ADS 2024, Project Assignment 2

**Adding a New Benchmark to DuckDB**

Github : <https://github.com/MaCoHa/duckdb/tree/main/benchmark/ssb>

Contents

[Answers to questions 2](#_Toc182749630)

[SSB implementation 2](#_Toc182749631)

[Experimental Methodology and Results 3](#_Toc182749632)

[Performance testing with varying threads 3](#_Toc182749633)

[Setup 3](#_Toc182749634)

[Results 4](#_Toc182749635)

[Performance testing with varying scale factors 4](#_Toc182749636)

[Setup 4](#_Toc182749637)

[Results 5](#_Toc182749638)

[Discussion of Results 5](#_Toc182749639)

[Performance testing with varying threads 5](#_Toc182749640)

[Performance testing with varying scale factors 6](#_Toc182749641)

[Conclusion 6](#_Toc182749642)

[Appendix 6](#_Toc182749643)

# Answers to questions

**Q1.** *What are the options for performance analysis and profiling in DuckDB? Does using mode detailed profiling options increase query runtime?*

Figure 1 benchmarks with and without measurement flags

Duckdb have built in a benchmark tool which allows users to run several different database benchmarks. This tool always measures the elapsed time of each benchmark but by adding the flag ***–profile*** or ***--detailed-profile*** the tool provides more performance measurements of each query. ***--profile*** provides a smaller overview while ***--detailed-profile*** provides all measurements. But do the different flags increase the runtimes. I did a small experiment where I ran the same 3 queries with no flags, ***–profile*** and ***--detailed-profile***. As can be seen on Figure 1 the runtime is not increased but decreases. I ran this multiple times, and all showed similar results. I would expect the times to be equal or a bit slower, since it needs to record the measurements which in theory should impact performance some way.

**Q2.** *What are the ways to evaluate standardized benchmarks in DuckDB? What are their pros / cons in terms of implementation complexity and benchmark flexibility?*

# SSB implementation

For my implementation of SSB I went for a similar approach as the tpcds benchmark. In my SSB folder I have 4 sub folders (**sf1**,**sf10**,**sf100** and **querys**). **querys** have each of the SSB sql queries stored in separate SQL files. **sf1**,**sf10** and **sf100** are all structured similar with a **data** and **answers** folder, an **ssb\_sf1.benchmark.in** (benchmark.in) file and all benchmarks in separate files.

Each benchmark files holds a reference to the query it should perform and the scale factor it should run, it then calls the benchmark.in with those parameters. benchmark.in takes the parameters and creates the needed tables and then loads the data from the local **data** folder into these tables. When that is done it finds the correct query from the **query’s** folder**.** It then compares the query results to the belonging answer from the **answers** folder.

The **data** folder inside each of the folders **sf1**,**sf10** and **sf100**, contains csv files with the data for each table for the SSB. The data have different scale factors depending on the main folder I belongs to. The **data** folder also holds an **load.sql** script which creates the tables and the loads the local csv data into those tables. To generate the data I used an SSB-dbgen[[1]](#footnote-1) which was provide. I tried to generate the data each time a benchmark was run but it kept chasing so instead in pre-generated the data with the correct scale factors and loaded the data each time instead.

The **answers** folder just holds csv files with the correct for each query, which just ensures the benchmark loaded and ran the query correctly.

I pick this approach because I felt it made getting an overview and making changes to any of the benchmarks easy. It is also easily expanded by adding more benchmarks or adding an additional scale factor. If I could have preferred to have the SSB-dbgen generate the data each time so the benchmarks could just be uploaded and work without having to add the data manually to each data folder.

# Experimental Methodology and Results

In this section I cover my different experiments on some of the SSB benchmarks that I created in Duckdb. I have pick 3 queries from the SSB benchmarks which I refer as query 1,7 and 12 they are equivalent to Q1.1,  Q3.1 and Q4.2 from the clickhouse[[2]](#footnote-2) examples for SSB benchmarks.

All benchmarks’ experiments were run on my own personal laptop. My laptop have a windows OS but hosted the Duckdb on a ubonto wsl.

|  |  |
| --- | --- |
| OS | Microsoft Windows 11 Home |
| CPU | Intel(R) Core(TM) i7-10750H CPU @ 2.60GHz |
| Cores / Logical processors | 6 cores / 12 processors |
| L1 cache / L2 cache / L3 cache | 384 KB / 1,5 MB / 12 MB |
| RAM / WSL RAM | 16 GB / 7.7Gi |
| Laptop model | GF65 Thin 10SER |

## Performance testing with varying threads

In this performance test I tested what would happen to SSB query 1,7 and 12 when they ran with a scale factor 1 but where the number of threads the query could use varied.

### Setup

Since I had implemented the SSB benchmark into the Duckdb benchmark tool I just used it to measure and collect the data. When running the benchmarks I used the ***--detailed-profile*** flag and the ***--log*** and ***--out*** to store the measurement results into files. This meant that for benchmarking I ran with the following permutations

**Repetitions:** Duckdb runs all benchmarks 5 times.

**Scale factor:** scale factor 1

**Query’s:** query 1, 7 and 12

**Number of threads:** 1, 4 and 8

### Results

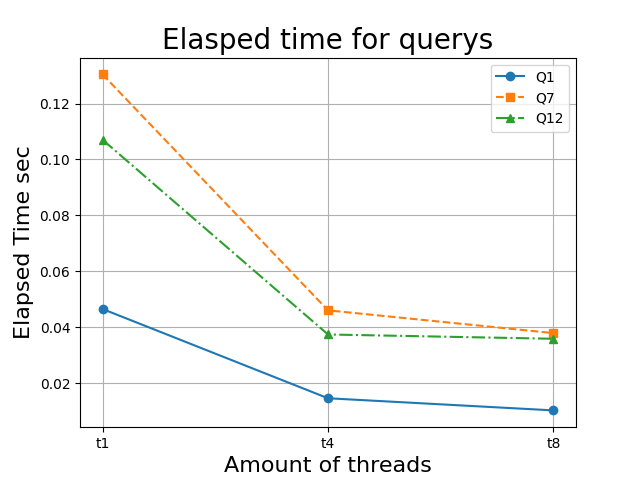


Figure 2 show the elapsed time of 3 queries SSB benchmarks with varying number of threads

From some of the data collected I plotted 3 different graphs. Figure 2 show the average elapsed time of each query with varying number of threads. Figure 3 show the average cumulative cardinality for each query with varying number of threads. Figure 4 show the average cumulative rows scanned for each query with varying number of threads

## Performance testing with varying scale factors

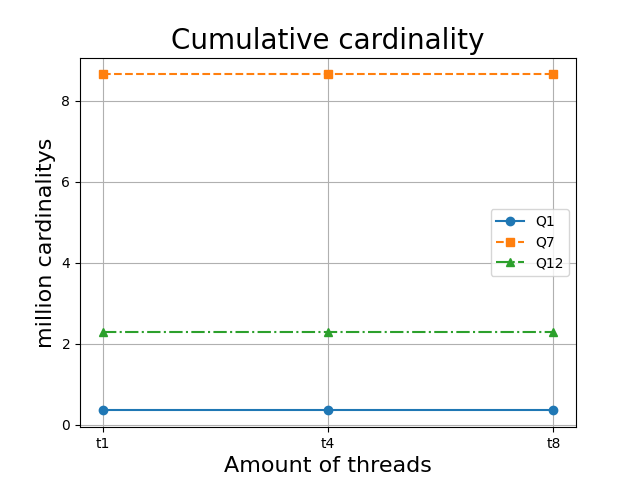


Figure 3 show the cumulative cardinality used by each query when running on different threads.

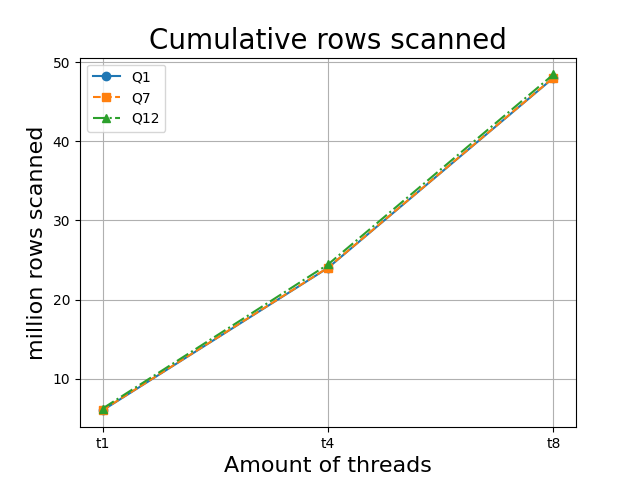


Figure 4 shows the cumulative rows scanned by each query when running on different threads.

In this performance test I tested what would happen to SSB query 1,7 and 12 when they with different scale factors.

### Setup

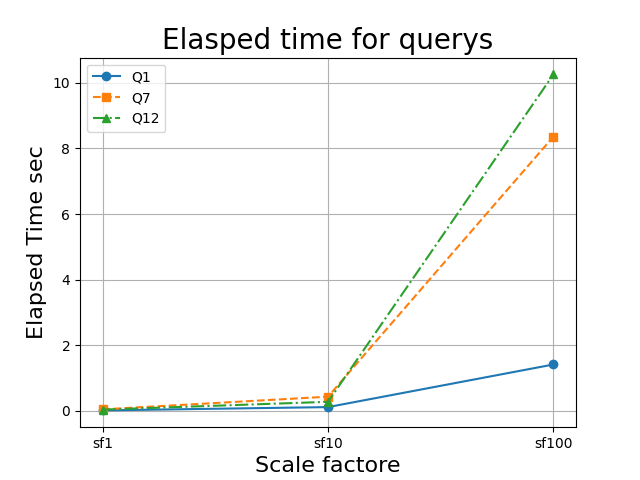


Figure 5 shows elapsed time of each query with different scale factors.

Since I had implemented the SSB benchmark into the Duckdb benchmark tool I just used it to measure and collect the data. When running the benchmarks I used the ***--detailed-profile*** flag and the ***--log*** and ***--out*** to store the measurement results into files. This meant that for benchmarking I ran with the following permutations

**Repetitions:** Duckdb runs all benchmarks 5 times.

**Scale factor:** scale factor 1, 10 and 100

**Query’s:** query 1, 7 and 12

**Number of threads:** 4

### Results

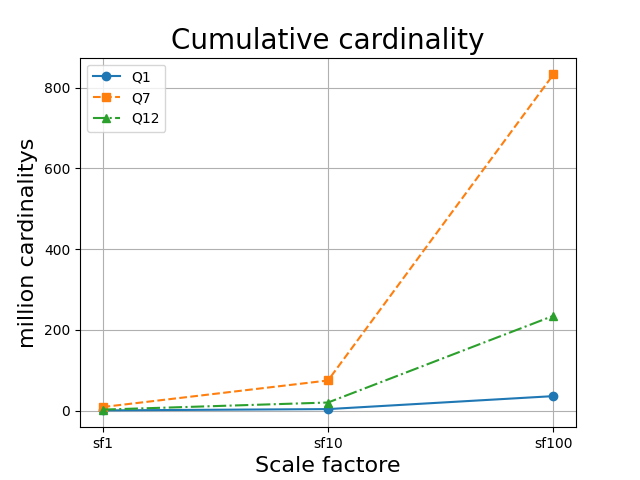


Figure 6 shows cumulative cardinality of each query with different scale factors.

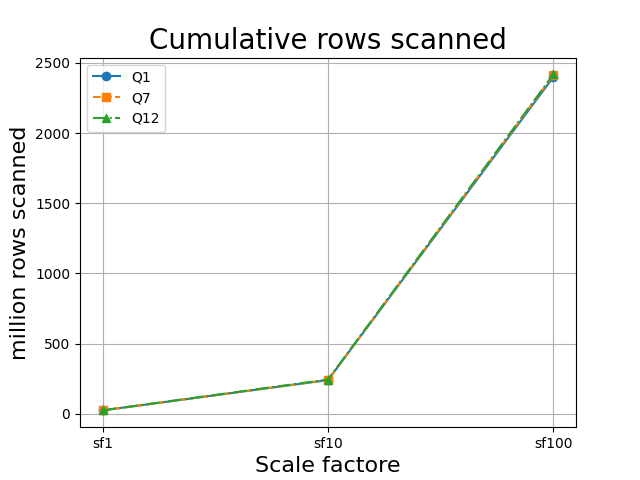


Figure 7 shows the cumulative rows scanned with different scale factors.

From some of the data collected I plotted 3 different graphs. Figure 5 show the average elapsed time of each query with varying scale factors. Figure 6 show the average cumulative cardinality for each query with varying scale factors. Figure 7 show the average cumulative rows scanned for each query with varying scale factors

# Discussion of Results

## Performance testing with varying threads

By looking at the results of figure 2 we can see that as the amount of threads increase the elapsed time of the queries decreases. This is to be expected since they can share the workload and decrease the processing time.

On figure 3 we see the cardinality remains the same when the threads increase, this is also expected since the query should return the same results no matter the amount of threads.

But on figure 4 increase in the cumulative rows scanned based on the number of threads used. By looking into the data a bit more it becomes clear that 1 thread cumulative rows scanned \* 4 is almost equal to 4 thread cumulative rows scanned and the same pattern with 8 threads cumulative rows scanned. This holds for all 3 queries. By looking deeper into the data the increase in rows scanned happens in the different joins in each query. What I think is happening is that each thread gets x number of rows that it needs to join to some rows in lineorder, but when It joins it still have to run through the entire table lineorder to find the rows. Since we then have 1/4/8 number of threads doing the same scan we see a linear increase in the cumulative rows scanned based on the number of threads.

## Performance testing with varying scale factors

By looking at figure 5 we can see that the elapsed time of each query begins variere for the others as the scale factor increases. We can also see on figure 7 that the cumulative rows scanned of each remain almost the same as the other queries as the scale factor increases.

# Conclusion

Summarize your main findings and learnings.

# Appendix

1. https://github.itu.dk/pito/ads2024-ssb-dbgen [↑](#footnote-ref-1)
2. https://clickhouse.com/docs/en/getting-started/example-datasets/star-schema [↑](#footnote-ref-2)