Distributed Execution of SQL Queries over Presto

Antonis Zakynthinos Georgia Manifava Ioannis Mytis

Electrical and Computer Engineering
National and Technical University of Athens
Athens, Greece
{el17441, el18001, el13133}@mail.ntua.gr

***Abstract*—This paper benchmarks the performance of the distributed SQL query engine PrestoDB when connected to three heterogeneous data sources: MongoDB, Cassandra, and PostgreSQL. The study examines the effectiveness of the query engine across varying data distribution strategies, worker configurations, and query complexities. Using the TPC-DS benchmark for data and query generation, we conduct experiments on a subset of queries to analyze and enhance query latency, bandwidth, and memory utilization. The findings culminate in a proposed optimization strategy that considers the entity-relationship model, query performance, and the unique characteristics of the underlying storage technologies. This research contributes to a deeper understanding of query processing in PrestoDB and provides recommendations for improving data distribution strategies in distributed query systems.**

***Index Terms*—PrestoDB, distributed SQL query engine, MongoDB, Cassandra, PostgreSQL, Docker, heterogeneous data sources, TPC-DS**

I. Introduction

The advent of Big Data has revolutionized the way organizations manage and analyze information, offering unprecedented insights but also introducing significant challenges. The volume, diversity, and complexity of data require innovative architectures, algorithms, and analytics to manage them and extract meaningful information efficiently. Distributed SQL query engines like PrestoDB have emerged to address these challenges by enabling federated queries across diverse data sources, such as relational, wide-column and BSON document databases. Companies like Netflix and LinkedIn have widely adopted PrestoDB for its ability to seamlessly process large-scale datasets across heterogeneous platforms.

The Presto ecosystem has expanded significantly since the original development of PrestoDB by Facebook. In particular, Trino emerged as a direct fork of PrestoDB, introducing new features and improvements. Additionally, commercial platforms such as Starburst and Ahana have built upon the foundations of PrestoDB, offering enterprise-grade features, enhanced support, and tailored solutions for large-scale analytics.

In this study, we evaluate the performance of PrestoDB by benchmarking its distributed query execution capabilities under various scenarios. Using the TPC-DS benchmark, we design and execute queries over data partitioned across three distinct storage systems: PostgreSQL, Cassandra, and MongoDB. These systems represent a range of data models, providing a versatile environment for performance evaluation. Key aspects under analysis include query latency, optimizer efficiency, memory utilization and scalability with varying numbers of worker nodes.

By experimenting with different data distribution strategies and analyzing their impact on PrestoDB's distributed execution and optimization mechanisms, we aim to identify engine's ability to query non uniform data sources and propose optimized partitioning strategies. Additionally, this paper provides a guide for replicating the PrestoDB setup.

II. System Components and Technologies Used

The developed system comprises four (4) Virtual Machines hosted on Okeanos Knossos [4]. In this setup, PrestoDB is connected with PostgreSQL, Cassandra and MongoDB databases. The system's performance is evaluated using the TPC-DS Benchmark Suite, with Presto's TPC-DS connector facilitating the execution of benchmarks. All four VMs are interconnected via a private network.

A. PrestoDB

PrestoDB is an open-source distributed SQL engine designed to efficiently query large amounts of data—ranging from gigabytes to petabytes—across various data sources [5]. It is ideal for federated querying, seamlessly integrating traditional relational databases (e.g., MySQL, PostgreSQL) with NoSQL systems (e.g., Cassandra, MongoDB). PrestoDB excels in analytics and data warehousing tasks, supporting online analytical processing (OLAP) and generating reports through SQL. Unlike traditional databases, PrestoDB does not store data. Instead, it interfaces with heterogeneous data sources such as Hadoop, relational DBMSs, and streaming systems via connectors. Its core architecture features a coordinator node and multiple worker nodes, enabling horizontal scaling and parallel query execution across a cluster.

Presto enables efficient distributed query execution by dividing data into splits and and assigning tasks across workers. This architecture allows the engine to process large datasets in parallel, providing high performance for federated queries.

The system uses a discovery service that ensures coordination by maintaining a registry of active worker nodes through periodic heartbeats. It simplifies deployment, as it is typically embedded within the coordinator.

a) Core Components:

- Coordinator: Receives SQL queries from users, parses and optimizes them, plans execution, and distributes tasks to worker nodes. It consolidates results from workers and serves them to the client using HTTP/HTTPS protocols.
- Workers: Execute tasks assigned by the coordinator, fetch data using connectors, and exchange intermediate results. These nodes are registered and monitored via a discovery service for task assignment.
- Connectors: Provide an abstraction layer for accessing different data sources, mapping their data into tables,columns and rows. PrestoDB supports connectors for a variety of sources, including Hive, MySQL and Redis.

b) Web UI: Presto also includes a web interface accessible at the same address as the coordinator server on port 8080 by default. The UI displays utilization metrics, active queries, and additional cluster management details for seamless operation and monitoring.

B. PostgreSQL

PostgreSQL is an open-source Object-Relational Database Management System (ORDBMS) that can store not only traditional relational data but also complex and NoSQL data types like JSON, XML and others. It is highly scalable, suitable for single machines or large-scale deployments, and operates on major operating systems.

Key features of PostgreSQL include its extensibility and SQL compliance, allowing support for custom data types, operators, and functions tailored to developer needs. It ensures data reliability and integrity through ACID compliance, which encompasses Atomicity, Consistency, Isolation, and Durability. The database offers advanced capabilities such as materialized views, foreign keys, and updatable views, enhancing functionality and control. With Multi-Version Concurrency Control (MVCC), PostgreSQL enables simultaneous transactions, ensuring high-performance operations and scalability. Additionally, its robust query optimization features include an advanced query planner for efficient execution of complex queries, such as parallel query execution and join optimizations.

C. Cassandra

Cassandra is an open-source, distributed NoSQL database designed for scalability, high availability, and reliable performance. It excels at managing massive amounts of data across commodity servers or cloud infrastructure with no single point of failure. Cassandra's architecture features peer-to-peer communication, enabling every node in the cluster to have the same role, and ensures fault tolerance through data replication across multiple nodes and data centers. This design enables Cassandra to handle high-speed data writes, manage hundreds of thousands of operations per second, and provide linear scalability by simply adding nodes to the cluster without downtime.

Key features include its efficient hashing algorithm for quick data storage decisions, the absence of a central ”master” node, and the use of the ”gossip protocol” for seamless node communication and metadata sharing. Cassandra prioritizes availability and partition tolerance over strict consistency, making it well-suited for mission-critical applications that require reliability. Data replication ensures high availability, allowing operations to continue uninterrupted even if individual nodes or entire data centers fail.

Additionally, Cassandra offers robust performance optimization for both synchronous and asynchronous operations and supports observability and security with minimal impact on workloads. Its proven fault tolerance, horizontal scalability, and high throughput make it an ideal choice for applications demanding speed, efficiency, and resilience.

D. MongoDB

MongoDB is an open-source document database designed for scalability, flexibility, and performance, making it ideal for building highly available internet applications. Unlike traditional SQL databases that store data in tables, MongoDB uses BSON, a binary format for documents, allowing flexible schemas and efficient querying and indexing. It supports various indexing methods, including text and partial indexing, and retrieves data in JSON format, which aligns seamlessly with native objects in modern programming languages.

MongoDB can handle high volumes of data and scale both vertically and horizontally to meet growing demands. Through sharding, it distributes data across multiple servers using a shard key for efficient load balancing and uniform data distribution. MongoDB also provides robust aggregation capabilities via pipelines, enabling advanced data processing like grouping, totaling, and calculating averages or maximum values.

E. TPC-DS Benchmark

TPC-DS is a decision support benchmark designed to evaluate the performance of data warehousing and OLAP systems. It models essential aspects of decision support systems, including query processing and data maintenance, and provides tools to generate datasets of varying sizes. TPC-DS employs a snowflake schema to reflect real-world retail scenarios. The benchmark includes 99 SQL queries, covering various workloads. These queries are designed to evaluate performance under complex, multi-user decision support workloads, making TPC-DS a reliable tool for assessing and optimizing modern data systems. In our project, TPC-DS is utilized for data and query generation.

The benchmark's schema consists of multiple fact and dimension tables. TPC-DS models the inventory, sales, and sales returns processes for an organization operating through three primary sales channels: stores, catalogs, and the internet. The schema is structured with seven fact tables, including pairs of tables dedicated to capturing product sales and returns for each sales channel and a single table representing inventory for catalog and internet sales. Dimension tables feature single-column surrogate keys, which are used to establish joins with fact tables, ensuring efficient relationships between the schema components.

According to the TPC-DS specification[], while there are many ways to categorize the queries it generates, one basic classification is:

- Reporting queries: They involve queries that are run regularly to answer specific, predefined questions regarding a business's financial and operational performance. While reporting queries are generally static, small adjustments are common.
- Ad hoc queries: These queries reflect the dynamic nature of a decision support system, where spontaneous queries are created to address immediate and specific business needs. The main distinction between ad hoc queries and reporting queries lies in the minimal level of prior knowledge available to the System Administrator (SysAdmin) when preparing for an ad hoc query.
- Iterative OLAP queries: These queries enable users to explore and analyze business data to uncover new and significant relationships and trends.
- Data Mining queries: Data mining is the technique of analyzing large amounts of data to identify relationships within the data. It helps forecast future patterns and trends, enabling businesses to make informed, proactive decisions. Queries in this category usually involve joins and extensive aggregations that generate large result sets, which can be extracted for further analysis.

F. Docker

Docker is a platform for developing, shipping, and running applications in lightweight, portable containers [6]. Containers package an application and its dependencies into a single unit, ensuring consistency across different environments (development, testing, production). Docker simplifies deployment, scaling, and management of applications.

Some key features that docker and it's tools include is:

- docker: service containerization, machine portability, workflow efficiency, workflow version control, service execution isolation, service scalability
- docker-compose: multi-container application, easy container configuration, file volume mapping
- docker swarm: seamless container scaling, container cluster management and networking across different hosts, load balancing, rolling updates

III. System Setup and Replication

The project is supported by a Github repository with scripts and howtos README.md which are accessible via the following link: GitHub Repository.

This section provides important information regarding the setup of the Presto cluster, the generation of the dataset, the generation of the queries and the installation, configuration and population of the data sources on each server node.

Resources Description

Four virtual machines (VMs) were provided by okeanos-knossos for this project (lets name them adis1, adis2, adis3, adis4). Each VM has the following specifications:

- CPU: Intel(R) Xeon(R) CPU E5-2650 v3 - 4 cores
- RAM: 8GB
- Disk: 30GB HDD (28GB actual)
- OS: Ubuntu 22.04.5 LTS
- External Network Bandwidth: 500 Mbps
- Internal Network Bandwidth: 800 Mbps

These four VMs were given in pairs (adis1-adis3, adis2-adis4). So the configuration for the networks is described with the format: network(subnet)

- adis1
  - public_ipv4_network1 (83.212.75.0/24)
  - public_ipv6_network1 (2001:648:2ffe:501::/64)
- adis2
  - localnet1 (192.168.0.0/24)
  - public_ipv6_network1 (2001:648:2ffe:501::/64)
- adis3
  - public_ipv4_network2 (83.212.75.0/24)
  - public_ipv6_network2 (2001:648:2ffe:501::/64)
- adis4
  - localnet2 (192.168.0.0/24)
  - public_ipv6_network2 (2001:648:2ffe:501::/64)

We notice that adis1 and adis3 belong both in an IPv4 and IPv6 public network, which is called dual stack [3]. The advantages of dual stack is that it can use both protocols with their corresponding services so if a service has only an IPv4 or IPv6 interface it has limited access to certain machines that got one of these protocols.

System Deployment

During the deployment process of Presto services and databases, we utilized a set of bash scripts and Docker services/ tools. This approach allows for smooth cluster reconfiguration, providing flexible and scalable options for redeploying Presto workers across various VMs with differing replication factors, as well as distributing databases with varied data distributions. Now let's get into the specific steps for the system deployment process.

Step0 - Prerequisites: Presto on most modern Linux distributions requires a Java Virtual Machine (JVM) and a Python installation. Our deployment relies solely on Docker images pulled from Docker Hub for Presto and the databases which includes the installation of any dependency needed.

Therefore, in order to get the repository and deploy the services someone needs to have installed the following in every Virtual Machine they use:

- git
- docker
- docker-compose

Detailed description on how to install them is given in the README.md file of our repository.

Step1 - Service Dockerization: It was deemed optimal to implement the services' deployment as docker containers. More precisely, all the services (presto coordinator, presto workers, databases) are described as specifications in a *./docker-compose.yaml* file. When we run the *docker-compose up* command, the required images are pulled from docker hub, which is an extensive online repository of images of various services, as specific versions to avoid any inconsistencies with the initial configuration. The Presto-coordinator, PostgreSQL, MongoDB and Cassandra containers include port forwarding in order for them to be available as services outside the scope of the internal docker network and on to our machine. Additionally, all the services can be referenced by localhost:service_port.

Step2 - Service Configuration: **PrestoDB**— Now we will try to give an overview about the configuration of the Presto cluster. Inside the repository, */presto-coordinator* and */presto-workers* directories are created in order to contain the Presto node configuration files. Through docker-compose setup for Presto services deploying, those files are mounted as volumes to *presto-server/etc* path inside the Presto containers. Those files are present in every node:

- node.properties: This file contains configuration specific to a single installed instance of Presto on a server, such as the environment name and the unique identifier.
  Most important attributes used: discovery.uri=container_name:port, query.max-total-memory
- config.properties: It contains the configuration of the Presto server. It provides critical information such as the role of the server (coordinator and/or worker) and the port of the HTTP server.
  Most important attributes used: discovery.uri=container_name:port, query.max-total-memory, query.max-stage-count
- jvm.config: It contains a list of command-line options used for launching the Java Virtual Machine running Presto.
  Most important attributes used: Xmx3G (defines heap memory)
- log.properties: This file is optional and allows setting the minimum log level (INFO-the default one, DEBUG, WARN, ERROR).
- catalog: This is a directory containing the files with the attributes needed for the Presto cluster to connect to the various data sources.
  Most important attributes used: connector.name, connection ur ls, connection ports, database credentials

For the workers additionally, we used spill-enabled in order to make use of the leftover secondary disk space whenever the main memory was filled completely and thus we were able to run more queries.

Databases— For PostgreSQL main parameters included: wal_level=minimal, wal_keep_size=0 essentially skipping the write ahead logging mechanism that sotres additional data as population checkpoints rendering the population stage slow. The rest was environment connection variables defined in *docker-compose.yaml*.
For MongoDB and Cassandra the only parametrization done was regarding to creating superusers for the database.

Step 3 - TPC-DS Utilization: The TPC-DS package was used to create a dataset containing all the available tables of the schema with the scale being set to 12, thus 12 GB of data were created. We faced restriction for the size of data as far as data location and database index tables that generated a lot of extra data.
Additionally, the dataset required further clean-up to replace all the "," characters with "-" (in order to maintain CSV field consistency) and then to be converted to CSV format with the *./utils/make_csv* script. This script gets as input the directory path of the generated data and creates the .csv interpretations in *./data/csv_data* directory and a *./data/tables_catalog.txt* file that contains all the table names. A key point is that the gcc version 9 package is required to generate the *dsdgen* and *dsqgen* programs.
Afterward, we generated all the available queries with the script that we made *./utils/generate_queries.sh* which uses the *dsqgen* program to create the "*.tpl" query interpretations. Then the script converts the queries to SQL format and saves them in the *./utils/initial_queries* directory.
The final and most crucial step before being ready to populate the databases with the data generated is to make the table definition translation for each table. With the script *./utils/table_scripts/database_population_scripts.sh* we can generate a piece of code that if executed it creates the table for the equivalent database and contains a command to copy all the matching csv data. This script gets all the table names from tables_catalog.txt and the tables definitions from *tpc-ds.sql* provided by tpc-ds package and generates separate files with tables definitions and table population commands and stores them in *./datasources/table_population_scripts/[database]* path.

Step 4 - Services Deployment: For the deployment of the services the docker swarm orchestration tool was used. For the setup of the cluster we initialize the docker swarm cluster on adis1. Then from each of the VMs adis2, adis3, adis4 we join these machines as worker nodes to the cluster. Then we create the necessary labels for each of the VMs.
Moreover, in order to really deploy each service to the proper VMs all we have to do is to add in the *./docker-compose.yaml*, to all the services, a deploy attribute with a label constraint of the VM that we want it to be deployed on. Any time we make a change to a file concerning a service that is matched with a VM we need to pass these changes to the project files of this VM as well. For this purpose the *./utils/file_mirroring.sh* script was made to automatically synchronize the changes made to

adis1 to the rest of the VMs (utilizing the passwordless ssh setup we have created for every machine from adis1).

A functionality to automate the process of populating the databases has been added too. With the use of the script *./utils/population_scripts/[database]_generate* and the table names that we want the database to be populated with, separated with space as parameters a *./datasources/db_population_scripts* file is created with the table definitions and copy commands for each database. This file is mounted on each database container under */docker-entrypoint-initdb.d/* and when the container is initialized it is executed. The sole exception is Cassandra where we have to create a separate docker service to populate it automatically.

The most paramount configuration that our *docker-compose.yaml* file includes, is the description of an overlay network that every service inside it is connected to. This gives us the opportunity to behave to each container on different VMs as if they run on the same internal docker network.

We created the cluster, distributed Presto services and populated the databases, but we are not done yet. The final step before we can commence our experiment, after we decided on the data distribution would be to, is to run the *./utils/fully_define_query.sh* in order to get the queries with the tables defined in the format ”database.schema.table” according to which database they belong for Presto to be able to recognize where each table comes from.

In order to run the different experiments we had to pass the CSV files containing the tables of the new data distribution to each VM, then repopulate the database. For the experiments with a different amount of workers all we have to do is define which worker need to be deployed on which VM.

To conclude, this deployment schema allows us to change the table distribution of the databases and the placement and cardinality of the services according to will. Separate *docker-compose.yaml* files can be created with the configuration needed for each experiment and then all we had to do to run this experiment would be to run a single script.

IV. Methodology

This section explores executing queries under various scenarios of data distribution and worker configurations to understand the behavior of our PrestoDB system and identify an optimized strategy. Before analyzing the experiments methodology, we provide a description of the query statement processing in Presto.

A. Query Execution Model

In Presto, coordinator and workers cooperate to process SQL queries efficiently. The coordinator receives SQL statements from the client, parses, and analyzes them. Using the metadata SPI, the coordinator gathers information about tables, columns, and data types, validates the query, and performs type and security checks. Then, Presto proceeds to query planning, which outlines the steps required to process the data.

The query plan is broken into distributed stages and tasks, with each task processing a split, the smallest unit of parallelism and work assignment. Workers execute these tasks in parallel, coordinated by the dependency tree created from the distributed query plan, enabling efficient processing. Optimizations like predicate pushdown, cross-join elimination, TopN, and partial aggregations further enhance performance by reducing data early, minimizing unnecessary operations, and streamlining intermediate computations.

Presto implements some Optimization Rules, but specifically for the Hive connector, it also supports cost-based optimization. Consequently, in our project, Presto utilizes only the rule-based optimizer. The optimization rules implemented in Presto are the following:

- Predicate Pushdown: This optimization minimizes data early in the query process by pushing filtering conditions closer to the data source.
- Cross Join Elimination: It rearranges the order of table joins to reduce or entirely eliminate cross joins, thereby improving efficiency.
- TopN: Applied when a query includes an ORDER BY followed by a LIMIT, this rule optimizes execution by maintaining only the required number of rows in a heap structure, updating it dynamically as input data is streamed.
- Partial Aggregations: It reduces the amount of data processed during query execution by performing pre-aggregations before joins or other downstream operations. For example, if a query involves calculating the sum of a column grouped by a key, Presto computes partial aggregates early in the process. Instead of passing all rows from the data source to subsequent stages, it aggregates the data to a smaller intermediate form. Although these pre-aggregated results are not final, they significantly reduce the volume of data, leading to faster and more efficient query execution.

B. Queries selection

To evaluate Presto’s performance and its optimizer, we executed a chosen subset of TPC-DS queries that met specific criteria. First, the selected queries needed to encompass all the input tables in the schema, allowing us to evaluate how well each table was distributed across the three databases.

Additionally, the selection included queries of varying complexity, ranging from basic SELECT statements to those involving multiple joins and aggregations. To ensure this, we followed the query classification from [1], which categorizes queries into five scenarios: basic queries, join queries, aggregate queries, subqueries, and window function queries. Furthermore, we ensured that the selected queries covered a range of operational requirements and spanned all the classes mentioned in the TPC-DS specification file [2], including reporting, ad-hoc, iterative, and data mining queries.

Table I shows the ten selected queries executed in Presto. It is evident that these queries cover the entire schema, except for the time_dim table, and span the full query complexity range.

TABLE I: Table Used In Queries

Tables	**Queries**									
	Basic query		**Join query**		**Aggregate query**		**Subquery**		**Window function**	
	Q9	***Q14***	***Q23***	***Q37***	***Q47***	***Q49***	***Q64***	***Q77***	***Q80***	***Q99***
call_center										✓
catalog_page									✓	
catalog_returns						✓	✓	✓	✓	
catalog_sales		✓	✓	✓		✓	✓	✓	✓	✓
customer			✓				✓			✓
customer_address							✓			
customer_demographics							✓			
date_dim		✓	✓	✓	✓	✓	✓	✓	✓	✓
household_demographics							✓			
income_band							✓			
inventory				✓						
item		✓	✓	✓	✓	✓	✓	✓	✓	
promotion							✓		✓	
reason	✓									
ship_mode										✓
store					✓		✓	✓	✓	
store_returns						✓	✓	✓	✓	
store_sales	✓	✓	✓		✓	✓	✓	✓	✓	
time_dim										
warehouse										✓
web_page								✓		
web_returns						✓		✓		
web_sales		✓	✓			✓		✓		
web_site									✓	

TABLE II: Storage Technology Data partition

Tables	***PostgreSQL***	***Cassandra***	***MongoDB***
call_center			✓
catalog_page			✓
catalog_returns		✓	
catalog_sales	✓		
customer			✓
customer_address			✓
customer_demographics	✓		
date_dim	✓		
household_demographics			✓
income_band			✓
inventory		✓	
item	✓		
promotion			✓
reason		✓	
ship_mode		✓	
store	✓		
store_returns		✓	
store_sales	✓		
time_dim	✓		
warehouse	✓		
web_page			✓
web_returns		✓	
web_sales	✓		
web_site	✓		

C. Data Distribution Strategy

As an initial approach, a basic method to distribute data involves equally splitting random gigabytes of data among databases. However, this approach is deemed suboptimal and excluded from performance experiments.

The partitioning strategies that will be evaluated are the following:

1) *Storage technology based strategy*: The goal of this partitioning strategy is to leverage the strengths of different database technologies. Generally, each database is optimized for specific data types and workloads based on its architecture.

   PostgreSQL is utilized for complex queries and transactional integrity, making it the ideal choice for storing large fact tables (‘store_sales‘, ‘web_sales‘, ‘catalog_sales‘) and supporting analytical workloads that require efficient joins and aggregations. Cassandra, known for its high-write performance, is utilized for tables with frequent updates (‘inventory‘) and time-series-like data. MongoDB, with its flexible schema design, is suited for semi-structured or hierarchical data, enabling efficient storage and retrieval of smaller dimension tables (‘customer‘). By taking into consideration the capabilities of each data storage, this approach is expected to sustain good query performance. Table II depicts the data distribution derived from this strategy.
2) *Entity-Relationship (ER) based strategy*: The previous

approach separates related tables, such as sales and returns fact tables, causing frequent data transfers during joins. Since our snowflake schema heavily relies on joins, causing bottleneck in bandwidth, the first method led to significant performance issues.

Thus, the second partitioning strategy focuses on optimizing query execution by leveraging data locality and balanced workloads across the databases. This approach is based on the entity-relationship (ER) model of the fact tables, while aiming to maintain a relatively even distribution of data across the three databases—PostgreSQL, Cassandra and MongoDB.

In this strategy, related fact tables, such as sales and returns for each category, are stored together in the same database. To further enhance performance, dimension tables directly linked to fact tables (sales, web_page, catalog_page) are stored in the same database as the respective fact table, following the entity-relationship (ER) model. The initial allocation of categories to databases is guided by the performance results of the first strategy, ensuring that larger datasets are assigned to more performant databases like PostgreSQL.

The inventory fact table, along with the rest of the dimension tables, will be allocated to the database containing the smallest amount of data following the initial distribution of the sales and returns fact tables. Any remaining tables shared across all categories will be distributed based on their size. Table III depicts the data distribution derived from this strategy.

This strategy achieves two key objectives. Firstly, the data distribution is balanced across the databases based on their inherent properties. Secondly, co-locating related tables ensures joins will take place locally in each node and minimizes inter-database data transfers, thereby reducing network traffic and improving join performance.

Considering PostgreSQL's superior support for complex operations like joins, critical dimension tables such as date_dim and item were centralized in PostgreSQL, leading to improved query performance. These tables are heavily used in queries and play a pivotal role in join operations.

D. Worker configurations

Up to this point, we have explored how various data distribution strategies impact query performance. In this section, we shift our focus to the Presto configuration and its effect on query execution times. Using the Storage technology based distribution strategy from the previous section, we evaluate the system's scalability by executing the same query suite with 1, 2, and 3 Presto workers.

For the 2-worker configuration, we deactivate the nodes hosting the PostgreSQL and Cassandra databases. This decision is based on insights from earlier experiments, which highlighted the higher bandwidth capabilities and data transmission rates of those databases. Allowing PostgreSQL and Cassandra to serve as the main databases for data transmission ensures efficient network utilization. Additionally, one of the workers is configured on the same node as the worker. This setup prevents one database from sharing RAM with the worker, allowing them to allocate more memory for query processing yet sacrificing data locality for in-node processing. Moreover, increased RAM availability reduces disk I/O, leading to faster query execution.

TABLE III: ER based Data partition

Tables	***PostgreSQL***	***Cassandra***	***MongoDB***
call_center		✓	
catalog_page		✓	
catalog_returns		✓	
catalog_sales		✓	
customer	✓		
customer_address	✓		
customer_demographics	✓		
date_dim	✓		
household_demographics		✓	
income_band			✓
inventory	✓		
item	✓		
promotion			✓
reason			✓
ship_mode			✓
store	✓		
store_returns	✓		
store_sales	✓		
time_dim	✓		
warehouse	✓		
web_page			✓
web_returns			✓
web_sales			✓
web_site			✓

In the single-worker setup, nodes hosting the three databases are deactivated. This configuration slightly compromises performance, as the coordinator and the worker share the same node's RAM, similar to the previous setup. This approach aims to optimize scalability while accounting for system resource allocation and data movement efficiency.

The configurations examined are shown in Table IV.

V. Results

In this section we will provide several results from the aforementioned queries across different clusters. In the first part we will discuss how different numbers of workers affect the query execution and in the second part we will examine the impact of the different data distributions. Also the metrics which will be discussed are the following:

- **Execution time**: Describes the overall time required to run the query.
- **CPU time**: The sum of the time spent from each node's CPU for the query processing, networking and query execution.

TABLE IV: Worker configurations

Number of Workers	***adis1***	***adis2***	***adis3***	***adis4***
3	Coordinator	Worker 1 + MongoDB	Worker 2 Cassandra	Worker 3 + PostgreSQL
2	Coordinator + Worker 1	Worker 2 + MongoDB	Cassandra	PostgreSQL
1	Coordinator + Worker 1	MongoDB	Cassandra	PostgreSQL

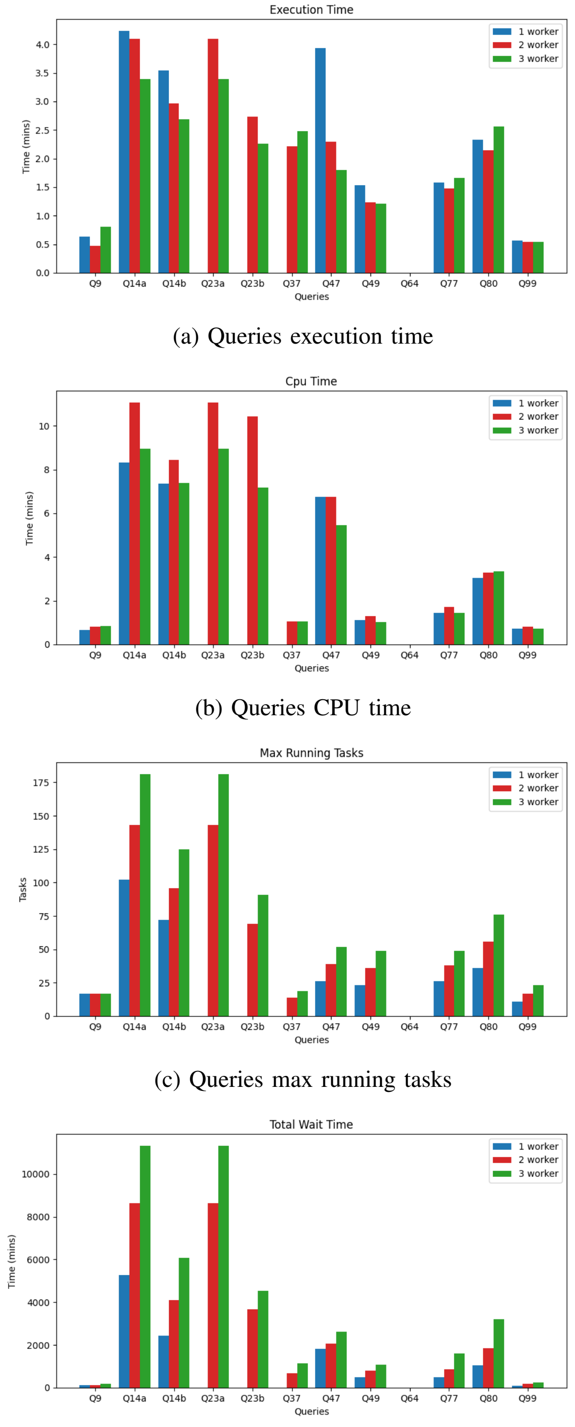
- **Max running tasks**: The maximum concurrently running tasks while the query was executing.
- **Total Wait Time**: The total time all tasks spent waiting due to various resource constraints during query execution.

Also we have implemented a disk spill strategy which offloads memory usage by utilizing disk space and therefore any memory related results do not represent actual memory usage and were omitted for this study.

A. Results per number of workers

First of all we have to mention that with this data distribution, presto was not able to execute query 64 with any of the selected worker configurations and encountered insufficient memory issues which will be discussed in the next comparative subsection. Also queries 14 and 23 contained more than one query in their definition which has resulted in 2 entries (variants a and b) in the result plots for each one of them.

Moving on, according to the plots of Figure 1 the configuration with 1 worker also encountered insufficient memory issues when trying to execute queries 23a, 23b and 37 aside from the query 64. Firstly we can view the execution and CPU time of each query according to the figures 1(a) and 1(b) and immediately we can distinguish simple queries from more complex ones. Queries with large table joins and aggregations such as queries 14 and 23 take longer to execute mainly due to the large sized tables used but that time decreases with the addition of more workers. At the same time the CPU time increases which suggests larger worker coordination and data shuffle costs. Queries 47 and 80 perform heavy calculations and while query 47 benefits from added workers both in execution and CPU time, the query 80 actually performs worse with more workers which can be attributed to excessive worker communication. The queries 37, 49 and 77 are intermediate complexity queries which seem to perform same or worse with the addition of a third worker while having very small differences in the CPU time which suggests I/O bottlenecks. Simple queries like query 9 and 99 are consistently fast but they do not benefit from additional workers because they introduce more network overhead, especially query 9 which shows increased execution time when processed by 3 workers. Additionally looking at the plots in the Figures 1(c) and 1(d) we can have a good idea of the way the optimizer works since we can see a universal increase in the number of max running tasks and total wait time per additional worker. For the more complex queries (14, 23), the number of additional running tasks per worker is greater since they process large data partitions in parallel which creates more



(a) Queries execution time

(b) Queries CPU time

(c) Queries max running tasks

(d) Queries total tasks wait time

Fig. 1: Queries results for multiple worker configurations

tasks and unavoidably leads to contention for resources and increased wait times. The same behavior in smaller scale can be seen for heavy computation query 47 and the simple query 99 which perform better with more workers while increasing the expected running tasks in parallel and their waiting times. From these plots we can further validate the excessive worker communication problems that query 80 is facing since there is a moderate increase in the total concurrent tasks and especially their wait times when adding a third worker while performing worse. Also the I/O bottleneck of the queries 37, 49 and 77 is visible again in these plots since there is an increase in tasks running in parallel while keeping relatively the same CPU time. On the other hand, query 9 is the only one that doesn't show an increase in maximum concurrent tasks which is mainly because the query is small enough to be described by the same amount of tasks regardless of the number of workers participating in it's processing which further validates the fact that 3 workers are not only unnecessary but also unfavorable in this specific scenario.

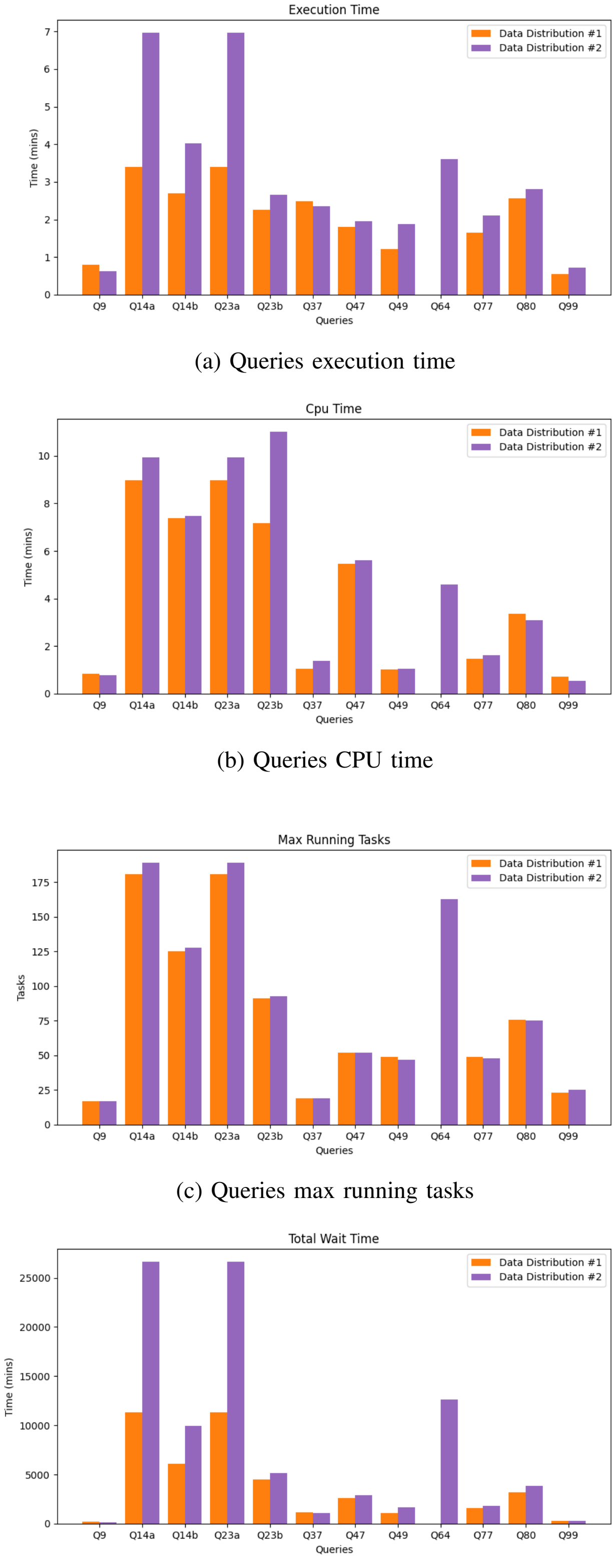
Finally it is worth mentioning that we observed a trade-off between maximum concurrent split processing and network overhead. If there is excessive parallelism then there exists an amount of lost time spent for worker-to-worker communication and while increasing parallelism helps with workload distribution, too many concurrent splits can lead to network congestion, synchronization overhead, and worker inefficiencies.

B. Results per data distribution

In Figures 2(a) , 2(b) , 2(c) , and 2(d) , we present the results of the metrics comparison between the previous configuration with Storage Technology-based data partitioning and the ER-based data partitioning, both using 3 workers in the cluster. Two main observations can be made: firstly, the initial distribution consistently exhibits lower execution times compared to the second one; and secondly, Q64 is successfully executed only in the second partition. This behavior can be explained by considering that these partitioning strategies possess complementary features.

As mentioned earlier, the first partitioning approach hosts large fact tables in PostgreSQL, enabling fast joins within the database's node. Notably, Q14 and Q23 exhibit significantly lower execution times under the database-based distribution compared to the ER-based distribution. This performance improvement is attributed to the fact that tables such as store_sales, catalog_sales, web_sales, date_dim, and item are stored in PostgreSQL. In contrast, the second distribution, which focuses on leveraging locality between fact and dimension tables, performs poorly for queries involving tables from different channels.

The ER-based strategy efficiently handles queries that involve joins between tables within the same channel (e.g., catalogs), such as Q64 (implicit joins: catalog_sales–catalog_returns and store_sales–store). This is lacking in the first partition, which exhibits slower cross-database joins (e.g., catalog_sales in PostgreSQL and catalog_returns in Cassandra) due to data



(a) Queries execution time

(b) Queries CPU time

(c) Queries max running tasks

(d) Queries total tasks wait time

Fig. 2: Queries results for Storage Technology based partition (orange) and ER based partition (purple)

shuffling. Thus, we can conclude that when using the first distribution, Q64 suffers from high network and memory overhead due to cross-database joins, an issue that is alleviated with the second distribution. However, apart from Q64, other queries involving intra-channel joins (e.g., Q49, Q77) do not benefit from the ER-based partition. This can be attributed to the fact that the entire date_dim table is stored in PostgreSQL, requiring data exchange with nodes containing catalog_sales and web_sales.

Also, the CPU time aligns with execution time. The second distribution generally consumes less CPU time than the first for most queries. Regarding the maximum number of running tasks across the data distributions, we observe that they are comparable, as both experiments are conducted with three workers.

Additionally, we observe that the second distribution shows more maximum parallel running tasks compared to the first one. However, the lack of corresponding performance improvements (as shown by execution and CPU times) suggests network and memory overheads. Therefore, ER distribution does not strike a balance between shuffles and parallelism, as evidenced by the increased wait time.

Overall, Storage Technology data distribution demonstrates superior efficiency compared to the ER distribution, consistently outperforming it in both execution time and CPU time, making it the more efficient choice for the given workload. Moreover, despite the increased drivers in the second distribution, there is no corresponding improvement in performance. This suggests potential issues with scalability or inefficiencies in the query execution plans associated with the ER distribution.

VI. Conclusions and Further Work

In our experiments, we assessed Presto's performance, scalability, and built-in optimizer with various data partitioning methods and different numbers of worker nodes. The results demonstrated that Presto is an exceptionally versatile and powerful tool when its operation is well understood. A thorough understanding of the data sources and query types, combined with a well-resourced infrastructure, can accommodate any requirements.

Future efforts might focus on developing a modular infrastructure to support experiments. In terms of resources, additional scripts could automate executing experiments with various configurations, such as worker count and data partitions. Eliminating the factor of human intervention to modulate the deployment could yield a larger number of results in shorter time. Moreover, this could help determine the most effective setup by organizing results in specific data formats and analytics could be performed to indicate the optimal partition. Finally, more infrastructure resources could be employed to evaluate Presto's scalability.

References

[1] https://byconity.github.io/docs/benchmarks/tpc-ds

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

[3] http://juniper.net/documentation/us/en/software/junos/is-is/topics/concept/ipv6-dual-stack-understanding.html

[4] https://okeanos-knossos.grnet.gr/about/what/.

[5] https://prestodb.io/what-is-presto/

[6] https://docs.docker.com/get-started/docker-overview/