



ECOLE  
POLYTECHNIQUE  
DE BRUXELLES

Master in Computer Science/Computer Engineering

**INFO-H-419 : Data Warehouses**

---

# TPC-DS SQLITE

---

TALHAOUI Youssef

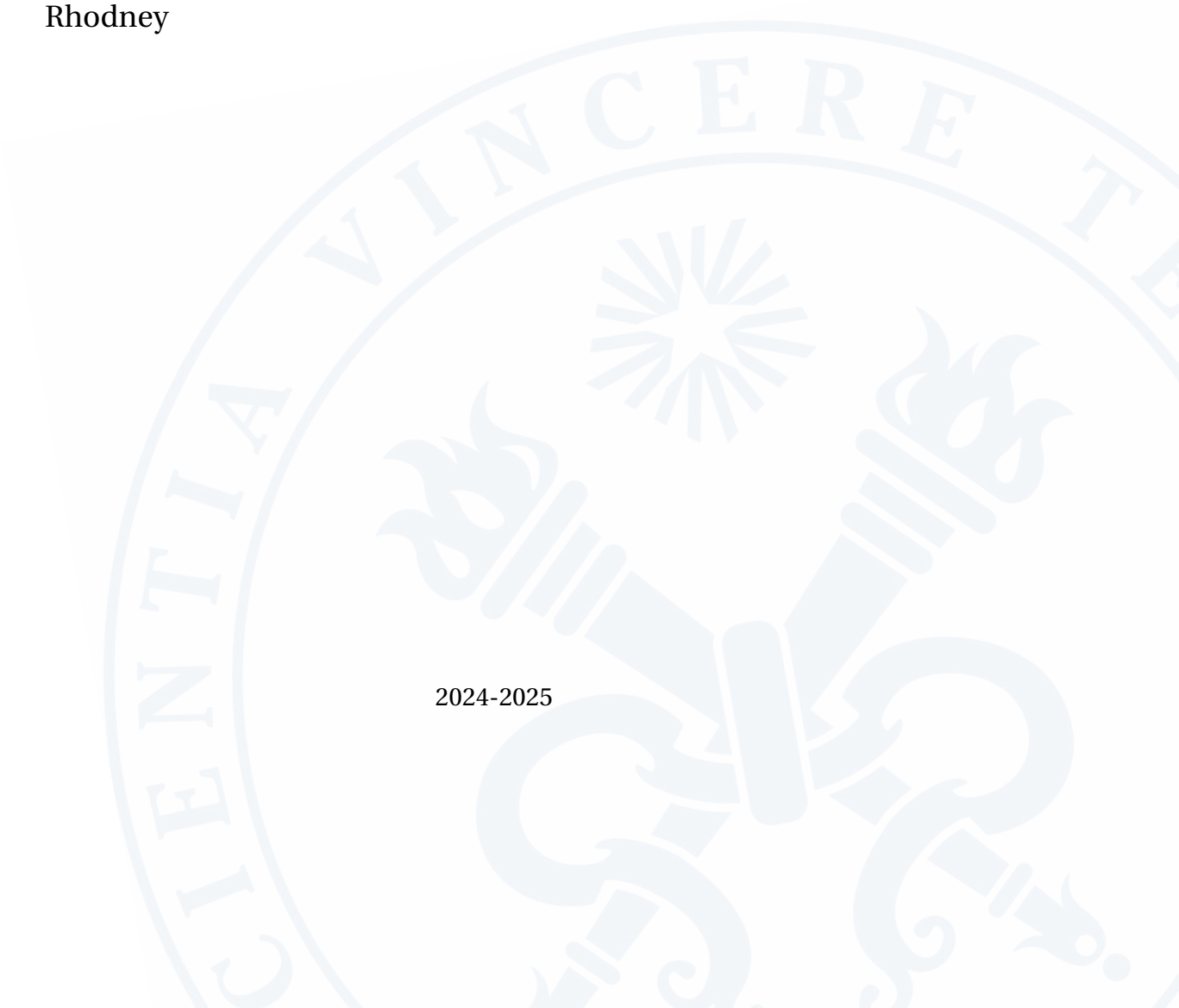
NOUBISSI KAMGANG Allan

MANGUNZA MUAMBA

Rhodney

ZIMANYI Esteban

2024-2025



# Table des matières

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>SQLite</b>	<b>3</b>
2.1	When to use it? . . . . .	4
2.2	Limitations . . . . .	4
2.3	Benchmarking . . . . .	4
2.4	TPC-DS . . . . .	4
2.4.1	Query Summary . . . . .	5
2.4.2	Performance Test . . . . .	5
<b>3</b>	<b>Implementation</b>	<b>7</b>
3.1	Hardware specification . . . . .	7
3.2	Scaling . . . . .	7
3.3	Generating and loading the data . . . . .	8
3.3.1	Data Generation . . . . .	8
3.3.2	Data Loading into SQLite . . . . .	8
3.4	Generating Queries . . . . .	9
3.5	Generating Queries . . . . .	9
3.5.1	Generating Queries Automatically . . . . .	9
3.5.2	Adapting Queries for SQLite Compatibility . . . . .	9
3.6	Power test . . . . .	10
3.7	Throughput test . . . . .	10
<b>4</b>	<b>Results</b>	<b>11</b>
4.1	Load test results . . . . .	11

4.2	Power test result . . . . .	11
4.2.1	Scale factor 1 . . . . .	13
4.2.2	Scale factor 3 . . . . .	14
4.2.3	Scale factor 5 . . . . .	15
4.2.4	Scale factor 7 . . . . .	16
4.3	Throughput test result . . . . .	17
4.4	Comparison between Power test and Throughput test . . . . .	17
4.5	Query Optimization . . . . .	18
4.5.1	Indexing . . . . .	18
4.5.2	Index Test Script . . . . .	19

**5 Conclusions 21**

5.1	Conclusions and Future Work . . . . .	21
5.1.1	Conclusions . . . . .	21
5.1.2	Future Work . . . . .	21

## Table des matières

In this project, we will explore these limitations by benchmarking SQLite's performance using the TPC-DS benchmark. This well-known benchmarking tool will provide insights into SQLite's capabilities and constraints at various data scales, allowing us to assess its suitability as a data warehouse solution for different workload demands.

SQLite is an in-process library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. Its code is in the public domain, meaning it is free for use for any purpose. Many popular applications, including Firefox and Skype, utilize SQLite for their data storage needs, and it is also embedded in the standard libraries of programming languages like PHP and Python.

Its name perfectly summarizes its design philosophy. It is an embedded database management system compatible with SQL, but differs by its lightness. It does not run as a separate server process. Everything is embedded inside one library to facilitate its integration into various applications. Its minimal setup, configuration requirements, and small footprint (often less than 600 KB) are suitable for limited resources environments like mobile apps or embedded systems.

SQLite respects the ACID (Atomicity, Consistency, Isolation and Durability), ensuring reliable transaction processing. It is compatible with multiple operating systems, including Windows, macOS, Linux, iOS, and Android, and it stores all data in a single disk file, which simplifies file management and backup procedures.



FIGURE 2.1: SQLite Logo

## 2.1 When to use it?

iOS and Android mobile apps use SQLite to locally store data inside the user's phone. It allows fast data access, even when there is no network. Since it requires no administration, so it is useful for devices that must operate without expert human support.

SQLite can also be used as the on-disk file format for desktop applications or as the database engine for low to medium traffic websites.

## 2.2 Limitations

Client/Server RDBMS may work better in many cases :

- Client/Server Applications : SQLite can work over a network filesystem, but the latency associated with most network filesystems will downgrade performance. File locking bugs in many network filesystem implementations, allowing two or more clients to modify the same part of the same database simultaneously.
- High-volume websites : SQLite works badly if the website is write-intensive or is so busy that it requires multiple servers.
- Very large datasets : An SQLite database is limited in size to 281 terabytes. Its whole database is stored inside one single disk file. Client/Server is therefore preferable if we consider spreading large content across multiple disk files.
- High concurrency : SQLite supports an unlimited number of simultaneous readers, but it will only allow one writer at any instant in time.

## 2.3 Benchmarking

Benchmarking plays a vital role in Decision Support System (DSS). It will be used to measure response times and data processing speeds. The provided results will help identify areas for improvement in performance and capabilities, and reduce operational costs.

## 2.4 TPC-DS

TPC-DS simulates a data warehouse. It models the decision support functions of a retail product supplier, allowing users to intuitively relate to the benchmark's components.

TPC-DS also uses the concept of scale factors to manage database size. These scale factors represent the estimated data volume in gigabytes. The benchmark results are tied to specific scale factors and cannot be compared with results obtained using different scale factors.

### 2.4.1 Query Summary

The TPC-DS benchmark includes a comprehensive set of 99 queries designed to test the performance of database management systems (DBMS) across various data analytics and reporting scenarios. These queries simulate real-world tasks commonly encountered in business intelligence and data warehousing applications and can be categorized as follows :

- **Reporting Queries :** These queries are used to retrieve and present data from the data warehouse, supporting reporting and analysis tasks. They typically involve operations like aggregation, filtering, grouping, and joining.
- **Ad-hoc Queries :** These queries are generated spontaneously to answer immediate, specific questions. They are characterized by user-driven interactivity and often require quick response times despite their unpredictable nature.
- **Iterative OLAP Queries :** Designed for interactive data exploration, these queries support multidimensional data analysis through operations like slicing and dicing, pivoting, and rolling up.
- **Data Extraction or Mining Queries :** These queries are used to extract valuable insights or patterns from large datasets, focusing on scalability, data analysis, pattern discovery, and algorithms tailored for extracting knowledge from data.

### 2.4.2 Performance Test

The performance test aims to evaluate the speed and efficiency of a system in executing decision-making queries under typical business scenarios. To provide a comprehensive assessment, we will conduct the following tests :

- **Load Test :** This test involves all activities required to configure the system to the state that directly precedes the beginning of the performance test.
- **Power Test :** In this test, the set of 99 queries is executed sequentially by a single simulated user to measure query response times under minimal load.



- **Throughput Test :** This test involves multiple query sessions (Sq), where each session runs all 99 queries concurrently. In our study, we will conduct tests with 4 simultaneous query sessions.

In this chapter, we will go through the various steps involved in the benchmark and the difficulties we experienced.

### 3.1 Hardware specification

When conducting a benchmark, it is essential to decide on the execution environment. We considered two options : running TPC-DS workloads in a cloud environment or on local infrastructure. Cloud platforms, such as Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure, offer scalable and flexible resources ideal for such benchmarks. However, these services operate on a pay-as-you-go basis, which can incur substantial costs depending on resource usage.

To maintain cost-effectiveness and full control over the environment, we opted to conduct the TPC-DS benchmark locally. The following table outlines the hardware specifications of the local setup used for this project :

Operating System	Windows 11
CPU	i5-1235U; 1.3 GHZ up to 4.4; 10 cores; 12 threads
RAM	16 GB DDR4-3200 MHz
Hard Disk	512 GO SSD

TABLE 3.1: Hardware specifications

### 3.2 Scaling

TPC-DS defines default scaling factors that correspond to the size of the raw data generated for analysis. The standard TPC-DS scale factors include 1TB, 3TB, 10TB, 30TB, and 100TB. However,

given the limitations of our hardware (as outlined in the hardware specifications section), running benchmarks at these scales would exceed our available computational resources.

To address this constraint, we have chosen to use smaller scaling factors of 1GB, 3GB, 5GB, and 7GB. This approach allows us to evaluate SQLite's performance at manageable data scales and observe any potential performance degradation as the scale increases.

## 3.3 Generating and loading the data

### 3.3.1 Data Generation

The TPC-DS file includes a data generation program, `dsdgen`. To initially build the binaries, the following steps were followed :

1. Copy `Makefile.suite` to `Makefile`.
2. Edit the `Makefile` and locate the line containing `OS=`.
3. Follow the comments to append your target OS. For example, set `OS=LINUX`.
4. Run `make`.

After completing these steps, data was generated using the command :

```
dsdgen -scale x -dir /tmp
```

where `/tmp` specifies the directory where `dsdgen` was built, and `x` represents the scale of data to be generated.

### 3.3.2 Data Loading into SQLite

To load the TPC-DS data into SQLite databases, data files were first cleaned to remove any trailing delimiters. This was done by iterating through each `.dat` file and removing the extra `|` character at the end of each line, producing a temporary cleaned file which then replaced the original.

For each scale (1GB, 3GB, 5GB, and 7GB), a separate database was created with the TPC-DS schema. The schema setup involved executing SQL commands from the schema file, `tpcds.sql`, to initialize the database structure. Once the database was prepared, data was loaded by importing each `.dat` file into its corresponding table, using appropriate import commands in SQLite.

## 3.4 Generating Queries

The TPC-DS benchmark suite provides a set of standardized templates meant to represent various complex analytical workloads. These templates are used to produce SQL queries compatible with different database dialects. The `dsqgen` utility is a tool provided by TPC-DS to transform these templates into executable SQL queries that match the SQL syntax of the target database.

## 3.5 Generating Queries

The TPC-DS benchmark suite provides a set of standardized templates meant to represent various complex analytical workloads. These templates are used to produce SQL queries compatible with different database dialects. The `dsqgen` utility is a tool provided by TPC-DS to transform these templates into executable SQL queries that match the SQL syntax of the target database.

### 3.5.1 Generating Queries Automatically

To produce a complete set of 99 TPC-DS queries, we utilize a Bash script that processes each query template in a loop. This script calls `dsqgen` with the appropriate settings for each query template, storing the output in SQL files that are compatible with general SQL dialects.

### 3.5.2 Adapting Queries for SQLite Compatibility

Since `dsqgen` does not offer direct support for SQLite, the generated queries need to be adapted to work with SQLite's syntax and functional limitations. Below are the primary modifications necessary to make the TPC-DS queries compatible with SQLite.

#### Replacing Unsupported Clauses like `ROLLUP` and `GROUPING`

SQLite lacks support for clauses such as `ROLLUP` and `GROUPING`, which are often used to produce hierarchical aggregations. To address this, we restructure the query by manually creating equivalent summaries without using these clauses.

#### Replacing Unsupported Statistical Functions

SQLite does not natively support statistical functions like `stddev_samp` (sample standard deviation). Where these functions are required, we either implement custom functions or use alternative formulas to approximate the intended result.

### Replacing Aggregations within Window Functions

Nested aggregations within window functions, such as `sum(sum(...)) over (partition by ...)`, are not supported by SQLite. Queries using these nested operations are rewritten to achieve similar results through alternative approaches.

### Using `LIMIT` in Place of `FETCH FIRST n ROWS ONLY`

Since SQLite does not support the `FETCH FIRST n ROWS ONLY` syntax, we replace it with the equivalent `LIMIT n` clause to restrict the number of rows returned.

### Adjusting Date Arithmetic Syntax

Date arithmetic in SQLite requires specific syntax, such as `date('YYYY-MM-DD', '+N days')`. We replace any instances of SQL-standard date manipulation (e.g., `DATE + INTERVAL`) with equivalent SQLite-compatible expressions.

## 3.6 Power test

In this setup, each query from the TPC-DS set of 99 queries is executed sequentially within a specified timeout limit, with the goal of capturing the total execution time of each query. The timeout limit per query varies across scales, set as follows : 3 minutes for 1GB, 5 minutes for 3GB, 7 minutes for 5GB, and 9 minutes for 7GB. These timeouts ensure that the test can capture individual query performance without exceeding reasonable execution windows for each scale. If a query exceeds the specified timeout, it is terminated, and the timeout duration is recorded as the execution time for that query.

## 3.7 Throughput test

The test is designed to simulate a multi-client environment where several clients execute queries simultaneously, measuring the overall execution time and observing the effects of concurrency on the system's performance. The test is run on each database scale with a fixed number of clients. The previous timeout is kepted for each query. Each client is implemented as a separate process and uses multi-threading. This approach enables monitoring of concurrent query performance under varied data loads, providing insights into the impact of multi-client concurrency.

## 4.1 Load test results

Between 1 and 5 GB, the execution time grows linearly with the amount of data we try to load. It is possible to notice a difference in the slope after the scale of 5 GB. Therefore, we can say that SQLite does not scale optimally and consistently. As it has been said in Chapter 1, SQLite is limited by the size of the dataset it can accept. When the limit has been reached, execution time becomes slower.

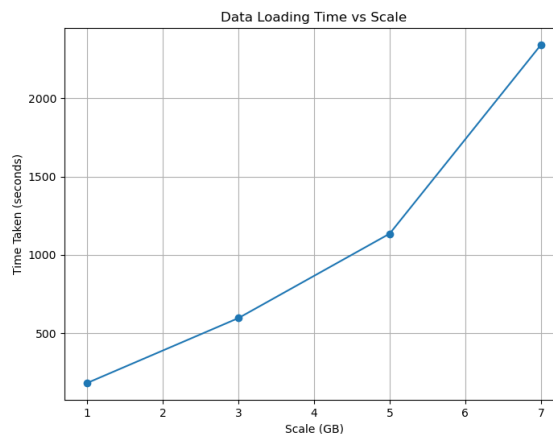


FIGURE 4.1: Evolution of data loading time in function of scale

## 4.2 Power test result

The Figure 4.2 illustrates a linear increase in execution time for the first three scales. However, at the 7 GB scale, there is a noticeable inflection, where the execution time is disproportionately high, significantly deviating from the previous linear trend. This sharp increase suggests potential scalability issues with SQLite under larger data volumes, particularly when handling complex TPC-

DS queries. The jump in execution time at 7 GB may indicate that SQLite's internal mechanisms, such as memory management and indexing, struggle to keep up with the increased data size.

To further understand this deviation, it would be beneficial to examine the execution time of each query across all scales. By analyzing individual query performance, we can identify specific queries or operations that may be disproportionately affected by the larger data size, contributing to the overall increase in execution time at the 7 GB scale.

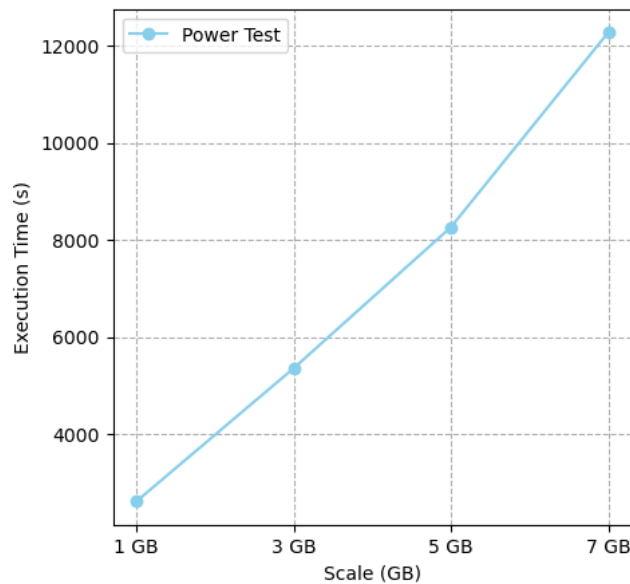


FIGURE 4.2: Evolution of execution time in function of scale

- **Scale 1 :** 4.3 The execution times for most queries are relatively short, with only a few (9) queries time-outed.
- **Scale 3 :** 4.4 An increase in dataset size led to longer execution times across multiple queries. Some queries began to reach the 300-second threshold, indicating performance bottlenecks.
- **Scale 5 :** 4.5 The execution times continued to rise, with more queries hitting or nearing the upper limit of execution time (420 seconds). This suggests that certain queries may not be scaling efficiently.
- **Scale 7 :** 4.6 The largest dataset size tested demonstrated a significant increase in execution times, with many queries either reaching the 540-second threshold or exceeding 300 seconds. This scale factor exposed the queries with the most performance issues.

### 4.2.1 Scale factor 1

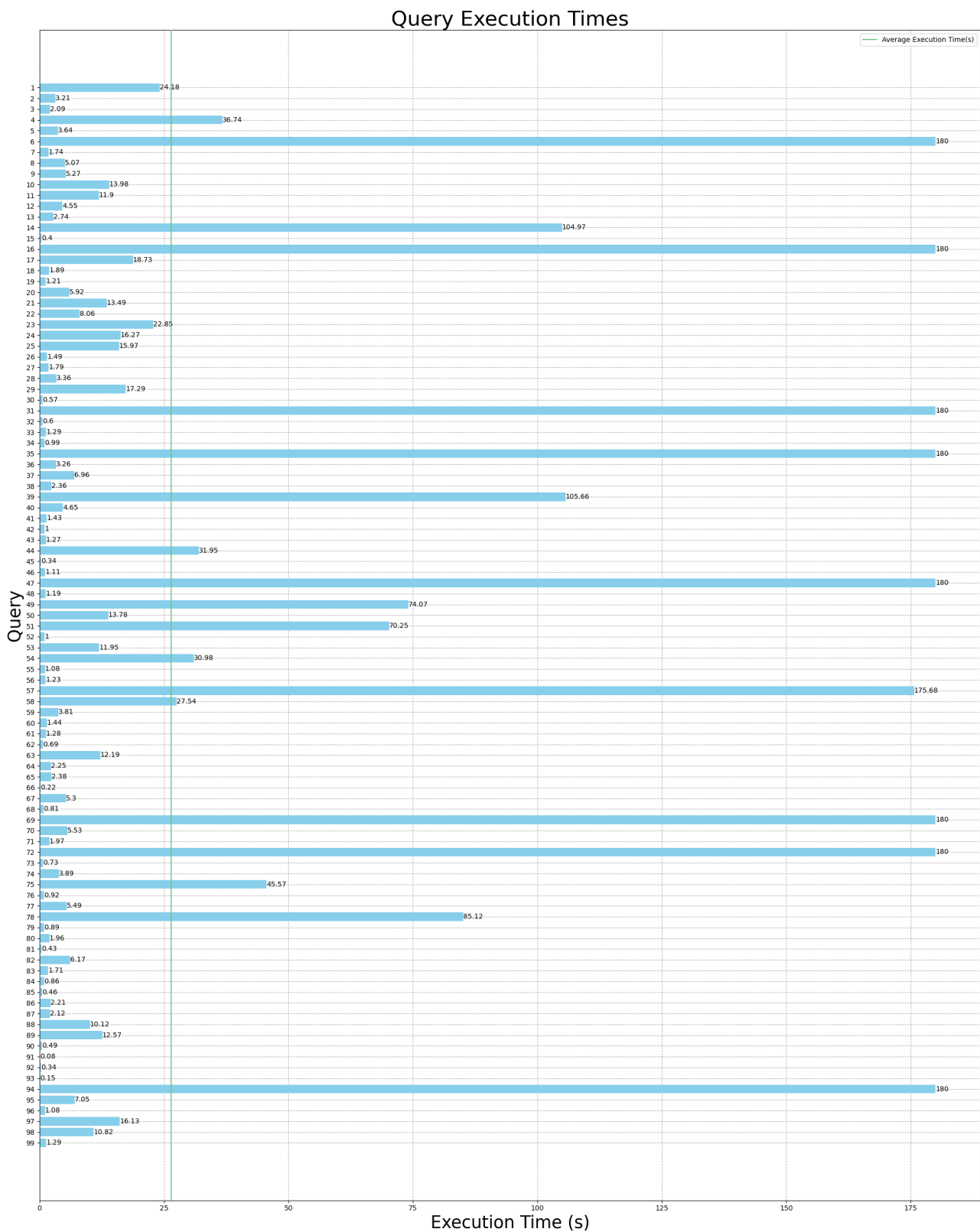


FIGURE 4.3: Execution time in seconds for scale 1.



### 4.2.2 Scale factor 3

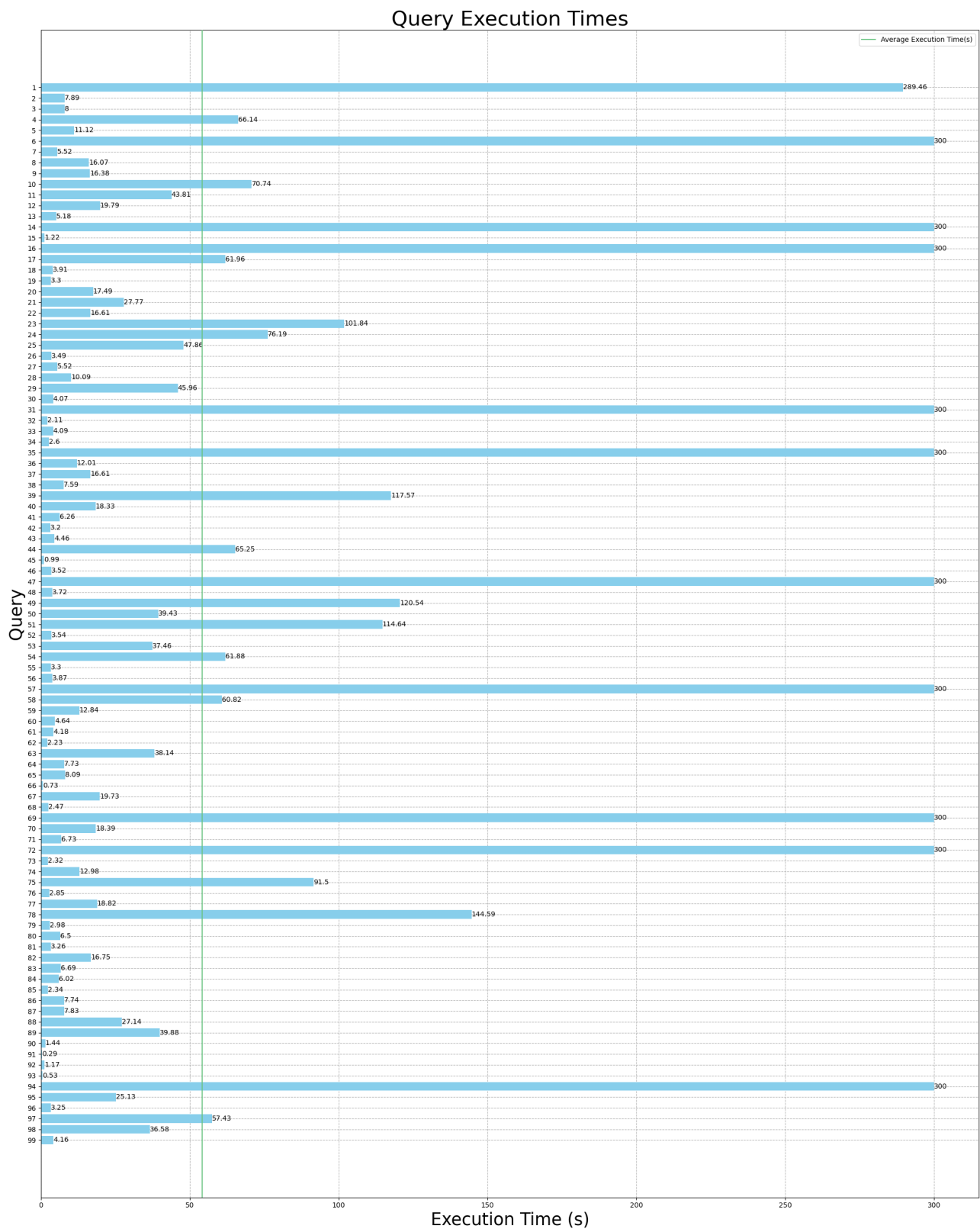


FIGURE 4.4: Execution time in seconds for scale 3.

## 4.2.3 Scale factor 5

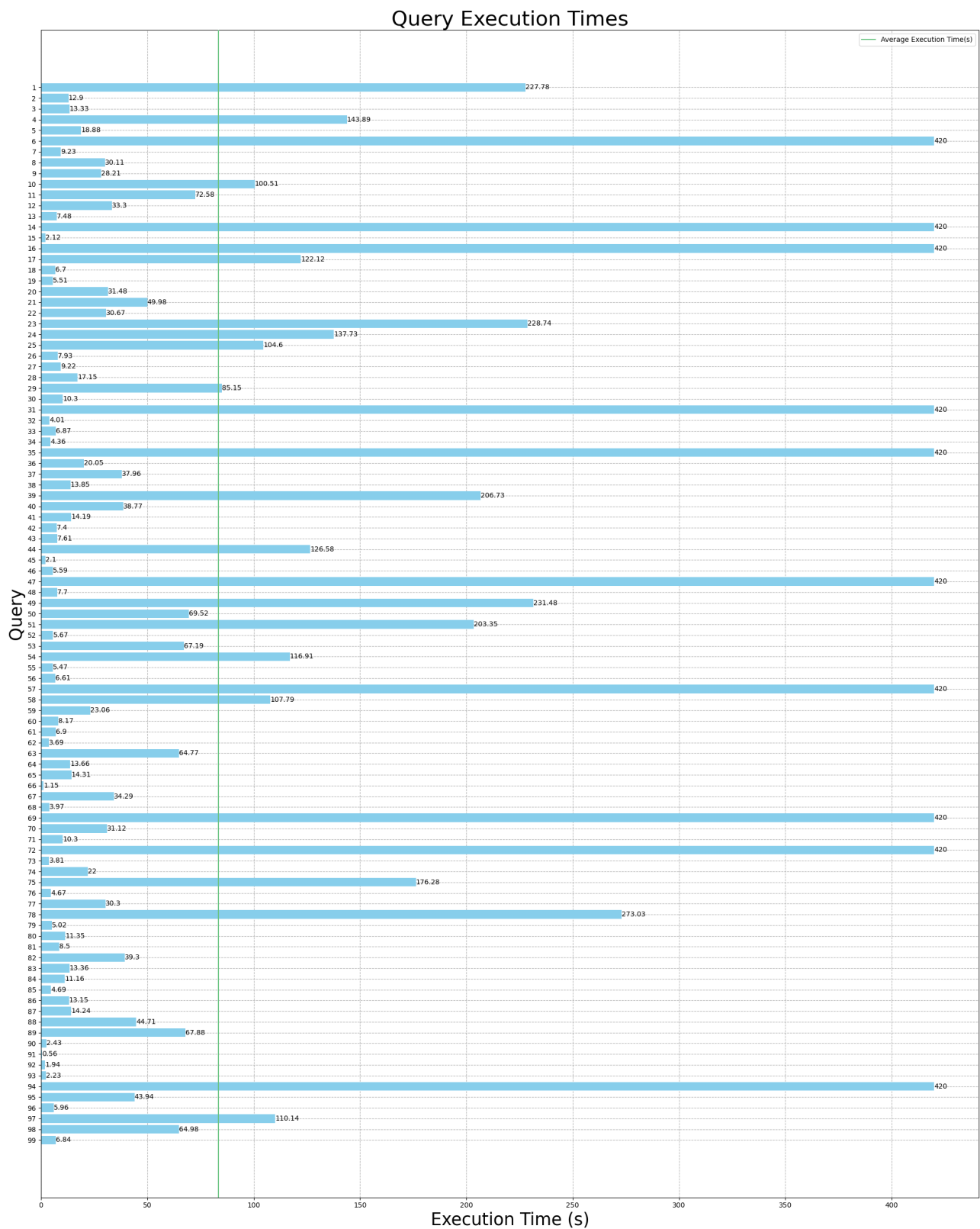


FIGURE 4.5: Execution time in seconds for scale 5.

## 4.2.4 Scale factor 7

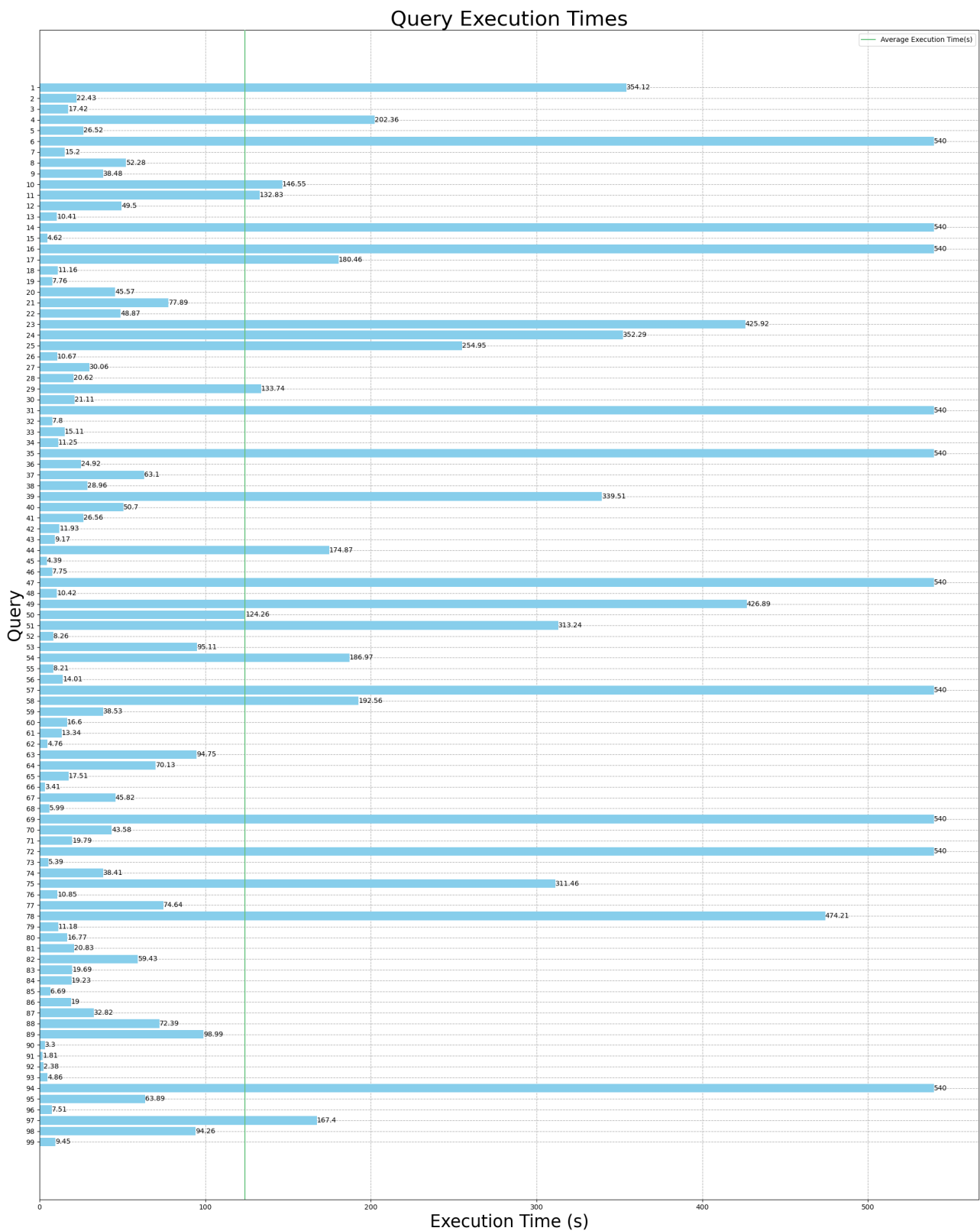


FIGURE 4.6: Execution time in seconds for scale 7.

### 4.3 Throughput test result

The total execution time for the throughput test grows consistently as the data scale increases. This near-linear progression implies that, up to 7 GB, the total time scales predictably with data size. This would generally suggest that SQLite can handle larger data sets under concurrent query loads in a somewhat scalable manner. In reality, the total execution times recorded in the throughput test include cases where queries hit the timeout threshold. Since these timeouts represent the maximum duration recorded for a query, they effectively cap the execution time for particularly long-running or complex queries. As a result, the actual execution time for those queries may be artificially limited to the timeout value rather than reflecting the true performance of SQLite.

**Hypothesis :** If we removed the timeouts, the total execution time might increase at a faster rate as data scales up. In other words, the current linear appearance could be masking performance limitations that would become more pronounced without the imposed time constraints.



FIGURE 4.7: Throughput test total execution time in second for each scale.

### 4.4 Comparison between Power test and Throughput test

The figure 4.9 and 4.8 shows that the Throughput test consistently displays higher execution times compared to the Power Test, which can be attributed to SQLite's handling of concurrency. Both tests demonstrate an exponential trend as the data scale increases. This exponential growth suggests that SQLite struggles to manage large datasets, even in a single-session context. These results suggest that while SQLite might be efficient for lightweight or single-user applications, it is not optimized for large, concurrent workloads.

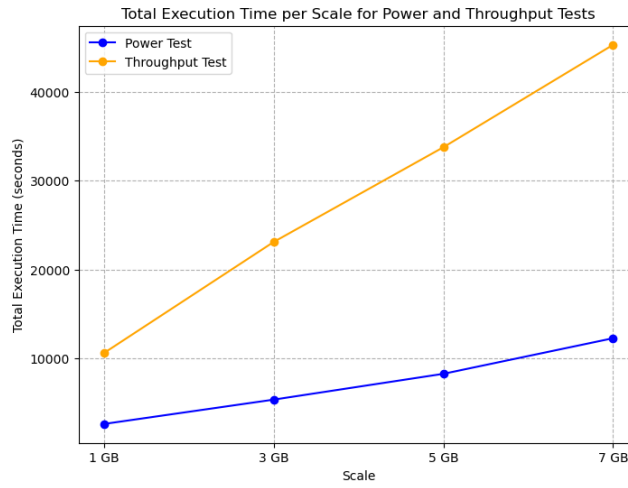


FIGURE 4.8: Comparison between the total execution time for the power test and the throughput test.

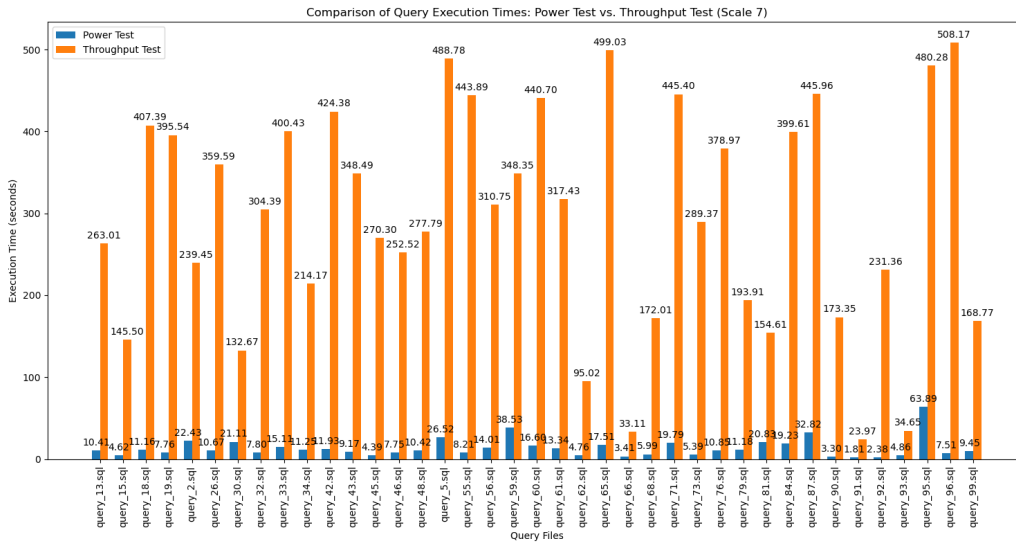


FIGURE 4.9: Comparison of Query Execution Times : Power Test vs. Throughput Test (Scale 7).

## 4.5 Query Optimization

### 4.5.1 Indexing

To enhance query performance on large tables, particularly for queries involving complex joins or aggregations, we introduced indexing to reduce query execution time. Indexes enable SQLite to access rows more efficiently by maintaining a sorted structure for specific columns, which is especially advantageous in high-volume datasets like those in TPC-DS.

**Example :** For queries that involve frequent joins on `customer_id` or filtering on `sales_date`, we created composite indexes on these columns to expedite the retrieval process. In the `store_sales` table, for instance, we created an index on `(customer_id, sales_date)` as follows :

```
CREATE INDEX idx_store_sales_customer_date
ON store_sales(customer_id, sales_date);
```

This indexing approach notably improved query performance, especially for joins between `store_sales`, `customer`, and `date_dim` tables, which would otherwise have required full table scans.

### 4.5.2 Index Test Script

To measure the effectiveness of these indexes, we used a Python script named `index_test`. This script runs each query twice — once without applying indexes and once with indexes applied — allowing us to compare the performance differences for 1 GB. By setting a timeout, we ensured that the queries had sufficient time to complete within a practical threshold.

**Results :** Below are two plots illustrating the execution time of queries with and without indexes. The comparison shows that indexing significantly reduces the execution time, especially for queries that involve large tables.

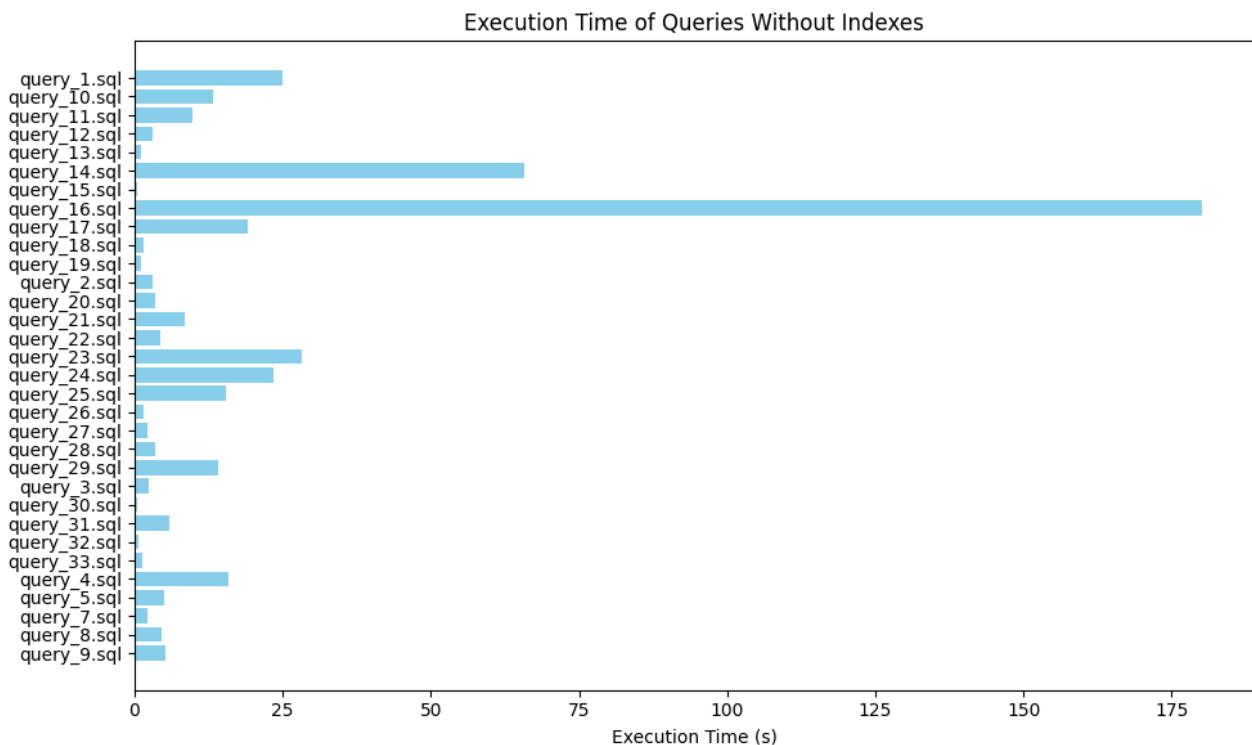


FIGURE 4.10: Execution Time of Queries Without Indexes

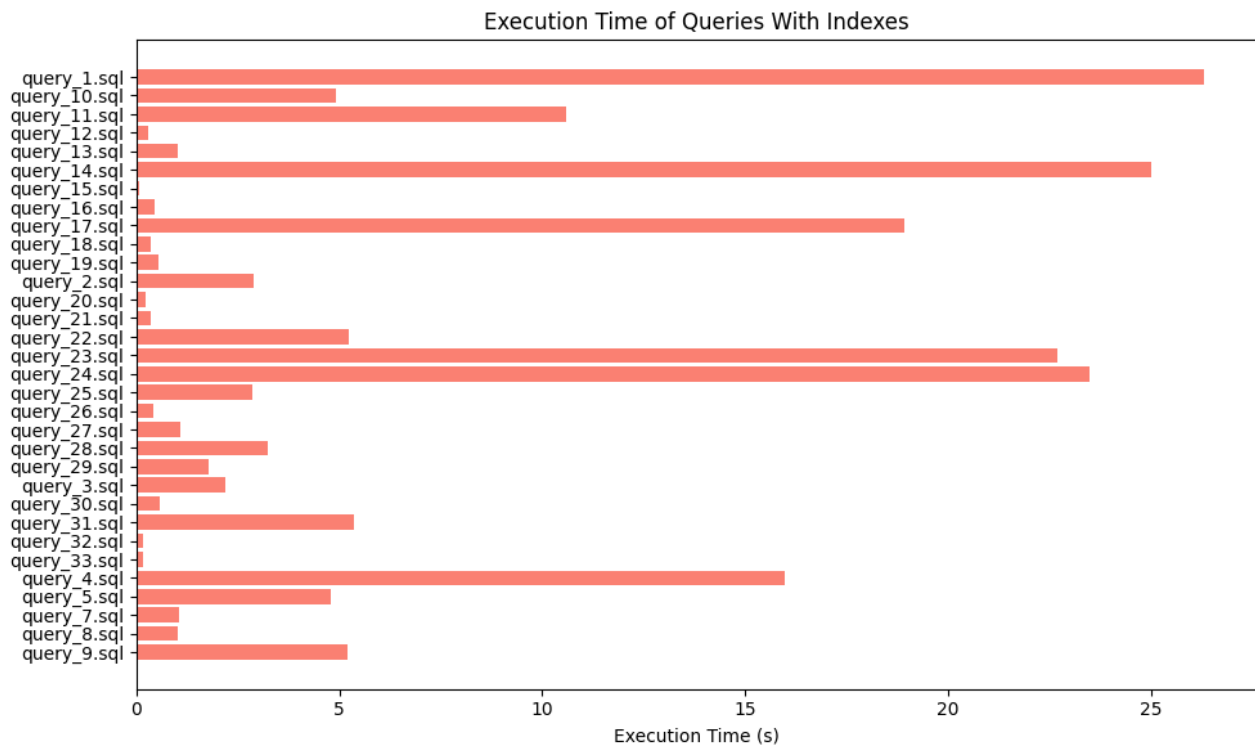


FIGURE 4.11: Execution Time of Queries With Indexes

These plots clearly demonstrate the drastic improvement in execution times, particularly for queries that interact with large tables. This confirms that indexing is a valuable optimization strategy in large-scale query execution.

These plots clearly demonstrate the drastic improvement in execution times, particularly for queries that interact with large tables. This confirms that indexing is a valuable optimization strategy in large-scale query execution.

## 5.1 Conclusions and Future Work

### 5.1.1 Conclusions

This project focused on evaluating the performance of SQLite when executing TPC-DS benchmark queries, with an emphasis on its scalability and overall efficiency. The results from the load, power, and throughput tests showed that SQLite's performance scales linearly with increasing data sizes up to a certain point. However, significant challenges were observed beyond the 5 GB scale, particularly at 7 GB, where execution times deviated from the linear trend and showed considerable increases. This indicates that while SQLite is effective for smaller-scale or lightweight applications, it struggles with larger, more complex datasets.

While the primary objective was to assess SQLite's handling of data scalability, we also briefly examined query optimization techniques. The introduction of indexing was shown to enhance performance. This demonstrates that although SQLite is not inherently optimized for extensive, concurrent workloads, targeted optimizations like indexing can yield noticeable improvements in query execution.

### 5.1.2 Future Work

The next phase of this project involves implementing the TPC-DI benchmark to assess SQLite's ability to handle data integration tasks—an essential component of modern data warehousing. This will provide deeper insights into SQLite's performance during data extraction, transformation, and loading (ETL) processes and uncover any additional optimizations needed for such workflows.

For more details and access to the code used in this project, please refer to the GitHub reposi-



tory.