Distributed Execution of SQL Queries Using Presto

Team ID 3

Pavlana Loukia

Παυλανά Λουκία

School of Electrical and Computer Engineering

National Technical University of Athens

Athens, Greece

AM el18711

el18711@mail.ntua.gr

Hadjisavvas Andreas

Χατζησάββας Ανδρέας

School of Electrical and Computer Engineering

National Technical University of Athens

Athens, Greece

AM el18701

el18701@mail.ntua.gr

GitHub Repository:

https://github.com/LoukiaPavlana/presto-data-distribution

***Abstract* —This project explores the integration of multiple Database Management Systems (DBMS)—MySQL, MongoDB, and In Memory Data —with Presto, a distributed query engine. The goal is to showcase the power of distributed query processing in handling large datasets across different databases. By distributing queries across multiple nodes using Presto, this approach enables efficient, scalable processing of data from diverse sources, demonstrating the potential of federated querying in modern data ecosystems.**

Keywords— query engines, Presto, heterogeneous data sources, MongoDB, In Memory Data, MySQL, Distributed databases.

I. INTRODUCTION

In recent years, the demand for data analytics on massive datasets spanning multiple, diverse data sources has grown significantly. As business operations become increasingly intricate and the volume of data expands, a major challenge in data processing lies in the ability to efficiently query data from inconsistent and heterogeneous sources. Query engines address this issue by enabling users to access and analyze data from multiple sources through a unified schema.

This project focuses on evaluating one such query engine, Presto. By employing the TPC-DS benchmark, we assess Presto's performance under various data partitioning techniques, utilizing data distributed across three distinct databases: MySQL, PostgreSQL, and In Memory Data. Initially, we conduct experiments with representative queries applied to partitions designed based on theoretical insights into the TPC-DS dataset. From these findings, we develop an optimized data partitioning approach informed by both theoretical analysis and experimental results on query performance.

Subsequently, we test the system's scalability by executing the same set of representative queries on the optimized data partition while varying the number of Trino worker nodes. Additionally, we analyze the performance of Presto's optimizer when applied to different data partitions, demonstrating its capability to consistently select the optimal query execution plan.

The overarching goal of this project is to thoroughly evaluate Presto's effectiveness in querying non-uniform data sources, assess its scalability under

varying conditions, and validate the efficiency of its built-in query optimizer.

II. DATABASES AND QUERY ENGINE OVERVIEW

For the implementation of this project, we are using three Database Management Systems (DBMS) and one query engine.

A. MySQL

MySQL is one of the most widely adopted and popular relational database management systems (RDBMS) in use today. Renowned for its strong performance, reliability, and user-friendly design, MySQL is particularly suited for managing structured data within a predefined schema consisting of tables, rows, and columns. This structure makes it an excellent choice for handling traditional transactional workloads such as e-commerce applications, customer relationship management (CRM) systems, and content management systems (CMS).

MySQL stores data in well-organized tables that adhere to predefined schemas, ensuring data consistency and referential integrity through the use of primary and foreign keys. This structured approach not only maintains data accuracy but also enables the efficient execution of complex SQL queries, including joins, aggregations, and subqueries. Such capabilities make MySQL a dependable solution for managing and querying relational data effectively.

B. MongoDB

Unlike MySQL, MongoDB is a NoSQL database specifically designed to manage semi-structured and unstructured data. As an open-source, document-oriented database, MongoDB stores data in BSON (Binary JSON) format, which is similar to JSON. This flexible data model is highly suitable for applications that require support for evolving schemas or hierarchical data structures.

Rather than using the fixed, tabular format typical of relational databases, MongoDB stores data in documents resembling JSON objects. Each document can have a unique structure, making MongoDB particularly effective at managing diverse and dynamic datasets. This adaptability enables agile development, as schema changes can be implemented without causing database downtime or requiring complex schema migrations.

In this project, MongoDB is utilized to handle semi-structured data that cannot be efficiently stored in traditional tables. Its flexibility makes it an ideal choice for managing user profiles, log data, and other types of hierarchical or nested data.

C. In-Memory Data Storage

In-memory data storage systems are designed to store and process data directly in a computer's main memory (RAM) rather than on disk-based storage. This approach significantly reduces latency and enhances performance, making it ideal for applications that require real-time data processing, such as analytics platforms, caching systems, and high-frequency trading.

Unlike traditional disk-based databases, in-memory systems prioritize speed by avoiding the overhead of disk I/O operations. Data is stored in a flat, memory-optimized structure, allowing rapid access and manipulation. These systems often support advanced indexing techniques and optimized algorithms to maximize throughput.

In-memory data storage is particularly suited for scenarios where low latency and high-speed data access are critical. For instance, it is commonly used to store session data, real-time metrics, and transient data that must be processed and retrieved at high speeds. Although it is not typically used for persistent storage due to the volatile nature of RAM, many in-memory systems provide features for periodic data persistence or replication to disk to ensure fault tolerance.

In this project, in-memory storage is used to accelerate query processing by caching intermediate results and frequently accessed data. This enables the system to handle large volumes of queries efficiently while maintaining low response times.

D. Presto

Presto is an open-source distributed SQL query engine designed for fast, interactive analytics on large datasets. It enables querying data from a variety of sources, including relational databases, NoSQL systems, and distributed file systems, without requiring data movement or duplication.

Presto’s architecture is optimized for executing complex queries across heterogeneous data environments with minimal latency, making it an excellent choice for federated querying and large-scale data analysis. Its ability to handle diverse data sources seamlessly allows organizations to unify their data infrastructure and perform high-performance analytics in real time.

III. METHODOLOGY

A. The TPC-DS Benchmark

The TPC-DS benchmark is a decision support benchmark designed to simulate various aspects of decision support systems, including query execution and data management tasks. It provides a standardized framework for evaluating the performance of a System Under Test as a general-purpose decision support platform.

This benchmark is particularly useful for assessing how well a system handles complex queries and large-scale data, offering a comprehensive measure of its capabilities. In this study, the System Under Test is Presto.

1) *Business Model:* TPC-DS utilizes the business model of a large retail company, and is thus able to model any industry that must manage, sell and distribute products. Beyond having multiple stores located nation-wide, the company also sells goods through catalogs and the Internet. Along with tables to model the associated sales and returns, it includes a simple inventory system and a promotion system [1]. The following are a few of the possible business processes of this retail company:

- 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 schema is a snowflake schema that consists of multiple dimension and fact tables. As mentioned before, it models the inventory, the sales and sales returns process for an organization that employs three primary sales channels: stores, catalogs, and the Internet. The schema includes seven fact tables:

- A pair of fact tables focused on the product sales and returns for each of the three channels.
- A single fact table that models inventory for the catalog and internet sales channels.

Each dimension table has a single column surrogate key. The fact tables join with dimensions using each dimension table's surrogate key [1]. The dimension tables can be classified into one of the following types:

- **Static:** The contents of the dimension are loaded once during database load and do not change over time. The date dimension is an example of a static dimension.
- **Historical:** The history of the changes made to the dimension data is maintained by creating multiple rows for a single business key value. Item is an example of a historical dimension.
- **Non-Historical:** The history of the changes made to the dimension data is not maintained. As dimension rows are updated, the previous values are overwritten and this information is lost. Customer is an example of a Non-Historical dimension.

Fig. 1 through Fig. 7 are the snowflake ER diagrams of the seven fact tables, as displayed in the TPC-DS Standard Specification.

B. Data Partitioning

In this section we discuss possible strategies for distributing the data among the three databases connected to our presto cluster.

1. Naive Partitioning:

The simplest approach to distributing the data involves equally dividing the gigabytes of data across the three databases. In practice, this means that unrelated tables would be stored in different databases, such as splitting the sales and returns fact tables of the same category due to their similar large sizes. However, this would result in queries requiring joins between related tables to involve significant data transfers between nodes. Given that our data follows a snowflake schema, queries involving joins are very frequent. Bandwidth limitations, a well-documented bottleneck in Big Data research [14], would significantly hinder performance in this approach. Due to these limitations, we consider this method of partitioning suboptimal and will exclude it from the performance evaluation experiments.

2. Proximity Partitioning:

To address the limitations of the naive method, we distribute the data based on the proximity of related tables while maintaining a balanced distribution of data among the three databases. For instance, the sales and returns tables of a particular category will be stored together in the same database, along with tables specific to that category (e.g., the store table will be stored alongside store sales and store returns). The assignment of categories to each database will be informed by the performance of each database observed during the baseline experiments, which will be explained in a subsequent section. Additionally, the inventory fact table, along with its related snowflake schema tables, will be placed in the database that has the least data after the initial

distribution of the sales and returns tables. Tables that are common to all categories will be distributed based on their size to ensure balance. This partitioning strategy minimizes data exchange between nodes by enabling most joins and filtering operations to occur locally within each database. This data partitioning approach will serve as the foundation for testing the Presto optimizer and making further optimizations

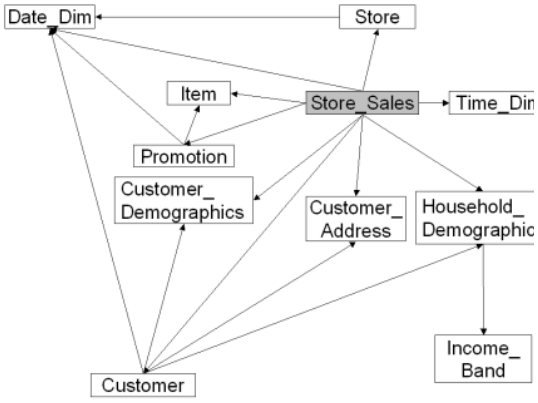

Figure 1: Store Sales ER-Diagram

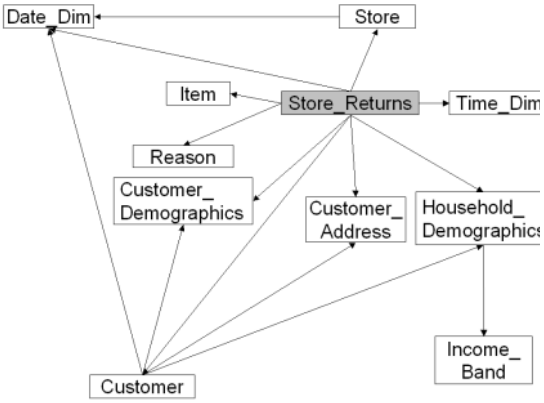

Figure 2: Store Returns ER-Diagram

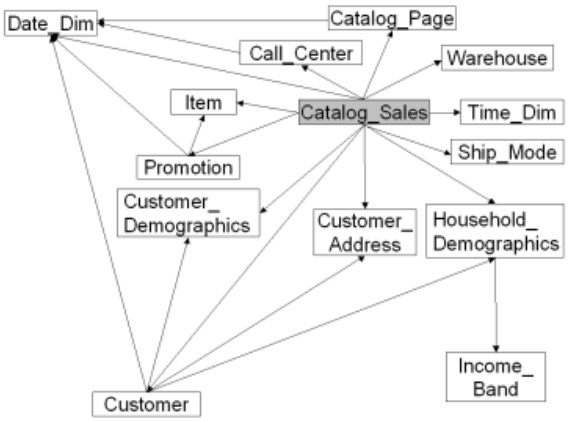

Figure 3: Catalog Sales ER-Diagram

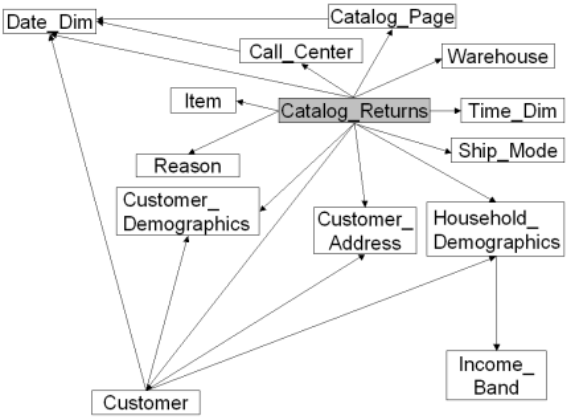

Figure 4: Catalog Returns ER-Diagram

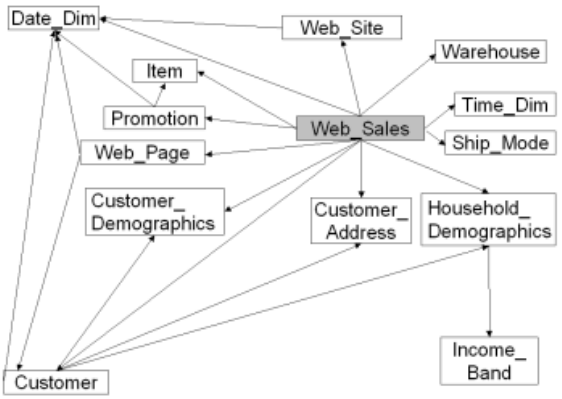

Figure 5: Web Sales ER-Diagram

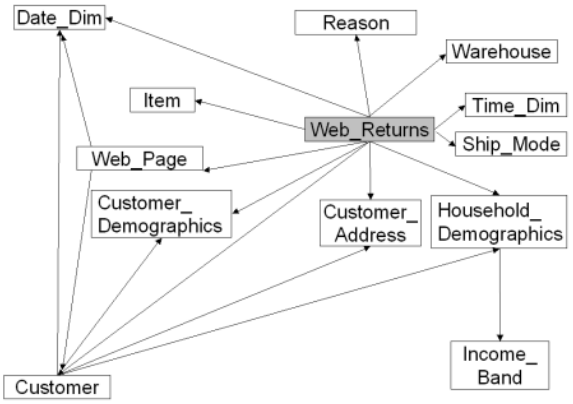

Figure 6: Web Returns ER-Diagram

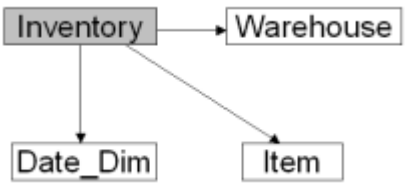

Figure 7: Inventory ER-Diagram

IV. ARCHITECTURE – DEPLOYMENT

A. Deployment & Configuration

All the experiments are carried in a cluster of 3 virtual machines as described in Table I.

Due to a shortage of Virtual Machines (VMs), we are required to deploy database management systems (DBMSes) on the same nodes that are running Presto workers. Additionally, we need to store in-memory data on these same machines, even though they are already hosting other databases.

TABLE I

	VM 1	VM 2	VM3
Cores	4	4	4
System Ram	8 GB	8 GB	8 GB
Disk Size	30 GB	30 GB	30 GB
Local IP	192.168.0.2	192.168.0.1	NaN
Public IP4	83.212.75.173	NaN	83.212.76.29
Public IP6	2001:648:2ffe:501:cc00:13ff:fe68:a322	2001:648:2ffe:501:cc00:13ff:fe51:6f0b	2001:648:2ffe:501:cc00:13ff:feef:b688

TABLE II

	VM 1	VM 2	VM3
Database	MySQL & Memory	MongoDB & Memory	Memory
Role	Coordinator & Worker	Worker	Worker

VM1: This node serves as the Presto Coordinator, responsible for query parsing, planning, and scheduling execution across all workers. Additionally, it also acts as a worker, processing part of the query workload. Alongside Presto, MySQL runs on this machine, sharing resources with in-memory Presto tables. Because of this, memory allocation must be carefully managed to prevent resource contention.

VM2: This node functions purely as a Presto Worker, responsible for executing tasks assigned by the coordinator. It also hosts a MongoDB instance, meaning that queries involving MongoDB data introduce additional processing overhead. Memory storage for in-memory tables is also enabled here, requiring careful monitoring of RAM usage to avoid excessive swapping.

VM3: This is a dedicated Presto Worker that does not run any database services, making it the most optimized for query execution. It is also responsible for storing and processing in-memory tables, working alongside the other two worker nodes to speed up distributed query execution.

B. Trino Configuration

Presto requires several configuration files to set up a cluster properly. These files are located in the etc/ directory, which we created, inside the Presto installation folder. Inside the folder the required Configuration Files are:

- config.properties
- jvm.config
- node.properties
- catalog/*.properties

Inside the catalog folder we configure the data sources which in our case where MySQL, Mongodb and In Memory and the TPCDS connector which provides a set of schemas to support the TPC Benchmark.

V. DATA GENERATION

A. Data generation and segmentation

The data used in this project was exclusively generated using the TPC-DS Benchmark, ensuring both consistency across databases and realistic, real-world data characteristics. After building the TCP-DS data generation tool for database benchmarking, we selected a dataset size of 10 GB, as this volume was deemed representative given the available resources. Considering the limitations of our Virtual Machines' disk capacities, the dataset was generated incrementally, one table at a time. This approach was chosen to avoid overloading the storage with the entire dataset and its corresponding populated database simultaneously, ensuring efficient use of system resources.

The following table provides a detailed overview of the database entities, partitioned using proximity partitioning:

TABLE III

Database	Name	Size	Entries
MySQL	Inventory	2.7G	133110000
	warehouse	4K	10
	item	28M	102000
	date_dim	10M	73049

MongoDB	web_returns	414M	719217
	store_returns	1.2G	2875432
	web_page	0.02M	60
	catalog_sales	3G	1440126 1
	catalog_returns	331M	1439749
	web_sales	1.5G	7068526
	store_sales	3.9G	2880099 1
	catalog_page	3.8M	12000
Memory	customer_address	27M	250000
	household_demographics	156K	7200
	reason	4K	45
	income_band	4K	20
	store	28K	102
	ship_mode	4K	20
	call_center	8K	24
	time_dim	5M	86400
	customer	65M	500000
	promotion	64K	500
	web_site	12K	42
	customer_demographics	79M	1920800
SUM		10.5 GB	191367388

The way we chose the segmentation of the data after considering the previous approaches is based on their ER diagrams that are presented in the official TPCDS instructions manual. According to how 'tightly' they are connected to each other we distinguished 3 main clusters of tables.

MYSQL: The reason we distinguished these 4 tables is the profound cluster of inventory-related entities according to the TPC-DS manual. The size for these data (2.7G) also seems manageable from a MySQl instance

Mongodb: The next cluster has to do with product sales and returns, so all the related tables were chosen to be inserted in MongoDB. Another reason is that these data (7.5G) comprise the majority of the total size, so this Database with its efficient functions will be up to the task when it is time for querying.

In memory: For the in memory part we chose to insert all the tables that seem miscellaneous and play a supportive role. Also, the limited size of these data is another reason they are suitable for this type of Database. Its in-memory architecture ensures high performance, especially when working with smaller datasets that can fit comfortably in memory.

B. Data Importation

MySQL:

To populate MySQL with data, we followed a systematic approach that included configuring the database with a predefined schema, transferring the data files produced by the TPC-DS tool, and utilizing MySQL commands to import the data. Below is a summary of the process:

Database and Schema Creation: We began by creating the MySQL database using a predefined schema based on our data partitioning method. The schema included all necessary tables, columns and data types required. The schema was imported using SQL scripts:

```
CREATE TABLE warehouse (
    w_warehouse_sk INT NOT NULL,
    w_warehouse_id CHAR(16) NOT NULL,
    w_warehouse_name VARCHAR(20),
```

Figure 8: MYSQL table creation

Generating Data Files: The TPC-DS tool was used to generate data files in .dat format, structured in a specific way. These files store table data in a delimited format, with fields typically separated by a pipe (|) character.

Transfer .dat files to the MySQL server's data directory: The generated .dat files were transferred to the MySQL server's data directory to ensure accessibility for database operations. This is particularly important for commands like LOAD DATA INFILE, which often require files to be located within MySQL's permitted directories.

Loading Data into MySQL: After setting up the database schema, we loaded the data into MySQL tables using the LOAD DATA INFILE command. The following example illustrates the process:

```
LOAD DATA INFILE '/var/lib/mysql-files/warehouse.dat'
INTO TABLE warehouse
FIELDS TERMINATED BY '|'
LINES TERMINATED BY '\n'
(@w_warehouse_sk, @w_warehouse_id, ...
SET
    w_warehouse_sk = IF(@w_warehouse_sk = '', NULL, @w_warehouse_sk),
    w_warehouse_id = IF(@w_warehouse_id = '', NULL, @w_warehouse_id),
    ...
```

Figure 9: MYSQL table population

- Fields terminated by '|' indicates that pipe (|) is used as the delimiter.

- Lines terminated by '\n' ensures proper handling of line breaks.
- SET statements help manage empty values, converting them to NULL where necessary.

Verification Process: Once the data was imported, we conducted verification by executing queries to confirm completeness and accuracy of the inserted records. This step ensured data integrity before proceeding with further operations. We started by creating the database and then creating a collection for each table.

MongoDB:

The fact that MongoDB handles semi-structured data well makes it suitable for importing the larger and more complex tables from the TPC-DS Schema. We started by creating the database and then creating a collection for each table.

To ensure accurate data importation we run custom Python scripts using the pymongo library for each table. The script imports data from a .dat file into a MongoDB collection by following a structured process. First, it establishes a connection to a local MongoDB instance and selects a specific database and collection. The script then defines a data type inference function to convert values to appropriate types (integers, floats, booleans, dates, or strings). Each line from the .dat file is parsed and processed , converting empty fields to None and mapping the extracted values to a dictionary (MongoDB document).

Finally, it reads the file line by line, processes each record, and inserts it into MongoDB. This ensures that the data is properly structured before being stored in the database. To manage the large scale of the data and improve performance, we inserted data in batches of 10,000 records.

Once data importation was complete, we verified the process was complete by checking the number of documents in each collection against the row count in the original .dat file.

In Memory:

Populating the in-memory database was a straightforward process. We created a Bash script to load data, ensuring it was readily available for fast queries. The script was then executed using the command:

./presto-290//load_memory_tables.sh

This script uses the Presto CLI to execute SQL commands. It connects to the Presto server, iterates over a list of tables and creates memory tables by copying data from the schema tpcds.sf10 into the memory catalog.

VI. QUERY EXECUTION

The TPC-DS code package includes a set of template queries designed for benchmarking. We utilize these pre-built SQL queries, making minor adjustments to ensure compatibility with Presto's catalog and schema naming conventions, so they can be used in our experiment. These templates are designed to replicate complex business analytics, utilizing various SQL operations such as JOIN, ORDER BY, WHERE, GROUP BY, and aggregations. By executing these queries on our large dataset, we can analyze Presto's performance and execution plans. For our study, we select 10 key queries from the TPC-DS templates, modifying them as necessary to resolve naming and compatibility issues. Additionally, we create several simple queries with SELECT and WHERE clauses to help clarify and interpret the results.

In order to ensure compatibility with Presto, we had to change the table names because Presto uses specific catalog and schema naming conventions that differ from those used in the original queries. For example, a table named store_sales in the original dataset might be referenced as store_sales in MySQL, but in Presto, it could be in the mongodb.mongodb_presto.store_sales catalog or mysql.prestodb.store_sales catalog, depending on the data source. To automate this process, we used a script that updates the table names in the SQL queries to match Presto's catalog format. This script mapped the original table names to their corresponding Presto catalog and schema names, ensuring that Presto could correctly interpret and access the tables across different data sources like MongoDB, MySQL, and in-memory databases.

A. The Queries

- Q1: This query finds Tennessee customers who returned more items than 120% of their store's average in 2000 by aggregating return fees from MongoDB, filtering by year using MySQL, and joining with in-memory store and customer data to retrieve the top 100 customers.
- Q9: Categorizes store sales into five quantity-based buckets and calculates either the average discount amount or net profit for each, depending on whether sales exceed a predefined threshold. It checks sales volume

in quantity ranges (e.g., 1-20, 21-40) and conditionally selects the relevant metric. The final values are returned as bucket1 to bucket5, filtered by a specific reason code.

- Q12: Retrieves sales revenue and revenue ratio for items in selected categories (Jewelry, Sports, Books), joining the web_sales, item, and date_dim tables.
- Q15: Calculates the total sales price for customers from specific zip codes, states, or with sales over $500, joining the catalog_sales, customer, customer_address, and date_dim tables. It filters the data for the second quarter of the year 2000 and groups the results by zip code, ordering them and limiting to 100 records.
- Q16: Calculates the order count, total shipping cost, and total net profit for orders shipped to Illinois addresses within a 60-day period starting from February 1, 1999. It applies specific filters: the orders must come from call centers located in Williamson County, exist in multiple warehouses (using the EXISTS clause), and not have corresponding entries in the catalog_returns table (using the NOT EXISTS clause). The results are ordered by the order count and limited to 100 records.
- Q18: Calculates the average of sales and demographic metrics for items sold in 2001. It filters data for male customers with a college education and specific birth months. The query also selects items sold in specific U.S. states and groups the results by item ID, country, state, and county using ROLLUP.
- Q22: Calculates the average quantity on hand (QOH) for products in the inventory over a specified 12-month period. It filters the data by item and date, using a ROLLUP to group the results by product name, brand, class, and category. The results are ordered by QOH, product name, brand, class, and category, and limited to 100 records.
- Q43: Calculates the total sales for each store, broken down by day of the week, for the year 1998. It sums the sales prices for each day (Sunday through Saturday) and groups the results by store name and ID. The sales data is filtered by store GMT offset and the year.
- Q52: Calculates the total sales price by brand for the month of December 1998. It groups the data by year, brand name, and brand ID, filtering for items managed by manager ID 1. The results are ordered by year, sales price (in descending order), and brand ID.
- Q80: his query calculates total sales, returns, and profit across store, catalog, and web sales channels for a 30-day period, filtering for high-priced items without TV promotions. It aggregates results for each channel, merges them, applies rollup grouping, and returns the top 100 results.
- Q90: Calculates the ratio of the number of web sales made during a specific hour in the morning (6-7 AM) to the number of web sales made during a specific hour in the afternoon (2-3 PM). It filters for sales where the household demographic count is 8 and the webpage character count is between 5000 and 5200. The results are ordered by the calculated ratio.

The query used to evaluate the overall performance of a query is:

```
SELECT
    node_id,
    MAX("end") - MIN(start) AS total_wall_time,
    SUM(split_cpu_time_ms) / 1000 AS total_cpu_time_sec,
    SUM(split_blocked_time_ms) / 1000 AS total_blocked_time_sec,
    SUM(raw_input_rows) AS total_rows,
    SUM(raw_input_bytes) / 1024 / 1024 AS total_data_MB,
    SUM(splits) AS splits
FROM system.runtime.tasks
WHERE query_id = 'QUERY ID'
GROUP BY node_id;
```

This query retrieves detailed execution statistics for a specific query from Presto's system.runtime.tasks table, grouped by worker node (node_id). It helps analyze how work is distributed across nodes by calculating how much data each node processed, the blocked time, the total cpu time and wall time.

TABLE IV

TABLES USED FOR QUERIES

	QUERY										
	1	9	12	15	16	18	22	43	52	80	90
store_sales		✓						✓	✓	✓	
store_returns	✓									✓	
catalog_sales				✓	✓	✓				✓	
catalog_returns										✓	
web_sales			✓							✓	✓
web_returns										✓	
inventory							✓				
store	✓							✓		✓	
call_center					✓						
catalog_page										✓	
web_site										✓	
web_page											✓
warehouse											
customer	✓			✓		✓					
customer_address				✓	✓	✓					
customer_demographics						✓					
date_dim	✓		✓	✓	✓	✓	✓	✓	✓	✓	
household_demographics											✓
item			✓			✓	✓		✓	✓	
income_band											
promotion										✓	
reason		✓									
ship_mode											
time_dim											✓

VII. THE RESULTS

A. The Raw Data

TABLE V

3 Workers							
Query	Total Time(s)	CPU Time(s)	Blocked Time(s)	Stages	Splits	Input Rows	Input Data(MB)
1	42.3	25	30671	14	957	14519027	507
9	259.2	136	54916	17	315	82516402	1179
12	59.9	29	23733	8	500	16627371	780
15	159.4	59	15279	9	571	45784814	1321
16	171.8	119	91631	11	765	60764709	2406
18	197.8	93	135407	15	1028	37893422	2538
22	872.9	458	265093	7	404	294772709	3107
43	327.5	82	115215	7	415	63046296	1779
52	162.9	73	57191	7	404	58541262	1626
80	338.3	402	457539	36	2682	16386125 7	9314
90	123.1	42	76817	15	966	30544702	850

TABLE VI

2 Workers							
Query	Total Time(s)	CPU Time(s)	Blocked Time(s)	Stages	Splits	Input Rows	Input Data(MB)
1	45	29	21537	14	645	14515725	508
9	341.8	142	65382	17	311	82516402	1179
12	76.5	26	20747	8	340	16626700	780
15	148.8	58	40363	9	387	45784258	1323
16	155.6	126	56115	11	517	60764693	2407
18	190.9	106	86952	15	692	38115525	2559
22	428.8	426	88897	7	276	29743506 9	3335
43	306.4	84	73235	7	283	63046237	1780
52	195.9	95	46757	7	276	58541953	1626
80	462.4	357	380219	36	1798	16385994 0	9315
90	94.6	41	32588	15	662	30544670	851

TABLE VII

1 Worker							
Query	Total Time(s)	CPU Time(s)	Blocked Time(s)	Stages	Splits	Input Rows	Input Data(MB)
1	44	30	10383	14	333	14509849	508
9	296.6	152	76888	17	343.8	82516402	1180
12	58.2	28	8353	8	78.5	16624016	780
15	93.1	60	13357	9	150.8	45783904	1324
16	137.4	117	26231	11	157.6	60764677	2408
18	180.1	102	42968	15	192.9	38113117	2560
22	296.7	385	32955	7	430.8	30159992 5	3693
43	196.7	92	25060	7	151	63046326	1783
52	190.1	88	24371	7	148	58541064	1629
80	341.5	350	152445	36	464.4	16385769 9	9319

90	64	44	1531 2	15	96.6	3054 4638	852

B. Visual Representation

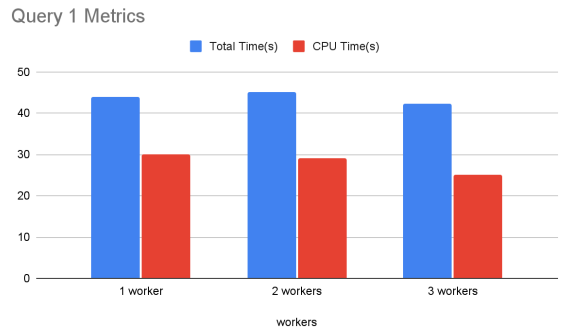

Figure 10: Query 1 Metrics

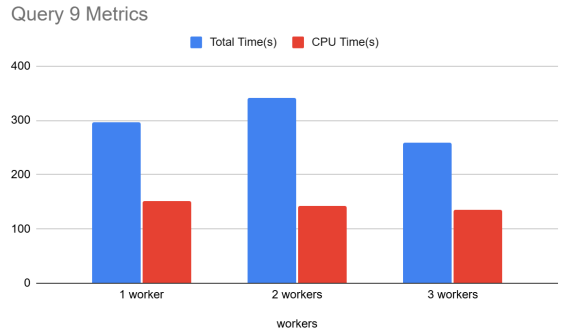

Figure 11: Query 9 Metrics

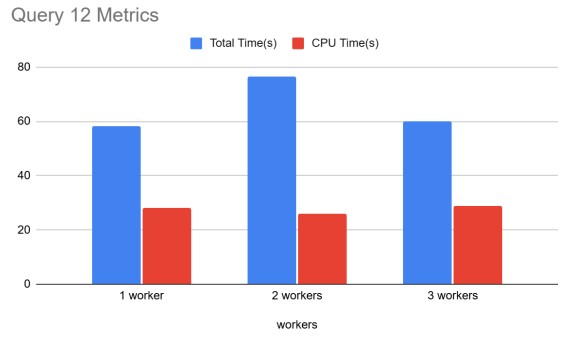

Figure 12: Query 12 Metrics

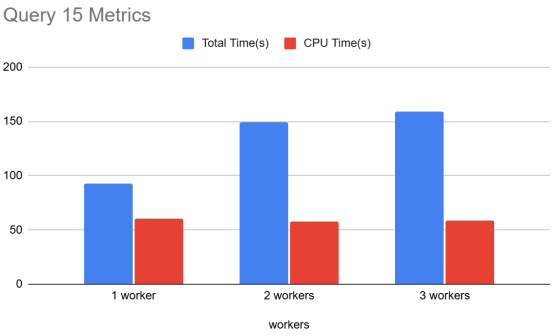

Figure 13 Query 15 Metrics

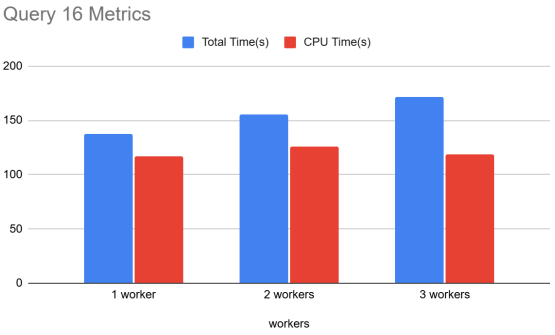

Figure 14: Query 16 Metrics

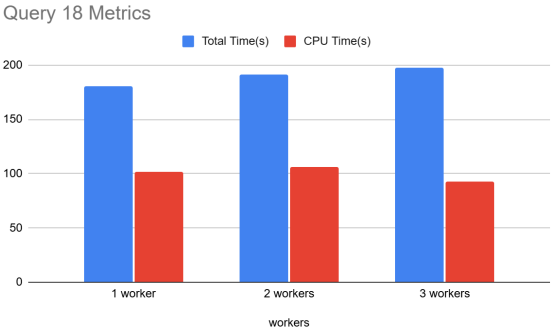

Figure 15: Query 18 Metrics

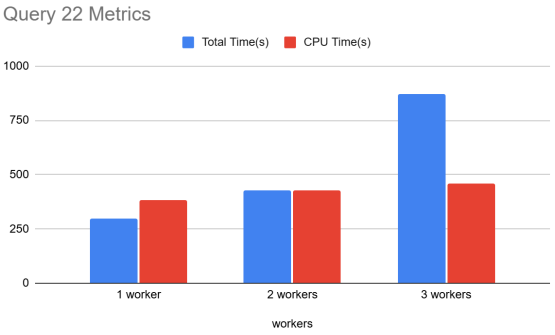

Figure 16: Query 22 Metrics

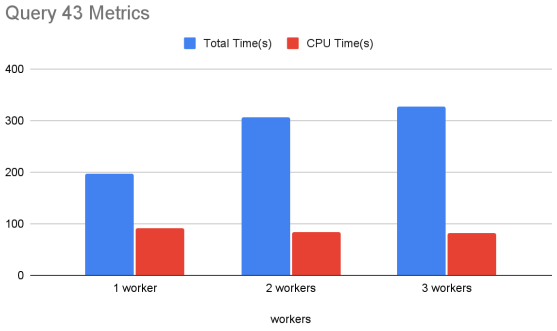

Figure 17: Query 43 Metrics

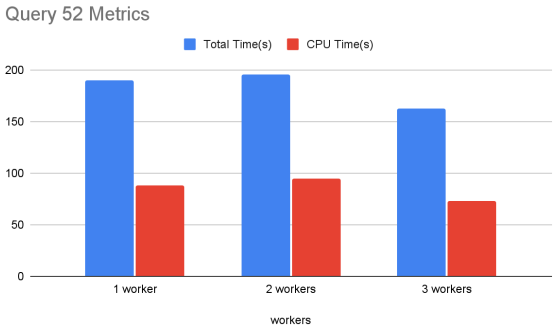

Figure 18: Query 52 Metrics

Figure 19: Query 80 Metrics

Figure 20: Query 90 Metrics

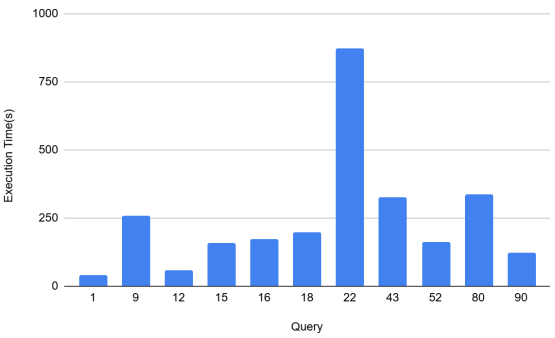

Figure 21: Execution Time of Queries with 3 Workers

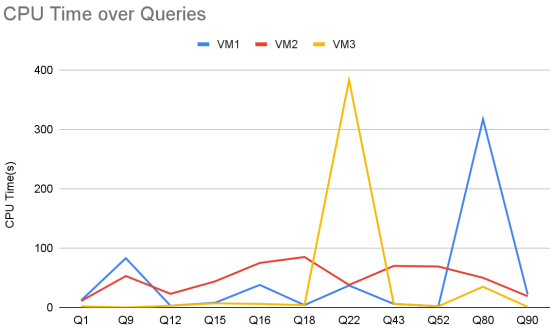

Figure 22: CPU Time Distribution Across VMs

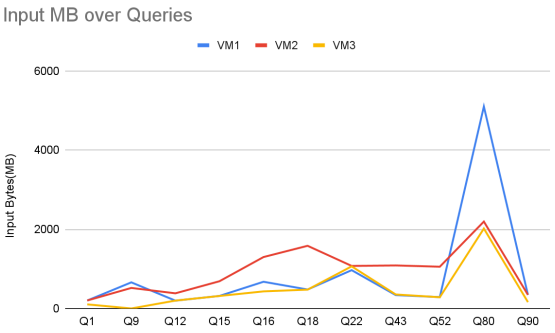

Figure 23: Input Data Distribution Across VMs

TABLE VIII

VM	*Total CPU Time(s)*
VM1	*532*
VM2	*537*
VM3	*449*

TABLE IX

VM	*Total Data (MB)*
VM1	*9567*
VM2	*10431*
VM3	*5409*

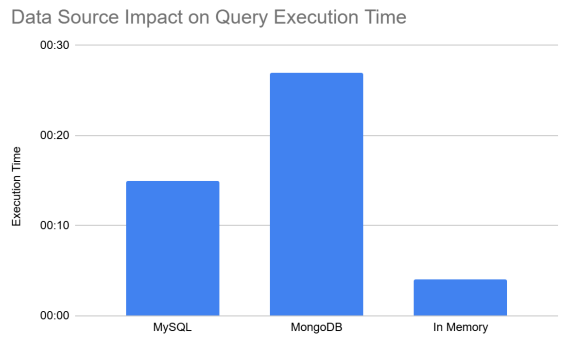

Figure 24: Data Source Impact on Query Execution Time for Query 1

C. Final Remarks

- **Execution Time:** In general, the total execution time from 2 to 3 workers improved. Increasing the number of workers helps distribute the workload better and decrease the overall time to execute a query. However when using just 1 worker sometimes we noticed that the total execution time was even smaller than 3 active workers. That suggests that when using 1 worker there is no communication overhead and execution happens locally. With 3 workers presto needs to coordinate work, which adds overhead due to task scheduling, data shuffling and result merging.
- **CPU Time:** In general CPU time seemed to decrease with the increase of workers. For example in query 16 which took 126 seconds with 2 workers the time reduced to 119 workers when adding the 3rd one.
- **Parallelism and Split: Parallelism and Split:** When executing a query with more workers we noticed a significant increase in splits. This is a positive result since increase in splits means that the resources are better used and the query execution becomes faster. However splits added more time overhead as a result execution time was negatively affected in some cases and in many examples the total execution time was greater when the number of workers was increasing.
- When evaluating Figure 21: '**Execution Time of Queries with 3 Workers**' we can visibly notice that query 22 took significantly more time to execute than other queries. It scanned a massive inventory table, performed expensive ROLLUP aggregations, sorted large datasets, and pulled raw data from MySQL instead of pushing down calculations. Other queries were faster because they used Presto's memory tables or MongoDB, avoided excessive sorting, or had better filtering.
- From Figure 22: '**CPU Time Distribution Across VMs**' and table VIII we can observe that VM1 and VM2 had the highest total CPU usage (532s and 537s), meaning they were assigned slightly more work. VM3 had the lowest CPU usage (449s), indicating it was either underutilized or skipped for some queries. Lastly VM2 had balanced usage but took on heavy loads for certain queries (Q16, Q18, Q43, Q52).
- From Figure 23: '**Input Data Distribution Across VMs**' we can determine Presto prioritizes data locality since queries run on the VM where data is stored. Large queries (Q80, Q52) show extreme skew, meaning Presto's scheduler is not distributing splits evenly. Balanced queries (Q12, Q15, Q43) indicate proper parallelization, meaning evenly distributed data results in better performance. Alongside Table IX it is safe to assume the data distribution was not fairly organized since VM3 took the least load with a significant difference from the other workers. Presto's scheduling could be improved by increasing parallelism and rebalancing data storage.
- In order to compare **query execution times across our data sources**, we ran Query 1 using only one data source at a time. The results were as expected, with the in-memory data source showing a significant performance advantage over the others. In-memory was faster because it stores data directly in RAM, offering extremely low-latency access compared to disk-based systems. In contrast, both MongoDB and MySQL rely on storage systems that involve disk I/O and, in many cases, network communication, which adds latency and overhead to query processing. Evaluating these results with different queries was not

feasible, as most queries required more space than was available.

VIII. CONCLUSION

Through our experiments, we evaluated Presto's performance, scalability, and built-in optimizer across different data partitioning methods. Our results demonstrated that Presto effectively leverages parallel processing, significantly improving query performance—especially when using an optimized partitioning strategy. Presto proves to be a powerful tool for querying heterogeneous data sources, successfully mitigating bottlenecks from underlying databases and, in some cases, even outperforming the fastest one due to its efficient distributed execution.

However, our findings also highlight that better configuration of Presto's workload distribution could have further optimized query execution. More balanced data distribution and improved scheduling strategies would have allowed for a more even workload across nodes, preventing certain workers from becoming overloaded.

Future work could involve scaling Presto to include additional nodes and handling even larger datasets, while also refining query execution strategies to ensure better workload balancing. Additionally, further analysis could be conducted to compare Presto's performance across different database architectures and partitioning schemes, assessing how configuration adjustments influence scalability and efficiency.

REFERENCES

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

[2] Michael Armbrust, Tathagata Das, Liwen Sun, Burak Yavuz, Shixiong Zhu, Mukul Murthy, Joseph Torres, Herman van Hovell, Adrian Ionescu, Alicja Łuszczak, Michał Switakowski, Michał Szafra ´ nski, Xiao Li, ´ Takuya Ueshin, Mostafa Mokhtar, Peter Boncz, Ali Ghodsi, Sameer Paranjpye, Pieter Senster, Reynold Xin, and Matei Zaharia. Delta lake: High-performance ACID table storage over cloud object stores. Proceedings of the VLDB Endowment, 13(12):3411–3424, August 2020.

[3] Duca, Angelica Lo, Tim Meehan, Vivek Bharathan, and Ying Su. Learning and Operating Presto: Fast, Reliable SQL for Data Analytics and Lakehouses. " O'Reilly Media, Inc.", 2023.

[4] Győrödi, Cornelia, Robert Győrödi, George Pecherle, and Andrada Olah. "A comparative study: MongoDB vs. MySQL." In 2015 13th international conference on engineering of modern electric systems (EMES), pp. 1-6. IEEE, 2015.

[5] Győrödi, Cornelia, Robert Győrödi, George Pecherle, and Andrada Olah. "A comparative study: MongoDB vs. MySQL." In 2015 13th international conference on engineering of modern electric systems (EMES), pp. 1-6. IEEE, 2015.

[6] Kabakus, Abdullah Talha, and Resul Kara. "A performance evaluation of in-memory databases." Journal of King Saud University-Computer and Information Sciences 29, no. 4 (2017): 520-525.

[7] Matt Fuller, Manfred Moser, and Martin Traverso. Trino: the definitive guide: SQL at any scale, on any storage, in any environment. O'Reilly Media, Sebastopol, first edition edition, 2021.

[8] Victor Giannakouris, Nikolaos Papailiou, Dimitrios Tsoumakos, and Nectarios Koziris. MuSQLE: Distributed SQL query execution over multiple engine environments. In 2016 IEEE International Conference on Big Data (Big Data), pages 452–461, December 2016.

[9] Jonathan Milton and Payman Zarkesh-Ha. Impacts of Topology and Bandwidth on Distributed Shared Memory Systems. Computers, 12(4):86, April 2023.

[10] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. Presto: SQL on Everything. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 1802–1813, April 2019.

[11] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. Presto: SQL on Everything. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 1802–1813, April 2019.