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 his 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.

Identify applicable funding agency here. If none, delete this.

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 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. 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

III. Technology Stack

A. Overview

- **Trino:** Was utilized as a distributed 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 [9] . 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:

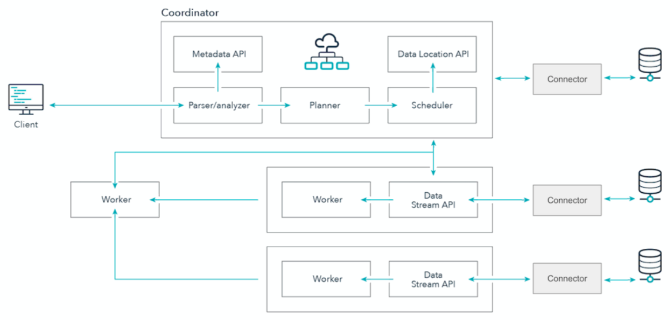

Fig. 1. Trino Architechture

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.
- In incorporates the tools to generate input data of different sizes.

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

IV. 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 [10]. 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:

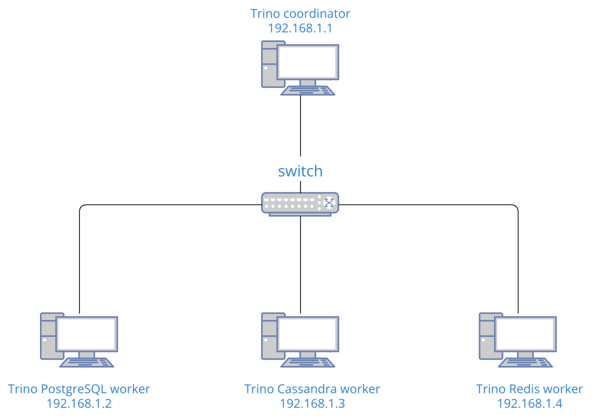

Fig. 2. Cluster's Network Topology

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 install:

- 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
- **authenticator:** class_name: org.apache.cassandra.auth.PasswordAuthenticator
- **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.

V. Process and Methodology

A. The TPC-DS Benchmark

As mentioned in III-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 any industry that needs to manage, sell, and distribute products such as food, electronics, furniture, music, and toys. It is structured around the business model of a large retail company with a nationwide presence. In addition to its brick-and-mortar stores, the company also conducts sales through catalogs and online channels. The benchmark includes models for key operational areas like sales, returns, inventory management, and promotions. Below 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 [8].

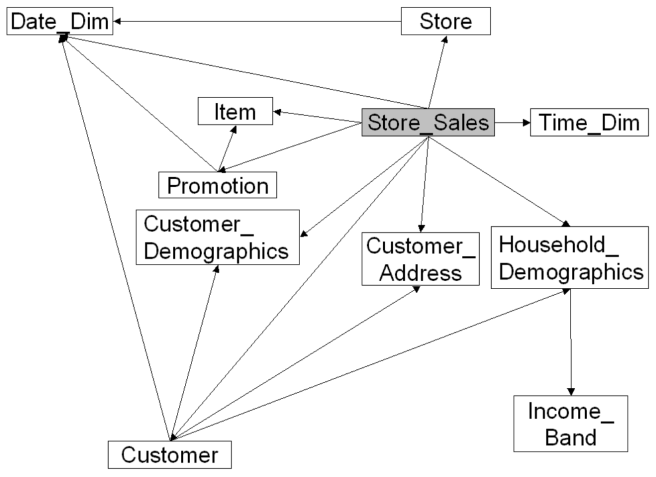

Fig. 3. Store Sales - ER Diagram

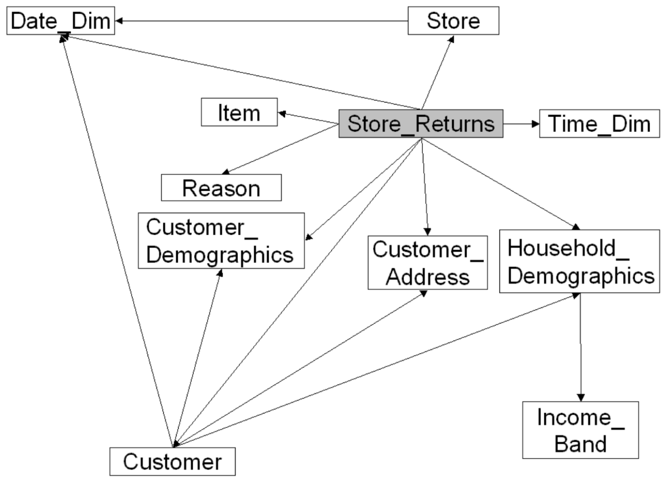

Fig. 4. Store Returns - ER Diagram

3) Query Templates: The TPC-DS also provides a query suite which we used to benchmark the performance of our different data distribution strategies. According to the TPC-DS specification [8] 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)

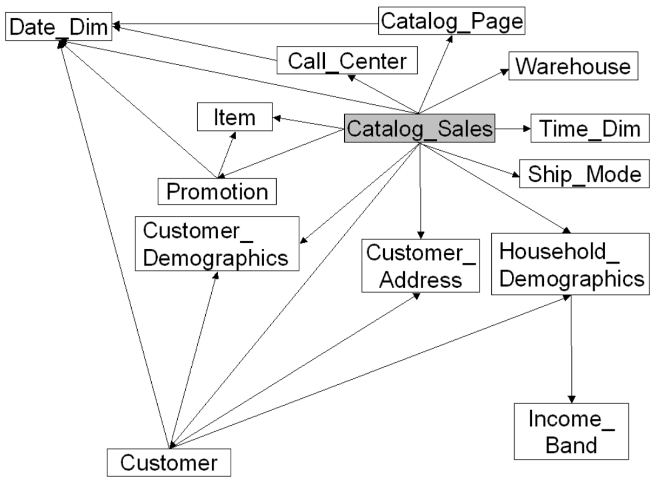

Fig. 5. Catalog Sales - ER Diagram

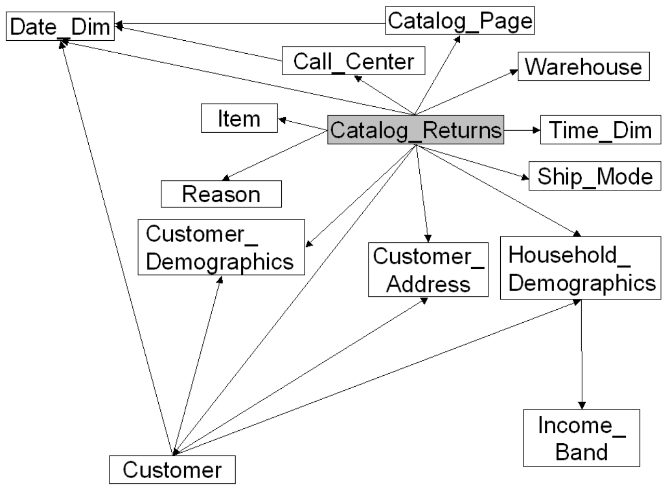

Fig. 6. Catalog Returns - ER Diagram

- Iterative OLAP Queries (queries that discover new trends)
- Data Mining Queries (queries that sift through large data, typically joins and aggregations)

B. Data Distribution Strategy

1) No data distribution: Initially, we measured the performance without any data distribution to get a base system performance value. We note here that we did not opt to test the Redis performance on the full data set due to the RAM limitations (IV-A) of the virtual machines used in our test system. We chose only to add the web sales and returns fact tables with all the accompanied dimension tables to compare the query run time on queries consisting those (or part of those) tables. The full comparison results between PostgreSQL, Cassandra and Redis on the same test queries are presented in Section VI-A

2) Distribution based on fact tables ER: The performance measurements from the first test with no data distribution

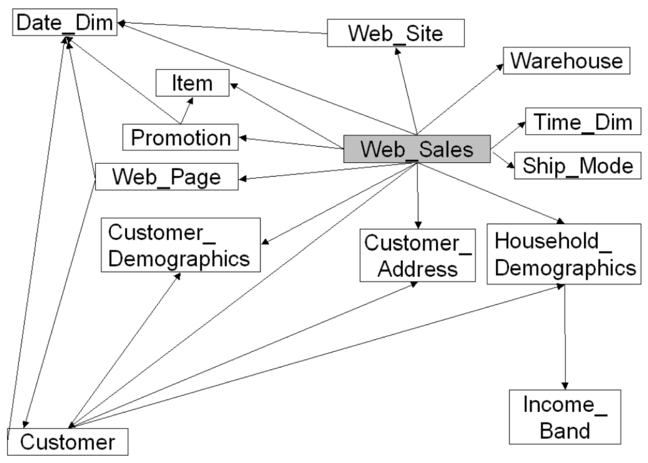

Fig. 7. Web Sales - ER Diagram

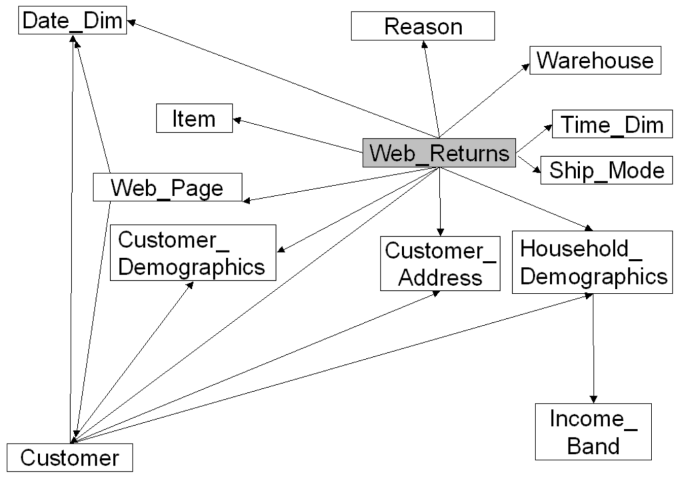

Fig. 8. Web Returns - ER Diagram

present (VI-A) highlighted a clear performance disparity between PostgreSQL, Cassandra and Redis. PostgreSQL was faster in most queries while Redis was significally slower than both of them which was expected due to the poor optimization of Redis on large datasets (add reference). Cassandra while trailing back comparatively to PostgreSQL in most cases, it was faster in some queries while significally slower in some 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 is one that partitions the data based on the ER of the fact tables. The principles that the strategy complies with are the following:

- Fact tables from the same channel are located in the same database (e.g. store sales and store returns)
- Load databases based on the first test result (e.g. larger tables on more performant databases)
- All dimension tables that accompany a fact table are also located in the same database as the fact table (e.g. inventory is located in PostgreSQL, warehouse, date_dim and item are also in PostgreSQL)
- If a dimension table is associated with multiple tables that are located in different databases the dimension table is saved in both databases (e.g. date_dim is associated with all the fact tables so we insert it in all 3 databases)
-

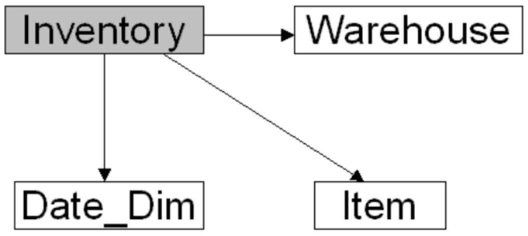

Fig. 9. Inventory - ER Diagram

VI. Results and Analysis

A. No data distribution

B. Distribution based on fact tables ER

C. 2nd distribution strategy

VII. 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. [11] address a gap in research on the performance of SQL-on-Hadoop systems when deployed in containerized 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.

VIII. Conclusion

IX. References

A. Units

- Use either SI (MKS) or CGS as primary units. (SI units are encouraged.) English units may be used as secondary units (in parentheses). An exception would be the use of English units as identifiers in trade, such as "3.5-inch disk drive".
- Avoid combining SI and CGS units, such as current in amperes and magnetic field in oersteds. This often leads to confusion because equations do not balance dimensionally. If you must use mixed units, clearly state the units for each quantity that you use in an equation.
- Do not mix complete spellings and abbreviations of units: "Wb/m^2" or "webers per square meter", not "webers/m^2". Spell out units when they appear in text: ". . . a few henries", not ". . . a few H".
- Use a zero before decimal points: "0.25", not ".25". Use "cm^3", not "cc".)

B. Equations

Number equations consecutively. To make your equations more compact, you may use the solidus (/), the exp function, or appropriate exponents. Italicize Roman symbols for quantities and variables, but not Greek symbols. Use a long dash rather than a hyphen for a minus sign. Punctuate equations with commas or periods when they are part of a sentence, as in:

$$a + b = \gamma \tag{1}$$

Be sure that the symbols in your equation have been defined before or immediately following the equation. Use "(1)", not "Eq. (1)" or "equation (1)", except at the beginning of a sentence: "Equation (1) is . . ."

C. LaTeX-Specific Advice

Please use "soft" (e.g., `\eqref{Eq}`) cross references instead of "hard" references (e.g., `(1)`). That will make it possible to combine sections, add equations, or change the order of figures or citations without having to go through the file line by line.

Please don't use the `{eqnarray}` equation environment. Use `{align}` or `{IEEEeqnarray}` instead. The `{eqnarray}` environment leaves unsightly spaces around relation symbols.

Please note that the `{subequations}` environment in LaTeX will increment the main equation counter even when there are no equation numbers displayed. If you forget that, you might write an article in which the equation numbers skip from (17) to (20), causing the copy editors to wonder if you've discovered a new method of counting.

BibTeX does not work by magic. It doesn't get the bibliographic data from thin air but from .bib files. If you use BibTeX to produce a bibliography you must send the .bib files.

LaTeX can't read your mind. If you assign the same label to a subsubsection and a table, you might find that Table I has been cross referenced as Table IV-B3.

LaTeX does not have precognitive abilities. If you put a `\label` command before the command that updates the counter it's supposed to be using, the label will pick up the last counter to be cross referenced instead. In particular, a `\label` command should not go before the caption of a figure or a table.

Do not use `\nonumber` inside the `{array}` environment. It will not stop equation numbers inside `{array}` (there won't be any anyway) and it might stop a wanted equation number in the surrounding equation.

D. Some Common Mistakes

- The word "data" is plural, not singular.
- The subscript for the permeability of vacuum μ_0, and other common scientific constants, is zero with subscript formatting, not a lowercase letter "o".
- In American English, commas, semicolons, periods, question and exclamation marks are located within quotation marks only when a complete thought or name is cited, such as a title or full quotation. When quotation marks are used, instead of a bold or italic typeface, to highlight a word or phrase, punctuation should appear outside of the quotation marks. A parenthetical phrase or statement at the end of a sentence is punctuated outside of the closing parenthesis (like this). (A parenthetical sentence is punctuated within the parentheses.)
- A graph within a graph is an "inset", not an "insert". The word alternatively is preferred to the word "alternately" (unless you really mean something that alternates).
- Do not use the word "essentially" to mean "approximately" or "effectively".
- In your paper title, if the words "that uses" can accurately replace the word "using", capitalize the "u"; if not, keep using lower-cased.
- Be aware of the different meanings of the homophones "affect" and "effect", "complement" and "compliment", "discreet" and "discrete", "principal" and "principle".
- Do not confuse "imply" and "infer".
- The prefix "non" is not a word; it should be joined to the word it modifies, usually without a hyphen.
- There is no period after the "et" in the Latin abbreviation "et al.".
- The abbreviation "i.e." means "that is", and the abbreviation "e.g." means "for example".

An excellent style manual for science writers is [7].

E. Authors and Affiliations

The class file is designed for, but not limited to, six authors. A minimum of one author is required for all conference articles. Author names should be listed starting from left to right and then moving down to the next line. This is the author sequence that will be used in future citations and by indexing services. Names should not be listed in columns nor group by affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization).

F. Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is "Heading 5". Use "figure caption" for your Figure captions, and "table head" for your table title. Run-in heads, such as "Abstract", will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced.

G. Figures and Tables

a) Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation "Fig. 10", even at the beginning of a sentence.

TABLE I
TABLE TYPE STYLES

Table Head	**Table Column Head**		
	Table column subhead	***Subhead***	***Subhead***
copy	More table copy[a]		

[a]Sample of a Table footnote.

Fig. 10. Example of a figure caption.

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity "Magnetization", or "Magnetization, M", not just "M". If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write "Magnetization (A/m)" or "Magnetization {A[m(1)]}", not just "A/m". Do not label axes with a ratio of quantities and units. For example, write "Temperature (K)", not "Temperature/K".

ACKNOWLEDGMENT

The preferred spelling of the word "acknowledgment" in America is without an "e" after the "g". Avoid the stilted expression "one of us (R. B. G.) thanks ...". Instead, try "R. B. G. thanks...". Put sponsor acknowledgments in the unnumbered footnote on the first page.

REFERENCES

Please number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use "Ref. [3]" or "reference [3]" except at the beginning of a sentence: "Reference [3] was the first ..."

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the abstract or reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors' names; do not use "et al.". Papers that have not been published, even if they have been submitted for publication, should be cited as "unpublished" [4]. Papers that have been accepted for publication should be cited as "in press" [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

REFERENCES

[1] R. Sethi et al., "Presto: SQL on Everything," 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," 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. 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] "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] Ahmad Houwaiji "Trino Deep Dive" 2023. [Online]. Available: https://whitestork.me/blog/13/Trino-Deep-Dive

[10] okeanos-knossos. [Online]. https://okeanos-knossos.grnet.gr/home/

[11] Cristian Cardas, José F. Aldana-Martín, Antonio M. Burgueño-Romero, Antonio J. Nebro, Jose M. Mateos, and Juan J. Sánchez. 2022. On the performance of SQL scalable systems on Kubernetes: a comparative study. Cluster Computing 26, 3 (Jun 2023)

IEEE conference templates contain guidance text for composing and formatting conference papers. Please ensure that all template text is removed from your conference paper prior to submission to the conference. Failure to remove the template text from your paper may result in your paper not being published.