International Journal of Computer Techniques–IJCT Volume 9 Issue 3, May - 2022

Open Access and Peer Review Journal ISSN 2394-2231 https://ijctjournal.org/

Trino as a Unified Query Layer for Heterogeneous Data Sources: Survey and Benchmarks

Kuladeep Sandra

Independent Researcher

kuladeepsandra90@gmail.com

Abstract*—Modern enterprises store data across heterogeneous systems relational databases, data lakes, cloud warehouses, message brokers and the cost of moving that data into a single physical store is rarely justifiable. Federated query engines that read each source in place have therefore become a foundational layer of enterprise data platforms. This paper surveys the federated query landscape and presents production operating experience with Trino across 14 heterogeneous data sources in a banking and insurance environment. The paper addresses three questions: how Trino's architecture enables federation across diverse backends; what the operational realities are when running Trino in production at enterprise scale; and how practitioners should think about its strengths and limits. Two non-obvious findings shape the operational picture. First, Trino's container cold-start time on Kubernetes is nearly 30 seconds, which makes it unsuitable for sub-second interactive dashboard SLAs without architectural workarounds. Second, network topology dominates query performance for data-intensive queries: relocating Trino worker pods onto racks physically closer to the source data improved data-intensive query latency by close to 40 percent. The thesis is that Trino is a powerful federation layer for ad-hoc analytics, self-service discovery, and cross-source joins, but it is not a magical query everything equally fast engine; it requires understanding per-connector trade-offs, careful memory and topology tuning, and clear architectural decisions about which use cases it serves and which it does not.

***Index Terms*—Trino, federated queries, query federation, multi-source, SQL analytics, query engine**

I. INTRODUCTION

The modern enterprise data landscape is plural. A bank may run an Oracle database for the operational core, Postgres for several departmental systems, an on-premises Hadoop cluster for historical archives, an Iceberg lakehouse on S3 for newer analytical workloads, Snowflake or BigQuery for specific business unit needs, and Kafka for event streams. None of these systems is going away, and consolidating them into a single store is rarely cost-justified. The pragmatic alternative is a federated query layer that reads each source in place and exposes a single SQL interface to the analyst, the data scientist, and the BI tool.

Trino the project formerly known as PrestoSQL, forked from the original Presto in 2019 has become the dominant open-source federated query engine. Its connector architecture supports more than 50 data sources out of the box. Its SQL dialect is reasonably standard and reasonably complete. Its query coordinator handles cross-source joins, predicate pushdown, and distributed execution across worker nodes. In our environment, Trino is the front door for ad-hoc analytics across 14 different sources, and it is the engine that powers several classes of workload that no single backend could serve cleanly.

This paper documents what we learned from running Trino in production, including the parts that the documentation does not emphasize. It addresses three questions:

RQ1. How does Trino's architecture enable federated query execution across heterogeneous data sources, and what are the implementation patterns that make federation viable in practice?

RQ2. What are the operational realities of running Trino at enterprise scale cold-start, memory management, topology sensitivity, connector maturity, and the gotchas that surface only in production?

RQ3. How should practitioners think about Trino's strengths and limits, and what is the right mental model for deciding which workloads belong on it and which do not?

The paper is organized as follows. Section 2 covers federated query background and related work. Section 3 presents Trino's architecture. Section 4 documents the production case study and the lessons. Section 5 presents informal benchmarks. Section 6 covers the tool selection framework. Section 7 concludes.

II. BACKGROUND AND RELATED WORK

A. The Federation Problem

Federated query systems have a long history in the database literature, going back at least to the 1980s with Multibase and similar projects. The conceptual framework a global schema mapped to local schemas, query rewriting that pushes work to the source systems where possible, materialization of intermediate results in a central engine is well established. What has changed in the last decade is the

diversity of source systems (the explosion of NoSQL stores, cloud-native databases, and event streams) and the volume of data involved (terabyte-scale federated joins were exotic in 2005 and routine in 2022).

B. Presto and Trino

Presto was originally developed at Facebook around 2013 to enable interactive SQL queries against the company's Hadoop-based data warehouse. The project was open-sourced and evolved through several stages: the original PrestoDB at Facebook, a community fork called PrestoSQL led by the original creators in 2019, and a subsequent rename to Trino later that year after a trademark dispute. Trino is the project this paper discusses; PrestoDB also continues to exist as a separate project.

Trino's architecture is well-described in Sethi et al.'s 2019 ICDE paper [1]. The high-level design has not changed substantially: a coordinator process that parses, plans, and schedules queries; worker processes that execute query stages; and connector implementations that handle the source-specific work of reading data and pushing predicates down. The connector pattern is the key abstraction that enables federation.

C. Alternatives

Trino is not the only federated query engine. Apache Drill, Apache Spark SQL with custom data sources, and the various cloud-vendor offerings (Athena on AWS, Synapse Serverless on Azure, BigQuery Omni on GCP) all occupy similar niches. The choice between them is largely driven by ecosystem fit. We chose Trino primarily because of its connector ecosystem and its open-source posture, which made it portable across our hybrid deployment.

III. TRINO ARCHITECTURE

A. The Coordinator-Worker Model

A Trino cluster has one coordinator and many workers. The coordinator receives SQL queries, parses them, plans the execution as a DAG of stages, and schedules the stages onto workers. Workers execute their assigned stages, exchange intermediate results with each other when needed, and return final results to the coordinator. The model is similar in shape to other distributed query engines (Spark SQL, Presto, Impala), but Trino's specifics differ in detail.

B. The Connector Pattern

Each data source is accessed through a connector a plugin that implements a small interface for metadata discovery, predicate pushdown, and data scanning. The connector translates Trino's internal representation of a query against a table into the source-specific operations that retrieve the data. For a Postgres connector, the translation is straightforward: most of Trino's SQL maps cleanly to Postgres SQL, and most predicates can be pushed down as WHERE clauses. For a Hive or Iceberg connector, the connector knows how to read Parquet and ORC files from object storage, how to interpret partition layouts, and how to use table-format metadata for pruning. For a Kafka connector, the work is different again translating SQL queries into consumer reads with appropriate offset handling.

The quality and completeness of connectors varies widely. The Postgres, MySQL, and S3-based connectors are battle-tested and reliable. Some of the connectors for newer or less-common sources are functional but rough pushdown is incomplete, type mappings have edge cases, performance is unoptimized. Treating all connectors as equivalent is a mistake; treating them as a spectrum with known reliability differences is the right mental model.

C. Predicate Pushdown

The single most important determinant of federated query performance is predicate pushdown. A query that filters a 10-billion-row source table and joins the filtered result with a smaller table will perform well if the filter pushes down to the source (where 10 million rows are returned to Trino) and badly if it does not (where 10 billion rows are pulled into Trino's memory). Knowing which predicates push down for which connectors is essential operational knowledge, and it is not always obvious from documentation.

D. Memory Management

Trino's memory model is strict. Each query has a memory limit; if the query exceeds it, the query fails rather than spilling to disk. (Spill-to-disk is supported but is off by default for some operations and is slow when it kicks in.) For analytical workloads with large intermediate results multi-way joins, large aggregations, distinct counts memory tuning is essential. The default memory configuration is conservative and most production deployments need to increase it.

IV. PRODUCTION CASE STUDY

A. Context and Sources

Our Trino deployment serves a 30-engineer data team across 6 business units. The 14 data sources we federate over include: four relational databases (Postgres for two operational systems, Oracle for the legacy core banking system, MySQL for a departmental system); three data lake locations (an Iceberg lakehouse on S3, a legacy Hive warehouse on on-premises HDFS, a partner data drop on a separate S3 bucket); two cloud data warehouses (Snowflake for one business unit's reporting, BigQuery for another); two Kafka topic groups exposed as queryable tables; and three smaller specialty stores (Elasticsearch for one application's logs, MongoDB for a document store, a custom JDBC source for a vendor system). Trino runs on Kubernetes on a 32-node bare-metal cluster.

B. The Cold-Start Discovery

We deployed Trino with the assumption that it would serve interactive dashboards with sub-second SLAs. The coordinator would receive a query, dispatch it to workers, and return results within the SLA. Worker pods would scale up and down based on load. The plan failed the day we put it under real interactive load.

The issue was Trino container cold-start time on Kubernetes. A Trino worker pod takes about 30 seconds to start from the moment Kubernetes schedules it: the JVM startup, the Trino runtime initialization, the connector loading, the registration with the coordinator. For a long-running batch query, 30 seconds is a rounding error. For an interactive dashboard refresh that needs sub-second response, 30 seconds means that any query arriving when no warm worker pods exist gets a 30-second penalty before it even starts executing. Auto-scaling Trino workers to zero between queries which would have been the cost-efficient default was therefore incompatible with interactive SLA.

The workaround was to keep a baseline pool of warm worker pods always running, sized for the expected concurrent interactive load, and to use auto-scaling only above the baseline. This costs more in steady-state compute than the auto-scaling-to-zero model would have, but it eliminates the cold-start penalty for the common case. It is an acceptable trade-off, and it is the kind of trade-off that is invisible until production.

C. The Network Topology Discovery

The second non-obvious discovery was about network topology. Some of our most data-intensive queries full-table scans of multi-terabyte Iceberg tables, large joins between Hive tables and S3 tables were running slower than the underlying storage should have allowed. The bottleneck was network I/O between the Trino worker pods and the storage backends.

The Kubernetes scheduler places pods on nodes based on resource availability and various affinity rules, but it does not by default consider physical rack topology or proximity to specific storage backends. Our Trino workers were being placed on whichever nodes had capacity, including nodes that were several rack hops away from the on-premises Iceberg storage. The network path from a worker pod on rack 7 to storage on rack 2 went through multiple switches and consumed bandwidth that pods on rack 2 itself would not have needed.

We added node affinity rules that placed Trino workers preferentially on nodes physically close to the storage backends they queried most. The improvement on the affected queries was approximately 40 percent latency reduction queries that had been taking 45 minutes started running in 25 to 27 minutes. The change was invisible in any architectural diagram and would never have been found by anyone who was not paying attention to the actual network paths the data was traversing.

The lesson generalizes beyond Trino: in any distributed system that moves significant amounts of data between compute and storage, physical topology matters and the scheduler does not know about it unless you tell it.

D. Connector Reliability

Our connector experience over two years confirms the spectrum view. The Postgres and S3/Iceberg connectors have been essentially trouble-free; we treat them as production-grade. The Hive connector has been reliable for the legacy HDFS workloads but requires careful predicate-pushdown verification for joins that span Hive and Iceberg tables. The Snowflake and BigQuery connectors work but have rough edges around data type mapping and predicate pushdown for complex types. The Kafka connector is functional but is the most operationally demanding because Kafka's offset semantics interact with Trino's query model in ways that are not always intuitive. The smaller specialty connectors (MongoDB, Elasticsearch, the custom JDBC source) have all required at least one round of debugging and tuning before they were stable enough for self-service use.

E. Memory Management in Practice

Out-of-memory query failures were the most common operational issue in the first six months of production. The default memory limits were too low for our analytical workloads. We tuned per-query memory limits up substantially to several gigabytes per query for typical workloads and to tens of gigabytes for the largest known queries. We added a query classification layer that routed obviously-large queries to a separate worker pool with higher memory limits, so that the classification could happen at submission time rather than after the query had failed once. The combination reduced OOM failures from several per day to a handful per month, and the remaining failures were almost all queries that were genuinely pathological and needed human review rather than more memory.

F. Observability

We expose Trino's per-query metrics through the standard JMX integration into Prometheus and Grafana. The dashboards we found most useful: per-query latency distributions broken down by source, predicate pushdown success rates per connector, memory consumption per query (with alerting on queries approaching their limit), and worker pod CPU and network utilization. The metric we wish Trino exposed more cleanly is per-source data volume scanned, which would let us measure the impact of pushdown improvements directly.

V. INFORMAL BENCHMARKS

A. Caveats

Vendor-published benchmarks for query engines are notoriously misleading because they tend to use synthetic workloads and ideal configurations. The numbers below are

from our actual production workloads, with all the variability that implies, and they should be treated as illustrative rather than definitive.

B. Query Categories

We classify our Trino workloads into four categories. Interactive dashboard queries: typically scanning small fact tables with predicate pushdown to the source, returning small result sets. Median latency in the 200-millisecond to 2-second range when warm workers are available. Self-service ad-hoc queries: analyst-written queries against medium-sized tables, often spanning two or three sources. Median latency in the 5-second to 60-second range. Large analytical queries: full or partial scans of multi-terabyte tables, often with significant aggregation. Median latency in the 5-minute to 30-minute range, occasionally longer for the heaviest queries. Cross-source joins: queries that join data from two or more federated sources, which is the use case Trino is most uniquely suited for. Latency varies enormously depending on whether the joins can be pushed down and how much data must be materialized in Trino itself.

C. The Pushdown Story

For interactive and ad-hoc queries, the difference between a query that pushes down filters and a query that does not is often the difference between a 2-second response and a 2-minute response. Educating analysts about predicate pushdown and providing templates for the queries that pushdown reliably was one of the highest-leverage things we did for query performance.

VI. TOOL SELECTION FRAMEWORK

When does Trino make sense as the federation layer? When the data sources are genuinely heterogeneous and consolidating them is not feasible, when the workloads are interactive or ad-hoc rather than scheduled batch, when SQL is the right interface for the consumers, and when the team has the operational capacity to tune Trino properly. When does it not? When all the data already lives in a single backend that has its own query engine, when the workloads are dominated by very large batch operations that would be cheaper on a dedicated engine, when sub-100-millisecond latency is required (Trino is not built for that), or when the team cannot sustain the operational investment that production Trino requires.

Trino is a federation layer, not a replacement for the underlying data systems. The right mental model is that Trino sits on top of the existing data architecture and provides a unified query interface to it, while the underlying systems continue to do what they do best.

VII. CONCLUSION

Returning to the three questions:

RQ1. Trino's connector pattern is the architectural feature that enables federation. The coordinator-worker execution model is conventional; the connectors are what allow the same SQL to run against 14 different backends. Predicate pushdown is the operational lever that determines whether the federation actually performs, and pushdown quality varies by connector.

RQ2. The operational realities are mixed. Cold-start time is ~30 seconds and rules out auto-scale-to-zero for interactive workloads. Network topology matters: physical rack proximity to the source data improved our data-intensive queries by ~40 percent. Memory tuning is non-default and OOM failures are common until the configuration is right. Connector reliability varies and treating all sources as equivalent is a mistake.

RQ3. Trino is a federation layer for ad-hoc analytics, self-service discovery, and cross-source joins. It is not a replacement for the underlying systems and it is not a real-time query engine. The teams that get the most from Trino are those that understand its trade-offs and tune for them, not those that expect it to make federation effortless.

The closing observation: federated query systems are powerful enough that the temptation is to treat them as universal solutions, and that temptation is wrong. Trino is excellent at what it is good at and unsuitable for what it is not. The discipline of knowing the difference is what separates a successful Trino deployment from a frustrated one.

REFERENCES

[1] R. Sethi, M. Traverso, D. Sundstrom, et al., "Presto: SQL on everything," in Proc. IEEE Int. Conf. Data Eng. (ICDE), 2019.

[2] Trino Software Foundation, "Trino documentation." [Online]. Available: trino.io

[3] Apache Software Foundation, "Apache Iceberg documentation." [Online]. Available: iceberg.apache.org

[4] Apache Software Foundation, "Apache Hive documentation." [Online]. Available: hive.apache.org

[5] Apache Software Foundation, "Apache Parquet documentation." [Online]. Available: parquet.apache.org

[6] Apache Software Foundation, "Apache Kafka documentation." [Online]. Available: kafka.apache.org

[7] M. Armbrust, A. Ghodsi, R. Xin, and M. Zaharia, "Lakehouse: A new generation of open platforms," in Proc. Conf. Innovative Data Syst. Res. (CIDR), 2021.

[8] A. Thusoo, J. S. Sarma, N. Jain, et al., "Hive: A warehousing solution over a map-reduce framework," Proc. VLDB Endowment, 2009.

[9] M. Kornacker, A. Behm, V. Bittorf, et al., "Impala: A modern, open-source SQL engine for Hadoop," in Proc. Conf. Innovative Data Syst. Res. (CIDR), 2015.

[10] M. Stonebraker and J. M. Hellerstein, "What goes around comes around," in Readings in Database Systems, 4th ed. 2005.

[11] Kubernetes Authors, "Kubernetes documentation." [Online]. Available: kubernetes.io

[12] Microsoft, "Azure Data Lake Storage Gen2 documentation." [Online]. Available: learn.microsoft.com

[13] Amazon Web Services, "Amazon S3 documentation." [Online]. Available: aws.amazon.com/s3/

[14] M. Kleppmann, Designing Data-Intensive Applications. Sebastopol, CA: O'Reilly Media, 2017.

[15] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Eds., Site Reliability Engineering. Sebastopol, CA: O'Reilly Media, 2016.