ULB ECOLE POLYTECHNIQUE DE BRUXELLES

Group Project: Part 1
2024

# INFO-H-419: Data Warehouses

TPC-DS: Decision Support Benchmark

*Alfio Cardillo, Kristóf Balázs, Stefanos Kypritidis and Josu Bernal*

# Contents

# 1 Introduction

This project focuses on evaluating the performance of PostgreSQL as a data warehouse through the TPC-DS benchmark.

## 1.1 What is PostgreSQL?

PostgreSQL is an open-source relational database management system (RDBMS) that emphasizes extensibility and SQL compliance. It was developed at the University of California, Berkeley and is known for its robustness, reliability, and performance.

## 1.2 What is a benchmark for a RDBMS?

A benchmark for a Relational Database Management System (RDBMS) is a standardized test used to evaluate and compare the performance and scalability of different database systems. These benchmarks provide a scalable dataset and a set of queries to assess how well a RDBMS performs under specific workloads.

## 1.3 Aim and objectives

In our project, we are analyzing TPC-DS benchmark performance in PostgreSQL data warehouses. The TPC-DS benchmark is a benchmark that emulates several aspects of a decision support system[1]. Our objective is to assess the efficiency and scalability of PostgreSQL in handling decision-support workloads. This project aims to provide valuable insights into the performance of PostgreSQL as a data warehousing solution.

Furthermore, we made a big effort to generalize the setup process in order to facilitate easy reproducibility in various environments. By generalizing the setup, we aimed to ensure that our project could be easily replicated by others in any type of system. The complete project, including detailed instructions and necessary resources, can be found on our GitHub repository. We also drew on various resources and references from other projects, which can be found in our bibliography for additional insights and context. To avoid duplicating existing work, we used the project conducted by previous students as a starting point. We made several improvements, including enhancing reproducibility across different operating systems and conducting a more thorough query optimization.

Finally, we also worked to improve the performance of PostgreSQL through various techniques such as indexing, query optimization, and partitioning. These efforts aim to provide a comparison between the efficiency of an unoptimized database and an optimized one in PostgreSQL. By implementing these strategies, we hope to highlight the

[1]OLAP system that helps enterprises to make decisions by providing relevant data and analysis

enhancements that can be achieved and offer insights into best practices for optimizing PostgreSQL for data warehousing applications.

## 1.4 GitHub repository

As the reader will see throughout this report, we reference our GitHub repository multiple times to avoid unnecessary explanations that do not contribute value to our findings. For those interested in exploring the insights of our work or replicating it, we have included the repository link here: `https://github.com/Carda01/tpcds-benchmark`

# 2 Technical specifications

In this section, we will outline the software and hardware used to conduct this benchmark, including the versions of the tools and the technical specifications of the machine on which the evaluation was performed. This information provides essential context for our benchmark, as the results are highly dependent on the machine executing the tests. By including these details, we aim to help readers better understand the significance of the results presented.

## 2.1 Software specifications

In this subsection, we display the specifications of the tools used for this project.

| **Tool** | **Version** | **Description** |
|---|---|---|
| PostgreSQL | 16.4 | This is the tool being evaluated. It is a relational database management system (RDBMS) known for its extensibility. |
| Docker Desktop | latest | We utilized this tool to ensure reproducibility across all operating systems. It functions as a container-based application launcher. |
| Python | latest | We used it to load the database and execute queries. It is a popular programming language. |
| Jupyter Notebook | latest | IDE for Python and iPython notebook. |
| MS PowerBI | latest | Power BI was utilized to create visualizations of benchmarking results for further analysis. |
| Git-GitHub | latest | Git and GitHub Desktop were used to share code files between the team members and to post the final repository. |

## 2.2 Hardware specifications

In this subsection, we display the hardware specifications of the machine used for this project.

| | |
|---|---|
| **CPU** | |
| OS Name | Microsoft Windows 11 Pro |
| Processor | 13th Gen Intel(R) Core(TM) i5-13500H, 2600 Mhz, 12 Core(s), 16 Logical Processor(s) |
| RAM | 32.0 GB |
| **GPU** | |
| Name | NVIDIA GeForce RTX 3050 |
| VRAM | 6GB |

## 2.3 Postgres configuration

In this subsection, we will go into technical details of the PostgreSQL configuration that we have used. We modified this by changing the "postgresql.conf" file, with the purpose of improving the allocation of resources of PostgreSQL to increase performance and take advantage of our hardware specifications. This information could be useful for anyone wanting to reproduce our results. We will only list the changes made because they are pretty self-explanatory.

- Memory
  - shared_buffers = 4GB (original is 128MB)
  - work_mem = 16MB (original is 4MB and is commented)
  - maintenance_work_mem = 1GB (original is 64MB and is commented)
- Asynchronous Behavior
  - max_worker_processes = 16 (original is 8 and is commented)
  - max_parallel_workers_per_gather = 8 (original is 2 and is commented)
  - max_parallel_maintenance_workers = 8 (original is 2 and is commented)
  - max_parallel_workers = 16 (original is 8 and is commented)
- Planner Cost Constants (This will play a big role later)
  - random_page_cost = 1.1 (original is 4.0 and is commented, uncomment also seq_page_cost )
  - effective_cache_size = 24GB (original 4GB and commented)

# 3 Limitations and failed attempts

In this section, we will explain the technical limitations that we have faced, the worklines that we could not follow and the attempts that we made to improve the project with no luck. We hope that this section brings a closer attention to the time and detail that we put in this project and we expect it to serve as guideline for future students on what they could try to achieve.

## 3.1 Technical Limitations

Benchmarking is only useful when conducted on a single machine. This process has become much longer and more tedious for this reason, making the group work highly dependent in the chosen computer. We did not have sufficiently powerful local computers to efficiently run this benchmark on a larger scale factor, as ideally we would have liked to, and as we show in the following sections the biggest we achieved to run successfully was SF-4. However, these limitations made us put our efforts in other parallel objectives such as generalization of the process, enhancing reproducibility and better optimization.

## 3.2 Failed attempts

In this subsection, we would like to underline several topics in which we invested time, even if they did not make it to the report due to different reasons. We think that it is important to take these into account since they show the magnitude of our work is not limited to what is shown in this report, and also provides guidelines of what the following steps could be if interested in continuing with this project.

1. *Complete automatization of the process:* Even if we worked meticulously on generalizing, easing, and enabling this process into different operating systems, we have tried to create a complete automatization of the process, making it trivial for the next person attempting to conduct this benchmark. However, due to the lack of time, technical difficulties of the idea itself and not being the main objective of the project, we abandoned it, not without making big improvements in all these fields, and making a much easier setup process.

2. *Using GPUs:* Due to the lack of a more powerful computer and counting with the advantage of having GPUs in the machine selected to run the benchmark, we attempted to use them to improve the performance. However, after spending several days reading the official PostgreSQL documentation regarding GPUs, and a not-easy driver installation we decided that using GPUs missed the main purpose of this project, which is benchmarking PostgreSQL over "bare metal". Despite this, experimenting with TPC-DS on Linux systems using PG-Strom, which is compatible with CUDA and other GPU libraries (unavailable in the Windows

subsystem for Linux due to driver incompatibilities), would be an interesting direction for future exploration.

# 4 Benchmark setup and data warehouse creation

In this section, we will explain how we performed the benchmark setup, what we did to make the process easier to replicate, and why we did what we did.

Although the TPC-DS benchmark process should be straightforward, it proved to be challenging due to the lack of examples and poor explanations in the official files. Also in our group we had different requirements as some of us had different operating systems. Recognizing these issues, we have endeavored to simplify the process by automating the setup, making it OS independent and providing a clear, step-by-step guide to assist anyone in the setup.

To achieve this goal, we've created a Docker image that automates the generation of data for different scale factors. This means that anyone who wants to perform a benchmark would only need to get our repository, unzip the original TPCDS file, build the Docker image that we have coded for this purpose and run a container. We won't explain the exact process to perform this setup in this report, if the reader is interested the process is clearly explained in our repository README.

Moreover, even if we will not dig into details, we will explain the main key points of how we programmed and automated this process. These are the main files that are the base of our automation:

- **auto_run_dsgen.sh :** It is the script that serves as entrypoint in the containers later created. It is responsible of generating the CVS files by running 4 parallel processes of dsgen data generator for the appropriate scale factor and at the same time logging the generation. The scale factor can be passed in the environment or if not present, the user will be prompted to give one, so that the image can work for any scale factor, without using the docker compose.
- **Dockerfile:** In this file we specify the image that will serve as base for the data generator containers. It is based on Ubuntu, it installs all the necessary dependencies to later build and run dsgen. It expects that in the working directory you have already extracted the DSGen zip so that it can add it in the image, run various commands that fix the workspace and without whom dsgen won't work and create a folder where the data will be finally stored. Lastly it will run the previously defined **auto_run_dsgen.sh**.
- **docker-compose.yml:** Here we actually build up the data. We run multiple containers of the previously defined image and for each we pass a different scale factor. In our current example we've created four containers with scale factor from 1 to 4, but one can run more or change the scale factors. Lastly for each container we mount the data folder of the local system inside the container. This will make sure that the data is generated not only inside the container, but

also in the local system at the same time. While it works perfectly on Linux, unfortunately on Windows doing that will dramatically slow down the generation, so we recommend removing the volume attachment and copying the files after the generation is finished.

After running this image, it will have generated the CSV files and queries. The next step in the benchmark process is to create the database, tables, and load the data into the different relations. We have created a Python file for this purpose. However, we recommend running it within a virtual environment, which is also provided in our repository. The process is explained in detail in the README so that we won't go over it here. Furthermore, for security reasons, certain local variables should be stored in the .env file since this document is part of an online repository.

After all of this is done, again we recommend to follow our step-by-step guide to not miss anything, your database should be ready to perform the benchmark. Additionally, the scripts we have used to perform the benchmark itself can be found also in our repository.

To sum up, we have worked hard in easing the process of setting up and running the benchmark, automating and generalizing some of the mist tedious parts. We have produced a system that is able to generate data in parallel independently from the OS you are using. We have also worked on a proper guide on how to do this, since the official guide did not provide enough details and missed some of the key points of this process.

# 5 Theoretical Basis

In this section, we will explore the theoretical basis of optimization that we will implement later, digging into detail in some of the key elements of database optimization, such as indexes and partitions.

Query optimization is a critical aspect of database management systems that aims to enhance the efficiency of data retrieval processes. We will explain the fundamentals of query optimization which will help the reader understand why and how we implemented methods to improve response times for some of the queries of the benchmark.

For a clearer explanation, we will divide the optimization techniques into two groups: one focused on improving the syntax of the queries, and the other focused on updating the indexing and the structure of the database.

## 5.1 Syntax optimization

There are many ways of improving the syntax of a query, but the goal is the same for all of them: finding an equivalent query that performs faster. We will present some of the several equivalences that could be used to improve the running times.

- Avoid subqueries in the FROM clause. They limit the query plan flexibility.
- Using ANY or IN and an array instead of OR in the WHERE clause.
- Avoid correlated subqueries that must be calculated for each row. Using CTEs could be a possible solution.
- Explicit joins usually perform better than filters in the WHERE clause.
- Using temporary tables instead of CTEs.

Let's take a closer look at this last optimization proposal, as it plays a significant role in our project and its importance may not be immediately clear.

The main reason for this change in our project is that some queries were declaring some CTEs multiple times, in order to reuse them. This makes our database perform the same calculation over and over again. That is why the use of temporary tables is more beneficial in these cases. In addition to that, we would like to add here some other benefits of temporary tables that we learned through this project.

- Temporary tables are fully optimized as part of query execution and can utilize indexing. CTEs do not support indexing (even if materialized), which often limits them to sequential scans. We will discuss indexing in more detail later.

- When it comes to concurrency and resource usage, temporary tables are stored on disk or in memory, depending on your configuration. PostgreSQL manages these resources efficiently, particularly with large datasets. The query planner can optimize access to temporary tables flexibly, utilizing disk-based operations when necessary. In contrast, if CTEs are materialized, they can consume substantial memory or spill to disk inefficiently, particularly when dealing with large result sets.

Since we have mentioned them, we have to explain that there are two types of CTEs: materialized and not materialized CTEs. A materialized CTE in PostgreSQL is one where the result of the CTE is computed and stored temporarily at the beginning of the query execution. A non-materialized CTE, on the other hand, is not stored temporarily. However, will not dig deeper into materialized and not materialized CTEs since we think that it is not mportant for this project, however we knowledge their existence and considered them as an option.

*IMPORTANT: Temporary tables and materialized CTEs are not the same.*

There are many more ways to improve syntax, but these often depend on the context of the query. For this reason, we will not explain this in-depth here. If the reader is interested in the changes made to the queries, they can check our repository.

## 5.2 Indexing and file types

In this subsection, we will provide a brief summary and explanation of the different file and index types used in databases. We will explain file types, as they are an essential first step in understanding database architectures. However, we will provide only a brief overview and focus primarily on indexes, as they will be one of the most important aspects of our project.

### 5.2.1 File types:

Databases store their data in structures called files, with each row referred to as a record. The way these files are structured can influence the time it takes to perform a query. Therefore, it is important to understand the different types of files, even at a general level. We normally consider three main types of files.

- *Heap files:* Records are storaged unsorted. They offer good storage efficiency and have effective insertion and deletion capabilities; however, they are slow when it comes to searching.
- *Sorted files:* Records are storaged sorted. Provides good storage efficiency. Insertions and deletions are slow, but searching is fast. It is the best structure for range selections.

- *Hashed files:* Uses a hashing algorithm for distributing records. Storage efficiency is not very good. Selection, insertion, and deletion are fast. However, there is no support for range selections.

Now that we know the advantages and disadvantages of each type we can think about what is suitable. However, we have to know that a lot of times we can improve the inefficiencies inherent to the file by using indexes.

### 5.2.2 Index types:

An index on a file is a data structure that accelerates selections on the preselected search key[2] fields. There are many types of indexes. In this section, we will explain in detail the most important ones.

- *B trees:* B-trees maintain a sorted, balanced tree structure that allows for range queries and ordered data traversal, making them suitable for scenarios where data needs to be accessed in a sorted manner or where range searches are common. Additionally, B-trees are dynamic, supporting efficient insertions and deletions.
- *Hashing indexes:* Hashing indexes use a hash function to map keys to specific locations in memory, providing constant time complexity for lookups on average, but they lack the ability to perform range queries since the keys are not stored in any particular order. Hashing indexes can suffer from collisions when insertion and deletions, requiring additional strategies to handle them.

Finally, we will explain what partitioning is since it plays an important role in our query optimization process.

## 5.3 Partitioning

Partitioning is a database design technique that involves dividing a large dataset into smaller, more manageable segments, known as partitions. This approach enhances performance and efficiency by allowing queries to target specific subsets of data rather than scanning the entire dataset. Citing PostgreSQL official webpage:

> *Query performance can be improved dramatically in certain situations, particularly when most of the heavily accessed rows of the table are in a single partition or a small number of partitions. Partitioning effectively substitutes for the upper tree levels of indexes, making it more likely that the heavily-used parts of the indexes fit in memory.*

[2]Search key is not a synonym of key. A search key doesn't have to apply uniqueness requirements.

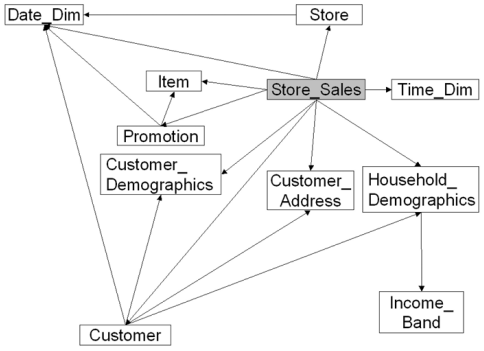
Partitioning can be implemented in several ways. In this project, we used range partitioning, where data is divided based on specific ranges of values of a column or set of columns. There are other types of partitions such as list partitioning and hash partitioning, but they don't add value to the project, and consequently, we will just mention them.

# 6 Data warehouse description

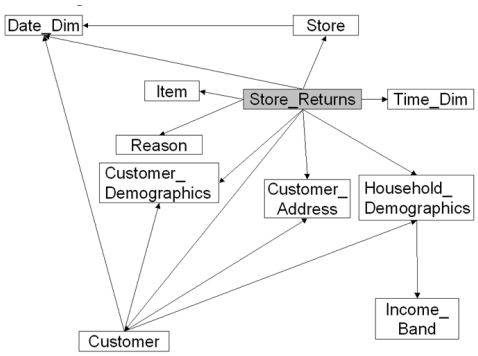
In this section, we present the data warehouse schema and a table that shows the total number of tuples stored in each table for each scale factor. The database includes multiple fact tables with numerous attributes. This description will focus solely on the fact tables and in the tables, providing official ER diagrams, found in the official TPC-DS files, for each. We will not delve deeply into the explanation of every attribute because we think that this doesn't add any value to this project. The purpose of this section is to understand the magnitude of the data warehouse being studied, not the data warehouse itself.

## 6.1 Fact table ER diagrams

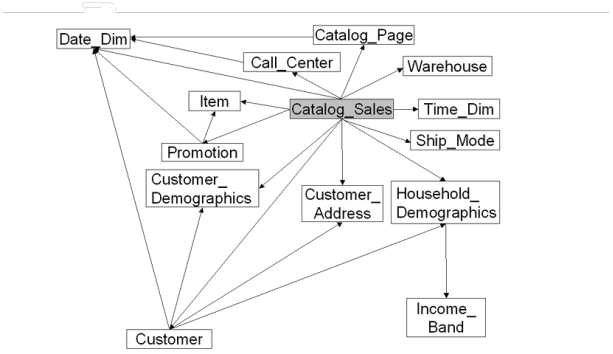
### 6.1.1 Store Sales



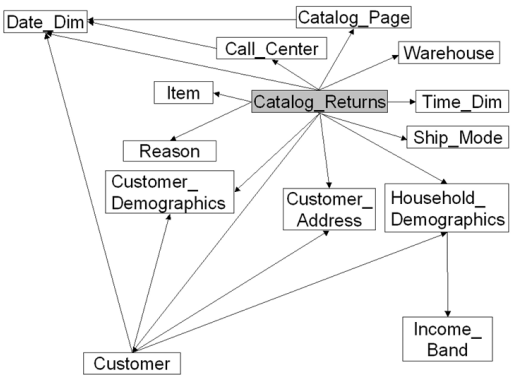

### 6.1.2 Store Returns



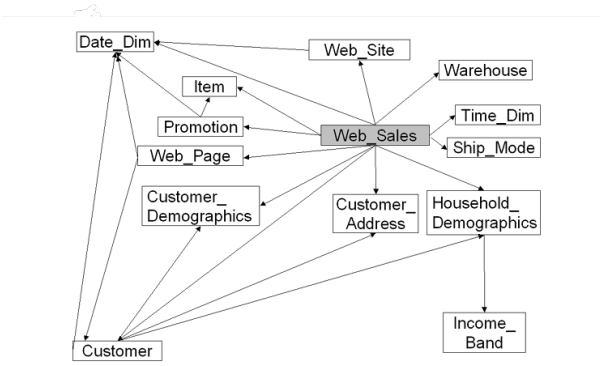

### 6.1.3 Catalog Sales



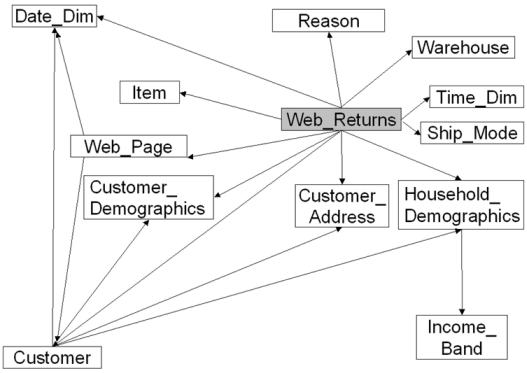

### 6.1.4 Catalog Returns



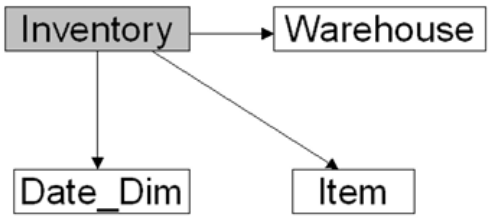
### 6.1.5 Web Sales




### 6.1.6 Web Returns




### 6.1.7 Inventory




## 6.2 Scaling factor table

Now we will introduce the scaling factor table, it show how many tuples each table has for each scaling factor. This table is essential for understanding the significance of the scale factors and the real meaning of the increase in time across the scales. *(Here we are showing the original tables. As we will explain later some of these tables have been partitioned for performance optimization.)*

| **Table** | **SF1** | **SF2** | **SF3** | **SF4** |
|---|---|---|---|---|
| call_center | 6 | 8 | 10 | 12 |
| catalog_page | 11718 | 11718 | 11718 | 11718 |
| catalog_returns | 144067 | 288491 | 432000 | 576254 |
| catalog_sales | 1441548 | 2880058 | 4319367 | 5759989 |
| customer | 100000 | 144000 | 188000 | 233000 |
| customer_address | 50000 | 72000 | 94000 | 116000 |
| customer_demographics | 1920800 | 1920800 | 1920800 | 1920800 |
| date_dim | 73049 | 73049 | 73049 | 73049 |
| household_demographics | 7200 | 7200 | 7200 | 7200 |
| income_band | 20 | 20 | 20 | 20 |
| inventory | 11745000 | 16966305 | 28188000 | 36019566 |
| item | 18000 | 26000 | 36000 | 46000 |
| promotions | 300 | 322 | 344 | 366 |
| reason | 35 | 36 | 37 | 38 |
| ship_mode | 20 | 20 | 20 | 20 |
| store | 12 | 22 | 32 | 42 |
| store_returns | 287514 | 575285 | 862834 | 1149565 |
| store_sales | 2880404 | 5760749 | 8639377 | 11519024 |
| time_dim | 86400 | 86400 | 86400 | 86400 |
| warehouse | 5 | 5 | 6 | 6 |
| web_page | 60 | 74 | 90 | 106 |
| web_returns | 71763 | 143629 | 215477 | 287614 |
| web_sales | 719384 | 1439247 | 2160165 | 2879360 |
| web_site | 30 | 30 | 32 | 34 |

# 7 Benchmark

In this section, we will provide a detailed explanation of each of the three experiments we conducted. The results of these experiments are the times that we will compare later. Each experiment involved running 99 different queries on different datasets. The variations among the experiments are essentially improvements made to the queries and the dataset architecture, allowing us to compare their performances.

The datasets used for these experiments varied in size to evaluate how performance changes as dataset size increases. We tested the following scale factors for each set of queries: 1, 2, and 3. Additionally, we will include scale factor 4 for the two experiments with optimized queries.

## 7.1 Base queries

The first experiment we conducted involved running the 99 official queries provided by TPC. Some of these queries were incompatible with PostgreSQL or contained syntax errors, making them unexecutable. This occurs because the most similar available TPC-DS queries for PostgreSQL are written in Netezza syntax, which resembles PostgreSQL, but is not identical. Therefore, we used a corrected version of these queries for our experiment. Instead of spending a long time correcting them ourselves, we utilized the previous work done by students from past years to streamline the process. All these queries, and the ones that we will mention in the following experiments, can be found in our Github repository.

To be more concrete and to emphasize the work done by our previous colleagues, we cite here the lines from their report where they explain the process conducted by them:

> *To ensure complete compatibility of queries with PostgreSQL, a few minor modifications were done.*
>
> - *Intervals: Postgres requires the specific keyword- interval to identify the interval of days, Query Syntax were modified.*
> - *Aliases: A few of the sub-queries and columns were given an alias "x".*
> - *Joins: Instead of generally selecting from Table A, Table B and specifying the JOIN condition in WHERE Clause, explicit joins (inner/left-/right) were used.*

This set of 99 queries will be referred as "base queries" in the following sections.

## 7.2 Referenced queries

In this second experiment, we aimed to understand the achievements of our previous colleagues in their optimization efforts. This knowledge would serve as a starting point for our own optimization process. Therefore, we executed the optimized queries they developed during this second round.

To provide better context regarding their work, we have included a citation from their report that explains this in detail:

> *The ways used in this project to optimize are:*
>
> - *Query plans were studied thoroughly to analyze possible points of optimization.*
> - *Correlated Sub-queries were rewritten using Common Table Expressions (CTE).*
> - *Indexes were added for fast retrieval of table tuples.*
> - *Tables in joins were re-ordered in the increasing order of their counts ensuring efficient joins.*
> - *Distinct keyword was eliminated by rewriting queries in the form of Common Table Expressions (CTE)*

In this optimization process, our colleagues only optimized 6 queries, more specifically the queries 1, 4, 6, 11, 74 and 81. This set of 99 queries will be referred as "referenced queries" in the following sections.

## 7.3 Our queries

In this third experiment, we aimed to optimize the performance of our data warehouse. Given the limited time available for our project, we focused on various techniques to achieve this goal. With more time, we could have explored additional methods and paid closer attention to each query individually. For this explanation, we will assume that the reader is familiar with the content of the previous chapters.

We started by looking at what had our previous colleagues done, to save time and not repeat already performed work, we used their already optimized queries as our starting point. From here we perform three different optimization techniques:

### 7.3.1 WHERE clause mining

As we have already a view on the use case of our database with all the queries we want to perform on it we've decided to gain some insights from them to help us decide what's the best index or on how to partition our tables. To do so, in the analytics

notebook, we've extracted all WHERE clauses from all the 99 queries and did some analysis on them.

During our first version of the analysis, we've decided to mine the clauses with regular expressions, but while trying to have a better catching pattern, we soon had reached a very complex matching structure that was hard to debug. Complexity came from the various subqueries and **CASE WHEN** conditions.

Finally we decided to use a SQL parser, called **sqlglot** that helped us extracting the wheres. Actually, more than the whole clause, we were interested in the actual conditions. So from the WHERE clauses we extracted the conditions concatenated with the various **AND** and **OR** operators and built a list of all the conditions from the 99 queries.

We then took the table pg_stats and we extended it by adding two new columns, **"equalities"** and **"comparisons"**. Basically for each column of the database we wanted to calculate the number of equalities where it appeared, so conditions where operators like =, **IN**, <> were used and the number of comparisons, i.e. conditions where $>, <, <=, >=$, **BETWEEN** operators were used.

Hence, for each column of our database we checked if it appeared in any condition and what was the operator used, and we incremented the "equalities" or "comparison" column counter based on the match. To do that we used regular expressions, as the edge cases were not that many.

Although the final analysis might not be perfect and some edge cases might yet not be covered, this analysis gave us some insights we used in the later sections.

Future work may focus on fine tuning the algorithm to increase the precision or also mining **GROUP BY** statements which can allow better index creation.

### 7.3.2 Indexing

Choosing on which column and of which type we should build the indexes can be a tedious task that requires knowledge on the use cases. But the analysis we did before was helpful in these regards.

Taking into account the number of equalities and comparison conditions performed for each column, we decided if we should create a B-tree index, a Hash index or none.

This process involves many more details; however, we will not explore them further, as they do not add value to this report. Based on this analysis, we decided to implement 37 different indexes (after first removing the existing ones), including 7 B-trees and 30 hash indexes. This entire process was automated; the relevant code can be found in the file titled "analytics.ipynb" in our GitHub repository.

Future work may involve a better analysis on the type of index that can be created based on the various statistics given by the extended pg_stats table and playing around with the thresholds on the minimum number of operations needed to create an index.

### 7.3.3 Partitioning

Secondly, we partitioned some of the tables in our data warehouse. We observed that large fact tables, such as catalog_sales, store_sales, and web_sales, were consistently being filtered by date. Therefore, we decided to apply range partitioning to these three tables based on the year of the date.

This process was implemented by some queries that can be found in the partition_creation folder.

### 7.3.4 Syntax Optimization

Finally, we updated the syntax of some of the queries. More specifically, we worked on 14 of the 99 queries. In this section, we will explain the main changes done to these queries. These changes have been already explained theoretically in previous chapters.

1. To take advantage of the new partitions, some of the queries were updated.
2. In queries 14, 23, and 39, we changed CTEs for temporal tables. We did this because the CTEs were being declared multiple times and, as explained before, by using temporary tables we can avoid repeating the same calculation over and over again.
3. Most of the changes done are specific to the context of the query itself. For an individual explanation of each, the changes are explained and supported with comments in our repository. Some examples are:
    - dividing long CTEs into multiple CTEs to avoid repetitions and dealing with huge table joins,
    - replacing huge fact tables joins, with filtered CTEs to utilize early filtering,
    - using explicit joins instead of implicit joins for better readability and potential optimization,
    - replacing the join of another fact table and the filtering if its primary key is null with "NOT EXISTS" subquery,
    - replacing long subqueries inside "exists" and "in" operators with CTEs,
    - replacing subquery in where condition and ">" operator with the creation of CTE and joining that CTE in the main query,

This set of 99 queries will be referred to as "our queries" in the following sections.

# 8 Results and discussion

In this section, we will explain the results obtained from the already explained experiments, we will compare them, and discuss the findings.

The times obtained were saved in a CSV file, and we created a Microsoft Power BI dashboard to visualize the results. If the reader is interested, the dashboard can be found in our GitHub repository: Microsoft Power BI Dashboard.

## 8.1 Individual results for each scale factor

**Base queries** In this paragraph, we will display the results obtained for the base queries. This will be helpful in understanding the magnitude of the problem itself, the capability of the machine used, and the difference between scale factors.

The total time required to run the 99 queries across the three scale factors was 22.47 hours (80,900.13 seconds). Specifically, the third scale factor took 16.91 hours (60,893.88 seconds), the second took 3.69 hours (13,303.13 seconds), and the first took 1.86 hours (6,703.12 seconds). To better visualize the differences in these times, a pie chart can be used to display the percentage of total time each scale factor required for execution.

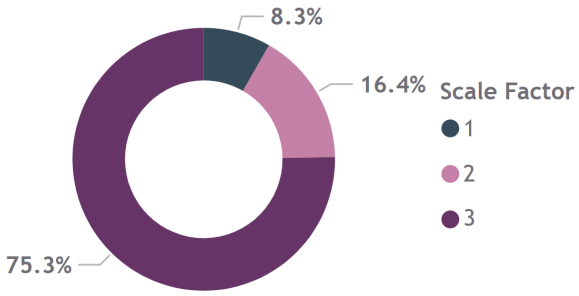



Figure 1: Percentages of total time per scale factor for base queries

Of course, the duration of queries can vary significantly. Therefore, it's important to focus on those that take the longest to execute or show the greatest increase in time with each scale factor. These are the queries we aim to optimize. For this reason, we are presenting a graph that displays the longest-running queries. The run time of the queries not shown in this graph is also important, but it is depreciable compared to the longest ones in this graph. The y-axis of the graph represents the number of queries, while the x-axis shows the time required to run them in seconds. The displayed time is the sum of three scale factors, each represented in a different color for easy comparison at a glance.


Figure 2: Longest-running queries for base queries

As we can see in the graph queries 4, 74, 11, and 6 take significantly longer than the other queries. In addition, we can notice strange behavior in some queries. Take a closer look at query 4 and query 11. How is it possible for scale factor 1 to take longer than scale factor 2, or in the case of query 11 even that scale factor 3? We will talk about this anomalies in the following subsection, for now, let's forget about this behavior and let's put our attention on the running times and comparison of different experiments.

**Referenced queries** In this paragraph, we will display the results obtained for the referenced queries. This will help us understand the improvements made by our previous colleagues and set the starting point of our optimization process.

The total time required to run the 99 optimized queries across the three scale factors was 2.30 hours (8,307.64 seconds). Notice that in this experiment we also ran scale factor 4, and in the previous one we didn't, so for proper comparison between experiments the total run time for the first three scale factors is 1.30 hours (4,687.44 seconds); significantly shorter than the previous one. Specifically, the fourth scale factor took 1 hour (3,620.20 seconds), the third took 0.71 hours (2,581.28 seconds), the second took 0.37 hours (1,350.42 seconds), and the first took 0.20 hours (755.74 seconds). To better visualize the difference in these times, again, a pie chart can be used to display the percentage of total time each scale factor required for execution.

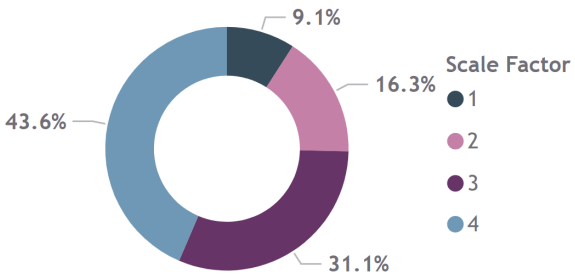



Figure 3: Percentages of total time per scale factor for the referenced queries

As explained in the previous paragraph, the duration of queries can vary significantly. Therefore, it's important to focus on those that take the longest to execute or show the greatest increase in time with each scale factor. Consequently, we are presenting a graph that displays the longest-running queries. Notice that this time we are displaying the fourth scale factor too.

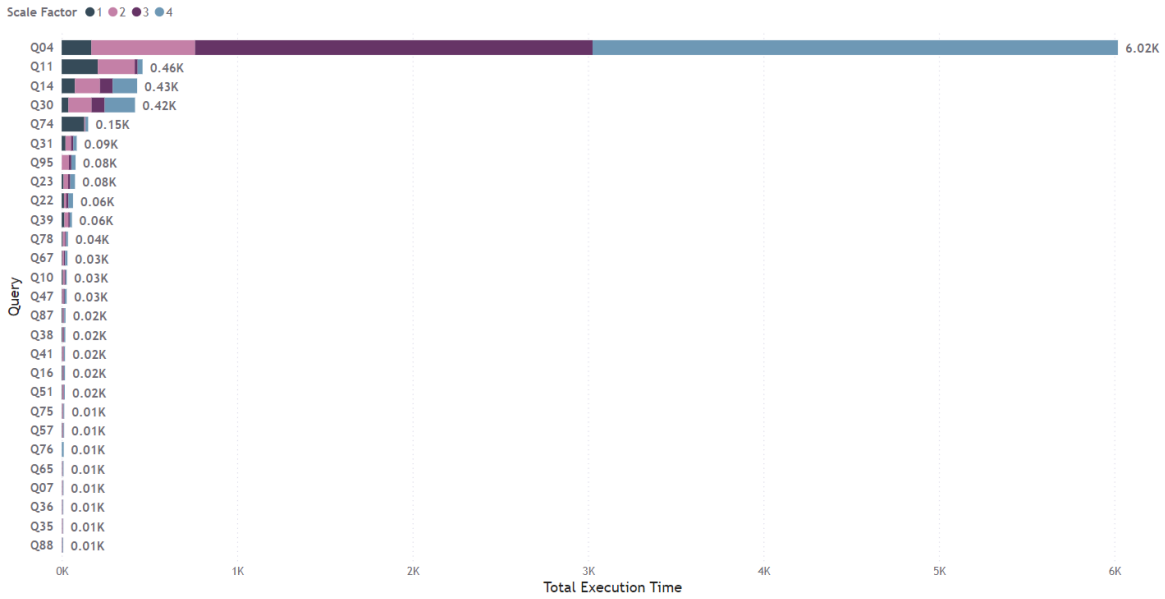



Figure 4: Longest-running queries for the referenced queries

As we can see again, even if the times are significantly lower, some of the same queries as before show still bad behavior and timing. Our optimization work mainly focused on lowering the running time of these queries and making the times more even.

**Our queries** In this paragraph, we will display the results obtained for our queries. This will show how our optimization performed and what we have archived through

this process.

The total time required to run the 99 optimized queries across the three scale factors was 0.32 hours (1,176.51 seconds). Notice that in this experiment we also ran scale factor 4, and in the first one we didn't, so for proper comparison between experiments the total run time for the first three scale factors is 0.25 hours (905.49 seconds); significantly shorter than the previous one, and much shorter than the first experiment. Specifically, the fourth scale factor took 0.07 hours (271.02 seconds), the third took 0.14 hours (533.97 seconds), the second took 0.06 hours (237.61 seconds), and the first took 0.03 hours (133.91 seconds). To better visualize the difference in these times, again, a pie chart can be used to display the percentage of total time each scale factor required for execution.

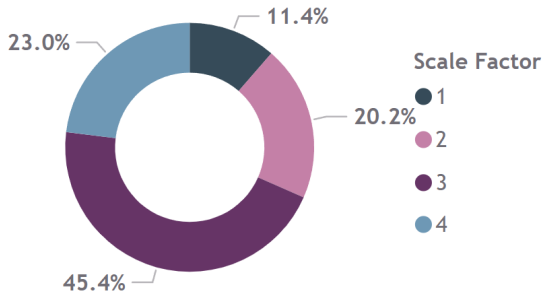



Figure 5: Percentages of total time per scale factor for our queries

As explained previously, the duration of queries can vary significantly. Therefore, it's important to focus on those that take the longest to execute or show the greatest increase in time with each scale factor. Consequently, we are presenting a graph that displays the longest-running queries. Notice that this time we are displaying the fourth scale factor too.

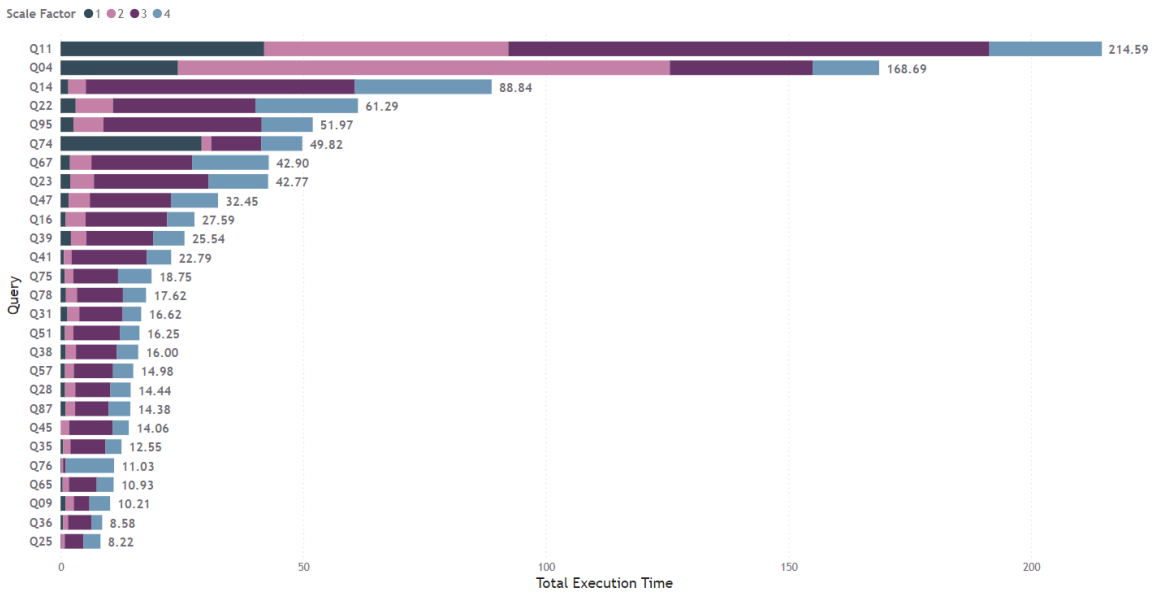



Figure 6: Longest-running queries for our queries

## 8.2 Comparision of results between experiments

We will now dig deeper into comparing the archived results for each experiment. We have already noticed that the times have improved considerably after each optimization. But, being objective, how much is "considerably"? Let's display some self-explanatory statistics.

| | Total Time | SF-1 | SF- 2 | SF-3 | SF-4 |
|---|---|---|---|---|---|
| Base queries - Our queries | 79,994.64 | 6,569.21 | 13,065.52 | 60,359.91 | X |
| Referred queries - Our queries | 7,131.13 | 621.83 | 1,112.81 | 2,047.31 | 3,349.18 |
| $\frac{Basequeries}{Ourqueries}$ | 89.34 | 50.05 | 55.98 | 114.03 | X |
| $\frac{Referedqueries}{Ourqueries}$ | 7.064 | 5.64 | 5.68 | 4.83 | 13.35 |

So as we can see in the table our queries are overall 89.34 times faster than the base queries or in other words we got an 8934% improved performance. For the referenced queries the numbers are still pretty impressive, being in overall 7.064 times faster. We also defined a different measure in the following way

$$G : \mathbb{R}^n \times \mathbb{R}^n \longrightarrow \mathbb{R}$$
$$(\vec{x}, \vec{y}) \longmapsto 100 - \frac{\overline{x} - \overline{y}}{x} \times 100$$

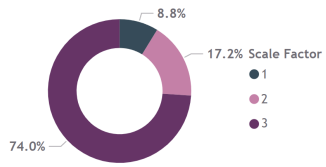
This measurement indicator shows which percentage of the time that takes x to run needs y to run. We calculated it to compare both the base and referenced queries with our queries. Getting the following results

$G(\text{Base queries}, \text{Our queries}) = 1.09\%, \quad G(\text{Referenced queries}, \text{Our queries}) = 14,16\%.$

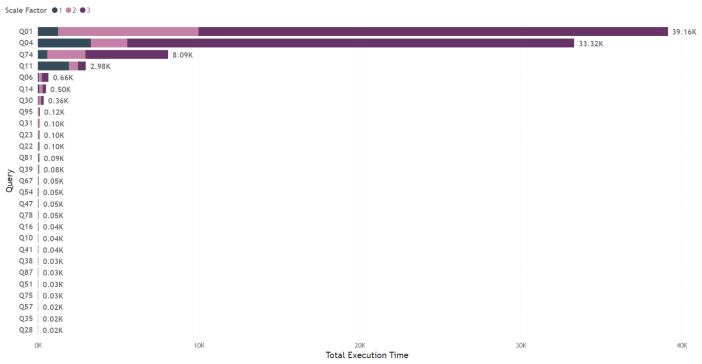
In other words, our queries will run in 1.09% of the base queries runtime and in 14.16% of the referenced queries.

*(This indicator is shown in the dashboard differently. It is under the name of performance gain (PG) and it actually shows the complementary percentage, in other words, $G = (100 - PG)$. We decided to show it here in this way since we think that it is a more understandable measurement indicator.)*

Additionally, we also could display for a better understanding of the scale factors' performance a pie chart of the overall time of every experiment for every scale factor and the overall longest-running queries. We will exclude the SF-4 for this graphs since it was not ran for every experiment.




(a) Overall pie chart




(b) Overall bar plot

## 8.3 Anomalies and irregular results

In this subsection, we will dig deeper into the anomalies that we previously noticed in some of the running times across different scale factors. These queries did not follow the expected behavior. The bigger the dataset is, the longer we expect the query to run. However, this pattern was not followed in some cases.

Actually, we are not the first who notice these anomalies. Our previous colleagues already looked into this and they gave the following explanation, which we find explanatory enough to cite here:

> *(Talking about query 4) The query worked very well for SF 1, but as the scale factor increased to 3 and 5, the run time increased exponentially, but then interestingly reduced when scaling to SF 10 and 15.*
>
> *In order to understand what changed between SF 5 and SF 15, a deeper dive was done into the query planner and statistics in terms of how the query*

*was executed in steps for both of these scale factors.*

*Firstly, inspecting the query plan for both scale factors, it's evident that different initial steps were taken. As seen from Figure 4.11 for SF 5, the query planner decided to make use of all indexes available but then proceeded to the next step with multiple nested loop inner joins. On the other hand, in Figure 4.13 for SF 15, the query planner felt that the customer table was small enough in comparison to two out of three of the large fact tables (catalog sales store sales) and decided to perform a sequential scan for them instead of using an index. This resulted in less usage of nested loop inner joins overall for the next steps.*

*Looking further into the statistics for the query run, it can be observed that the additional nested loop inner joins resulted in a much more expensive execution overall.*

Even if we are not attaching the figures mentioned in their report, since we don't think that they are necessary to understand what is going on, we have a first answer and somehow an explanation of what is happening in these cases. However, we are not happy with just knowing that different query plans are being utilized. That is why, we wanted to know why these choices were being made.

To understand why these choices are being made by the query planner we have to take into account two different concepts: I/O cost and the filtering condition threshold.

### 8.3.1 I/O cost

I/O cost, or Input/Output cost, refers to the time and resources needed to read from or write to a database. In database operations, I/O costs are often the most significant factor affecting query performance, particularly when dealing with large datasets. This is because accessing data from disk is slower than processing it in memory (RAM).

We can differentiate different types of I/O cost, but in this case, we will only explain the two types concerning our problem here:

- **Random I/O**: Accessing non-contiguous sections of data on disk is what we call Random I/O." Index lookups often involve random I/O because they locate specific rows by jumping to various locations on the disk. This is usually slower due to mechanical and latency limitations in storage.
- **Sequential I/O**: Sequential I/O occurs when data is read in contiguously, such as in a sequential scan. Sequential reads are generally much faster than random I/O because the data is stored in order, minimizing disk movement and enabling faster read speeds.

Then, why would we ever use indexes, if the I/O cost is greater than the sequential I/O cost? This is where the other concept comes into play.

### 8.3.2 Filtering condition threshold

To understand this concept we need to know some guidelines of how the query planner works, more specifically, we should at least know that the query planner chooses what to do by first inferring how many rows and cost will return each operation. Citing PostgreSQL official documentation:

> *The planner uses the system statistics to estimate the number of rows each stage in a query might return. This is a significant part of the planning / optimizing process, providing much of the raw material for cost calculation.*

For more information about this topic, refer to the official documentation. So now we know that the query planner estimates beforehand how many rows will be returned from the filtering. This is how it decides whether to use indexes or not. If the filtering condition returns a big portion of the original table, the I/O cost of using indexes will make using sequential scanning better than using indexes. On the other hand, if the filtering returns a small set of rows, the faster capacity of searching that indexes provide will increase the performance, even if the I/O cost is bigger. We will call this the filtering condition threshold.

Then, why does this behavior change between smaller tables and bigger tables? There are many possibilities, and all these explanations even if true, can change from query to query, and in the end, how the query planner operates is difficult to fully understand. However, the main reason that we found in this case, is that with bigger data tables the inference of this threshold changes making the query planner make a better decision, and therefore, improving the time. Additionally, in the research that we have done about this topic, we have learned that the query planner usually considers the use of indexes when the table is small and changes to sequential scans with bigger tables. This tendency is related to several reasons that we won't explain in detail, but we will list some of the most important ones.

1. Larger tables contain more data points in the statistics tables and consequently the query planner can make more accurate decisions.
2. The high cost of traversing large indexes.
3. Small indexes can fit in memory often.
4. Bigger and larger joins are more efficient with sequential scan.

*NOTE: In the section regarding the PostgreSQL configuration file we changed the planner cost constants which are in charge of calculating filtering condition threshold*

*in an attempt to increase the use of indexes and random scanning. This is suggested by several blogs and resources since PostgreSQL is based on HDDs which are not good at random scanning. However, since we were using SDDs we did not want our query planner to have this tendency of not using indexes.*

# 9 Conclusion

In conclusion, this project successfully conducted a TPC-DS benchmark using PostgreSQL, providing valuable insights into the database's performance and scalability.

The results demonstrate the effectiveness of PostgreSQL as a robust solution for data warehousing applications after applying proper database optimization. We highlighted the importance of indexing, partitioning, and query optimization techniques that significantly enhance response times and throughput.

To conclude, this benchmark lays also ideas for future explorations, suggesting pathways for improving database performance and for improving automatization and reproducibility.

# References

[1] Ahmad et al. *tpc-ds-benchmark*. Oct. 2022. URL: https://github.com/risg99/tpc-ds-benchmark.

[2] Pavan Patibandla. *How a single PostgreSQL config change improved slow query performance by 50x*. URL: https://amplitude.engineering/how-a-single-postgresql-config-change-improved-slow-query-performance-by-50x-85593b8991b0.

[3] PostgreSQL. *PostgreSQL - Partitioning*. URL: https://www.postgresql.org/docs/current/ddl-partitioning.html.

[4] Hironobu Suzuki. *Cost Estimation in Single-Table Query*. URL: https://www.interdb.jp/pg/pgsql03/02.html.

[5] TPC. *TPC-DS*. URL: https://www.tpc.org/tpcds/.