A study of the Trino (Presto) polystore for executing SQL analytic queries

Kapsali Eleni-Elpida
dept. of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el17817@mail.ntua.gr

Lymperopoulos Eleftherios
dept. of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el17061@mail.ntua.gr

Stoikou Theodoti
dept. of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el17085@mail.ntua.gr

***Abstract*—The purpose of this project is to connect the distributed SQL query engine Trino with the databases MongoDB, Apache HBase, Apache Cassandra and study the properties "Query optimization", "Scalability" and "Performance" of the system. The code implemented for the project is available on GitHub [1].**

***Index Terms*—Trino, SQL query engine, MongoDB, Apache HBase, Apache Cassandra**

I. Introduction

Big data and data analysis play important role in our days. Distributed query execution over great amounts of data creates a lot of challenges that search for a solution. Big data has forced companies to develop better data management tools to ensure data scientists and analysts can manage all their data. In this context Trino engine was developed.

Trino provides a solution to organizations that are forced to deploy multiple incompatible SQL-like systems to solve different classes of analytics problems. The implemented system tests Trino under different scenarios. This study comes to conclusions about the parameters which affect Trinos' performance.

At the first section of this paper we present the components and the architecture of the designed system. At the second section we present all the steps we followed to build the system for someone who wants to reproduce it. After that we present all the experiments and at the last section we summarize our results.

II. System's Components and Architecture

The implemented system consists of three (3) Virtual Machines (VMs) hosted by Okeanos Knossos. In our implementation Trino is connected with MongoDB, Apache Cassandra and Apache HBase databases. We study the performance our system using the TPC-DS Benchmark Suite. We use Trino's TPC-DS connector to run the benchmarks. The figure below (Fig 1) describes the implemented architecture. Only VM1 is connected to the public network since it is the only one with public IPv4 address. The three VMs are connected to a private network.

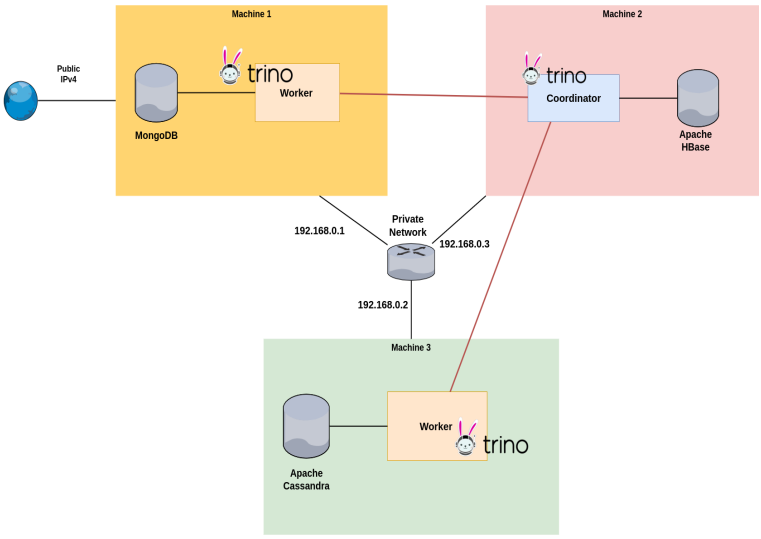

Fig. 1. System's Architecture

A. Trino

Apache Trino (Fig.2) is an open source, distributed SQL query engine. It was designed and written from the ground up to efficiently query data against disparate data sources of all sizes, ranging from gigabytes to petabytes.

Trino is a tool designed to efficiently query vast amounts of data by using distributed execution. If you have terabytes or even petabytes of data to query, you are likely using tools such as Apache Hive that interact with Hadoop and its Hadoop Distributed File System (HDFS). Trino is designed as an alternative to these tools to more efficiently query that data. Analysts, who expect SQL response times from milliseconds

[1] https://github.com/el2kaps/Information-Systems.git

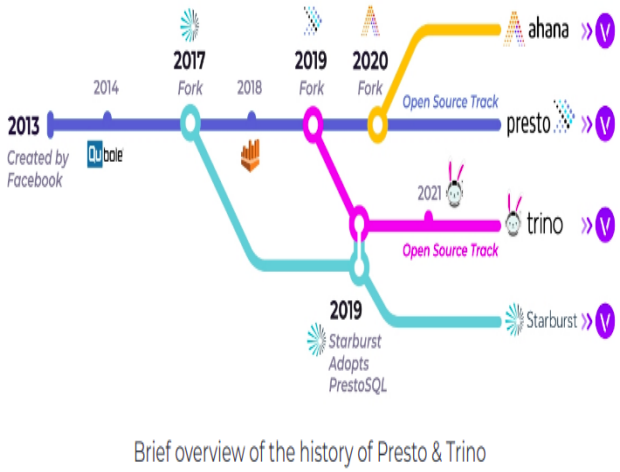

Fig. 2. History of Presto and Trino [12]

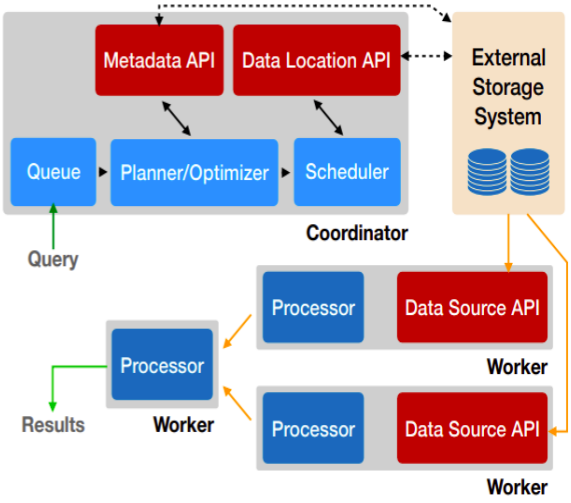

Fig. 4. Trino (Presto) Architecture [7]

for realtime analysis to seconds and minutes, should use Trino. Trino supports SQL, commonly used in data warehousing and analytics for analyzing data, aggregating large amounts of data, and producing reports. These workloads are often classified as online analytical processing (OLAP).

Even though Trino understands and can efficiently execute SQL, Trino is not a database, as it does not include its own data storage system. It is not meant to be a general-purpose relational database that serves to replace Microsoft SQL Server, Oracle Database, MySQL, or PostgreSQL. Instead, Trino uses Connector API to provide a high performance I/O interface to a lot of data sources. Trino supports a lot of famous data sources such as Hadoop data warehouses, relational DBMSs, NoSQL systems, and stream processing systems. [1]

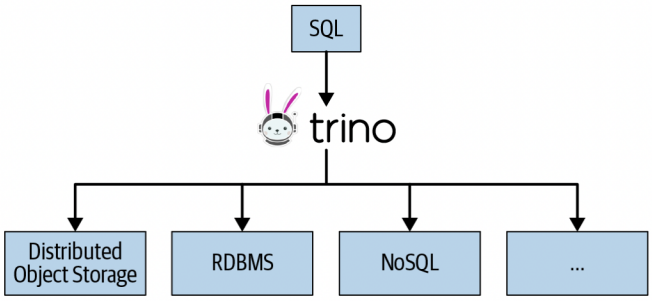

Fig. 3. Capabilities of Trino connectors

”A Presto cluster consists of a single coordinator 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.” [9]

Rather than relying on vertical scaling of the server running Trino, it is able to distribute all processing across a cluster of servers in a horizontal fashion. This means that more nodes can be added to boost processing power. Leveraging this architecture, the Trino query engine is able to process SQL queries on large amounts of data in parallel across a cluster of computers, or nodes. Trino runs as a single-server process on each node. Multiple nodes running Trino, which are configured to collaborate with each other, make up a Trino cluster.

1) Coordinator: Every Trino installation must have a coordinator alongside one or more workers. For development or testing purposes, a single instance of Trino can be configured to perform both roles. The Trino coordinator is the server responsible for receiving SQL statements from the users, parsing these statements, planning queries, and managing worker nodes. Users interact with the coordinator via the Trino CLI, applications using the JDBC or ODBC drivers, or any other available client libraries for a variety of languages. The coordinator accepts SQL statements from the client such as SELECT queries for execution.

Once it receives a SQL statement, the coordinator is responsible for parsing, analyzing, planning, and scheduling the query execution across the Trino worker nodes. The statement is translated into a series of connected tasks running on a cluster of workers. As the workers process the data, the results are retrieved by the coordinator and exposed to the clients on an output buffer. Once an output buffer is completely read by the client, the coordinator requests more data from the workers on behalf of the client. The workers, in turn, interact with the data sources to get the data from them. As a result, data is continuously requested by the client and supplied by the workers from the data source until the query execution is completed.

Coordinators communicate with workers and clients by using HTTP/HTTPS protocols.

2) Discovery service: [1]

Trino uses a discovery service to find all nodes in the cluster. Every Trino instance registers with the discovery service on startup and periodically sends a heartbeat signal. This allows

the coordinator to have an up-to-date list of available workers and use that list for scheduling query execution. If a worker fails to report heartbeat signals, the discovery service triggers the failure detector, and the worker becomes ineligible for further tasks. To simplify deployment and avoid running an additional service, the Trino coordinator typically runs an embedded version of the discovery service. It shares the HTTP server with Trino and thus uses the same port.

Worker configuration of the discovery service therefore typically points at the host name and port of the coordinator.

```
user@master:~/Trino$ ./trino --server localhost:8080 --catalog tpcds --schema default
trino:default> SHOW SCHEMAS FROM tpcds;
       Schema
--------------------
 information_schema
 sf1
 sf10
 sf100
 sf1000
 sf10000
 sf100000
 sf300
 sf3000
 sf30000
 tiny
(11 rows)

Query 20220320_150641_00001_szs4x, FINISHED, 1 node
Splits: 7 total, 7 done (100.00%)
2.20 [11 rows, 128B] [4 rows/s, 58B/s]
```

Fig. 5. Trino's Command line interface

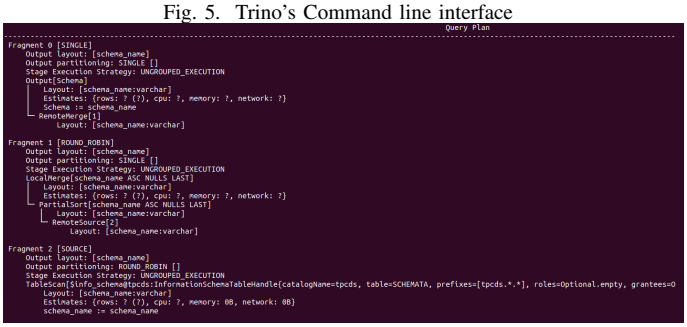

Fig. 6. Trino's Query Optimizer

3) Workers: [1]

A Trino worker is a server in a Trino installation. It is responsible for executing tasks assigned by the coordinator and for processing data. Worker nodes fetch data from data sources by using connectors and then exchange intermediate data with each other. The final resulting data is passed on to the coordinator. The coordinator is responsible for gathering the results from the workers and providing the final results to the client. During installation, workers are configured to know the hostname or IP address of the discovery service for the cluster. When a worker starts up, it advertises itself to the discovery service, which makes it available to the coordinator for task execution. Workers communicate with other workers and the coordinator by using an HTTP, HTTPS protocols.

4) Connectors: [1]

A connector provides Trino an interface to access an arbitrary data source. Each connector provides a table-based abstraction over the underlying data source. As long as data can be expressed in terms of tables, columns, and rows by using the data types available to Trino, a connector can be created and the query engine can use the data for query processing.

5) Trino Web UI: [1] Every Trino server provides a web interface, commonly referred to as the Trino Web UI. The Trino Web UI is accessible at the same address as the Trino server, using the same HTTP port number. By default, this port is 8080. So a local installation, the Web UI is located at http://localhost:8080.

The main dashboard shows details about the Trino utilization and a list of queries. Further details are available in the UI. All this information is of great value for operating Trino and managing the running queries.

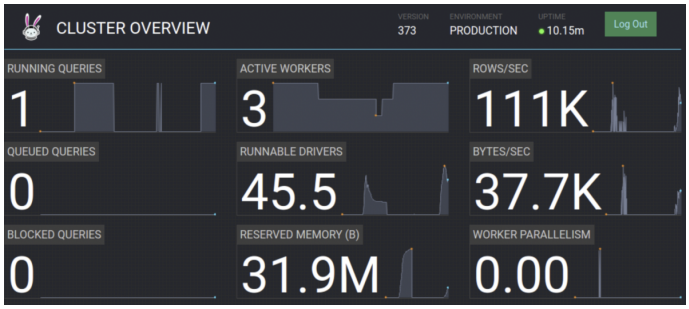

Fig. 7. The Trino Web UI

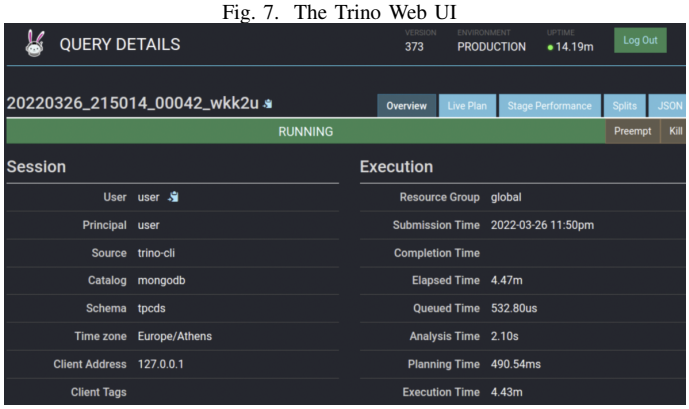

Fig. 8. Trino provides statistics for each query

B. MongoDB

MongoDB is an open-source document document database characterized by scalability and flexibility, providing querying and indexing. It is used to build highly available and scalable internet applications. With its flexible schema approach, it's popular with development teams using agile methodologies. Offering drivers for all major programming languages, MongoDB allows developers to immediately start building applications without the need to configure a database.

Instead of storing data in tables of rows or columns like SQL databases, each record in a MongoDB database is a document described in BSON, a binary representation of the data. Data stored in BSON can be searched and indexed, tremendously increasing performance. MongoDB supports a wide variety of indexing methods, including text, decimal, geospatial, and partial. Applications can then retrieve this information in a JSON format. This format directly maps to native objects in most modern programming languages, making it a natural

choice for developers. MongoDB can also handle high volume and can scale both vertically or horizontally to accommodate large data loads. Regarding the possible deployment options, MongoDB is available in any major public cloud (such as AWS, Azure, and Google Cloud) through MongoDB Atlas. [2]

Additionally, MongoDB supports massive numbers of reads and writes. Sharding is a way to split data across multiple servers. In a MongoDB Sharded Cluster, the database will handle distribution of the data and dynamically load-balance queries. MongoDB uses a shard key to split collections, which determines how the data will be distributed into chunks. Alternatively, the shard key can be hashed, enabling a uniform distribution of data. [3]

Finally, aggregation in MongoDB is supported by aggregation pipelines and Single purpose aggregation methods. Aggregation pipelines provide better performance for most common operations. An aggregation pipeline consists of one or more stages that process documents. Each stage performs an operation on the input documents. The documents that are output from a stage are passed to the next stage. An aggregation pipeline can return results for groups of documents. For example, return the total, average, maximum, and minimum values. The single purpose aggregation methods aggregate documents from a single collection. The methods are simple but lack the capabilities of an aggregation pipeline. [4]

C. Apache HBase

Apache HBase is the Hadoop database, a distributed, scalable, open-source, NoSQL big data store. It enables random, strictly consistent, real-time read/write access to petabytes of data. HBase is very effective for handling large, sparse datasets, as it can host very large tables – billions of rows X millions of columns – atop clusters of commodity hardware and it provides Bigtable-like capabilities on top of Hadoop and HDFS. HBase integrates seamlessly with Apache Hadoop and the Hadoop ecosystem and runs on top of the Hadoop Distributed File System (HDFS) or Amazon S3 using Amazon Elastic MapReduce (EMR) file system, or EMRFS. HBase serves as direct input and output to the Apache MapReduce framework for Hadoop, and works with Apache Phoenix to enable SQL-like queries over HBase tables. [5]

HBase is a column-oriented, non-relational database. This means that data is stored in individual columns, and indexed by a unique row key. This architecture allows for rapid retrieval of individual rows and columns and efficient scans over individual columns within a table. Both data and requests are distributed across all servers in an HBase cluster, allowing the developer to query results on petabytes of data within milliseconds. HBase is most effectively used to store non-relational data, accessed via the HBase API. Apache Phoenix is commonly used as a SQL layer on top of HBase allowing the usage of familiar SQL syntax to insert, delete, and query data stored in HBase.

The most significant benefits presented by Apache HBase relate to the scalability, the speed and the tolerance for error. More specifically, HBase is designed to handle linear and modular scaling across thousands of servers and managing access to petabytes of data. HBase provides low latency random read and write access to petabytes of data by distributing requests from applications across a cluster of hosts. Each host has access to data in HDFS and S3, and serves read and write requests in milliseconds. Furthermore, HBase splits data stored in tables across multiple hosts in the cluster and is built to withstand individual host failures. Because data is stored on HDFS or S3, healthy hosts will automatically be chosen to host the data once served by the failed host, and data is brought online automatically. Additional advantages of HBase regard the feature of strictly consistent reads and writes, the automatic and configurable sharding of tables, the automatic failover support between RegionServers and the convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables. It also provides query predicate push down via server side Filters, thrift gateway and a RESTful Web service that supports XML, Protobuf, and binary data encoding options, extensible jruby-based (JIRB) shell and support for exporting metrics via the Hadoop metrics subsystem to files or Ganglia; or via JMX. [6]

D. Apache Cassandra

Apache Cassandra is an open source NoSQL distributed database, characterized by scalability, high availability, reliable performance and speed. It quickly stores massive amounts of incoming data and can handle hundreds of thousands of writes per second, providing linear scalability and proven fault-tolerance on commodity hardware or cloud infrastructure, to fulfill mission-critical data.

Cassandra allows managing large amounts of data quickly. Firstly, regarding performance and speed, specific architectural choices make Cassandra a beneficial technology for processing data and at a faster pace than database alternatives. There are two ways Cassandra achieves a fast speed. It makes quick decisions on where to store data using a hashing algorithm and it lets any node to make data storage decisions. This eliminates the need for a centralised “master node” that needs to be consulted on storage decisions. Cassandra consistently outperforms popular NoSQL alternatives in benchmarks and real applications, primarily because of fundamental architectural choices. Additionally, Cassandra is highly scalable and the developer can increase performance just by adding a new rack. First of all, there is no “master” that needs to be super-sized to handle orchestrating and managing data. This means all the nodes can be cheaper, commodity servers. Second, it achieves scalability by putting less emphasis on data consistency. Consistency typically requires a master node to track and enforce what consistency means either based on rules or previously stored data. Finally, it uses peer-to-peer communication, named “gossip protocol”. This lets nodes communicate and pass metadata among themselves, which makes adding new nodes very easy. Read and write throughput both increase linearly as new machines are added, with no downtime or interruption to applications. [8]

Cassandra's reliability is based on data replication and HA, as it is a robust store of data and the hashing algorithm stores data as well as makes copies and stores them in other locations, meaning that if a node goes down, there is a copy of it. In contrast, traditional databases are very thoughtful and slow in replicating data, as there needs to be a plan on how to make sure different copies are up to date.

To ensure reliability and stability, Cassandra is tested on clusters as large as 1,000 nodes and with hundreds of real world use cases and schemas tested with replay, fuzz, property-based, fault-injection, and performance tests. For each update, the developer can choose between synchronous or asynchronous replication, while there are features responsible for the optimization of asynchronous operations. In addition, security and observability are achieved with minimal impact to normal workload performance, allowing the capture and replay of production workloads for analysis. As Cassandra is a distributed system, it is suitable for applications that can't afford to lose data, even when an entire data center goes down. Cassandra streams data between nodes during scaling operations such as adding a new node or datacenter during peak traffic times. [7]

III. TPC-DS Benchmark

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. A benchmark result measures query response time in single user mode, query throughput in multi user mode and data maintenance performance for a given hardware, operating system, and data processing system configuration under controlled, complex, multi-user decision support workload. The purpose of TPC benchmarks is to provide relevant, objective performance data to industry users. TPC-DS enables emerging technologies, such as Big Data systems, to execute the benchmark.

In practice, TPC-DS models the decision support functions of a retail product supplier so that users can relate intuitively to the components of the benchmark. The imaginary retailer sells its goods via three different distribution channels: store, catalog and internet. TPC-D and its successors TPC-H -R were built using a 3rd Normal Form (3NF) schema. Over the years, however, the industry has expanded towards star schema approaches. As a consequence, TPC-DS implements a multiple snowflake schema, which is a hybrid approach between a 3NF and a pure star schema.

Regarding the query workload, TPC-DS specifies a set of 99 distinct SQL-99 queries (including OLAP extensions), which are designed to cover the entire dataset. Thanks to the schema design, which falls apart into a reporting part (Catalog sales channel, 40% of the entire dataset) and an ad-hoc part (Internet Store sales channels). On the other hand, the query workload consists of the following four query classes: pure reporting queries, pure ad-hoc queries, iterative OLAP queries and extraction or data mining queries. Reporting queries are very well known in advance so that clever data placement methods and auxiliary data structures (e.g., materialized views, indexes) can be used for optimization.

IV. Systems Set Up

A. Installation Guide

To set up the system we have to complete the steps below. Steps are also presented with examples in project's repository README.md.

1) Install the databases using the official guides
   - MongoDB on Machine 1
     Start MongoDB:
     ```
     sudo systemctl start mongod.service
     ```
     Stop MongoDB:
     ```
     sudo systemctl stop mongod.service
     ```
   - Apache Cassandra on Machine 2
   - Apache HBase on Machine 3
2) Set up the databases
   Now we must set up the databases so as to LISTEN to the private network's addresses 192.168.0.1, 192.168.0.2, 192.168.0.3 rather than localhost (127.0.0.1). This step is very important for the coordinator to access all the databases located on different machines through their workers.
   - MongoDB
     Change network interfaces in *mongod.conf* to:
     ```
     # network interfaces
     net:
       port: 27017
       bindIp: 192.168.0.1
       and restart the database. <br>
     ```
     Use the command:
     ```
     sudo lsof -iTCP -sTCP:LISTEN \
     | grep mongo
     ```
     to check that the database listens to the desired IP. *mongod.conf* in our system is located in *etc* directory.
   - Apache Cassandra
     Replace *cassandra.yaml* with the *cassandra.yaml* in the Cassandra directory of project's repository. Use the command:
     ```
     sudo lsof -iTCP -sTCP:LISTEN \
     | grep cassandra
     ```
     to check that the database listens to the desired IP (192.168.02). cassandra.yaml in our system is located in */etc/cassandra* directory.
   - Apache HBase
     HBase's set up is a more complicated because Trino doesn't provides an HBase connenctor but a Phoenix connector. As Trino documentation states to query HBase data through Phoenix, you need:
     - Network access from the Trino coordinator and workers to the ZooKeeper servers. The default port is 2181.

- A compatible version of Phoenix. We use version 5.
- Copy *azul-zulu* directory to each Machine.
  Azul Zulu is an open source implementation of the Java Standard Edition ("SE") specification. It is a binary build of the OpenJDK open source project. Zulu provides a Java Runtime Environment needed for Java applications to run.
  We used version zulu11.54.25-ca-jdk11.0.14.1-linux_x64 (Java 11).
  Add Azul Zulu to your PATH environment variable, so that you can execute java from any directory without specifying the full path.
  ```
  export PATH= \
  /home/user/azul-zulu/ \
  zulu11.54.25-ca-jdk11.0.14.1- \
  linux_x64/bin:$PATH
  ```
- Install Phoenix 5 using the official guide. In our system we use phoenix-hbase-2.4-5.1.2-bin.
- Replace *hbase-site.xml* in both *phoenix-hbase-2.4-5.1.2-bin/bin* and *hbase-2.4.11/conf* with the *hbase-site.xml* in Hbase-conf directory (Fig.3).

```
  <property>
    <name>hbase.cluster.distributed</name>
    <value>false</value>
  </property>
  <property>
    <name>hbase.tmp.dir</name>
    <value>./tmp</value>
  </property>
  <property>
    <name>hbase.unsafe.stream.capability.enforce</name>
    <value>false</value>
  </property>
  <property>
    <name>phoenix.schema.isNamespaceMappingEnabled</name>
    <value>true</value>
  </property>
</configuration>
```

Fig. 9. hbase-site.xml

3) Set up Trino
   - Copy repository's *Trino* directories to the respective Machine.
   - Start Trino servers (one for each machine)
     ```
     cd Trino/trino/server-373
     bin/launcher run
     ```
   - To stop a Trino server run
     ```
     cd Trino/trino/server-373
     bin/launcher stop
     ```
4) Run Trino CLI
   - At the machine where the coordinator Trino node run
     ```
     cd Trino
     ./trino
     ```

B. Post-installation checks

We present some useful commands that help us to study the system in more detail and understand some of its parameters.

- Check that all workers are connected (Fig.3)
  ```
  SELECT * FROM system.runtime.nodes;
  ```

```
trino> select * from system.runtime.nodes;
 node_id |        http_uri         | node_version | coordinator | state
---------+-------------------------+--------------+-------------+--------
 3       | http://192.168.0.3:8080 | 373          | true        | active
 2       | http://192.168.0.2:8080 | 373          | false       | active
 1       | http://192.168.0.1:8080 | 373          | true        | active
(3 rows)
```

Fig. 10. System's Nodes

- Check that we have access to all databases (Fig.4)
  ```
  SHOW SCHEMAS FROM <catalog name>;
  ```

```
trino> show schemas from phoenix;
       Schema
--------------------
 default
 information_schema
 system
(3 rows)

Query 20220325_163710_00001_2pfd8, FINISHED, 3 nodes
Splits: 17 total, 17 done (100.00%)
3.84 [3 rows, 46B] [0 rows/s, 12B/s]

trino> show schemas from mongodb;
       Schema
--------------------
 admin
 config
 information_schema
 local
 myshinynewdb
 test_database
(6 rows)

Query 20220325_163721_00002_2pfd8, FINISHED, 3 nodes
Splits: 17 total, 17 done (100.00%)
0.24 [6 rows, 89B] [25 rows/s, 372B/s]
```

Fig. 11. Show Schemas

- To work on a specific schema for example *mongodb.tpcds* run:
  ```
  USE mongodb.tpcds
  ```
  Now tables *store_returns*, *date_dim*,*store*, *customer* of *Query-1* refers to the tables with the same names stored in MongoDB. After *USE* the query of Fig.5 is equivalent to the query at Fig.6.
- Run
  ```
  DESCRIBE <table name>
  ```
  to view sql columns of a table (Fig.8).
- Use
  ```
  SHOW STATS
  ```
  for approximated statistics for the named table.

V. Benchmark Preparation

In this study we test Trino using the TPC-DS Benchmark. In order to use the TPC-DS benchmark, we used Trino's TPC-DS

```
-- TPCDS-Query-1
WITH customer_total_return
     AS (SELECT sr_customer_sk AS ctr_customer_sk,
                sr_store_sk AS ctr_store_sk,
                Sum(sr_return_amt) AS ctr_total_return
         FROM   store_returns,
                date_dim
         WHERE  sr_returned_date_sk = d_date_sk
                AND d_year = 2001
         GROUP  BY sr_customer_sk,
                   sr_store_sk)
SELECT c_customer_id
FROM   customer_total_return ctr1,
       store,
       customer
WHERE  ctr1.ctr_total_return > (SELECT Avg(ctr_total_return) * 1.2
                                FROM   customer_total_return ctr2
                                WHERE  ctr1.ctr_store_sk =
ctr2.ctr_store_sk)
       AND s_store_sk = ctr1.ctr_store_sk
       AND s_state = 'TN'
       AND ctr1.ctr_customer_sk = c_customer_sk
ORDER  BY c_customer_id
LIMIT 100;
```

Fig. 12. TPCDS-Query 1 (1)

```
-- TPCDS-Query-1
WITH customer_total_return
     AS (SELECT sr_customer_sk AS ctr_customer_sk,
                sr_store_sk AS ctr_store_sk,
                Sum(sr_return_amt) AS ctr_total_return
         FROM   mongo_db.tpcds.store_returns,
                mongo_db.tpcds.date_dim
         WHERE  sr_returned_date_sk = d_date_sk
                AND d_year = 2001
         GROUP  BY sr_customer_sk,
                   sr_store_sk)
SELECT c_customer_id
FROM   customer_total_return ctr1,
       mongo_db.tpcds.store,
       mongo_db.tpcds.customer
WHERE  ctr1.ctr_total_return > (SELECT Avg(ctr_total_return) * 1.2
                                FROM   customer_total_return ctr2
                                WHERE  ctr1.ctr_store_sk =
ctr2.ctr_store_sk)
       AND s_store_sk = ctr1.ctr_store_sk
       AND s_state = 'TN'
       AND ctr1.ctr_customer_sk = c_customer_sk
ORDER  BY c_customer_id
LIMIT 100;
```

Fig. 13. TPCDS-Query 1 (2)

```
trino:tpcds> describe customer;
        Column         |    Type     | Extra | Comment
-----------------------+-------------+-------+---------
 c_customer_sk         | bigint      |       |
 c_customer_id         | char(16)    |       |
 c_current_cdemo_sk    | bigint      |       |
 c_current_hdemo_sk    | bigint      |       |
 c_current_addr_sk     | bigint      |       |
 c_first_shipto_date_sk | bigint     |       |
 c_first_sales_date_sk | bigint      |       |
 c_salutation          | char(10)    |       |
 c_first_name          | char(20)    |       |
 c_last_name           | char(30)    |       |
 c_preferred_cust_flag | char(1)     |       |
 c_birth_day           | integer     |       |
 c_birth_month         | integer     |       |
 c_birth_year          | integer     |       |
 c_birth_country       | varchar(20) |       |
 c_login               | char(13)    |       |
 c_email_address       | char(50)    |       |
 c_last_review_date_sk | bigint      |       |
(18 rows)
```

Fig. 14. DESCRIBE command

connector. This connector provides Trino with TPC-DS data schemas in various scale factors, ranging from 1 up to 100000. Each scale factor unit is equivalent to 1 GB of data [10]. Due to memory constraints of the VMs in Okeanos-Knossos we utilize the schema with scale factor 1. In order to load the dataset into each database, we firstly plugged the TPC-DS connector into the Trino coordinator node. Then, after running the Trino server and Trino's CLI, the command

`SHOW SCHEMAS FROM tpcds;`

shows all available TPC-DS schemas.

```
[trino> show schemas from tpcds;
       Schema
--------------------
 information_schema
 sf1
 sf10
 sf100
 sf1000
 sf10000
 sf100000
 sf300
 sf3000
 sf30000
 tiny
(11 rows)

Query 20220327_183021_00036_eyg9x, FINISHED, 3 nodes
Splits: 17 total, 17 done (100.00%)
0.42 [11 rows, 128B] [26 rows/s, 305B/s]
```

Fig. 15. All available TPC-DS schemas

Then we proceeded to copy all tables of the schema sf1 to each database. We for example, in the case of MongoDB, the

command to create the table 'customer' and insert all data of the TPC-DS 'customer' table is:

```
CREATE TABLE mongodb.tpcds.customer
AS SELECT * FROM tpcds.sf1.customer;
```

We followed the same procedure to insert the data in Cassandra and HBase. Note that in the case of Cassandra, as Cassandra does not support some SQL data types such as char and decimal, we had to convert some tables in TPC-DS to varchar and double respectively. After insertion of all data, the databases are then ready to be queried. The TPC-DS queries are taken from Apache Spark's official Github repository at [11]

VI. Trino's Query Optimizer

A. Overview

Trino supports statistics based optimizations for queries. Table statistics are provided to the query planner by connectors. Those statistics are used by the query in order to perform optimizations. The set of statistics available for a particular query depends on the connector being used including the total number of rows in the table, the size of the data that needs to be read from a column, the fraction of null values in a column, the number of distinct values in a column, the smallest value in the column, the largest value in the column.

During planning, the cost associated with each node of the plan is computed based on the table statistics for the tables in the query. Trino enables viewing the cost associated with each node of the plan using "EXPLAIN" keyword before each query.

Fig. 16. Trino's EXPLAIN functionality

Cost information is displayed in the plan tree using the above format where *r*ows refers to to the expected number of rows output by each plan node during execution and *c*pu, *m*emory, *n*etwork refers to the estimated of CPU, memory, and network utilized by the execution of a plan node.

```
{rows: XX (XX), cpu: XX,memory: XX,
network: XX}.rows
```

Example of usage of Trino's Query Optimizer on query

```
SHOW SCHEMAS FROM tpcds;
```

using Trino's CLI (Fg.4, Fig.5)

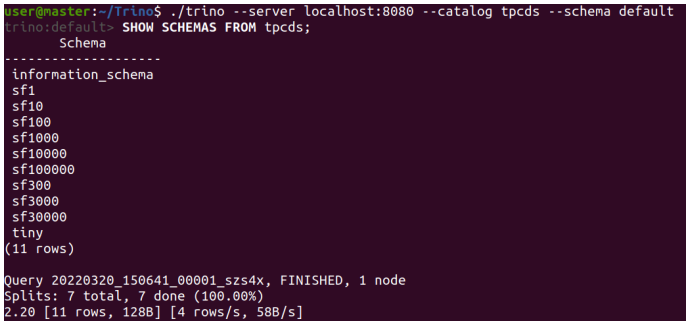

Fig. 17. Trino's Command line interface

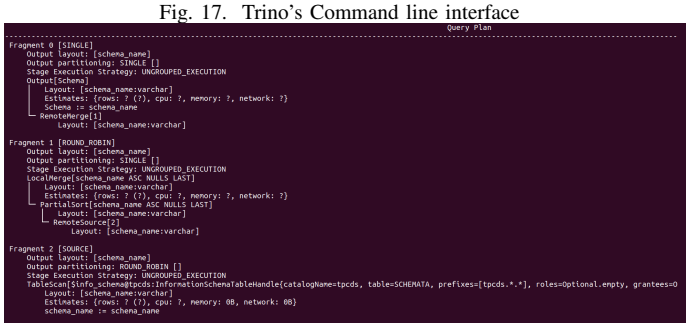

Fig. 18. Trino's Query Optimizer

We use Trino's CLI to execute queries. At the image Fig.4 we see the cost for each plan estimated by Query Optimizer.

B. Query Planning

Before a query can be planned for execution, it needs to be parsed and analyzed. Trino follows the steps below to parse and analyze a query.

- Identifying tables used in a query
- Identifying columns used in a query
- Identifying references to fields within ROW values

Then, Trino proceeds to query plannings.

1) Initial Query Planning: Query planner and optimizer decide the sequence of steps to process the data for the desired result.
2) Query optimization: transforms and evolves the initial plan into an equivalent plan that can be executed fast. At this step Trino follows some Optimization Rules. We'll describe the most common Optimization Rules in the next section.

C. Optimization Rules

The most common and important Optimization Rules that Trino uses are:

- Predicate Pushdown: it tries to reduce data as soon as possible. For this it moves the filtering condition as close to the source of the data as possible.
- Cross Join Elimination: reorders the tables being joined to minimize the number of cross joins, ideally reducing it to zero.
- TopN: This rule is applied to queries where an *ORDER BY* clause is followed by a *LIMIT*. During query execution, TopN keeps the desired number of rows in a heap

data structure, updating the heap while reading input data in a streaming fashion.

VII. Results

First of all we open coordinator's Trino Web UI.Trino UI provides information about active workers, runnable drivers, reserved memory, bytes/sec transmitted, worker parallelism etc. The information provided can be used to understand and tune the Trino system overall as well as individual queries. In this project we use this information to extract results about the bhavior of the system and to do comparison based on system's resources, databases connected on it and from which database we extract queries' tables.

A. Monitoring with the Trino Web UI

We explain some of the dashboard's parts and what information about the system they provide. We will use this information at the next sections.

- *Runnable Drivers*: The average number of runnable drivers in the cluster.
- *Reserved Memory*: The total amount of reserved memory in bytes in Trino. It doesn't specifies amount of reserved memory in each node.
- *Rows/Sec*: The total number of rows processed per second across all queries.
- *Bytes/Sec*: The total number of bytes processed per second across all queries.
- *Worker Parallelism*: The total amount of worker parallelism, which is the total amount of thread CPU time across all workers, across all queries running in the cluster.
- *Query List*: Provides information for a specific query like Completed Splits, Running Splits, Wall Time and Current Total Reserved. Memory

B. Load TPC-DS data to databases

First of all we load TPC-DS data to each database with CREATE TABLE AS queries. Because of memory limitations we weren't able to load very big scale TPC-DS data. For example:

```
CREATE TABLE customer
AS SELECT * FROM tpcds.sf10.customer;
```

At this point loading TPC-DS data to Cassandra causes the error *"Unsupported type: decimal(7,2)"*. We use SQL's type cast function to convert the data to suitable for Cassandra format. For example:

```
select
web_site_sk,
CAST(web_site_id,
web_rec_start_date,
web_rec_end_date,
web_name,
web_open_date_sk,
web_close_date_sk,
web_class,
web_manager,
web_mkt_id,
web_mkt_class,
web_mkt_desc,
web_market_manager,
web_company_id,
CAST(web_company_name AS VARCHAR)
  as web_company_name,
CAST(web_street_number AS VARCHAR)
  as web_street_number,
...
```

C. Test Scenarios

We present five (5) scenarios on which we tested our system using TPC-DS Benchmark. The results focus on systems behavior depending on how many databases are connected in Trino, their resources and where queries tables are located. For the experiments we chose some TPC-DS queries so as to have scalability in complexity. Query complexity is usually described by the *joins* it contains.

1) Scenario 1: 3 workers, DBs: MongoDB, Cassandra, HBase
   Results:

```
QUERY 1:
 Splits: 255 total, 255 done (100.00%)
 3.00 [25.6K rows, 0B]
 [8.53K rows/s, 0B/s]
QUERY 7:
 Splits: 2,886 total, 2,886 done (100.00%)
 44.59 [2.93M rows, 2.75MB]
 [65.6K rows/s, 63.1KB/s]
QUERY 12:
 Splits: 384 total, 384 done (100.00%)
 2.27 [103K rows, 71.9KB]
 [45.3K rows/s, 31.7KB/s]
QUERY 15:
 Splits: 400 total, 400 done (100.00%)
 4.82 [191K rows, 97.7KB]
 [39.6K rows/s, 20.3KB/s]
QUERY 62:
 Splits: 1,108 total, 1,108 done (100.00%)
 9.87 [720K rows, 703KB]
 [72.9K rows/s, 71.2KB/s]
```

2) Scenario 2: 3 workers, DBs: MongoDB
   Results:

```
QUERY 1: 7.07 seconds, 95.7K rows/s
QUERY 7: 16.39 seconds 97.1K rows/s
QUERY 12: 8.50 seconds 95.3K rows/s
QUERY 15: 7.92 seconds 192K rows/s
QUERY 62: 8.26 seconds 87.2K rows/s
```

3) Scenario 3: 3 workers, DBs: Cassandra
   Results:

```
QUERY 1: 13.67 seconds 60.1K rows/s
QUERY 7: 27.64 seconds 76.5K rows/s
```

```
QUERY 12: 8.19 seconds 19.8K rows/s
QUERY 15: 26.29 seconds 56.9K rows/s
QUERY 62: 16.85 seconds 47K rows/s
```

It is noteworthy that there are much more splits observed in the case of Cassandra (of the order of 1000) than in the case of the MongoDB (of the order of 100).

4) Scenario 4: 2 workers, DBs: Cassandra
Results:

```
QUERY 1: 10.53 sec 78K rows/s
QUERY 7: 25.94 seconds 83.4K rows/s
QUERY 12: 5.72 seconds 25.1K rows/s
QUERY 15:  22.70 seconds 73.3K rows/s
QUERY 62: 16.53 seconds 47.9K rows/s
```

5) Scenario 5: 3 workers, DBs: MongoDB, Cassandra
Results:

```
QUERY 1:
 Splits: 1,023 total,
         1,023 done (100.00%)
 10.27 [821K rows, 145KB]
 [80K rows/s, 14.1KB/s]
QUERY 7:
 Splits: 4,925 total,
         4,925 done (100.00%)
 20.98 [2.18M rows, 2.07MB]
 [104K rows/s, 101KB/s]
QUERY 12:
 Splits: 802 total,
         802 done (100.00%)
 8.92 [810K rows, 703KB]
 [90.8K rows/s, 78.8KB/s]
QUERY 15:
 Splits: 1,732 total,
         1,732 done (100.00%)
 17.74 [1.59M rows, 1.42MB]
 [89.7K rows/s, 82.2KB/s]
QUERY 62:
 Splits: 1,364 total,
         1,364 done (100.00%)
 10.36 [720K rows, 703KB]
 [69.5K rows/s, 67.9KB/s]
```

ndicatively, we present some screenshots of the queries we executed. (Figs. 18-20)

VIII. CONCLUSIONS

We summarize some observations from *Results* in the following diagrams (Figs. 21-23). In general, we can observe that even by connecting all 3 databases, Trino is capable of executing the queries in low execution times. Trino exploits distributed query processing and execution, and produces better results than executing queries to a single database. Moreover, since Trino splits its data among its workers to parallelize the execution, it achieves remarkable throughput even on demanding queries such as the ones from TPC-DS.

```
trino:tpcds> SELECT ca_zip,
          ->        Sum(cs_sales_price)
          -> FROM   mongodb.tpcds.catalog_sales,
          ->        cassandra.tpcds.customer,
          ->        cassandra.tpcds.customer_address,
          ->        mongodb.tpcds.date_dim
          -> WHERE  cs_bill_customer_sk = c_customer_sk
          ->        AND c_current_addr_sk = ca_address_sk
          ->        AND ( Substr(ca_zip, 1, 5) IN ( '85669', '86197', '88274', '83405',
          ->                                        '86475', '85392', '85460', '80348',
          ->                                        '81792' )
          ->              OR ca_state IN ( 'CA', 'WA', 'GA' )
          ->              OR cs_sales_price > 500 )
          ->        AND cs_sold_date_sk = d_date_sk
          ->        AND d_qoy = 1
          ->        AND d_year = 1998
          -> GROUP  BY ca_zip
          -> ORDER  BY ca_zip
          -> LIMIT 100;
  ca_zip   |  _col1
-----------+---------
 30069     |  959.53
 30150     |  765.39
 30162     |  231.12
 30169     | 1031.83
 30399     |  383.91
 30411     | 1509.07
 30451     |  241.92
 30499     | 1461.13
 30534     |  192.28
 30540     |  483.43
 30587     | 1041.40
 30631     |   84.32
 30725     |  546.89
 31143     |  240.65
 31147     |   86.13
 31289     |  885.39
```

Fig. 19. TPC-DS Query-15 (1)

```
Query 20220326_212614_00082_54kxs, FINISHED, 3 nodes
Splits: 656 total, 656 done (100.00%)
37.80 [1.59M rows, 147KB] [42.1K rows/s, 3.9KB/s]
```

Fig. 20. TPC-DS Query-15 (2)

```
trino:tpcds_db> SELECT i_item_id,
             ->        Avg(ss_quantity)    agg1,
             ->        Avg(ss_list_price)  agg2,
             ->        Avg(ss_coupon_amt)  agg3,
             ->        Avg(ss_sales_price) agg4
             -> FROM   cassandra.tpcds.store_sales,
             ->        phoenix.tpcds_db.customer_demographics,
             ->        mongodb.tpcds.date_dim,
             ->        mongodb.tpcds.item,
             ->        phoenix.tpcds_db.promotion
             -> WHERE  ss_sold_date_sk = d_date_sk
             ->        AND ss_item_sk = i_item_sk
             ->        AND ss_cdemo_sk = cd_demo_sk
             ->        AND ss_promo_sk = p_promo_sk
             ->        AND cd_gender = 'F'
             ->        AND cd_marital_status = 'W'
             ->        AND cd_education_status = '2 yr Degree'
             ->        AND ( p_channel_email = 'N'
             ->              OR p_channel_event = 'N' )
             ->        AND d_year = 1998
             -> GROUP  BY i_item_id
             -> ORDER  BY i_item_id
             -> LIMIT 100;
```

Fig. 21. TPC-DS Query-7 distributed on all databases

Even when switching from 2 workers to 3, as shown in scenarios 3 and 4, the increase in execution time was mininal.

”Presto’s architecture enables it to service workloads that require very low latency and also process expensive, long running queries efficiently” [9]. Trino is an in-memory query engine and as a result it can only process data as fast as the storage layer can provide it. For this reason we can boost Trino’s speed by choosing the fastest data source. Furthermore, from the observations of the previous section we understand the importance of picking the right kind of instance for worker nodes. Those decisions are based on network IO availability since most analytical workloads are IO intensive. In terms of loading the data into Trino, we found that Apache Cassandra is the most reliable option, as the other two databases often had memory or network issues.

In general, the fact that simple SQL can be used to run workloads that combine multiple, even NoSQL, databases efficiently, makes Trino a flexible and powerful solution.

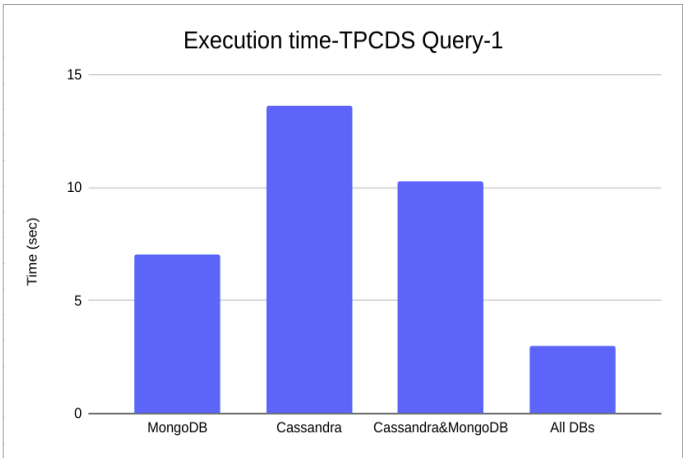

Fig. 22. Execution time of Query-1

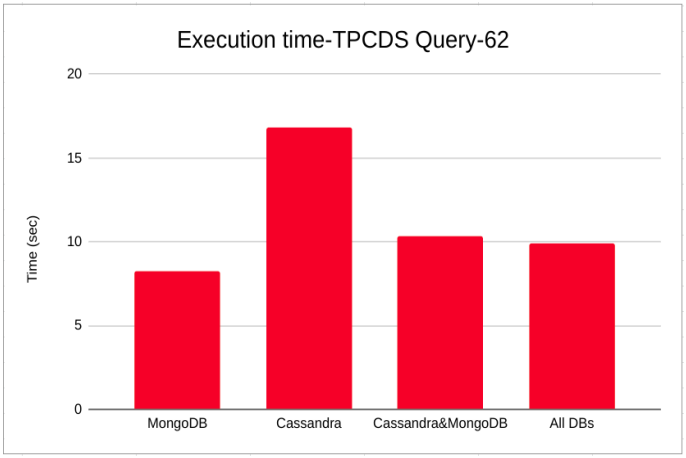

Fig. 23. Execution time of Query-62

REFERENCES

[1] Matt Fuller, Martin Traverso, Manfred Moser, “Trino: The Definitive Guide” O’Reilly Media, April 2021.
[2] https://www.mongodb.com/why-use-mongodb
[3] https://severalnines.com/database-blog/turning-mongodb-replica-set-sharded-cluster
[4] https://www.mongodb.com/docs/manual/aggregation/
[5] https://hbase.apache.org/
[6] https://aws.amazon.com/big-data/what-is-hbase/
[7] https://cassandra.apache.org/_/index.html
[8] https://ubuntu.com/blog/apache-cassandra-top-benefits
[9] R. Sethi et al., ”Presto: SQL on Everything,” 2019 IEEE 35th International Conference on Data Engineering (ICDE)
[10] https://trino.io/docs/current/connector/tpcds.html
[11] https://github.com/apache/spark/tree/master/sql/core/src/test/resources/tpcds
[12] https://varada.io/blog/presto/prestodb-trino-indexing/