Distributed Execution of SQL Queries Using Trino

Konstantinos Sideris
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el18134@mail.ntua.gr

Constantinos N. Papadopoulos
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el18220@mail.ntua.gr

Yiannos Paranomos
School of Electrical and Computer Engineering
National Technical University of Athens
Athens, Greece
el18021@mail.ntua.gr

***Abstract*—This paper will study the performance and scalability of the distributed query engine Trino with various data distribution plans amongst three heterogeneous stores: MySQL, PostgreSQL and Apache Accumulo. For that, we propose different strategies for partitioning the TPC-DS dataset, aiming at improving query latency and bandwidth, taking into account the data model, query performance and the business logic behind the data. Through experiments with a variety of queries and optimizer options, we devise an optimal data partitioning strategy for our configuration and showcase that data-dependent and source-dependent data distribution can significantly improve query performance.**

***Index Terms*—query engines, heterogeneous data sources, Trino, MySQL, PostgreSQL, Apache Accumulo**

I. Introduction

During the past years, there is an ever increasing demand for data analytics against voluminous data which are distributed across diverse data sources. As business processes become more and more complex and the scale of data increases, an emerging challenge in data processing is the need to effectively query data across inconsistent sources. The use of query engines alleviates this problem by allowing users to query multiple data sources under a unified schema. In this paper we will examine one such query engine, Trino. Using the TPC-DS benchmark we measure the performance of Trino using different data partitioning techniques, with data being distributed across three diverse databases: MySQL, PostgreSQL and Apache Accumulo. After an initial round of experiments executing various representative queries on data partitions devised through our theoretical analysis of the TPC-DS dataset, we design an optimal data partition method based on both our theoretical analysis and experimental evidence on query performance. We then test the scalability of the system by executing the same set of representative queries on the optimal data partition using different numbers of Trino worker nodes. Finally, we evaluate the Trino optimizer on different data partitions. After analyzing queries we show that the optimizer chooses the optimal query plan in each case. Our goal by the end of this paper is to successfully measure Trino's ability to query non uniform data sources, test the query engine scalability and evaluate the effectiveness of its built in optimizer.

II. Technologies Used

In this section we will discuss the System Under Test, Trino, and the three underlying databases that make up our three node cluster.

A. Trino

Trino is an open source, distributed SQL query engine. It was designed and written from the ground up to efficiently query data against different data sources of all sizes, ranging from gigabytes to petabytes. The data stored across different sources usually have different logical and business models. Trino bridges possible logical gaps between the data by enforcing a common schema that is mapped to each data source through a connector specific to each source. It leverages both well known and novel methods for query processing, including in-memory parallel processing, pipelined execution across nodes, multithreading to ensure high CPU core utilisation, efficient flat-memory data structures to minimize Java garbage collection, and Java bytecode generation [**?**]. Trino queries data where it is stored and is thus a viable replacement to Extract, Transform, Load (ETL) processes traditionally used in Big Data processing [2].

B. MySQL

MySQL is an open source Relational Database Management System (RDBMS). Databases within MySQL are structured collections of data, organised according to the relational model. In this model, database tables consist of rows and columns, and relationships between data elements all follow a strict logical structure. The MySQL RDBMS is a set of software tools used to implement, manage, and query relational databases.

C. PostgreSQL

PostgreSQL is an open source Object Relational Database Management System (ORDBMS), meaning that, in addition to storing relational data, it is able to store complex and NoSQL objects such as JSON, xml, blobs and more. PostgreSQL is richer in features compared to MySQL and handles larger datasets better, however in certain application MySQL is preferred due to its simplicity.

D. Use Cases for Relational Databases

Relational Databases ensures data integrity and excel at performing complex queries. Their use cases include applications that require strong data integrity and business models that perform transactions. The TPC-DS dataset is organised in structured tables, using the relational model, and thus a Relational Database is better suited for it. The TPC-DS dataset will be discussed in further detail in the next section.

E. Accumulo

Accumulo is an open source distributed Key Value Store that provides robust, scalable data storage and retrieval. Internally, Accumulo stores data in key value pairs. The key consists of three parts: the row ID, the column family and the column qualifier, making searches efficient and enabling the grouping of key value pairs within a table. Accumulo is a NoSQL database meaning that non structured data can be directly stored without needing any sort of transform, however this also means that it cannot be queried using any query language and subsequently requires an application to be built on top of it in order to parse the data it contains.

F. Use Cases for Key Value Stores

Key Values Store Databases are most commonly used for applications that require quick data retrieval such as managing session data in applications, storing logs etc. Internally, data is treated as blobs, meaning that it cannot be prepossessed and filtered before being sent to an application. In practice, this means that queries which require filtering on certain value, range queries an queries requiring joins between large tables would potentially perform poorly as the whole table would have to be sent to the Trino workers before being filtered, instead of being filtered locally. Despite Key Value Store Databases being better suited to Big Data and OLAP applications, since in our use case joins are inevitable, it would benefit query performance to store smaller tables in Accumulo so that performance is not throttled by the network bandwidth while transporting large amounts of data between Trino nodes.

III. Methodology

A. The TPC-DS Benchmark

TCP-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 the System Under Test's performance as a general purpose decision support system. As discussed before, the System Under Test in this paper is Trino [1].

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

- Record customer purchases (and track customer returns) from any sales channel
- Modify prices according to promotions
- Maintain warehouse inventory
- Create dynamic web pages
- Maintain customer profiles (Customer Relationship Management)

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

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

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

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

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

B. Data Partitioning

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

1) Naive Partitioning: An initial approach to distributing the data would be to equally divide the gigabytes of data among the three databases. In practice, this would mean that each database would store tables that are not related to each other as the sales and returns fact tables of the same category

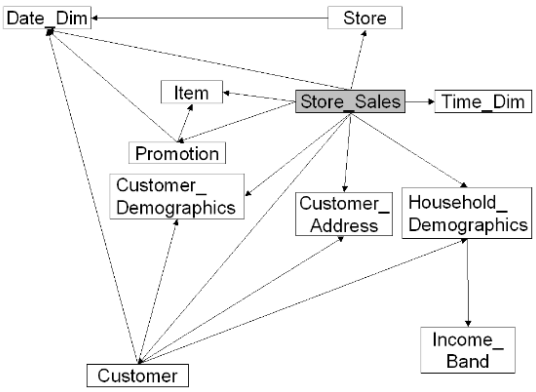

Fig. 1. Store Sales ER Diagram.

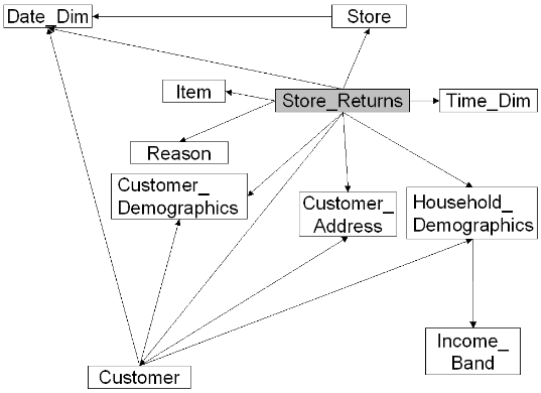

Fig. 2. Store Sales ER Diagram.

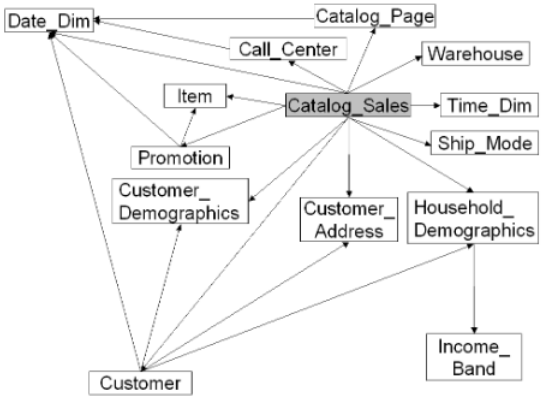

Fig. 3. Catalog Sales ER Diagram.

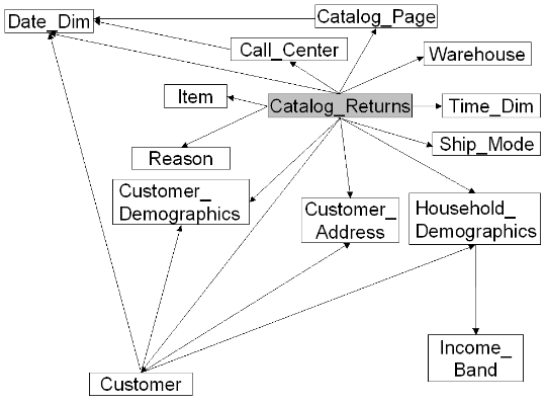

Fig. 4. Store Sales ER Diagram.

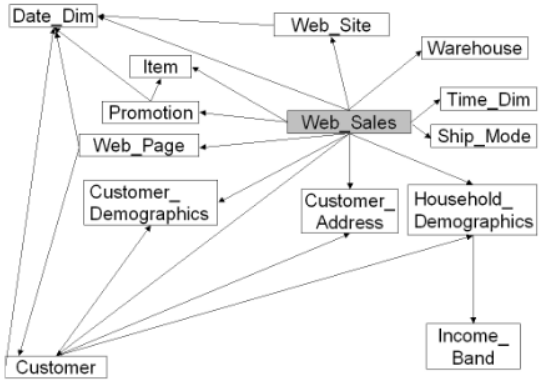

Fig. 5. Web Sales ER Diagram.

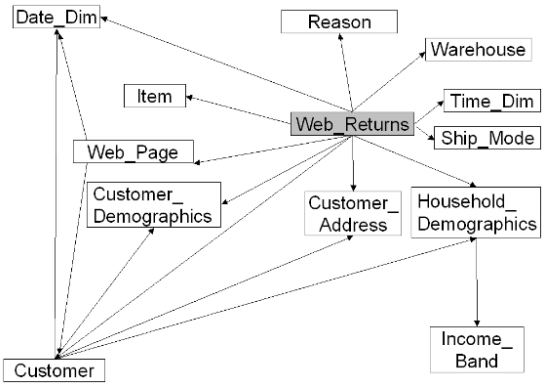

Fig. 6. Web Sales ER Diagram.

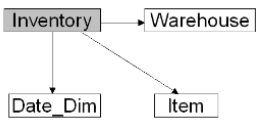

Fig. 7. Inventory ER Diagram.

tend to have similarly large sizes and would subsequently be split up. Subsequently, performing queries joining two tables would require data transfers between nodes. Our data is organised in a snowflake schema meaning that queries requiring a join between tables are very common. Bandwidth is a well known and documented bottleneck [3] in the Big Data research field, thus this approach would result in significant performance issues. Based on the issues mentioned above, we deem this method of partitioning sub optimal and thus will not include it in the performance measurement experiments.

2) Proximity Partitioning: In order to combat the performance issues that would arise from our initial approach, we will distribute the data among the three databases based on the data proximity between tables, while simultaneously trying to keep the amounts of data in each database close. The sales and returns table of each category will be stored in the same database along with the tables that are only used only by the corresponding category (e.g. the table store will be stored in the same database as the tables store_sales and store_returns). We will choose which category will be stored in each database based on the performance of each database in the baseline experiment. The experiment process will be discussed in further detail in a subsequent section. The fact table inventory table and the rest of the tables included in its snowflake ER diagram will be stored in the database with the least amount of data after the initial distribution of the sales and returns fact tables. The remaining tables are common to all categories and are thus stored based on size. This approach ensures that joins and filtering of data can take place locally in each node so as to decrease the amount of data exchanged between nodes as much as possible. The data partition produced by this method will be used in the following sections to test the Trino optimizer and to make optimisations based on query performance.

3) Partitioning Based on Query Optimization: While our previous approach is a significant improvement to the naive method, the tables that are common to every category are still divided in a somewhat arbitrary way. This approach will expand on and optimise the method of proximity partitioning through experiments. The performance of the previous data partition will be tested by a set of representative queries. These queries and their performance will be analysed in order to determine which tables are stored efficiently and which are stored inefficiently. The resulting data partition will simultaneously ensure data proximity and optimal query performance.

IV. System deployment

A. System description

In this work, all the experiments are carried in a cluster of 3 virtual machines as described in Table I. Due to shortage of VMs, we are forced deploy the DBMSes in nodes which will act also as Trino nodes, since running two DBMSes in the same node was prohibitive, because it would lead to bottlenecks in bandwidth during the data movement to the Trino workers and also in memory. Thus, this can potentially lead to underutilization since the memory of each node will have to be shared between the database system and the Trino worker. On the other hand it has the desirable property of eliminating the need for data movement in the case where the data in the local database will be processed by the Trino node in the same VM. As it will become evident from the following experiments, this is something often exploited by the cost based optimizer and leads to overheads where it is not.

TABLE I
CLUSTER RESOURCES AND CONFIGURATION

Node	cores	system RAM	disk size
1	4	8 GB	30 GB
2	4	8 GB	30 GB
3	4	8 GB	30 GB

Node	Role	Additional processes	RAM devoted to Trino
1	coordinator+worker	Accumulo	4 GB
2	worker	MySQL	4 GB
3	worker	Postgres	4 GB

B. Data generation

The data generated for this work is provided exclusively by the TPC-DS Benchmark, ensuring consistency across databases and real-world type data. We opted for a factor of 10 GB for our datasets, which given our resources, was determined to be a representative amount of data. Since we are dealing with a data volume that is large, in relation to our Virtual Machines' disk size, actions have been taken, to generate the dataset one table at a time, so as not to stress the Virtual Machines' storage with the whole dataset and the populated database, at the same time.

C. Data loading

After each table is generated, it is further divided into 'chunks', to ensure that the loaders do not face bandwidth bottlenecks. The sharded data is then used by the respective loaders, to populate each database, independently of Trino. In every case the loaders are custom made for the specific database, whilst ensuring compatibility with the formats that Trino requires.

D. Deployment

In relation to the deployment of the underlined structures required for this work, we provide a brief overview of the technologies used and their versions. Even if the same results can theoretically be extracted by different stack configurations, compatibility issues may arise. For more details on the deployment visit the project's GitHub repository.

1) DBMS Deployment: At first, we downloaded and installed the necessary triad of databases to be used when benchmarking Trino:

- MySQL: Version '8.0.34' is used, the latest Long Term Support version.
- PostgreSQL: Version '22.04' is used, which is also the latest version (LTS).

- Accumulo: Version '1.10.3' is used, chosen for compatibility with the rest of the stack.

2) Additional technologies:

- JAVA: 'JDK-17' is used, due to its compatibility with "Okeanos" servers.
- Apache Hadoop: Version '3.0.3' is used.
- Apache-ZooKeeper: Version '3.6.4' is used, as it is the oldest version to be compatible with JDK-17.

3) Trino Deployment: Trino version '422', released on 13 July 2023, has been used since it is one of the latest releases and is compatible with our stack. Trino is based on JAVA, therefore the installation of 'JDK-17' is necessary prior to the Trino Installation.

V. Evaluation

In this section we describe the designed experiments, highlighting the underlying rationale which lead us into each one, while in the next section we present, interpret and discuss our results.

A. Baseline partitions vs proximity partition

As a first experiment we try to qualify the effectiveness of the proximity partition described in section III-B. To do so, we compare it with 3 baseline partitions, each containing all the data in one of the 3 databases and with Trino running on 3 worker nodes. For the queries we choose to run, we select a small subset from the TPC-DS queries satisfying the following conditions:

- cover the entire schema, in order to be able to reason about the location of each table across the 3 stores
- exhibit low memory utilization, since the infrastructure at Okeanos was proven unable to handle the most demanding queries of TPC-DS at the current data size of 10GB
- utilize a wide variety of functions such as aggregates and joins

These requirements are summarized on table II for each of the chosen queries.

B. Proximity partition vs Optimized partition

After conducting the appropriate experiments on the proximity partition, we will form a group of queries, called bottleneck queries, consisting of all the queries who had poor CPU utilisation, used large amounts of system memory and queries whose execution stalled. After carefully examining the query plans of the bottleneck queries, we will identify which execution stages caused the query to perform poorly. By the end of this process we will be able to pinpoint which tables have not been distributed correctly and subsequently re distribute them to a database which will be able to handle them better. As a result, a partition that simultaneously takes into account both proximity and query performance will be created.

C. Scalability

Having shown that the query optimization partition -with insight of the query operation- is superior to the proximity one, we continue into measuring the scalability of the cluster for that partition. Keeping data size the same (10GB) we scale horizontally by measuring performance of the same queries with 1, 2 and 3 Trino workers. For the executions with 2 workers, we choose to deactivate the worker node which hosts also the PostgreSQL DBMS, since as we will discuss in the next section, from the previous experiments it is clear that this database exhibits higher bandwidth and hence it is a effective choice to let PostgreSQL be the DBMS which will be moving data across the network to supply one of the remaining workers. Correspondingly, for the executions with 1 worker we deactivate the nodes which host PostgreSQL and Accumulo, again to avoid forcing Accumulo to transmit data to the workers, sacrificing performance only to a small extend and only because coordinator and worker will be sharing RAM of the same node.

D. Optimizer efficiency across different partitioning

To evaluate how the cost based optimizer behaves across different data partitioning schemes one must take into account the interwind between optimizer estimation and data distribution. Trino's cost optimizer fetches metadata from the connectors, based on which forms an optimized distribution query execution plan. This means that the choice of data location and store will impact the operation of optimizer as it will lead to different query plans. Exactly that, is a second way that data partitioning affects query execution, apart from the evident one, which is that the query or table data format might benefit or not from the data model of the store. To test if the optimizer produces effective query plans we need to study these 2 ways independently. For that, we choose to measure as a metric related to the optimizer the number of produced shuffles (i.e. number of data transfers between 2 stages which happen buffered, in memory allowing exchange of data) and not some other performance related metric (such as query latency) since that would be both affected by the optimized plan and the store, while our goal is to separate them.

E. System performance with various optimizer options

Additionally, for a complete study of the cost optimizer we consider necessary to show that it actually contributes in better performance. Hence, in this experiment we measure performance related metrics (query latency, bandwidth, total cumulative memory transferred) shifting the option for partioned of broadcasted hash tables during a hash join. Trino uses hash joins for performing joins between tables, that is, an algorithm which works by building a hash table of the one table and probing against it the other table. Depending on whether the hash table will be replicated across all the nodes or split, this can lead to improved performance and so by default, Trino's optimizer inferes the best of the two strategies automatically, since this is something that changes for each query/table

TABLE II
TABLES USED FOR QUERIES Q1, Q12, Q15, Q86, Q90, Q9, Q62, Q80, Q84 AND Q91

	Q1	Q12	Q15	Q86	Q90	Q9	Q62	Q80	Q84	Q91
store_sales						✓		✓		
store_returns	✓							✓	✓	
catalog_sales			✓					✓		
catalog_returns								✓		✓
web_sales		✓		✓	✓		✓	✓		
web_returns								✓		
inventory										
store	✓							✓		
call_center										✓
catalog_page								✓		
web_site							✓	✓		
web_page					✓					
warehouse							✓			
customer	✓		✓						✓	✓
customer_address			✓						✓	✓
customer_demographics									✓	✓
date_dim	✓	✓	✓	✓		✓	✓	✓		✓
household_demographics					✓				✓	✓
item		✓		✓				✓		
income_band									✓	
promotion								✓		
reason						✓				
ship_mode							✓			
time_dim					✓					

because smaller tables might benefit more from broadcasted hashes, but without divide-and-conquer hash building, joins for bigger datasets would be impossible. For these reasons, we execute all queries for the query optimized partitioned with both strategies (always broadcasted and always partitioned) to investigate whether the optimizer with the automatic strategy makes always the best choice.

VI. RESULTS AND ANALYSIS

In this section we will present the results of our experiments and evaluate the performance of Trino and its optimizer for various different data partitioning methods, numbers of worker nodes and optimizer options.

A. Experiment 1 - Baseline partitions vs proximity partition

From Fig. 8 and Fig. 9 we can see that in our baseline experiments PostgreSQL outperforms both other databases with Accumulo trailing behind both PostgreSQL and MySQL for most queries. The reason why Accumulo performs so poorly is that since, internally, data is treated as blobs. This means that data cannot be filtered before being sent to a Trino worker, resulting in large amounts of data being exchanged between nodes. Accumulo combats this by distributing its data across various nodes enabling parallel processing, however due to the constraints of our servers Accumulo runs in standalone mode and is thus slower. MySQL performance is not ideal for all queries. Adding indexes to whose processing tables causes stalls could significantly improve performance, however identifying the optimal indexing strategy for MySQL is outside of this paper's scope. Overall, querying partition 1 (proximity partition) using Trino outperforms MySQL or Accumulo and in some cases even PostgreSQL. This result is very satisfactory as Trino manages to overcome the bottleneck of the slowest database while, in some cases, also managing to outperform the quickest database by employing parallel processing.

B. Experiment 2 - Proximity partition vs optimized partition

Using the data we extracted from our first experiment well fine tune the proximity partition in order to devise the optimal data partitioning method for our system configuration and

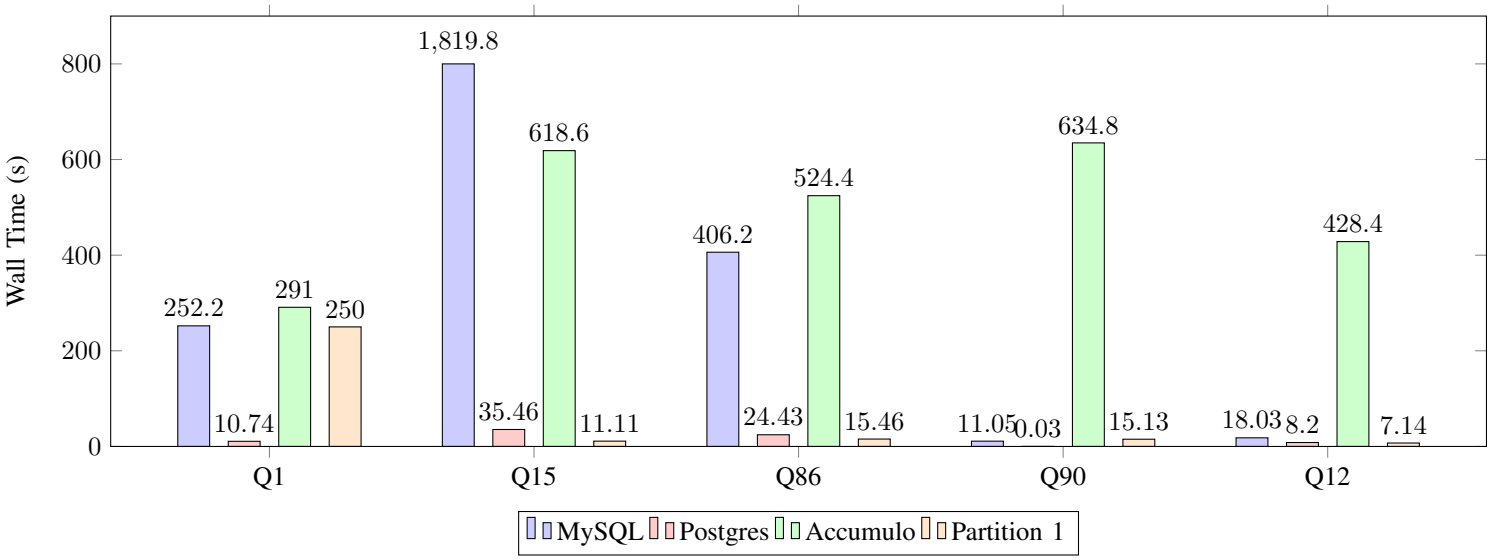

Fig. 8. Wall Time Comparison across Partitions for Queries Q1, Q15, Q86, Q90, and Q12

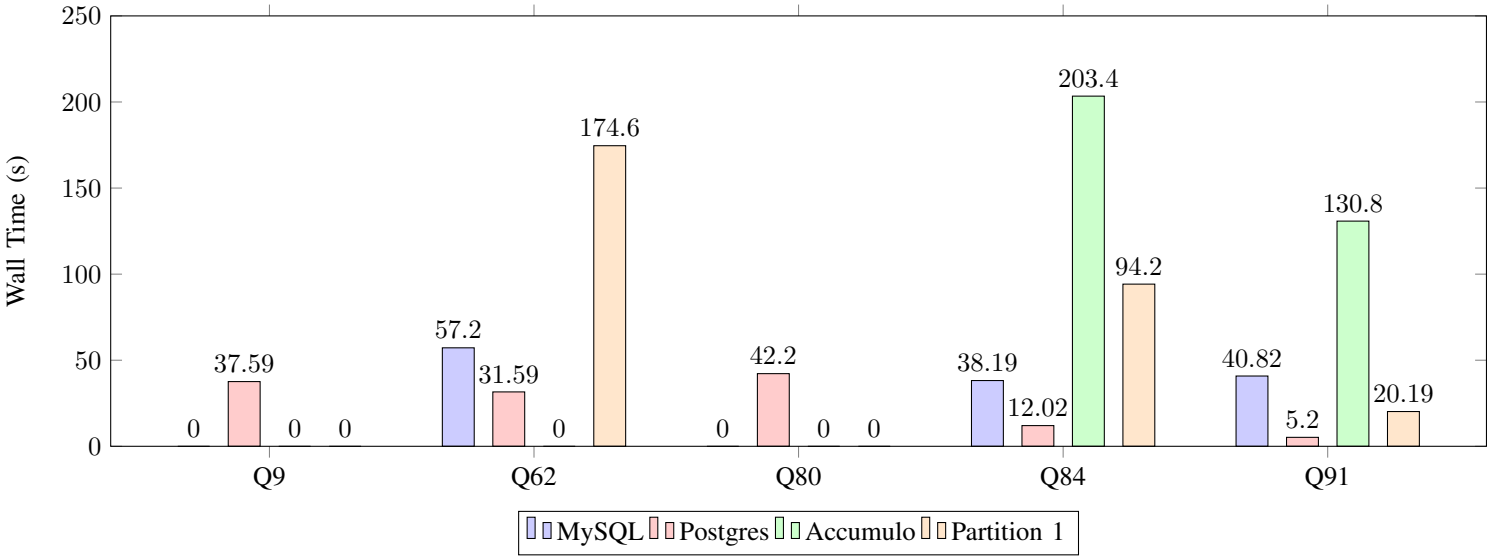

Fig. 9. Wall Time Comparison across Partitions for Queries Q9, Q62, Q80, Q84, and Q91

dataset. In more detail, using the Trino UI we individually examine each bottleneck query. Using the Stage Performance tab we can identify the problematic stages, which stall or use too much RAM. After, using the Live Plan tab we extract information about data transfers between nodes, filtering processes and tables joins within nodes for each stage. By the end of this process we can identify the sub query, the table and the database that cause the problem. In table III we can see the optimal partition compared to the proximity partition. From Fig. 10 and Fig. 11 we can see that, with the exception of query 1, the optimal partition outperforms the proximity partition and is even able to execute queries that failed because of excessive RAM use for the proximity partition. For query 1, partition 2 performs worse compared to partition 1 because in order to optimise the overall partition we had to sacrifice the data proximity for some of the tables used in query 1. From Fig. 12 and Fig. 13 we can also see that partition 2 also results in better memory management as the partition design takes into account both the proximity of the data as well as the processing strengths of each database.

TABLE III
TABLES PLACED IN EACH PARTITION

Name	Partition 1			Partition 2		
	A	P	M	A	P	M
store_sales	✓				✓	
catalog_sales		✓				✓
inventory		✓			✓	
web_sales			✓		✓	
store_returns	✓				✓	
catalog_returns		✓				✓
customer_demographics			✓			✓
web_returns			✓	✓		
customer			✓			✓
customer_address			✓			✓
item		✓			✓	
date_dim	✓				✓	
time_dim	✓				✓	
catalog_page		✓				✓
promotion	✓			✓		
household_demographics	✓			✓		
store	✓				✓	
web_page			✓	✓		
web_site			✓	✓		
call_center		✓				✓
warehouse		✓			✓	
reason	✓			✓		
ship_mode			✓			✓
income_band	✓			✓		

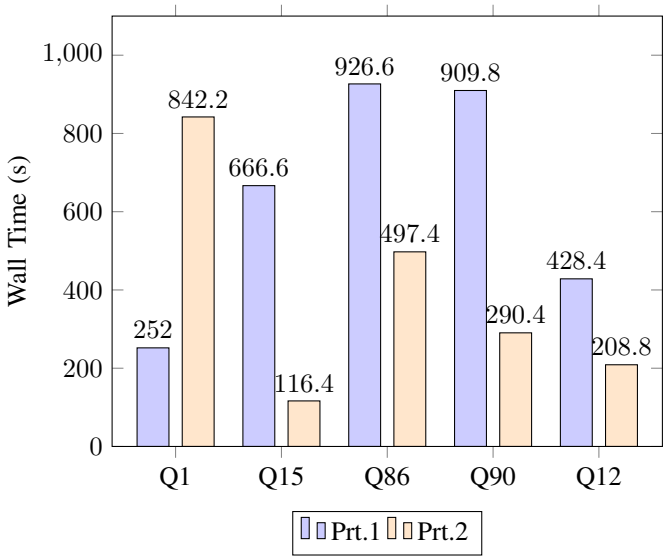

Fig. 10. Wall Time Comparison for Proximity and Optimized Partition

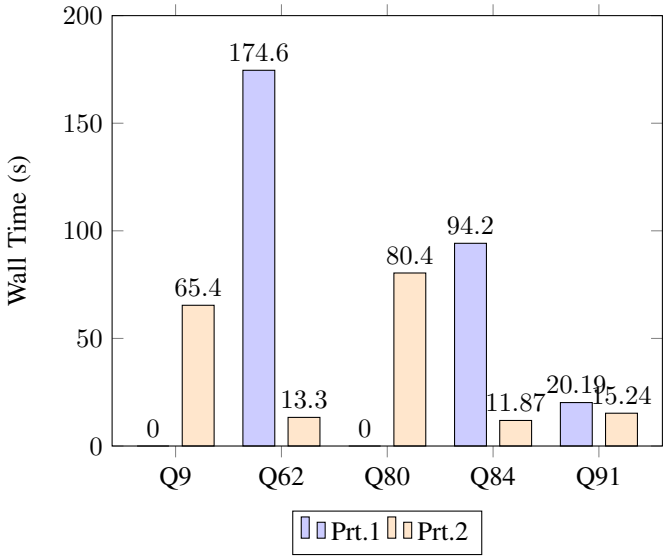

Fig. 11. Wall Time Comparison for Proximity and Optimized Partition

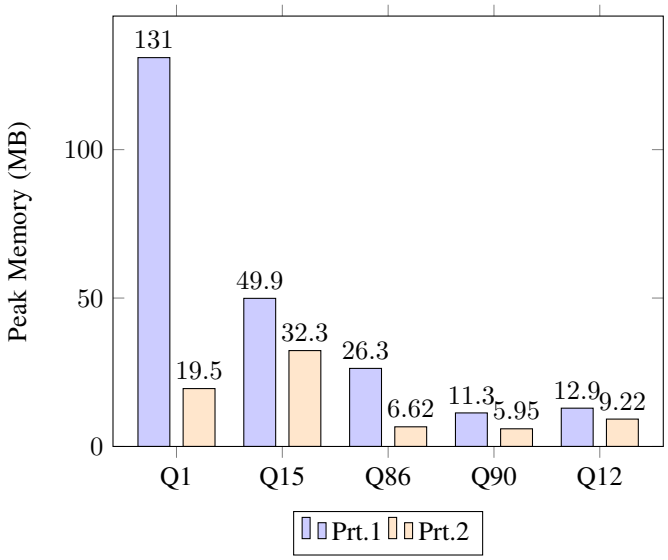

Fig. 12. Peak Memory Comparison across Partitions for Queries Q1, Q15, Q86, Q90, and Q12

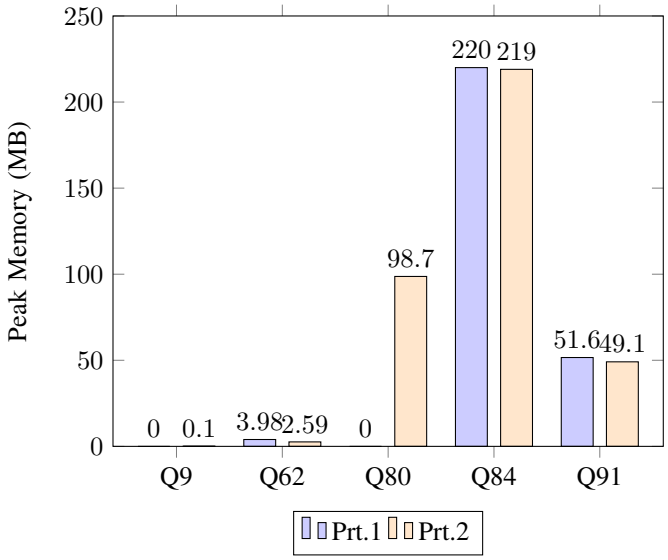

Fig. 13. Peak Memory Comparison across Partitions for Queries Q9, Q62, Q80, Q84, and Q91

C. Experiment 3 - Scalability

For scalability experiments as shown in Figure 14 the general trend is the expected one, that is, increasing the number of Trino workers improves performance by reducing query latency. In some cases such as in the queries 1 and 86, including only 2 workers leads to better performance than using 3, which can be explained as the overheads of creating and maintaining the extra worker reduce the available memory of the coordinator (which is also a worker, so it is underutilized). Additionally, in these abnormal cases, the latency tied to the network communication of splits between stages on different workers or the limited parallelism available in the query semantics also can disfavor the 3 worker configuration against the 2 worker one.

TABLE IV
OPTIMIZER METRICS IN RELATION TO PARTITIONING: NUMBER OF SHUFFLES AND ROWS TRANSFERRED

		q86	q15	q12	q1	q90
Prt.1	splits	6	6	6	12	14
	rows	105.5K	854.2K	80.4K	2.1M	1.3M
Prt.2	splits	4	5	4	6	12
	rows	102K	750K	54.1K	218K	12.7K

D. Experiment 4 - Optimizer efficiency across different partitioning

From Table IV it is clear that the Optimized Partition (*Prt.2*) reduces the number of shuffles exchanged between stages as well as the total number of rows transferred, for all queries. The number of shuffles and the number of stages are both dictated by the cost based optimizer as part of the optimized the query plan and because high number of shuffles incurs performance overheads, consume buffer and user memory and add additional latency, the optimizer targets at keeping them low [**?**]. *This however is a tradeoff*, because low number of shuffles means no parallelism in query execution which also impacts performance negatively. What we deduce from this experiment is that with an improved partitioning such as the Optimized Partition, the optimizer is able to reduce the number of shuffles and transferred rows while maintaining (or in our case, compared to Proximity Partition, *improving*) query latency, cpu overheads and bandwidth. Therefore, this is a second way that the data partitioning affects performance, through enabling better optimizing of query plans.

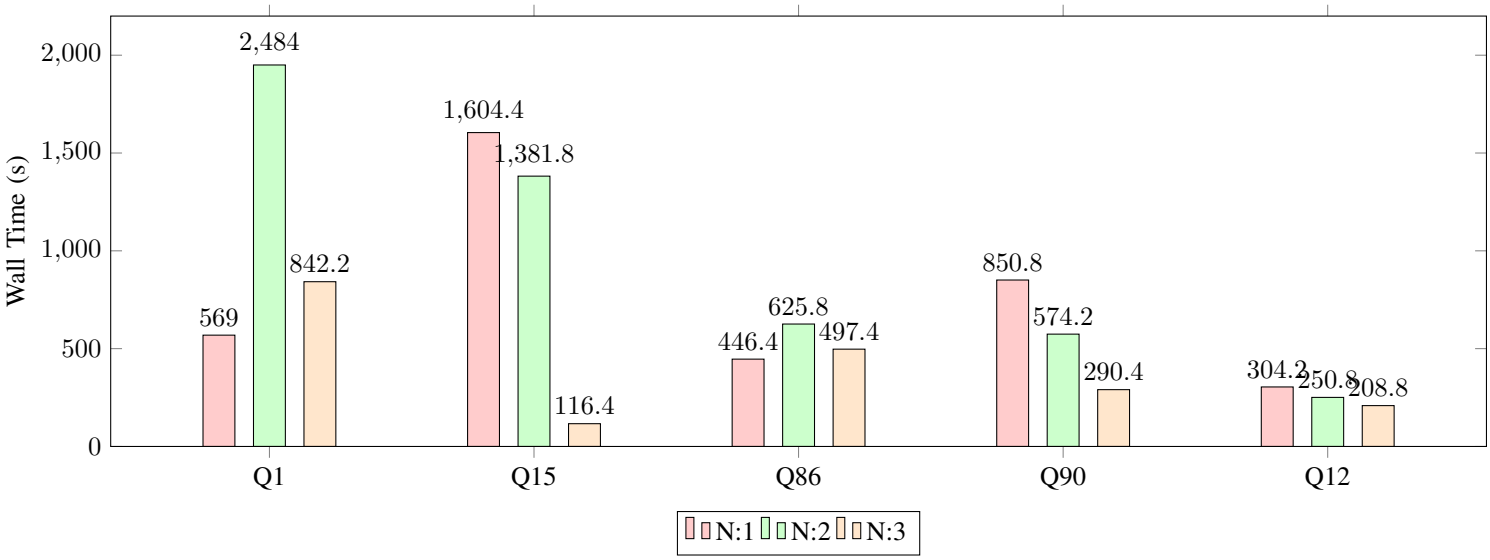

Fig. 14. Wall Time Comparison while scaling Partition 2.

E. Experiment 5 - System performance with various optimizer options

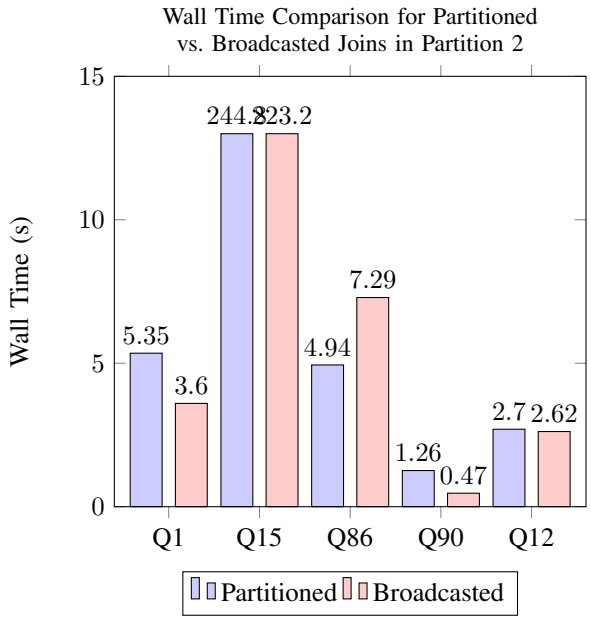

Fig. 15. Wall Time Comparison for Partitioned vs. Broadcasted Joins for Queries Q1, Q15, Q86, Q90, and Q12

In figures 15 and 16 we plot query latency and maximum required memory for each query (for the Optimized Partition and 3 Trino workers), while varying the hash join table distribution between ”distributed” and ”broadcasted”. One can immediately observe that while no choice between the 2 is uniformly better across all the queries for query latency, regarding memory utilization, broadcast joins are superior. These results are justifiable: we expect partitioned joins to be generally slower, since nodes need to communicate in order to request hashes for table parts they do not have locally. However in Figure 15 most queries exhibit smaller latency for partitioned joins because they take advantage of parallelism. In Figure 16, broadcasted joins display lower memory utilization since the smaller table is always the one being transmitted to the workers.

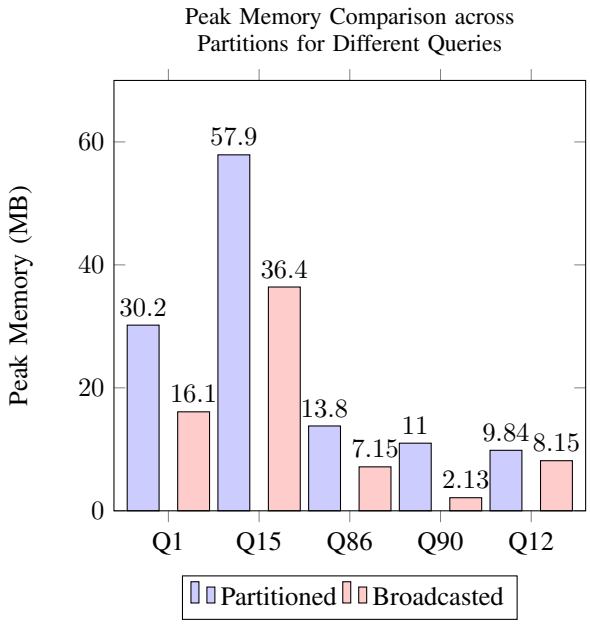

Fig. 16. Peak Memory Comparison across Partitions for Queries Q1, Q15, Q86, Q90, and Q12

VII. Conclusions and Further work

To summarise, through our experiments we measured Trino’s performance and scalability as well as its built in optimizer for different data partition methods. We found the

results very satisfactory, especially when using an optimised data partitioning method. Trino is a powerful tool for querying non uniform data sources that overcomes the bottlenecks of its underlying databases and, in some cases, even manages to outperform the fastest one through the utilisation of parallel processing. Further work could include scaling Trino to include more nodes and even bigger datasets, as well as evaluating its performance for different databases and datasets.

REFERENCES

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

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

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