Exploration of DuckDB

Jorrit Vander Mynsbrugge –group 14 – sid 0606134 – working student

Contents

1 Abstract

This report presents an introduction to DuckDB, a relatively new OLAP database system (Online Analytical Processing). It provides a description of the position DuckDB takes up in the overall database landscape, and evaluates its performance through three experimental analyses. The first experiment benchmarks DuckDB against a traditional RDBMS (Relational Database Management System) under analytical workload conditions. The second investigates the scalability characteristics of DuckDB. The third experiment compares its performance to other in-memory OLAP processing systems.

2 Introduction

2.1 The exponential growth of hardware

What volumes constitutes 'big data' has always been a moving target, evolving alongside advancements in storage and processing technologies. Since the term was coined by Josh Mashey in the late 1990s [1], [2], it has referred to datasets that were growing too large for traditional systems to handle efficiently. In the early 2000s, this meant datasets ranging from gigabytes to low terabytes. By the 2010s, the proliferation of social media, IoT devices, and cloud computing pushed the threshold to tens or hundreds of terabytes, with organizations like Facebook and Google processing petabyte-scale data regularly. Today, due to the rise of video streaming platforms and further uptake of distributed cloud systems, data volumes have reached zettabyte levels globally [3], with FAANG organizations often working with petabytes or exabytes of data in real time.

On the other end of the spectrum, the 'small data' range has also been ever-growing, driven by the same continual advancement of hardware capabilities in single-node systems. In the early 2000s, datasets that could fit within a few gigabytes were considered small, as they could be processed efficiently on desktop computers of the time. By the 2010s, datasets in the tens of gigabytes became manageable on commodity hardware, and today, single-node systems with terabytes of RAM and high-performance multi-core processors can handle hundreds of gigabytes or even low-terabyte datasets entirely in memory. To make these numbers more tangible, at the launch of Amazon EC2 in 2006 a single instance of m1.small offered 1 CPU core and 1.7GB of memory [4]. As of September 2024, a memory optimized x8g.metal-48xl instance has 192 vCPU and 3TB or memory at 18.7$/hour [5]. To take it to the extreme, Amazon offers even High-Memory (U-1) instances that boast 448 vCPU and 24TB of main memory [6]. This evolution has expanded the niche for systems like DuckDB, which are optimized for high-performance analytics on local or single-node environments.

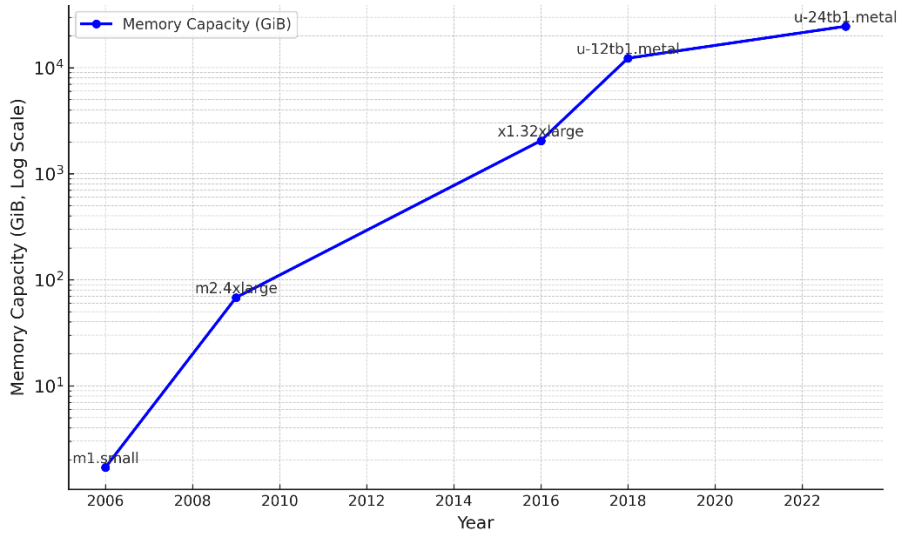


Figure 1 Evolution of Amazon EC2 instance memory capacity (note: logarithmic Y-axis).

2.2 The rise of small data

In this context, small data acolytes have risen [7]. They argue that real data volumes aren't as big as initially estimated, with most workloads now easily fitting in one machine [8]. Even though companies gain more data over time, many will mainly focus on the most recent data as this is most relevant for decision making. Secondly, scaling single-node hardware no longer follows an exponential cost curve, as cloud suppliers slice up their machines into smaller compute units [9]. As a consequence, the cost per unit does not change whether you are working on a tiny or larger slice. Finally, the hidden cost of big data analytics is found in its complexity. By adopting a single-node scale up solution instead of a distributed scale-out solution, the additional complexity of consensus algorithms (Paxos, ...), data shuffling for joins, data alignment, the CAP theorem, ... is no longer a concern [10]. To turn these theoretical arguments into practice, the company MotherDuck has developed a cloud-based data warehouse solution surrounding DuckDB, demonstrating its value in real-life use cases [11].

2.3 Central research questions

In my current employer's 'data chapter', an anti-pattern has emerged where every project must leverage expensive distributed cloud infrastructure for the sake of scalability, often disregarding actual data volumes. This leads to situations where simple log file analysis is performed on costly Databricks clusters. Conversely, our 'software chapter' dictates the use of SQL Server for all data problems, regardless of whether the workload is transactional or analytical. These observations, coupled with my interest in the 'small data' movement, prompted an investigation into several key questions: Does an OLAP query engine truly outperform a mature OLTP database engine for analytical workloads? How effectively can these engines scale within a single node? And lastly, how does the performance of DuckDB specifically compare to its competitors in this context?

2.4 Situating DuckDB in the database landscape

With over 380 different database solutions ranked in the Redgate Software DB-engines ranking [12] it is important to situate DuckDB properly. DuckDB was developed in 2019 [13] by Hannes Mühleisen and Mark Raasveldt at the Database Architectures group of the CWI Amsterdam [14]. It is an OLAP database providing high-performance analytical capabilities for in-memory, single-node analytics. Similar to SQLite it is both open-source and embedded. The open-source design of DuckDB facilitates auditable security by enabling comprehensive examination and verification of its codebase whilst the embedded architecture means it can operate directly within applications, without the need for a standalone database server. As such it targets the niche of use cases where traditional OLAP systems (e.g. Clickhouse, Spark) would be overkill or too complex. It is meant for scenarios like ad-hoc analytics, data science workflows, interactive data apps (e.g. mobile) and lightweight ETL pipelines (e.g. in conjunction with Data Build Tool), particularly when working on datasets that fit in memory or on fast local storage like NVMe SSDs [15]. It is directed at users that want to interact with data using SQL rather than a library-specific DSL (polars, data.table).

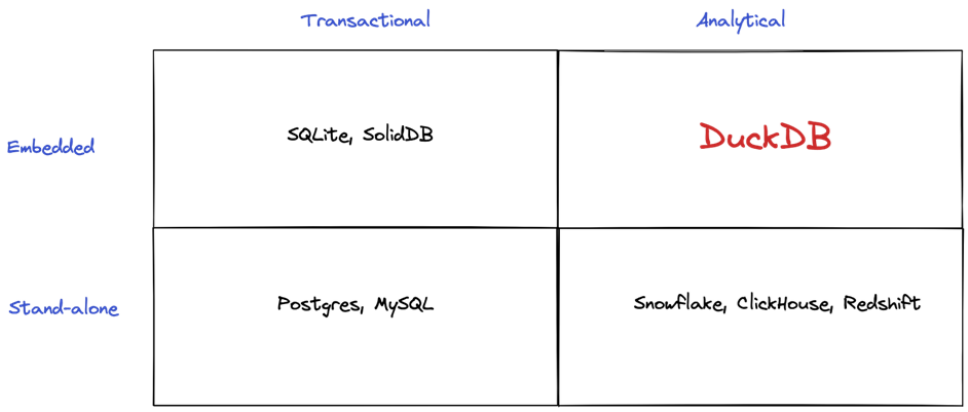


Figure 2 Positioning DuckDB in the DB landscape (source [16])

2.5 Features of DuckDB

DuckDB is a database with several compelling features. It can operate entirely in memory, allowing for fast query execution. While it can work directly with CSV files loaded into memory, it operates more efficiently when used with the Parquet file format. In this mode it enables lazy loading, column pruning, and predicate pushdown, significantly minimizing disk I/O and optimizing query performance. DuckDB also offers its own proprietary format that bundles the engine and data together, providing ACID guarantees for enhanced data integrity.

Furthermore, DuckDB is lightweight, with an executable size of only 25MB. This makes it highly portable and easy to deploy. It boasts a rich ecosystem of extensions that enhance its functionality, some of which were used for this report. It supports complex SQL queries, including joins, window functions, and CTEs (common table expressions). It also features quality-of-life SQL extensions, such as SELECT * EXCLUDE, SELECT * REPLACE, and GROUP BY ALL. Finally, DuckDB caters to a diverse user base by providing bindings for numerous programming languages, including Python, R, and Java. This flexibility allows users to seamlessly integrate DuckDB into their existing workflows.

3 Methodology

3.1 Experiments

As stated in section 2.3, to put to DuckDB to the test 3 research questions were identified:

- How does DuckDB perform against a traditional RDBMS under analytical workload conditions?
- How does DuckDB scale with growing database size?
- How does DuckDB hold up against other in-memory OLAP processing systems?

In this section, the methodology to evaluate these questions is elaborated. In section 4, the results of the benchmarks are discussed.

3.2 Hardware

All of the experiments were performed on a standard workstation. At the time of writing, the hardware listed below can be bought for 534 EUR. The test system operates on Windows 10 22H2 and runs Ubuntu 22.04.5 LTS in WSL-2.

Component	Model
CPU	AMD Ryzen 7 5700X 8 cores/16 threads 3.4 GHz / 4.6 GHz Boost
RAM	G.Skill DDR4 Trident-Z 4x16GB 3200MHz 64GB
Storage	Samsung 990 PRO 2TB PCI-E 4.0 x4 M.2 SSD
Chipset	Gigabyte B550M AORUS ELITE

3.3 Three reference benchmarks

The Transaction Processing Performance Council (TPC) offers two widely recognized benchmarks for evaluating analytical workloads: TPC-H and TPC-DS. Both benchmarks are designed to assess the performance of database systems in executing complex analytical queries, but they differ in their focus and scope. The fundamental distinction lies in TPC-H's emphasis on relatively simple, static workloads versus TPC-DS's focus on the dynamic and multifaceted nature of modern analytical workloads, making the latter more representative of real-world use cases in data warehousing environments.

The third benchmark considered in this test is the database-ops-like benchmark. In this benchmark the data and operations align more closely with data processing and manipulation tasks. The datasets are simpler than the ones used in TPC-H and the operations, such as filtering, grouping, joining, and summarization are performed directly in memory, which is more representative of data science and analytics workflows**.** The main goal for this benchmark is to compare similar data wrangling libraries and tools to one another, which is why it was are withheld in this study.

3.4 Comparing DuckDB to traditional RDBMS for analytical workloads

In section 2.4, DuckDB is proposed as an engine for OLAP workloads. This begs the question why a traditional RDBMS would not be equally well suited for such a workload. On the site of Bicortex [17] an interesting statement can be found:

> *To efficiently support OLAP workload, it is critical to reduce the amount of CPU cycles that are expended per individual value. The state of the art in data management to achieve this are either vectorized or just-in-time query execution engines. DuckDB contains a columnar-vectorized query execution engine, where queries are still interpreted, but a large batch of values from a single (a 'vector') are processed in one operation. This greatly reduces overhead present in traditional systems such as PostgreSQL, MySQL or SQLite which process each row sequentially. Vectorized query execution leads to far better performance in OLAP queries.*

The goal of this experiment is to verify this statement by analyzing the impact of DuckDB's vectorized query engine on performance on analytical workloads. For this, Postgres 17 and DuckDB 1.1 were pitched against each other on the same machine using the standard TPC-DS benchmark.

TPC-DS [18] is a sophisticated benchmark that models a retail data warehouse environment with a complex schema star schema as shown in Appendix A, existing of 24 tables. It incorporates 99 queries that represent a broad and diverse range of decision-support scenarios. These include hierarchical data processing, advanced analytical functions, and reporting queries that mimic real-world business intelligence and OLAP tasks. Results are shown in section 4.1.

3.5 Evaluating the scalability of DuckDB

In section 2.4, DuckDB is linked to 'small scale' data problems. This begs the question how fast the problem size exceeds single node limitations. The goal of the second experiment is therefore to execute multiple queries on growing datasets, each one roughly 3x the size of the previous, to evaluate the impact on performance. By normalizing the execution time by the size of the database, one can

assess the scalability of the product. For this experiment, and to gain familiarity with multiple benchmarks, this time TPC-H was chosen as reference benchmark.

The TPC-H [19] is a decision support benchmark that evaluates systems on their ability to process analytical queries over a relatively simple schema. The dataset, representing a wholesale supplier's operations, is spread across 8 interrelated tables, forming a relatively simple schema that facilitates efficient evaluation of query performance. It features a set of 22 queries emphasizing straightforward joins, aggregations, and filtering operations. These queries represent real-world business intelligence tasks such as sales trends analysis, supply chain evaluation, and revenue forecasting. The benchmark supports various scale factors, ranging from 1GB to multiple terabytes, making it suitable for environments of varying sizes and complexity.

DuckDB features a TPC-H extension [20] which embeds the benchmark queries, and facilitates the creation of the datasets for the desired scale factor. However, as generating these datasets requires a lot of time and system memory, DuckDB also provides pre-generated datasets. These datasets are stored in DuckDB's proprietary format, which features high compression and bundles the engine with the data.

File name	Scale Factor (GB uncompressed)	Size on disk in MB (compressed)
tpch-sf1.db	1	254
tpch-sf3.db	3	771
tpch-sf10.db	10	2.612
tpch-sf30.db	30	7.965
tpch-sf100.db	100	26.962

Table 1 Overview of the TPC-H databases used for the benchmark.

3.6 Evaluating DuckDB against other popular in memory analytical query engines.

Back in 2014, Matt Dowle, creator of the popular R library 'data.table' created a comparative benchmark between data.table, and very early versions of the 'tidyverse' default data wrangling package dplyr and the python counterpart pandas [21]. The benchmark consisted of 5 simple 'group by' queries: large groups and small groups (different cardinality) on different columns of various types (char, int, float). For every engine and query 2 runs were executed, to show cache effects. Matt argued in data analytics, the most relevant measurement is the time required for the first execution. The dataset varied in size from 0.5 to 100GB. The benchmark was executed on a 32 threads, 240GB RAM EC2 instance, the biggest available at the time.

A few years later, Jan Gorecki, working at H2O.ai, took that benchmark and converted it into the 'Database-like ops benchmark' [22], [23]. In this new version many other query engines (Clickhouse, Polars, cuDF, spark, dask, DuckDB, ...) were added. In addition, 10 new group by and 5 new join queries were defined. Finally, the 100GB dataset was removed and a smaller EC2 instance with 128GB of RAM was used to periodically perform the benchmark on the latest versions of the supported engines. This benchmark was maintained and updated until July 2021.

Because the latest version of DuckDB in this benchmark was still 0.2.7 and due to the renown this benchmark had acquired over the years, the DuckDB team has decided to give the H2O.ai benchmark new life and maintain it for the foreseeable future [24], [25], [26] so as to be able to show the performance of their latest development on DuckDB. The same 10 group by and 5 join queries are evaluated, although the hardware has changed more than once already [27].

This latest DuckDB benchmark shows that DuckDB 1.1.0 is an all-round top performer and scales well over the tested dataset sizes. On the 50GB join test, DuckDB and Polars are the only 2 solutions out of the 15 tested that are able to perform the join queries on a 250GB memory machine.

The goal of running this benchmark locally is to compare the runtimes of recent versions of a few popular in-memory query engines using common workstation hardware, as opposed to relying on the resources of a large cloud machine. By identifying the point at which the limitations of the workstation are reached for the different solutions, this experiment aims to provide insights into the comparative performance and scalability of the engines in real-world data science scenarios.

4 Evaluation

4.1 OLTP vs. OLAP results

In a very recent article [28] pitching DuckDB against Postgres for a single query speedups of 1500x were achieved. In order to estimate proper problem size, all queries were first run on the DuckDB engine. Using Python scripts, the DuckDB TPC-DS extension was harnessed to save the datasets into the proprietary DuckDB file format. This was done for scale factor 1, 10, 50, 100. Another script then executed all 99 queries for these different datasets.

The slowest query on scale factor 10 ran in ~4 seconds. The same queries were run on Postgres using a timeout of roughly 100x this execution time. The idea being, if all queries would time out, the benchmark would still complete in 10 hours. During the query execution, on scale factor 10, available memory was never a bottleneck. In the end 15 out of 99 queries timed out. As a result it was decided not to reiterate the experiment using a larger scale factor. For the queries that did not time out, the execution time was divided by its execution time on DuckDB. These 'speedup' ratios were then sorted and are shown using a monotonically increasing bar chart in Figure 3. Note that the y-axis uses a logarithmic scale. The speedup values ranged between 9.4 (for query 84) and 22.764 for (query 32).

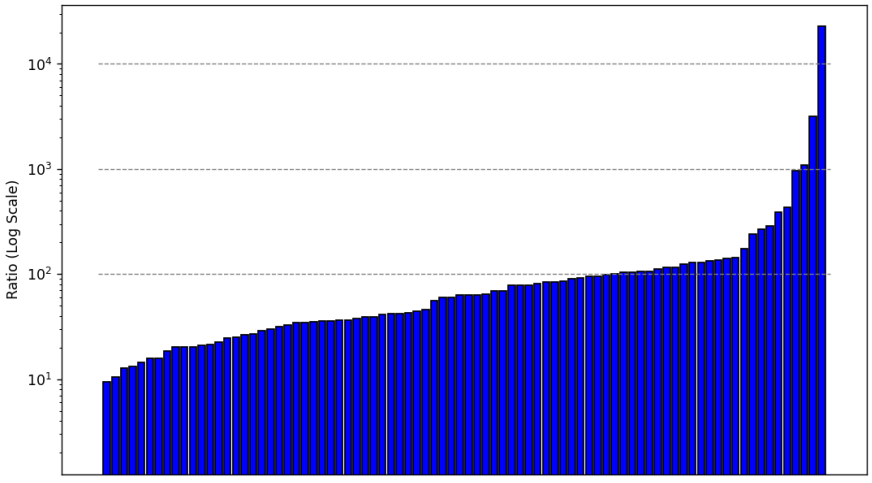


Figure 3 Speedup of DuckDB over Postgress for the 84 TPC-DS queries that did not time out on postgres.

4.2 Scalability

Before starting the scalability test a quick test was performed to identify any form of caching, which would result in shorter execution times, the second time a query is executed. The result depicted in Figure 4 shows this is not the case. As a result, in the following results only the first run is shown.

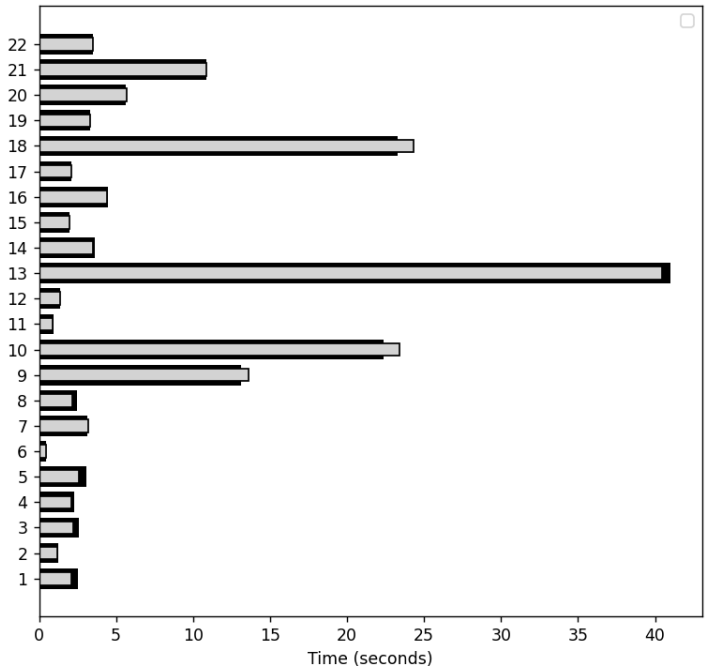


Figure 4 Query execution time for first run (black) and second run (gray) on TPC-H with SF=100

From this result 4 queries with rather different runtimes are selected: queries 7, 9, 10 and 13. These queries are subsequently running on the 5 different TPC-H databases. Results are depicted in Figure 5. For every query the runtimes increase with increasing database. However, none of these queries had issues executing correctly on this simple workstation, not even on the SF100 dataset.

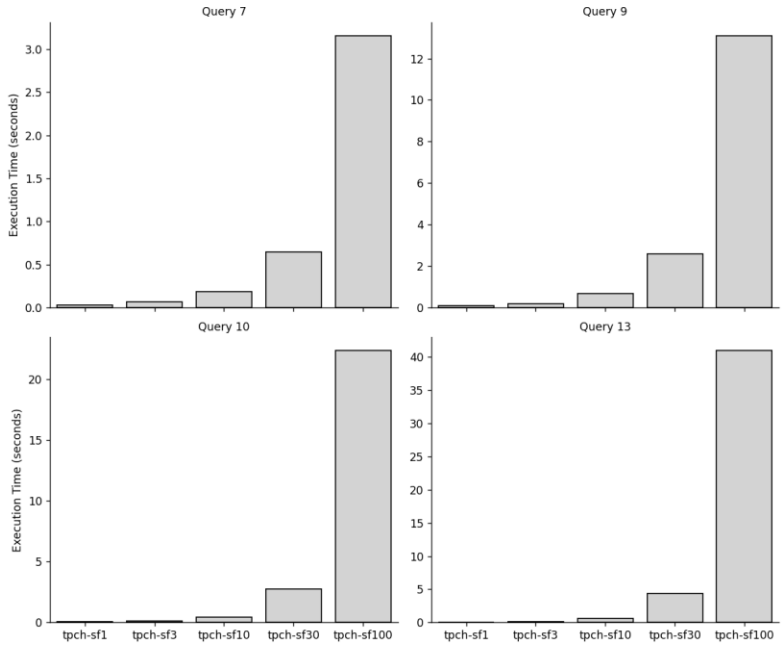


Figure 5 Query execution time for 4 selected queries on different TPC-H databases.

To see if the query performance is linear the execution times are normalized by dividing by the dataset size. Note that this size corresponds to the compressed 'disk size'. Results are shown in Figure 6. From this analysis it becomes clear that slow queries (joins) do not scale linearly with the expanding dataset. This is especially visible for query 13 on larger datasets. This query contains a LEFT OUTER JOIN and

the intermediate tuples from this operation cause the increase in execution time to exceed the 3x linear growth of the dataset itself.

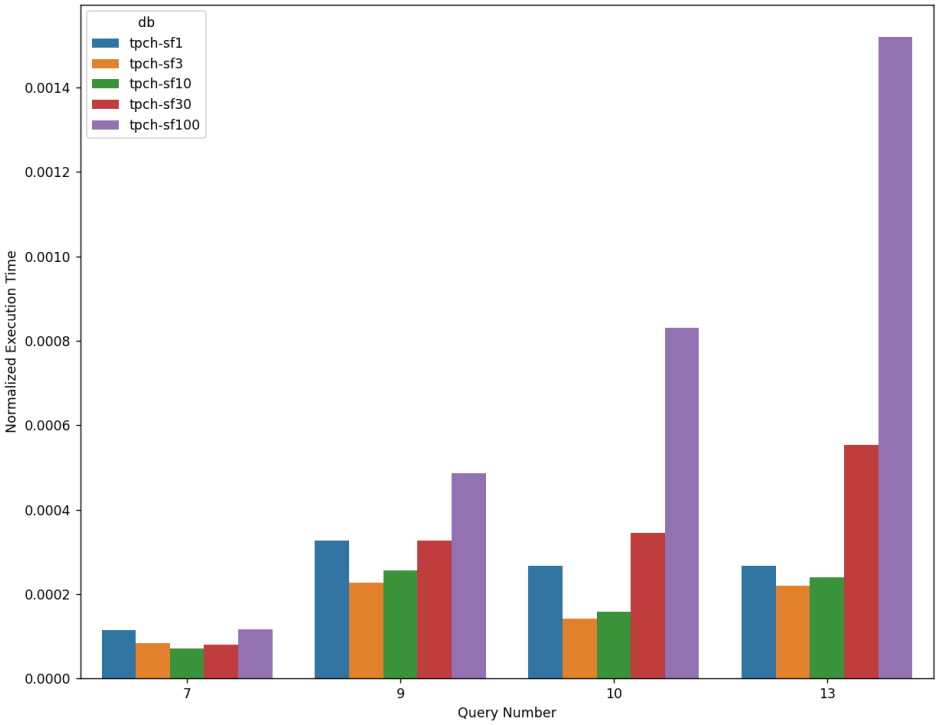


Figure 6 TPC-H query execution time divided by database size for 4 selected queries.

4.3 DuckDB versus other in-memory engines

Running this benchmark on a test workstation is not straightforward. The documentation is incomplete and execution relies on the availability of specific system packages (e.g. pandoc), globally installed python & R packages (readr, bit64, ...), environment variables, tweaking code files to circumvent hard coded constraints, and even manual file copy operations, ... As a result, the first 7 days were spent merely getting sensible output from the provided scripts.

Details on the dataset structure and executed queries can be found in appendices D and E.

In this experiment DuckDB is compared to data.table, polars, and pandas. Benchmarks were run on the 1e7 (0.5GB), the 1e8 (5GB) and 3e8 (15GB) datasets. At this latest stage some timeouts occurred on the 'group by' queries, and not a single engine was able to complete any of the join queries. While at first this seemed odd due to the fact that the system had over 50GB of available RAM for this test, it does show the downside of working with .csv input files, which first require in memory parsing. It does beg the question how much further single node scalability extends if data would have been converted into 'parquet'.

The 1e8 'group by' results, shown in Figure 7, confirm the results of the original public benchmark and show DuckDB as an overall top performer, only occasionally beaten by data.table or polars. The join results, depicted in Figure 8 for both the 1e7 and 1e8 dataset show strong results for polars when joining with the 'medium' right dataset, but DuckDB leading on small join and large join. These benchmarks also indicate that Pandas, today still the uncontested most popular of the tested solutions, remains the worst performer even after its upgrade to an Arrow backend in version 2.0 (here tested as 2.2.3).

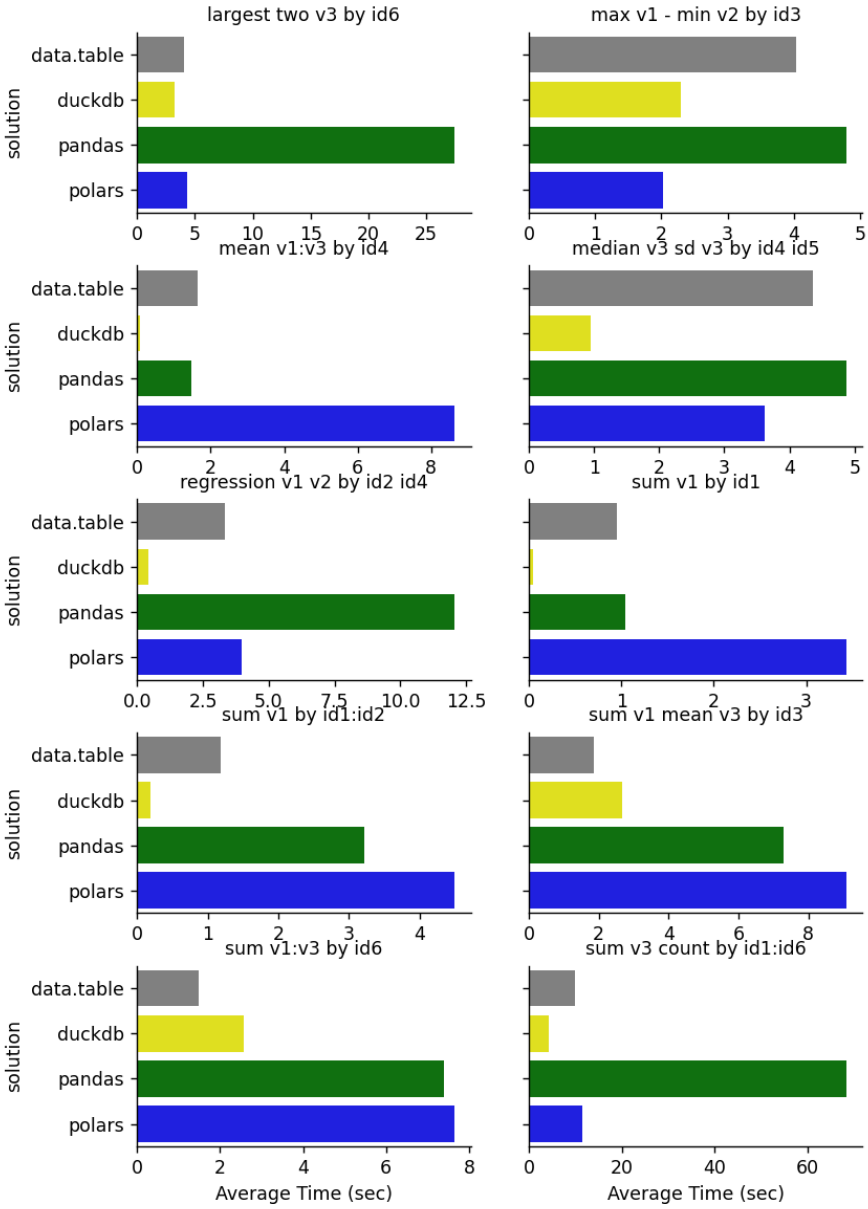


Figure 7 Runtimes for the first run of 10 different 'group by' queries on the 1e8 dataset.

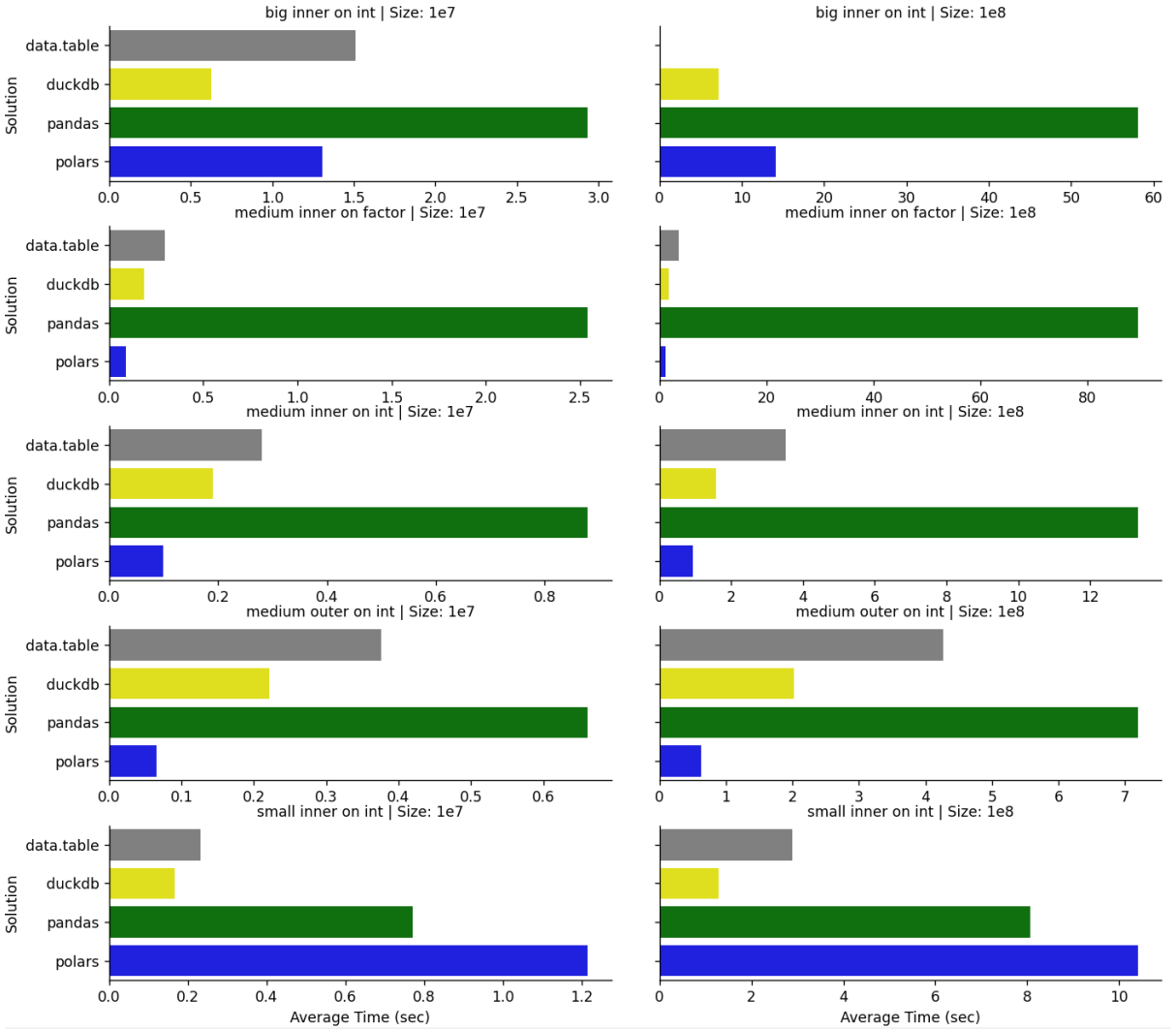


Figure 8 Runtimes for the first run of 5 different join queries on the 1e7 and 1e8 dataset. Missing runtimes indicate the query execution failed.

5 Conclusions

In performing this research, much was learned about the evolving database landscape and the position DuckDB occupies through its features. The exploration of the "small data" movement highlighted sensible arguments in its favor, particularly given the enormous advancements in single-node hardware capabilities and the reduced complexity of query execution. The first experiment reinforced the power of OLAP engines, while also validating the concerns of DBAs about running analytical workloads on traditional RDBMS. DuckDB's PostgreSQL extension elegantly bridges this gap, enabling offloading of analytical queries without requiring complex transformations of data source systems. The second experiment demonstrated DuckDB's scalability, which, while not linear, proved surprisingly robust, processing queries on a 100GB dataset effortlessly on a modest workstation. Lastly, the final experiment showed DuckDB's strong performance compared to specialized in-memory query engines for data manipulation tasks. Combined with its comprehensive feature set, DuckDB emerges as a solid choice for future OLAP and small data processing tasks, proving its potential to excel in the modern database landscape.

6 Online repository

The code repository accompanying this research report can be found at https://github.com/jorritvm/duckdb_bench/

7 List of Abbreviations

- CAP: Consistency, Availability, and Partition Tolerance
- CPU: Central Processing Unit
- CWI: Centrum voor Wiskunde en Informatica
- CTE: Common Table Expressions
- DBT: Data Build Tool
- DSL: Domain Specific Language
- EC2: Elastic Compute (Amazon cloud compute service)
- ETL: Extract, Transform and Load
- FAANG: Facebook, Amazon, Apple, Netflix and Alphabet (Google)
- NVMe: Non-Volatile Memory Express
- OLAP: Online Analytical Processing
- OLTP: Online Transactional Processing
- RAM: Random Access Memory
- RDBMS: Relational Database Management System
- SF: Scale Factor
- SSD: Solid State Disk
- TPC: Transaction Processing Performance Council
- vCPU: Virtual Central Processing Unit
- WSL: Windows Subsystem for Linux

8 References

[1] “Big data,” *Wikipedia*. Nov. 21, 2024. Accessed: Nov. 30, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Big_data&oldid=1258765268

[2] S. Lohr, “The Origins of ‘Big Data’: An Etymological Detective Story,” Bits Blog. Accessed: Nov. 30, 2024. [Online]. Available: https://archive.nytimes.com/bits.blogs.nytimes.com/2013/02/01/the-origins-of-big-data-an-etymological-detective-story/

[3] “Zettabyte Era,” *Wikipedia*. Nov. 27, 2024. Accessed: Nov. 30, 2024. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Zettabyte_Era&oldid=1259844892

[4] “EC2 instance timeline.” Accessed: Nov. 30, 2024. [Online]. Available: https://instancetyp.es/

[5] “Amazon EC2 X8g instances – AWS,” Amazon Web Services, Inc. Accessed: Nov. 30, 2024. [Online]. Available: https://aws.amazon.com/ec2/instance-types/x8g/

[6] “Amazon EC2 High Memory Instances – Amazon Web Services (AWS),” Amazon Web Services, Inc. Accessed: Nov. 30, 2024. [Online]. Available: https://aws.amazon.com/ec2/instance-types/high-memory/

[7] “Small Data SF 2024: Join the Movement to Build Bigger with Small Data and AI.” Accessed: Nov. 30, 2024. [Online]. Available: https://www.smalldatasf.com/2024/

[8] S. Sitaram, “Small Data is bigger and hotter than ever,” MotherDuck. Accessed: Oct. 31, 2024. [Online]. Available: https://motherduck.com/blog/small-data-sf-recap/

[9] MotherDuck, “Big Data is Dead - MotherDuck Blog,” MotherDuck. Accessed: Nov. 17, 2024. [Online]. Available: https://motherduck.com/blog/big-data-is-dead/

[10] J. Tigani, “The Simple Joys of Scaling Up,” MotherDuck. Accessed: Oct. 31, 2024. [Online]. Available: https://motherduck.com/blog/the-simple-joys-of-scaling-up/

[11] MotherDuck, “MotherDuck: Ducking Simple Data Warehouse based on DuckDB,” MotherDuck. Accessed: Nov. 30, 2024. [Online]. Available: https://motherduck.com/

[12] Redgate Software, “DB-Engines Ranking,” DB-Engines. Accessed: Nov. 30, 2024. [Online]. Available: https://db-engines.com/en/ranking

[13] H. Mühleisen and M. Raasveldt, “Releases · duckdb/duckdb,” GitHub. Accessed: Nov. 30, 2024. [Online]. Available: https://github.com/duckdb/duckdb/releases

[14] CWI Amsterdam, “DuckDB: a high-performance analytical database management system.” Accessed: Nov. 30, 2024. [Online]. Available: https://www.cwi.nl/en/results/software/duckdb-a-high-performance-analytical-database-management-system/

[15] S. Späti, “The Enterprise Case for DuckDB: 5 Key Categories and Why Use It,” MotherDuck. Accessed: Nov. 17, 2024. [Online]. Available: https://motherduck.com/blog/duckdb-enterprise-5-key-categories/

[16] K. Osei, “DuckDB and the next frontier of OLAP databases,” Kojo’s blog. Accessed: Dec. 05, 2024. [Online]. Available: https://kojo.blog/duckdb/

[17] Martin, “DuckDB – The little OLAP database that could. TPC-DS Benchmark Results and First Impressions.,” bicortex | Business Intelligence & Analytics. Accessed: Nov. 30, 2024. [Online]. Available: https://bicortex.com/duckdb-the-little-olap-database-that-could-tpc-ds-benchmark-results-and-first-impressions/

[18] “TPC-DS Homepage.” Accessed: Dec. 04, 2024. [Online]. Available: https://www.tpc.org/tpcds/default5.asp

[19] “TPC-H Homepage.” Accessed: Dec. 04, 2024. [Online]. Available: https://www.tpc.org/tpch/default5.asp

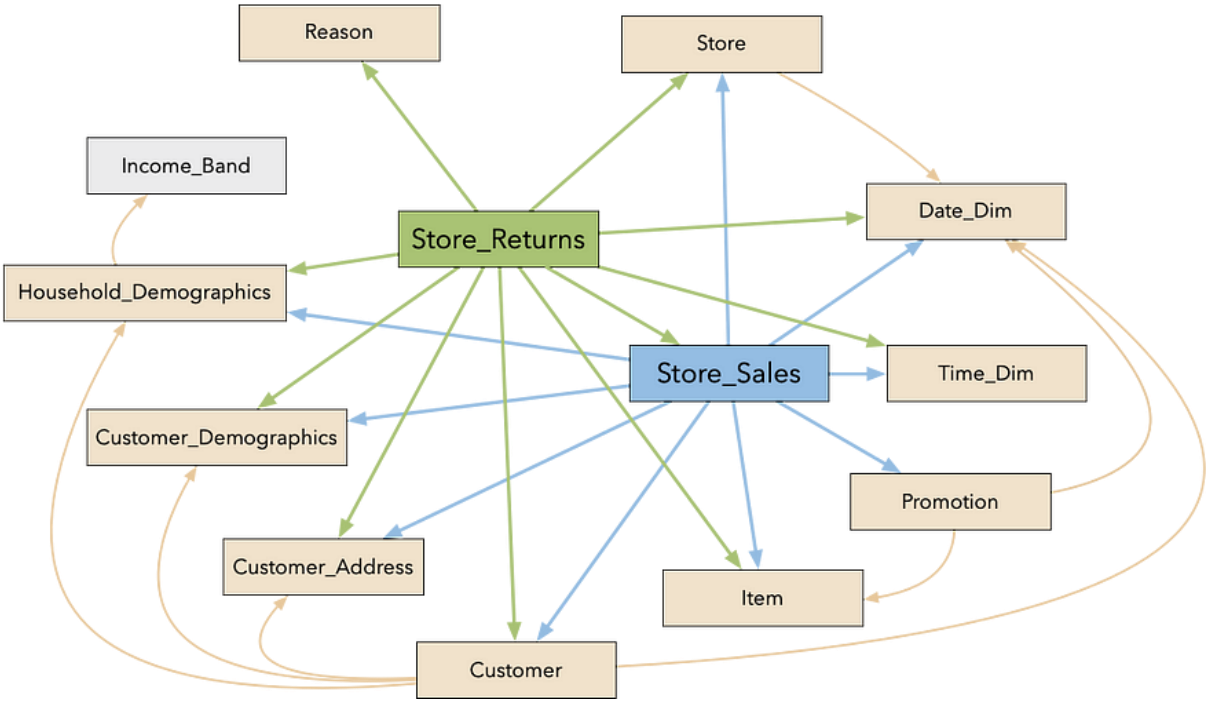
[20] G. User, “TPC-H Extension,” DuckDB. Accessed: Dec. 04, 2024. [Online]. Available: https://duckdb.org/docs/extensions/tpch.html

[21] M. Dowle, “Benchmarks : Grouping,” GitHub. Accessed: Nov. 30, 2024. [Online]. Available: https://github.com/Rdatatable/data.table/wiki/Benchmarks-:-Grouping

[22] J. Gorecki, “Database-like ops benchmark.” Accessed: Nov. 30, 2024. [Online]. Available: https://h2oai.github.io/db-benchmark/

[23] h2oai, *h2oai/db-benchmark*. (Nov. 24, 2024). R. H2O.ai. Accessed: Nov. 30, 2024. [Online]. Available: https://github.com/h2oai/db-benchmark
[24] T. Ebergen, “The Return of the H2O.ai Database-like Ops Benchmark,” DuckDB. Accessed: Nov. 30, 2024. [Online]. Available: https://duckdb.org/2023/04/14/h2oai.html
[25] DuckDB, “Database-like ops benchmark.” Accessed: Nov. 30, 2024. [Online]. Available: https://duckdblabs.github.io/db-benchmark/
[26] *duckdblabs/db-benchmark*. (Nov. 14, 2024). R. DuckDB Labs. Accessed: Nov. 30, 2024. [Online]. Available: https://github.com/duckdblabs/db-benchmark
[27] T. Ebergen, “Updates to the H2O.ai db-benchmark!,” DuckDB. Accessed: Nov. 30, 2024. [Online]. Available: https://duckdb.org/2023/11/03/db-benchmark-update.html
[28] MotherDuck, “pg_duckdb beta release : Even faster analytics in Postgres - MotherDuck Blog,” MotherDuck. Accessed: Dec. 07, 2024. [Online]. Available: https://motherduck.com/blog/pgduckdb-beta-release-duckdb-postgres/

Appendix A – TPC-DS database schema



Source: https://medium.com/hyrise/a-summary-of-tpc-ds-9fb5e7339a35

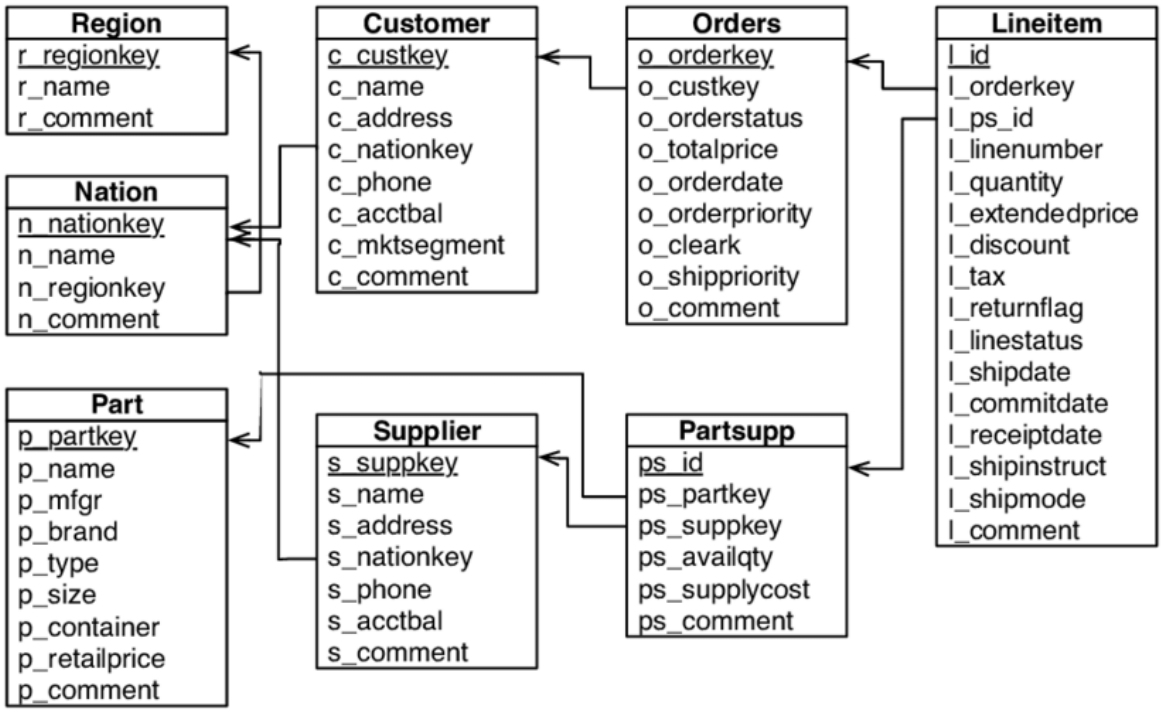
Appendix B – TPC-DS benchmark results

Query	Postgres [s]	DuckDB [s]	Ratio
1	timeout	0,097	
2	3,748	0,141	27
3	6,891	0,082	84
4	timeout	1,589	
5	4,945	0,201	25
6	timeout	0,162	
7	4,175	0,117	36
8	1,358	0,067	20
9	13,401	0,369	36
10	64,713	0,067	966
11	timeout	0,981	
12	0,536	0,042	13
13	3,056	0,122	25
14	timeout	1,356	
15	1,176	0,052	23
16	timeout	0,031	
17	timeout	0,099	
18	4,094	0,190	22
19	3,271	0,074	44
20	2,224	0,053	42
21	10,645	0,037	288
22	158,019	1,838	86
23	156,759	1,112	141
24	5,430	0,171	32
25	timeout	0,077	
26	4,080	0,096	42
27	4,674	0,250	19
28	28,235	0,295	96
29	11,100	0,142	78
30	timeout	0,123	
31	43,010	0,162	265
32	250,402	0,011	22764
33	6,754	0,074	91

Query	Postgres [s]	DuckDB [s]	Ratio
34	4,232	0,101	42
35	timeout	0,305	
36	8,791	0,250	35
37	18,750	0,043	436
38	45,684	0,262	174
39	108,574	0,280	388
40	3,121	0,056	56
41	33,772	0,031	1089
42	4,282	0,041	104
43	8,448	0,089	95
44	16,202	0,118	137
45	1,506	0,095	16
46	6,077	0,168	36
47	58,687	0,528	111
48	12,484	0,149	84
49	17,403	0,129	135
50	8,067	0,175	46
51	33,327	2,090	16
52	3,868	0,049	79
53	4,718	0,078	60
54	timeout	0,103	
55	3,911	0,048	81
56	8,690	0,111	78
57	24,801	0,359	69
58	9,338	0,090	104
59	37,343	0,289	129
60	8,755	0,127	69
61	7,974	0,125	64
62	6,321	0,063	100
63	4,446	0,069	64
64	18,913	0,545	35
65	26,490	0,415	64
66	5,969	0,183	33

Query	Postgres [s]	DuckDB [s]	Ratio
67	85,954	4,063	21
68	4,034	0,200	20
69	17,325	0,149	116
70	22,054	0,189	117
71	7,351	0,246	30
72	44,301	0,414	107
73	3,864	0,106	36
74	timeout	0,842	
75	28,367	0,446	64
76	9,813	0,162	61
77	9,212	0,093	99
78	26,832	1,856	14
79	4,845	0,180	27
80	11,426	0,328	35
81	timeout	0,262	
82	19,376	0,080	242
83	1,169	0,089	13
84	0,929	0,099	9
85	5,123	0,124	41
86	4,751	0,121	39
87	47,340	0,379	125
88	32,602	0,359	91
89	6,456	0,164	39
90	2,888	0,027	107
91	0,735	0,036	20
92	148,510	0,047	3160
93	7,959	0,275	29
94	timeout	0,089	
95	timeout	0,730	
96	4,541	0,035	130
97	14,309	0,381	38
98	5,216	0,494	11
99	14,206	0,099	143

Appendix C – TPC-H database schema



Source: https://www.researchgate.net/figure/The-TPC-H-database-schema_fig3_315535249

Appendix D – db-benchmark database schema

The datasets need to be generated using the scripts in the repository (_data/ folder). All data is nonsensical but has to adhere to certain configuration parameters:

- amount of tuples N: typical values are
  - 1e7: corresponds to datastets of 500MB
  - 1e8: corresponds to datastets of 5.000MB
  - 1e9: corresponds to datastets of 50.000MB
  - for this benchmark values 3e8, 5e8, 7e8, 8e8 were also used to generate different datasets but processing limits were reached at 3e8 already.
- cardinality K: typically chosen as 1e2 (N >> K)
  - Some columns are sampled from values in [1, K] leading to large groups.
  - Other values are sampled from values in [1, N/K] leading to small groups.
- amount of NA (percentage)
  - A first test with 5% NA lead to many libraries not performing the queries
  - For the final tests 0% NA were accepted in the test data
- order of the data: options are sorted or random
  - The benchmark was performed on randomy ordered data only

The join data has additional features

- The left dataset contains N tuples, each with 6 fields
- The right dataset varies both row and column dimension to allow for queries of different complexity. There is a small, medium and large variant.
- 10% of tuples are non-matching
- There are no duplicate ids on the matching rows

The data schema and sample is given below:

'groupby' data								
id1	id2	id3	id4	id5	id6	v1	v2	v3
(str)	(str)	(str)	(int)	(int)	(int)	(int)	(int)	(float)
id016	id016	id0000042202	15	24	5971	5	11	37,21
id039	id045	id0000029558	40	49	39457	5	4	48,95
id047	id023	id0000071286	68	20	74463	2	14	60,47
id043	id057	id0000015141	32	43	63743	1	15	7,69
id054	id040	id0000011083	9	25	16920	2	9	22,86

'join' data (left)						
id1	id2	id3	id4	id5	id6	v1
(int)	(int)	(int)	(str)	(str)	(str)	(float)
10	5159	8771892	id10	id5159	id8771892	91,83
7	8469	6597185	id7	id8469	id6597185	48,15
6	7510	10742226	id6	id7510	id10742226	53,13
10	3114	4468842	id10	id3114	id4468842	96,49
3	6139	7725507	id3	id6139	id7725507	56,10

'join' data (right L)						
id1	id2	id3	id4	id5	id6	v2
(int)	(int)	(int)	(str)	(str)	(str)	(float)
3	10245	8387993	id3	id10245	id8387993	5,56
7	5594	3737573	id7	id5594	id3737573	38,50
8	4039	1827242	id8	id4039	id1827242	12,20
6	8636	7628870	id6	id8636	id7628870	29,46
6	2343	7508068	id6	id2343	id7508068	76,45

'join' data (right M)				
id1	id2	id4	id5	v2
(int)	(int)	(str)	(str)	(float)
6	6026	id6	id6026	27,0767
1	4272	id1	id4272	9,35905
9	9803	id9	id9803	81,1463
10	1432	id10	id1432	48,136
10	5851	id10	id5851	15,4546

'join' data (right S)		
id1	id4	v2
(int)	(str)	(float)
4	id4	40,5468
6	id6	20,4259
9	id9	79,7015
1	id1	79,0451
8	id8	10,0004

Appendix E – db-benchmark queries

The queries are given below in simplified terms. For full syntax (dependant on the solution) please consult the official report: https://duckdblabs.github.io/db-benchmark/

There are 5 basic groupby queries:

- `sum v1 by id1`
- `sum v1 by id1:id2`
- `sum v1 mean v3 by id3`
- `mean v1:v3 by id4`
- `sum v1:v3 by id6`

These are followed by 5 advanced groupby queries:

- `median v3 sd v3 by id4 id5`
- `max v1 - min v2 by id3`
- `largest two v3 by id6`
- `regression v1 v2 by id2 id4`
- `sum v3 count by id1:id6`

There are 5 join queries, joining the left table on differently sized right tables

- `small inner on int`
- `medium inner on int`
- `medium outer on int`
- `medium inner on factor`
- `big inner on int`