UNIVERSITÉ LIBRE DE BRUXELLES

ÉCOLE POLYTECHNIQUE DE BRUXELLES

A TCP-DS benchmark implementation using Oracle

INFOH419 - DATA WAREHOUSES

Fall 2022

Authors:
Ivanović, Nikola,
Lorencio Abril, Jose Antonio,
Yusupov, Sayyor,
Živković, Bogdana

Professor: **Zimányi, Esteban**

Contents

List of Figures

Listings

Abstract

In this project we have reproduced the TPC-DS Benchmark specification to execute in an Oracle environment. More precisely, we have used Oracle XE, the free version of Oracle SQL systems.

The present report starts reviewing what Data Warehouses and Decision Support Systems are in Chapter 1. In Chapter 2, we reflect on the importance of benchmarks in general and its particular importance when talking about Decision Support Systems. We also analyze the TPC-DS benchmark by means of a summary of its specification. In Chapter 3 the implementations details on how we have taken the specification from theory to practice are explained, with emphasis in the different tests performed in the benchmark, whose results are analyzed in Chapter 4. In this chapter we also explain how we improved the performance of some of the queries of the benchmark. Finally, a brief conclusion statement is presented in Chapter 5.

All the explanations provided in this report refer to the github repository created for the project, which can be accessed with the following link https://github.com/Lorenc1o/TPC-DS_Oracle.

Chapter 1

Data Warehouses and Decision Support Systems

In this chapter, we will briefly review two concepts that are key for the understanding of the present work, as well as its scope.

1.1 Overview on Data Warehouses

According to Vaisman and Zimányi [VZ22], a **data warehouse** can be defined as a *collection of subject-oriented, integrated, nonvolatile, and time-varying data to support management decisions.* These concepts are worth to be detailed:

- Data is *subject-oriented* when it is stored and managed with the idea to analyze specific areas of interest for the organization.
- It is *integrated* when the different sources of data are combined in such a fashion that we output a unique uniform and consistent joined source of data.
- It is *nonvolatile* in the sense that modification and removal of data is disabled. This is possible because we are analyzing data from the past, which cannot be changed (by now, at least!).
- Lastly, we store *time-varying* data, which translates into introducing the possibility to retain the same information with different levels of temporal aggregation.

Usually, a data warehouse is thought as organization-level, but it is not a rare situation that a particular division of the organization only needs a portion of all this stored data, which might be specially modeled for the division's needs. In such a case, this specialized portion of the data warehouse is called **data mart**, and they can be seen as the building blocks of the bigger data warehouse. This definition allows us to think of two main approaches to model a data warehouse:

- The bottom-up approach starts by defining the different datamarts that each department will need, and then integrating them into a full data warehouse.
- The top-down approach is the other way round. We design the data warehouse and, after that, we would generate the datamarts as proper subsets from it.

In a nutshell, a data warehouse should be thought as a database specifically designed for analytical purposes, instead of operational purposes, with all of its consequences.

1.2 Overview on Decision Support Systems

Efraim Turban is a widely known author that has published a wide variety of books and articles on the topic of applying technology to address business problems. For instance, he is one of the pioneers of **Decision Support Systems (DSS)**, which he defined in [Tur83] as *advanced computer-based information systems that provide direct, personal support for data processing and decision making.* He then mentions the following characteristics of DSS:

- On-line interaction user-computer.
- Supporting of the user's judgement.
- DSS aims to improve effectiveness rather than efficiency.
- The decision process is a flexible process that can deal with unstructured problems.

And he then emphasizes the five major components of a DSS, being:

- The data base, meaning the sources of information relative to the process. In our case (and most modern scenarios) this data base will be the data warehouse.
- The model base. With this concept, Turban was referring to all models and programs at hand that could be used to analyze the database, e.g. statistical measures or a Python data analysis library.
- The computer.
- The user, aimed as the center of the process. The DSS should be easy to use, flexible and fast enough to meet with the analysis requirements.
- The communication module. It can be divided into different categories:
  - Computer languages: which should be different for a technical user than for a business-specific user.
  - Entry and display: usually offered via a GUI.
  - Immediate response to questions.
  - Quick experimentation.
  - Monitoring and controlling production processes in real time.

As we can grasp from what have been stated so far, Turban reached a deep understanding of the business process analysis task[1], which today is a highly developed area, with specialized roles for each of the subtasks that have been mentioned.

The importance of Data Warehouses in this area is today categorical, as they serve as the data source from which all rest of the work will take place and on which all decisions will be based. This means that

[1] Bear in mind that he wrote this in 1983.

special attention has to be paid at the moment of Data Warehouse design, tooling and implementation, which justifies the importance of this project for us as aspirants to be experts in this field.

1.3 Data Warehousing with Oracle® Database

This project studies the benchmarking of Oracle as a Data Warehouse Management System from the point of view of Decision Support, so it is mandatory to review Oracle® documentation regarding data warehousing, which, in fact, is thorough and intensive. Thus, we will refer to [Dat21] when some solution is already known for a problem at hand.

Chapter 2

Benchmarking with TPC-DS

As we have already remarked, Data Warehouses play a central role in Decision Support Systems and, as such, their design needs to be addressed with a corresponding level of importance and thoroughness. One of the first decisions an organization has to make when designing a Data Warehouse is which of the existing tools is suitable for the organization's purposes. This is certainly a difficult question to answer, as there is a number of potential tools that can be used with different characteristics in a wide spectrum of aspects. Also, the organization is constrained by the knowledge of their engineers, who will not be able to understand all available tools in depth. This is where benchmarks play a crucial role.

2.1 Overview on benchmarking

In a general perspective, **benchmarking** refers to the process of measuring different indicators of effectiveness, efficiency, correctness,... of a system. In such a way that we can compare the obtained results by different systems, aiding in the difficult task of choosing among distinct and different options. A **benchmark** is, thus, a concrete description of such a process, aimed to address a certain group of problems.

Although the concept is straightforward at first glance, we need to take into account a wide variety of factors when we have an objective and different tools to solve it. The benchmark to use will be itself another decision to make, as the better it adjusts to our particular problem, the more reliable will be the conclusions obtained from its use. At this point, we encounter ourselves with a double-edged sword in our hands:

- The more we want the benchmark to adequate to our problem, the less likely it is to already exist and the more effort will need to be made to develop it.
- The more we take into account the application domain and select the widest used benchmark to make comparisons easier, the less reliable will our conclusions be.

These are the reasons why standardization and categorization are important:

1. Standardization allows developers to decide in a narrower spectrum of possibilities and also makes different results more easily comparable.
2. Categorization, i.e., the grouping of different problems in bigger categories, helps both the task

of standardization and the task of benchmark selection, as we can decide among those standard benchmarks which address the problems that are similar to the one at hand.

This way, when addressing the development of a Decision Support System, it would be suitable to find a standard benchmark that focuses on this problem. That is exactly what TPC-DS is designed for.

2.2 The TPC-DS benchmark

2.2.1 Introduction

TPC-DS [TPC21] is an industry standard benchmark that addresses the benchmarking of several aspects of Decision Support Systems, such as querying the Data Warehouse or data maintenance, and providing a *representative evaluation of the System Under Test's (SUT)*[1] *performance as a general purpose decision support system.*

It is thought to measure the performance of a DSS with the following characteristics:

- Works with large amounts of data.
- Aims at answering real-world questions.
- Makes use of queries with different operational requirements and complexities.
- Entails high CPU and I/O loads.
- Is periodically synchronized with new data from an operational database through data maintenance functions.
- Runs on *Big Data* solutions.

A benchmark result measures different characteristics in different scenarios. It measures:

- Query response time in single user mode.
- Query throughput in multi user mode.
- Data maintenance performance.

All this for a given hardware, OS, and data processing system configuration under a controlled, complex, multi-user decision support workload.

2.2.2 Business and Benchmark Model

TPC-DS uses as model a retail product supplier, considering two main components, which they claim to be the *'two most important components of any mature decision support system'*:

- **User queries**, that converts operational facts into business intelligence.
- **Data maintenance** to synchronize the process of analysis with the operational data source that it needs to be feed with incoming data.

[1] That is, the system that we are testing.

As not all possible operations can be taken into account, they decided to create a set of operations that could be representative for a general analysis system. In their case, their queries are designed to imitate the activity of a multi-channel retailer, tracking store, web and catalog sales channels. The election of this example has also didactic purposes, as a retail product supplier uses concepts that can be easily understood. Obviously, it can be used to assess the performance of systems that model different realities, provided that they must transform operational and external data into business intelligence. Note that TPC-DS does not benchmark the operational systems that would source the data to our warehouse, it is assumed to exist (and for benchmarking the data is (more or less) randomly generated).

TPC-DS' model operates over three sets of assumptions:

- **Data Model and Data Access assumptions**:

  1. The system allows potentially long running and multi-part queries that make the system inavailable for a period of time.
  2. Data tracks the state of an operational DB through data maintenance functions.
  3. It follows a snowflake schema in which each dimension has a single column surrogate key. The dimensions are classified into static (never changing dimensions), historical (to keep historical truth) and non-historical (no need of historical truth).
  4. The administrator is able to set, once and for all, the locking levels and the concurrent scheduling rules for queries and maintenance functions.
  5. The size is variable and can be set by the parameter **scale factor**.

- **Query and User Model assumptions**: the users and queries modeled in the benchmark:

  1. Address complex business problems.
  2. Use a variety of access patterns, query phrasing, operators and answer set constraints.
  3. Employ query parameters that change accross different query executions.

  Thus, TPC-DS uses a generalized query model. It takes into account different query classes:

  - Ad-hoc query workload: simulates an environment in which users connected to the system send individual queries that are not known in advance, so they cannot be optimized by design.
  - Queries in a reporting workload: represent the other side of the coin. The administrator can optimize the system to run some queries that are known prior to use.

  To achieve this separation, TPC-DS uses the catalog sales channel for the reporting queries, and the store and web channels are dedicated for the ad-hoc queries.

- **Data Maintenance assumptions**: the process of feeding data from the OLTP system to the analytical DSS system is of capital importance. This process usually involves three steps that are commonly known as ETL: Extraction, Transformation and Load. The ETL process is what is called Data Maintenance (DM) in TPC-DS, and the tasks performed are the following:

  1. Load the refresh data set, consisting of new, changed and deleted data in operational format.

2. Load refresh data into the warehouse applying data transformation. For example, data de-normalization or data cleaning.

3. Insert new fact records and delete obsolete fact records by date.

2.2.3 Logical Database design

We are not going to enter into much detail regarding the definition of the tables and the schemas[2], but it worth mentioning some general aspects in this respect.

The TPC-DS general schema models sales and returns of an organization by three main means: stores, catalogs and the internet. To do this, it has seven fact tables, a pair for product sales and returns for each mean and a fact table modeling inventory for the catalog and internet sales. There are, in addition, 17 dimension tables, regarding time, customers, locations,...

The standard also proposes some terms and requirements:

- **Base table**: fact and dimension tables. When we generate a flat file with data using dsdgen[3] it is called a **base table data**. A structure that contains a base table is a **base table data structure**.
- **Auxiliary Data Structure (ADS)**: any database structure that contains a copy of, reference to, or data computed from base table data. They can be **explicit** if created by means of a directive or **implicit** otherwise.
- **Horizontal partitioning** occurs when we divide a table or EADS into different files, disks or areas.
- **Vertical partitioning** occurs when we assign groups of columns to files, disks or areas, different from those storing the other columns.
- **Primary key**: one or more columns that uniquely identifies a row.
- **Foreign key**: one or more columns used to establish a link between data in two different tables.
- **Referential Integrity**: data property that asks that a foreign key in one table must correspond to a primary key in another one.
- **Data Access Transparency**: property of the system that removes from the query text any knowledge of the physical location and access mechanism of partitioned data.

2.2.4 Scaling and Database Population

The TPC-DS benchmark defines a set of **scale factors**, that correspond to the size of the raw data generated to later perform the analysis. The data is produced using the tool **dsdgen**.

The standard set defined in the benchmark is $\{1TB, 3TB, 10TB, 30TB, 100TB\}$, but due to storage, computing and Oracle XE restrictions[4], we opted for much smaller scale factors to perform our tests:

[2] Refer to [TPC21] for further details.
[3] The provided data generator. We will talk more about this function later.
[4] Oracle XE only allows for 12 GB of data.

$\{1GB, 3GB, 5GB, 7GB\}$[5].

This data needs to be loaded into the database, measuring how good the system does this job. This is called the **Load Test**, and must follow some guidelines:

- The data loading procedure has to be done automatically, not involving manual interaction.
- If the raw data is generated into flat files, then only the load time should be measured. Otherwise, if the data is streamlined directly from generation to the system, then the whole procedure needs to be measured.
- The measures to be taken are:
  - **Load Start Time**: timestamp taken right before the beginning of the loading data phase.
  - **Load End Time**: timestamp taken right after the end of the loading data phase.
  - **Load Time**: difference between the two previous measures.

The specific details for our Load Test implementation can be consulted in Section 3.5.

2.2.5 Query overview

The TPC-DS benchmark defines a set of 99 queries to be run against a populated database with several scale factors. The queries are defined by four components:

- **Business question**: real question that the query answers.
- **Functional query definition**: a template that implements the query and can be transformed into the different SQL dialects, e.g. Oracle SQL. Some values are left as variables, to be later substituted by a value.
- **Substitution parameters**: parameters specifying the values to substitute in each query.
- **Answer set**: a set of answers developed to validate the correctness of the system[6]. This set contains the 99 queries, ran against a 1GB database with specified parameters.

For example, *query1* has the following components:

- Business question: find customers who have returned items more than 20% more often than the average customer returns for a store in a given state for a given year.
- Functional query definition: see *query1.tpl*.
- Substitution parameters:
  - YEAR.01 = 2000
  - STATE.01 = TN

[5]It is not possible to use bigger scale factors, as this is the size of the raw data, which taking into account all necessary metadata and data structures add up to almost the 12 GB allowed by Oracle XE.

[6]We are not performing this validation, as all the parameters need to be manually replaced in each query, and it has also been reported that the results are not reliable (see https://github.com/gregrahn/tpcds-kit/issues/16.). Some efforts have been made to address this issue (see https://github.com/cwida/tpcds-result-reproduction). We have manually checked some queries, and they are correct. This suffices for the scope of the project.

  - AGG_FIELD.01 = SR_RETURN_AMT

- Answer set: see *query1.sol*.

The benchmark provides a tool that can be used to translate the query templates into functional queries in a given dialect and imputes the substitution parameters. This tool is called **dsqgen** and we used it to generate the queries in Oracle SQL dialect.

Once generated, the queries have to be run against the database, measuring the time that it takes for each query to complete, and the total time it takes to run all queries. This is called the **Power Test**, and it also has some guidelines to follow:

- The queries shall be executed in a single query stream and using a single session on the system under test, i.e., we need to be able to run all the queries with a single connection to the SUT.
- Only one query can be executed at a time, so it is not possible to allow parallel processing of the queries in the Power Test.
- The measures to be taken are:
  - **Power Test Start Time**: timestamp that must be taken before the beginning of the execution of the first query.
  - **Power Test End Time**: timestamp taken right after the end of the execution of the last query.
  - **Power Test Time**: difference between the two previous measures.

For further details on the Power Test refer to Section 3.6.

Related to this measure, there is also the **Throughput Test**, which consists of executing several times the set of 99 queries, concurrently with several users. This measures the ability of the system to serve multiuser functionalities and needs to follow some guidelines, too:

- A total of $S_q \geq 4$ users needs to be used for test.
- Each user has to execute a permutation of the 99 queries, generated at random.
- The measures to be taken are:
  - **Throughput Test Start Time**: timestamp taken before the first query of the first user starts.
  - **Throughput Test Start Time**: timestamp taken after the last query of the last user finishes.
  - **Throughput Test Time**: difference between the two previous measures.

For more details about the Throughput Test, see Section 3.7.

2.2.6 Data Maintenance

Data maintenance is another part of the benchmark that deals with data maintenance functions. That is, we want to see how the SUT performs when new data arrives to the database.

To achieve this purpose, new data is generated with the data generation tool, **dsdgen**, and different functions are applied to the database:

- **Fact insertion**: new facts are inserted into the database.
- **Fact deletion**: some facts are deleted from the database, using time filters.
- **Inventory data deletion**: some data is deleted from the inventory table, using time filters.

These aspects are measured in the **Data Maintenance Test**, which must follow:

- $S_q/2$[7] refresh runs need to be executed.
- Each refresh run uses its own refresh data set, generated with **dsdgen**.
- Refresh runs do not overlap.
- A failure in the physical system must be triggered during the Data Maintenance Test.
- The measures to be taken are:
  - $DS(i, s)$: timestamp taken before the function i of refresh run s starts.
  - $DE(i, s)$: timestamp taken after the function i of refresh run s ends.
  - $DI(i, s)$: difference between the two previous measures, i.e., the time it takes to execute function i in run s.
  - $DS(s)$: timestamp taken before run s starts.
  - $DE(s)$: timestamp taken after run s ends.
  - $DI(s)$: difference between the two previous measures, i.e., the time it takes to complete run s.

This test has not been implemented for our project.

[7] Remember S_q is the number of users used in the Throughput Test.

Chapter 3

Implementation Details

3.1 Setting up Oracle Database

Some members of our group used Windows OS and some Ubuntu OS so we had two types of setups for the project. The Ubuntu users set up a virtual machine using Oracle VM VirtualBox to run Oracle Linux 8.6. Windows users could directly install Oracle DB to their computers. All of the results in Chapter 4 were obtained using a computer running on Windows, with Intel Core i5-11400H as CPU and 16 GB of RAM.

All of us used Oracle Database Express Edition (XE) 21c as the tool. It is a free version of Oracle Database. However, it had limitations of up to 12 GB of user data, up to 2 GB of database RAM and up to 2 CPU threads [Dat22].

3.2 Generating and Loading the Data

Data Generation was done with the TPC-DS program *dsdgen*. Initially we loaded the data generated with the Integrated Development Environment(IDE) Oracle SQL Developer. However, the terminal-based bulk loader utility SQL*Loader turned out to be much faster [Smi18]. Also, it allowed us to develop a tool for making automatic tests, as the TPC-DS benchmark requires.

3.3 Generating queries

3.3.1 Adapting Templates to Oracle

Before generating queries, we had to fix syntax in some TPL files so that they can generate queries that can work in our Oracle Database (i.e. uses Oracle SQL dialect). In particular, we edited TPL files for queries 5, 12, 16, 20, 21, 30, 32, 37, 40, 58, 77, 80, 82, 83, 87, 92, 94, 95, 98.
Here are the specific changes we made:

- Changing `cast(as date)` function to `to_date()`.
- Using set operator `minus` instead of `except`.
- Fixing some variable names.

3.3.2 Generating queries from the templates

To generate the queries, we have developed a python script, *separate_queries.py*[1]. This script is basically used the previously generated queries using the Oracle templates in one file called *query_0.sql* to separate them into 99 different files, named *query_n.sql*, $n \in 1, ..., 99$. Note that we add in the generation step a clause that enables us to measure the time taken for the query to run, as well as a file to which redirect the output.

3.4 Running queries

For running queries, we used Python API (Application Programming Interface) *cx_Oracle* which enables access to Oracle Database through Python. We also used SQL*Plus, a command-line application to execute SQL files in the Oracle Database. The graphical user interface of SQL Developer was also helpful to quickly modify the queries and execute them.

3.5 Load Test

Our implementation for the Load Test can be accessed in our repo[2], and the logic behind it is pretty simple:

```
assumptions:
    - The schema has been loaded to the database.
    - sqlldr command is accessible.
    - The tpcds-ri.sql script has not been executed, and will be executed after the
    execution of this program.

algorithm:
    set timer start
    load data into database
    set timer end
```

Listing 3.1: Load Test pseudocode

The loading of the data is done concurrently, using several cores.

In an attempt to make the process automatic, we tried to develop a variant of this script which drops all tables from the database, loads the data and runs the tpcds-ri.sql. Nonetheless, we experienced some problems when using this program, as apparently some metadata of the data remains in the database after dropping the tables, and this somehow made some queries to fail[3], so we couldn't ensure the correct functioning of this program[4].

So, finally, we decided to go for consistency of results and we could not fully automatize the Load Test. The schema needs to be created prior to the execution of the script, and the referential integrity script must be run after the Load Test.

[1] See https://github.com/Lorenc1o/TPC-DS_Oracle/blob/main/scripts/utils/separate_queries.py.
[2] See https://github.com/Lorenc1o/TPC-DS_Oracle/blob/main/scripts/test/Load_test.py
[3] For completeness, we experienced these failures with query35.
[4] It would work for the first execution of the benchmark, but would fail in consecutive trials.

3.6 Power Test

The implementation for the Power Test can also be consulted in our repo[5]. Again, let's see how it works:

```
assumptions:
    - The Load Test has already been executed.
    - The tpcds-ri.sql has already been executed.
    - sqlplus command is accessible.
    - All the queries are located in a single script.

algorithm:
    set timer start
    run the full script with all the queries
    set timer end
```

Listing 3.2: Power Test pseudocode

Note that the queries should have been previously generated as we explained before.

This way, we get the time to execute all the queries with the external timer and the time to execute each query with the internal timer that each query has in it.

3.7 Throughput Test

The implementation of the throughput test is also available in our repo[6]. The algorithm is as follows:

```
assumptions:
    - The tpcds-ri.sql has already been executed.
    - sqlplus command is accessible.
    - All the queries are located in a single script, one for each thread accessing the
    database. Each of this scripts is called a stream.

algorithm:
    create 4 threads
    set timer start
    for each thread:
        run all the queries
    wait for the slowest thread
    set timer end
```

Listing 3.3: Throughput Test pseudocode

We have to do some observations now:

- We have modeled the multiuser scenario as a single user accessing concurrently to the database.
- We have chosen 4 users to access the database concurrently because it is the minimum value for S_q allowed in the TPC-DS specification, taking into account that Oracle XE only allows two cores functioning concurrently.

With these remarks, we want to transmit that this throughput test could be improved by setting up a server and configuring several users to be able to access it. Then, using this setup would provide more reliable results. Nonetheless, we have taken what we considered the best approach taking into account our circumstances, while maintaining the specifications of the benchmark as much as possible.

[5] See https://github.com/Lorenc1o/TPC-DS_Oracle/blob/main/scripts/test/Power_test.py.
[6] See https://github.com/Lorenc1o/TPC-DS_Oracle/blob/main/scripts/test/throughput_test.py.

Chapter 4

Results and discussion

As mentioned in section 3.1, our setup allowed to load only 12 gigabytes of data to the database. We figured out that scale 7 had a little less than 12 gigabytes when we loaded it to the database, so it had to be our maximum scale factor. We chose lower scale factors to be 1, 3 and 5, so our scale factors increase linearly.

4.1 Load Test Result

You can see the duration of loading the databases with the selected scale factors below.

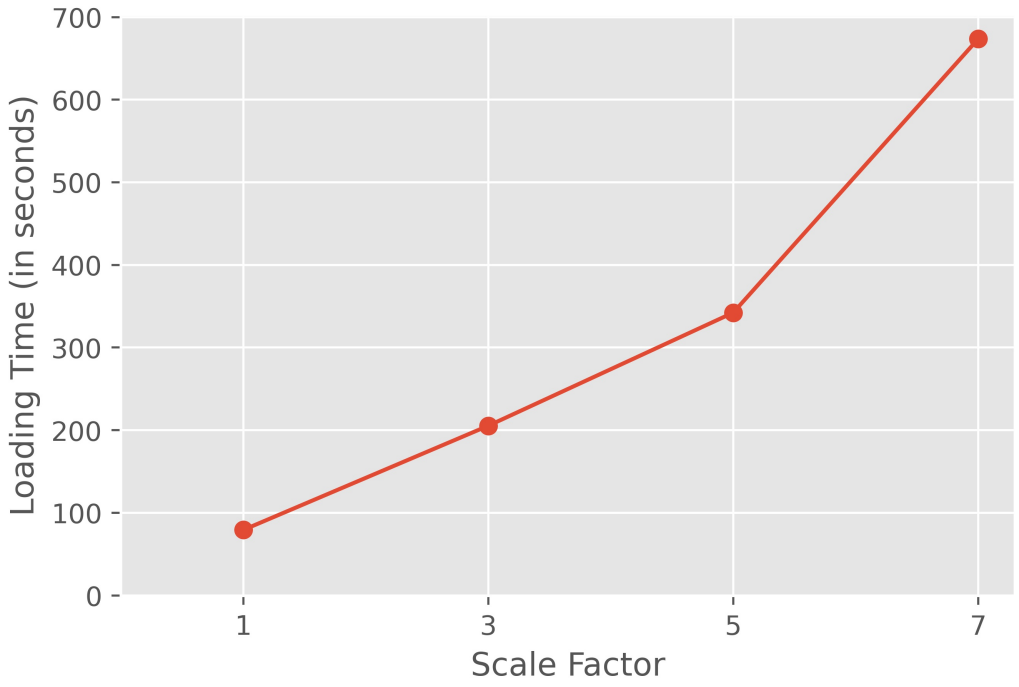

Figure 4.1: Load Time for the 4 scale factors.

It can be seen that the loading time increased with about the same rate for factors 1, 3 and 5 with the gradient being around 65 seconds per scale factor, but the gradient between 5 and 7 was much higher with 165 seconds per scale factor. This means that the loading time increases more than linearly and,

thus, the system does not scale optimally regarding loading time. Nonetheless, we have to take into account that the scale factors account for more data than the raw data, meaning that the increased time is not as pronounced as it seems. This effect is visible when we compare the previous plot with Figure 4.2, in which the loading times are plotted against the (estimated) full sizes for each scale factor. The times are increasing more similarly to the increase factor in the total amount of data.

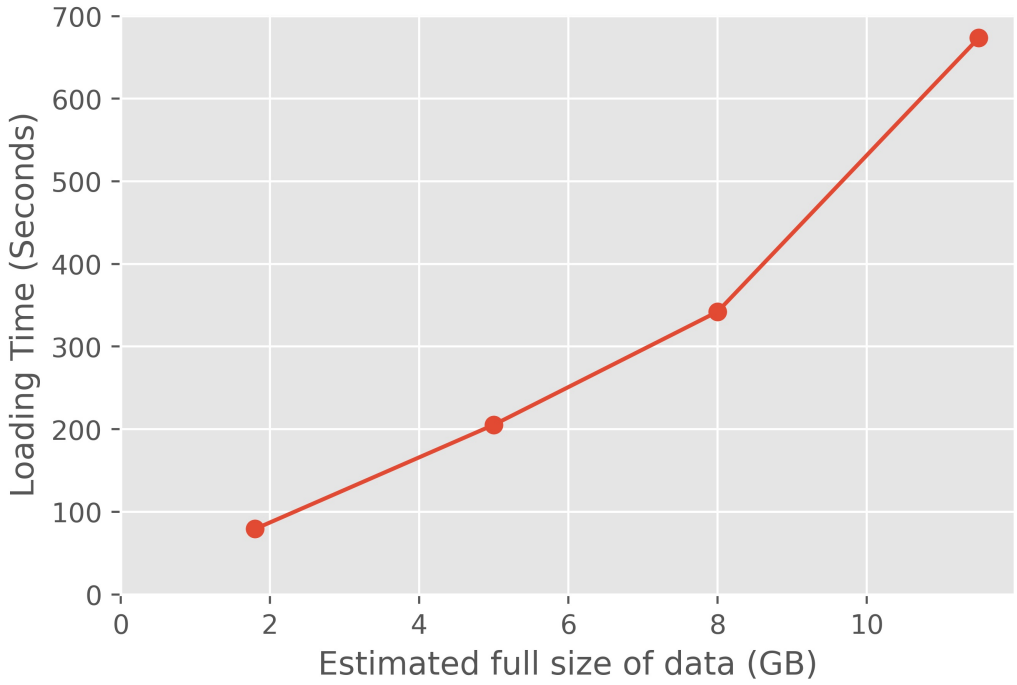

Figure 4.2: Load Time for the 4 scale factors (estimated full size).

4.2 Power Test Results

Now, let's see the results of the Power Test. First, we can see the Power Test Time against each scale factor in Figure 4.3(up) and against the estimated full size of the database for each scale factor in the lower part of the figure. As can be observed, the increase is really close to be linear, which means that the system is escalating almost optimally with respect to query processing.

Now, although both Load and Power tests present an almost linear increase behavior, it is interesting to compare them and check whether they have the same slope or not. For this, let's analyze Figure 4.4.

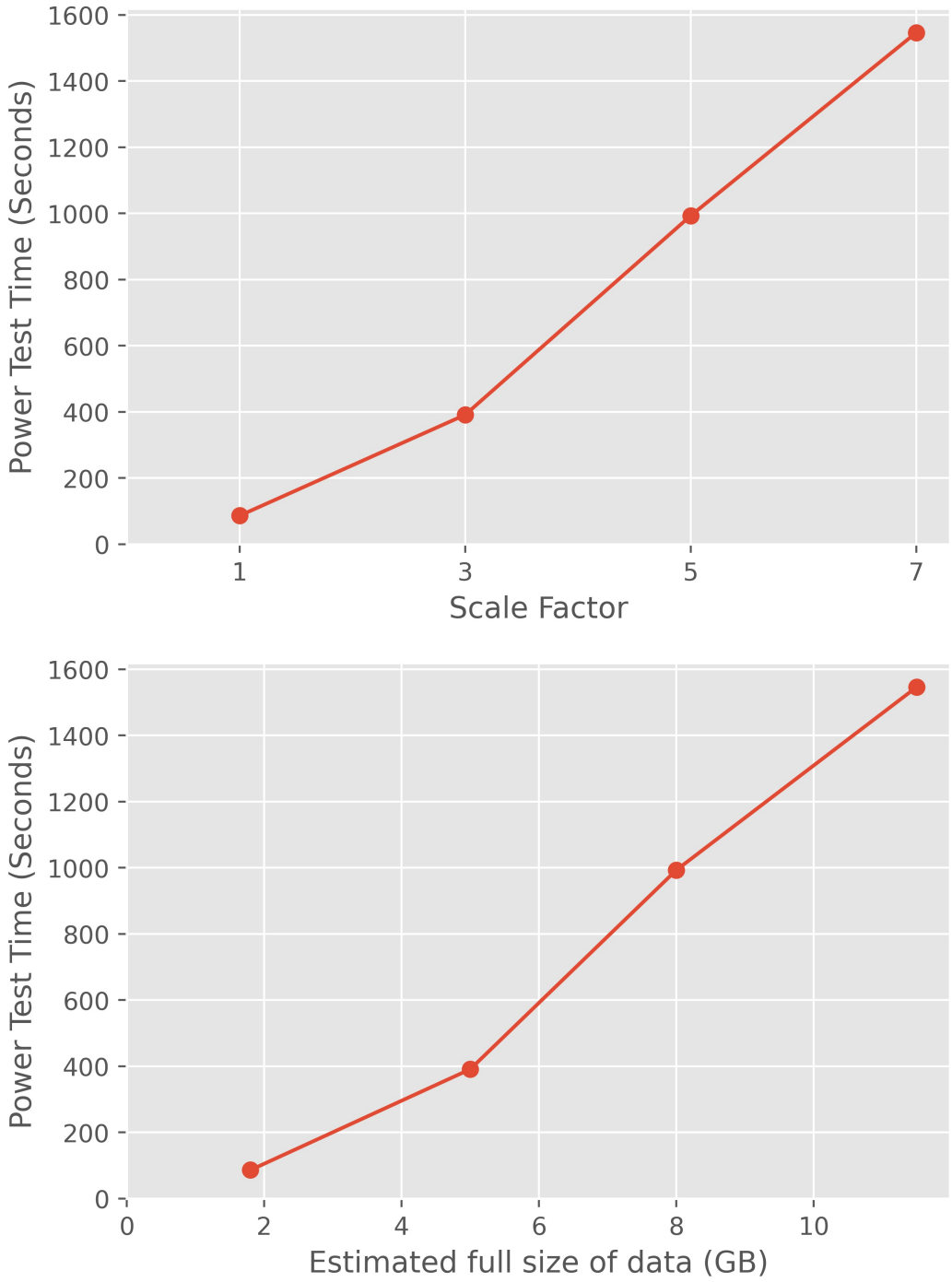

Figure 4.3: Power Test Time against the 4 scale factors (up) and against the estimated full size of the databases (down).

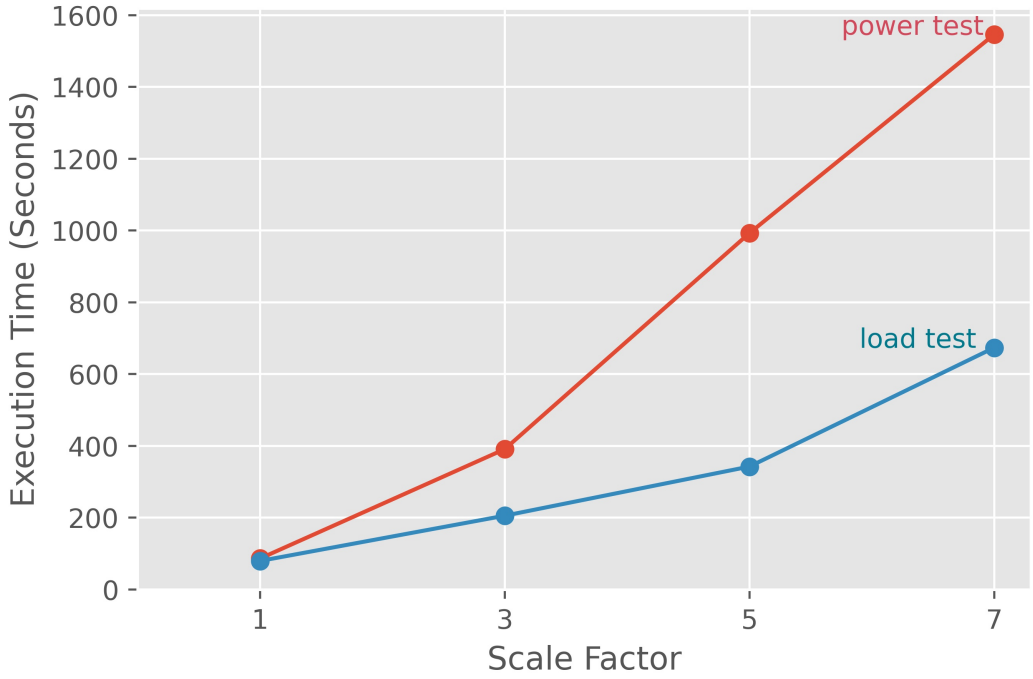

Figure 4.4: Load and Power test times plotted against the 4 scale factors.

We can clearly see how the power test presents a steeper increase, which means the slope of the linear increase that it follows is bigger than that of the load test. This fact is important, because, as shown, even though both test times increase close to linearly, we observe how they start from a very similar time for scale 1 to finish with the Power Test Time more than doubling the load test time: scaling factor[1] matters!

We are now going to continue analyzing the execution times for each query and each scale factor.

4.2.1 Scale 1

In Figure 4.5, it is shown how much time it took to run the 99 queries on scale factor 1. Query 69 is the clear outlier with around 155 seconds of execution time compared to the mean of 0.66 seconds for all the queries.

In Figure 4.6, we plotted the execution times of all the queries except for query 69 to see their values more clearly. Query 10 is another outlier with around 24 seconds.

4.2.2 Scale 3

Figure 4.5 shows the execution times of the queries on scale factor 3. In this case, the outliers are Queries 14, 17, 23, 24 and 95. Note that for this and the bigger scales query 69 was run using its optimized version[2], as the original version was taking several orders of magnitude the time it takes for the rest of the queries. Also, query 35 was measured using its optimized version, because its execution time increased several times, leading to similar measures to those obtained for query 69, which opaque the other measures.

[1] Here we are not referring to the scale factor of the database, but to the parameters we encounter in the linear model.
[2] The optimizations made are analyzed and explained later. See Section 4.4.

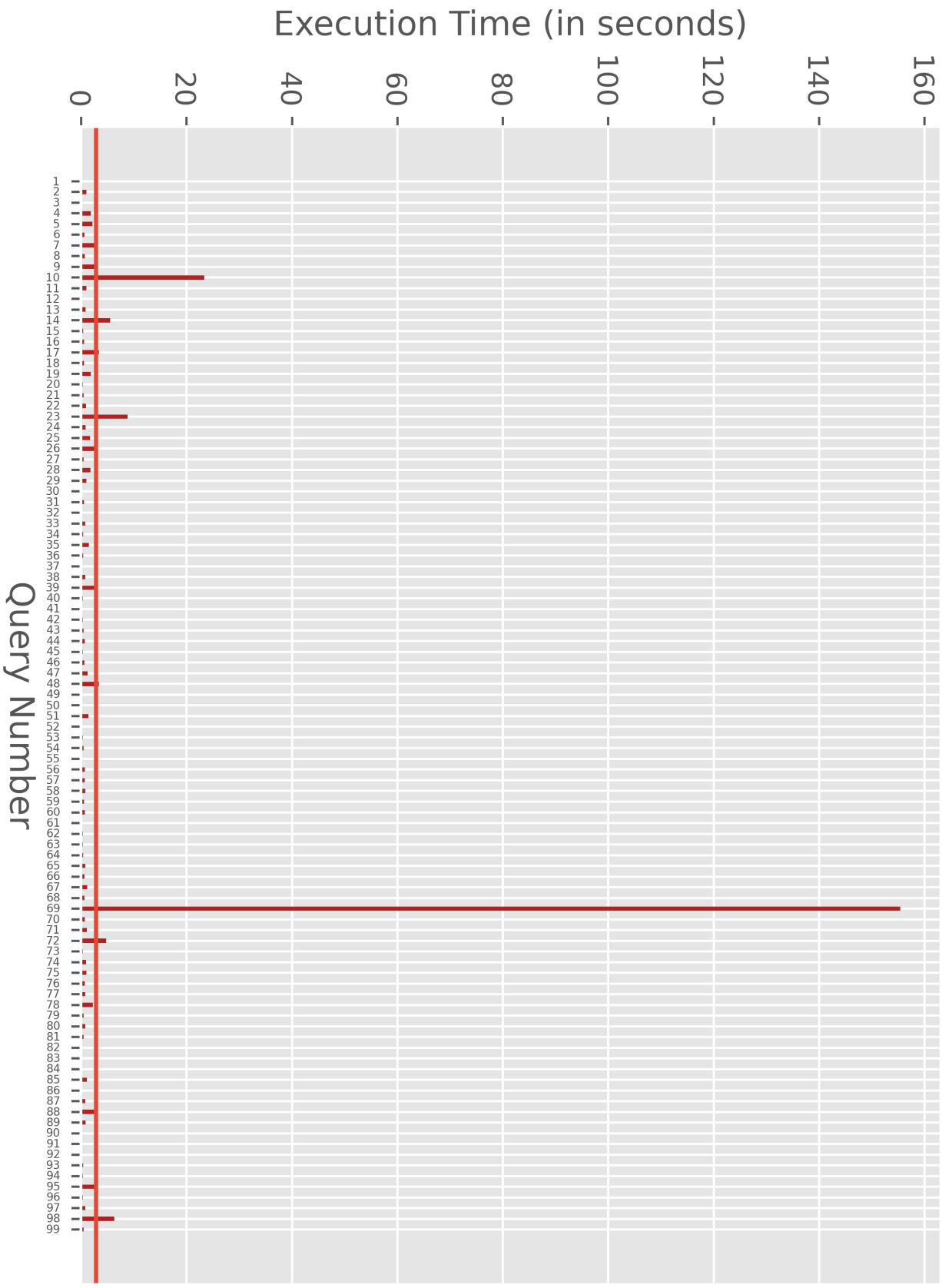

Figure 4.5: Execution Time of the original 99 queries for scale 1. The light red vertical line is the average execution time.

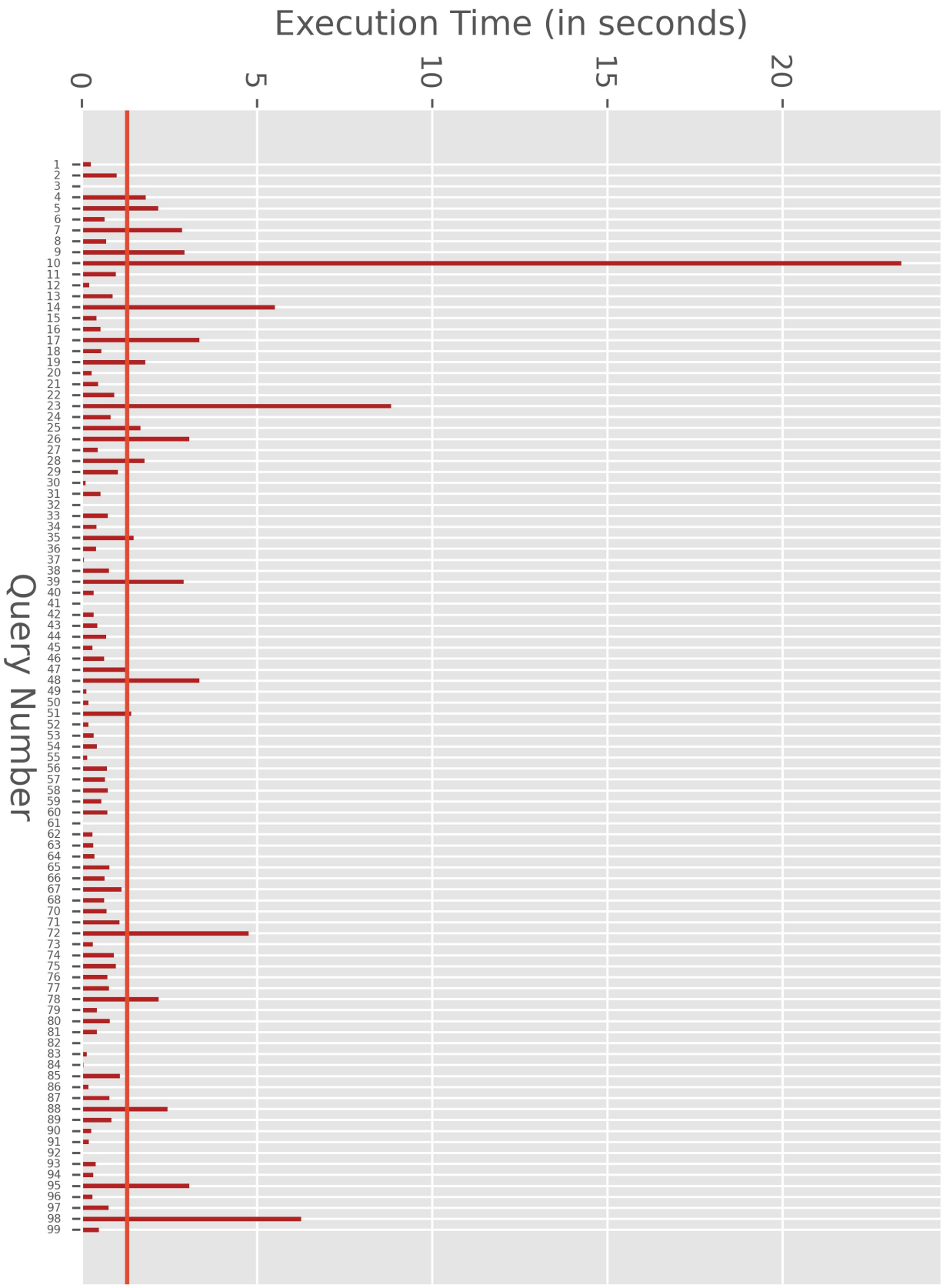

Figure 4.6: Execution Time of all the queries except for #69 for scale 1. The light red vertical line is the average execution time.

4.2.3 Scale 5

Execution times of the queries on scale factor 5 can be seen on Figure 4.8. The outliers are Queries 17, 24 and 95.

4.2.4 Scale 7

Figure 4.9 shows the results for scale 7. The outliers are Queries 24 and 25.

4.2.5 Visualizing scale factor differences

To better visualize how the execution times evolve when increasing the scale factor, we have also plotted the four time measurements against the scale factor for each of the queries. We don't believe it is very interesting to show all of them, so we have selected a subset of these that are representative of what can be found in this data.

We are going to start with those queries that represent the most usual behavior of the executing times, and which is also the expected behavior, i.e., an increase in time which should be, at least linear. This kind of queries is depicted in Figure 4.10. As we can see, regardless of the specific values of time, we observe an increasing trend in execution time against the scale factors. Also, the increase seems linear, which means the system is escalating good regarding query execution time. This is observed in most of the queries. More precisely, this is the behavior in approximately 90% of the queries.

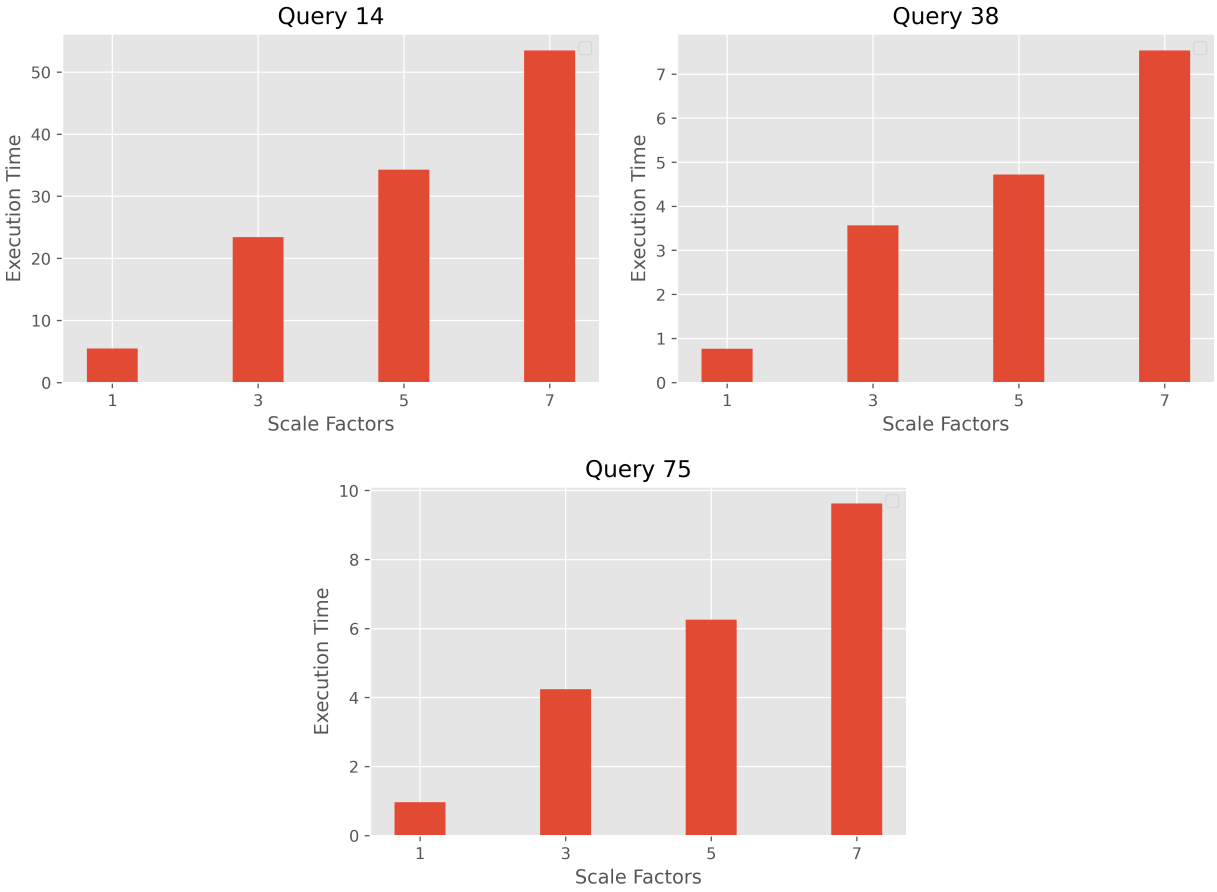

Figure 4.10: The usual shape of the execution time plots.

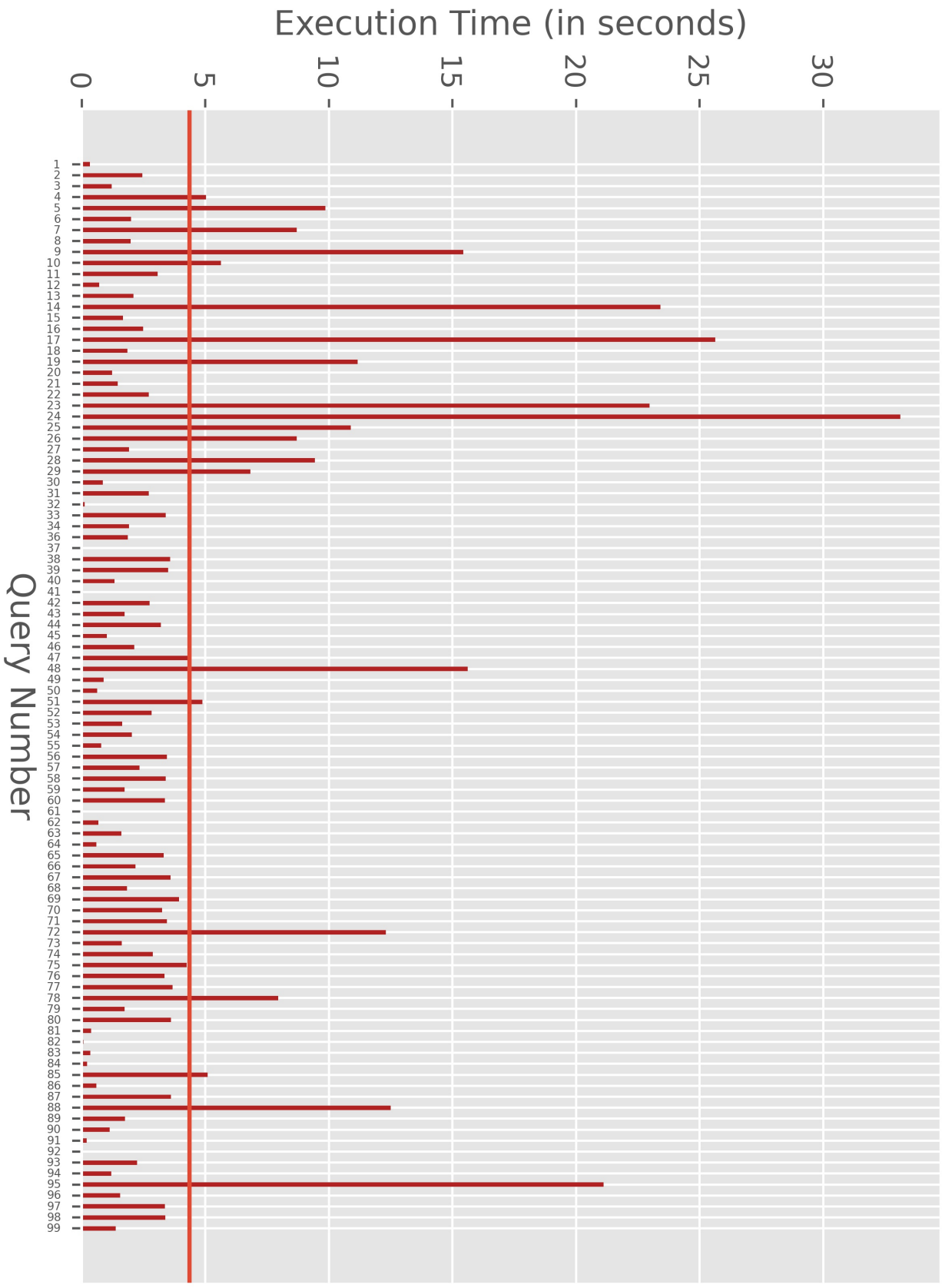

Figure 4.7: Execution Time of all the queries for scale 3. The light red vertical line is the average execution time.

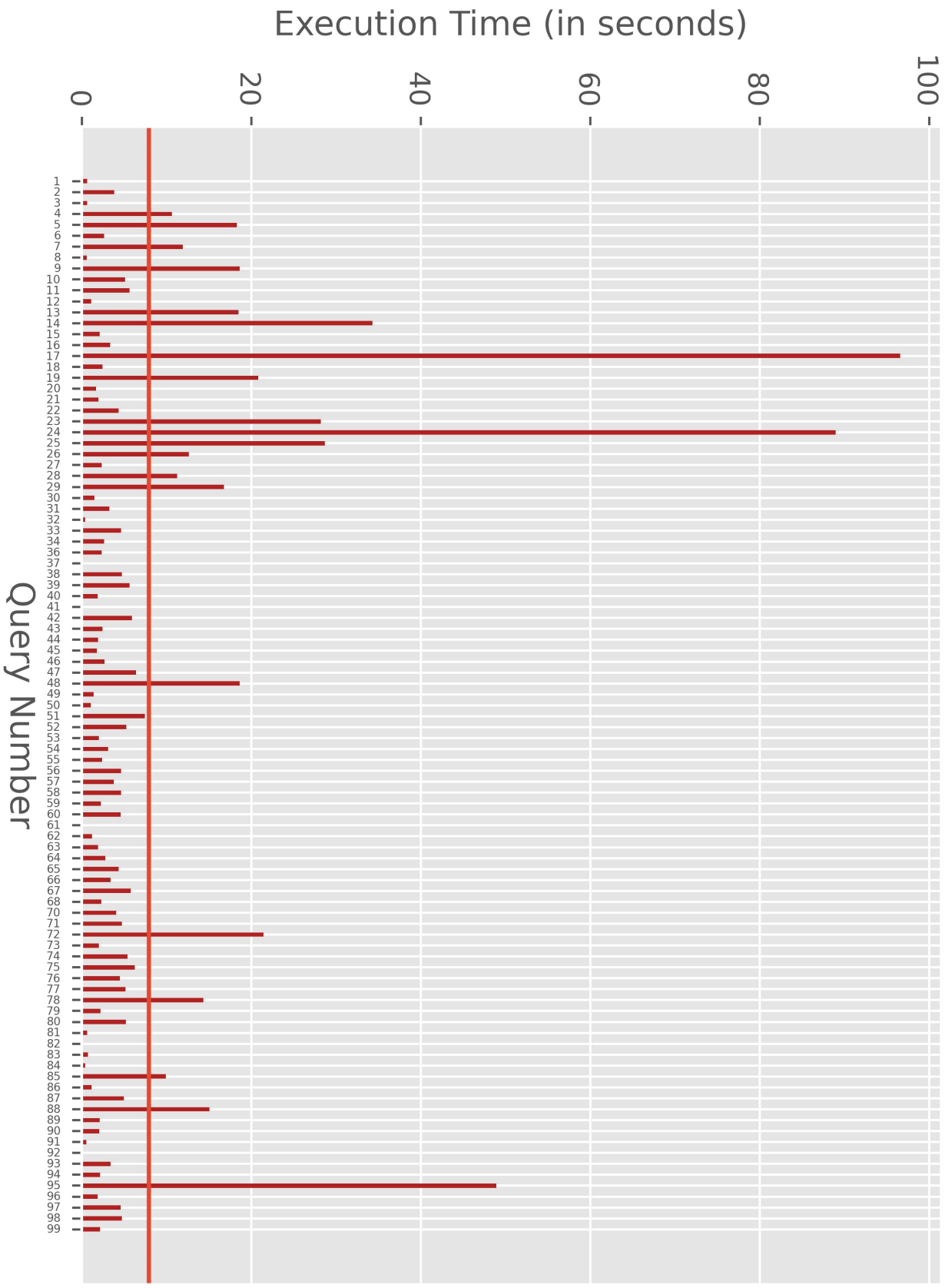

Figure 4.8: Execution Time of all the queries for scale 5. The light red vertical line is the average execution time.

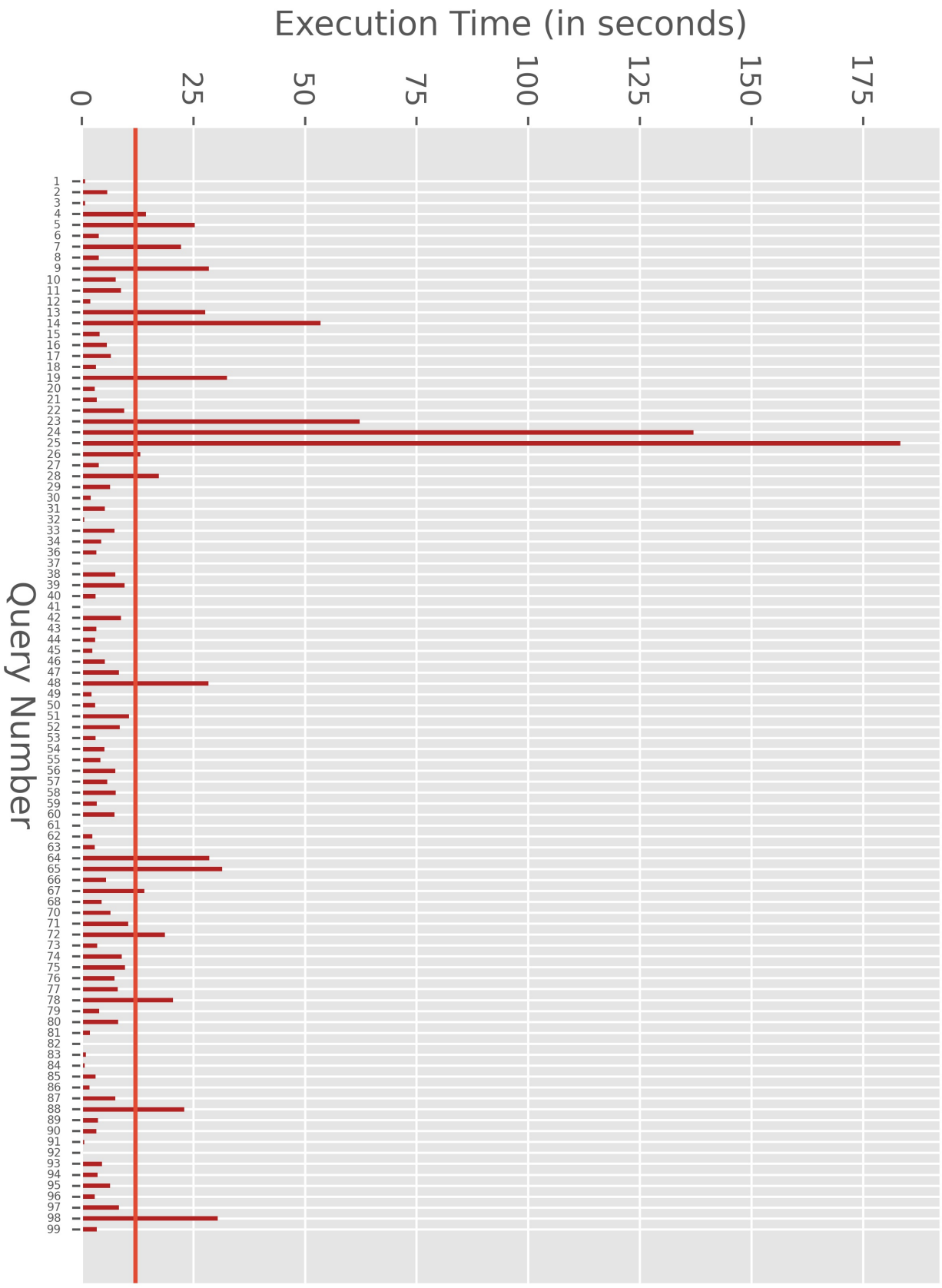

Figure 4.9: Execution Time of all the queries for scale 7. The light red vertical line is the average execution time.

Now, we are going to analyze some queries that present a different behavior from this vast majority. We find two kinds of queries.

The first kind are those whose time increase is bigger than linear, such as queries 65 and 92, depicted in Figure 4.11. This behavior is an indicative that the system is not optimally escalating for queries like these.

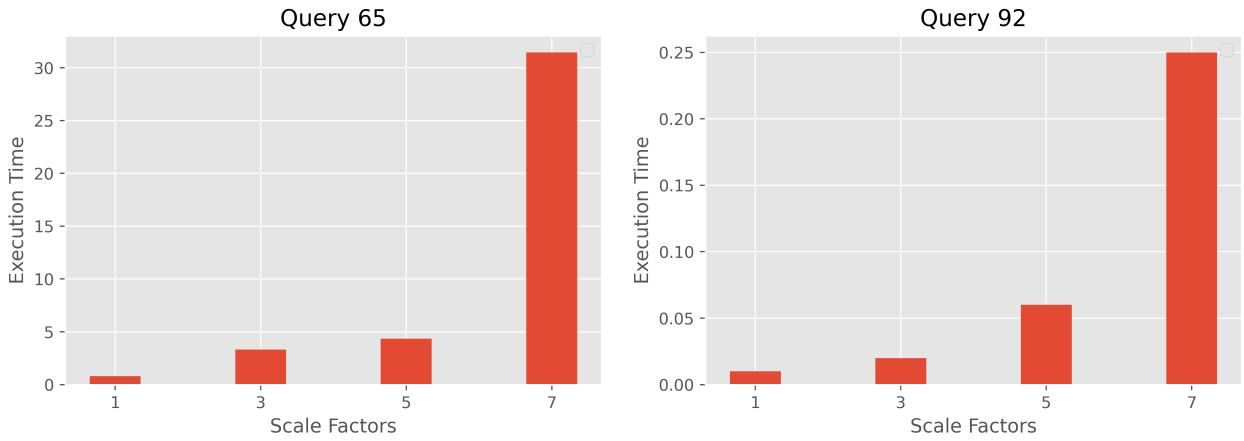

Figure 4.11: Two queries whose execution time increased more than linearly.

Finally, we encountered some queries with a totally different behavior, which is somewhat erratic, presenting higher times for smaller scale factors. These kind of queries are represented in Figure 4.12.

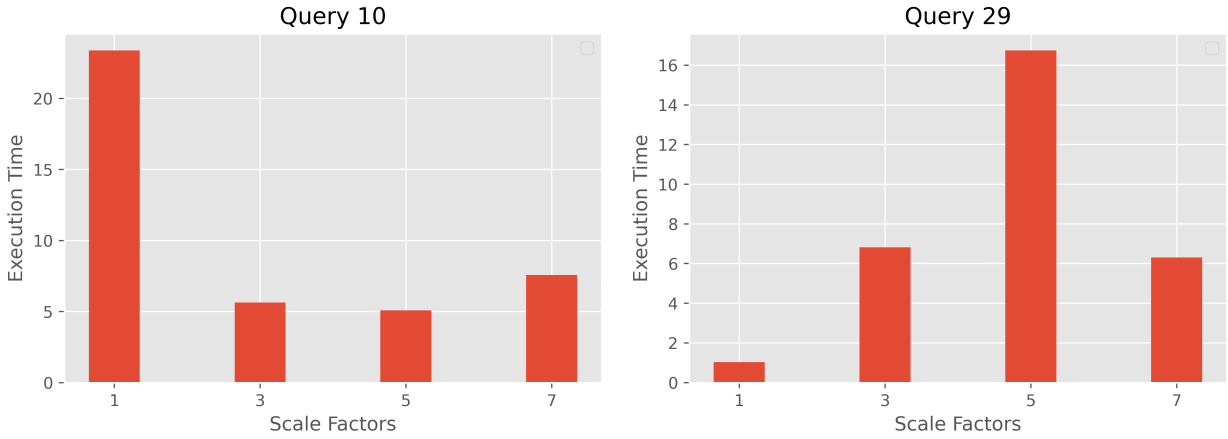

Figure 4.12: Two queries with erratic behavior.

We have not been able to determine the cause of this erratic behavior, but we have thought on some possibilities:

- As all sizes are not so big, the tables involved in these queries might fit in just one disk page, and so they do for the bigger scale factors. But, if by chance the tables are split into different pages for one of the scale factors, it could take longer for it to fetch the data and, thus, to get the final result.
- Another possibility is that the CPU or the RAM were occupied with other processes when we

ran the smaller scale factors.

It is not easy to determine if one of this possibilities causes this issue, and further efforts should be done to address this question.

4.3 Throughput Test Result

Figure 4.13 below shows the results for our implementation of the Throughput Test. We observe, again, an increasing trend which is similar to be linear. Nonetheless, it is also observable that this time we are facing much steeper results than for the other tests. This might indicate that the system is not very good in multi-user scenarios, but that it escalates well, taking into account this poor behavior.

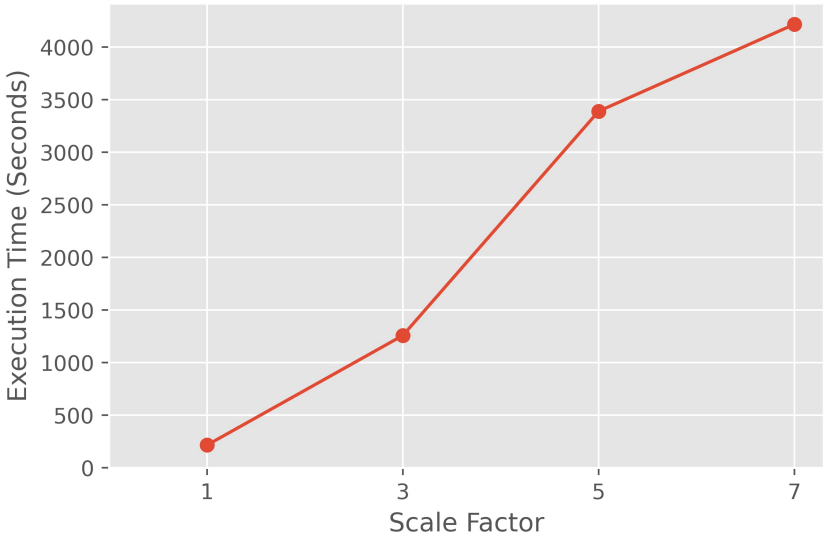

Figure 4.13: Throughput Test Time for the 4 scale factors

In Figure 4.14, we can compare the execution times for the Throughput Test with those of Power and Load Tests. Like with the case of Power Test, the scale factor 1 of Throughput Test has about twice the execution time as Power Test, as should be expected because we ran 4 streams and the system allows for 2 cores, which means that the best case scenario is doubling the Power Test time. The execution times for greater scale factors also increase very similarly to those of Power Test just by a different factor, as can be seen by the dotted lines of $2 \times$ *power test* and $4 \times$ *power test*. As can be observed, this increase factor is bigger than twice the one for the power test (which would be the best case) but smaller than 4 times the one for the power test (which would be the same as executing the 4 streams sequentially).
From these results, we might assume that our setup is worse at scaling up for Throughput Test than the other Tests, although the system is able to outperform a sequential run, it is clear that the multi-user behavior is far from optimal.

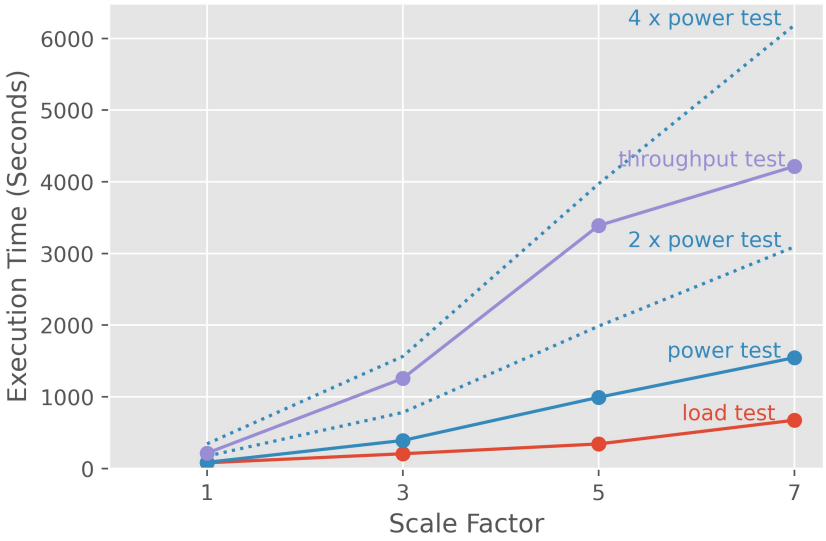

Figure 4.14: Throughput, Power and Load Tests Time for the 4 scale factors

4.4 Optimizing Queries

As can be seen in section 4.2, queries 10 and 69 were taking considerably more time than others. Therefore we decided to optimize them. We have been able to improve their execution time, and also those of other queries that did not take as long as these, but could be improved using similar ideas, namely 9, 88 and 35.

We will analyze them individually, in descending order of their initial execution time.

4.4.1 Query 69

This query answers the next business question: *'Count the customers with the same gender, marital status, education status, purchase estimate and credit rating who live in certain states and who have purchased from stores but neither from the catalog nor from the web during a two month time period of a given year'.*

The original query is as in Listing 4.1.

```
select * from
(
    select
      cd_gender, cd_marital_status, cd_education_status, count(*) cnt1,
    cd_purchase_estimate, count(*) cnt2, cd_credit_rating, count(*) cnt3
    from
      customer c,customer_address ca,customer_demographics
    where
      c.c_current_addr_sk = ca.ca_address_sk and
      ca_state in ('CO','IL','MN') and
      cd_demo_sk = c.c_current_cdemo_sk and
      exists (select *
              from store_sales,date_dim
              where c.c_customer_sk = ss_customer_sk and
                    ss_sold_date_sk = d_date_sk and
                    d_year = 1999 and
                    d_moy between 1 and 1+2) and
```

```
17      (not exists (select *
18                from web_sales,date_dim
19                where c.c_customer_sk = ws_bill_customer_sk and
20                      ws_sold_date_sk = d_date_sk and
21                      d_year = 1999 and
22                      d_moy between 1 and 1+2)) and
23      (not exists (select *
24                from catalog_sales,date_dim
25                where c.c_customer_sk = cs_ship_customer_sk and
26                      cs_sold_date_sk = d_date_sk and
27                      d_year = 1999 and
28                      d_moy between 1 and 1+2)
29    group by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
      cd_credit_rating
30    order by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
      cd_credit_rating
31  ) where rownum <= 100;
```

Listing 4.1: query69.sql

It can be basically decomposed as follows:

```
1  SELECT attributes
2  FROM
3      A JOIN B JOIN C
4  WHERE
5      easy_conds
6      AND
7      EXISTS (query involving D x date)
8      AND
9      NOT EXISTS (query involving E x date)
10     AND
11     NOT EXISTS (query involving F x date)
12 GROUP BY some attributes
13 ORDER BY some attributes
```

Listing 4.2: query69 structure

The problem in this query is that, for each row of A $\bowtie$ B $\bowtie$ C we have to compute three temporal tables that involve the date table. This can be a really long process if the optimizer treats the query as is.

As an example, suppose A,B,C have N rows each, date has M rows and D,E,F have K rows. Then, we have to traverse approximately $N^3 \cdot (M \cdot K)$ rows. It is obvious that this is a large number for not so large values of N, M, K, and M, K are likely to be large because they are a date dimensional table and sales fact tables. Thus, we can expect poor measures from this query.

The idea to amend this problem is to split the query in two parts:

1. In the first part, we obtain all customers that have purchased in a store in the given period.
2. In the second part, we obtain all customers that have not purchased neither from the catalog nor from the web in the given period.

After we have this, we can simply intersect the results of both selections, and we will get the desired answer. An equivalent way to do this is to use the difference between the customers that have

purchased in a store and those who have purchased either at the web or the catalog[3]. After the intersection/difference is performed, we aggregate the results.

If we translate this into relational algebra, we are saying that:

$$\sigma_{p_1 \ AND \ p_2}(A) \equiv \sigma_{p_1}(A) \wedge \sigma_{p_2}(A) \equiv \sigma_{p_1}(A) \setminus \sigma_{NOT \ p_2}(A).$$

Proof. If $r \in \sigma_{p_1 \ AND \ p_2}(A)$, it means that the record r in A fulfills both conditions p_1 and p_2, thus, we have $r \in \sigma_{p_1}(A)$ and $r \in \sigma_{p_2}(A)$, which means $r \in \sigma_{p_1}(A) \wedge \sigma_{p_2}(A)$.

Conversely, if $r \in \sigma_{p_1}(A) \wedge \sigma_{p_2}(A)$, then $r \in \sigma_{p_1}(A)$ and $r \in \sigma_{p_2}(A)$, which means that r fulfills both conditions, and $r \in \sigma_{p_1 \ AND \ p_2}(A)$.

The second equivalence is proven by De Morgan laws. □

This way we are sure of the validity of the alternative, and we developed the alternative query that can be read in Listings 4.3 and 4.4. In the first listing we show our first improved version, using the intersection, in the second one we show the approach that uses the difference.

```
select * from (
    select cd_gender, cd_marital_status, cd_education_status, count(*) cnt1,
    cd_purchase_estimate, count(*) cnt2, cd_credit_rating, count(*) cnt3 from (
        (select  c.c_customer_sk,
             cd_gender,
             cd_marital_status,
             cd_education_status,
             cd_purchase_estimate,
             cd_credit_rating
         from customer c,customer_address ca,customer_demographics cd
         where
         c.c_current_addr_sk = ca.ca_address_sk and
         ca_state in ('KY','GA','NM') and
         cd.cd_demo_sk = c.c_current_cdemo_sk and
           exists (select *
             from store_sales,date_dim
             where c.c_customer_sk = ss_customer_sk and
                ss_sold_date_sk = d_date_sk and
                d_year = 2001 and
                d_moy between 4 and 4+2
          ))
          INTERSECT
          (
          select
             c.c_customer_sk, cd_gender, cd_marital_status, cd_education_status,
    cd_purchase_estimate, cd_credit_rating
             from
             customer c,customer_address ca,customer_demographics cd
             where
             c.c_current_addr_sk = ca.ca_address_sk and
             ca_state in ('KY','GA','NM') and
             cd.cd_demo_sk = c.c_current_cdemo_sk and
          (not exists (select *
              from web_sales,date_dim
              where c.c_customer_sk = ws_bill_customer_sk and
                 ws_sold_date_sk = d_date_sk and
                 d_year = 2001 and
```

[3]It is proven by De Morgan laws.

```
                d_moy between 4 and 4+2)) and
          (not exists (select *
              from catalog_sales,date_dim
              where c.c_customer_sk = cs_ship_customer_sk and
                 cs_sold_date_sk = d_date_sk and
                 d_year = 2001 and
                 d_moy between 4 and 4+2)))
     group by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
    cd_credit_rating
     order by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
    cd_credit_rating
 ) where rownum <= 100;
```

Listing 4.3: Optimized query69 - Intersection version.

```
select * from (
    select
      cd_gender, cd_marital_status, cd_education_status, count(*) cnt1,
    cd_purchase_estimate, count(*) cnt2, cd_credit_rating, count(*) cnt3
     from
      customer c,customer_address ca,customer_demographics,
       ((select ss_customer_sk, customer_sk
              from store_sales,date_dim
              where ss_sold_date_sk = d_date_sk and
                    d_year = 1999 and
                    d_moy between 1 and 1+2)
        minus
        (select ws_bill_customer_sk customer_sk
                from web_sales,date_dim
                where ws_sold_date_sk = d_date_sk and
                      d_year = 1999 and
                      d_moy between 1 and 1+2)
        minus
        (select cs_ship_customer_sk customer_sk
                from catalog_sales,date_dim
                where cs_sold_date_sk = d_date_sk and
                      d_year = 1999 and
                      d_moy between 1 and 1+2))
     where
      c.c_current_addr_sk = ca.ca_address_sk and
      ca_state in ('CO','IL','MN') and
      cd_demo_sk = c.c_current_cdemo_sk and
      c.c_customer_sk = customer_sk
     group by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
    cd_credit_rating
     order by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
    cd_credit_rating
  ) where rownum <= 100;
```

Listing 4.4: Optimized query69 - Difference version.

In Figure 4.15, we can observe the execution times of query 69 for each of the scale factors. As we can see, the measured time before the optimization has only been taken into account for the scale factor 1. The reason for this is simple: it took too long to execute this query, several orders of magnitude more time than the rest of the queries. As can be seen in the plot, the optimization worked perfectly, making the optimized version to last a fraction of its original time. The intersection and difference versions of the optimized query 69 had roughly the same execution times in all scale factors.

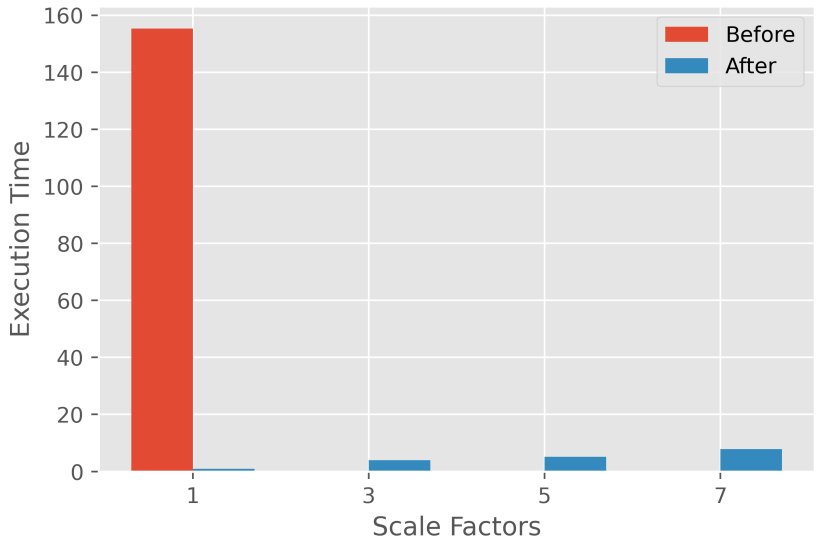

Figure 4.15: The execution time of Query 69 before and after the optimizations.

4.4.2 Query 10

This query answers the next business question: *'Count the customers with the same gender, marital status, education status, purchase estimate, credit rating, dependent count, employed dependent count and college dependent count who live in certain counties and who have purchased from both stores and another sales channel during a three month time period of a given year'.*

The original query is as in Listing 4.5.

```
select * from (
    select
      cd_gender, cd_marital_status, cd_education_status, count(*) cnt1,
    cd_purchase_estimate, count(*) cnt2, cd_credit_rating, count(*) cnt3, cd_dep_count,
    count(*) cnt4, cd_dep_employed_count, count(*) cnt5, cd_dep_college_count, count(*)
    cnt6
     from
      customer c,customer_address ca,customer_demographics
     where
      c.c_current_addr_sk = ca.ca_address_sk and ca_county in ('Walker County','Richland
    County','Gaines County','Douglas County','Dona Ana County') and
      cd_demo_sk = c.c_current_cdemo_sk and
      exists (select * from store_sales,date_dim
              where c.c_customer_sk = ss_customer_sk and
                    ss_sold_date_sk = d_date_sk and
                    d_year = 2002 and
                    d_moy between 4 and 4+3) and
       (exists (select * from web_sales,date_dim
                where c.c_customer_sk = ws_bill_customer_sk and
                      ws_sold_date_sk = d_date_sk and
                      d_year = 2002 and
                      d_moy between 4 ANd 4+3) or
        exists (select *  from catalog_sales,date_dim
                where c.c_customer_sk = cs_ship_customer_sk and
                      cs_sold_date_sk = d_date_sk and
                      d_year = 2002 and
                      d_moy between 4 and 4+3))
     group by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
```

```
    cd_credit_rating, cd_dep_count, cd_dep_employed_count, cd_dep_college_count
25   order by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
    cd_credit_rating, cd_dep_count, cd_dep_employed_count, cd_dep_college_count
26   ) where rownum <= 100;
```

Listing 4.5: query10.sql

The structure is really similar to that of query 69, so we propose a similar solution, with the same ideas involved, but using the union of tables to replace an *or* clause. The proposed alternative is shown in Listing 4.6.

```
1  select * from (
2      select
3        cd_gender, cd_marital_status, cd_education_status, count(*) cnt1,
     cd_purchase_estimate, count(*) cnt2, cd_credit_rating, count(*) cnt3, cd_dep_count,
     count(*) cnt4, cd_dep_employed_count, count(*) cnt5, cd_dep_college_count, count(*)
     cnt6
4      from
5        customer c,customer_address ca,customer_demographics,
6        ((select ss_customer_sk customer_sk
7                from store_sales,date_dim
8                where ss_sold_date_sk = d_date_sk and
9                      d_year = 2002 and
10                     d_moy between 4 and 4+3)
11         intersect
12        ((select ws_bill_customer_sk customer_sk
13                from web_sales,date_dim
14                where ws_sold_date_sk = d_date_sk and
15                      d_year = 2002 and
16                      d_moy between 4 ANd 4+3)
17         union
18         (select cs_ship_customer_sk customer_sk
19                from catalog_sales,date_dim
20                where cs_sold_date_sk = d_date_sk and
21                      d_year = 2002 and
22                      d_moy between 4 and 4+3)))
23     where
24       c.c_current_addr_sk = ca.ca_address_sk and
25       ca_county in ('Walker County','Richland County','Gaines County','Douglas County','
     Dona Ana County') and
26       cd_demo_sk = c.c_current_cdemo_sk and
27       c.c_customer_sk = customer_sk
28     group by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
     cd_credit_rating, cd_dep_count, cd_dep_employed_count, cd_dep_college_count
29     order by cd_gender, cd_marital_status, cd_education_status, cd_purchase_estimate,
     cd_credit_rating, cd_dep_count, cd_dep_employed_count, cd_dep_college_count
30 ) where rownum <= 100;
```

Listing 4.6: Optimized query10.

The timings for this query are shown in Figure 4.16. We observe a general improvement in this query, specially taking into account the time it took for the smallest scale factor. Nonetheless, the erratic behavior of this query makes comparisons difficult.

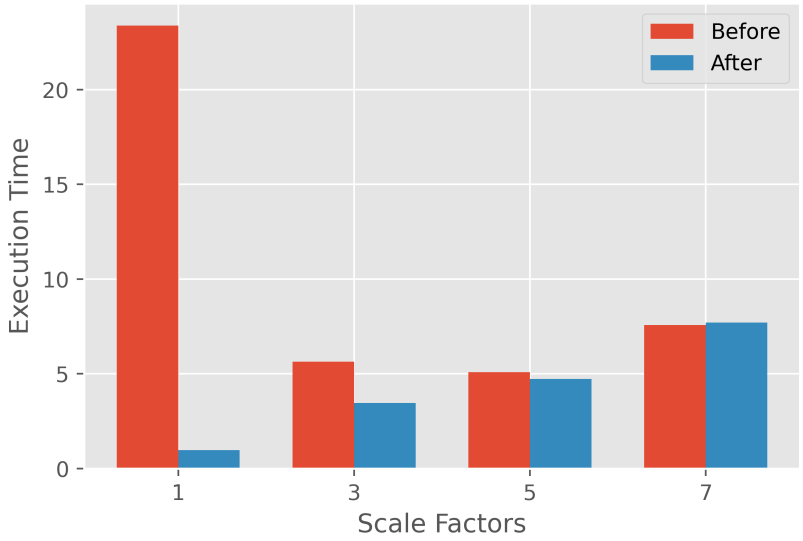

Figure 4.16: The execution time of Query 10 before and after the optimizations.

4.4.3 Query 9

In this case, the business question is: *'Categorize store sales transactions into 5 buckets according to the number of items sold. Each bucket contains the average discount amount, sales price, list price, tax, net paid, paid price including tax, or net profit..'.*

The original query is shown in Listing 4.7.

```
select case when (select count(*)
                  from store_sales
                  where ss_quantity between 1 and 20) > 25437
            then (select avg(ss_ext_discount_amt)
                  from store_sales
                  where ss_quantity between 1 and 20)
            else (select avg(ss_net_profit)
                  from store_sales
                  where ss_quantity between 1 and 20) end bucket1 ,
       case when (select count(*)
                  from store_sales
                  where ss_quantity between 21 and 40) > 22746
            then (select avg(ss_ext_discount_amt)
                  from store_sales
                  where ss_quantity between 21 and 40)
            else (select avg(ss_net_profit)
                  from store_sales
                  where ss_quantity between 21 and 40) end bucket2,
       case when (select count(*)
                  from store_sales
                  where ss_quantity between 41 and 60) > 9387
            then (select avg(ss_ext_discount_amt)
                  from store_sales
                  where ss_quantity between 41 and 60)
            else (select avg(ss_net_profit)
                  from store_sales
                  where ss_quantity between 41 and 60) end bucket3,
       case when (select count(*)
                  from store_sales
```

```
                  where ss_quantity between 61 and 80) > 10098
            then (select avg(ss_ext_discount_amt)
                  from store_sales
                  where ss_quantity between 61 and 80)
            else (select avg(ss_net_profit)
                  from store_sales
                  where ss_quantity between 61 and 80) end bucket4,
       case when (select count(*)
                  from store_sales
                  where ss_quantity between 81 and 100) > 18213
            then (select avg(ss_ext_discount_amt)
                  from store_sales
                  where ss_quantity between 81 and 100)
            else (select avg(ss_net_profit)
                  from store_sales
                  where ss_quantity between 81 and 100) end bucket5
from reason
where r_reason_sk = 1;
```

Listing 4.7: query9.sql

As we can see, for each row, we are checking some exclusive conditions and classifying it in a bucket depending on the outcome of a nested query and getting an attribute. But another attribute is asked for the rest of the buckets. Thus, the checking condition is made 5 times for each record. The idea is to classify the records into the different buckets prior to the query, so we don't have to compute the nested query so many times.

The resulting query is shown in Listing 4.8.

```
with ss_bucket_1 as
 (select *
 from store_sales
 where ss_quantity between 1 and 20),
ss_bucket_2 as
 (select *
 from store_sales
 where ss_quantity between 21 and 40),
ss_bucket_3 as
 (select *
 from store_sales
 where ss_quantity between 41 and 60),
ss_bucket_4 as
 (select *
 from store_sales
 where ss_quantity between 61 and 80),
ss_bucket_5 as
 (select *
 from store_sales
 where ss_quantity between 81 and 100)
select case when (select count(*) from ss_bucket_1) > 25437
            then (select avg(ss_ext_discount_amt) from ss_bucket_1)
            else (select avg(ss_net_profit) from ss_bucket_1) end bucket1,
       case when (select count(*) from ss_bucket_2) > 22746
            then (select avg(ss_ext_discount_amt) from ss_bucket_2)
            else (select avg(ss_net_profit) from ss_bucket_2) end bucket2,
       case when (select count(*) from ss_bucket_3) > 9387
            then (select avg(ss_ext_discount_amt) from ss_bucket_3)
            else (select avg(ss_net_profit) from ss_bucket_3) end bucket3,
       case when (select count(*) from ss_bucket_4) > 10098
            then (select avg(ss_ext_discount_amt) from ss_bucket_4)
```

```
            else (select avg(ss_net_profit) from ss_bucket_4) end bucket4,
        case when (select count(*) from ss_bucket_5) > 18213
            then (select avg(ss_ext_discount_amt) from ss_bucket_5)
            else (select avg(ss_net_profit) from ss_bucket_5) end bucket5
from reason
where r_reason_sk = 1;
```

Listing 4.8: Optimized query9.

The execution times before and after the aforementioned optimizations can be observed in Figure 4.17. As it can be seen, our editions of the query led to around 25-30% decrease in the execution time for all 4 scale factors.

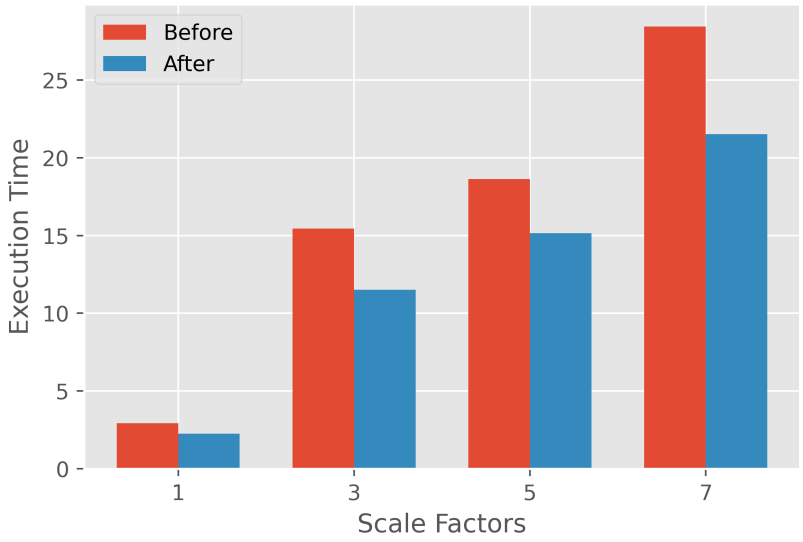

Figure 4.17: The execution time of Query 9 before and after the optimizations.

4.4.4 Query 88

This query answers the business question: *'How many items do we sell between pacific times of a day in certain stores to customers with one dependent count and 2 or less vehicles registered or 2 dependents with 4 or fewer vehicles registered or 3 dependents and five or less vehicles registered. In one row break the counts into sells from 8:30 to 9, 9 to 9:30, 9:30 to 10 ... 12 to 12:30'.*

The original query is presented in 4.9.

```
select  *
from
 (select count(*) h8_30_to_9
 from store_sales, household_demographics , time_dim, store
 where ss_sold_time_sk = time_dim.t_time_sk
     and ss_hdemo_sk = household_demographics.hd_demo_sk
     and ss_store_sk = s_store_sk
     and time_dim.t_hour = 8
     and time_dim.t_minute >= 30
     and ((household_demographics.hd_dep_count = 3 and household_demographics.
    hd_vehicle_count<=3+2) or
          (household_demographics.hd_dep_count = 0 and household_demographics.
    hd_vehicle_count<=0+2) or
          (household_demographics.hd_dep_count = 1 and household_demographics.
    hd_vehicle_count<=1+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 = 0 and household_demographics.
    hd_vehicle_count<=0+2) or
23           (household_demographics.hd_dep_count = 1 and household_demographics.
    hd_vehicle_count<=1+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 = 0 and household_demographics.
    hd_vehicle_count<=0+2) or
34           (household_demographics.hd_dep_count = 1 and household_demographics.
    hd_vehicle_count<=1+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 = 0 and household_demographics.
    hd_vehicle_count<=0+2) or
45           (household_demographics.hd_dep_count = 1 and household_demographics.
    hd_vehicle_count<=1+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 = 0 and household_demographics.
    hd_vehicle_count<=0+2) or
56           (household_demographics.hd_dep_count = 1 and household_demographics.
    hd_vehicle_count<=1+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 = 0 and household_demographics.
    hd_vehicle_count<=0+2) or
67           (household_demographics.hd_dep_count = 1 and household_demographics.
    hd_vehicle_count<=1+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 = 0 and household_demographics.
    hd_vehicle_count<=0+2) or
78           (household_demographics.hd_dep_count = 1 and household_demographics.
    hd_vehicle_count<=1+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 = 0 and household_demographics.
    hd_vehicle_count<=0+2) or
89           (household_demographics.hd_dep_count = 1 and household_demographics.
    hd_vehicle_count<=1+2))
90      and store.s_store_name = 'ese') s8;
```

Listing 4.9: query88.sql

We can observe that the join between the tables *store_sales, household_demographics, time_dim* and *store* is made several times in this query, so our idea was just to execute this join prior to the query. The proposed alternative is as in Listing 4.10

```
1  with sales as (
2      select *
3      from store_sales, household_demographics , time_dim, store
4      where ss_sold_time_sk = time_dim.t_time_sk
5          and ss_hdemo_sk = household_demographics.hd_demo_sk
6          and ss_store_sk = s_store_sk
7          and ((household_demographics.hd_dep_count = 3 and household_demographics.
    hd_vehicle_count<=3+2) or
8               (household_demographics.hd_dep_count = 0 and household_demographics.
    hd_vehicle_count<=0+2) or
9               (household_demographics.hd_dep_count = 1 and household_demographics.
    hd_vehicle_count<=1+2))
10         and store.s_store_name = 'ese'
11 )
12
```

```
13 select  *
14 from
15   (select count(*) h8_30_to_9 from sales
16   where sales.t_hour = 8
17      and sales.t_minute >= 30) s1,
18   (select count(*) h9_to_9_30 from sales
19   where sales.t_hour = 9
20      and sales.t_minute < 30) s2,
21   (select count(*) h9_30_to_10 from sales
22   where sales.t_hour = 9
23      and sales.t_minute >= 30) s3,
24   (select count(*) h10_to_10_30 from sales
25   where sales.t_hour = 10
26      and sales.t_minute < 30) s4,
27   (select count(*) h10_30_to_11 from sales
28   where sales.t_hour = 10
29      and sales.t_minute >= 30) s5,
30   (select count(*) h11_to_11_30 from sales
31   where sales.t_hour = 11
32      and sales.t_minute < 30) s6,
33   (select count(*) h11_30_to_12 from sales
34   where sales.t_hour = 11
35      and sales.t_minute >= 30) s7,
36   (select count(*) h12_to_12_30 from sales
37   where sales.t_hour = 12
38      and sales.t_minute < 30) s8;
```

Listing 4.10: Optimized query88

Reusing the join between the tables whenever it is needed led to significant improvements (from 3.7 to 5.8 times faster) in the execution time, as we can see in Figure 4.18.

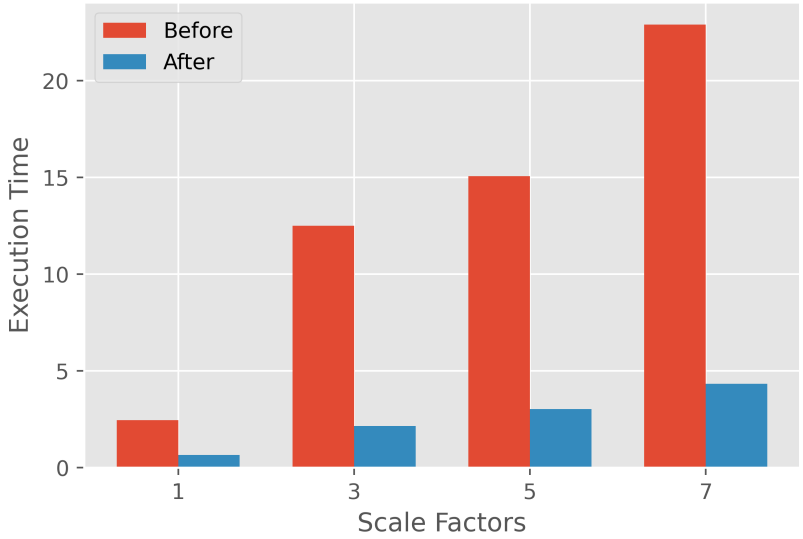

Figure 4.18: The execution time of Query 88 before and after the optimizations.

4.4.5 Query 35

This query was not very slow for scale factor 1, but we saw that it could be improved in the same way as query 69 and query 10, and we did it, improving its times. The details are not so interesting now, as it is the same as before. As it was mentioned before, the not optimized version of query 35

was running extremely long for bigger scale factors, so we did not run it for these. For scale factor 1, the optimized query 35 had a considerable improvement in the execution time and it could be run relatively fast for the other scale factors, as can be seen in Figure 4.19.

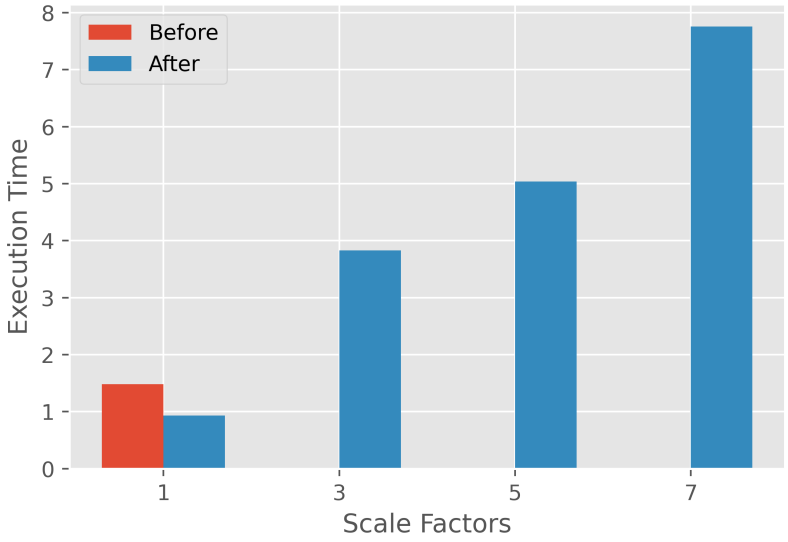

Figure 4.19: The execution time of Query 35 before and after the optimizations.

Chapter 5

Conclusion

5.1 Conclusion

Along this project, we have experienced all the difficulties it entails to benchmark a system. There are lots of considerations to take into account, as well as decisions to be made, that are critical to the results.

Moreover, we are now more aware of how careful one has to be when assessing a system. The benchmarking is an essential step to perform in order to be able to do a proper decision between several systems. Sometimes, the benchmarking process has already been overtaken by the company which provides the system as a service, but even in these cases one has to be careful and analyze the results they obtained as if they were one's results (or even more).

5.2 Further work

Here, we have performed a partial implementation of the benchmark, which can be improved in a number on ways. This section aims to name a few of them:

- Developing a more realistic scenario for the throughput test could lead to more reliable results.
- The Data Maintenance Test is the next step to be done to continue with the development of the benchmark.
- The metrics defined in the specification are also a must to complete the benchmark.
- It could be a good thing to fully automatize the process, so all the test could run one after one another. We have tried to advance in this line, but the problems we had with the Load Test when automatized made it impossible. Nonetheless, we believe there should be alternative ways to achieve such automation.

Bibliography

[TPC21] Transaction Processing Performance Council (TPC). *TPC BENCHMARK™ DS - Standard Specification*. 3.2.0. TPC. June 2021. URL: https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp.

[Dat21] Oracle® Database. *Data Warehousing Guide*. Oracle®. Aug. 2021. URL: https://docs.oracle.com/en/database/oracle/oracle-database/21/dwhsg/index.html.

[Dat22] Oracle® Database. *Oracle Database Express Edition*. Oracle®. 2022. URL: https://www.oracle.com/be/database/technologies/appdev/xe.html.

[Smi18] Jeff Smith. *Yes, SQL*Loader is Faster at Loading Records*. ThatJeffSmith. 2018. URL: http://www.thatjeffsmith.com/archive/2018/10/yes-sqlloader-is-faster-at-loading-records/.

[Tur83] Efraim Turban. "Decision support systems (DSS) a new frontier for industrial engineering". In: *Computers & Industrial Engineering* 7.1 (1983), pp. 41–48. ISSN: 0360-8352. DOI: https://doi.org/10.1016/0360-8352(83)90007-4. URL: https://www.sciencedirect.com/science/article/pii/0360835283900074.

[VZ22] Alejandro Vaisman and Esteban Zimányi. *Data Warehouse Systems*. Springer Berlin Heidelberg, 2022. DOI: 10.1007/978-3-662-65167-4.