Distributed Execution of SQL Queries over Trino

Alexandros Ionitsa
Dept. of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
el19193@mail.ntua.gr

Emmanouil Emmanouilidis
Dept. of Electrical & Computer Engineering
National Technical University of Athens
Athens, Greece
el19435@mail.ntua.gr

***Abstract*—The explosion of data in our world has made analyzing large datasets—commonly known as big data—a crucial factor for competition and productivity growth. This exponential growth has necessitated innovative solutions for efficient data processing. Trino, formerly known as PrestoDB [1], [2], emerges as a pivotal tool designed for distributed SQL queries across large and diverse data sets. In this study, we assess the performance of Trino in various scenarios, such as different data distribution strategies, worker configurations, and query complexities. For this project, we utilize three prominent databases—PostgreSQL, Cassandra, and Redis—as integral components of our evaluation framework and execute queries against them. Our methodology involves systematically posing queries under different scenarios, adjusting data distribution strategies and the number of workers. We measure performance metrics, including query latency and optimizer plans, offering valuable insights into how Trino behaves under diverse conditions. This research contributes to understanding Trino's efficiency and optimization strategies. The findings provide practical recommendations for optimizing data distribution strategies.**

***Index Terms*—Big Data, Trino, Distributed SQL Queries**

I. Introduction

With the term "Big Data" we refer to data sets whose scale, diversity and complexity require new architecture, techniques, algorithms and analytics to manage them and extract value from them. Big data has transformed the way organizations handle information, providing unprecedented amounts of context and insights. However, big data makes queries slow and is very difficult and expensive to manage. So, this rapid evolution of big data has forced companies and organizations to put a lot of effort on developing new specialised tools designed to manage and analyze those large data sets in an efficient manner. In response to these challenges, new cutting-edge solutions emerged, including Trino (formerly known as PrestoDB).

Trino [3], is an open-source distributed SQL query engine designed to query large data sets distributed over one or more heterogenous data sources. Among others, Trino has the ability to run federated queries that query tables in different data sources such as MySQL, PostgreSQL, Cassandra, Kafka, Redis and many more. Trino, and its ancestor, PrestoDB, have draught a lot of attention and have now been widely embraced by a high number of companies, such as Uber [4], Twitter [5], [6] and Pinterest [7]. Therefore, it is significant to evaluate its performance under different scenarios and circumstances.

In this research, we benchmark the performance of Trino across different types of queries, data locations and underlying storage technologies. In more detail, we establish a Trino cluster consisting of four nodes, a coordinator and three workers, where each of them is equipped with a different storage system: PostgreSQL on worker1, Cassandra on worker2 and Redis on worker3. The diversity among these three database management systems (DBMS) helps us cover three different data models: Relational, Wide-Column and Key-value respectively providing a more versatile data and query management in our cluster. After, the installation and set up of those three specific stores, we generate and load data using the well-known TPC-DS [8] benchmark which is also utilized for query generation. The size of the data is large enough so that it won't fit in memory, in the order of several GB (milions of records). The queries imposed to our dataset have a large variety from simple queries (selects) to complex ones (multiple joins and aggregations) while targeting all the dataset tables. After generating and loading the data, we pose the generated queries into each of the data sources and we come up with some initial measures. Based on those measures we devise different data distribution strategies and we measure the performance (query latency, optimizer plan) over different number of workers and different data distribution plans. Our final goal is to identify how the distributed execution engine and optimization works under a varying amount of data, data distributed in different engines and with queries of varying difficulty.

This project is backed by a GitHub repository, that contains scripts and howtos on how to replicate our Trino cluster, generate and load the data and generate and perform the queries. Additionally, the repository includes the results of our measurements, along with a detailed analysis to provide a thorough understanding of the outcomes.

II. Related Work

The need to efficiently process large-scale datasets has driven the development of distributed SQL query engines, which can handle the challenges posed by diverse data sources and computational environments. Numerous studies have explored the performance, scalability, and optimization strategies of these systems.

Cardas et al. [9] address a gap in research on the performance of SQL-on-Hadoop systems when deployed in con-

tainerized environments using Kubernetes. Their study evaluates four widely-used SQL platforms—Apache Drill, Apache Hive, Apache Spark SQL, and Trino—deployed on a Hadoop cluster managed by Kubernetes. Using the TPC-H benchmark, they found that Trino outperformed the other systems in most query scenarios, ranking highest in performance, followed by Apache Drill, Apache Spark SQL, and Apache Hive. This research highlights the advantages of Kubernetes in managing distributed SQL workloads, emphasizing Trino's superior query execution times compared to other SQL engines in similar environments. These findings are particularly relevant to our work, as we also evaluate Trino's performance, although our focus is on different data distribution strategies and worker configurations rather than the deployment infrastructure.

Similarly, the MuSQLE framework [10] explores distributed SQL query execution across multiple engine environments, aiming to optimize performance by leveraging the strengths of different SQL engines. The study presents a system that dynamically selects the most appropriate execution engine based on query characteristics and data distribution, thereby enhancing overall query performance and resource utilization. By comparing various execution strategies, MuSQLE provides insights into how heterogeneous SQL engines can be orchestrated to handle complex queries efficiently. This approach complements our research by offering alternative strategies for optimizing distributed SQL queries, particularly in scenarios involving diverse data sources like PostgreSQL, Cassandra, and Redis.

III. Source Code

As mentioned before, the project is backed by a GitHub repository with scripts and howtos which are accessible through the following link:

GitHub Repository

IV. Technology Stack

A. Overview

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

B. Trino

1) ***Overview****:* Trino, formerly known as PrestoDB, is an open-source distributed SQL query engine, build in Java. It is designed to query large amounts of data distributed over one or more heterogeneous data sources, using distributed queries. It was firstly designed as an alternative to tools that query Hadoop Distributed File System (HDFS) using pipelines of MapReduce jobs such as Hive or Pig. However, Trino has been extended to operate over different kinds of data sources, including traditional relational databases such as PostgreSQL and other data sources such as Cassandra. Moreover, it excels in executing federated queries that span tables in different data sources like MySQL, PostgreSQL, Cassandra, Redis and MongoDB. Furthermore, Trino is also designed to handle data warehousing and analytics tasks and its functionalitites extend to data analysis, the aggregation of huge data sets and the generation of reports.

2) ***Nodes****:* As a distributed query engine, Trino processes data in parallel across multiple servers and it runs on a cluster of servers that contains two types of nodes, a coordinator and a worker [11] . Users interact with the coordinator using their SQL query tool, and the coordinator collaborates with workers to access connected data sources. The configuration for this access is stored in catalogs, where parameters like host, port and passwords are defined. Acting as the orchestrator, the coordinator distributes workloads in parallel across all workers. Each worker node runs Trino in a seperate JVM instance and processing is further parallelized through threads.

The Trino coordinator is responsible for parsing and optimizing statements, planning queries and managing worker nodes and it is the only node to which clients can submit statements for execution. It maintains oversight of worker activity, coordinates query execution and translates the logical query models into connected tasks. Additionally, coordinator communicates with clients and workers through a REST API and can be configured to also serve as a worker.

On the other hand, Trino workers are servers responsible for executing tasks and processing data. They fetch data from connectors and exchange intermediate data with each other. The coordinator is responsible for fetching results from workers and returning the final reuslts to the client. Like coordinator, workers use a REST API with which they communicate to each other.

3) ***Components****:* Accessing diverse data sources, executing queries, and delivering results to clients in a distributed manner poses a formidable challenge. Trino addresses this challenge through the incorporation of various components, with connectors, catalogs, schema, and tables standing out as the most significant and critical among them.

Connectors play a crucial role as they are plugins that enable communication with different data sources and storage systems. Trino supports various connectors for traditional SQL databases like MySQL, PostgreSQL, Oracle, for NoSQL databases like MongoDB and Cassandra and for modern data lakes like Hive, Iceberg and Delta Lake. As a result, users can query data from a large variety of different data sources.

Schemas are a way to organize tables. Together, a catalog and schema define a set of tables that can be queried. When accessing Hive or a relational database such as MySQL with Trino, a schema translates to the same concept in the target database.

A table is a set of unordered rows, which are organized into named columns with types. This is the same as in any relational database. The mapping from source data to tables is defined by the connector.

Catalogs in Trino, are logical containers that hold metadata about the available sources, including schemas, tables and connectors. Trino can connect to multiple catalogs, each representing a distinct set of data sources and their associated metadata. Users can switch between catalogs in their queries and access data from different environments without the need to change their SQL statements.

By combining the components mentioned before, Trino gains the ability to represent table structures and query data in SQL, from data sources that do not support the concept of tables or even SQL (such as Redis).

*4) **Architecture**:* Summarizing the above, a Trino cluster consists of a single node and one or more worker nodes. The coordinator is responsible for admitting, parsing, planning and optimizing queries as well as query orchestration. Worker nodes are responsible for query processing. Figure 1 illustrates a simplified view of Trino architechture:

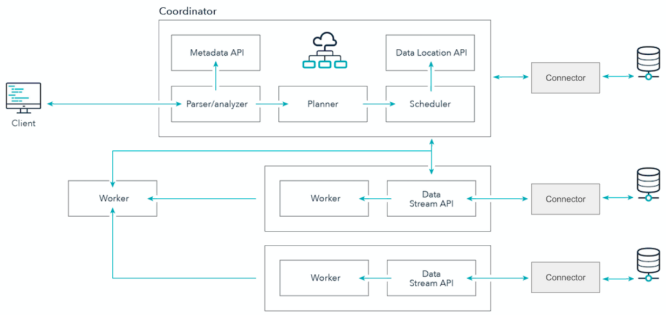

Fig. 1. Trino Architechture

The client sends an HTTP request containing a SQL statement to the coordinator. The coordinator processes the request by parsing and analyzing the SQL text, creating and optimizing distributed execution plan. The coordinator distributes this plan to workers, starts execution of tasks and then begins to enumerate splits, which are sections of a larger data set. Splits are assigned to the tasks responsible for reading this data.

C. PostgreSQL

PostgreSQL, also known as Postgres, is a highly regarded free and open-source relational database management system (RDBMS) that emphasizes extensibility and SQL compliance. The system is designed to handle a diverse range of workloads, from single machines to large-scale data warehouses or web services with numerous concurrent users. PostgreSQL operates seamlessly on major operating systems, including Linux, FreeBSD, OpenBSD, macOS, and Windows. Some key features that make PostgreSQL very interesting and powerful, are the followings:

- **SQL Compliance and Extensibility:** PostgreSQL allows users to define their own data types, operators, functions, and aggregates. This extensibility provides flexibility for developers to customize and extend the database according to their specific requirements.
- **ACID Properties:** The system ensures data reliability through transactions with Atomicity, Consistency, Isolation and Durability (ACID) properties, enhancing data integrity and consistency.
- **Advanced Functionality:** PostgreSQL supports a rich set of features such as automatically updatable views, materialized views, foreign keys providing flexibility and control over database operations.
- **Concurrency Control:** PostgreSQL employs Multi - Version Concurrency Control (MVCC), which allows multiple transactions to occur simultaneously without interfering with each other. This ensures high concurrency and scalability.
- **Advanced Query Optimization:** The query planner and optimizer in PostgreSQL are sophisticated, enabling efficient execution plans for complex queries. It includes features like join optimization, subquery optimization, and parallel query execution.

These features make PostgreSQL the go to selection relational DBMS making it the most suitable selection for a relational database in our cluster.

D. Cassandra

Cassandra is an open - source, distributed, wide-column store, NoSQL database management system, designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure. Some of its main features are listed below:

- **Distributed:** Every node in the cluster has the same role and there is no master. Also there is no single point of failure and data is distributed across the cluster, so each node is responsible for different data.
- **Replication - Fault Tolerance:** Cassandra supports replication across multiple datacenters, providing lower latency for users. Data is automatically replicated to multiple nodes the number of which can be easily configured.
- **Scalability:** Cassandra scales horizontally and is designed in such a way, that allows linear increase in write and read throughput by adding new machines in the cluster.

These features, in addition to its efficiency and easy scalability contribute to Cassandra being a very powerful and widely used NoSQL database and makes it a perfect database to experiment on.

E. Redis

Redis (Remote Dictionary Server) is an open-source in-memory storage, used as a distributed, in-memory key-value database, cache and message broker, with optional durability. Because it holds all data in memory and due to its design, Redis offers low-latency reads and writes, making it particularly suitable for cases that require a cache. Additionally, Redis provides a notable speedup to disk read/write operations and is commonly employed as a cache to a non-volatile database. Moreover, Redis is one of the most popular database systems and it is frequently used in combination with other databases

in order to leverage the strengths of each database for specific use cases. Some key features are listed below:

- **In - Memory Storage:** Redis stores data in-memory, which allows for extremely fast read and write operations.
- **Data Structures:** Redis supports a variety of data structures, including strings, hashes, lists, sets and sorted sets. Each data type comes with its own set of operations.
- **Persistence:** While Redis is an in-memory store, it provides options for persistence. It can be configured to periodically write data to disk, ensuring data durability.
- **Replication:** Redis supports master-slave replication, allowing for the creation of replicas of the master Redis server. This provides high availability and fault tolerance, as the slaves can take over if the master fails.
- **Paritioning:** Redis can be partitioned across multiple nodes, enabling horizontal scaling. This is particularly useful for large datasets and high-throughput scenarios.

Redis is utilized in our project in order to store a subset of the data set and compare its efficiency and speed.

F. TPC-DS Benchmark

As TPC-DS specification [8] mentions: "TPC-DS is a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance. The benchmark provides a representative evaluation of performance as a general purpose decision support system."

In general, TPC-DS is:

- Industry standard benchmark (OLAP / Data Warehouse).
- Implemented for many analytical processing systems - RDBMS, Apache Spark, Apache Flink, etc.
- It provides a wide range of different SQL queries.
- It incorporates the tools to generate input data of different sizes.

In our project, TPC-DS is utilized for data and query generation.

V. Installation and Configuration

In this section we outline the steps that were followed in order to set up our cluster and replicate our working environment. We describe the installation and configuration of Trino, the setup of the data sources on each node, and the process of generating and loading the dataset using the TPC-DS Benchmark. More in-depth details, instructions and commands can be found in our GitHub Repository.

A. Cluster and Virtual Machines

For our project we used four virtual machines (1 coordinator and 3 workers) that were provided by okeanos-knossos [12]. Okeanos-knossos is a GRNET'S cloud service (IaaS) for the Greek Research and Academic Community that gives access to resources such as virtual machines and virtual networks. Each virtual machine has the same following specifications:

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

B. Networking

For networking we were given one public IPv4 address, which was attached to our master node which also behaves as the Trino coordinator. In order to allow the communication between the cluster nodes, we set up a local area network (LAN: 192.168.1.0/24). In more details, 192.168.1.1 was assigned to the coordinator, 192.168.1.2 to worker 1 (PostgreSQL), 192.168.1.3 to worker 2 (Cassandra) and 192.168.1.4 to worker 3 (Redis) as shown in the figure below:

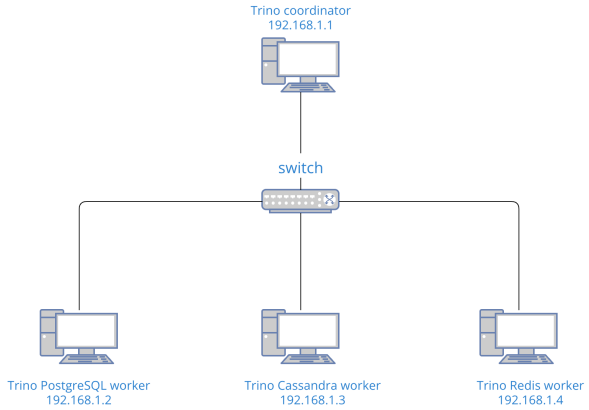

Fig. 2. Cluster's Network Topology

After setting up our machines, we used SSH in order to have access to them and proceeded wtih configurations such as default gateway, firewall rules, hostnames etc. We mention, that okeanos provides Ubuntu Server 16.04 so we had to upgrade to 22.04.

C. Prerequisites

Before we begin, we mention the tools that need to be installed:

- Java 17.03 or later version
- Python 3.10.12 or later

D. Trino Cluster Set Up & Configuration

For Trino cluster set up we followed the "Deploying Trino" guide from Trino's official site. First of all, we downloaded the latest Trino release (435 at the time being). Next, we created the `trino-server-435/etc` directory which contains the necessary configuration files:

- **Node properties:** The node properties file, `node.properties`, contains configuration specific to each node such as the name of the environment and the unique identifier.
- **JVM config:** The JVM config file, `jvm.config`, contains a list of command line options used for launching the Java Virtual Machine.
- **Config properties:** The config properties file, `config.properties`, contains the configuration for the Trino server. It provides information such as

the port of the HTTP server and the type of node (coordinator, worker or both).
- **Log levels:** The optional log levels file, `log.properties`, allows setting the desired log level (INFO, DEBUG etc).

These files, are present in every node. They also contain the `trino-server-435/etc/catalog` directory. As mentioned before, Trino accesses data via connectors, which are mounted in catalogs and the connector provides all of the schemas and tables inside of the catalog. So under this directory, there are several `.properties` file, one for each used data source, that includes information such as port, host and password, that allow Trino nodes to access the different data sources.

In order to be able to query the data located in those databases, we downloaded the Trino CLI executable.

E. Database Installation and Connection with Trino

Following the instructions from the original sites, we install and set up the databases below:

- **PostgreSQL:** We installed PostgreSQL 14 on worker 1.
- **Apache Cassandra:** We installed Cassandra 5.0 on worker 2, following the Debian packages installation guide.
- **Redis:** We installed Redis on worker 3.

To ensure accessibility to every database from each node within the Trino cluster and enable querying of the data via Trino CLI, we configured each database as outlined below:

1) PostgreSQL: To expose PostgreSQL to the cluster network on the LAN, we first modified the `/etc/postgresql/14/main/pg_hba.conf` file, adding the IP addresses of the cluster nodes to allow database access. Next, we updated the `postgres.properties` file with the correct IP address, username, and password for authentication. Finally, we edited the `/etc/postgresql/14/main/postgresql.conf` file, setting the `listen_addresses` parameter to the desired LAN IP (192.168.1.2) to permit connections from that address.

2) Cassandra: For Cassandra, we adjusted parameters like authenticator, authorizer, seeds, listen_address and rpc_address that are located in `/etc/cassandra/cassandra.yaml` as follows:

- **authenticator:** class_name : org.apache.cassandra.auth.PasswordAuthenticator
- **authorizer:** CassandraAuthorizer
- **rpc_address, listen_address and seeds:** 192.168.1.3 (IP address of worker 2 to whom Cassandra is installed)

We also configured `cassandra.properties` accordingly.

3) Redis: In configuring Redis, we made modifications to the `/etc/redis/redis.conf` file, specifying the worker's 3 IP address in the `bind` property to ensure Redis listens to that specific IP. Similar adjustments were made to the `redis.properties` file as mentioned earlier.

It's important to note that Redis lacks support for schema and table definitions. To address this limitation, Trino tackles the issue by requiring users to define the schema for each table using specific table schema definition files (in our case, JSON files). Trino can access these files based on their location, which is declared in the `redis.properties` file.

F. TPC-DS Benchmark installation and data & query generation

We download the TPC-DS benchmark source code and compiled it utilizing the Makefile that is included in the zip file. Next, we used dsdgen executable to generate a data set of 8GB size and dqgen to generate the 99 queries that the benchmark provides.

G. Data loading

In order to load the data into the different data sources, we created proper database schemas in PostgreSQL and Cassandra through `tpcds.sql` file that creates the tables and `tpcds_ri.sql` that specifies foreign key constraints. Since Cassandra and Redis don't support foreign key constraints, they were applied only to PostgreSQL. Our study is focusing only on querying already existing data and there is no intention to insert, delete or update those data. Since data obbey the constraints in PostgreSQL, we can be sure that the same holds for Cassandra and Redis.

In Redis we utilized table schema definition files as mentioned earlier. For that purpose, we created `create_trino_json_table_definitions.sh` script which is responsible for generating those JSON files.

After creating the schema and tables, we created scripts in order to load the data into the different databases. For PostgreSQL and Cassandra we took advantage of the COPY command that both, SQL and CQL provide. For Redis we used python modules and loaded the data using hashes.

We note that since Redis is an in-memory database and certain tables contain millions of records, not all records can fit in memory and it wouldn't be efficient to overload RAM. Consequently, the insertion process focuses on a subset of the data for practical considerations.

In conclusion, we also note, that for our initial experiments, we loaded the complete dataset into both Postgres and Cassandra. Following the initial measurements and theoretical insights, we came up with some early conclusions which helped up devise a better data distribution strategy as explained later.

VI. Process and Methodology

A. The TPC-DS Benchmark

As mentioned in IV-F, the TPC-DS benchmark is a comprehensive tool for evaluating decision support systems, offering a range of SQL queries and the capability to generate data of varying sizes. In addition to its technical features, TPC-DS also simulates a realistic business environment, which is crucial for understanding its applicability in real-world scenarios.

1) Business Model: TPC-DS models industries involved in managing, selling, and distributing products, ranging from food and electronics to furniture, music, and toys. It reflects the business structure of a large retail company with a nationwide presence, incorporating both brick-and-mortar stores as well as catalog and online sales channels. The benchmark simulates key operational areas such as sales, returns, inventory management, and promotions. Here are examples of the business processes represented in TPC-DS:

- 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 carefully crafted to model the sales and returns processes for an organization utilizing three main sales channels: physical stores, catalogs, and the Internet. 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 comprehensive transactional data. Each of these tables is represented by an ER-diagram that details the relationships between the fact tables and their associated dimension tables. Specifically, the schema includes paired fact tables dedicated to tracking product sales and returns across the three sales channels, along with a single fact table for inventory management, particularly for catalog and online sales.

To support these fact tables, the schema incorporates 17 dimension tables that provide contextual details and are shared across all sales channels. These tables form a snowflake schema, where the central fact tables are connected to dimension tables that may themselves be further normalized. The logical design of the TPC-DS schema is detailed, with clearly defined tables and relationships, visualized through high-level diagrams that illustrate the connections in the snowflake schema.

Figures 3 through 9 represent the ER diagrams of each of the seven fact tables, as provided by the TPC-DS Standard Specification [10].

3) Query Templates: The TPC-DS also provides a query suite of 100 queries which we used to benchmark the performance of our different data distribution strategies. According to the TPC-DS specification [10] the queries can be categorized as follows:

- **Reporting Queries:** Queries that answer pre-defined questions.
- **Ad hoc Queries:** Dynamic queries - they answer immediate and specific business questions.
- **Iterative OLAP Queries:** Queries that discover new trends.
- **Data Mining Queries:** Queries that sift through large data, typically joins and aggregations.

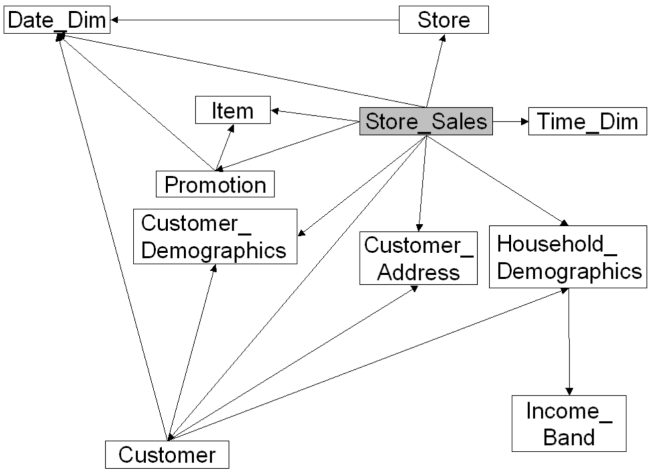

Fig. 3. Store Sales - ER Diagram

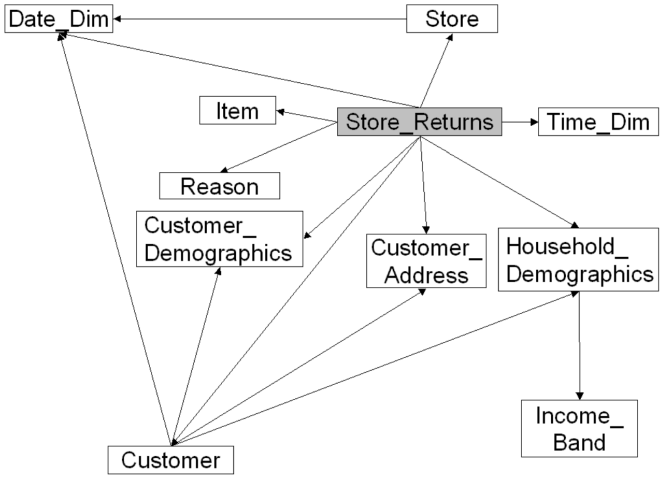

Fig. 4. Store Returns - ER Diagram

B. Data Distribution Strategy

1) No Data Distribution Applied: Initially, we measured the performance without any data distribution to establish a baseline system performance value. It is important to note that we did not test Redis performance on the full data set due to the RAM limitations (V-A) of the virtual machines used in our test system. Instead, we limited Redis to store only the web sales and returns fact tables along with their associated dimension tables, in order to compare query run times for queries involving these (or subsets of these) tables.

Due to the limitations of Redis and the diverse set of queries provided by TPC-DS, we selected a subset of queries and categorized them into the following groups:

- **Group 1:** Queries involving multiple fact tables (e.g., catalog sales and store sales).
- **Group 2:** Queries involving a single fact table.
- **Group 3:** Queries that access tables from the web channel (the only channel that can be fully stored in Redis).

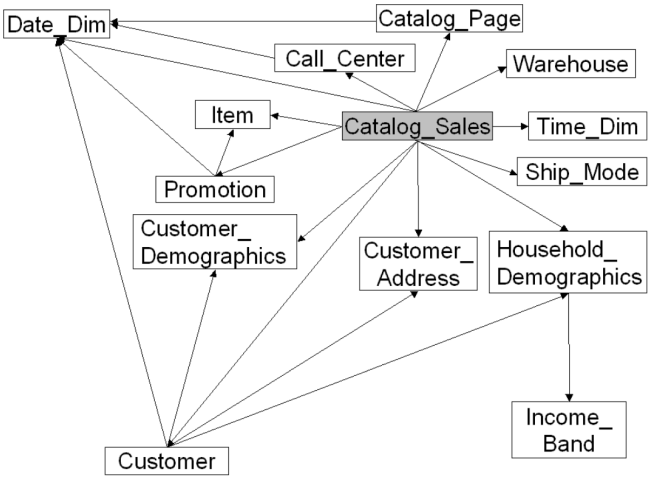

Fig. 5. Catalog Sales - ER Diagram

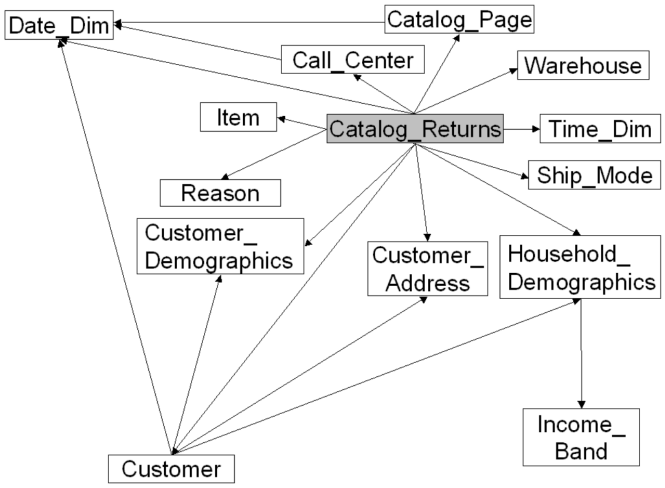

Fig. 6. Catalog Returns - ER Diagram

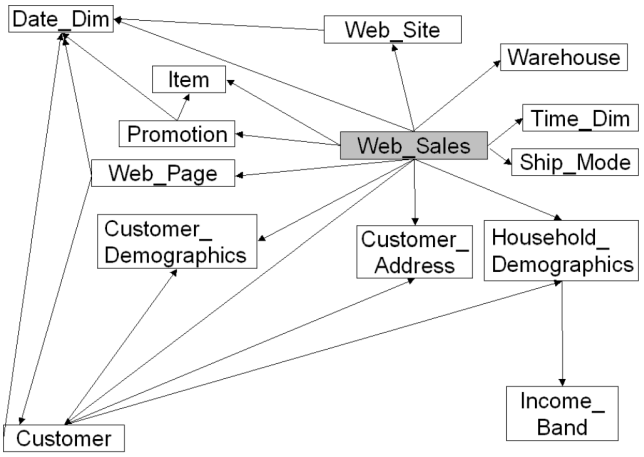

Fig. 7. Web Sales - ER Diagram

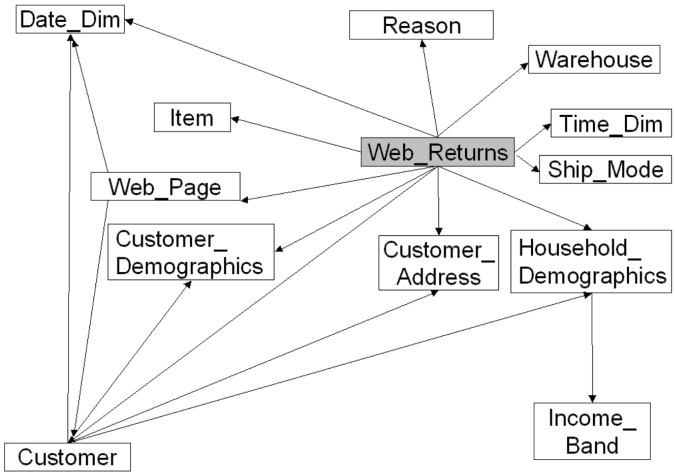

Fig. 8. Web Returns - ER Diagram

Table I shows the categorization of the selected TPC-DS queries into their respective groups.

We note that, for each benchmark and distribution strategy, we measured the performance 10 times to ensure more accurate and reliable results. The full comparison results between PostgreSQL, Cassandra and Redis on the same query group are presented in Section VII-A.

2) Entity-Relationship (ER) Fact Table Distribution with Replicated Dimensions: The performance measurements from the first test without data distribution (VII-A) revealed a clear performance disparity between PostgreSQL, Cassandra, and Redis. PostgreSQL outperformed the others in most queries, while Redis was significantly slower, as expected, due to its poor optimization for large datasets. Cassandra, while generally trailing PostgreSQL, showed better performance in some queries, though there were significant slowdowns in certain outliers.

Running the base benchmark provided us with a better understanding on each database's performance on different queries resulting in a clearer view for our following distribution strategies. The first distribution strategy we tested involves partitioning data based on the entity-relationship (ER) model of the fact tables.

For this strategy, we focused solely on the queries from **group 1**, which involve multiple fact tables. The principles guiding this strategy are as follows:

- Fact tables from the same channel are located in the same database (e.g., store_sales and store_returns are both in the same database).
- Databases are loaded based on the results of the initial test (e.g., larger tables are placed in more performant databases).
- Dimension tables that correspond to a fact table are stored in the same database as the fact table (e.g., if inventory is stored in PostgreSQL, then related tables like warehouse, date_dim, and item are also stored in PostgreSQL).

TABLE I
CATEGORIZATION OF SELECTED TPC-DS QUERIES

Group	Queries
Group 1: Multiple Fact Tables	2, 4, 5, 11, 33, 51, 66, 72, 75, 82, 83, 97
Group 2: Single Fact Table	6, 13, 18, 27, 30, 39, 40, 50, 62, 84, 85, 88, 90, 91, 99
Group 3: Web Channel Tables	30, 62, 90

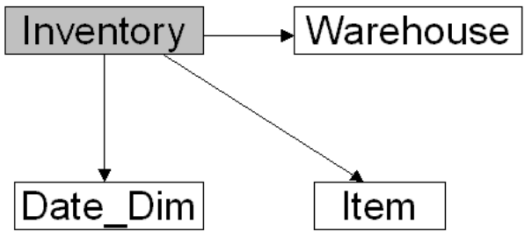

Fig. 9. Inventory - ER Diagram

- If a dimension table is associated with multiple fact tables located in different databases, the dimension table is replicated across all relevant databases (e.g., date_dim, which is related to all fact tables, is stored in all three databases).

The exact distribution selected in this experiment is detailed in II.

TABLE II
DATA DISTRIBUTION BASED ON FACT TABLES ER

Table	PostgreSQL	Cassandra	Redis
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			✓

By following these principles, we aim to achieve two main goals:

1) Equally partition data across PostgreSQL, Cassandra, and Redis while considering the results of our base test, which indicated that PostgreSQL should handle the higher load and Redis the lowest.
2) Exploit data locality by storing fact tables alongside their respective dimension tables, since fact tables are joined with them (according to the ER model), reducing network traffic and optimizing join performance.

This approach balances the data distribution across the three databases while leveraging local joins to enhance performance and reduce data movement.

3) Optimized Fact and Dimension Consolidation Strategy: In the previous strategy based on the entity-relationship (ER) model of the fact tables, we observed a performance improvement over Cassandra and Redis but lower performance compared to PostgreSQL alone. We identified Redis as the primary bottleneck, especially due to its limited ability to handle large datasets efficiently. Notably, the `web_sales` table, the largest fact table stored in Redis, was a major contributor to this bottleneck.

To address this, we decided to move the `web_sales` table from Redis to PostgreSQL. This change aimed to reduce the load on Redis while leveraging PostgreSQL's ability to handle larger fact tables more effectively.

Additionally, we wanted to minimize the replication of dimension tables across multiple databases, which added complexity to query execution. By centralizing frequently joined dimension tables in PostgreSQL, we could take advantage of Trino's pushdown capabilities.

Pushdown optimization [13] in Trino allows specific query operations, such as predicates, aggregation functions, or other operations, to be pushed down to the underlying data source for processing. This results in several key benefits:

- Improved overall query performance.
- Reduced network traffic between Trino and the data source.
- Lower load on remote data sources.

Given that PostgreSQL supports more complex operations, such as joins, better than Cassandra or Redis, we moved key dimension tables like `date_dim`, `item`, and `time_dim` into PostgreSQL. These tables are frequently used in join operations across many queries, so by centralizing them in PostgreSQL, we allow the database to handle most of the join workload internally, minimizing the number of records sent back to Trino for further processing.

By restructuring the data in this way, we aimed to:

- Eliminate the bottleneck caused by Redis.
- Reduce the complexity of managing replicated dimension tables.
- Leverage PostgreSQL's ability to execute joins and other operations locally, thus improving query performance and reducing the data transferred to Trino.

The revised distribution strategy reflects these optimizations, balancing the workload between the databases while keeping PostgreSQL at the core of handling complex operations. Table III depicts the data distribution for this strategy.

TABLE III
OPTIMIZED FACT AND DIMENSION CONSOLIDATION STRATEGY - DATA PARTITION

Table	PostgreSQL	Cassandra	Redis
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			✓

4) Scalability impact on query performance: Up until this section, we discussed the impact of different distribution strategies on the query performance. Fine tuning is required to find the optimal strategy for the data you want to query on and is something that we tested in our system and faced the various difficulties.

In this section though, we focus to the Trino configuration and the impact it has on query run time. We use the distribution used in the previous experiment as seen in Table III as it was the best performing distribution we used. We run the query suite described previously with 1, 2 and 3 workers. The configuration for each worker count is the following:

- **1 Worker:** Only the Trino coordinator was kept active.
- **2 Workers:** The Trino coordinator and the PostgreSQL are active.
- **3 Workers:** All nodes (except Redis node) are active.

Generally, the Redis node we chose to keep it inactive in all experiments as it had conflicting RAM usage, both for the Redis process and the Trino worker process so the system was resulting in many OOM (out-of-memory errors).

VII. RESULTS AND ANALYSIS

In this section we are presenting the results of our experiments that measure the performance of Trino in different settings regarding the data distribution across workers and the configuration of the cluster.

A. Base System Benchmark - No data distribution applied

As aforementioned, in this first experiment our goal is to establish a base benchmark score between the three databases (PostgreSQL, Cassandra and Redis) while executing the same query in each database from the Trino CLI.

In Figures 10 to 12 we showcase the performance of each database in each query group.

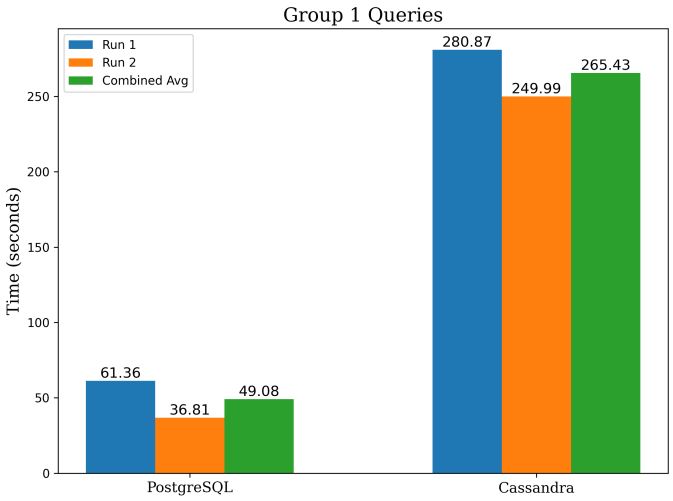

Fig. 10. PostgreSQL vs Cassandra

From Figure 10 we conclude that PostgreSQL has on average a 4.5x speedup comparing to Cassandra on queries that consist of multiple big tables (fact tables) with multiple joins (group1 queries). PostgreSQL was expected to be faster in these type of queries considering that Cassandra do not natively support joins and are entirely handled by Trino.

In Figure 11 we compared PostgreSQL and Cassandra on queries that consisted only one fact table and its dimension tables. The queries in this group generally had more frequent aggregations and less frequent joins, especially between large tables. PostgreSQL in this test was only 1.5x faster on average than Cassandra on the same queries. This changes, though, on the second consecutive run where PostgreSQL queries run 3x times faster due to better query caching and dynamic filtering [14] compared to Cassandra and Trino.

Lastly, Redis queries performed on average 22x slower than PostgreSQL and 5.6x slower than Cassandra 12. This performance penalty is mostly attributed on slower table retrieval from Redis. Redis being a key-store database has no table indexing and is required to use SCAN on all of its records and Trino performs any required filtering on all the initial dataset.

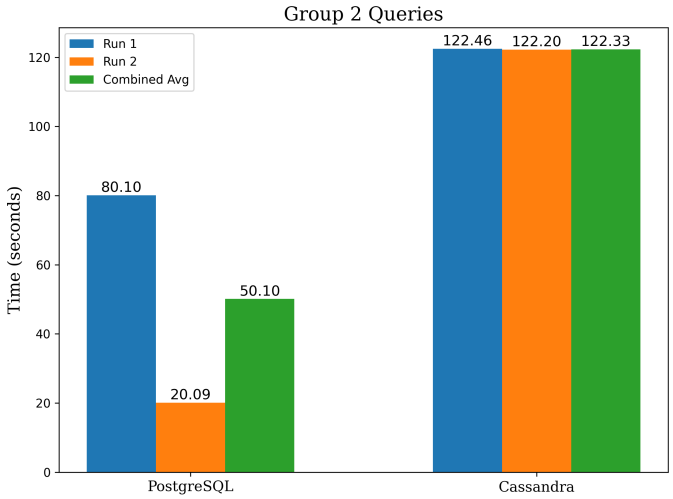

Fig. 11. PostgreSQL vs Cassandra

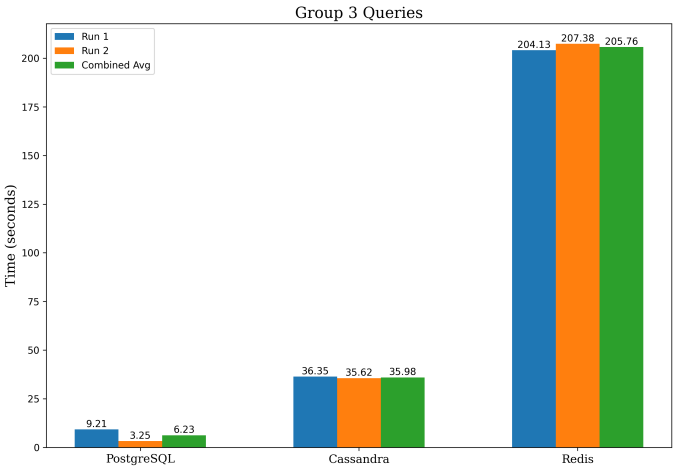

Fig. 12. PostgreSQL vs Cassandra vs Redis

To sum up, our first test showed a clear ranking between the 3 databases, with PostgreSQL performing considerably better than Cassandra with Redis being a distant third.

B. Entity-Relationship (ER) Fact Table Distribution with Replicated Dimension

In Figure 13 we showcase the performance of Trino on group3 queries while executing the queries entirely on PostgreSQL or Cassandra, or while using the distribution strategy specified in VI-B2.

With this experiment we wanted to showcase the performance of the Trino cluster while trying to have an equal distribution of data between each data source. The results of the query runtime showcase that such a distribution choice performs on average 23% faster than Cassandra and 3.1x slower than PostgreSQL on the same set of queries.

Diving deeper on the execution plans of the queries we observed a pattern present on most queries. The system

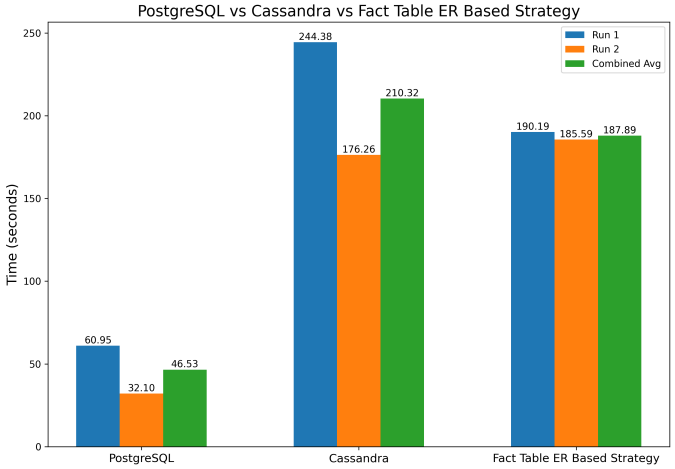

Fig. 13. PostgreSQL vs Cassandra vs Distributed Run

bottleneck was the retrieval rate of data from the Redis node. Also abscent on operations targeting data located in the Redis database was any dynamic filtering that Trino would impose on the original query while in both PostgreSQL and Cassandra dynamic filtering could speed up substantially subquery operations.

C. Optimized Fact and Dimension Consolidation Strategy

In the current experiment, we opted for a data distribution with no data replication and with the PostgreSQL and Cassandra to hold most of the large fact tables. We elected for a significant lower load in Redis as this was the main bottleneck in our first strategy where we loaded all databases at approximately the same capacity. The performance comparison between the two strategies is illustrated in Figure 14.

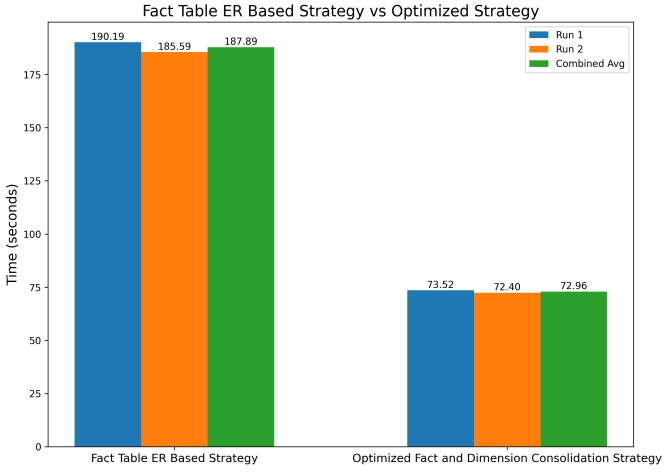

Fig. 14. Distribution Strategy 1 vs Distribution Strategy 2

Figure 14 showcases that a distribution strategy with a lighter load on Redis results in a far better query performance. On average, the current strategy is 2.5x faster than the previ-

ous strategy and only 12% slower than the original run on PostgreSQL only.

Analyzing the execution plans of the queries we again see that the Trino cluster takes a significant time retrieving data from each data store and the networking penalty is affecting negatively the query performance.

We compared also CPU times between runs with distribution present and no distribution active. In PostgreSQL the networking time is far less noticable than on every other test. Generally, queries that could be handled on their entirety by PostgreSQL have less parallelism present and less data retrieval time due to less data being fetched and as a result sent to other workers.

D. Scalability impact on query performance

In Figure 15 we present the performance impact on the query performance based on the amount of Trino nodes used in the cluster.

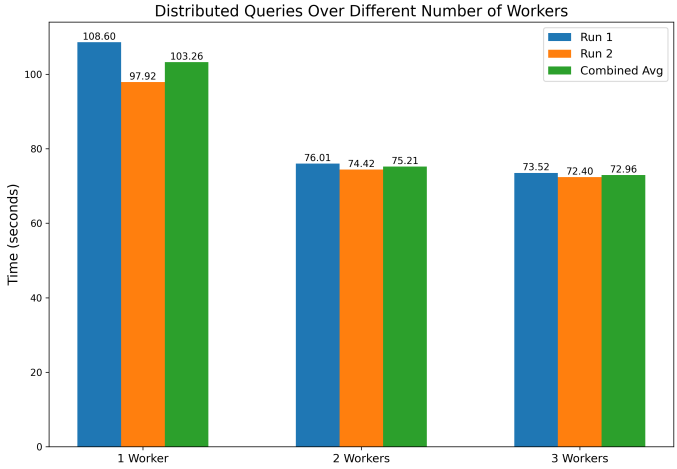

Fig. 15. Node number performance impact on 2nd experiment

As anticipated, increasing the number of nodes led to a noticeable performance improvement. The most significant gain occurred when scaling from 1 to 2 nodes, with an average query run time reduction of 27%. However, further node increases provided only a moderate performance boost, yielding an additional 5% improvement.

Observations showcased that network data transfers were proportional with the number of nodes which was again expected. We also checked the LAN performance using the `iperf3` tool. The results showcased that the network connectivity was sub optimal for such a use case and our test bench was taking a hit from it.

VIII. Conclusion

To summarize, in this paper we measured the Trino performance on distributed query execution of a subset of the TPC-DS benchmarking suite with various distribution strategies and configurations. Our best tests while on average performed marginally slower than a direct run on PostgreSQL they provided us with valuable information. In many use cases, data distribution from various sources is a necessity and not an option. We showcased that Trino can perform on par with a native SQL database while running SQL queries on data sources that do not natively support these kind of operations. Further work could test Trino performance on more powerful clusters with better resources and more advanced distribution strategies. A good staring point could be to use the Redis database as an intermediate query result caching database instead of a direct data source. Nonetheless, Trino can be a powerful and a performant distributed engine in cases where many data sources have to be used simultaneously without having to rely on ETL operations to make the data queryable.

References

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

[2] Z. Luo et al., "From Batch Processing to Real Time Analytics: Running Presto® at Scale," in *2022 IEEE 38th International Conference on Data Engineering (ICDE)*, Kuala Lumpur, Malaysia, 2022, pp. 1598-1609.

[3] Trino. (2024). "Trino — Distributed SQL query engine for big data." [Online]. Available: https://trino.io/

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

[5] C. Tang et al., "Hybrid-cloud SQL federation system at Twitter," in *ECSA (Companion)*, 2021.

[6] C. Tang et al., "Serving hybrid-cloud SQL interactive queries at Twitter," in *European Conference on Software Architecture*, Springer, 2022, pp. 3-21.

[7] "Presto at Pinterest," 2019. [Online]. Available: https://medium.com/pinterest-engineering/presto-at-pinterest-a8bda7515e52

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

[9] C. Cardas, J. F. Aldana-Martín, A. M. Burgueño-Romero, A. J. Nebro, J. M. Mateos, and J. J. Sánchez, "On the performance of SQL scalable systems on Kubernetes: a comparative study," in *Cluster Computing*, vol. 26, no. 3, Jun. 2023.

[10] V. Giannakouris, N. Papailiou, D. Tsoumakos, and N. Koziris, "MuSQLE: Distributed SQL query execution over multiple engine environments," in *2016 IEEE International Conference on Big Data (Big Data)*, Washington, DC, USA, 2016.

[11] A. Houwaiji, "Trino Deep Dive," 2023. [Online]. Available: https://whitestork.me/blog/13/Trino-Deep-Dive

[12] " okeanos-knossos." [Online]. Available: https://okeanos-knossos.grnet.gr/home/

[13] "Pushdown in Trino." [Online]. Available: https://trino.io/docs/current/optimizer/pushdown.html

[14] "An Overview of Caching for PostgreSQL." [Online]. Available: https://severalnines.com/blog/overview-caching-postgresql/