Distributed Execution of SQL Queries with Presto

Χρίστος Κουστένης
Dept. of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
el20227@mail.ntua.gr

Χαράλαμπος Καμπουγέρης
Dept. of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
el20098@mail.ntua.gr

Δάφνη Γεωργιάδη
Dept. of Electrical & Computer Engineering
National Technical University of Athens
Athens, *Greece*
el20189@mail.ntua.gr

Team ID: 12

***Abstract* — As data continues to grow at an exponential rate, the need for efficient methods to process and analyze large datasets has become critical for maintaining competitiveness and driving productivity. Presto [1], a distributed SQL query engine, plays a vital role in querying large and diverse datasets. This study evaluates the performance of Presto across various data distribution strategies, cluster configurations, and query complexities. The focus is on three key databases—PostgreSQL, MongoDB, and Cassandra—used within the evaluation framework. Through a series of experiments, we test Presto's query execution under different conditions, adjusting both data distribution approaches and worker configurations. The performance is analyzed based on query execution times and optimization plans, offering insights into Presto's behavior under various operational settings. The findings provide guidance for optimizing data distribution strategies and improving query processing efficiency in distributed database environments.**

Keywords—Big Data, Presto, Distributed SQL Queries, PostgreSQL, MongoDB, Cassandra, Data Distribution, Query Optimization,

I. Introduction

Big Data refers to data sets whose scale, diversity, and complexity demand the development of new technologies, architectures, and algorithms for effective management and analysis. The massive amounts of data now available have dramatically transformed how organizations manage and derive insights, offering unprecedented context. However, the challenges associated with handling such large datasets include slower query performance and the high cost of management. To address these issues, organizations have invested in creating specialized tools designed to process and analyze big data more efficiently. PrestoDB is one such tool: an open-source, distributed SQL query engine tailored for querying large datasets spread across multiple data sources.

PrestoDB has gained considerable attention for its ability to perform federated queries across a variety of databases, including MySQL, PostgreSQL, Cassandra, and MongoDB, among others. Its flexibility in working with different data storage solutions has led to its widespread adoption by prominent companies like Uber [2], Meta, and Adobe. As a result, assessing its performance under various conditions is essential to fully understanding its capabilities.

In this study, we benchmark the performance of PrestoDB through a series of queries, considering factors such as data distribution and the type of storage systems used. Our PrestoDB setup consists of four nodes: one coordinator and three worker nodes, each utilizing a distinct storage system. The first worker runs PostgreSQL, the second uses Cassandra, and the third employs MongoDB. By leveraging these three different database management systems, we cover a wide range of data models, including relational, wide-column, and document-oriented models, providing a comprehensive perspective on query performance across different types of data storage.

We use the TPC-DS [3] benchmark to generate and load large datasets, which span several gigabytes in size and consist of millions of records, ensuring that the data cannot fit into memory. The queries executed on this dataset vary from simple SELECT statements to more complex queries that involve joins and aggregations across multiple tables. After loading the data into the respective data stores, we execute these queries and record key performance metrics.

From these initial measurements, we devise different strategies for distributing the data and assess PrestoDB's performance with varying numbers of workers and distribution plans. The goal is to understand how the distributed query engine and its optimization techniques perform when handling different amounts of data, data distributed across multiple engines, and queries of varying complexity.

The project is supported by a GitHub repository that contains all necessary scripts, along with detailed instructions for replicating our PrestoDB cluster, generating and loading the data, and running the benchmark queries. The repository also includes the results of our performance tests.

II. Source Code

The project is backed by a GitHub repository with scripts and step-by-step guide for our set up which are accessible through the following link:

Github Repository

III. Technology Stack

A. Overview

- **PrestoDB**: Was utilized as a distributed SQL query engine.
- **PostgreSQL**: Database management systems (DBMS) on second worker.
- **MongoDB**: DBMS on first worker.
- **Cassandra**: DBMS on third worker.
- **TPC-DS Benchmark**: Tool utilized for data and query generation.

B. PrestoDB

1) ***Overview:*** PrestoDB is an open-source, distributed SQL query engine built in Java, designed to efficiently query large datasets across multiple heterogeneous data sources. Originally developed by Facebook, PrestoDB was created as a high-performance alternative to query engines that relied on MapReduce pipelines, such as Hive and Pig, for processing data stored in the Hadoop Distributed File System (HDFS). Over

time, its capabilities have expanded beyond Hadoop, enabling it to query a wide range of data sources, including traditional relational databases like PostgreSQL, as well as NoSQL databases such as Cassandra and MongoDB.

One of PrestoDB's key strengths is its ability to execute federated queries, allowing users to seamlessly combine and analyze data from multiple sources within a single query. This makes it a powerful tool for organizations that manage diverse data infrastructures, as it eliminates the need for costly and time-consuming data migration. Additionally, PrestoDB is optimized for high-speed analytics and data warehousing workloads, supporting complex aggregations, interactive reporting, and real-time data exploration.

Designed for scalability and performance, PrestoDB operates in a distributed architecture where queries are executed across multiple worker nodes in parallel. This architecture ensures efficient resource utilization and enables PrestoDB to handle large-scale data processing with low latency. Its extensibility and broad compatibility have made it a popular choice for companies seeking a flexible and efficient solution for querying vast and distributed datasets.

2) **Nodes:** As a distributed SQL query engine, PrestoDB processes data in parallel across multiple servers, running on a cluster that consists of two types of nodes: a coordinator and workers. Users interact with the coordinator via their SQL query tool, and the coordinator is responsible for managing query execution by distributing workloads across worker nodes. The configuration for accessing various data sources is defined in catalogs, which specify parameters such as host, port, and authentication details.

The PrestoDB coordinator acts as the central orchestrator of query execution. It parses and optimizes SQL statements, creates execution plans, and assigns tasks to workers. The coordinator is the only node that directly interacts with clients, maintaining an overview of worker activity and ensuring efficient workload distribution. Additionally, it communicates with both clients and workers using a REST API. While the coordinator's primary role is query management, it can also be configured to function as a worker if necessary.

PrestoDB workers are responsible for executing query tasks and processing data. They retrieve data from configured connectors and exchange intermediate results among themselves as needed. The coordinator gathers these results, compiles the final output, and returns it to the client. Each worker runs in a separate JVM instance, enabling further parallelization at the thread level for efficient query execution. Workers communicate with each other and the coordinator via REST APIs, ensuring smooth data flow and optimized performance across the cluster.

3) ***Components:*** PrestoDB is designed to facilitate seamless querying across diverse data sources, overcoming the challenges of distributed query execution through a well-structured architecture. At the core of this architecture are connectors, catalogs, schemas, and tables, which work together to enable efficient data retrieval and analysis from multiple systems.

Connectors are the bridge between PrestoDB and various storage backends. They function as plugins that establish communication with different data sources, ranging from relational databases like PostgreSQL and MySQL to NoSQL systems such as Cassandra and MongoDB. By leveraging these connectors, PrestoDB can query data directly from multiple sources without requiring prior data transformation or migration.

A catalog in PrestoDB acts as a logical grouping of data sources. It contains metadata about schemas, tables, and the corresponding connectors that facilitate access to external systems. Since PrestoDB can maintain multiple catalogs simultaneously, users can query across distinct data environments without altering their SQL queries, making data integration more streamlined.

Within each catalog, schemas provide an additional layer of organization, allowing tables to be structured efficiently. In relational databases like PostgreSQL, schemas function as namespaces that logically separate tables. When interacting with external systems through PrestoDB, schemas help maintain consistency and structure across various data sources.

At the most granular level, tables store data in structured form, consisting of rows and columns. The format of these tables is defined by the underlying data source, and PrestoDB accesses them through the designated connector. This ensures that even non-relational data sources, such as NoSQL databases, can be queried in a SQL-compatible format.

Through the integration of these core components, PrestoDB enables users to execute federated queries across different storage technologies, unifying disparate data environments under a single query engine. Whether querying structured relational data or handling schema-less NoSQL storage, PrestoDB's flexible architecture provides a scalable and efficient solution for distributed analytics.

4) **Architecture:** PrestoDB operates on a distributed architecture designed for executing SQL queries efficiently across large-scale datasets stored in diverse data sources. As illustrated in Figure 1 [4], the system consists of several core components: client, coordinator, workers, and connectors, each playing a specific role in query execution.

The client submits SQL queries through the Presto CLI, JDBC, or other interfaces. These queries are received by the coordinator, which acts as the master node, responsible for parsing, planning, and orchestrating execution. Within the coordinator, the parser, planner, and scheduler work together to break queries into tasks and assign them to workers efficiently.

To interact with various data sources, PrestoDB employs connectors that serve as storage plugins for systems like PostgreSQL, MongoDB, and Cassandra. These connectors provide two main functionalities: the Metadata API, which helps the coordinator retrieve schema and table structures, and the Data API, which enables workers to fetch the actual records required for query execution.

The worker nodes execute the assigned tasks by retrieving data from the connectors, processing the necessary computations, and exchanging intermediate results. Running in parallel, they significantly improve query performance and scalability. Once processing is complete, the coordinator gathers the results and returns them to the client.

This distributed architecture ensures efficient query execution, scalability, and seamless access to heterogeneous data sources, making PrestoDB an optimal choice for large-scale analytics and federated querying.

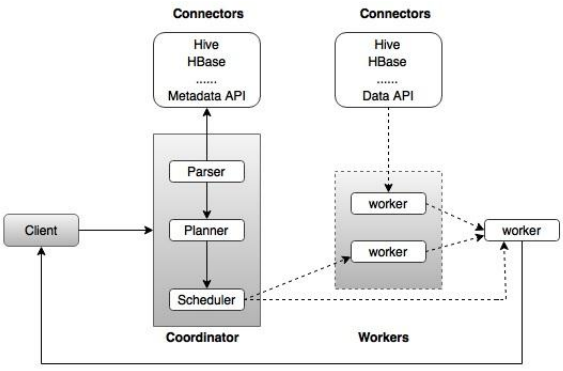

Figure 1 Presto - Architecture

C. PostgreSQL

PostgreSQL is a highly acclaimed open-source relational database management system (RDBMS). It is praised for its rich set of features, strong SQL compliance, and flexibility, making it a top choice for a wide variety of applications. Whether you're working on a small project or building a large-scale enterprise system, Postgres is capable of handling diverse workloads efficiently. It operates seamlessly across various platforms such as Linux, macOS, Windows, and more, making it adaptable to a variety of environments.

Here are some standout features that make PostgreSQL an excellent choice for developers:

- **Extensibility and Customization:** One of PostgreSQL's most powerful features is its ability to be extended. Users can define their own custom data types, operators, functions, and even create new index types. This flexibility allows developers to design solutions that precisely meet their application's requirements, making it highly adaptable to different use cases.
- **ACID Transactions:** PostgreSQL guarantees data integrity and consistency through its support for ACID (Atomicity, Consistency, Isolation, Durability) properties. These principles ensure that all database transactions are processed reliably, which is crucial for applications where data accuracy is essential, such as financial systems or e-commerce platforms.
- **Robust Data Integrity:** The system provides comprehensive support for enforcing data integrity with constraints like foreign keys, unique keys, and check constraints. These features prevent invalid data from entering the database and ensure relationships between tables are maintained, resulting in more reliable and consistent data management.
- **High Concurrency with MVCC:** PostgreSQL's Multi-Version Concurrency Control (MVCC) allows for multiple transactions to run simultaneously without blocking each other. This enhances the system's ability to handle high loads and large numbers of concurrent users, making it ideal for real-time applications and high-traffic websites.
- **Advanced Query Optimization:** PostgreSQL is known for its sophisticated query planner and optimizer. It analyzes queries and determines the most efficient execution plan, allowing for faster data retrieval. Features like join optimization, parallel query execution, and subquery optimization make it well-suited for complex queries and large datasets.

In summary, PostgreSQL offers a combination of flexibility, reliability, and performance that makes it an exceptional choice for developers building anything from small-scale applications to large, high-traffic systems. Its extensibility, robust data integrity features, and powerful query optimization ensure that it remains a top contender for any database project.

D. MongoDB

MongoDB is a popular, open-source NoSQL database that focuses on flexibility, scalability, and high performance. It is designed to handle large volumes of unstructured and semi-structured data, making it ideal for modern applications that require dynamic and evolving data models. MongoDB operates seamlessly on various platforms, including Linux, macOS, and Windows, and is particularly favored for its ease of use and developer-friendly features.

Here are the key features that make MongoDB a powerful and highly regarded database solution:

- **Document-Oriented Storage:** MongoDB stores data in a flexible, JSON-like format called BSON (Binary JSON). This document-based approach allows for nested data structures, making it well-suited for applications that require dynamic or hierarchical data models. Each document can have a different structure, providing flexibility and enabling easy adaptation to changing data requirements.
- **Scalability:** MongoDB is designed to scale horizontally across many machines, enabling it to handle large volumes of data and high traffic loads. The system automatically distributes data across multiple nodes using a technique called sharding, which allows for seamless scaling as the database grows, making it ideal for big data and cloud applications.
- **High Availability and Replication:** MongoDB provides built-in replication through replica sets, ensuring data is automatically duplicated across multiple nodes. If one node goes down, another can take over, guaranteeing that the database remains highly available and that there is no downtime. This feature is essential for applications that require uninterrupted access to data.
- **Tunable Consistency and Read/Write Operations:** MongoDB offers tunable consistency levels, allowing developers to adjust the trade-off between consistency, availability, and partition tolerance (as per the CAP theorem). For instance, you can configure read preferences and write concerns to balance performance and consistency based on your application's needs.
- **Rich Query Capabilities:** MongoDB provides a powerful query language that allows developers to perform complex queries on the document data. It supports a wide range of query operations, including filtering, sorting, aggregation, and text search. MongoDB's aggregation framework is particularly powerful, enabling complex data transformations and calculations.
- **Schema Flexibility:** MongoDB does not require a predefined schema, which means that data can be stored in a flexible, unstructured way. This allows developers to quickly iterate on application features without worrying about rigid database schemas. As the application evolves, new fields can be added to documents without affecting existing data.
- **Indexing Support:** MongoDB supports a wide variety of indexing options to optimize query performance, including single-field, compound, geospatial, and full-text indexes. These indexes significantly speed up data retrieval, making MongoDB an excellent choice for performance-sensitive applications, such as real-time analytics or search engines.

In summary, MongoDB offers a robust combination of flexibility, scalability, and performance, making it well-suited for modern applications. Its document-based model, ability to scale horizontally, and advanced querying capabilities enable the efficient management of large and complex data sets.

E. Cassandra

Apache Cassandra is a highly scalable, distributed NoSQL database designed to handle large amounts of data across many commodity servers without a single point of failure. It is built to ensure high availability, fault tolerance, and fast performance for applications requiring massive scale. Cassandra is

particularly well-suited for real-time applications with big data needs and works across a variety of platforms such as Linux, macOS, and Windows.

Here are some of the key features that make Cassandra a powerful and unique database choice:

- **Scalability and Distribution:** One of Cassandra's standout features is its ability to scale horizontally across multiple nodes, allowing it to handle petabytes of data. As demand grows, new nodes can be added to the cluster without downtime, making it ideal for applications with fluctuating or rapidly growing data requirements.
- **High Availability and Fault Tolerance:** Cassandra is designed for continuous availability. Data is replicated across multiple nodes, ensuring that even if a node or an entire data center goes down, the system remains operational. This feature is crucial for mission-critical applications that cannot afford downtime, such as online banking or e-commerce platforms.
- **No Single Point of Failure:** Unlike traditional relational databases that rely on a single server, Cassandra uses a peer-to-peer architecture where all nodes are equal. Each node can handle read and write requests independently, and there is no central master node. This distributed nature eliminates the risk of a single point of failure and enhances the overall reliability of the system.
- **Flexible Schema Design:** Unlike traditional relational databases, Cassandra uses a flexible schema, which allows users to modify the data structure as the application evolves. It is optimized for write-heavy workloads and can store large volumes of semi-structured and unstructured data, making it well-suited for big data applications and real-time analytics.
- **Tunable Consistency:** Cassandra offers tunable consistency levels, allowing users to decide the level of consistency required for each query. You can configure it for strong consistency (ensuring all replicas agree on data) or eventual consistency (allowing for faster writes and higher availability). This flexibility lets developers optimize the system for specific use cases, whether prioritizing speed or data accuracy.
- **Efficient Write Operations:** Cassandra is optimized for high write throughput, using a write-optimized storage engine that can handle large amounts of data being ingested at high speeds. This makes it ideal for use cases where the volume of incoming data is enormous, such as logging, sensor data, and real-time metrics collection.
- **Distributed Data Architecture:** Data in Cassandra is automatically partitioned across nodes based on a partition key, and each node can store data locally while replicating it to other nodes for redundancy. This architecture ensures that the database can efficiently handle massive datasets distributed across multiple geographic locations.

In conclusion, Cassandra's distributed architecture, fault tolerance, and horizontal scalability make it an excellent choice for high-volume, real-time applications that require massive scalability and availability. Its flexibility, tunable consistency, and high write throughput provide developers with the tools needed to build efficient, reliable systems capable of handling big data and global applications.

F. TPC-DS Benchmark

According to the TPC-DS specification, "TPC-DS is a decision support benchmark that simulates various aspects of a decision support system, including data maintenance and query execution. It provides a representative performance evaluation for general-purpose decision support systems."
In essence, TPC-DS is:

- An industry-standard benchmark for OLAP and Data Warehousing.
- Implemented across various analytical processing systems, including RDBMS, Apache Spark, Apache Flink, and more.
- It offers a diverse set of SQL queries for evaluation.
- It includes tools to generate input data in various sizes.

For our project, TPC-DS is used for generating both data and queries.

IV. INSTALLATION & CONFIGURATION

In this section we will describe in detail the steps followed to setup the presto cluster (coordinator and workers) and docker environment including docker-swarm, overlay-network and docker containers used for presto nodes and data sources. This analysis contains the different configurations used in order to provide the knowledge for replication of the system we experimented with.

A. Host Machines

For our project we used 3 virtual machines (1 Leader and 2 Nodes docker-wise) that were provided by okeanos-knossos an IaaS cloud service powered by GRNET for the Greek Research and Academic Community [5]. The specifications of each machine are given below:

- **CPU:** 4 Cores - Intel® Xeon® CPU E5-2650 v3
- **RAM:** 8GB
- **Disk:** 30GB
- **Operating System:** Ubuntu Server 22.04 LTS

B. Prerequisites

- Docker 27.4.1 installed on all hosts [6]. A step by step guide for a vanilla docker installation can be found on our repository's README.md file.

C. Networking

Splitting the analysis into two layers will give a valuable insight into the workings of our network configuration.

1) ***1st Layer - Host machines:*** For network connections, two public IPv4 addresses were used, one of which was attached to the machine 1 that was used for the docker containers of PrestoDB coordinator and MongoDB data source and the other to the machine 2 which was used for the PostgreSQL database container and the PrestoDB worker container. The machines 1 and 2 were accessed via ssh using their public IPv4 address.The machine 3 was accessed via ssh using his IPv6 local address through the machine 1. It is obvious, that this type of network communication is perplexing and not very scalable. This is why

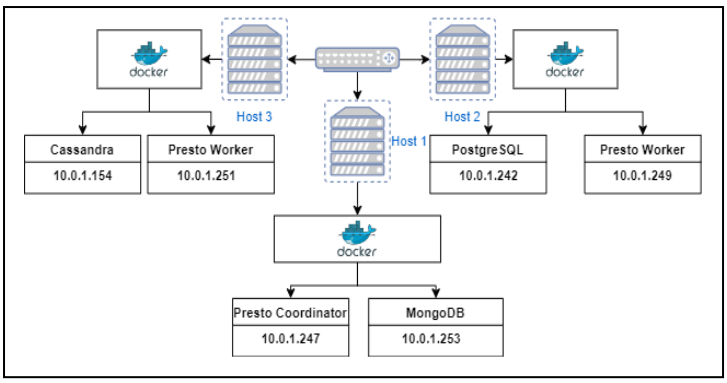

Figure 2 Physical network topology of our system (IPv4 addresses belong to the VLAN of docker)

in the next section we will introduce a network abstraction with the help of docker overlay network.

2) ***2nd Layer – Docker Networking:*** After verifying that the same version of docker engines (27.4.1) were installed among the hosts, we spawned a new docker swarm on machine 1, electing it a leader and using as its advertising address one of its IPv6 addresses (2001:648:2ffe:501:cc00:13ff:fea7:ddf9). We proceeded to adding the machines 2 and 3 to the swarm as worker-nodes. Furthermore, we created a new docker network using the overlay network driver which provides a distributed virtual network to fulfill the purpose of our distributed query engine. The overlay network "overnet", as we named it, uses

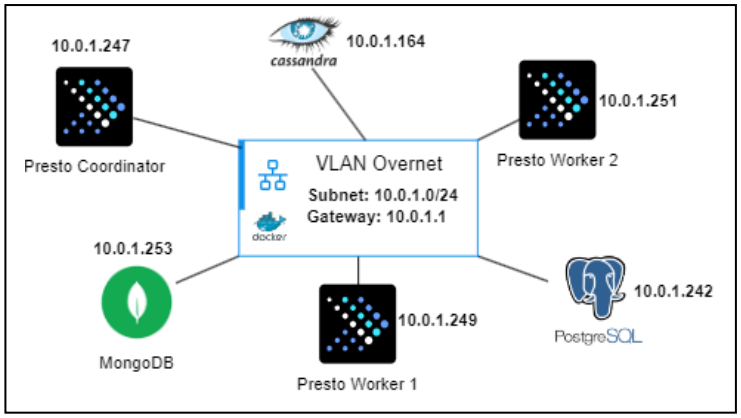

Figure 3 VLAN docker overlay network

VLAN networking and gives a new set of IPv4 addresses to all of our containers that will be running on different hosts succeeding in simplifying our network across different hosts upon deployment of our services. Consequently, we achieved a satisfying level of abstraction transforming the network on Figure 2 to the virtual network on Figure 3 where all running services share the same subnet (10.0.1.0/24).

D. Presto Cluster Set Up & Configuration

For our Presto cluster set up we followed the steps on the official Presto documentation for deploying presto from a docker image [7] but we modified the `config.properties` and `jvm.config` files and also used a `node.properties` for the coordinator as well as for the Presto workers that will be deployed from the same image but with different configuration files.

1) Presto Coordinator

- **Configuration properties:** The config.properties file, contains the configuration for the Presto server. It specifies that the server plays a dual role: coordinating query planning and task distribution while also executing tasks as part of the processing workload. The file also outlines how the nodes within the system discover and communicate with each other, ensuring smooth collaboration across the network. Additionally, it sets general guidelines for resource management, controlling how much memory is allocated for query processing and providing mechanisms to handle large datasets by temporarily offloading data to disk when necessary (Spill to disk technique).
- **JVM configuration:** This configuration file defines a set of Java Virtual Machine (JVM) options that shape the overall runtime environment for a server application. It establishes how much memory the application should use, along with tuning parameters that optimize memory allocation and garbage collection processes. The settings ensure that the server operates within a defined memory footprint, handles resource-intensive tasks efficiently and responds gracefully to potential memory issues by triggering diagnostic measures like heap dumps when errors occur. Moreover, the file enables advanced performance optimizations and diagnostic features, creating a robust and fine-tuned environment that balances resource usage, stability, and performance for demanding production workloads.
- **Node properties:** The node.properties file, contains configuration specific to each node such as the name of the environment and the unique identifier.

All three files are part of the docker-presto-integration/coordinator directory and will be mounted into the `/opt/presto-server/etc/` upon deployment.

2) Presto Workers

- **Configuration properties:** It specifies that the node will not function as a coordinator, meaning it will focus solely on executing tasks rather than managing query planning. The file sets a designated port for network communications and provides a discovery URI that connects the worker node to the cluster's central coordination service, ensuring seamless integration and communication across the system. Additionally, it enables the mechanism for disk spilling, allowing the node to offload intermediate data to a specified directory when handling large datasets, thereby efficiently managing memory usage. This file is exactly the same for both workers.
- **Node properties:** Node specific info (same as coordinator)
- **JVM configuration:** Exactly the same as the coordinator's jvm.config file.

The `config.properties` and `node.properties` files will be mounted from `docker-presto-integration/workers /<worker_number>` into the `/opt/presto-server/etc/` directory of the running worker containers. The `jvm.config` will be mounted into the same directory but will be found on parent directory of the previous files.

E. Connect Databases with Presto cluster

To connect the databases to our Presto cluster, we must first download the corresponding Docker images with the appropriate configuration settings. This process occurs automatically upon deployment of our stack, so further explanation is unnecessary. To enable PrestoDB to access the databases, it is required to mount the respective connectors for each database into the `/opt/presto-server/etc/catalog` directory of the running Presto containers (both coordinator and

workers).The directory of our repository: `./docker-presto-integration/config/catalog`, contains four files that specify the ports and credentials for our databases:

- cassandra.properties
- mongodb.properties
- postgresql.properties
- tpcds.properties: explained on section H. TPC-DS Data Loading

Once the database containers are deployed, the Presto coordinator will be able to access each database using its designated port and host.

F. Composing the Docker stack

To deploy different containers as services on different hosts using overlay network driver, we will compose a proper docker-compose.yaml file and then execute the command:

```
docker stack deploy -c docker-compose.yaml presto
```

Code Snippet 1

This docker-compose.yaml file orchestrates the deployment of a distributed Presto cluster alongside three database services (MongoDB, PostgreSQL, and Cassandra) within a Docker Swarm environment. It specifies the configuration of a Presto coordinator and two Presto workers, each with dedicated configuration files, environment variables, port mappings, and volume mounts to ensure proper communication and data persistence as described above. Finally, the file deploys the database services with their respective images, environmental settings and volumes for persistent storage, while assigning explicit placement constraints to ensure that each container runs on its predetermined host. This file serves as the definitive blueprint outlining the architecture of our entire system.

G. Benchmarking

For the benchmarking the benchmark driver provided by Presto was used [8]. Following the steps in the documentation of presto version 0.290 we downloaded the jar file in the directory docker-presto-integration/coordinator and mounted it into the /opt/benchmark-driver/presto-benchmark-driver of our presto coordinator container running on the main node (host 1) of our swarm. In this way, we automated the greater part of the benchmarking process.

H. TPC-DS Data Loading

To facilitate benchmarking, we employed an Extract, Transform, Load (ETL) process to populate PostgreSQL, Cassandra, and MongoDB with TPC-DS data. This approach ensures consistency across database systems and enables comparative performance analysis. We utilized Presto to create and populate the tables by executing `CREATE TABLE AS SELECT` (CTAS) queries. This allowed us to directly load TPC-DS data from the source dataset into each database. The data was loaded for scale factors (SF) 1 and 10 using the following command:

```
docker exec -it \
presto presto-cli --server presto:8080 \
--catalog <db_name> --schema <sfx> \
--file /opt/presto-server/etc/tpcds_to_<dbname>.sql
```

Code Snippet 2

This command executes a SQL script containing CREATE TABLE AS SELECT queries for each database. Since different databases have varying data type constraints, we applied transformations during the loading process. For PostgreSQL, the schema was preserved without modifications, allowing direct ingestion. For Cassandra and MongoDB, data types that were not natively supported were transformed to ensure compatibility. Specifically, string-based attributes were cast to VARCHAR, and decimal values were cast to DOUBLE before ingestion. This transformation standardizes the dataset, enabling both databases to store and process the data efficiently. By leveraging Presto's CREATE TABLE AS SELECT functionality, we streamlined the ETL process for loading TPC-DS data into PostgreSQL, Cassandra, and MongoDB. The transformation phase ensured compatibility across databases, enabling meaningful performance comparisons. The `tpcds.properties` that was mentioned in the *Connect Databases with Presto cluster* section was used to achieve the loading from data through Presto.

V. Process and Methodology

A. The TCP-DS Benchmark

The TPC-DS benchmark is a versatile framework for assessing decision support systems, providing a diverse set of SQL queries and the ability to generate datasets of different size.

1) ***Business Model:*** TPC-DS models adopts the business model of a large retail enterprise, making it adaptable to any industry involved in the management, sales, and distribution of products. The model features not only a network of nationwide stores but also incorporates sales channels such as catalogs and online platforms. In addition to tables for tracking sales and returns, the model includes a basic inventory management system and a promotional system to simulate business operations comprehensively.

- Record customer purchases (and track customer returns) from any sales channel
- Modify prices according to promotions
- Maintain warehouse inventory
- Create dynamic web pages
- Maintain customer profiles (Customer Relationship Management)

2) ***Logical Database Design:*** The TPC-DS shema is a snowflake schema that consists of multiple dimension and fact tables. As mentioned before, it is designed to represent the sales and returns operations of a company that uses three primary sales channels: physical stores, catalogs, and online platforms. At the core of this schema are seven fact tables—inventory, store sales, store returns, catalog sales, catalog returns, web sales, and web returns — which store detailed transactional data. Each of these tables is represented through an ER diagram, Ih depicts the connections between the fact tables and the related dimension tables. Specifically, the schema includes:

- Paired fact tables that track product sales and returns across the three sales channels: physical store, catalogs, and online sales.
- A single fact table dedicated to inventory management, specifically for catalog and online sales

Figures 4 through 10 represent the ER diagrams of each of the seven fact tables, as provided by the TCP-DS Standard Specification.

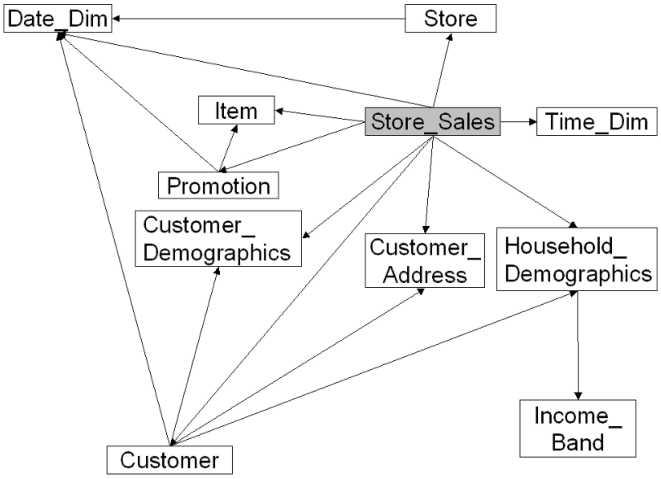

Figure 4 Store Sales ER Diagram

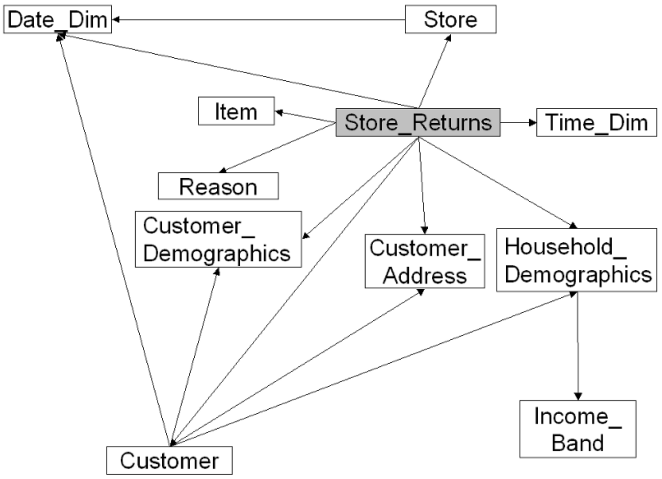

Figure 5 Store Returns ER Diagram.

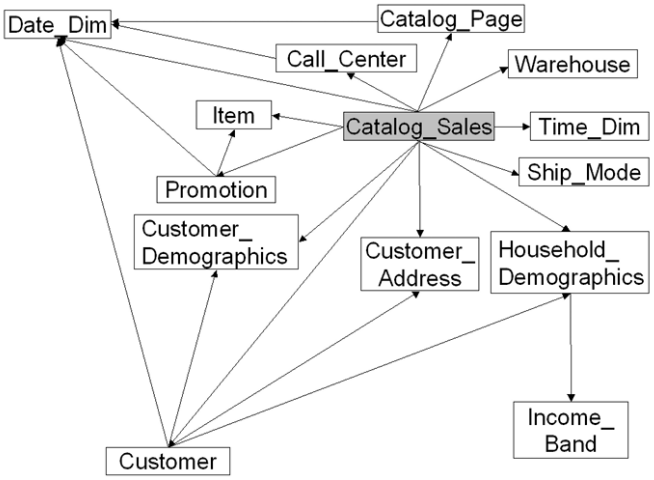

Figure 6 Catalog Sales ER Diagram.

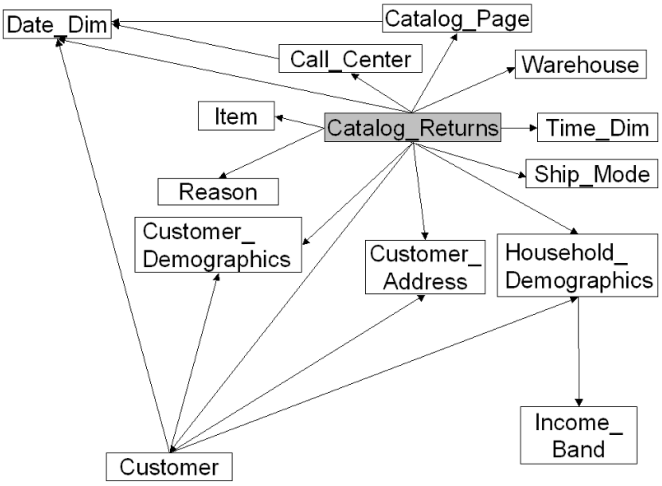

Figure 7 Catalog Returns ER Diagram.

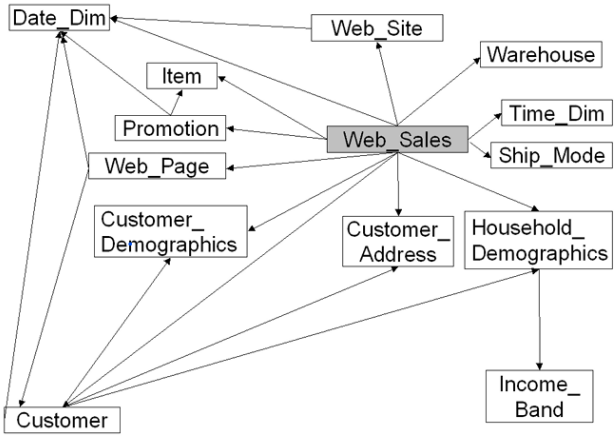

Figure 8 Web Sales ER Diagram.

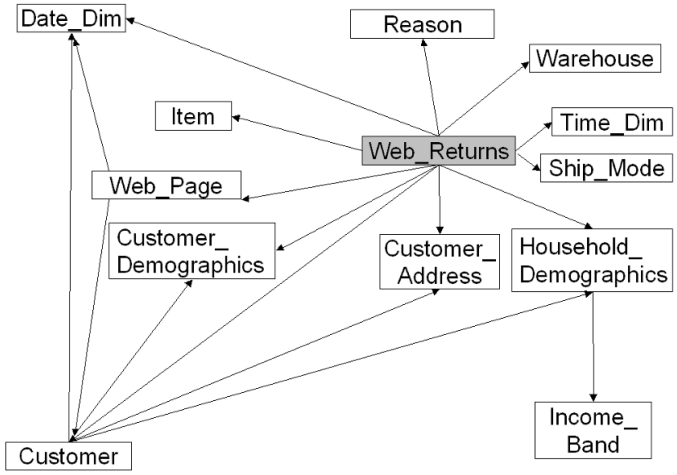

Figure 9 Web Returns ER Diagram.

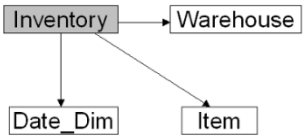

Figure 10 Inventory ER Diagram.

3) ***Query Templates:*** The TPC-DS includes a suite of 99 queries, which we utilized to evaluate the performance of our various data distribution strategies. As outlined in the TPC-DS specification, the queries are classified into the following categories:

- **Reporting Queries:** Periodic queries answering predefined business questions with minor variations like date ranges or locations.
- **Ad hoc Queries**: Spontaneous queries created to address immediate, specific business needs.
- **Iterative OLAP Queries:** Sequential queries for exploring data to uncover trends and relationships.
- **Data Mining Queries:** Queries analyzing large datasets to identify patterns and predict trends.

B. Data Partitioning

Data partitioning in Presto helps optimize query performance by dividing large datasets into smaller segments. Below, we outline the approach we used to implement it.

1) ***TPC-DS Queries Subset Selection:*** Initially, we conducted an in-depth review of the full set of 99 queries from TPC-DS to select a subset of queries for our analysis and evaluation. This selection was based on an effort to categorize the queries into three groups, according to specific factors that determine their complexity. Specifically, we examined the number of tables involved in each query, focusing on whether they rely on a single fact table or multiple fact tables. Additionally, we considered the size of the tables used, as well as the complexity of the queries themselves, as determined by the use of aggregation functions, complex operations, or special operators. The table I shows the categorization of the selected TPC-DS queries into their respective groups. This categorization served as the foundation for evaluating the impact of data partitioning techniques in subsequent stages of our analysis.

TABLE 1 CATEGORIZATION OF SELECTED TPC-DS QUERIES

Groups	Queries
Group 1: Single Fact Table	Q1, Q15, Q30, Q51, Q65
Group 2: Multiple Fact Table Medium	Q21, Q35, Q76, Q94
Group 3: Multiple Fact Table Difficult	Q5, Q39, Q49, Q77, Q78

2) ***No Data Distribution Applied:*** As a starting point, we measured the performance without any data distribution to establish a baseline system performance value. For each benchmark and distribution strategy, we performed two initial warm-up runs followed by three measurement runs to ensure the accuracy and reliability of the results. The two warm-up runs were conducted to mitigate any caching effects, ensuring that subsequent measurements reflected the true performance without being influenced by cached data. The full comparison results between MongoDB, Cassandra, and PostgreSQL are presented in Section VI-A.

3) ***Table Distribution:*** The initial performance measurements, conducted without data distribution (Section VI-A), revealed a substantial performance disparity among PostgreSQL, MongoDB, and Cassandra. PostgreSQL consistently outperformed the other databases across the majority of queries, followed by MongoDB, while Cassandra demonstrated the longest execution times. To inform the partitioning of tables across the databases, we utilized benchmark results provided by the Presto Benchmark suite. Specifically, we measured both the average wall-clock time (wallTimeMean) and the total CPU time consumed by all Presto processes (processCpuTimeMean) for each query. By analyzing the ratio of wallTimeMean to processCpuTimeMean, we assessed the relative efficiency of each database under varying workloads. As illustrated in Figure 11, PostgreSQL and MongoDB exhibited significant parallelism, as evidenced by their lower wallTimeMean relative to processCpuTimeMean. However, for queries Q30 and Q77, Cassandra achieved notably better resource utilization compared to the other databases. Leveraging the unique strengths and performance characteristics of each database, we partitioned the tables according to their query-specific execution profiles, absolute execution times, and the wallTimeMean to processCpuTimeMean ratio. Table II summarizes the resulting partitioning strategy. PostgreSQL's superior ability to support complex operations led to its assignment as the primary repository for tables involved in computationally intensive queries. Key dimension table date_dim was allocated to PostgreSQL. Dimension tables are frequently involved in join operations across multiple queries. Centralizing them in PostgreSQL minimizes the data volume transferred to Presto, allowing PostgreSQL to handle the majority of the join workload locally and efficiently. Furthermore, tables associated with complex queries and substantial data volumes were primarily allocated to PostgreSQL and MongoDB due to their superior performance. Conversely, Cassandra was assigned smaller and medium-sized tables, which are located in the Web_Return fact table, and demonstrated better efficiency in specific queries (Q30 and Q77) based on its wallTimeMean to processCpuTimeMean ratio. This partitioning strategy effectively exploits the complementary capabilities of PostgreSQL, MongoDB, and Cassandra, optimizing query performance and resource utilization across the heterogeneous database environment.

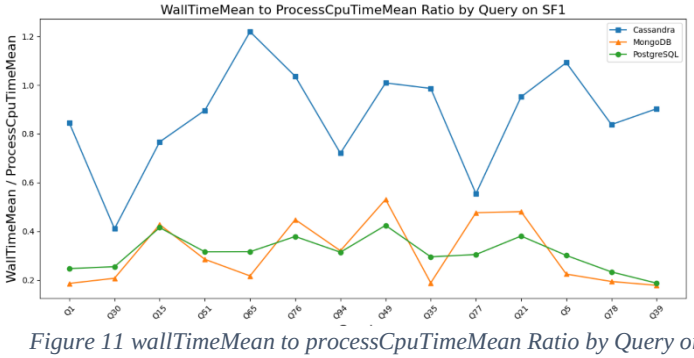

Figure 11 wallTimeMean to processCpuTimeMean Ratio by Query on SF1

TABLE 2 DATA DISTRIBUTION BASED

Table	PostgreSQL	MongoDB	Cassandra
catalog_sales		✓	
customer			✓
customer_address			✓
customer_demographics	✓		

date_dim	✓		
inventory	✓		
item		✓	
store		✓	
store_returns		✓	
store_sales	✓		
warehouse		✓	
web_page			✓
web_returns			✓
web_sales	✓		
web_site			✓
catalog_returns	✓		
catalog_page		✓	

C. *Presto Query Optimization*

Query optimization is crucial to Presto's ability to deliver fast and efficient analytics on large datasets. By leveraging both cost-based and history-based optimization, Presto ensures that complex queries are executed effectively, minimizing resource usage and execution time.

1) ***Cost based optimizations:*** [9] Presto leverages table statistics, such as row counts and data distribution, to estimate query execution costs and optimize join order and distribution methods. This reduces resource usage and execution time by reordering joins and selecting between partitioned or broadcast joins based on estimated costs. However, a key limitation is that CBO relies on table statistics, which are only available through the Hive Connector. [10] This means that they cannot be fully utilized with other connectors.

2) ***History Based optimizations (HBO):*** [11] HBO enhances query execution by leveraging historical statistics from past queries rather than relying on table statistics. It optimizes join distribution by selecting between broadcast and partitioned joins based on previous execution data. However, HBO requires a storage provider, such as Redis, to retain historical statistics. Since this setup is not utilized in our analysis, HBO is not considered in our evaluation.

3) ***Push Partial Aggregation Through Exchange:*** Since neither CBO nor HBO is enabled in our environment, we rely on PushPartialAggregationThroughExchange, a rule-based optimization that is enabled by default in Presto. This approach minimizes the volume of data that needs to be shuffled and processed, leading to faster execution times. In distributed environments—where reducing network traffic and processing overhead is critical—this optimization proves highly effective, especially when handling large datasets. [12]

Presto automatically determines whether to push partial aggregation through an Exchange based on the available statistics (defaulting to final aggregation statistics when historical data is not tracked). Given that our queries heavily utilize GROUP BY operations, and that PostgreSQL supports more complex operations (such as joins) more effectively than MongoDB or Cassandra, we confirmed our choice to store key dimension tables in PostgreSQL.

VI. RESULTS

In this section, we present the results of our experiments, which evaluate Presto's performance under various data distribution scenarios across workers and different cluster configurations. Specifically, we report results for each database individually (PostgreSQL, MongoDB, Cassandra) to assess isolated performance. Subsequently, we analyze the combined setup, where tables are distributed across the databases, followed by an evaluation of parallelization through the addition of workers. Finally, we provide results for an optimized configuration. We first analyze the results for sf1, followed by sf10.

A. *Base System Benchmark – No data Distribution applied*

To evaluate query execution performance, we primarily focus on the wallTimeMean metric, as it provides a comprehensive view of the average query completion time.

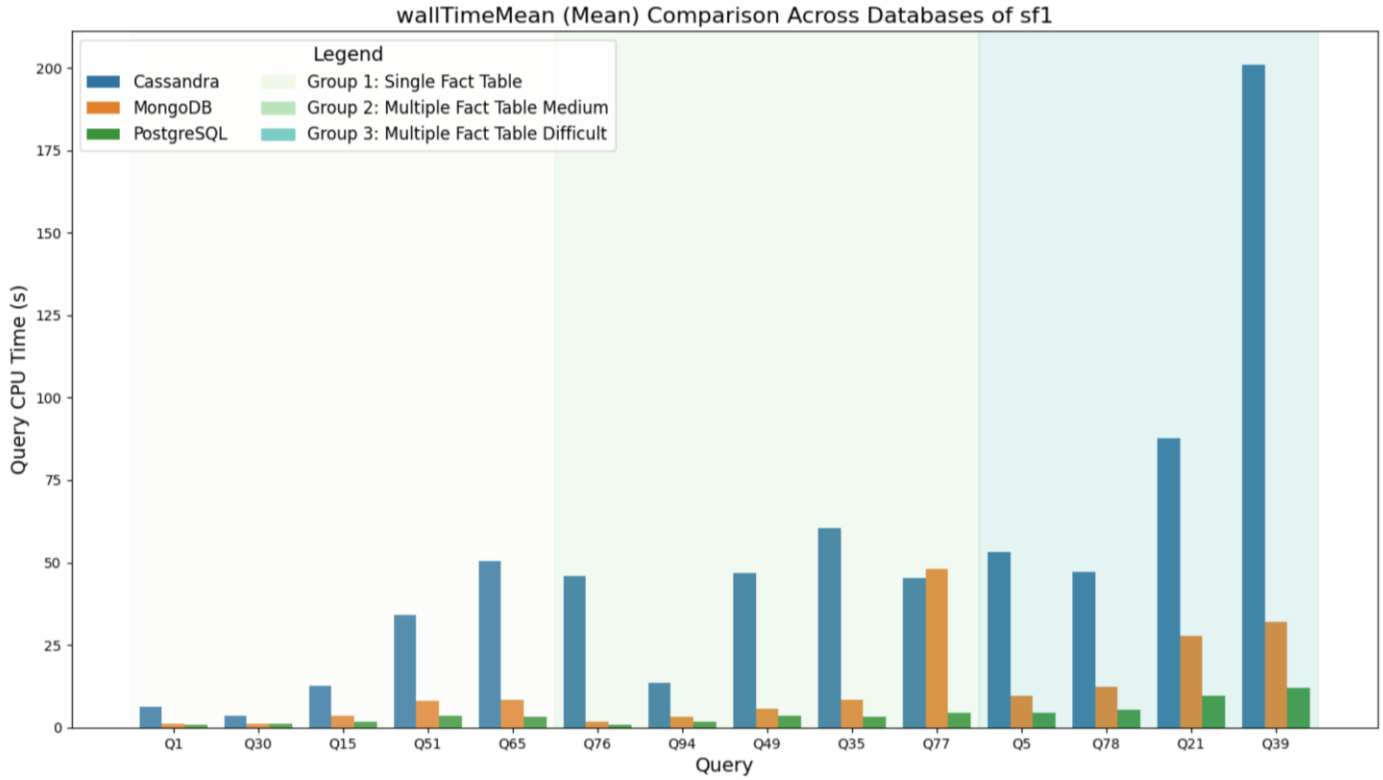

Figure 12 No data Distribution applied for sf1

Figure 12 presents the query execution times across PostgreSQL, MongoDB, and Cassandra without data distribution. PostgreSQL consistently achieves the lowest wallTimeMean, indicating superior performance for most queries, while MongoDB shows moderate execution times. In contrast, Cassandra exhibits significantly higher query times, especially for complex queries, highlighting its inefficiency in this setup. These results emphasize the impact of the underlying database engine on query execution performance when using Presto.

Figure 13 further analyzes execution times by grouping queries based on complexity. PostgreSQL remains the most efficient across all groups, with execution times significantly lower than the other databases. Specifically, PostgreSQL is 5x faster than MongoDB and 10x faster than Cassandra for Group 1 queries. For Group 2 queries, PostgreSQL is 4x faster than MongoDB and 7x faster than Cassandra. In Group 3 queries, the efficiency of PostgreSQL becomes even more pronounced, as it is 3x faster than MongoDB and 12x faster than Cassandra. MongoDB maintains moderate performance, handling complex queries with consistency, though its performance drops slightly for Group 3 queries. Cassandra, however, struggles significantly in Group 3, with execution times far exceeding those of the other databases, indicating severe performance challenges with more demanding workloads.

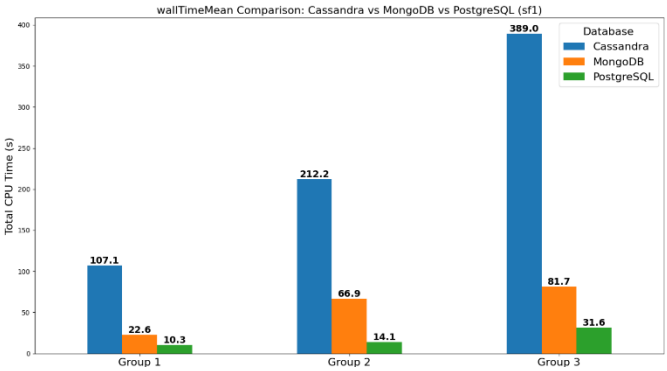

Figure 13 Complexity Group Across Databases for sf1

Building on the SF1 results, we now examine the SF10 dataset to assess scalability and performance under a larger workload.

Figure 14 shows the query execution times for the SF10 dataset, where we observe a notable increase across all databases compared to the SF1 results, indicating the impact of scaling up the dataset. PostgreSQL continues to perform efficiently, although execution times are significantly higher than in SF1, particularly for more complex queries. MongoDB also experiences a considerable rise in query times, especially for larger and more intricate queries, further demonstrating its limitations as the dataset size grows.

Cassandra, however, faces the most significant challenges when scaling from SF1 to SF10. While it shows some improvement in less complex queries, the execution times for more complex queries soar dramatically. This highlights Cassandra's difficulty in efficiently handling larger and more complex workloads as the dataset grows, reinforcing the results observed in SF1, where Cassandra also exhibited relatively high query execution times for complex queries.

Overall, the performance trends observed in SF10 align with those of SF1, with PostgreSQL maintaining its position as the most efficient database, MongoDB showing moderate performance, and Cassandra struggling with scalability and handling complex queries.

During the transition from scale factor 1 (SF1) to scale factor 10 (SF10), we observed that certain queries were failing due to insufficient memory availability. To mitigate these issues, we enabled Presto's spill functionality, allowing intermediate query data to be offloaded to disk. This was achieved by setting `` `experimental.spill-enabled` `` to true. These adjustments ensured more reliable query execution by alleviating memory constraints during the system scaling process. However, currently spilling is supported only for aggregations and joins (inner and outer), so this property will not reduce memory usage required for window functions, sorting and other join types [13]. As a result, query 51 which is using windowing functions continues to fail on SF10 setups and is therefore not presented.

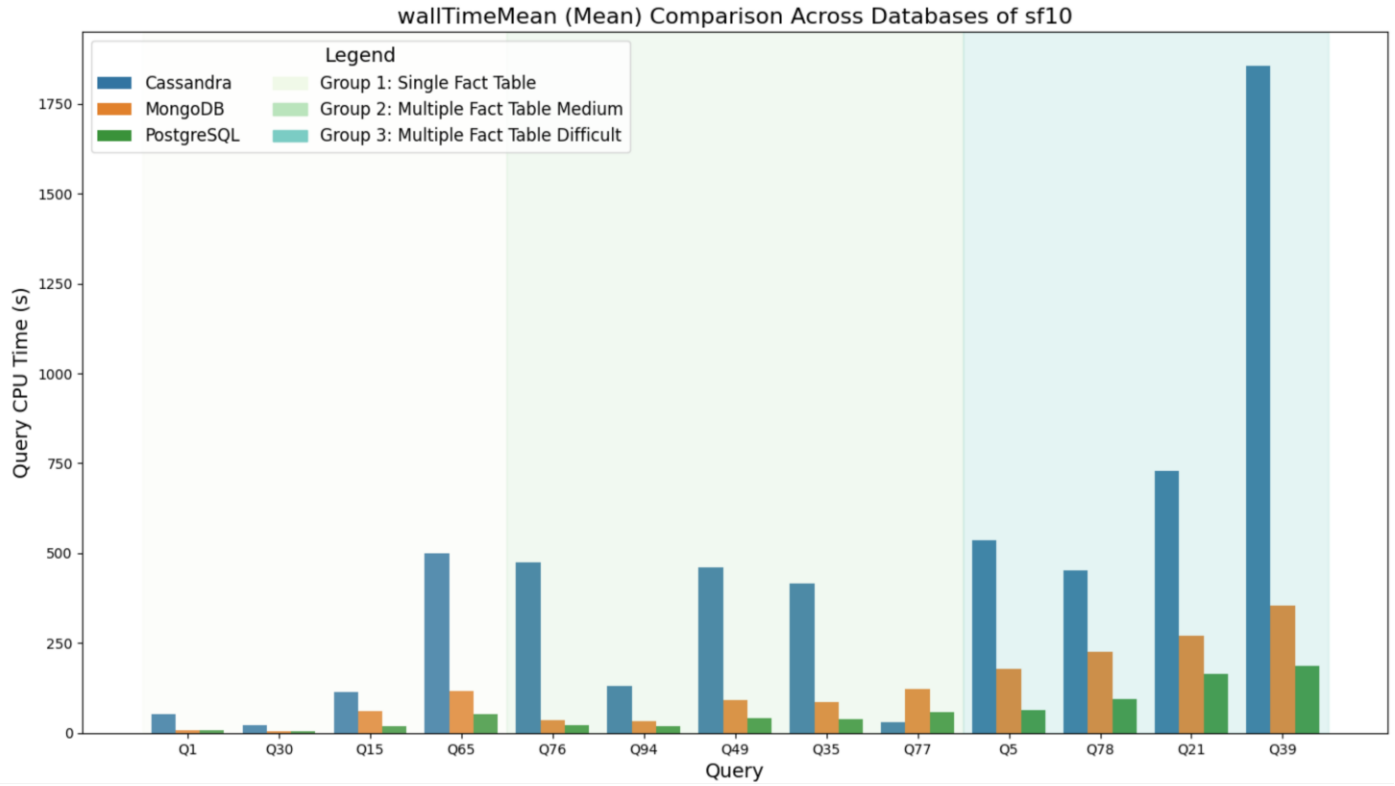

Figure 14 No data Distribution applied for sf10

Figure 15 analyzes execution times for SF10, grouping queries by complexity. Similar to SF1, PostgreSQL consistently performs as the most efficient database across all query groups, though execution times increase with the larger scale of SF10. Specifically, PostgreSQL is 2x faster than MongoDB and 8x faster than Cassandra for Group 1 queries. For Group 2 queries, PostgreSQL is 2x faster than MongoDB and 8x faster than Cassandra. In Group 3 queries, PostgreSQL remains 2x faster than MongoDB and 7x faster than Cassandra. MongoDB exhibits moderate performance, handling Group 1 and Group 2 queries effectively, but its efficiency decreases for Group 3 queries compared to SF1, where the gap widens. Cassandra struggles significantly in all query groups for SF10, with a dramatic increase in execution times, particularly for Group 3 queries, where it becomes the least efficient option. The scaling challenges of Cassandra are evident as its execution times increase disproportionately compared to PostgreSQL and MongoDB.

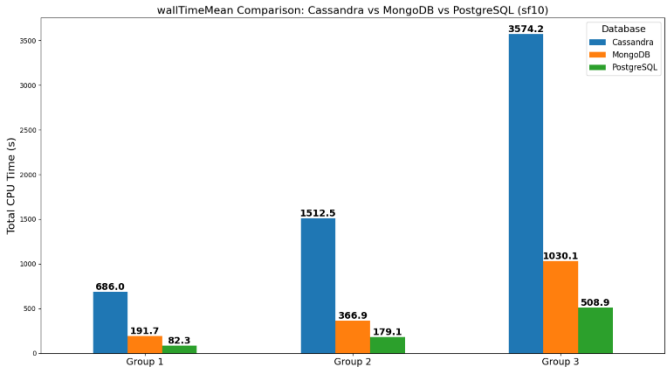

Figure 15 Complexity Group Across Databases for sf10

B. Base System Benchmark – Data Distribution applied

After presenting the results for each database individually, we proceed to analyze the performance of the data distribution strategy. Specifically, we compare the results of applying data distribution with the performance of PostgreSQL, which demonstrated the best performance without data distribution. The comparison includes both the processCpuTimeMean and wallTimeMean metrics to evaluate the impact of data distribution on performance across different configurations.

We begin by comparing the processCpuTimeMean values for both the combined installation and PostgreSQL, starting with dataset SF1 and extending to SF10. This comparison will help evaluate how data distribution affects CPU time as query complexity increases across different groups.

Figure 16 reveals the processCpuTimeMean values for SF1, comparing PostgreSQL to the combined setup across different query groups. The results indicate that for Group 1 and Group 2, CPU time remains relatively similar between PostgreSQL and the combined setup. Although the combined setup initially results in higher CPU time, the difference decreases from Group 1 to Group 2. In Group 1, the combined setup records 31% higher CPU time than PostgreSQL, but in Group 2, this difference drops to 3.5%, indicating that data distribution becomes increasingly beneficial as query complexity grows. However, in Group 3, where queries involve more complex joins and larger data volumes, PostgreSQL alone exhibits a slightly higher CPU time compared to the combined setup. This suggests that distributing workload across databases helps mitigate computational overhead for complex queries, reducing the strain on a single database engine. While the differences in SF1 are not substantial, they suggest that as query complexity increases, partitioning data across multiple databases starts to provide efficiency benefits.

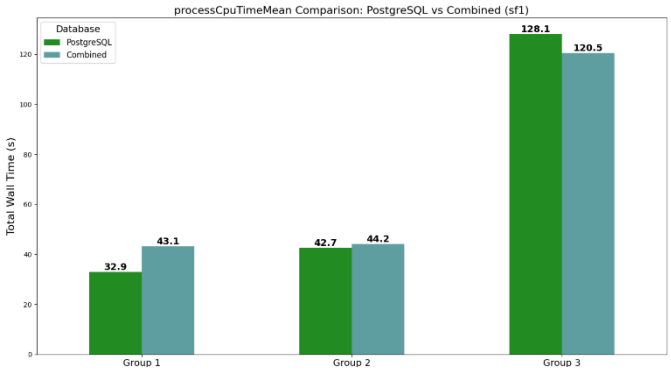

Figure 16 ProcessCpuMean Postgesql vs Combined for sf1

Then SF10 shows even greater performance improvements due to the increased data size, as shown in Figure 17. For Group 1, the combined setup reduces CPU time by 23.5% compared to PostgreSQL, suggesting that distributing tables across multiple databases benefits even simple queries at a larger scale. In Group 2, the gap widens, with the combined setup achieving a 27.7% reduction in CPU time over PostgreSQL, further demonstrating the efficiency of distribution. The most substantial difference appears in Group 3, where workload partitioning results in a 28.8% reduction in CPU time. This highlights the increasing benefits of workload distribution for complex queries as dataset size grows, reinforcing the efficiency of the partitioning strategy.

To further examine the impact of data distribution, we analyze the wallTimeMean metric for both the combined setup and PostgreSQL, beginning with SF1 and then extending to SF10. This analysis provides insight into query execution efficiency and potential performance gains from distributing workload across multiple databases.

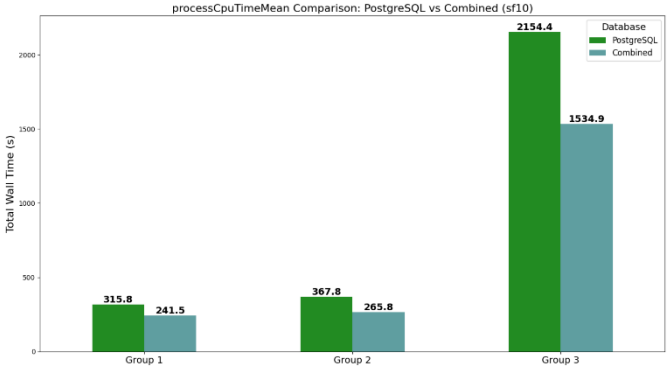

Figure 17 ProcessCpuMean Postgesql vs Combined for sf10

Figure 18 presents the wallTimeMean values for SF1, highlighting differences across query groups. For Group 1, the combined setup results in a 23.4% increase in execution time compared to PostgreSQL, indicating that for simpler queries, the overhead introduced by data distribution outweighs any parallelization benefits. However, in Group 2, where queries involve more complex structures, the trend shifts, with the combined setup achieving a 14.3% reduction in execution time over PostgreSQL. The most notable difference occurs in Group 3, where workload distribution leads to an 18.2% improvement, suggesting that as query complexity increases, workload distribution becomes increasingly beneficial, minimizing execution delays and optimizing overall query processing.

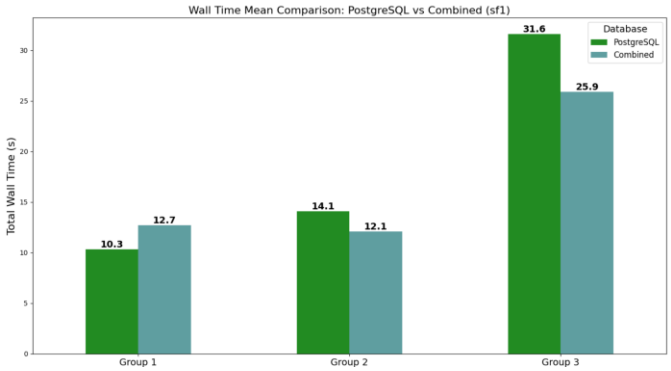

Figure18 wallTimeMean Postgesql vs Combined for sf1

Building on these findings, Figure 19 reveals the impact of dataset scaling on wallTimeMean for SF10. As observed in SF1, the combined setup consistently outperforms PostgreSQL for Group 2 and Group 3, while Group 1 shows a smaller benefit. For Group 1, workload partitioning results in a 23.5% reduction in execution time, suggesting that at larger scales, even simpler queries start to benefit from data partitioning and parallel execution. In Group 2, the performance gap widens, with distribution yielding a 27.7% improvement over PostgreSQL. Finally, in Group 3, the impact of distribution is most pronounced, as execution time is reduced by 28.8%, reinforcing the advantages of a partitioned database architecture, particularly for complex queries at scale and larger table sizes, where execution time bottlenecks become more significant.

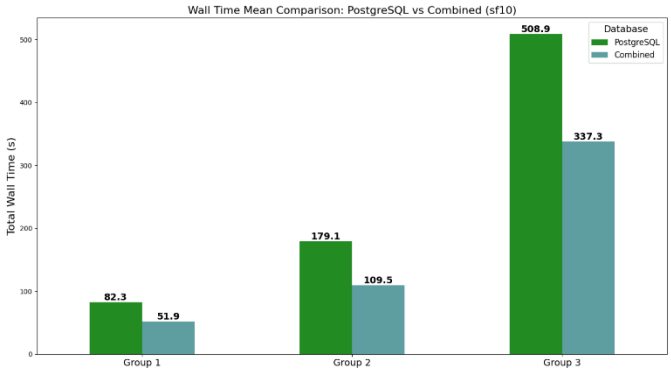

Figure 19 wallTimeMean Postgesql vs Combined for sf10

The observed improvements in wallTimeMean align with the trends identified in the processCpuTimeMean analysis. In both cases, query complexity and dataset size play a crucial role in determining the effectiveness of workload distribution. While SF1 demonstrated only marginal gains, SF10 highlighted significant improvements in both CPU efficiency and execution time, particularly for Group 3 queries. The reduction in both CPU time and query execution time for complex workloads suggests that workload partitioning not only optimizes resource utilization but also mitigates performance bottlenecks in large-scale analytical processing. These findings further reinforce the benefits of multi-database architectures, particularly in high-volume query processing environments.

C. Base System Benchmark – Different Worker Set up

In this study, we analyze the transition from a single-worker setup (W1) to progressively more distributed architectures (W2 and W3), evaluating the impact on query performance. In W1, a single coordinator also functions as a worker, handling all query processing on **Machine 1**. W2 extends this model by introducing an additional worker node on **Machine 2**, enabling partial workload distribution. Finally, W3 further expands the setup with two worker nodes, running on **Machine 3**, further enhancing parallel query execution.

Figure 20 presents the wallTimeMean values for SF1 across different query groups, comparing W1, W2, and W3 configurations.

For Group 1, W2 achieves a 14.7% reduction in execution time compared to W1, showing that the introduction of an additional worker node improves performance. However, transitioning from W2 to W3 results in only a 1.9% increase, indicating that for simpler workloads, the extra worker adds limited benefit and may even introduce slight overhead due to coordination costs.

In Group 2, W2 reduces execution time by 13.4%, reinforcing the advantage of distributing the workload across multiple workers. However, when moving from W2 to W3, the reduction slows down, with a 2.2% increase in execution time.

For Group 3, W2 results in a 12.9% reduction in execution time, showing substantial improvements. However, moving from W2 to W3 only results in a 2.1%.

In summary, the addition of a second worker (W2) leads to significant performance improvements, especially for medium and complex queries. However, the addition of a third worker (W3) results in diminishing returns and may even cause minor overhead due to the cost of parallelism.

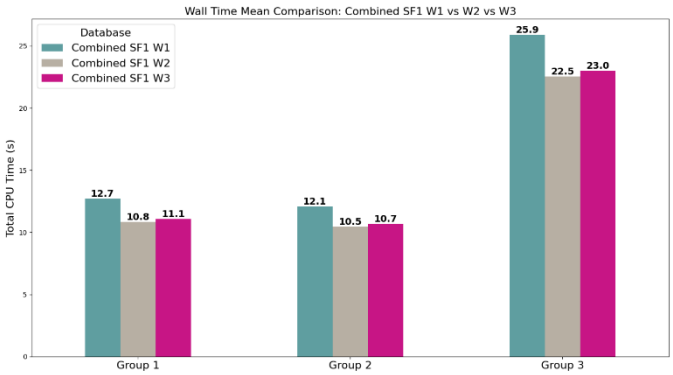

Figure 20 wallTimeMean Comparison: Combined SF1 W1 vs W2 vs W3

Figure 21 presents the wallTimeMean values for SF10 across different query groups, comparing W1, W2, and W3 configurations.

In Group 1, where queries are simpler, W2 results in a 17.5% reduction in execution time compared to the W1 setup, and W3 further reduces execution time by 3.4%. These results suggest that adding worker nodes significantly enhances performance for simpler queries, with each additional worker providing a improvement.

In Group 2, the benefits of scaling become more pronounced. W2 reduces execution time by 15% compared to W1, while W3 leads to an additional 3.3% improvement. As query complexity and table sizes increase, workload distribution continues to deliver strong performance improvements, though the rate of reduction starts to level off after the introduction of the third worker.

For Group 3, which involves the most complex queries, W2 reduces execution time by 14.5%, and W3 further reduces it by 3.8%. Although the performance gains from adding additional workers are evident, the improvements become smaller as the queries increase in complexity, suggesting that the benefits of scaling are somewhat limited for highly complex queries.

In conclusion, as seen in SF10, the addition of a second worker (W2) continues to show performance improvements across all groups, especially for simpler and medium-complexity queries. However, the improvements are smaller for

complex queries, and the rate of performance gains tends to decrease as query complexity increases. This reinforces the notion that while scaling helps, the benefits start to diminish after a certain point, especially for more demanding workloads.

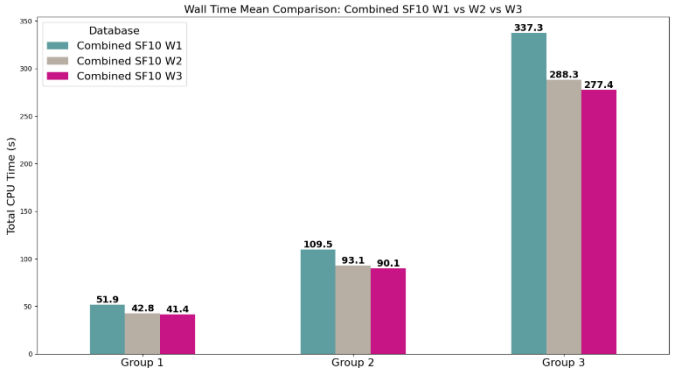

Figure 21 wallTimeMean Comparison: Combined SF10 W1 vs W2 vs W3

It is noteworthy that in SF1, Groups 1 and 2 exhibit similar execution times, whereas in SF10, Group 2 experiences significantly higher execution times compared to Group 1. As regards SF1, Groups 1 and 2 exhibit similar execution times, whereas in SF10, Group 2 experiences higher execution times compared to Group 1. This discrepancy suggests that the scaling of data impacts Group 2 more substantially, since the queries of this group have significantly larger tables and perform complex functionality. The observed difference highlights that while certain queries scale efficiently, others may be more sensitive to data growth, leading to disproportionate increases in execution time.

VII. CONCLUSION

In conclusion, this paper presented a comprehensive evaluation of Presto's performance in distributed query execution, focusing on various data distribution strategies and cluster configurations across PostgreSQL, MongoDB, and Cassandra. Our findings indicate that PostgreSQL consistently outperforms other databases in the base system without data distribution. However, the introduction of data distribution and additional worker nodes led to significant performance improvements, particularly for complex queries and larger datasets. Although the combined setup resulted in slightly higher execution times in certain scenarios, the advantages of parallel processing and workload distribution were evident, especially in managing intricate queries at scale. These results highlight the potential of Presto's distributed architecture to handle diverse data sources efficiently, even those without native SQL support.

For future work, we propose testing larger datasets, such as SF100, to further evaluate the impact of scaling with multiple workers. Additionally, we plan to investigate the use of Redis as a Historical Statistics Provider (HBO) to improve query planning and execution. Lastly, we aim to explore different partition keys in Cassandra to assess their influence on performance in distributed query processing environments.

REFERENCES

[1] R. Sethi et al., "Presto: SQL on Everything," in 2019 IEEE 35th International Conference on Data Engineering (ICDE), Macao, China, 2019

[2] "Engineering data analytics with Presto and Apache Parquet at Uber," 2017. [Online]. Available: https://www.uber.com/blog/presto

[3] TPC Benchmark™ DS- Standard Specification, Version 3.2.0, June 2021.

[4] "Apache Presto – Architecture" [Online]. Available: https://www.tutorialspoint.com/apache_presto/apache_presto_architecture.htm

[5] "okeanos-knossos" [Online]. Available: Link

[6] Brian Hogan, Tony Tran, "How to install and use Docker on Ubuntu 22.04", 22 April 2022 [Online] Available: Link

[7] Deploy Presto From a Docker Image [Online]. Available: https://prestodb.io/docs/current/installation/deploy-docker.html

[8] "Benchmark Driver" [Online]. Available:: https://prestodb.io/docs/current/installation/benchmark-driver.html

[9] "Cost based optimizations"(CBO) [Online]. Available: https://prestodb.io/docs/current/optimizer/cost-based-optimizations.html

[10] "Table Statistics": https://prestodb.io/docs/current/optimizer/statistics.html

[11] "History based optimizations" (HBO) [Online]. Available: https://prestodb.io/docs/current/optimizer/history-based-optimization.html

[12] "Presto-Planner" [Online]. Available: https://prestodb.io/blog/2019/12/23/improve-presto-planner

[13] "Spill to Disk" [Online]. Available: https://prestodb.io/docs/current/admin/properties.html#tuning-spilling