Adding a New Benchmark to DuckDB

Veron Hoxha (veho@itu.dk)
IT University of Copenhagen
November 25, 2024

1 Introduction

This project details the work to become familiar with DuckDB, an in process data analytics system. The Star Schema Benchmark (SSB) was implemented as a standardized benchmark in DuckDB, and experiments were conducted to analyze the scalability of the queries and their operators with respect to data size and thread count.

In Section 2, we describe the performance analysis and profiling options in DuckDB . In Section 3, we present the process of evaluating standardized benchmarks in DuckDB. In Section 4, we explain the SSB implementation. Section 5 explains the experimental methodology of the project. Section 6 presents the results together with the discussion. Finally, in Section 7, we summarize our findings.

Detailed project codes and additional resources are available on GitHub at `https://github.com/veronhoxha/adding-a-new-benchmark-to-duckdb`.

2 Performance Analysis and Profiling Options in DuckDB

DuckDB is a relational Database Management Systems (DBMS) that supports the Structured Query Language (SQL) and provides robust tools for performance analysis and profiling. These tools are designed to optimize SQL query performance and offer deep insights into how queries are executed, making them essential for database operation optimization and understanding DuckDB [1].

Profiling Commands DuckDB makes profiling possible through the use of `PRAGMA` commands:

- `PRAGMA enable_profiling;` - Activates profiling to collect detailed performance data for each query executed within a session.
- `PRAGMA disable_profiling;` - Deactivates profiling, useful for reducing overhead when profiling is not needed.

Profiling Modes Profiling in DuckDB can be customized according to the level of detail required/needed:

- The default mode is *standard*, which provides a basic level of profiling.
- For more granular insights, the *detailed* mode can be enabled with `SET profiling_mode = 'detailed';`.

Additionally, DuckDB allows users to specify the format of the profiling output, supporting formats such as `query_tree`, `json`, `query_tree_optimizer`, and `no_output`. The default setting is `query_tree`.

Query Plan Analysis The `EXPLAIN` command is used to display the query execution plan, whereas `EXPLAIN ANALYZE` provides runtime performance metrics, making it an invaluable tool for detailed performance analysis.

2.1 Impact of Detailed Profiling on Query Runtime

Detailed Profiling Mechanics When profiling mode is set to *detailed*, the output of this mode includes profiling of the planner and optimizer stages. To empirically determine whether detailed profiling impacts query runtime, a series of SQL commands were executed within the DuckDB CLI for a reason of experimenting:

```sql
1 create table table1 (i integer, j double);
2 insert into table1 (i, j) select k, random() from range(1000000) tbl(k);
3
4 -- Set the appropriate profiling mode: 'standard' or 'detailed'
5 -- e.g., SET profiling_mode = 'standard';
6 -- or SET profiling_mode = 'detailed';
7
8 -- Query 1
9 EXPLAIN ANALYZE
10 SELECT a.*
11 FROM table1 a
12 WHERE a.j > 0.5 AND a.i IN (SELECT i FROM table1 b WHERE b.j < 0.5);
13
14 -- Query 2
15 EXPLAIN ANALYZE
16 SELECT MOD(a.i, 10) AS group_id, AVG(a.j) AS average_j, COUNT(*) AS count
17 FROM table1 a
18 WHERE a.j > 0.5
19 GROUP BY group_id
20 HAVING COUNT(*) > 100;
21
22 PRAGMA disable_profiling;
```

Starting from line 4 in Listing 2.1, the code snippet is executed twice: once under detailed profiling and once under standard profiling. Each run uses the `EXPLAIN ANALYZE` command to monitor query performance. The results of these runs are documented in Table 1.

The results in Table 1 indicate a slight increase in execution times when detailed profiling is enabled, reflecting the trade off between obtaining in depth performance insights and maintaining query efficiency. This is particularly relevant in production environments where performance is critical.

	Query 1		Query 2	
	Standard	**Detailed**	**Standard**	**Detailed**
Total Time	0.0286s	0.0302s	0.0081s	0.0091s

Table 1: Comparison of query runtimes under standard and detailed profiling settings.

3 Evaluation of Standardized Benchmarks in DuckDB

Evaluating standardized benchmarks in DuckDB, such as TPC-H and Clickbench, is crucial for assessing the performance and scalability of this database system. These benchmarks provide a systematic approach to measuring how DuckDB handles different types of data operations and query loads.

Evaluation Methods Standardized benchmarks in DuckDB are configured using specific benchmark scripts that order data loading, query execution, and result validation. These scripts are highly structured, allowing for reproducible tests that are consistent across different environments. The benchmarks typically involve several steps:

- Data generation and loading using predefined schemas and data distributions.
- Execution of a series of queries that reflect common and complex operations in database management.
- Collection and comparison of results against known outcomes to ensure accuracy.

Pros and Cons The implementation of standardized benchmarks in DuckDB, while providing a robust framework for performance evaluation, comes with its own set of advantages and challenges. Table 2 summarizes the key pros and cons related to implementation complexity and benchmark flexibility.

4 SSB implementation

The implementation of the Star Schema Benchmark (SSB) in DuckDB can be approached in various ways, one of which involves integrating directly into the DuckDB code base. For this project, however, I chose to use DuckDB's Python API for its straightforwardness and ease of use [2]. This method uses Python's broad range of tools and DuckDB's effective API, making it easy and powerful to operate the database.

I based my implementation on the ClickHouse version of the Star Schema Benchmark (SSB, 2009) [3], which provided a comprehensive framework for generating data, as well as defining tables

Aspect	Pros	Cons
Implementation Complexity	Standardized scripts make it possible for consistent and automated testing procedures, reducing human error and increasing reliability.	The initial setup and configuration of these benchmarks can be complex and time consuming, requiring detailed understanding of both DuckDB and the benchmark specifications.
Benchmark Flexibility	Templates and parameters can be adjusted to simulate different scales and loads, making the benchmarks adaptable to various testing needs.	Modifications to benchmarks or their configurations might require deep technical knowledge, limiting flexibility for users not familiar with the internal workings of DuckDB or the benchmark structure.

Table 2: Pros and Cons of implementing standardized benchmarks in DuckDB.

and queries. Although the original benchmark specifications were designed for ClickHouse, I adapted them to fit DuckDB's context. This adaptation involved modifying data type when creating tables to align with those supported by DuckDB, ensuring that the benchmark would run correctly and efficiently on this platform.

Due to limited memory on my hardware platform from another heavy large project on data, I selected scaling factors (SF) of 1, 5, and 10. This choice helps manage the data volume within my current hardware constraints while still obtaining useful insights into DuckDB's performance. The following command was used to generate the data:

```
./dbgen -s [SF] -T a
```

For each SF, a unique DuckDB database instance was established. This setup process involved systematically creating specific tables, namely customer, supplier, part, date, and lineorder within each distinct database. Data corresponding to each SF was then loaded into these tables. The entire procedure was executed using SQL commands through DuckDB's Python API, ensuring that each database was customized to reflect the different data volumes associated with its respective SF.

By using DuckDB's Python API and the ClickHouse version of SSB, I was able to implement the SSB in a manner that was both straightforward and efficient, enabling detailed performance analysis under different data scales and computational settings. This method proved to be exceptionally adaptable which was the reason this particular approach was picked by my side.

5 Experimental Methodology

Hardware specification The performance tests were conducted on an Apple M3 Pro chip with 11-core CPU, 14-core GPU, 16-core Neural Engine, 18GB unified memory and 512GB SSD storage.

The operating system used was macOS Sequoia 15.0.

SSB Queries Analyzed I selected three queries from ClickHouse [3]:

- **Query 1 (Q4.3):** Focuses on aggregating profit across multiple joined tables.
- **Query 2 (Q1.2):** Assesses performance on simple aggregations with conditional filtering.
- **Query 3 (Q3.1):** Tests data grouping and sorting by geographical and time dimensions.

These queries were chosen randomly to test various aspects of the database's performance. For a full view of the queries, refer to the code in my GitHub repository which can be found in Section 1 or visit the ClickHouse link [3] and search for the query names mentioned in parentheses in List 5.

Profiling options To profile the performance of queries, DuckDB's built-in **EXPLAIN ANALYZE** command was used. This command provides detailed insights into query execution plans and runtime statistics, including execution time and the cost of different query operations.

Number of repetitions of the runs Each query was executed five times to ensure the reliability of the results. This number of repetitions aligns with DuckDB's default setting for running benchmarks and helps mitigate any variations in performance.

Scaling Factors and Thread Counts The experiments were conducted with SF of 1, 5, and 10 to explore how increasing data sizes impact performance. Concurrently, thread counts of 1, 4, and 8 were tested to examine the effects of parallel processing capabilities in DuckDB. Each query was executed across all SF with a thread count of 1, and separately, all thread counts with a SF of 10.

6 Results and Discussion of Results

As specified in Paragraph 5, each query was executed five times to ensure reliability and to mitigate variations in performance. The average runtimes presented in Tables 3 and 4 were calculated from these executions and are expressed in seconds. It should be noted that slight variations in these times may be observed each time the experiments are conducted, attributable to fluctuations in system performance.

Query Performance by Thread Count Table 3 illustrates the effect of increasing thread counts on the execution times of three distinct queries.

	Query 1			Query 2			Query 3		
Threads	**1**	**4**	**8**	**1**	**4**	**8**	**1**	**4**	**8**
Average Time	0.6174s	0.1250s	0.0871s	0.2786s	0.0613s	0.0409s	0.7396s	0.2006s	0.1540s

Table 3: Average execution times for each query after five runs across different thread counts with SF = 10.

Query Performance by SF Table 4 demonstrates how query execution times escalate with increasing SF.

	Query 1		Query 2		Query 3	
SF	**1**	**5**	**1**	**5**	**1**	**5**
Average Time	0.0674s	0.3154s	0.0333s	0.1634s	0.0864s	0.4106s

Table 4: Average execution times for each query after five runs across different SF's with thread count = 1.

Due to the large amount of data from query profiling, it's not possible to also show here the detailed results of the query operators of those queries. You can find these details in the `results` folder on the GitHub repository link found in Section 1. A detailed analysis of these results is discussed in Section 6.1.

6.1 Query Performance Analysis

Analysis of Query 1 performance The performance analysis of Query 1 reveals that `TABLE_SCAN` and `HASH_JOIN` are the primary operations affected as scaling factors increase. The `TABLE_SCAN` operation, which reads a significant number of rows, becomes increasingly time consuming with larger amount of data, illustrating the direct impact of data volume on scan operations. Despite `HASH_JOIN` maintaining relatively quick execution times, its efficiency is overshadowed by the more time intensive table scans required for larger datasets.

Analysis of Query 2 performance In Query 2, the `TABLE_SCAN` again stands out as the most impacted operation with increasing data scales, consuming more time as more rows are processed. This trend demonstrates the linear relationship between data volume and the time required for scan operations. Additionally, `HASH_JOIN` also shows longer execution times with larger data, reflecting the increased computational load as more data undergoes join operations. This increase is particularly notable when the thread count rises, suggesting that while parallelism introduces some efficiency, it also brings additional overhead from synchronization.

Analysis of Query 3 performance For Query 3, similar patterns happen, with `TABLE_SCAN` and `HASH_JOIN` being the critical operations influencing execution time. As the data volume increases, both operations take longer, indicating the scalability challenges faced when handling larger amounts of data in complex queries involving multiple joins and group by operations.

How does the query performance change as you increase the number of threads running the query? Significant improvements in performance are observed as the number of threads increases for all queries as seen in Table 3. For instance, Query 1's execution time drops from ≈0.6174 seconds with 1 thread to ≈0.0871 seconds with 8 threads. Query 2 shows a reduction from ≈0.2786 seconds to ≈0.0409 seconds, and Query 3 from ≈0.7396 seconds to ≈0.1540 seconds when changing from 1 to 8 threads. This demonstrates the benefits of parallel processing in reducing the execution time of queries as more threads are used.

How does the query performance change as you increase the size of the database? A trend of increasing execution times is consistent across all queries as the size of the database grows as shown in Table 4. For Query 1, as the scaling factor (SF) increases from 1 to 5, the total time escalates from ≈0.0674 seconds to ≈0.3154 seconds. Query 2 also shows an increase in execution time, rising from ≈0.0333 seconds at SF=1 to ≈0.1634 seconds at SF=5. Similarly, Query 3 sees an increase in execution time as the scaling factor changes from 1 to 5.

Are these results expected? Why or why not? Yes, the observed results are in line with expectations for database performance. As databases grow in size, the time required to process queries increases correspondingly. This phenomenon is due to the larger volumes of data that must be processed, evident in the detailed analysis, where operations such as table scans and joins become progressively slower as data volume expands. Furthermore, using more threads typically increases performance by enabling simultaneous task execution, thus speeding up query processing.

Database sizes for different SF The size of the database was calculated for each SF to evaluate how data volume increases. These measurements are illustrated in Figure 1, showing the relationship between SF and database size.

The relationship between the scaling factor and database size is direct and expected as seen in Figure 1. As the scaling factor increases, the database size also increases. This outcome is due to the tables loaded into the database, which are generated by dbgen. With higher scaling factors, dbgen produces tables with more rows, naturally leading to a larger database size.

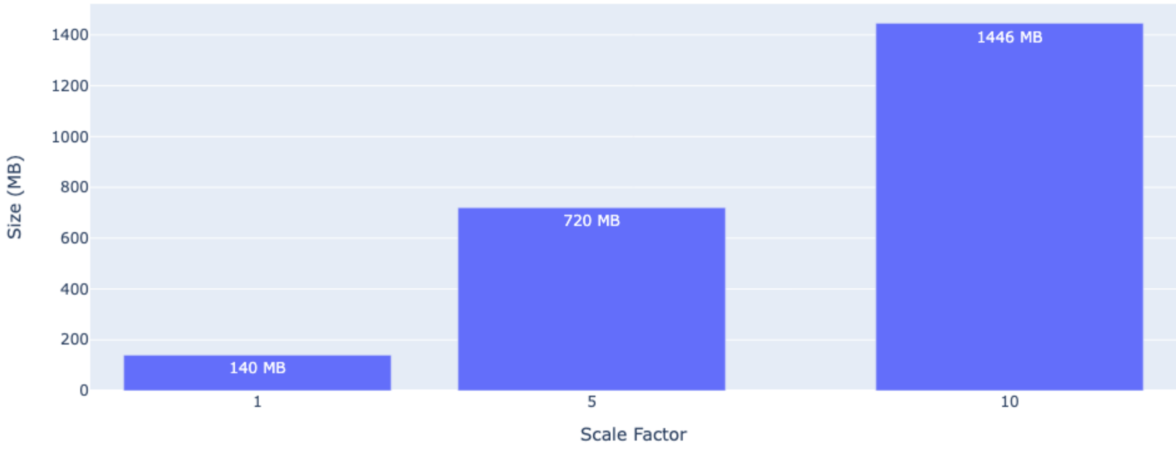

Figure 1: Database sizes across various SF's, demonstrating how data volume scales with increasing SF.

7 Conclusion

In this project, we explore and got familiar with DuckDB, we delved deeply into how changes in scaling factors and thread counts affect the performance of database queries. Our findings confirmed that larger databases, resulting from higher scaling factors, slow down query execution times. This slowdown occurs because more data takes longer to process, especially during operations like table scans, joins, and aggregations.

On the other hand, we saw that using more threads generally boosts performance, showcasing the power of parallel processing in today's database systems. However, the benefits of adding more threads decrease after a point due to the complexities of synchronization and the competition for resources.

The insights from this project highlight the balance needed between operational efficiency and resource management in database systems.

References

[1] DuckDB. URL https://duckdb.org/docs/.

[2] DuckDB Python API. URL https://duckdb.org/docs/api/python/overview.html.

[3] ClickHouse SSB 2009. URL https://clickhouse.com/docs/en/getting-started/example-datasets/star-schema.