SQLBigBench: Benchmarking SQL based query engines

Saumitra Chaskar
chaskar2@illinois.edu

Kinjalk Parth
kinjalk2@illinois.edu

Jai Agarwal
jaia4@illinois.edu

Shubham Thakar
sthakar3@illinois.edu

Rishi Mundada
rishirm3@illinois.edu

Abstract

When it comes to Online Analytical Processing (OLAP), choosing the right SQL query engine, storage format, and configuration settings can make a significant difference in system performance, especially when it comes to reducing latency. However, there is currently no standard framework in the industry to help users select the best combination of these elements for their specific needs.

To address this gap, we conducted an extensive benchmarking study on four popular SQL query engines: Hive, Spark, Presto, and Flink. We evaluated their performance using a range of synthetic and real-world datasets, query workloads, and common storage formats such as Parquet, ORC, and Iceberg, which we identified as potential candidates for our analytics framework.

Our goal was to investigate how different combinations and configurations of these components affect query processing efficiency and determine the most effective setups for various types of query workloads. By analyzing individual and combined query execution times, the impact of storage formats and configurations on performance, and each query engine's ability to handle specific types of query operations (e.g., filtering, aggregation, sorting), we developed a novel framework to guide users in selecting the optimal query engine or combination of engines based on their unique workload characteristics.

This framework aims to simplify the process of optimizing OLAP system performance by providing clear recommendations tailored to specific use cases, ultimately helping users achieve faster query processing and improved overall efficiency.

1 Introduction

At DeltaSigma, an investment banking and asset management firm, we initially relied on Greenplum MPP DB [23] for our data infrastructure. However, as our data volume and consumer base grew, Greenplum's limitations in scalability, concurrent access, autonomy, and licensing costs became apparent. Transitioning to a big data warehouse became necessary to manage large data repositories, scale SQL platforms for ad hoc analytics, and avoid vendor lock-ins with open-source components.

While Hadoop [1] serves as an open-source solution for data storage, we're actively exploring distributed SQL query engines native to Hadoop to fulfill scalability and autonomy requirements. A SQL Query Engine efficiently accesses petabyte-scale data from various sources through a SQL interface, separate from the storage layer.

Existing academic and industry evaluations of data systems often overlook comprehensive evaluations of query engines alongside storage configurations. Academic studies focus on synthetic datasets, failing to generalize to real-world data distributions and query workloads. Industry evaluations are often use-case specific, neglecting broader applicability and resource constraints common in commercial settings. There is a need for a structured benchmarking process which is comprehensive and addresses prevalent problems related to selecting appropriate query engines.

In this study we benchmark 4 popular [2] SQL query engines Apache Hive, Apache Flink, Spark SQL and Presto SQL engine as these query engines offer native integration with HDFS and provide powerful SQL-based analytics capabilities. This benchmarking study tries to identify a suitable SQL query engine that would be an ideal candidate for supporting our unique analytic workload. Since storage strategies have significant performance implications, along with a suitable SQL query engine we try to carve out a compatible storage configuration setup by evaluating our query workloads on data stored in different types of Hadoop compatible storage formats, compression techniques. In addition we also try to assess the impact and benefits of adding managed table format layer called Apache Iceberg. We expose our benchmarking suite as an extendable test bench for other organizations or users wanting to test existing and new SQL query engines on a widely accepted decision support benchmark or onboard their own analytical datasets and workloads.

2 Related Work

Many early academic studies evaluate SQL query engine performance [8, 13, 18, 19, 22, 24], focusing on engines like Shark, Hive, Tez, and Impala using Pavlo [24] and TPC-DS [25] synthetic datasets. While they establish performance baselines, they lack justification for engine performance and tuning guidelines. These studies highlight configurations like storage [16] and data distribution's impact. Some suggest ensemble query engines [20, 27], maintaining multiple engines and routing queries based on past performance, yet this approach is maintenance-intensive. Work on storage configurations' impact [17, 21] focuses on Spark's performance with different formats, neglecting combinations of storage configurations and engines on real-world data. These evaluations often overlook practical aspects like cost, maintainability, scalability, dataset distribution, and workload fit, crucial for organizational adoption.

We extensively reviewed industry benchmarking evaluations, mainly found in blog posts, documentations, and vendor success stories. One notable study at Nielsen NMC [15] benchmarked Hive, Spark, Snowflake, and Presto against Parquet, with Spark showing scalability and low latency but lacking reliability under heavy workloads, while Presto exhibited high memory usage. Another blog post assessed query engine choices alongside dbt workloads [14], favoring DuckDB in 75% of cases but not fully utilizing Spark and Presto's distributed capabilities. Additionally, a study [11] examined Parquet, ORC, and Iceberg for query workload characteristics but lacked broader applicability.

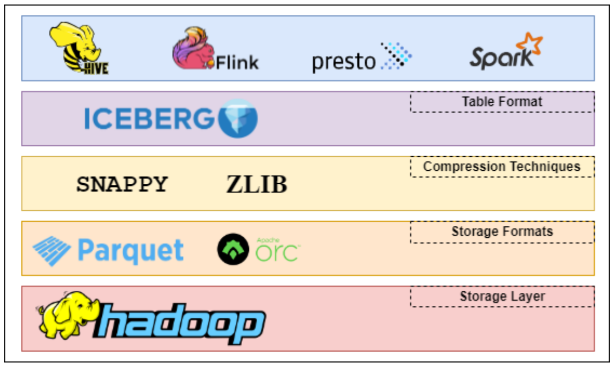

Figure 1: Benchmarking components

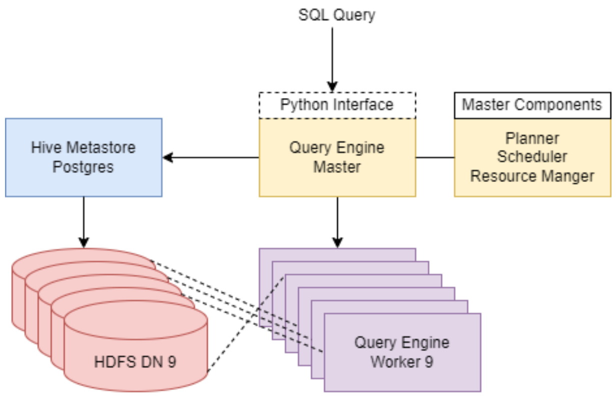

Figure 2: Example Query Engine benchmarking setup

These evaluations were highly contextual and couldn't be easily extended for further studies. Our findings reveal a gap in comprehensive benchmarks and decision-making frameworks, highlighting the need for systematic approaches to guide organizations in selecting optimal query engines and storage formats, thus enhancing OLAP system performance and deriving greater value from data analytics.

Our research introduces a novel benchmarking study evaluating four popular SQL query engines—Hive [3], Spark [4], Presto [7], and Flink [9]—across diverse datasets, workloads, and storage formats like Parquet [6], ORC [5], and Iceberg [10]. By systematically analyzing their performance, we aim to change the way how organizations select query engines and storage formats. Our comprehensive framework aids decision-making, enabling tailored engine choices for specific workload characteristics.

To address the limitations of existing benchmarks, we not only evaluate on synthetic datasets but also on domain-specific data and workloads. Additionally, our platform allows users to onboard their own datasets and queries for benchmarking or prototyping studies.

The paper is organized as follows: we discuss our system architecture, benchmarking approach, and present evaluation results, followed by discussion of our findings.

3 Methodology

3.1 System Architecture

Our process for benchmarking has been designed to provide a thorough evaluation of the selected query engines. By leveraging the Hadoop Distributed File System (HDFS) as the storage layer, we explore the performance implications of various table storage formats in conjunction with each query engine. This approach allows us to not only assess the raw performance of each engine but also to understand how different storage formats influence query execution times and overall system efficiency. Through this detailed analysis, our framework offers actionable insights into the most suitable query engine and storage format combinations for specific datasets and workloads.

Our benchmarking process adheres to a modular, layered design approach to facilitate the evaluation of diverse query engines. This architectural paradigm enables a plug-and-play functionality, allowing seamless integration and substitution of individual components as desired. Figure 1 illustrates the four distinct layers that comprise of our benchmarking framework.

In the following sections, we will provide a details of each layer and components.

3.1.1 Data Storage Layer

Our benchmarking system relies on Hadoop Distributed File System (HDFS) for robust and scalable data storage. HDFS's distributed nature enables parallel query execution, crucial for our workload. Leveraging HDFS's features like automatic data replication and distributed processing ensures data reliability and high availability. This allows us to focus on benchmarking tasks without concerns about storage infrastructure.

3.1.2 Data Format Layer

The data format layer abstracts underlying data representation, facilitating evaluation of query engines across various formats. Positioned between HDFS-stored data and the table format layer, it supports multiple formats like Parquet[6] and ORC[5]. These formats, both column-oriented, offer efficient compression, schema evolution support, and high performance. Parquet is favored for large, complex datasets in tools like Apache Spark, while ORC is preferred in Hive-based frameworks, with full ACID transaction support. We will compare the effects of these storage formats on query execution times.
.

3.1.3 Data Compression Techniques

The data format layer additionally supports multiple compression algorithms and formats such as Snappy and ZLIB. This allows for analyzing the balance between storage efficiency and query performance, aiding in selecting optimal data formats and compression settings
Snappy and ZLIB share fast compression and decompression speeds but differ in compression ratios. Snappy offers quicker processing but lower compression, suitable for real-time systems. Conversely, ZLIB provides higher compression at the expense of slower speeds, making it preferable for storage-intensive tasks like archiving.

We do not fine tune storage format and compression algorithm configurations but use the configurations derived in the study done by Zeng Et al. [28]

3.1.4 Table Format Layer

Our benchmarking system extends beyond the data format layer to include a table format layer, focusing on data organization and representation. We use Iceberg as our table format and evaluate its effect on query execution time.

Icebergs versioning capability ensures reproducibility and consistency in data operations making it a very popular table format. It also supports table partitioning and indexing, facilitating optimized data retrieval and query performance. We anticipate that Iceberg's capabilities can contribute to increasing query execution efficiency, making it a valuable layer in this study.

3.1.5 Query Engines

At the topmost layer of our benchmarking system architecture, we have the query engines layer. This layer serves as the entry point for evaluating the performance and capabilities of the various query engines under investigation.

A query engine put simply, is a dedicated piece of software that interprets SQL commands issued by a user and creates an execution plan which is used by worker nodes in a distributed cluster to access the data from the storage layer efficiently. We have chosen four widely adopted query engines in the industry for our benchmarking - Hive, Presto, Spark and Flink.

Hive[3] is built on top of Hadoop and provides HiveQL, which is a SQL-like interface for querying data stored in our HDFS layer. It reduces every query into a series of MapReduce jobs which are then executed on the nodes in our cluster. We chose Hive because it performs well on batch processing tasks and can handle large datasets, which makes it a good fit for our analytical workloads

Presto[7] is a query engine which supports fast, interactive analytical queries at a petabyte scale. Unlike Hive, Presto does all its processing in the memory and pipelines it across the network between stages. This avoids unnecessary I/O and associated latency overhead. We chose Presto because of its ability to query data from multiple sources and provide low latency queries which can greatly benefit data exploration, ad-hoc analysis, and real-time reporting scenarios in our use cases.

SparkSQL[4] offers a robust solution for structured data processing on a large scale. Similar to Presto it also performs in-memory computation. This feature is particularly advantageous for iterative data processing and complex query execution tasks prevalent in data analytics. Unlike Presto's legacy rule-based engine, Spark SQL's Catalyst optimizer leverages cost-based optimization techniques. This dynamic approach significantly enhances query performance by optimizing both logical and physical plans. These capabilities make Spark SQL a versatile tool for comprehensive data analysis, adeptly handling real-time and batch processing across various datasets and computational scenarios.

Apache Flink[9] is a unified stream processing and batch processing engine. Flink's query optimizer works to minimize data shuffling and network traffic by employing techniques such as pipelined execution, operator chaining, and dynamic task scheduling. We chose Flink because of its stream processing capabilities, event-time processing, exactly-once semantics, scalability, and fault tolerance which makes it a strong candidate for handling real-time data processing and continuous SQL queries for our workloads.

Incorporating diverse query engines in our framework ensures a thorough and unbiased evaluation. Analyzing Hive, Presto, Spark SQL, and Flink across different data formats enables users to make informed decisions for their production environments.

3.2 Benchmarking Steps

Evaluating large number of combinations of storage formats, compression techniques and a table format technique is a non trivial process. A naive benchmarking approach of considering all the possible combinations leads to a large number of candidate configurations to be evaluated, which is highly impractical and resource intensive. We take the advantage of decoupling between these configurations and follow a layered benchmarking technique where we fix components in a bottom up manner (storage format to query engine) to minimize the candidates to configure but still make right choice at every step leading to a more refined and optimal stack.

We use the following methodology to arrive at the best candidate query engine and associated storage configurations for optimizing query performance on the real world dataset.

(1) Run TPC-DS queries, clustered by their query characteristics, on tables with different *combinations of storage formats and compression techniques.*
(2) Identify the combination with the lowest query execution time across engines and freeze this configuration.
(3) Add the Iceberg layer on top of the frozen configurations to analyze its impact. If the *inclusion of Iceberg* leads to better performance, include it in the set of frozen configurations; otherwise, exclude it.
(4) Once the best candidate storage format, compression technique, and presence or absence of Iceberg layer are ascertained, use this to benchmark each SQL query engines on *concurrent or parallel queries.*
(5) Perform similar benchmarking as in Step 4 on the *real-world datasets* using the associated queries.
(6) *Check the level of conformity* between the results obtained from the evaluation conducted with synthetic datasets and real-world datasets.
(7) Arrive at the *best-performing SQL query engine* and associated *storage format.*

We elaborate on every step mentioned above and discuss their results in our evaluation section.

4 Evaluation

4.1 Experimental Setup

In this section, we elaborate on our benchmarking methodology designed to assess the performance of our system on analytical workloads

4.1.1 Dataset:

Our evaluation comprises two datasets: the widely used TPC-DS benchmark and a subset of our financial data for real-world simulation. For the TPC-DS dataset, consisting of 1GB across 7 fact tables, we selected 5 queries for each query characteristic, guided by their relevance to typical analytical scenarios. Additionally, our real-world dataset, 2GB in size with 5 tables, contains daily stock prices and financial information spanning 2015 to 2023. We hand-picked 3 queries per characteristic from our production workload to simulate real-world demands. These datasets enabled a comprehensive assessment of query engine performance across OLAP workloads, ensuring fair comparisons under similar data volume constraints.

4.1.2 Queries Analyzed:

We categorized the queries from the TPC-DS dataset into five broad categories:

(1) **Set operations:** Assess engines' efficiency in combining and manipulating multiple result sets using `UNION`, `INTERSECT`, and `EXCEPT` clauses.
(2) **Filter:** Test engines' ability to filter records using `WHERE` and `HAVING` clauses based on specified conditions.
(3) **Aggregations:** Evaluate performance in computing aggregate functions like `SUM`, `COUNT`, `AVG`, `MAX`, and `MIN` on large datasets.
(4) **Joins:** Measure engines' join optimization and handling of various join types such as `INNER`, `LEFT`, `RIGHT`, `FULL OUTER`, and `CROSS`.
(5) **Nested:** Assess handling of hierarchical data structures and recursive operations through subqueries or CTEs.

Table 1: Query Types and Numbers

Query Types	Query Numbers	SQL Clauses
Set operations	2, 14, 41, 80, 91	UNION, INTERSECT
Filter	1, 97, 101, 102, 103	WHERE, HAVING
Aggregations	6, 93, 96, 99, 100	GROUP BY, SUM
Joins	44, 53, 76, 79, 82	INNER JOIN, OUTER JOIN
Nested queries	8, 53, 83, 85, 92	SELECT * FROM (..)

This categorization allowed a systematic analysis of the engine's strengths and weaknesses across diverse query types and complexities typical of OLAP workloads. We chose 5 queries, representative of our workload, from each query type.

4.1.3 Hardware and Hadoop Setup:

We present our hardware setup in Table 2. This hardware setup was not particularly powerful, but it ensured that the performance evaluations were not heavily constrained by resource limitations, allowing us to measure the query engines' inherent capabilities and scalability without overwhelming the system with excessive resources.

We employed a Hadoop cluster comprising of 1 NameNode and 9 DataNodes on a cluster of 10 machines with the previously mentioned hardware specifications as the underlying infrastructure for running the benchmarks. This cluster served as the distributed computing environment for evaluating multiple query engines.

Table 2: Benchmarking Cluster Configuration

Parameter	Value
Processor	1 Core, Intel(R) Xeon(R) CPU@ 2.10GHz
RAM	4 GB
Disk Storage	100 GB
Operating System	Red Hat Enterprise Linux 8.9
Number of Nodes	10

4.1.4 Query Engine Configurations:

The query engines evaluated in this study were Hive (SQL-on-Hadoop engine), Presto (distributed SQL query engine), Spark (unified analytics engine), and Flink (stream and batch processing framework). These engines were deployed and executed on top of the Hadoop cluster, allowing us to assess their performance and capabilities in a distributed computing environment for handling large-scale data processing and complex analytical workloads commonly encountered in big data environments.
Hive: All query engines in our study connected to Hadoop via the Hive Metastore, hosted on Postgres for its ability to handle multiple concurrent connections. Exposed through a ThriftServer, it enabled access by other query engines. We utilized MapReduce as the Hive execution engine for its reliability, and YARN served as our resource manager.
Presto: The Presto cluster had 1 resource manager, 1 coordinator, and 9 worker nodes. We set query.max-memory to 5GB in config.properties for per-query memory limits and query.max-memory-per-node to 500MB for node memory usage. Additionally, timeout was configured as 10 minutes to prevent premature query timeouts. This balanced configuration aimed to ensure efficient execution while avoiding resource exhaustion.
SparkSQL: The SparkSQL was set-up with 1 master and 9 worker nodes, with default configuration as per [12] which we found to be working perfectly for us. Each node had a memory allocation of 500 MB.
Flink: Flink's batch processing performance was assessed with a cluster comprising 1 Job Manager and 9 Task Managers, each with 2 slots. We increased the network buffer to 200 MB to handle queries with large record outputs, enhancing throughput and resource efficiency for smooth operation under high demand.

4.2 Evaluations and Results

Evaluation Overview: As outlined in section 3.2, we follow a stratified benchmarking technique fixing query engine configurations starting with storage format, compression technique, iceberg configuration and lastly the query engine. Our evaluation consists of following steps: **Ascertain Storage Format and Compression Technique → Iceberg Evaluation → Concurrent Evaluation** The last step consists of two distinct evaluations on the TPC-DS Dataset and then on the Realworld Dataset. Each step above is elaborated in their own subsections below.

To simulate real-world use cases and for evaluation purposes, we used Python 3.11 with specific database connectors like PySpark for SparkSQL, PyHive for Presto and Hive, and PyFlink for Flink (used python3.8 due to compatibility issues with PyFlink). We wrote Bash scripts to run the Python code in parallel, leveraging the *&* operator at the end to promote parallel execution. For sequential execution, we did not need a Bash script; a simple for loop inside the Python code sufficed.

4.2.1 Ascertaining Storage and Compression Formats:

The first objective of the analysis is freezing our storage layer as it forms the base of our further study, We analyze the impact of storage and compression format combination by running our benchmarking queries on 6 combinations (2 storage formats, 3 compression techniques). Figures 5, 6, 7, and 8 present box plots comparing the relative latencies of various storage and compression combinations for each query engine.

Hive (Figure 5) exhibited the highest overall query latencies. We saw no significant difference in query latencies across different configurations. Hence these results were inconsequential for our first objective. However, the query latency was almost 10X than the other engines in our study - discussed later. This performance limitation is attributed to Hive's reliance on MapReduce, which introduces significant overhead from disk I/O, network transfers, and job setup time. Furthermore, Hive's lack of efficient indexing capabilities contributes to slower query performance. *Due to these factors Hive was excluded from further benchmarking.*

Flink (Figure 6) achieved the lowest average latency with the ORC storage format irrespective of the compression technique. This could be attributed to ORCs large stripe size of 250MBs which worked out well for the access patterns of queries in our workload and Flinks inherently optimized stream processing capabilities. However,while Flink's batch processing outperformed Hive, it did not match the query latency levels of Presto and Spark for our read-heavy analytical workloads. Future work may explore Flink's stream processing capabilities for real-time stock price analysis in high-frequency trading algorithms. *Consequently, Flink was also excluded from subsequent benchmarking.*

Presto (Figure 7) and **Spark** (Figure 8) demonstrated significantly lower latencies compared to Hive and Flink, with Parquet-GZIP yielding the lowest average latencies for sequential query execution in Presto while Spark performed more or less equally good on all storage configurations. Presto's performance on Parquet can be attributed to Parquet Page Indices [26]. While Spark has traditionally performed better on Parquet, but with recent introduction of ORC native vectorizer, the performance is seen at par with Parquet files. Both engines employ in-memory processing techniques, such as Spark's Resilient Distributed Datasets (RDDs) and Presto's materialized views, which minimize disk I/O. Given their superior performance, *Presto and Spark were selected for further benchmarking.*

Figures 3 and 4 compare storage consumption for the top three TPC-DS tables using no compression (baseline), GZIP, and Snappy. GZIP consistently achieved higher compression ratios than Snappy due to its use of Huffman coding.

While Spark was indifferent to storage configurations, Presto showed lower latency results on the Parquet format. Since GZIP offered a higher compression compared to Snappy, we finalized Parquet with GZIP as our storage configurations. However, more in-depth analysis on the intricate relations between storage configurations, datatypes, data access patterns and query latencies needs to be done to make a more data backed choice.

4.2.2 Query Engine behaviour on different query types:

During our sequential query execution carried out in the previous step, we noticed an interesting pattern in query latency variations across query engines for certain query types and decided to explore it further. We present this auxiliary result in Figure 15. We saw that for Set Operations and Filter Queries, Presto and Spark performed equally well with Spark having some outlier query execution latencies which can be attributed to its task distribution and resource management. For Join queries Presto performed considerably worse than Spark specially in cases where there were more than 3 relation being joined. This can be attributed to Presto's sub-optimal query plan generator and Spark's superior predicate push-down feature. So it's a good practice to analyse your query workload to determine your dominant query type and then choose the query engine which has the best performance over that workload.

4.2.3 Iceberg evaluation:

The evaluation in the previous step identified Parquet-GZIP as the optimal storage and compression format for the analytical workloads. Subsequently, we tested the effect of incorporating Iceberg on this configuration. As evident from Table 3, Iceberg introduces overhead to query execution. This overhead can be attributed to the features provided by Iceberg, such as snapshot isolation, schema evolution, and partition evolution. To facilitate these features, Iceberg maintains a metadata tree for each table, which grows with each write. Querying this metadata to determine which files to read can add latency, especially for tables with a large number of snapshots or partitions. Since the benchmarking dataset is a read-heavy workload and does not currently utilize features like snapshot history, the increased read latencies associated with Iceberg's metadata led to the decision *to exclude Iceberg from further benchmarking.*

4.2.4 Concurrent Query Execution:

In order to comprehensively assess concurrent query execution as they mimic the actual usage patterns over data in query engines setup on commercial datasets. We selected 1 representative query from distinct query types - 2, 1, 6, 44, 8 (refer Table 1) - and conducted executions across varied environments. Specifically, we performed individual executions, as well as executions alongside 5, 10, and 15 concurrent queries of all the different query types in our benchmarking study. Our observations indicate a pronounced disparity in latency escalation between Spark and Presto. For instance, as illustrated in figure 9, the latency increase in Spark, transitioning from individual to 5 concurrent queries, was nearly quadrupled, whereas the corresponding rise in Presto was merely 1.6 times as seen in figure 10. This discernible trend persists across diverse query types and is further validated by real-world data analysis which is represented in figure 11 and figure 12. Evidently, these findings underscore Presto's superior capability over Spark in managing concurrent queries effectively.

We noted that under the current configurations, Presto was able to execute all the queries in the 5,10,15 concurrent queries

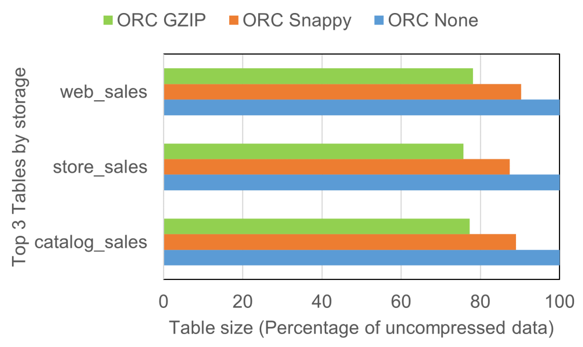

Figure 3: Effeciency of GZIP and Snappy compression technique on Parquet Tables

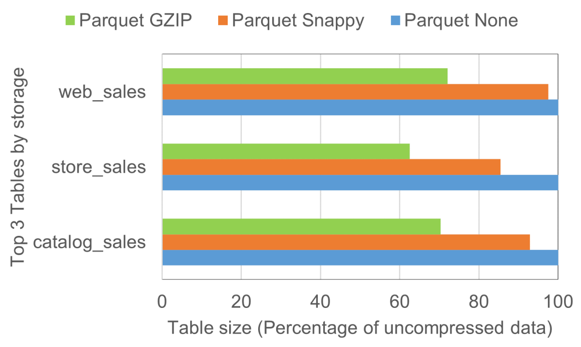

Figure 4: Effeciency of GZIP and Snappy compression technique on ORC Tables

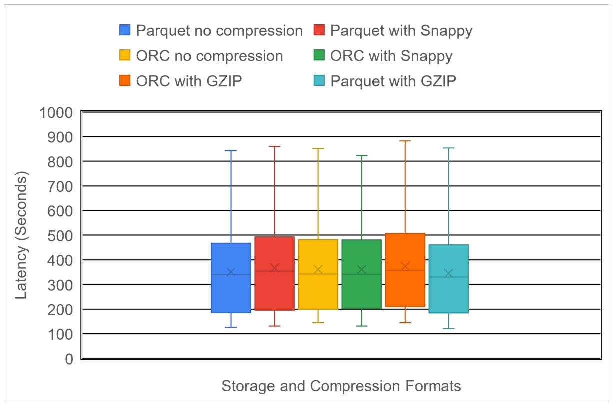

Figure 5: Hive query execution times with different storage and compression formats

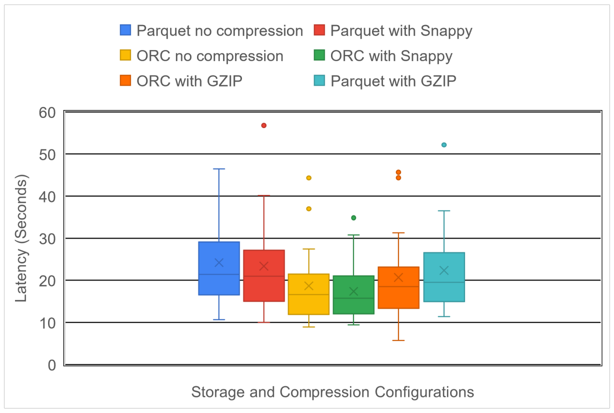

Figure 6: Flink query execution times with different storage and compression formats

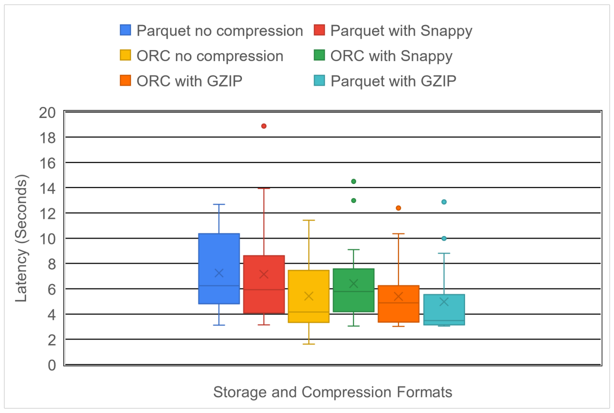

Figure 7: Presto query execution times with different storage and compression formats

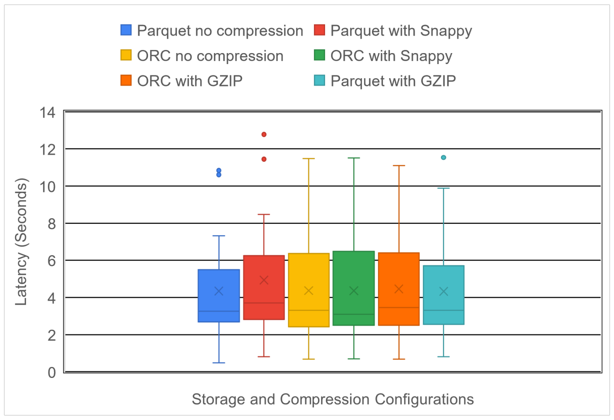

Figure 8: Spark query execution times with different storage and compression formats

workloads, Spark saw significant delay in execution and in the case of 15 concurrent queries it was able to complete 12-14 queries in each test run and failed with resource constraint issues. Presto was also able to scale better at the cost of high memory consumption, but was more resilient than Spark in terms of overall execution and failure recovery.

Query Type	Spark		Presto	
	Without Iceberg	With Iceberg	Without Iceberg	With Iceberg
Set operations	22.28	26.39	3.51	9.38
	25.47	29.39	3.27	10.21
Filter	3.30	3.91	3.13	6.54
	3.98	5.11	3.05	7.86
Aggregations	3.93	4.48	8.12	9.06
	17.57	20.06	3.19	4.63
Joins	5.90	7.55	3.04	3.98
	9.89	14.11	37.82	39.71
Nested queries	5.52	6.86	3.84	7.14
	6.80	8.40	3.86	12.70

Table 3: Representative execution times for queries with and without Iceberg for Parquet as storage and GZIP as compression formats

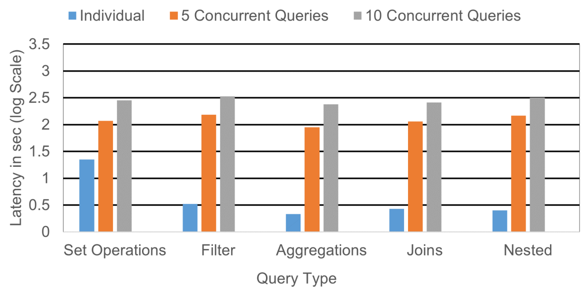

Figure 9: Spark: Individual and Concurrent execution time for a single query selected from each query type from the TPC-DS benchmark

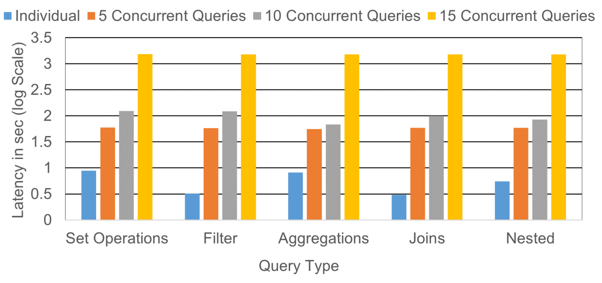

Figure 10: Presto: Individual and Concurrent execution time for a single query selected from each query type from TPC-DS benchmark

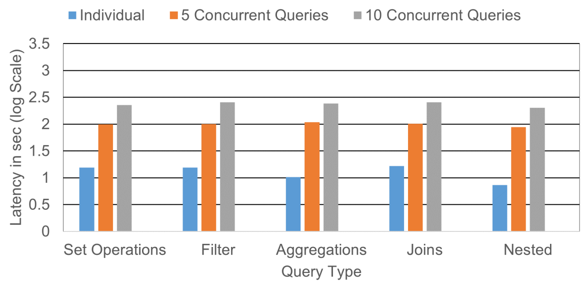

Figure 11: Spark: Individual and Concurrent execution time for a single query selected from each query type from the Realworld dataset

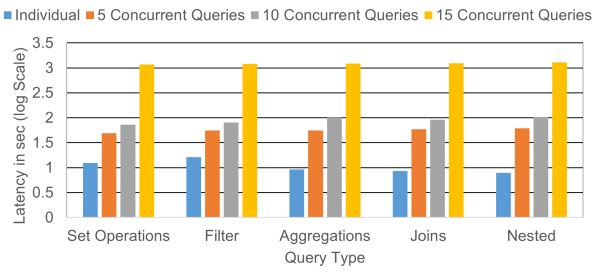

Figure 12: Presto: Individual and Concurrent execution time for a single query selected from each query type from the Realworld datasets

5 Conclusion

We were able to create a structured and extensive benchmarking structure which offers valuable insights into the performance and suitability of SQL-based query engines and storage formats for analytical workloads. By assessing various engines across diverse datasets and categorizing queries, we gained a deep understanding of their strengths and limitations.

Reliability, scalability, and completeness are crucial factors alongside query latency for our business needs. Presto emerged as the preferred choice, meeting all these requirements. While different engines showed minimal variance across storage formats, a deeper analysis is needed to understand the complex relationships between data storage formats, data distribution, access patterns, and query latencies to identify the ideal configuration for your workloads

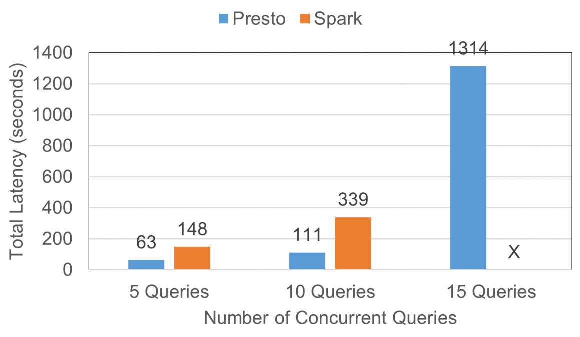

Figure 13: Concurrent query response time for 5, 10 and 15 queries from the Realworld Workload submitted simultaneously to the query engines

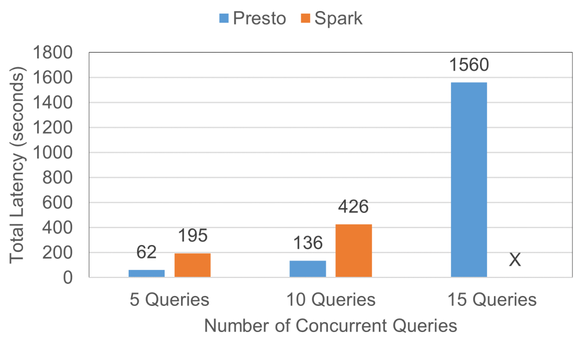

Figure 14: Concurrent query response time for 5, 10 and 15 queries from the TPC-DS Workload submitted simultaneously to the query engines

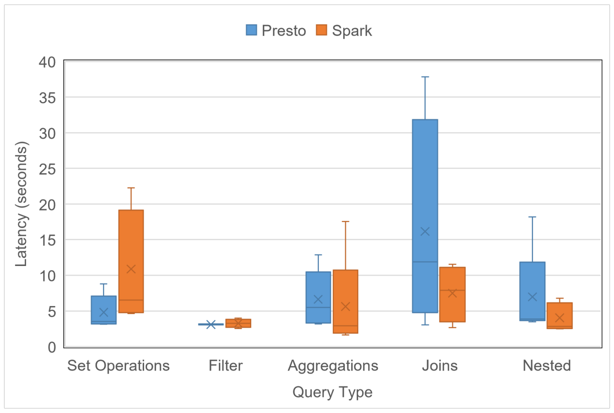

Figure 15: Query Latency distribution across Query Types for Presto and Spark

and datasets. Nonetheless, Presto exhibited slightly better performance on Parquet with GZIP compression, leading us to adopt this configuration.

These findings enabled us to make data-driven decisions on the most suitable query engine and storage configurations. Moreover, our benchmark offers a comprehensive framework for organizations to select optimal SQL-based engines and configurations for their OLAP systems, enhancing data analytics operations' efficiency and effectiveness.

6 Future Work

There are number of directions that we would like to explore in order to enhance this study. The most important one would be increasing the size and diversity of datasets. While we tried to mimic a resource constrained environment in our current setup for prototyping, we plan to extend this to match an actual production workload with increased storage and processing resources. The next direction that we would like to explore is one of our findings regarding indifference of query engines to storage configurations. To grant credibility to our findings and to bolster our framework, we would like to explore the intricate relations between storage formats and query engine throughput in greater detail and present our findings. Another interesting extension of this work is to include a Cloud based version of our benchmarking system for easy deployment and testing on popular cloud platforms as data warehousing and data management has seen a rapid movement from on-premise to cloud. On top of this in order to make the benchmark more accessible we would like to include standard skeleton configuration files in-order to onboard any new query engine to our platform with few lines of code.

References

[1] [n.d.]. Apache Hadoop. https://hadoop.apache.org/. Accessed: April 27, 2024.
[2] [n.d.]. DB-Engines Ranking. https://db-engines.com/en/ranking. Accessed: April 27, 2024.
[3] 2007. *Apache Hive.* https://hive.apache.org/
[4] 2010. *Apache Spark.* https://spark.apache.org/
[5] 2013. *Apache ORC.* https://orc.apache.org/
[6] 2013. *Apache Parquet.* https://parquet.apache.org/
[7] 2013. *PrestoDB.* https://prestodb.io/
[8] 2014. *Big Data Benchmark.* Retrieved Feb 15, 2024 from https://amplab.cs.berkeley.edu/benchmark/
[9] 2016. *Apache Flink.* https://flink.apache.org/
[10] 2018. *Apache Iceberg.* https://iceberg.apache.org/
[11] Ali Amin. [n.d.]. Mastering Data Storage: Iceberg, Parquet, or ORC Formats. https://www.linkedin.com/pulse/mastering-data-storage-iceberg-parquet-orc-formats-ali-amin/. Accessed: April 27, 2024.
[12] Apache Spark. [n.d.]. Apache Spark Configuration. https://spark.apache.org/docs/latest/configuration.html. Accessed: August 10, 2024.
[13] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797
[14] Author Name. [n.d.]. Head-to-Head Comparison of dbt SQL Engines. https://medium.com/datamindedbe/head-to-head-comparison-of-dbt-sql-engines-497d71535881. Accessed: April 27, 2024.
[15] Author Name. [n.d.]. SQL Query Engines: Intro and Benchmark. https://medium.com/nmc-techblog/sql-query-engines-intro-and-benchmark-44a658b47810. Accessed: April 27, 2024.
[16] Vladimir Belov and Evgeny Nikulchev. 2021. Analysis of Big Data Storage Tools for Data Lakes based on Apache Hadoop Platform. *International Journal of Advanced Computer Science and Applications* 12 (08 2021), 551–557. https://doi.org/10.14569/IJACSA.2021.0120864
[17] Vladimir Belov, Andrey Tatarintsev, and Evgeny Nikulchev. 2021. Choosing a Data Storage Format in the Apache Hadoop System Based on Experimental Evaluation Using Apache Spark. *Symmetry* 13, 2 (2021). https://doi.org/10.3390/

sym13020195

[18] Avrilia Floratou, Umar Farooq Minhas, and Fatma Özcan. 2014. SQL-on-Hadoop: full circle back to shared-nothing database architectures. *Proc. VLDB Endow.* 7, 12 (aug 2014), 1295–1306. https://doi.org/10.14778/2732977.2733002

[19] Ahmad Ghazal, Tilmann Rabl, Minqing Hu, Francois Raab, Meikel Poess, Alain Crolotte, and Hans-Arno Jacobsen. 2013. BigBench: towards an industry standard benchmark for big data analytics. In *ACM SIGMOD Conference.* https://api.semanticscholar.org/CorpusID:207202897

[20] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. 2016. MuSQLE: Distributed SQL query execution over multiple engine environments. *2016 IEEE International Conference on Big Data (Big Data)* (2016), 452–461. https://api.semanticscholar.org/CorpusID:16113085

[21] Todor Ivanov and Matteo Pergolesi. 2020. The impact of columnar file formats on SQL-on-hadoop engine performance: a study on ORC and Parquet. *Concurrency and computation : practice & experience* 32.2020, e5523 (2020). https://doi.org/10.1002/cpe.5523

[22] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Vasile Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Conference on Innovative Data Systems Research.* https://api.semanticscholar.org/CorpusID:2428443

[23] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2530–2542. https://doi.org/10.1145/3448016.3457562

[24] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. 2009. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 165–178. https://doi.org/10.1145/1559845.1559865

[25] Meikel Poess, Raghu Nambiar, and David Walrath. 2007. Why You Should Run TPC-DS: A Workload Analysis. 1138–1149.

[26] Presto. 2022. *Faster Presto Queries with Parquet Page Index.* https://prestodb.io/blog/2022/05/10/faster-presto-queries-with-parquet-page-index/

[27] Marco Vogt, Alexander Stiemer, and Heiko Schuldt. 2017. Icarus: Towards a multistore database system. In *2017 IEEE International Conference on Big Data (Big Data)*. 2490–2499. https://doi.org/10.1109/BigData.2017.8258207

[28] Xinyu Zeng, Yulong Hui, Jiahong Shen, Andrew Pavlo, Wes McKinney, and Huanchen Zhang. 2023. An Empirical Evaluation of Columnar Storage Formats. arXiv:2304.05028 [cs.DB]