

# Benchmarking Query Processors

## Introduction

For this project, I chose DuckDB with an experimental dimension of joins (1, 2, 3, 5, and 10 joins). I chose threads as my first configuration option since thread performance is important when choosing a CPU, as I expect performance gains to vary depending on the number of threads used. I chose memory as my second configuration option since memory performance provides similar insights when choosing an amount of RAM/system memory. I intend for both configuration options to provide useful information regarding hardware choices for running DuckDB.

## Methodology

Regarding environment control, the following specs describe the system used for this project:

- CPU Model: Intel Core i5-14400F 10 Core (6 Performance, 4 Efficiency) (2.50 GHz)
- Memory: 32 GB RAM
- OS: Windows 11 Home (25H2)
- Database Version: DuckDB v1.4.1 (Andium) b390a7c376

Data was generated with TPC-H using the DuckDB tpch extension by calling dbgen. I used a scale factor of 30 after I tested queries on factors of 1, 10, 30, and 100 and found 30 to be the largest dataset with reasonable performance (minutes/hours instead of days).

See appendix (p. 4-5) for queries.

As shown in comments and variable names in all queries, the queries increase in join complexity using the provided number of joins from the experimental dimension. The 10-Join query SELECTs from region instead of customer (like the other queries do) due to complexity constraints with SELECTing from customer. However, the 10-Join query is still significantly more computationally expensive than the 5-Join query, as expected.

## Results and Analysis

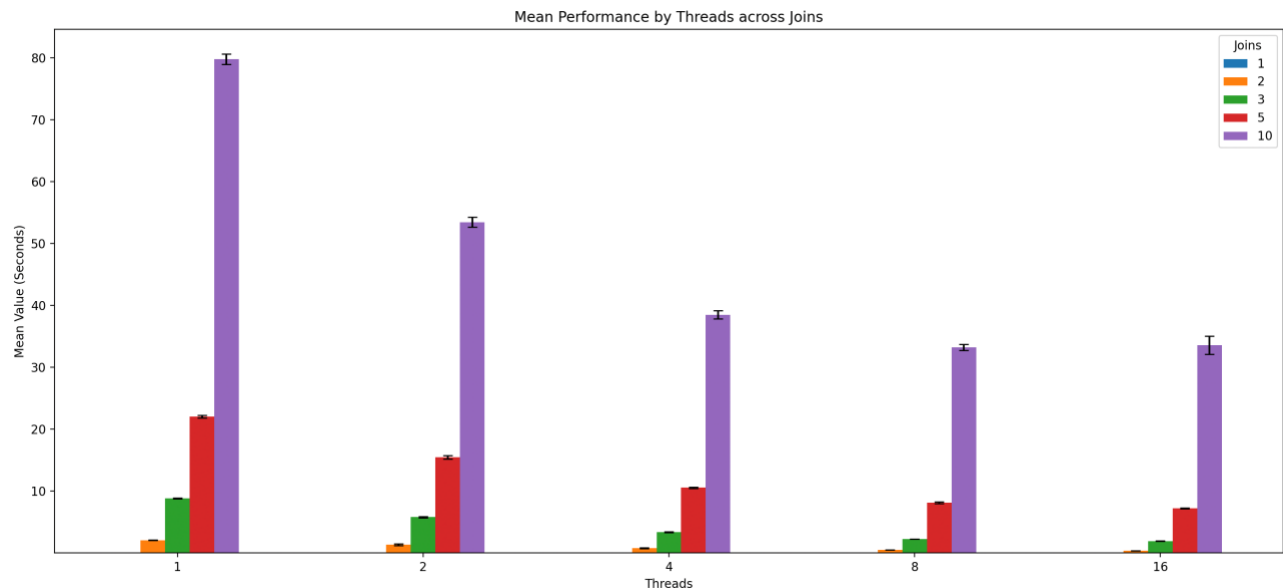


Figure 1 (Above): Mean query execution time at different thread amounts across joins, using a normal scale for y-axis.

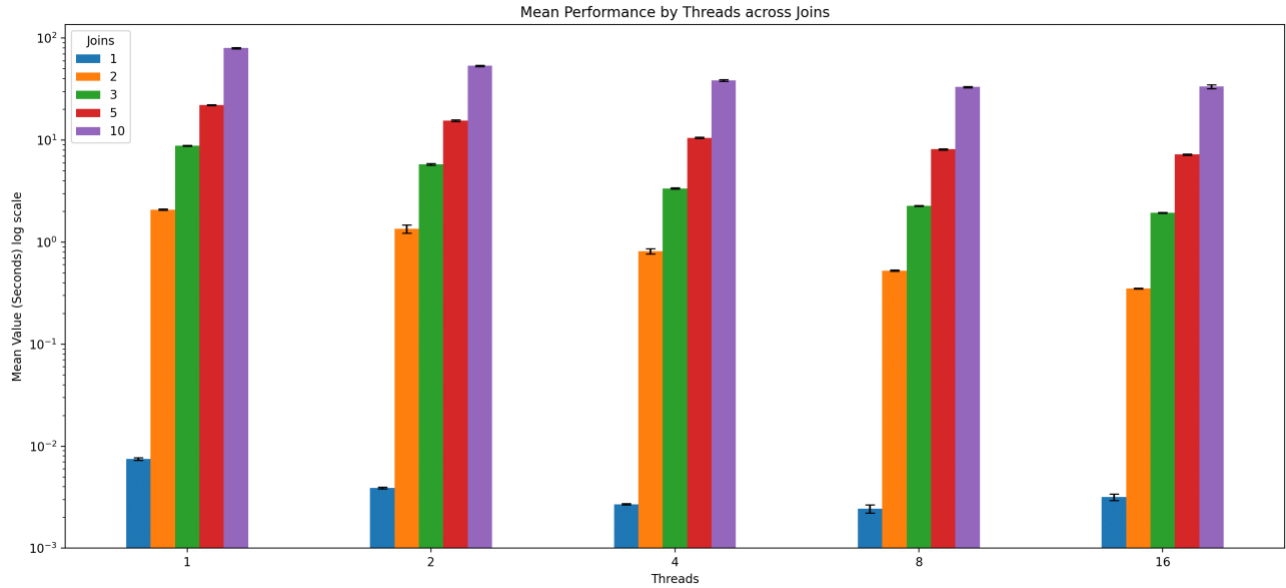


Figure 2 (Above): Mean query execution time at different thread amounts across joins, using a log scale for y-axis

The two figures above (Figure 1 and Figure 2) show a general improvement (faster completion) in query performance as more threads are allocated. In particular, the larger (and thus more computationally intense) queries show greater improvements in runtime. Overall, the runtime improvements appear to cease after 8 threads, as the results are nearly identical between 8 and 16 threads for all 5 joins. Although the plot does not exhibit a strict inflection point, it does show a practical saturation point around 8 threads where gains diminish sharply. Given that the CPU I used had 6 performance cores and 4 efficiency cores, I hypothesize the performance plateau occurs due to DuckDB's vectorized, parallel operators benefiting mostly from higher-throughput threads (thus using performance cores). Once all performance cores were utilized, additional threads were placed on slower efficiency cores, adding additional overhead when DuckDB synchronized processes. Running the same performance test at 6 threads (the number of performance cores on my CPU) could potentially provide additional supporting data to this hypothesis.

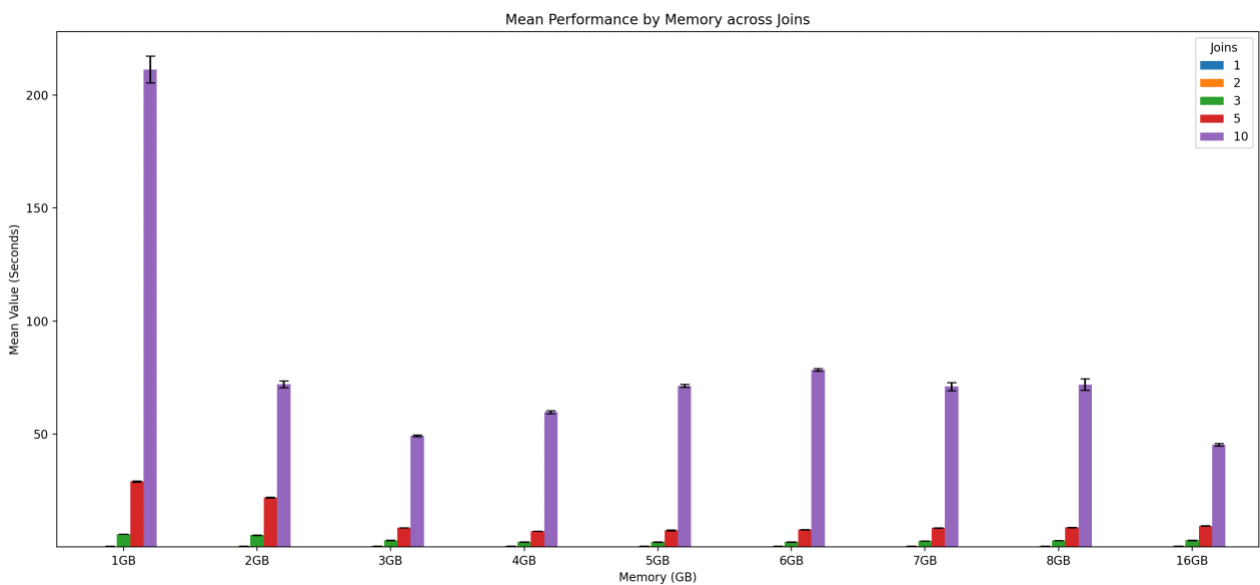


Figure 3 (Above): Mean query execution time at different memory amounts across joins, using a normal scale for y-axis.

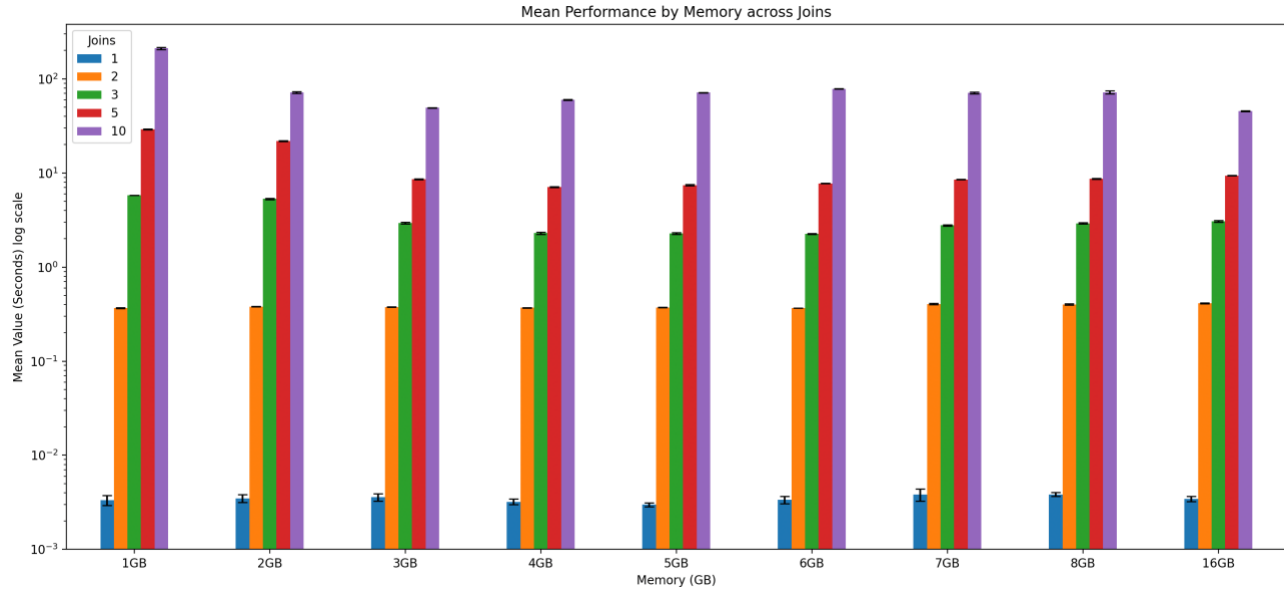


Figure 4 (Above): Mean query execution time at different memory amounts across joins, using a log scale for y-axis.

The two figures above (Figure 3 and Figure 4) show a more interesting trend regarding query performance. As shown in the graphs, early increases in memory (from 1 to 3 GBs) result in clear performance gains. However, the graphs contain a clear inflection point at 3 or 4 GBs (depending on the number of joins), where the performance then appears to get worse for increasing memory amounts. For smaller join amounts (1, 2, 3, and 5), the performance is relatively identical across all remaining memory values (4GB or more). However, the performance of the 10-join query improves at 16 GB of memory, returning to a nearly identical level as at 3GB. I hypothesize the non-monotonic trend occurs due to buffer availability constraints for DuckDB and the size of intermediate join results. For the smaller join amounts (1, 2, 3, and 5), I suspect the intermediate join results are less than 4GB, thus resulting in no performance changes after 4GB. Performance trends of the 10-join query are more difficult to predict, but I suspect certain DuckDB operations (ex: per-thread buffering) may result in higher overhead at memory values between 4 and 16 GB. At 16 GB, I suspect the intermediate join results may fit into memory, resulting in a performance gain. It would be wise to test this theory in more complexity by testing at higher memory amounts (ex: 32 GB) and potentially with different memory values. Given the time constraints of the project, I chose not to pursue testing every possible combination of memory and thread values.

## Conclusion and Recommendations

Based on the results and discussion above, I recommend allocating memory based on the size of intermediate join results (or overall size of intermediate results) and allocating threads based on the number of higher-performance CPU cores available. Specific to my machine, scale factor (30), and queries, I would allocate 6 threads for all queries and 4GB of memory for 1, 2, 3, and 5 join queries. For the 10-join query, I would allocate 32 GB memory (maximum available), though additional testing with larger memory allocations would be necessary to determine potential gains. Although these recommendations are limited to the specifications of my tests, I believe they provide useful information for choosing components in a machine for DB query processing. Specifically, a user should not add additional memory beyond the expected amount needed for intermediate results

and should spend remaining resources on CPUs. CPUs should have identical cores to avoid potential overhead with process synchronization between threads.

## GenAI Reflection

I used Generative AI (specifically Google Gemini 2.5 Pro and OpenAI GPT-5) primarily for query generation, plotting results in Matplotlib, and discussion of analysis. Overall, I found Generative AI to be less helpful for query generation and much more helpful for plotting and understanding results. When generating queries, both LLMs initially generated queries that had nearly identical performance at 2,3, and 5 joins, likely due to trivial joins with small tables (ex: region). This led to a discussion with Google Gemini, where I was ultimately able to get queries for 2, 3, and 5 joins that scaled decently, as well as a 10-join query that could be executed in an acceptable amount of time. I ultimately hand wrote queries based on recommendations made by both LLMs.

ChatGPT was very helpful for generating Matplotlib code (after failing to guide me through plot generation in Excel, but I think Excel is the root cause of the problem). After asking for several minor graph changes, the graphs appear as I intended. When interpreting the results, I gave ChatGPT my initial thoughts (ex: CPU cores limiting improvements with threads) and received helpful feedback.

Sample prompt to ChatGPT:

Given that the CPU I used had 6 performance cores and 4 efficiency cores, could a lack of performance improvements after 8 cores be attributed to CPU limitations. How might the balance of performance vs efficiency cores be relevant? Limit your answer to a few sentences.

Response from ChatGPT:

Yes — the lack of performance improvement after 8 threads can be attributed to CPU limitations. DuckDB's parallel execution benefits most from high-throughput cores, so once all 6 performance cores are fully utilized, additional threads run on slower efficiency cores, which have lower single-thread performance and may increase synchronization overhead. This imbalance means adding threads beyond the number of performance cores yields diminishing returns.

## Appendix

Test Queries:

```
join_1_query = """
```

```
-- Query 1: 1 Join (Tables: Customer and Nation)
```

```
SELECT COUNT(*)
```

```
FROM customer c
```

```
JOIN nation n ON c.c_nationkey = n.n_nationkey
```

```
"""
```

```
join_2_query = """
```

```
-- Query 2: 2 Joins (Tables: Customer, Nation, and Orders)
```

```
SELECT COUNT(*)
```

```
FROM customer c
```

```
JOIN nation n ON c.c_nationkey = n.n_nationkey
```

```
JOIN orders o ON c.c_custkey = o.o_custkey;
```



```
join_3_query = ""
```

```
-- Query 3: 3 Joins (Tables: Customer, Nation, Orders, and Lineitem)
```

```
SELECT COUNT(*)
```

```
FROM customer c
```

```
JOIN nation n ON c.c_nationkey = n.n_nationkey
```

```
JOIN orders o ON c.c_custkey = o.o_custkey
```

```
JOIN lineitem l ON o.o_orderkey = l.l_orderkey
```

```
""
```

```
join_5_query = ""
```

```
-- Query 5: 5 Joins (Tables: Customer, Nation, Orders, Lineitem, Part, and Partsupp)
```

```
SELECT COUNT(*)
```

```
FROM customer c
```

```
JOIN nation n ON c.c_nationkey = n.n_nationkey
```

```
JOIN orders o ON c.c_custkey = o.o_custkey
```

```
JOIN lineitem l ON o.o_orderkey = l.l_orderkey
```

```
JOIN part p ON l.l_partkey = p.p_partkey
```

```
JOIN partsupp ps
```

```
ON p.p_partkey = ps.ps_partkey
```

```
AND l.l_suppkey = ps.ps_suppkey;
```

```
""
```

```
join_10_query = ""
```

```
-- Query 10: 10 Joins (Tables: Region x2, Nation x3, Customer, Orders, Lineitem, Supplier, Part, Partsupp)
```

```
SELECT COUNT(*)
```

```
FROM region r -- Starting from region instead of customer due to performance
```

```
JOIN nation n1 ON r.r_regionkey = n1.n_regionkey
```

```
JOIN nation n2 ON n1.n_regionkey = n2.n_regionkey
```

```
JOIN customer c ON n2.n_nationkey = c.c_nationkey
```

```
JOIN orders o ON c.c_custkey = o.o_custkey
```

```
JOIN lineitem l ON o.o_orderkey = l.l_orderkey
```

```
JOIN supplier s ON l.l_suppkey = s.s_suppkey
```

```
JOIN nation n3 ON s.s_nationkey = n3.n_nationkey
```

```
JOIN part p ON l.l_partkey = p.p_partkey
```

```
JOIN partsupp ps
```

```
ON p.p_partkey = ps.ps_partkey
```

```
AND s.s_suppkey = ps.ps_suppkey
```

```
JOIN region r2 ON n3.n_regionkey = r2.r_regionkey;
```

```
""
```

**Source Code and Results:**

Source code and benchmark results are available in a public [GitHub Repository](#).