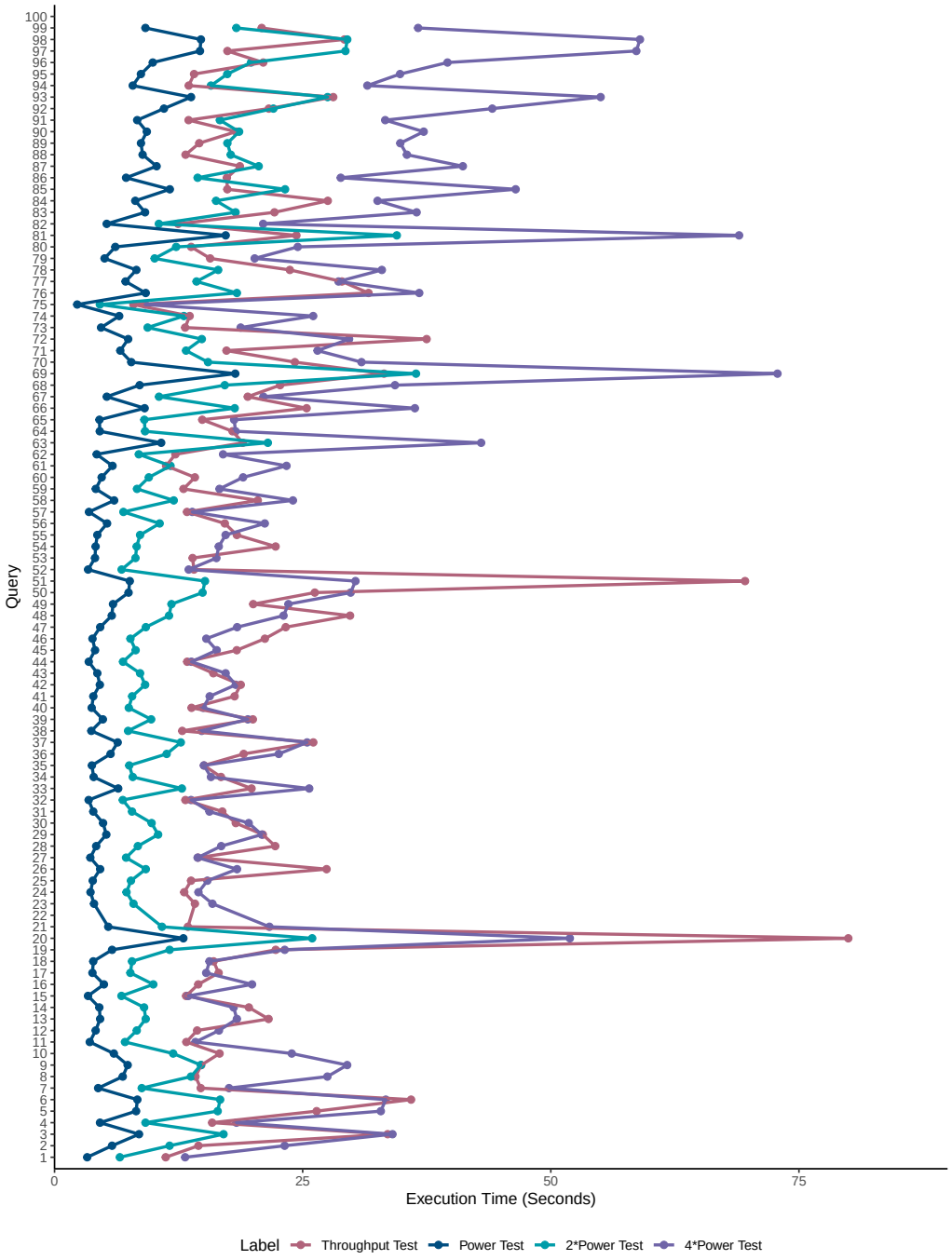


Figure 4.12: Compare Throughput Test with Power Test

Upon closer examination of Figure 4.12, our initial expectations were that the execution time for the Throughput Test would align closely with that of the Power Test. However, we observed that query 22 in the Throughput Test was an outlier. To maintain the integrity and accuracy of our analysis, we made the decision to exclude the results of query 22 from this comparison. Post this exclusion, although the Throughput Test demonstrated a slightly prolonged duration compared to the Power Test, the difference is modest. This indicates that while minor latency might be en-

countered when the system manages multiple tasks concurrently, such delays remain within an acceptable range.

In light of these findings, it's evident that our system exhibits commendable performance in multi-user scenarios, closely paralleling its efficiency in single-user contexts. This underscores the robustness of our setup in a multi-user environment and lays a solid foundation for future optimization endeavors.

4.5 Maintenance Test Results

Our testing framework, comprising four query streams, dictated the implementation of two sequential data refresh and throughput tests. Figure 4.13 aptly visualizes the duration taken by these data update functions.

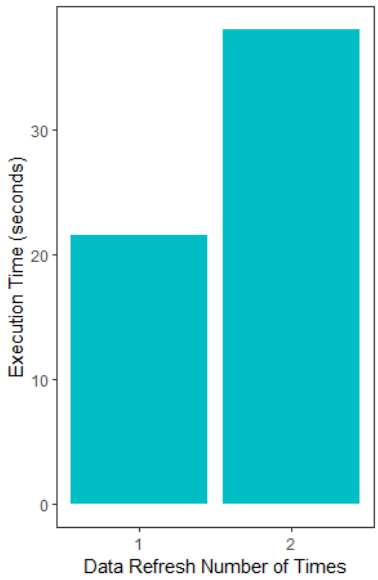


Figure 4.13: Execution Time of Data Update Functions.

Figure 4.14 juxtaposes the results of both the preliminary and subsequent throughput tests, casting light on the dynamics of their performance.

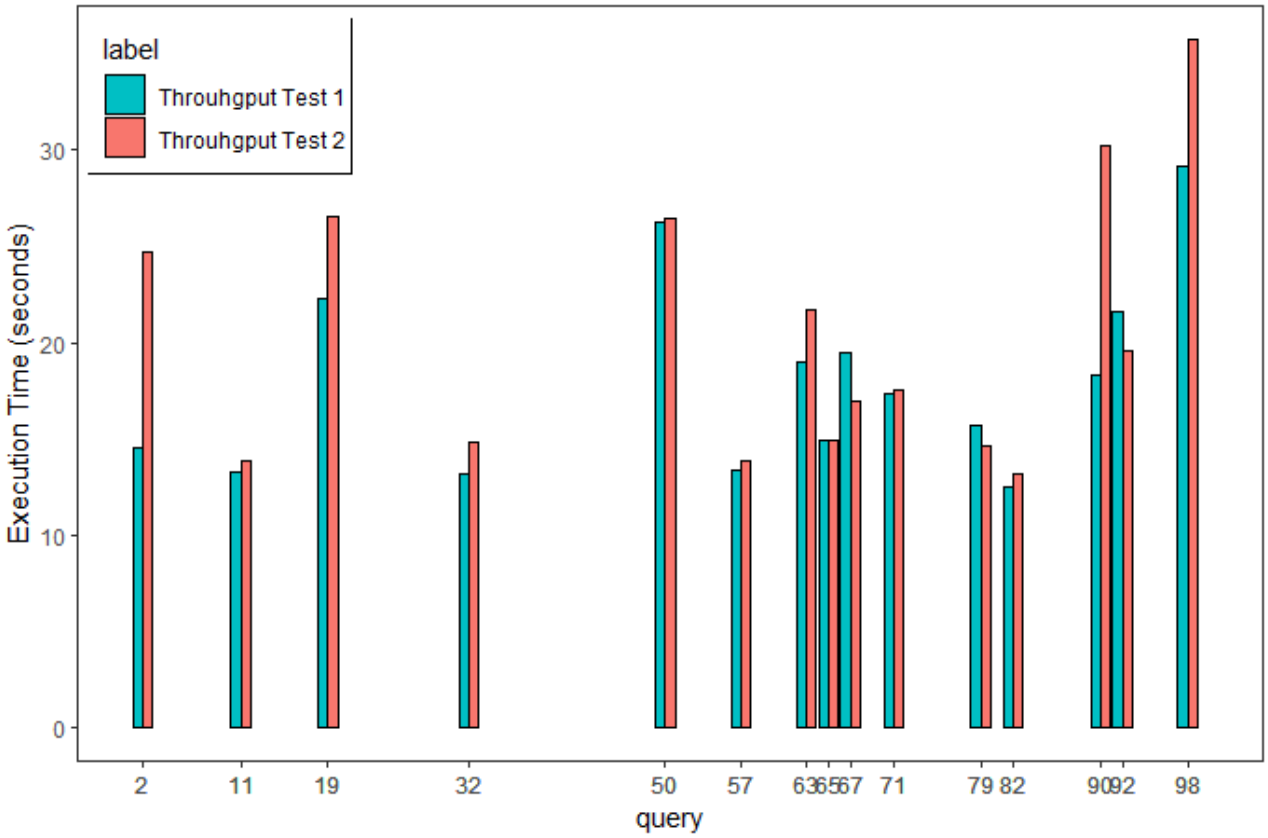


Figure 4.14: Comparative Throughput Test Results.

Interestingly, a marginal uptick in time consumption was observed post data deletion, a trend that at first glance seemed counter-intuitive. This prompted us to run the experiment afresh on our Arch Linux device, results of which are presented in Figure 4.15.

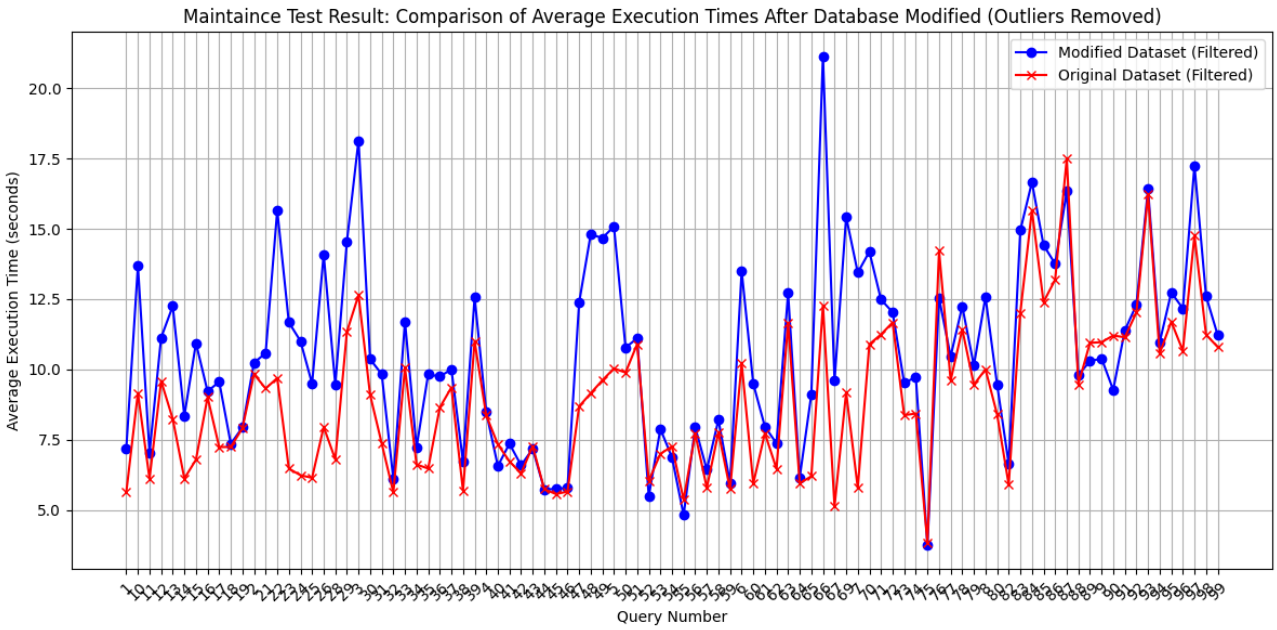


Figure 4.15: Comparative Throughput Test Results through Arch Linux.

The results reaffirmed our initial findings, suggesting longer runtimes despite reduced data volume. And as we mentioned before, due to the feature of Hive HDFS, when we delete the data, we create a new namenode for the system, which will make the query waste more time to traverse all the namenode. Apart from this, there are three plausible explanations:

1. **Metadata Caching:** Hive's metadata might have been cached. Post data removal, a lag in metadata update might be causing queries to still operate based on the erstwhile metadata, potentially influencing query performance.

2. **Hive Statistics:** Hive leans on statistics to optimize queries. With data deletion, unless statistics are updated, Hive could still be referencing outdated statistics, which could degrade performance. Regular execution of `ANALYZE TABLE` is essential to refresh statistics.

3. **Data Partitioning:** If partitions are in use and data is removed from certain partitions, yet the partitions remain intact, queries could be expending unnecessary time processing these now empty or partially populated partitions.

In light of these findings, it's evident that efficient database management goes beyond mere data volume. While data reduction intuitively signals faster query processing, underlying system intricacies can affect the performance. It underscores the importance of periodic metadata updates, consistent statistical analyses, and judicious partition management to harness optimal performance from Hive databases.

4.6 Query Optimization

To enhance query performance and improve the maintainability of the SQL code, we optimized several selected queries. Our optimizations have resulted in shorter execution times and made the queries more modifiable and comprehensible. Specifically, we focused on optimizing queries 19, 40 and 66.

One of the optimization method for the queries is creating indexes. Indexes can help speed up query lookups on certain columns of tables. However, the latest version of Hive that we are using has removed indexing to reduce the index creation cost and the storage of the indexes. Therefore, we were not able to use indexes for the query optimization.

We will now introduce the optimization of these queries in descending order of the performance improvements.

4.6.1 Query 66

The purpose of this query is to count the number of records in the "store_sales" table within specified time intervals. The original query involves joining multiple tables and calculating separate counts for each time interval, aiming to discern the number of sales transactions during different time periods throughout the day.

The original query 66 is listed as Listing 4.1.

```
1  select  *
2  from
3   (select count(*) h8_30_to_9
4   from store_sales, household_demographics , time_dim, store
5   where ss_sold_time_sk = time_dim.t_time_sk
6       and ss_hdemo_sk = household_demographics.hd_demo_sk
7       and ss_store_sk = s_store_sk
8       and time_dim.t_hour = 8
9       and time_dim.t_minute >= 30
10      and ((household_demographics.hd_dep_count = 3 and household_demographics.
            hd_vehicle_count<=3+2) or
11           (household_demographics.hd_dep_count = 1 and household_demographics.
                hd_vehicle_count<=1+2) or
12           (household_demographics.hd_dep_count = 0 and household_demographics.
                hd_vehicle_count<=0+2))
13      and store.s_store_name = 'ese') s1,
14 (select count(*) h9_to_9_30
15  from store_sales, household_demographics , time_dim, store
```

```
16 where ss_sold_time_sk = time_dim.t_time_sk
17     and ss_hdemo_sk = household_demographics.hd_demo_sk
18     and ss_store_sk = s_store_sk
19     and time_dim.t_hour = 9
20     and time_dim.t_minute < 30
21     and ((household_demographics.hd_dep_count = 3 and household_demographics.
           hd_vehicle_count<=3+2) or
22          (household_demographics.hd_dep_count = 1 and household_demographics.
               hd_vehicle_count<=1+2) or
23          (household_demographics.hd_dep_count = 0 and household_demographics.
               hd_vehicle_count<=0+2))
24     and store.s_store_name = 'ese') s2,
25 (select count(*) h9_30_to_10
26 from store_sales, household_demographics , time_dim, store
27 where ss_sold_time_sk = time_dim.t_time_sk
28     and ss_hdemo_sk = household_demographics.hd_demo_sk
29     and ss_store_sk = s_store_sk
30     and time_dim.t_hour = 9
31     and time_dim.t_minute >= 30
32     and ((household_demographics.hd_dep_count = 3 and household_demographics.
           hd_vehicle_count<=3+2) or
33          (household_demographics.hd_dep_count = 1 and household_demographics.
               hd_vehicle_count<=1+2) or
34          (household_demographics.hd_dep_count = 0 and household_demographics.
               hd_vehicle_count<=0+2))
35     and store.s_store_name = 'ese') s3,
36 (select count(*) h10_to_10_30
37 from store_sales, household_demographics , time_dim, store
38 where ss_sold_time_sk = time_dim.t_time_sk
39     and ss_hdemo_sk = household_demographics.hd_demo_sk
40     and ss_store_sk = s_store_sk
41     and time_dim.t_hour = 10
42     and time_dim.t_minute < 30
43     and ((household_demographics.hd_dep_count = 3 and household_demographics.
           hd_vehicle_count<=3+2) or
44          (household_demographics.hd_dep_count = 1 and household_demographics.
               hd_vehicle_count<=1+2) or
45          (household_demographics.hd_dep_count = 0 and household_demographics.
               hd_vehicle_count<=0+2))
46     and store.s_store_name = 'ese') s4,
```

```
47 (select count(*) h10_30_to_11
48 from store_sales, household_demographics , time_dim, store
49 where ss_sold_time_sk = time_dim.t_time_sk
50     and ss_hdemo_sk = household_demographics.hd_demo_sk
51     and ss_store_sk = s_store_sk
52     and time_dim.t_hour = 10
53     and time_dim.t_minute >= 30
54     and ((household_demographics.hd_dep_count = 3 and household_demographics.
           hd_vehicle_count<=3+2) or
55          (household_demographics.hd_dep_count = 1 and household_demographics.
               hd_vehicle_count<=1+2) or
56          (household_demographics.hd_dep_count = 0 and household_demographics.
               hd_vehicle_count<=0+2))
57     and store.s_store_name = 'ese') s5,
58 (select count(*) h11_to_11_30
59 from store_sales, household_demographics , time_dim, store
60 where ss_sold_time_sk = time_dim.t_time_sk
61     and ss_hdemo_sk = household_demographics.hd_demo_sk
62     and ss_store_sk = s_store_sk
63     and time_dim.t_hour = 11
64     and time_dim.t_minute < 30
65     and ((household_demographics.hd_dep_count = 3 and household_demographics.
           hd_vehicle_count<=3+2) or
66          (household_demographics.hd_dep_count = 1 and household_demographics.
               hd_vehicle_count<=1+2) or
67          (household_demographics.hd_dep_count = 0 and household_demographics.
               hd_vehicle_count<=0+2))
68     and store.s_store_name = 'ese') s6,
69 (select count(*) h11_30_to_12
70 from store_sales, household_demographics , time_dim, store
71 where ss_sold_time_sk = time_dim.t_time_sk
72     and ss_hdemo_sk = household_demographics.hd_demo_sk
73     and ss_store_sk = s_store_sk
74     and time_dim.t_hour = 11
75     and time_dim.t_minute >= 30
76     and ((household_demographics.hd_dep_count = 3 and household_demographics.
           hd_vehicle_count<=3+2) or
77          (household_demographics.hd_dep_count = 1 and household_demographics.
               hd_vehicle_count<=1+2) or
78          (household_demographics.hd_dep_count = 0 and household_demographics.
```

```
                hd_vehicle_count<=0+2))
79      and store.s_store_name = 'ese') s7,
80  (select count(*) h12_to_12_30
81  from store_sales, household_demographics , time_dim, store
82  where ss_sold_time_sk = time_dim.t_time_sk
83      and ss_hdemo_sk = household_demographics.hd_demo_sk
84      and ss_store_sk = s_store_sk
85      and time_dim.t_hour = 12
86      and time_dim.t_minute < 30
87      and ((household_demographics.hd_dep_count = 3 and household_demographics.
            hd_vehicle_count<=3+2) or
88            (household_demographics.hd_dep_count = 1 and household_demographics.
                hd_vehicle_count<=1+2) or
89            (household_demographics.hd_dep_count = 0 and household_demographics.
                hd_vehicle_count<=0+2))
90      and store.s_store_name = 'ese') s8
91  ;
```

Listing 4.1: Query 66 - Original version

The primary problem within the original query lies in the repetitive joining of the same set of tables in various sub queries, resulting in redundancy. This duplication of logic for each time interval also implies that any desired time setting modification would need to be implemented in multiple locations.

To address the problem, the optimized query employs conditional aggregation to directly sum the counts based on specific time and minute conditions within a single query. By using conditional SUM function for distinct time interval, it eliminates the redundancy in multiple sub queries. Besides, it adheres to the original logic by joining the same tables and filtering results according to the required criteria.

The optimized query is shown as Listing 4.2.

```
1  SELECT
2    SUM(CASE WHEN t_hour = 8 AND t_minute >= 30 THEN count ELSE 0 END) AS h8_30_to_9,
3    SUM(CASE WHEN t_hour = 9 AND t_minute < 30 THEN count ELSE 0 END) AS h9_to_9_30,
4    SUM(CASE WHEN t_hour = 9 AND t_minute >= 30 THEN count ELSE 0 END) AS h9_30_to_10,
5    SUM(CASE WHEN t_hour = 10 AND t_minute < 30 THEN count ELSE 0 END) AS h10_to_10_30,
6    SUM(CASE WHEN t_hour = 10 AND t_minute >= 30 THEN count ELSE 0 END) AS h10_30_to_11,
7    SUM(CASE WHEN t_hour = 11 AND t_minute < 30 THEN count ELSE 0 END) AS h11_to_11_30,
8    SUM(CASE WHEN t_hour = 11 AND t_minute >= 30 THEN count ELSE 0 END) AS h11_30_to_12,
9    SUM(CASE WHEN t_hour = 12 AND t_minute < 30 THEN count ELSE 0 END) AS h12_to_12_30
```

```
FROM (
  SELECT
    time_dim.t_hour,
    time_dim.t_minute,
    COUNT(*) AS count
  FROM
    store_sales
    JOIN household_demographics ON store_sales.ss_hdemo_sk = household_demographics.
        hd_demo_sk
    JOIN time_dim ON store_sales.ss_sold_time_sk = time_dim.t_time_sk
    JOIN store ON store_sales.ss_store_sk = store.s_store_sk
  WHERE
    store.s_store_name = 'ese'
    AND time_dim.t_hour BETWEEN 8 AND 12
    AND (
      (household_demographics.hd_dep_count = 3 AND household_demographics.hd_vehicle_count
          <= 3 + 2) OR
      (household_demographics.hd_dep_count = 1 AND household_demographics.hd_vehicle_count
          <= 1 + 2) OR
      (household_demographics.hd_dep_count = 0 AND household_demographics.hd_vehicle_count
          <= 0 + 2)
    )
  GROUP BY time_dim.t_hour, time_dim.t_minute
) AS subquery;
```

Listing 4.2: Query 66 - Optimized version

The optimization of this query provides several benefits:

1. The length of the query is greatly shortened, which simplifies the original query and enhance its efficiency and comprehensibility.

2. It reduces the need for multiple sub queries, thus lowering the execution time.

4.6.2 Query 19

The objective of this query is to determine the total sales revenue and their respective standard deviations for distinct product categories. In the original query, the process involves creating three revenue buckets, each categorizing items from various sales channels based on their total sales revenue. These results are then aggregated, standard deviations are calculated for each item, and

items meeting specific criteria are filtered based on the sales revenue for each category. The original query 66 is listed as Listing 4.3.

```
with ss_items as
 (select i_item_id item_id
        ,sum(ss_ext_sales_price) ss_item_rev
 from store_sales
     ,item
     ,date_dim
 where ss_item_sk = i_item_sk
   and d_date in (select d_date
                  from date_dim
                  where d_week_seq = (select d_week_seq
                                      from date_dim
                                      where d_date = '2000-07-21' limit 1))
   and ss_sold_date_sk   = d_date_sk
 group by i_item_id),
 cs_items as
 (select i_item_id item_id
        ,sum(cs_ext_sales_price) cs_item_rev
  from catalog_sales
      ,item
      ,date_dim
 where cs_item_sk = i_item_sk
  and  d_date in (select d_date
                  from date_dim
                  where d_week_seq = (select d_week_seq
                                      from date_dim
                                      where d_date = '2000-07-21' limit 1))
  and  cs_sold_date_sk = d_date_sk
 group by i_item_id),
 ws_items as
 (select i_item_id item_id
        ,sum(ws_ext_sales_price) ws_item_rev
  from web_sales
      ,item
      ,date_dim
 where ws_item_sk = i_item_sk
  and  d_date in (select d_date
                  from date_dim
                  where d_week_seq =(select d_week_seq
```

```
                                    from date_dim
                                    where d_date = '2000-07-21' limit 1))
  and ws_sold_date_sk    = d_date_sk
group by i_item_id)
  select  ss_items.item_id
        ,ss_item_rev
        ,ss_item_rev/((ss_item_rev+cs_item_rev+ws_item_rev)/3) * 100 ss_dev
        ,cs_item_rev
        ,cs_item_rev/((ss_item_rev+cs_item_rev+ws_item_rev)/3) * 100 cs_dev
        ,ws_item_rev
        ,ws_item_rev/((ss_item_rev+cs_item_rev+ws_item_rev)/3) * 100 ws_dev
        ,(ss_item_rev+cs_item_rev+ws_item_rev)/3 average
from ss_items,cs_items,ws_items
where ss_items.item_id=cs_items.item_id
  and ss_items.item_id=ws_items.item_id
  and ss_item_rev between 0.9 * cs_item_rev and 1.1 * cs_item_rev
  and ss_item_rev between 0.9 * ws_item_rev and 1.1 * ws_item_rev
  and cs_item_rev between 0.9 * ss_item_rev and 1.1 * ss_item_rev
  and cs_item_rev between 0.9 * ws_item_rev and 1.1 * ws_item_rev
  and ws_item_rev between 0.9 * ss_item_rev and 1.1 * ss_item_rev
  and ws_item_rev between 0.9 * cs_item_rev and 1.1 * cs_item_rev
order by item_id
        ,ss_item_rev
limit 100;
```

Listing 4.3: Query 19 - Original version

The original query encounters a challenge as it employs the same sub query to compute the date dimension (date_dim) in three separate sub queries. This leads to redundancy and inefficiency in the query execution.

The optimized query makes efficient use of the WITH clause. It focuses on creating a date dimension bucket, thus eliminating redundant date dimension calculations. With this optimization in place, it calculates the total sales revenue for each item based on the date dimension bucket. The final step involves combining the results, computing the standard deviations, and filtering selected items based on the specific criteria, as was done in the original query.

The optimized query 19 is shown as Listing 4.4

```
WITH date_dim_bucket AS (
  SELECT *
  FROM date_dim
```

```
  WHERE d_week_seq = (SELECT d_week_seq
                      FROM date_dim
                      WHERE d_date = '2000-07-21' LIMIT 1)
)

, ss_items AS (
  SELECT i_item_id AS item_id,
         SUM(ss_ext_sales_price) AS ss_item_rev
  FROM store_sales
  JOIN item ON ss_item_sk = i_item_sk
  JOIN date_dim_bucket ON ss_sold_date_sk = d_date_sk
  GROUP BY i_item_id
)

, cs_items AS (
  SELECT i_item_id AS item_id,
         SUM(cs_ext_sales_price) AS cs_item_rev
  FROM catalog_sales
  JOIN item ON cs_item_sk = i_item_sk
  JOIN date_dim_bucket ON cs_sold_date_sk = d_date_sk
  GROUP BY i_item_id
)

, ws_items AS (
  SELECT i_item_id AS item_id,
         SUM(ws_ext_sales_price) AS ws_item_rev
  FROM web_sales
  JOIN item ON ws_item_sk = i_item_sk
  JOIN date_dim_bucket ON ws_sold_date_sk = d_date_sk
  GROUP BY i_item_id
)

SELECT
  ss.item_id,
  ss_item_rev,
  ss_item_rev / ((ss_item_rev + cs_item_rev + ws_item_rev) / 3) * 100 AS ss_dev,
  cs_item_rev,
  cs_item_rev / ((ss_item_rev + cs_item_rev + ws_item_rev) / 3) * 100 AS cs_dev,
  ws_item_rev,
  ws_item_rev / ((ss_item_rev + cs_item_rev + ws_item_rev) / 3) * 100 AS ws_dev,
```

```
    (ss_item_rev + cs_item_rev + ws_item_rev) / 3 AS average
FROM ss_items ss
JOIN cs_items cs ON ss.item_id = cs.item_id
JOIN ws_items ws ON ss.item_id = ws.item_id
WHERE ss_item_rev BETWEEN 0.9 * cs_item_rev AND 1.1 * cs_item_rev
  AND ss_item_rev BETWEEN 0.9 * ws_item_rev AND 1.1 * ws_item_rev
  AND cs_item_rev BETWEEN 0.9 * ss_item_rev AND 1.1 * ss_item_rev
  AND cs_item_rev BETWEEN 0.9 * ws_item_rev AND 1.1 * ws_item_rev
  AND ws_item_rev BETWEEN 0.9 * ss_item_rev AND 1.1 * ss_item_rev
  AND ws_item_rev BETWEEN 0.9 * cs_item_rev AND 1.1 * cs_item_rev
ORDER BY item_id, ss_item_rev
LIMIT 100;
```

Listing 4.4: Query 19 - Optimized version

The results achieved through the optimized query mirror those of the original query. However, the optimized version boasts a considerably shorter execution time, leading to better query performance.

4.6.3 Query 40

The original query is structured into two separate sub queries, one designed for the morning (AM) and the other for the afternoon (PM). Both sub queries aim to calculate the count of sales data within their respective time intervals. The ultimate goal is to compute the sales ratio by utilizing the count results from these two sub queries.

The original query 40 is listed in Listing 4.5

```
select  cast(amc as decimal(15,4))/cast(pmc as decimal(15,4)) am_pm_ratio
 from ( select count(*) amc
       from web_sales, household_demographics , time_dim, web_page
       where ws_sold_time_sk = time_dim.t_time_sk
         and ws_ship_hdemo_sk = household_demographics.hd_demo_sk
         and ws_web_page_sk = web_page.wp_web_page_sk
         and time_dim.t_hour between 10 and 10+1
         and household_demographics.hd_dep_count = 6
         and web_page.wp_char_count between 5000 and 5200) at,
      ( select count(*) pmc
       from web_sales, household_demographics , time_dim, web_page
       where ws_sold_time_sk = time_dim.t_time_sk
         and ws_ship_hdemo_sk = household_demographics.hd_demo_sk
         and ws_web_page_sk = web_page.wp_web_page_sk
```

```
15          and time_dim.t_hour between 18 and 18+1
16          and household_demographics.hd_dep_count = 6
17          and web_page.wp_char_count between 5000 and 5200) pt
18  order by am_pm_ratio
19  limit 100;
```

Listing 4.5: Query 40 - Original version

The issue with the original query lies in the similarity of the filtering conditions between the two sub queries. This redundancy results in inefficient resource utilization, as both sub queries execute nearly identical conditions.

To optimize the original query, we replace the two separate sub queries with a JOIN function, allowing us to match the tables and consolidate the query. The time conditions that are originally duplicated in both sub queries have been transformed into a CASE WHEN function within the SELECT clause.

The optimized query 40 is shown as Listing 4.6

```
1  select
2    cast(count(case when t1.t_hour between 10 and 11 then 1 else null end) as decimal(15,4))
         /
3    cast(count(case when t1.t_hour between 18 and 19 then 1 else null end) as decimal(15,4))
         as am_pm_ratio
4  from
5    web_sales ws
6    join household_demographics hd on ws.ws_ship_hdemo_sk = hd.hd_demo_sk
7    join time_dim t1 on ws.ws_sold_time_sk = t1.t_time_sk
8    join web_page wp on ws.ws_web_page_sk = wp.wp_web_page_sk
9  where
10   hd.hd_dep_count = 6
11   and wp.wp_char_count between 5000 and 5200
12   and (t1.t_hour between 10 and 11 or t1.t_hour between 18 and 19)
13 limit 100;
```

Listing 4.6: Query 40 - optimized version

By merging the two sub queries into a single query, we have streamlined the query structure, significantly reducing complexity and avoiding redundant filtering operations. This optimization has yielded improvements in execution time.

5 Conclusion

5.1 Conclusion

In the course of our project, our team utilized TPC-DS to rigorously test the performance capabilities of Apache Hive. Our methodical approach began with an exploration into the evolution of distributed data warehousing, as elaborated in Charper 1. This foundation was further strengthened with insights into the principles of benchmarking, detailed in Charper 2.

Our study transitioned from theoretical groundwork to practical implementations, as described in Charper 3. Herein, we outlined the specificities of our system environment, our adaptations of DDLs for Hive, the methods of data and query generation, and our strategies for executing diverse tests, such as Load, Power, Throughput, and Maintenance.

The empirical results and their corresponding analyses were consolidated in Charper 4. Reflecting upon our findings, it is evident that Apache Hive, despite its prominence in distributed cloud servers, demonstrates suboptimal performance for smaller datasets. Its inherent HDFS architecture further complicates its efficiency, particularly in operations related to data manipulation.

Bibliography

[Beass] Beam, Apache (Year of Access). *Apache Beam TPC-DS Documentation*. URL: https://beam.apache.org/documentation/sdks/java/testing/tpcds/.

[Couss] Council, Transaction Processing Performance (Year of Access). *TPC Benchmark DS (Decision Support)*. URL: https://www.tpc.org/tpcds/.

[cwi23] cwida (2023). *TPC-DS Result Reproduction*. https://github.com/cwida/tpcds-result-reproduction. Accessed: 2023-10-31.

[DBB07] Darmont, J., F. Bentayeb, and O. Boussaïd (2007). "Benchmarking Data Warehouses". In: *International Journal of Business Intelligence and Data Mining* 2.1, pp. 79–104.

[Hivssa] Hive, Apache (Year of Access[a]). *Apache Hive GitHub Repository*. URL: https://github.com/apache/hive.

[Hivssb] — (Year of Access[b]). *Language Manual DDL*. URL: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+DDL.

[NP06] Nambiar, R. O. and M. Poess (Sept. 2006). "The Making of TPC-DS". In: *VLDB* 6, pp. 1049–1058.

[Tra15] Transaction Processing Performance Council (Nov. 2015). *TPC BENCHMARK DS Standard Specification*. http://www.tpc.org. Version 2.1.0. Accessed: 2023-10-31.

[VZ14] Vaisman, Alejandro and Esteban Zimányi (2014). "Data warehouse systems". In: *Data-Centric Systems and Applications*.