Distributed Query Processing with PrestoDB: Performance Evaluation Across Heterogeneous Data Sources

Polyneikis Krimitsas
Department of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
el18037@mail.ntua.gr

***Abstract*—The exponential growth of digital data creates a pressing need for scalable systems capable of querying and analyzing heterogeneous datasets. PrestoDB, an open-source distributed SQL query engine, allows federated queries across multiple database technologies. This paper presents a performance study of PrestoDB using PostgreSQL, MongoDB, and Cassandra as data backends. The evaluation relies on the TPC-DS benchmark, under varying data distributions, query complexities, and cluster configurations. The results highlight the trade-offs between centralized and distributed strategies, emphasizing the role of worker scaling and partitioning schemes. The insights derived can guide the design of efficient architectures for large-scale analytical processing.**

***Index Terms*—Big Data, PrestoDB, Distributed SQL Queries, Query Optimization, PostgreSQL, MongoDB, Cassandra, TPC-DS**

I. Introduction

The rise of "Big Data" has transformed the way organizations manage information, posing challenges in storage, access, and analysis. Data sets ranging from terabytes to petabytes cannot be processed efficiently using traditional database systems. To address this, new generations of distributed and federated query engines have been developed to support high-performance analytics across heterogeneous storage systems.

In recent years, federated query engines have become increasingly important in enterprise settings where organizations maintain polyglot persistence environments. Unlike costly extract–transform–load (ETL) pipelines that introduce latency, engines such as PrestoDB allow analysts to query data directly across multiple systems without duplication. Compared to related technologies such as Trino, Apache Drill, and Spark SQL, PrestoDB emphasizes low-latency, interactive analytics rather than batch processing. Trino, as a community-driven fork of Presto, extends functionality with additional connectors, while Spark SQL integrates tightly with the Spark ecosystem but often incurs higher overhead due to materialization of intermediate results. Apache Drill provides schema-free SQL over semi-structured data but lacks the same maturity in cost-based optimization. These characteristics have driven adoption in leading companies such as Uber, Meta, and Adobe, where diverse data models must be unified seamlessly. From an application perspective, federated analytics is increasingly adopted in domains such as financial fraud detection, IoT sensor monitoring, and large-scale log analytics, highlighting PrestoDB's role as a lightweight yet powerful query layer for modern enterprises.

In this work, we investigate PrestoDB's query execution efficiency using a custom deployment with PostgreSQL, Cassandra, and MongoDB for data storing, as well as one coordinator and one worker node. The diverse storage models of these databases provide valuable insights into Presto's performance.

The TPC-DS benchmark was used for generating and loading the datasets, due to its ability to provide various lengths of records which spanned some gigabytes in size. The queries executed varied in complexity in order to measure performance metrics such as execution time.

II. Source Code

The full project is available on this GitHub repository, along with detailed instructions on how to set-up the environment and execute the code.

III. Technology Stack

A. Overview

Our environment consists of:

- **PrestoDB:** the distributed SQL query engine.
- **PostgreSQL:** a relational DBMS.
- **MongoDB:** a document-oriented NoSQL system.
- **Cassandra:** a wide-column NoSQL store.
- **TPC-DS Benchmark:** used for dataset and query generation.

B. PrestoDB

PrestoDB is an open-source, distributed SQL query engine originally developed at Facebook to provide interactive analytics over large-scale datasets. Unlike traditional database systems, PrestoDB does not store data itself but instead serves as a federated query engine that can connect to multiple underlying data sources. These data sources can range from relational databases such as PostgreSQL and MySQL to NoSQL

systems like Cassandra and MongoDB, as well as distributed file systems such as HDFS or object stores like Amazon S3. By abstracting these heterogeneous systems under a unified SQL interface, PrestoDB enables users to issue a single query that can access and join data across diverse backend servers. This flexibility makes it particularly suited for organizations with heterogeneous data environments.

At its core, PrestoDB follows a massively parallel processing (MPP) architecture, where queries are decomposed into stages and executed across a cluster of nodes. A Presto cluster consists of a coordinator and multiple workers. The coordinator, who can also function as a worker, is responsible for parsing SQL statements, generating query execution plans, and orchestrating task distribution. The workers, on the other hand, execute these tasks in parallel, processing data directly from the underlying storage systems. This division of responsibilities allows PrestoDB to scale horizontally by adding more worker nodes, improving performance for large and complex queries. Since the engine operates in memory and avoids writing intermediate results to disk whenever possible, it is optimized for low-latency, ad-hoc analytics rather than long-running batch jobs.

Query execution in PrestoDB involves several steps, beginning with SQL parsing and logical plan generation, followed by optimization and the creation of a distributed execution plan. The coordinator applies cost-based optimizations to determine efficient join strategies, task parallelism, and data distribution techniques. Tasks are then assigned to workers, which communicate with each other using a pipeline of operators such as scans, filters, joins, and aggregations. Each operator processes data in a streaming fashion, passing results downstream without materializing entire intermediate datasets. This pipelined execution model allows PrestoDB to achieve high throughput and fast response times, even when operating on terabytes of data across multiple sources.

As mentioned before, in comparison to other distributed query engines such as Apache Hive and Spark SQL, PrestoDB emphasizes low-latency, interactive analytics rather than batch-oriented processing. Hive, for example, was originally designed for batch ETL workloads on Hadoop and often incurs significant overhead due to its reliance on MapReduce or Tez for query execution. Spark SQL, while faster than Hive and capable of both batch and streaming workloads, typically materializes intermediate results in memory or on disk, which can introduce delays in highly interactive scenarios. PrestoDB, by contrast, is optimized for pipelined, in-memory execution, allowing it to return results quickly even for complex, multi-join queries. This makes it particularly effective in use cases where analysts need sub-second to few-second response times when querying large and heterogeneous datasets, bridging the gap between traditional databases and big data platforms.

Another important aspect of PrestoDB is its cost-based query optimizer. The optimizer considers factors such as join reordering, predicate pushdown, and parallelism when generating execution plans. By estimating the cardinality of intermediate results, it can choose between hash joins, broadcast joins, or partitioned joins to minimize data shuffling. PrestoDB also incorporates adaptive features, such as spilling to disk when memory thresholds are exceeded, ensuring robustness in the presence of large intermediate datasets. In distributed deployments, fault tolerance is achieved through task-level retries, which allow queries to continue despite transient node failures. These mechanisms, combined with PrestoDB's lightweight architecture, make it well-suited for cloud-native deployments on container orchestration platforms such as Kubernetes, where elasticity and resilience are critical.

C. PostgreSQL

PostgreSQL is an advanced open-source relational database management system (RDBMS) known for its robustness, extensibility, and adherence to SQL standards. Originally developed at the University of California, Berkeley, PostgreSQL has evolved into one of the most feature-rich and reliable database systems used in both academic and industrial settings. At its core, PostgreSQL follows a traditional client–server architecture where the server, called postmaster, manages database processes, handles client connections, and coordinates query execution. Queries are written in SQL, which PostgreSQL parses, validates, and optimizes through a cost-based query planner that determines the most efficient execution strategy using techniques such as indexing, joins, and parallel scans. Execution is then carried out by a set of operators that process tuples (rows) and return results to the client. PostgreSQL's transactional integrity is maintained through the use of Multi-Version Concurrency Control (MVCC), which ensures that multiple users can access and modify data concurrently without conflicts or locks blocking read operations. Beyond standard relational features, PostgreSQL supports advanced data types (JSON, arrays, hstore, geometric types), user-defined functions, and extensions, allowing it to handle both structured and semi-structured data efficiently. Its write-ahead logging (WAL) mechanism provides durability and crash recovery, while replication and sharding solutions extend its use to large-scale, high-availability environments. This combination of strong theoretical foundations and practical enhancements makes PostgreSQL a versatile choice for workloads ranging from transactional systems to analytical queries.

D. MongoDB

MongoDB is a leading open-source NoSQL database designed to provide flexibility, scalability, and high performance for modern applications that work with large volumes of semi-structured or unstructured data. Unlike relational systems such as PostgreSQL, MongoDB adopts a document-oriented data model, where records are stored as BSON (binary JSON) documents that can contain nested fields and arrays. This schema-less approach allows developers to store heterogeneous data without the need to define rigid table structures in advance, making it particularly suitable for rapidly evolving applications. Internally, MongoDB uses a client–server architecture in which queries are expressed in a JSON-like syntax and executed by a query engine that can leverage indexing,

sharding, and aggregation pipelines for efficient retrieval and transformation of data. To handle scalability, MongoDB supports horizontal scaling through automatic sharding, which distributes collections across multiple nodes in a cluster, while replication provides fault tolerance and high availability by maintaining multiple copies of the data. Concurrency control in MongoDB is achieved using a reader–writer lock system combined with journaling for durability, ensuring data consistency across distributed environments. The aggregation framework, inspired by functional programming paradigms, enables complex operations such as filtering, grouping, and joins within the document model. By balancing ease of use, dynamic schema design, and horizontal scalability, MongoDB has become a popular choice for applications that require fast iteration and the ability to manage diverse datasets at scale.

E. Cassandra

Cassandra is a highly scalable, distributed NoSQL database originally developed at Facebook and later open-sourced through the Apache Software Foundation. It is designed to manage extremely large datasets across clusters of commodity hardware while providing high availability and fault tolerance with no single point of failure. Cassandra follows a wide-column store model, where data is organized into keyspaces, tables, and partitions, with rows identified by primary keys and grouped by partition keys for efficient distribution. Unlike traditional relational systems, Cassandra does not enforce strict schema constraints, allowing columns to vary across rows in the same table. At the architectural level, Cassandra employs a peer-to-peer distributed design where all nodes in a cluster are equal, avoiding centralized coordination. Data is distributed across nodes using consistent hashing, and replication strategies (e.g., simple, network topology) ensure fault tolerance by maintaining multiple replicas of each partition. Queries are executed through the Cassandra Query Language (CQL), which resembles SQL but is optimized for denormalized, wide-column data models. To guarantee durability and high throughput, Cassandra uses a log-structured storage engine with commit logs, memtables, and SSTables, as well as a tunable consistency model where clients can choose between strong consistency and eventual consistency depending on application requirements. This flexibility, combined with linear scalability and fault-tolerant replication across multiple data centers, makes Cassandra particularly well-suited for mission-critical systems that require continuous up-time and the ability to handle high write and read throughput at scale.

F. TPC-DS

The TPC-DS (Transaction Processing Performance Council – Decision Support) benchmark is an industry-standard framework designed to evaluate the performance of decision support systems through a combination of complex queries and large-scale datasets. It models the data and workload of a retail decision-support environment, including sales, inventory, customer, and promotion information, providing a schema that spans multiple fact and dimension tables. The benchmark defines 99 parameterized SQL queries that represent a wide range of decision-support operations, including reporting, ad-hoc analysis, iterative queries, and data mining. These queries vary in complexity, involving multiple joins, aggregations, subqueries, and filtering conditions, making them representative of real-world analytical workloads. TPC-DS also incorporates scalability by allowing datasets to be generated at different scale factors, ranging from gigabytes to terabytes, enabling systems to be tested under varying data volumes. By standardizing the schema, queries, and metrics, the benchmark provides a fair and reproducible means of comparing different database engines and query optimizers. For systems like PrestoDB, TPC-DS serves as a rigorous testbed for assessing query execution efficiency, optimizer effectiveness, and scalability across diverse cluster and data distribution configurations.

IV. Installation and Configuration

The experimental benchmarking environment was deployed using containerization technology in order to ensure reproducibility, portability, and isolation from the host operating system. Docker[1] was selected as the container runtime, and Docker Compose was employed to orchestrate the deployment of multiple services through a declarative YAML file. This approach enabled the provisioning of a lightweight distributed cluster consisting of one query coordinator, one worker node, and three standalone database services: MongoDB, PostgreSQL, and Cassandra.

A. Docker Compose Setup

The `docker-compose.yml` file defines the cluster architecture, describing how each service is built, the networks used for communication, and the persistent volumes for configuration and logs. Networking was handled by Docker's internal bridge driver, which allowed containers to communicate through service names defined in the Compose file (e.g., `postgres`, `mongodb`, `cassandra`).

The two containers dedicated to the query engine followed the typical architecture of Presto/Trino: a coordinator node and one worker node. The coordinator was configured to accept incoming client queries, parse and optimize them, and schedule tasks. The worker was responsible for executing query fragments and returning intermediate results. This division of roles mimics real-world deployments where scalability is achieved by adding more workers.

B. Configuration Files

In addition to container orchestration, several configuration files were mounted into the query engine containers to control system behavior. These files are critical for distinguishing node roles and connecting external databases. The most relevant are described below.

[1] https://www.docker.com/

1) `jvm.config`: This file specifies the Java Virtual Machine (JVM) options used to launch the Presto process. Because Presto is a JVM-based system, memory management is particularly important. The following snippet illustrates a minimal configuration:

```
-Xmx4G
-XX:+UseG1GC
-XX:+ExitOnOutOfMemoryError
```

Here, the maximum heap size is set to 4 GB, the G1 garbage collector is enabled, and the system is instructed to terminate on an out-of-memory error. Such parameters prevent instability during large query execution.

2) `config.properties`: This file defines the role of each node (coordinator or worker) and sets the discovery and communication properties. For instance, the coordinator configuration may include:

```
coordinator=true
node-scheduler.include-coordinator=true
http-server.http.port=8080
query.max-memory=2GB
query.max-memory-per-node=512MB
query.max-total-memory-per-node=1GB
discovery-server.enabled=true
discovery.uri=http://presto-coordinator:8080
```

Meanwhile, the worker node configuration sets `coordinator=false` and points to the same `discovery.uri` so that it can register automatically.

3) `node.properties`: Each node requires a unique identity to avoid conflicts. This file provides such metadata:

```
node.environment=production
node.id=coordinator
node.data-dir=/var/presto/data
```

The `environment` parameter allows nodes to be grouped logically, while the `node.id` ensures uniqueness. The data directory stores local metadata and spill files generated during query execution.

4) Catalog Directory: The `catalog/` directory contains connector configuration files, each corresponding to a database to be queried and containing necessary information. The following entries are included in our environment :

- `postgres.properties`
- `mongodb.properties`
- `cassandra.properties`
- `tpcds.properties`

These configurations allow Presto to treat heterogeneous systems as catalogs, enabling queries to join and aggregate data across MongoDB, PostgreSQL, and Cassandra using standard SQL.

C. Database Containers

Each database container was instantiated from the official Docker images, ensuring standard configurations and minimizing the risk of incompatibility. Ports were exposed within the Docker network but not published to the host for security reasons. Data persistence was enabled through Docker volumes, allowing the benchmark datasets to survive container restarts.

D. System Architecture

Figure 1 illustrates the deployed architecture. The coordinator manages query parsing and scheduling, while the worker executes tasks. The three database containers run independently but are connected to the query engine through their respective catalog definitions. This design provides a unified SQL interface over multiple backends.

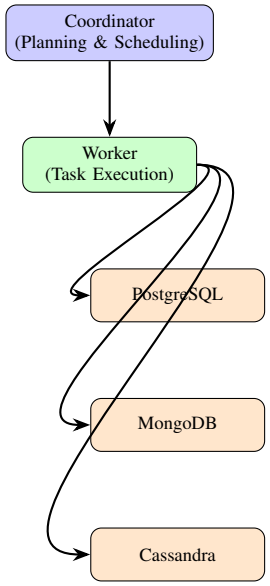

Fig. 1. The coordinator orchestrates the worker; the worker reads from each database.

E. Summary

By leveraging Docker and Docker Compose, the installation process was streamlined, reproducible, and portable. The modularity of Presto's configuration files (`jvm.config`, `config.properties`, `node.properties`, and `catalog`) provided fine-grained control over system behavior, while containerization allowed the databases to remain isolated yet easily accessible. This setup provided a robust foundation for executing benchmarking experiments across heterogeneous data sources.

V. TPC-DS Benchmarking Process

To evaluate query performance across different database engines, we adopted the **TPC-DS benchmark**, a widely recognized industry standard for decision support systems. The benchmark is composed of a set of SQL queries that simulate complex business intelligence workloads. These queries stress various aspects of a system, such as joins, aggregations, subqueries, and sorting, which makes TPC-DS particularly effective for testing the query planner, optimizer, and execution engine of analytical databases.

A. Benchmarking Process

The benchmarking workflow involved the following steps:

1) **Schema Generation**: Using the TPC-DS toolkit, the schema definition was generated at Scale Factor 1 (SF1). This corresponds to approximately 1 GB of data, making it manageable for experiments while still including realistic data distributions.
2) **Data Population**: Synthetic data was generated and loaded into the databases under evaluation. The data generation process follows the distributions defined by TPC-DS, ensuring non-uniformity and correlation between tables.
3) **Query Execution**: A subset of the 99 TPC-DS queries was executed in order to get information about different types of queries and compare the results.
4) **Metrics Collected**: Primary metrics included query runtime, throughput (queries per hour), and resource utilization (CPU, memory, disk I/O).

B. SF1 Schema

The TPC-DS schema at SF1 represents the data model of a large retail company operating in multiple sales channels: store, catalog, and web. It contains 24 tables, including both fact tables (e.g., `store_sales`, `web_sales`, `catalog_sales`) and dimension tables (e.g., `customer`, `item`, `date_dim`, `store`, `promotion`).

Fact tables store large transactional datasets, while dimension tables provide descriptive attributes for analysis. Fact tables reference dimension tables through foreign keys, enabling both star-schema and snowflake-schema queries.

C. ER Diagram

The schema can be visualized as a constellation schema, with central fact tables surrounded by multiple shared dimension tables. Figure 2 (referencing store returns table), Figure 3 (referencing catalog sales table), Figure 4 (referencing store sales table), Figure 5 (referencing catalog returns table) and Figure 6 (referencing web sales table) provide some insight on the schema.

D. Query Categories

The 99 TPC-DS benchmark queries are categorized into multiple classes that reflect real-world decision support operations. These include reporting queries that perform aggregations over large fact tables, iterative queries that refine results over multiple stages, ad-hoc queries that explore subsets of data with selective predicates, and deep analytics queries that involve complex joins and nested subqueries. The subset of queries selected in this study (Q1, Q10, Q13, Q19, Q35, Q54, Q76, Q94) spans these categories, ensuring coverage of both aggregation-heavy and join-intensive workloads. For example, Q1 and Q54 belong to reporting-style workloads dominated by group-by operations, while Q10 and Q35 are multi-join queries that stress join reordering and dimension lookups. Q19 and Q94 introduce nested subqueries, testing the optimizer's ability to rewrite them into efficient join plans. This diversity makes the chosen subset a representative sample for evaluating backend performance under PrestoDB.

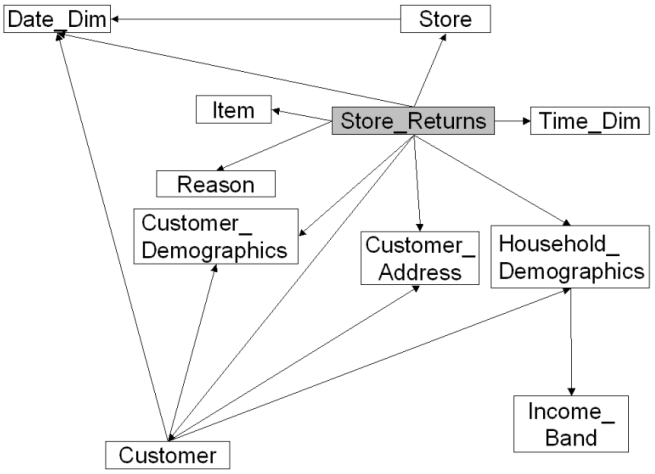

Fig. 2. Store Returns ER Sub-Schema.

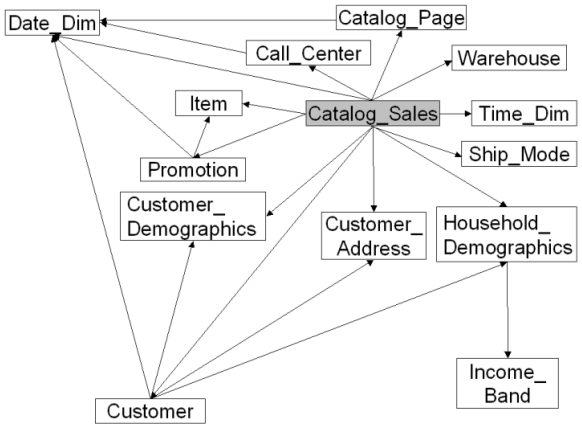

Fig. 3. Catalog Sales ER Sub-Schema.

VI. TPC-DS Benchmark Results and Analysis

We executed a representative subset of TPC-DS benchmark queries at Scale Factor 1 (SF1) on three database systems: Cassandra, MongoDB, and PostgreSQL. As mentioned above, the chosen queries (Q1, Q10, Q13, Q19, Q35, Q54, Q76, Q94) represent diverse workloads covering aggregations, joins, subqueries, and constellation-schema queries. The results provide comparative insights into query optimizer maturity, aggregation pipelines, and join performance.

A. Insights on Selected Queries

- **Q1**: Aggregation-heavy query over `store_sales` and `promotion`. PostgreSQL showed the fastest runtime (∼1.3s mean), followed by MongoDB (∼1.6s), while Cassandra lagged significantly (∼3.3s).
- **Q10**: Join-intensive query on `item` and `date_dim`. MongoDB (∼6.9s) and PostgreSQL (∼4s) both outperformed Cassandra (∼32.8s).

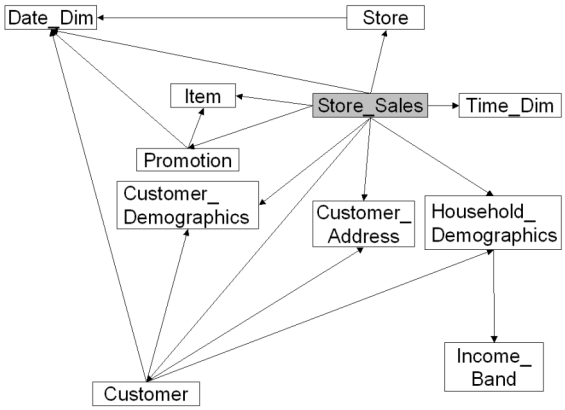

Fig. 4. Store Sales ER Sub-Schema.

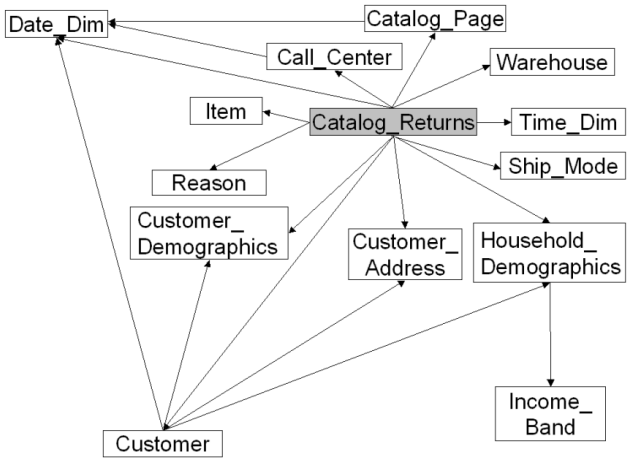

Fig. 5. Catalog Returns ER Sub-Schema.

- **Q13**: Customer-centric query with filters. Cassandra struggled (∼20.9s), MongoDB performed moderately (∼10.1s), and PostgreSQL achieved competitive runtimes (∼6s).
- **Q19**: Subquery-heavy workload. PostgreSQL (∼3s) and MongoDB (∼6.2s) both handled correlated subqueries well compared to Cassandra (∼11.9s).
- **Q35**: Multi-fact constellation query. PostgreSQL (∼3.5s) and MongoDB (∼8.1s) were far ahead of Cassandra (∼31.4s).
- **Q54**: Complex aggregation query. PostgreSQL (∼3s) led again, MongoDB (∼6.6s) was competitive, while Cassandra (∼23.8s) lagged.
- **Q76**: Semi-join and filtering query on web sales. PostgreSQL's runtime was excellent (∼0.9s), MongoDB performed moderately (∼3.1s), and Cassandra was much slower (∼22.4s).
- **Q94**: Multi-join with temporal filtering. PostgreSQL (∼1.5s) and MongoDB (∼2s) again vastly outperformed Cassandra (∼6.7s).

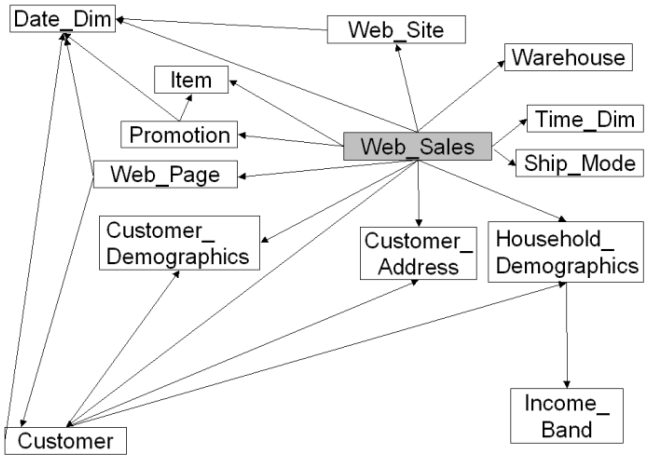

Fig. 6. Web Sales ER Sub-Schema.

B. Performance Observations

- **PostgreSQL** consistently delivered the best runtimes, dominating across all queries. Its optimizer efficiently handled both aggregation-heavy and join-heavy workloads.
- **MongoDB** showed solid mid-tier performance, often twice as slow as PostgreSQL but still significantly faster than Cassandra. It excelled in Q10 and Q13, reflecting reasonable support for joins and selective filters.
- **Cassandra** was consistently the slowest, particularly in join-heavy queries (Q10, Q35). Its architecture, designed for high write throughput and horizontal scalability, is less suitable for analytical workloads.

Variance analysis showed PostgreSQL and MongoDB delivered stable performance with relatively low standard deviation, while Cassandra exhibited larger fluctuations, especially in CPU utilization.

C. Summary of Query Runtimes

Table I summarizes the mean wall-clock runtimes (in milliseconds) for each of the evaluated queries across Cassandra, MongoDB, and PostgreSQL. These results highlight the consistent advantage of PostgreSQL, the intermediate performance of MongoDB, and the significantly slower execution in Cassandra for join-intensive queries.

TABLE I
MEAN WALL-CLOCK RUNTIMES (MS) FOR SELECTED TPC-DS QUERIES

Query	Cassandra	MongoDB	PostgreSQL
Q1	3267	1637	1322
Q10	32822	6906	3989
Q13	20917	10122	5957
Q19	11955	6224	3052
Q35	31392	8161	3482
Q54	23809	6560	3034
Q76	22387	3116	906
Q94	6747	2003	1547

D. Visualization of Results

Figure 7 compares per-query runtimes across the three systems, while Figure 8 shows average runtimes aggregated over all queries. Figure 9 highlights variability in runtimes, and Figure 10 presents process CPU time differences. Finally, Figure 11 provides a radar chart summarizing normalized relative performance.

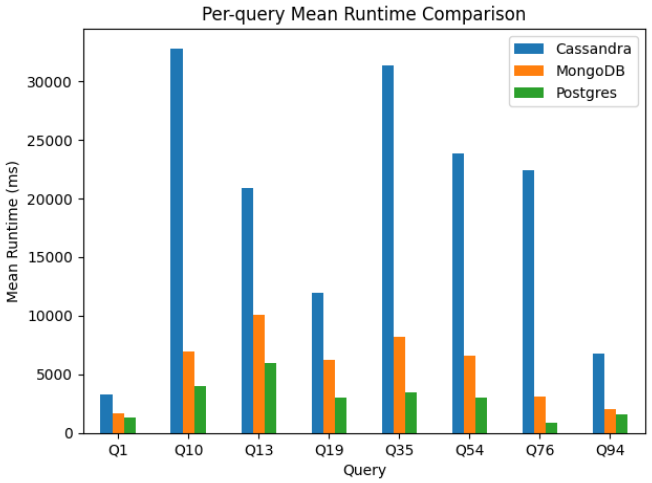

Fig. 7. Per-query mean runtime comparison across Cassandra, MongoDB, and PostgreSQL.

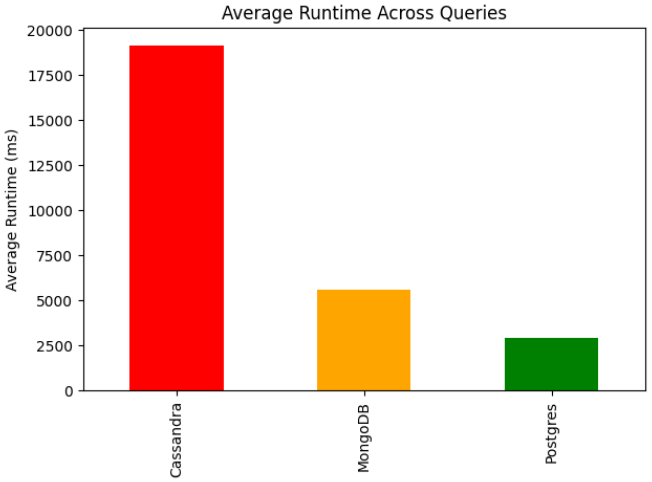

Fig. 8. Average runtime across all tested queries. Lower values indicate better performance.

E. Lessons Learned

Our experiments highlight several lessons relevant to practitioners and researchers. First, performance cannot be generalized across systems: query structure strongly influences which backend performs best. While PostgreSQL dominated overall, MongoDB proved competitive for selective joins, showing that document-oriented systems can still support analytical

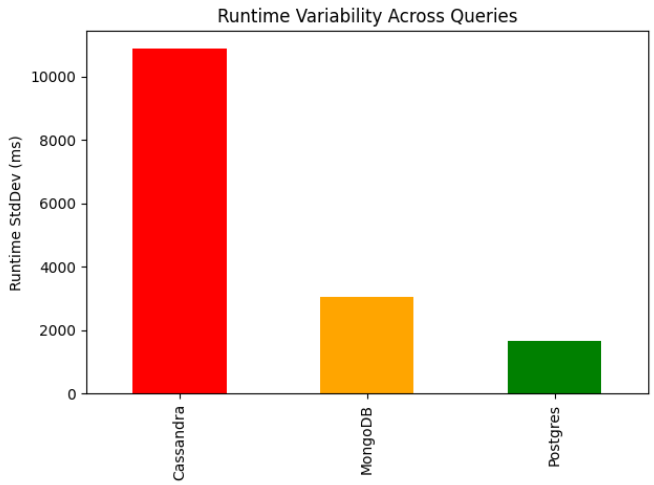

Fig. 9. Standard deviation of runtimes across queries, indicating stability of execution.

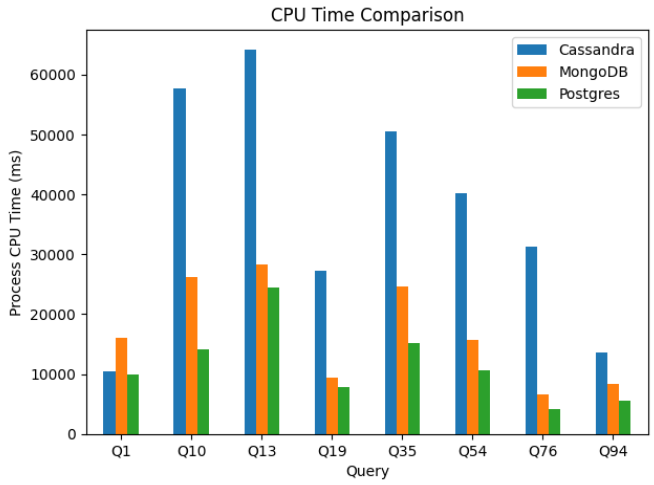

Fig. 10. Per-query process CPU time comparison across the three systems.

workloads when queries align with their strengths. Second, variance in runtime stability is an important metric often overlooked. Systems with lower average runtime variance, such as PostgreSQL, provide more predictable performance, which is critical for service-level agreements in production. Third, the limitations of Cassandra underscore the importance of aligning technology choice with workload profile. Although Cassandra excels in high-throughput OLTP scenarios, its lack of advanced query optimization makes it unsuitable for federated analytics. Finally, the role of PrestoDB as a federated query layer was validated: despite differences in backend performance, it successfully abstracted heterogeneous systems under a unified SQL interface, demonstrating its practicality for hybrid enterprise architectures.

F. Threats to Validity

While the experiments provide valuable insights, several factors may limit the generality of the results. First, the

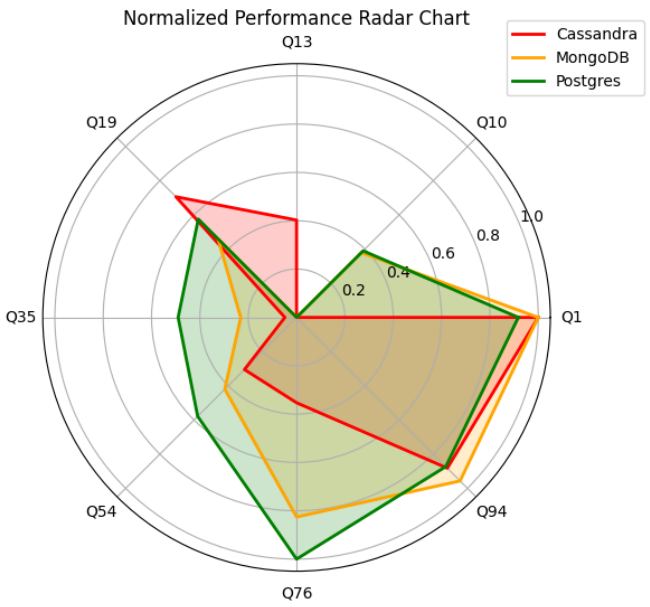

Fig. 11. Radar chart of normalized query runtimes, summarizing relative performance across all queries.

datasets were generated at Scale Factor 1 (approximately 1 GB), which, although representative for small-scale analysis, may not fully capture the behavior of systems at terabyte scale. Second, the experiments were executed in a Docker-based environment, which provides isolation and reproducibility but may introduce performance differences compared to bare-metal deployments. Third, the evaluation focused on a subset of eight TPC-DS queries, which, while diverse, do not cover the full spectrum of the 99 queries defined by the benchmark. Finally, the cluster size was limited to one coordinator and up to two workers, restricting conclusions about large-scale scalability. These constraints should be considered when interpreting the results.

G. Summary

From these results, we conclude:

1) **Relational systems outperform NoSQL for analytics.** PostgreSQL's dominance demonstrates the strength of mature optimizers for complex decision-support queries.
2) **Document stores offer a balance.** MongoDB was competitive in selective joins and filters, though less optimized than PostgreSQL for large aggregations.
3) **Wide-column stores are poorly suited for TPC-DS workloads.** Cassandra's poor join performance reflects its architecture, which favors OLTP and scalability rather than analytical query optimization.
4) **Scalability implications.** At higher scale factors (e.g., SF100), the gaps are likely to widen: PostgreSQL should remain strong, MongoDB will scale moderately, while Cassandra may degrade further under join-heavy workloads.

Overall, the benchmark highlights clear trade-offs. PostgreSQL is the most effective choice for analytical workloads, MongoDB offers flexibility with moderate performance, and Cassandra is unsuitable for TPC-DS style decision-support queries.

In summary, the TPC-DS benchmark provided clear evidence of trade-offs inherent in database system design. The observed performance differences illustrate how query engine architectures are tuned for specific classes of workloads, reinforcing the importance of aligning database choice with workload characteristics.

VII. Conclusion

This study assessed PrestoDB's performance under multiple backend configurations, focusing on the TPC-DS benchmark at SF1. Aside from what was covered above, the experiments also highlighted the benefits of distributed execution through PrestoDB. Distributing queries across multiple backends enabled parallel processing of large fact tables, which is essential for scaling decision-support workloads. Worker scaling reduced runtimes, especially in join-intensive queries where data shuffling could be parallelized but also added issues due to hardware limitations. However, the benefits diminished beyond two workers, suggesting that network overhead and coordination costs begin to outweigh the advantages of parallelism at smaller scale factors. This plateau effect indicates the need for careful cluster sizing in practice, balancing hardware resources with workload complexity.

From a practical perspective, these results reaffirm that database selection must be aligned with workload characteristics. PostgreSQL, as a mature relational engine, is most suitable for analytics requiring complex joins, aggregations, and subqueries. MongoDB can serve as a reasonable alternative when flexibility and semi-structured data handling are prioritized, but its query engine remains less optimized than PostgreSQL's. Cassandra, while not well-suited for TPC-DS style workloads, may still complement analytical systems in hybrid environments, where it can act as a scalable ingestion layer for high-velocity operational data while analytical queries are executed on other engines.

Future work can extend this study in several directions. First, experiments at larger scale factors (e.g., SF100 or SF1000) will provide deeper insight into scalability trends and the extent to which performance gaps widen under realistic data volumes. Second, enabling history-based optimization using caching layers such as Redis could reduce repetitive query costs and further improve throughput for recurring workloads. Third, exploring alternative partitioning and data distribution strategies in Cassandra could mitigate some of its limitations for analytical workloads, particularly in queries dominated by selective scans. Finally, extending the benchmark to hybrid configurations, where different query classes are routed to different backends, would provide a richer perspective on how multi-database ecosystems can be orchestrated to achieve both scalability and performance.

Overall, this study demonstrates that while PrestoDB provides a flexible platform for federated analytics, backend system design remains a decisive factor in performance. PostgreSQL remains the strongest choice for analytical queries, MongoDB offers a trade-off between performance and flexibility, and Cassandra's role is more limited for decision-support. Effective scaling through worker nodes improves performance up to a point, but future improvements will require deeper optimization of data placement and query planning strategies.

REFERENCES

[1] M. Bornea et al., "Presto: A Distributed SQL Query Engine for Big Data," in *Proc. ACM SIGMOD*, 2021.

[2] M. Roman, D. Phillips, and M. Seidman, "Trino: The Definitive Guide," O'Reilly Media, 2022.

[3] M. Armbrust et al., "Spark SQL: Relational Data Processing in Spark," in *Proc. ACM SIGMOD*, 2015.

[4] MongoDB Inc., "MongoDB Architecture Guide," 2023. [Online]. Available: https://www.mongodb.com/docs/manual/core/architecture/

[5] Apache Software Foundation, "Apache Cassandra Documentation," 2023. [Online]. Available: https://cassandra.apache.org/doc/latest/

[6] R. O. Nambiar and M. Poess, "The Making of TPC-DS," in *Proc. VLDB*, 2006.