Scale Processing Architecture for Banking Operations

AFONSO ARAÚJO MACHADO
(BSc)

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Orientadores: Doutor José Manuel de Campos Lages Garcia Simão
Eng. Abel Filipe Vogado Torres Manso Mascarenhas

Júri:

Presidente: Doutor Nuno Miguel Soares Datia
Vogais: Doutor José Manuel de Campos Lages Garcia Simão
Doutor Carlos Jorge de Sousa Gonçalves

December 2024

Scale Processing Architecture for Banking Operations

AFONSO ARAÚJO MACHADO

(BSc)

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Orientadores: Doutor José Manuel de Campos Lages Garcia Simão, ISEL

Eng. Abel Filipe Vogado Torres Manso Mascarenhas, Millennium bcp

Júri:

Presidente: Doutor Nuno Miguel Soares Datia, ISEL

Vogais: Doutor José Manuel de Campos Lages Garcia Simão, ISEL

Doutor Carlos Jorge de Sousa Gonçalves, ISEL

December 2024

Acknowledgements

I express my deepest gratitude to Banco Comercial Português for allowing me to undertake my thesis project within their organization. The support, resources, and expertise shared by my colleagues and supervisors were invaluable in shaping the outcome of this work. Thanks to Abel Mascarenhas and Nuno Reis for their insightful guidance and feedback throughout the process. Additionally, I would like to extend my sincere appreciation to my university supervisor, José Simão, whose academic guidance, encouragement, and expertise were essential in shaping the direction of this thesis.

I would also like to extend my heartfelt thanks to my family for their unwavering support and encouragement. Their constant belief in my abilities and patience during the more challenging moments of this journey has been my greatest source of strength. To my parents, siblings, and extended family, your motivation has been instrumental in helping me reach this milestone.

Finally, thank you to my friends for being my source of laughter and escape during stressful times. Whether through discussions about my work or simply providing a space to relax, your support has been crucial in keeping me balanced and focused.

Statement of integrity

I declare that this dissertation is the result of my personal and independent research. Its content is original, and all sources listed in the bibliographic references were consulted and are duly mentioned in the text. I further declare that all scientific and technical references relevant to the development of the work are duly cited and included in the bibliographic references.

The author

Assinado por: **AFONSO ARAÚJO MACHADO**
Num. de Identificação: BI14910642
Data: 2024.12.09 20.00.23 GMT Standard Time

Lisbon, ... , ...

Scale Processing Architecture for Banking Operations

This document was created using the (pdf)LaTeX processor, based in the "iselthesis" template [63], developed at the DEETC of ISEL-IPL.

Abstract

Pervasive to most digital transformation stacks is the overwhelming presence of massive parallel processing (MPP) ecosystems. An MPP ecosystem is here defined as a set of special purpose MPP engines with socio-technical value, cooperatively combined to deliver end-to-end mission-critical performance with near-optimal constancy at scale. Maintaining an optimal utilization threshold of all computing power without jeopardizing workload or component functioning, safety, or design conditions is a central challenge.

This project explores digital advantage towards the outline of a Scale Processing Architecture founded on an MPP/SQL paradigm to support banking platform needs for scalability, availability, and high-performance data processing. The architecture's central components are Trino and Singlestore. Trino is an open-source distributed query engine capable of adapting to changing workload demands and performing massively parallel virtualized data processing. SingleStore is a recent translytical database optimized for hybrid transactional/analytical processing. The present work aims to enhance Trino workload management alongside SingleStore, facilitating big data processing and analysis at multiple timescales and consumption archetypes while combining and contrasting dominant design patterns, including data warehouse, lakehouse, transactional, translytical, embedded, and streaming.

Keywords: Massive parallel processing; MPP ecosystem; Data-intensive; Workload; Trino; SingleStore; Object Storage; Embedded; Scale processing architecture; Banking; OLAP; OLTP.

Resumo

Presente na maioria das conjuntos de transformação digital é a presença avassaladora de ecossistemas de processamento paralelo em massa (MPP - *massive parallel systems*). Um ecossistema MPP é aqui definido como um conjunto de motores MPP especializados com valor sociotécnico, combinados de forma cooperativa para fornecer desempenho crítico de propósito específico com constância quase ótima em grande escala. Manter um limiar de utilização ótimo de toda a capacidade de processamento sem comprometer o funcionamento da carga de trabalho ou dos componentes, as condições de segurança ou de design, é um desafio central.

Este projeto explora a vantagem digital em direção ao esboço de uma *Scale Processing Architecture* , fundada em um paradigma MPP/SQL para atender às necessidades de escalabilidade, disponibilidade e processamento de dados de alto desempenho de uma plataforma bancária. Os componentes centrais da arquitetura são Trino e SingleStore. Trino é um mecanismo de consulta distribuído open-source, capaz de se adaptar às demandas de carga de trabalho em mudança e realizar processamento de dados virtualizado massivamente paralelo. SingleStore é um banco de dados translitico recente, otimizado para processamento transacional/analítico híbrido. O presente trabalho visa aprimorar o gerenciamento de carga de trabalho do Trino em conjunto com o SingleStore, facilitando o processamento e análise de big data em múltiplas escalas temporais e arquétipos de consumo, ao mesmo tempo em que combina e contrasta padrões de design dominantes, incluindo *data warehouse*, *lakehouse*, transacional, *embedded* e *streaming*.

Palavras-chave: Massive parallel processing; Ecossistema MPP; Data-intensive; Workload; Trino; SingleStore; Object Storage; Embedded; Scale processing architecture; Banca; OLAP; OLTP,

Contents

List of Figures

List of Tables

Listings

Glossary

Data-intensive system	Data is its primary challenge of the system, the quantity and complexity of data, or the speed at which it is changing 3
MPP ecosystem	Set of special purpose MPP engines with socio-technical value, cooperatively combined to deliver end-to-end mission-critical performance with near-optimal constancy at scale. plural 1, 2, 27
MPP system	Is the collaborative processing of a system using multiple processors, allowing the system to perform at higher speeds. plural 6
workload	The amount of time and computing resources a system takes to complete a task. Refers to the total system demand at a given moment. plural 4, 5, 6, 7, 9, 10, 11, 14, 16, 17, 18, 27, 28

Acronyms

CLI	Command Line Interface 31
DAG	Directed Acyclic Graph 10, 15
DBMS	Database Management System 36, 38
DML	Data Manipulation Language 16
ETL	Extract-Transform-Load 9, 10, 18, 23
HTAP	Hybrid Transactional and Analytical Processing 33
JDBC	Java Database Connectivity 16, 17, 21, 22, 31
LLVM	Low-Level Virtual Machine 33
LRS	Locally Redundant Storage 42
MBC	MemSQL Bytecode 33
MPL	MemSQL Plan Language 33
MPP	Massively Parallel Processing 4, 25, 27, 40, 41, 42
NFS	Network File System 41, 42
ODBC	Open Database Connectivity 17, 21
OLAP	Online Analytical Processing 4, 29, 33, 36, 37, 50, 51
OLTP	Online Transaction Processing 3, 4, 29, 33, 37, 38
RDBMS	Relational Database Management System 2
SD	Shared Disk 2
SM	Shared Memory 2
SN	Shared Nothing 2
SPI	Service Provider Interface 31, 32
WM	Workload Manager 33, 34

1 Introduction

1.1 Context

The socio-technical phenomenon under study, MPP ecosystem, was historically shaped by multiple trends, tensions, and technological innovations. Figure 1.1 reveals the dominant forces at play that set in motion different conceptions and approaches over the years. The overall dynamics are chronologically explained in this section.

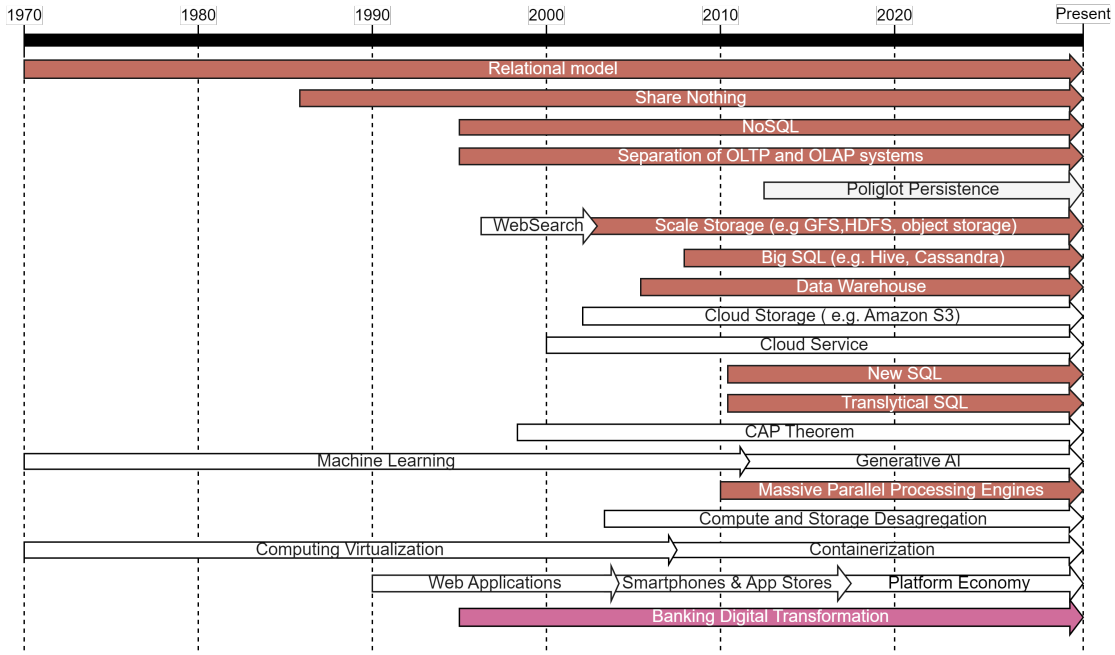


Figure 1.1: *Timeline highlighting dominant forces.*

Starting in 1970, Edgar Frank Codd proposed a relational model for large shared data banks [9]. Adopting this model would also allow the development of a quasi-universal high-level data language. Following Codd's paper, IBM [45] developed System-R [2] and the first implementation of SEQUEL [5]. Around the same time, researchers like Michael Stonebraker developed the Ingres relational system and the language QUEL. In the late 1970s, Oracle Cooperation released the Oracle database. In the early 80s, Ingres and

System-R evolved to PostgreSQL [41] and DB2 [44]. The SQL standard was defined in 1986, and in 1989, Microsoft released the first version of Microsoft SQL Server [59] in collaboration with Sybase [67]. SQL server was Microsoft's entry point to the enterprise-level database market, competing against Oracle, IBM, and PostgreSQL. Also, in the late 80s, Informix [48] was released. Figure 1.2 provides a glimpse of this emergent dynamics.

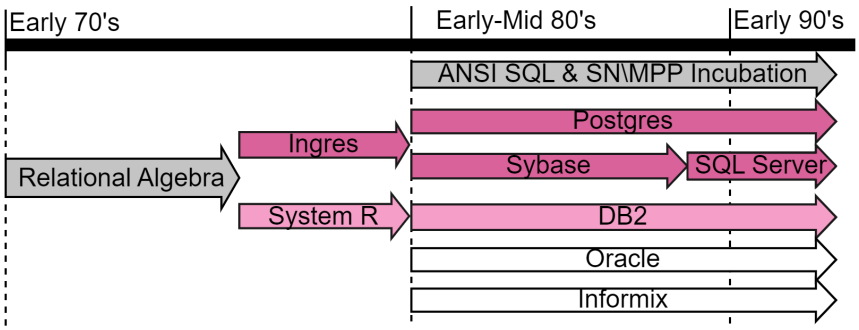


Figure 1.2: *Relational data system emergence.*

In the late 90s, the term NoSQL[78] appeared as an open-source relational database by Carlo Strozzi. The primordial motivation referred to a database system that does not use SQL as a query language. The expression was never well defined throughout history, shifting between non-sql and not-only-sql interpretations, which currently signifies any distributed approach to database design that enables storage/processing of data outside the traditional structures found in a relational database. Driving NoSQL technology's widespread adoption was the need for processing large datasets at a massive scale (when compared with traditional relational databases), the emergence of the open-source movement and specialized workload engines, and the need for semi-structured data designs [50] due to traditional Relational Database Management System (RDBMS) schema restrictiveness. Another vital tech trend propelled by micro-service engineering was polyglot persistence [49], where organizations combine different domain data systems under a range of circumstances, leading to the pervasive coexistence of relational and non-relational data technologies on modern MPP ecosystem.

In distributed database computing architecture, there are different approaches, namely Shared Memory (SM), Shared Disk (SD), and Shared Nothing (SN) [75]. Those are the most commonly mentioned architectures for multiprocessor high transaction rate systems. SM only shares the node memory, SD has a collection of shared disks per node private memory, and SN neither shares memory nor disks, only sharing the network communication links. Since share-nothing architecture is a free side-effect of distributed database systems and adequately addresses the common problem of scale, it became the preferred approach in 1986. However, the public cloud later changed the definition of shared storage at all levels. Public clouds' simplified object store design, offered at a much lower and more flexible cost, ensures permanent deletion and needs to set up security. Amazon launched the first commercial cloud, the Elastic Computer Cloud (EC2), in 2006. Followed by Google in 2008 and Microsoft Azure in 2010.

The search for scalability and storage continued as companies needed to meet the rapidly growing demands of their data processing needs but with a different approach. Hence, in 2003, Google released the Google File System (GFS) [38] that can store hundreds of terabytes of data on thousands of disks and parallelize computation to handle all this data. In 2004, Google released the MapReduce programming model [14] as an abstraction from the complexity of parallelization, fault tolerance, data distribution, and load balancing on the distributed file system. Finally, in 2006, Google released BigTable [6], which is a distributed storage system based on the MapReduce principles for storing structured data that implements a simple data model, instead of the fully relational data model, that supports dynamic control over data that is designed to scale to an enormous size.

Meanwhile, an employee of Yahoo, Doug Cutting (creator of Apache Lucene [23]), and Mike Cafarella were working on a distributed web crawler project called Apache Nutch [17]. Like Google, Nutch needed a distributed file system to achieve massive scale. Therefore, after the release of GFS, they understood the correct approach and started a subproject of Nutch named Hadoop [18]. Hadoop primary data storage is the Hadoop Distributed File System (HDFS) [69], and processes data through the MapReduce model. In contrast with Google, Yahoo decided to open-source the project as part of the Apache Software Foundation. Hadoop came to be a popular choice for data warehousing.

In 2007, Amazon also needed reliability and scalability, releasing Dynamo [15]. Dynamo is a scalable and highly available key-value distributed storage. In 2007, Amazon released the S3 object storage for ample data storage.

At this stage, Hadoop is a widespread open-source implementation used in companies like Yahoo and Facebook to store massive data sets. However, Facebook also needed to meet its reliability and scalability demands for high write throughput, resulting in the development of a distributed storage system called Cassandra [53]. Right after 2010, Facebook saw the MapReduce programming model as a low-level approach that was difficult to maintain in the context of business intelligence. Consequently, Facebook developed Hive [84] on top of Hadoop, which uses SQL-like declarative language (HiveQL) to facilitate querying. In 2013, Facebook also released Presto [68], as an open-source distributed query engine that supports SQL querying for a dozen data sources.

MapReduce continued to succeed on large-scale Data-intensive systems on commodity clusters. Even so, it is unsuitable for different applications like iterative jobs, typical in machine learning and interactive analysis, that require repeatedly querying the same dataset without the latency from reading from the disk every time. In 2010, Spark [87], Apache Spark [81] in 2013, presented a new cluster computing framework that supported these applications, maintained MapReduce core properties, and optimized response time.

Since the relational model, enterprises adopted it for Online Transaction Processing (OLTP) systems databases. Some even believed in the strategy of "one size fits all"[77], an idea that opposed the creation of special-purpose databases. Even so, data warehouses increased, which gathered data from the OLTP systems via ETL (Extract-Transform-Load) to use it for business intelligence. This evolution creates a pendulum between data lake and data

warehousing, where data lake stores raw data for easy access that will then go to data analysis tools (Online Analytical Processing (OLAP)).

In 2011, New SQL [76] came as an alternative to relational databases and NoSQL databases for OLTP systems. Focused on the need for far more throughput and real-time business analysis, New SQL preserves the querying capabilities of SQL and the relational model while preserving the high throughput from NoSQL solutions. VoltDB [13] is an example of a New SQL database. Later, translytical databases presented a unified data platform capable of handling data engineering, analytical, and operational workloads.

In 2011, Kafka [21] [51] was also introduced as a distributed messaging system developed to collect and deliver high volumes of log data with low latency, responding to the large amount of generated log data that needs to be processed.

As different storage implementations came to life to fulfill various requirements, a solution may not result in a single data source. Therefore, the need for support from multiple data sources leads to the orchestration of more than one data source. This orchestration requires scalability and availability, hence compute and storage separation. In the past, databases were used only for transactional processing that needed to be as fast as possible, thus the coupling of computing and storage. Currently, databases are no longer focused on just transactional processing, needing to fulfill other requirements. The disaggregation between compute and storage enables independent scaling, meaning that adding a storage node to respond to things other than transactional processing is possible. If more computing is needed, a compute node is added, improving data availability. In addition, it helps reduce costs.

Distributed query engines, like Trino [35], use a distributed architecture similar to Massively Parallel Processing (MPP) databases [50], which supports querying of large data sets distributed over one or more heterogeneous data sources, including Hadoop data warehouses, RDBMSs, NoSQL systems and stream processing systems.

1.2 Motivation

The motivation for engineering a scalable processing architecture is underscored by the critical needs of modern banking institutions and platform-driven innovation, in this case, Banco Comercial Português, where I am currently employed. Banking systems generate *terabytes* of banking data daily and store an even more significant volume of historical transaction data, which requires a high-performance data processing system. Hence, exploring the digital advantage [57] towards the outline of a Scale Processing Architecture founded on an MPP/SQL data systems provides a unified approach for managing and processing diverse workloads that can efficiently handle increasing applicational demands. A distributed query engine is a critical component in such architecture so that it can optimize for querying large-scale datasets stored in data lakes, warehouses, and other data sources. As such, Trino will be a focal object of study, functioning as a central virtualizer for heterogeneous data tasks and sources. Moreover, it will also play the role of a data lakehouse, which is enabled by its dual nature.

The approach is designed to facilitate the attainment of numerous benefits:

- **Cloud-native containerization system:** Designed to leverage containers and microservices architecture within a cloud computing infrastructure, where the architecture will reside.
- **Big data processing and streaming:** Support for processing and analysis of large volumes of data (big data) and real-time streaming data.
- **Data virtualization and federation:** By providing capabilities for accessing and querying data from various sources in a unified and seamless manner.
- **Single eco-logic for multi-dimensional workloads:** The platform aims to establish a unified and cohesive logic that governs how different types of workloads are managed and executed.
- **Governance model:** Deployed to manage heterogeneous workforce and related socio-technical patterns.
- **Real-time translytical servicing:** The platform will support real-time data processing and analysis for analytical (business intelligence, reporting) and transactional (operational) purposes.

Fundamentally, the project aims to explore a scalable processing architecture and a standard framework providing global control of computing resources for workload combinations inherent in a consumer/producer platform.

1.3 Problem

Given the overall problematics, the following challenges are particularly relevant for the ensuing research. The Scale Processing Architecture (SPA) provides a unified platform that can adapt to changing workload demands and performs real-time processing and streaming. It constitutes a single point of access and control, providing an abstraction from the underlying data source, RDMSs, NoSQL systems, data warehouse, or stream processing systems, bringing high flexibility and extensibility to the overall solution. The organization can add or change data sources without affecting the client's computation due to their native separation from storage. Trino constitutes this abstraction layer permitting agile on-demand increasing or decreasing of necessary computing power

Additionally, it is responsible for establishing a connection with the data source and optimizing the execution of workload processing to the best of one's ability. To perform such optimization, it uses a *connector* that can be seen as the "bridge"between compute and storage, where the connector knows the architecture of the data source and how to take advantage of it, translating into job splitting and parallelism execution. Since Trino supports multiple data sources, each one has a connector maintained by the community. The advancement of specific connectors over others often correlates with their higher utilization. In the current architecture, SingleStore [72] is the central data source.

SingleStore connector doesn't fully explore the database distributed architecture, negatively affecting the Trino optimization process and resulting in less concurrent execution and more synchronous work. Therefore, using Trino as a SingleStore virtualizer implies connector enhancement to minimize the overhead of querying the databases directly due to the current suboptimal implementation.

Another problem is understanding SingleStore's best workload fit and finding supplemental options toward optimal ecosystem workload allocation. To reach a decisive verdict for supporting the Bank's productive continuum, challenger data sources must be stress-tested in comparable conditions.

The final investigation should address the control mechanisms that must be in place for harmonious coexistence of concurrent workloads, including isolation and congestion control.

In summary, the central challenge of a scale processing architecture is maintaining an optimal utilization threshold of all computing power without jeopardizing any workload or component functioning, safety, or design conditions.

1.4 Contributions

The present study's main objective is to explore digital advantage [57] towards the outline of a Scale Processing Architecture founded on a set of special-purpose MPP systems with socio-technical value, cooperatively combined to deliver end-to-end mission-critical performance with near-optimal constancy at scale. Thus, the contributions of this thesis focus on several critical areas in modern distributed databases, cloud-native computing systems, and digital banking transformation while also updating the digital banking landscape [46]. Introducing the historical force field concept provides new insights into understanding temporal changes and trends within data infrastructures. This is complemented by creating a Kubernetes laboratory, offering a practical framework for deploying and evaluating scalable, performance-driven applications. Additionally, implementing a benchmark for banking operations introduces a domain-specific benchmark approach tailored to evaluate financial systems' scalability, real-time processing, and performance needs. This effort aligns with the growing importance of data-intensive industries in banking, particularly in the context of exponential technology and the need for real-time data processing.

Further contributions comprise a contrastive characterization of MPP database paradigms, analyzing virtualizer mediation overhead, and champion-challenger differential dynamics. Alongside benchmarking outcomes, the present work provides an updated analysis of modern database technology, including an in-depth review of nuanced engine internals and serviceable patterns, paradigms, and practices, with an eye on emerging software architectures and dominant digital banking trends. Finally, SQL streaming was also an experimental target addressing reactive mechanisms for real-time data replication, materialization, and stream pattern matching.

1.5 Structure of the document

This document is divided into six chapters. Chapter 2 provides an overview of the current state of both general workload orchestrators and SQL workload processors, explaining their architecture and workload management approaches. The chapter also points out some related work in the context of scale processing architectures.

Chapter 3 defines an architecture approach that will adapt to the workload demands while processing reads and writes in real-time. Furthermore, it explores each architecture component's internals while reviling the target areas for optimization.

Chapter 4 describes the environment implemented and the benchmark definition focused on the banking context. Additionally, presenting an experimental streaming setup.

Chapter 5 analyzes the results of the benchmark execution and, with knowledge obtained from the exploration of each database's internals, evaluates them. Also, presenting the results obtained when using a routing mechanism.

Finally, chapter 6 the key conclusions drawn from the research and discusses potential areas for future work based on the final findings.

2 Background and Related Work

2.1 Background

This section provides an overview of MPP prototypical technologies, which are key to understanding the landscape of Scale Processing Architectures. The technologies reviewed are focused on workload processing, dividing into two categories of analysis: General workload orchestrators and SQL workload processing.

2.1.1 Distributed Workflow Orchestrators

General workload orchestrators are vital in orchestrating and coordinating tasks, services, and processes across distributed environments. This section navigates through some candidates as workload orchestrators that fit on a complex architecture, such as the one explored in this thesis, to promote the seamless coordination of tasks within complex computing infrastructures.

2.1.1.1 Argo

Argo [3] is an open-source container-native workflow engine for orchestration jobs on Kubernetes. The container orchestration system provides tools for deploying, managing, and scaling containerized applications and is implemented as a Kubernetes [34] Custom Resource Definition.

Argo Workflows is used in many contexts, such as machine learning pipelines, data and batch processing, infrastructure automation, CI/CD, and Extract-Transform-Load (ETL). Argo is designed as a lightweight and scalable engine, built from the ground up for containers without the overhead and limitations of legacy VM and server-based environments.

The workflow engine supports the orchestration of parallel pods and workflows, where workflow defines a multi-step model as a sequence of tasks, where each step corresponds to a container. Furthermore, Argo has Workflow Templates that persist on the cluster for reoccurring jobs and can be referenced by other workflows or workflow templates. Since the engine is implemented using GO [40] programming language, it achieves concurrency using a *goroutine*. A Goroutine is a function or method that executes independently and

simultaneously in connection with any other Goroutines in your program, viewed as a lightweight thread.

Architecture Argo is based on two Kubernetes deployments, workload Controller and Argo Server, that run in the argo *namespace*. The workload controller is responsible for reconciling, where a set of worker goroutines process the workflows added to a queue based on adds and updates to Workflows and workflow pods via Informers from Kubernetes. The controller processes a single workflow at a time from the queue. The Figure 2.1 illustrates the process for reconciliation. As for the Argo Server, it serves the API that enables you to submit a workflow, obtain all or a specific workflow, and delete a single workflow from the argo *namespace*. The controller can also be used standalone.

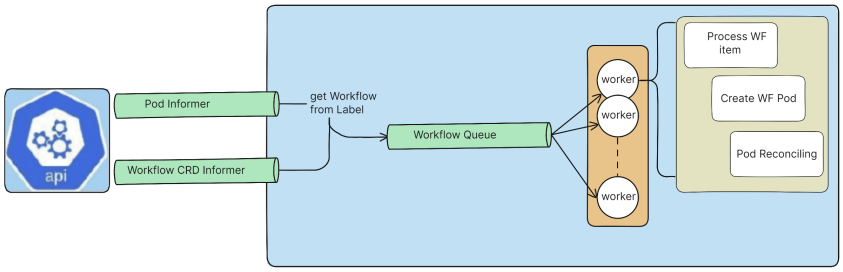


Figure 2.1: *Argo Workflow controller reconciliation process, extracted from [3]*

Argo executes steps corresponding to a task within a workflow or a Directed Acyclic Graph (DAG) task, a collection of steps that depend on each other and are connected through graph structure. Each of these causes a generation of a pod in the user *namespace*, with three containers:

- main - runs the image provided by the user;
- init - fetching artifacts and parameters for the main container;
- wait - performs tasks needed for clean up, including saving off parameters and artifacts.

The engine can be used for CD (continuous delivery), which helps manage and automate the deployment of applications to Kubernetes clusters and the rollout of new versions using different rollout strategies. It also supports an event-driven Kubernetes architecture to trigger workflows, functions, and other actions.

2.1.1.2 Apache DolphinScheduler

Apache DolphinScheduler [83] is an open-source platform that provides a distributed and scalable workflow orchestration. Workflows translate to complex and interdependent data processing tasks that can be challenging to manage and scale without a distributed workflow scheduling platform like DolphinScheduler designed to manage and execute batch jobs, data pipelines, and ETL processes.

The platform enables users to create and manage consecutive jobs to run efficiently, including support for various tasks.

DolphinScheduler uses a powerful DAG visual interface, where each node represents a task and a direction represents the next task to be processed, ensuring no loops exist. It also supports operations like retry, pause, resume, kill, or recover failed tasks.

Architecture The architecture is based on a master and a worker server, where there can be multiple masters and multiple workers. The master will perform the DAG task segmentation, task submission, and monitoring while simultaneously monitoring the health of the other master and worker. The worker is responsible for task execution and logging. The task status is stored in a database since its submission via API, and it is always visible on the UI.

The DolphinScheduler decentralizes by registering the Master and the Worker in Zookeeper, which will elect the manager to perform the task via ZooKeeper [24] distributed lock. After registering, they will keep a heartbeat, enabling fault tolerance in case of server disconnection, which will resubmit the task to a new server. Furthermore, the orchestrator also prioritizes tasks within the same process or between processes.

2.1.2 Big Data Processing Frameworks

This section focuses on workload processors, specialized to efficiently manage and optimize the execution of workloads that involve SQL queries. It will go through some systems built on top of scalable query engines capable of effectively executing SQL queries on large amounts of data. The systems are explored as alternatives to the engines chosen for our architecture and as inspiration to optimize architecture further.

2.1.2.1 Apache Hadoop

Hadoop [18] is a distributed data storage and analysis system to shorten the analysis times when reading data from multiple hard drives.

When reading and writing data in parallel to or from multiple disks, we have two main problems. The first is hardware failure due to the increasing chance of failure when reading from multiple disks; this will be mitigated with Hadoop Distributed File System (HDFS) [19]. The second problem is the capability of combining data from multiple disks, which MapReduce resolves. MapReduce provides a programming model that abstracts the disk reads and writes, transforming it into a computation over sets of keys and values.

MapReduce is a *batch* query processor that enables the execution of ad-hoc queries against the whole dataset. The premise is that the entire dataset, or at least a good portion, can be processed for each query.

YARN [20], as in Yet Another Resource Negotiator, is a framework where the MapReduce executes, and it is responsible for Job scheduling and Resource Management, featuring multi-tenancy, scalability, compatibility, and cluster-utilization.

HDFS Architecture The Hadoop Distributed File System (HDFS) was designed under some assumptions and goals, such as:

- Hardware Failure is the norm rather than the exception. When dealing with many components with a non-trivial probability of failure, some elements of HDFS are always non-functional. Therefore, the detection of faults and quick automatic recovery are the architecture goals.
- Values high throughput of data access, highlighting its nature of batch processing. Even when this means yielding POSIX requirements.
- Should support large data sets. A typical file in HDFS is gigabytes to terabytes in size.
- To avoid data coherency issues and enable high throughput data access, HDFS applications need a write-once-read-many access model for files.
- HDFS Applications must move closer to where the data is located. This will minimize network congestion and increase the system's overall throughput.
- Designed to be easily portable from one platform to another, facilitating widespread adoption.

The HDFS consists of a master/slave architecture. An HDFS cluster is a single *NameNode* and several *DataNodes*, usually one per node in the cluster. The *NameNode* is the master node, responsible for managing the file system *namespace* and regulates access to files by clients. It is a repository for all HDFS metadata. The slaves are the *DataNodes*, usually one per node in the cluster, which manages storage attached to the nodes they run on.

HDFS provides a file system *namespace* and allows user data to be stored in files. When entering the HDFS file system, each of these files is divided into blocks that will be stored in a set of *DataNodes*. All the blocks are the same size except for the last block of a file. The *DataNodes* is responsible for performing block-related operations upon instruction of the master, such as block creation, deletion, and replication, or requests from the file system clients, read and write. The mapping of the blocks to the *DataNodes* is determined by the *NameNode*. That will also execute file system *namespace* operations like opening, closing, and renaming files and directories.

Figure 2.2 represents the HDFS Architecture, including all functions mentioned above.

HDFS is built using Java so that any machine supporting Java can run the *NameNode* and *DataNode* software. Typically, the *NameNode* is deployed in a different machine and a machine for each *DataNode*. Although it is possible to run multiple *DataNodes* in one machine, it is not common.

HDFS supports hierarchical file organization, user or application can create directories and store files inside these directories, also supports access permissions and user quotas.

The *NameNode* maintains the file system *namespace*, and any change to it or it is properties is recorded by the *NameNode*. An application can specify the number of replicas of a file that should be maintained by HDFS. The number of copies of a file is called the *replication factor* of that file, stored by the NameNode.

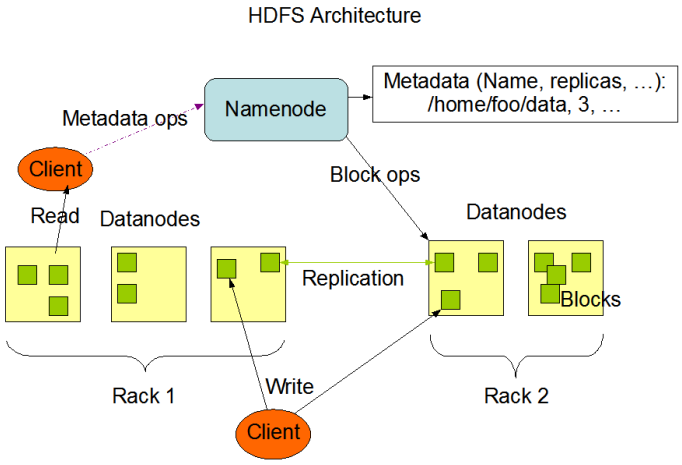


Figure 2.2: *HDFS architecture, from Apache Hadoop Documentation [19].*

HDFS stores each file as a sequence of blocks, each of these blocks is replicated for fault tolerance. The block size and *replication factor* are configurable per file. All decisions regarding the replication of blocks are made by the *NameNode*.

MapReduce MapReduce is an algorithm based on the YARN framework that performs parallel distributed processing in a Hadoop cluster. It is divided into two tasks, *Map* and *Reduce*. The Map function receives data as input, broken into key-value pairs by the record reader, and then processed by our map code. These processed pairs are sent to the Reduce function, where they are sorted and gathered per reducer. After which, the reducer code is executed to aggregate the data. Finally, the output is written into files.

YARN framework YARN splits resource management and job scheduling/monitoring into separate background components known as daemons. Therefore, there is a global Resource Manager (RM) used by all applications and an Application Master (App Mstr) per application.

Since the Resource Manager is global, each machine has a Node Manager responsible for containers, monitoring their resource usage, and reporting it to the RM.

The Application Master is in charge of negotiating resources directly with the RM and working with the Node Manager to execute and monitor tasks in the machine.

The Figure 2.3, represents YARN architecture and how all components described above interact.

The Resource Manager has two main components: Scheduler and Applications Manager. The Scheduler's only purpose is to allocate resources to the various running applications, meaning it isn't responsible for monitoring the task or the application status. Therefore, if the task or application fails at any level, the Scheduler will not perform any extra action. The scheduling function is based on the application's resource requirements, having an abstract notion of the state of the container resources. The Scheduler uses a pluggable policy that is responsible for partitioning the cluster resources among the various queues, applications, etc. Consequently, it is possible to add plugins to fit the application's needs. The Application manager is responsible for accepting job submissions, settling on the first

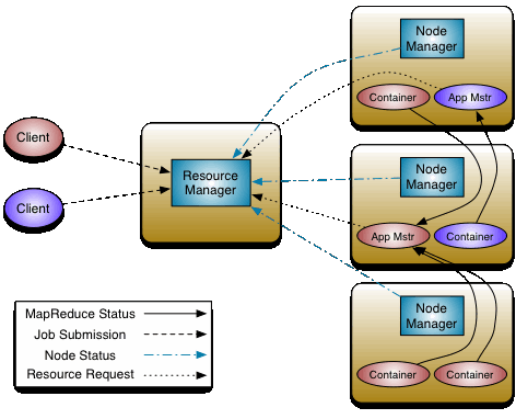


Figure 2.3: *YARN architecture, from Apache Hadoop Documentation [20].*

container to execute the application-specific Application Master, and offering to restart the Application Master container in case of failure.

The Resource Manager determines YARN's scalability based on the number of nodes, running applications, containers, and heartbeat frequency (of both nodes and applications). YARN is capable of scaling to thousands of nodes. When having a cluster with thousands of nodes, we can divide it into smaller clusters, called sub-clusters, with the same components as a cluster. The YARN Federation will join these sub-clusters and present them as one YARN cluster to the applications. This may be used to create greater scalability and/or to allow numerous separate clusters to be utilized together for very big workloads, or tenants, with capacity across all of them.

2.1.2.2 Apache Spark

Apache Spark [81] is an open-source framework-based component for parallel data processing on computer clusters for big data analytics and iterative jobs, whether the data is structured or not. Spark was capable of improving data processing by utilizing in-memory cluster computing. The core concept of in-memory cluster computing is to store recently accessed data in RAM. Therefore, the subsequent access to the same data is already available, and there is no need to obtain it from storage, resulting in real-time computing. The core features are:

- Fast data processing (performs up to 100 times faster than MapReduce) and is capable of splitting the data into chunks;
- Powerful caching and disk persistence - at the programming layer;
- Deployment can be made by Spark's cluster manager, YARN from Hadoop, Apache Mesos, or Kubernetes;
- Real-Time processing and low latency, by using in-memory processing;
- Supports various common languages, including libraries from SQL to streaming and machine learning.

Abstractions of Apache Spark Apache Spark data storage and processing framework, as two main abstraction layers, Resilient Distributed Datasets (RDD) and DAG. Resilient Distributed Datasets is a data computation tool representing a read-only collection of objects partitioned across a set of machines that can be rebuilt in case of a partition failure. RDDs can be modified via transformations and action methods.

Directed Acyclic Graph is a driver that converts the program into a DAG for each job. A sequence of connections between nodes is referred to as a driver. Therefore, you can read volumes of data using the Spark shell and use the Spark context to cancel and run a job or task (work).

Architecture The Apache Spark cluster comprises the driver program, cluster manager, worker nodes (or applications), and the executors.

The*Spark Context* runs in the driver program, which is the main program responsible for scheduling tasks on the cluster. Every application also has an instance of the *Spark Context*.

The cluster manager is responsible for allocating resources across applications. Spark is agnostic to the underlying cluster manager so that it can support other applications. The cluster manager has to be capable of acquiring executor processes, and these must communicate with each other.

Once the *Spark Context* is connected to the cluster manager, Spark obtains an executor in every worker node. These executors are processes that execute multiple tasks in parallel and store data for the application. Therefore, the applications are independent processes coordinated by the *Spark Context*. Then, the executors send their application code to the other executors. The cluster is ready, and the *Spark Context* can send tasks to the executors. Figure 2.4 represents the cluster and all communications previously explained.

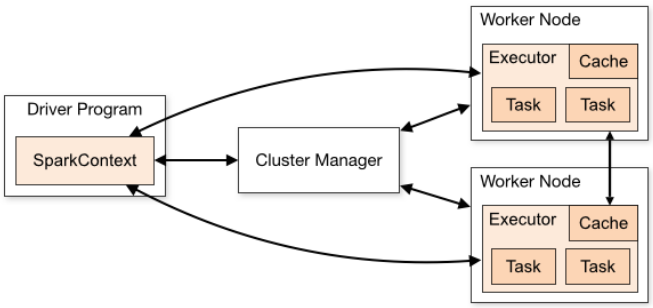


Figure 2.4: *Cluster Mode Overview, from Apache Spark Documentation [82].*

The fact that a worker node has its executors isolates the worker nodes from each other. This isolation is for both the scheduling side and executor side, meaning that each schedules its tasks and tasks from different applications run in different JVMs, respectively. This means that the *Spark Context* instance is isolated in the application, to share information a shared storage is required.

Job Scheduling Spark has two types of schedulers: scheduling across applications and fair schedulers. Scheduling across applications runs in the cluster manager. The fair Scheduler runs in the applications to schedule resources within each *Spark Context*.

There are two distinct approaches for scheduling across applications. The simplest approach is static partitioning of resources, where each application receives the maximum amount of resources. These resources are the same throughout the whole duration. This option allows for different configurations depending on the cluster type, as such:

- Spark's standalone - By default, the application will run in a FIFO order, and all applications will try to use all nodes. It is possible to configure the number of nodes an application uses, change the default applications and memory use;
- YARN - configure the number of executors allocated on the cluster, as for their memory and cores;
- Mesos - Using the static configuration, we can configure the maximum resource used by applications for the memory.

The second approach uses Mesos dynamic sharing of CPU cores, where the memory allocation is still static per application. Still, other applications may use their executors if an application has no tasks. For all the approaches, the memory is never shared.

Spark provides a mechanism to dynamically adjust the resources your application occupies based on the workload. Where an application may give resources back to the cluster if they are no longer used and request them when needed again. There is a resource allocation policy where Spark requests executors in rounds, and the application may request more executors if it has pending tasks for more than a timeout configured. For each round, the application increases the number of executors requested exponentially until it has enough resources to fulfill the backlog of tasks. The policy to remove the requested executors is based on another configurable timeout, the maximum number of seconds an executor can be idling before it is removed.

As for scheduling within the application, that is a *Spark Context* instance, multiple parallel jobs can run simultaneously when submitted from separate threads.

2.1.2.3 Dremio

Dremio [16] is a data lakehouse platform providing self-service SQL analytics, data warehouse performance and functionality, and data lake flexibility. Data analytics can explore and view data with sub-second query response times. As for data, engineers can ingest and transform data directly in the data lake with Data Manipulation Language (DML) operations.

Architecture A basic Dremio cluster comprises three major components: one or more Coordinator nodes, one or more Executor nodes, and Metadata Storage. The queries and access to the coordinator can be granted via web applications, REST API, or Java Database Connectivity (JDBC) drivers, none of which belong to the cluster. This components are represented in Figure 2.5.

A node can be defined as a master coordinator, secondary coordinator, or executor role. As a master coordinator node, it is responsible for managing metadata, query planning,

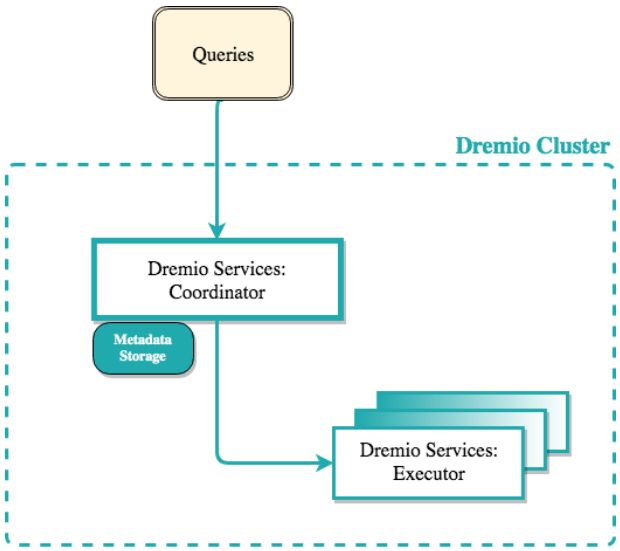


Figure 2.5: *Dremio basic cluster architecture, from Dremio Documentation [16].*

serving Dremio's UI, and handling client connections. A secondary coordinator node improves concurrency and distributes query planning for Open Database Connectivity (ODBC) and JDBC clients originated to your deployment. The executor is responsible for executing queries.

It is important to know that Dremio maintains physically optimized representations of source data, called data reflections, that are like indexes of data, where the query optimizer can accelerate a query by utilizing them.

The Dremio Metadata Storage saves metadata about users, space, and datasets. This storage services support two types of workload, one where the user can request queries and refreshes of data reflections, and another where during metadata refreshes, Dremio gathers and logs data regarding the source datasets. Taking performance into consideration the first type is affected by concurrent queries. Where the second type is affected by the number of tables that Dremio connects and the frequency of refreshes, therefore it requires greater throughput.

Job management Dremio has 3 static job classes for CPU slicing, which is a specific timeframe that a job has to execute using a CPU. These classes have different priorities, starting from the highest priority:

- Near-realtime - submitted through the user UI, and ensures responsive user experience;
- Interactive - submitted user UI as well as ODBC and JDBC connections;
- Background - not submitted by the user, corresponds to reflection creation and maintenance jobs.

The jobs are submitted into one of two static queues, large and small, based on a configured query resource cost threshold. These queues are duplicated in a way that user queries go to a Query Queue Control and reflection queries com to a Reflection Queue Control.

The query threshold, mentioned previously, is established by analyzing the overall cost for a Variety of workloads and setting it based on one expected workload, SLAs, and available cluster resources.

It is possible to configure maximum concurrent queries and timeout per queue. To ensure that small queries are executed when large queries are running, it is recommended that the number of concurrent large queries be smaller than large queries. It is also possible, to configure a maximum memory per query.

Dremio supports queue routing that allows route refresh jobs for all reflections on a dataset, folder, or space to a specific queue.

Workload Management Dremio has a distributed SQL workload processor with clusters that are deployed in multi-tenant environments and provide a workload Management (WLM) to manage cluster resources and workloads. The processor implements a flow based on queues with an associated rule, meaning that a submitted query is assigned to a queue based on a specific rule. These queues can be defined based on a variety of conditions, such as CPU priority, Concurrency limit, queue and job memory limit and runtime timeouts, runtime and enqueued time limits, or tags.

In terms of memory management, each node in the Dremio cluster can have a homogeneous environment, where all nodes have the same memory, or it can have a heterogeneous environment, assigning to the nodes a minimum amount of memory.

2.1.2.4 Presto

Presto is an open-source distributed SQL query engine developed by Facebook [68] and available since 2013. Other large companies also adopted it due to its success. In 2015, Teradata [80] engineers were focused on adding enterprise features. Later in 2017, Starburst [74] was created, a company dedicated to driving the success of Presto everywhere. At the end of 2018, the original creators left Facebook and founded the Presto Software Foundation ensuring that the project remains independent and collaborative, later to be known as *PrestoSQL*.

It provides an ANSI SQL [73] interface to query data stored in a variety of environments or systems, like open-source or proprietary RDBMSs, and stream processing systems such as Kafka [28].

The query engine supports SQL analytic workload, including interactive/BI queries and long-running batch ETL jobs. Also supports several end-user-facing analytics tools. It is designed as an adaptive multi-tenant system managing all worker nodes and resources efficiently and supports multiple data sources. For this reason, it is extensible and flexible to a variety of use cases and is built to high performance, where multiple running queries share a single long-lived Java Virtual Machine (JVM) process on worker nodes [68]. This shared process reduces response time but demands integrated scheduling, resource management,

and isolation.

Architecture A Presto cluster consists of a single *coordinator* node, responsible for admitting, parsing, planning, and optimizing queries as well as query orchestration. Cluster also as one or more *worker* nodes, that will process the query. The Figure 2.6 shows a simplified view of Presto architecture.

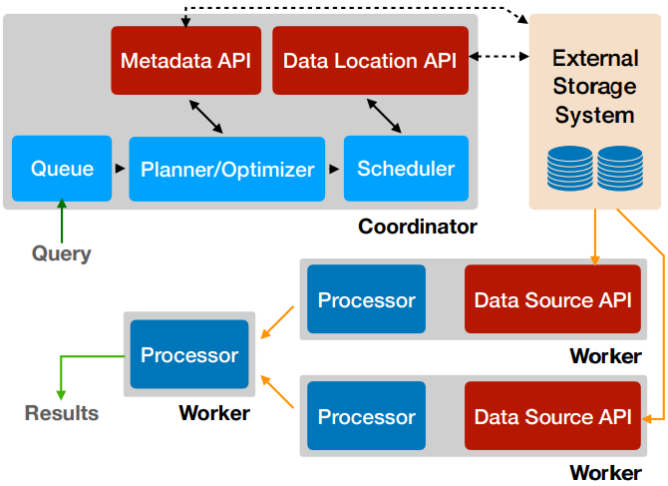


Figure 2.6: *Presto [68] Architecture.*

When a client submits a query the coordinator will process it by evaluating queue policies, parsing and analyzing the SQL text, and creating and optimizing the distributed execution plan. With that plan, the coordinator will distribute it to the workers. Starts execution of *tasks* and then begins to enumerate *splits*, which represent a chunk of data in an external storage system. These splits are assigned to the tasks responsible for reading this data.

Next, worker nodes run these tasks processing the splits by fetching the data or processing intermediate results produced by other workers. The processing follows a cooperative multi-tasking approach, allowing workers to concurrently manage tasks from multiple queries. The execution is pipelined as much as feasible to facilitate the flow of data between tasks as soon as it becomes available, meaning that the different stages are interconnected sequentially. This also means that Presto is capable, for certain queries, of returning results before all data is processed. Data is stored in-memory whenever it is possible.

As mentioned before Presto is designed to be extensible supporting multiple data sources, for that reason it provides an adaptive plugin interface. Plugins can provide custom data types, functions, access control implementations, event consumers, queuing policies, configuration properties, and *connectors*. These *connectors* allow Presto to communicate with external data stores through the Connector API, composed of four parts: the Metadata API, Data Location API, Data Source API, and Data Sink API. The APIs enable the development of high-performance connectors that operate seamlessly within the context of a physically distributed execution engine.

System Design and Features The Presto engine was built around key design decisions and features that are important for understanding how Scheduling, Query Execution, and Resource Management are performed.

The SQL Dialect used is based on the ANSI SQL specifications. Although the engine does not implement every feature, it adds some extensions to the language to simplify and improve usability.

The queries go through a parser that converts SQL statements into a syntax tree, from this tree and analysis information an intermediate representation (IR) is generated. The IR is encoded as a tree of *plan nodes*, where each node represents a physical or logical operation, where their children are the inputs.

The logical plan is divided into *plan fragments*, which are then transformed into a more physical structure that represents an efficient execution strategy for the query. This optimization divides into multiple *shuffles*, which are the stages to collect data for the output. A *stage* is the runtime incarnation of a plan fragment, and it encompasses all the *tasks* of the work described by the stage's plan fragment, that will then be distributed to the workers. The coordinator will decide how many tasks those stages should be scheduled and for which worker nodes. At the *task* level, the Scheduler will examine the plan tree and classify stages as *leaf* or *intermediate*. Leaf stages are at the lowest level of a distributed query plan responsible for retrieving data, and intermediate stages are at the higher level of a distributed query plan to recover data from other stages.

As tasks are assigned to the worker nodes, the coordinator starts assigning splits to these tasks. The worker node will queue those splits for all his tasks. The number of splits is decided by the *connector*, defined by the catalog, for each data source. Once a split is assigned to a thread, it will be executed by a *driver loop*. Every iteration of the loop will move the data, called *page* and is a sequence of rows, between all pairs of operators that can make progress. Drivers act upon data and combine operators to produce output that is then aggregated by a task and then delivered to another task in another stage. An operator consumes, transforms, and produces data. The driver loop will stop iterating when the operators can't make progress or *quanta* is complete, where quanta is the maximum time slice available for execution. Presto uses in-memory buffered shuffles, where tasks will insert their output into those buffers for consumption by other workers. TThe engine is constantly monitoring this output buffer utilization if the utilization is high the engine reduces the concurrency by reducing the number of splits. When the produced data needs to be written to other tables, Presto will dynamically increase writing tasks to free up output buffers, which is triggered by exceeding a certain threshold. Being those two adaptive approaches, how buffers memory is managed.

Presto executes hundreds of queries concurrently maximizing the use of resources. Therefore, CPU usage is tracked at the task level ato minimize coordinator overhead nd scheduling decisions are made locally. As said before, splits are only allowed to run for a *maximum quanta* of one second, after which a thread is yielded and the tasks return to the queue. In a case where output buffers are full, meaning that downstream stages cannot consume data fast enough, or input buffers are empty, this means upstream stages cannot produce data fast enough, or the system is out of memory, the local Scheduler simply switches to processing another task even before the quanta is complete. When the split relinquishes the thread, the engine needs to decide which task to run next.

Presto gives higher priority to queries with the lowest resource consumption, using tasks aggregated CPU time, that is the accumulated quanta, to classify into one of the five levels of a multi-level feedback queue. As time is collected, the task moves to higher levels of the queue. Each level of the five levels has a configurable fraction of the available CPU time.

Presto does not guarantee Fault Tolerance, the clients are responsible for automatically retrying failed queries in the worker node. As for failures at the coordinator, the cluster becomes unavailable.

2.1.2.5 Trino

Trino [55] is an open-source distributed SQL query engine, enabling SQL access to any data source. Widely used by many well-known companies to query large data sets by horizontally scaling the query processing, providing a high performance.

Mapping the historical journey, Trino was originally PrestoSQL (or Presto), when in 2020, to reduce confusion between the legacy PrestoDB project and other versions, the community decided to rename the project as Trino. The foundation was also renamed the Trino Software Foundation. Consequently, all versions from 351 forward use Trino, meaning the last released under PrestoSQL was version 350. The core maintainers keep innovating with new SQL support, updating runtime usage of Java, and modernizing code and architecture. A community of developers and contributors from all adherent companies also help maintain the project.

Therefore, Trino inherits the previously explained features from Presto (section 2.1.2.4). Even so, we will revisit the cluster architecture, giving a high-level view of all involved components.

High-level Architecture Trino is a distributed SQL query engine resembling massively parallel processing (MPP) style databases and query engines. The engine distributes all processing across a cluster of servers horizontally. Therefore, a Trino cluster is composed of two types of servers, coordinator and workers, where the coordinator collaborates with one or more workers. The coordinator and workers will access the connected data sources via data source connectors, all of which are configured through catalogs. Figure 2.7 illustrates such a cluster, including the client that submits a query to the coordinator. Querying Trino can be done with the Trino CLI as well as any database management tool connected with JDBC or ODBC, represented as a *client*.

Originally Trino was designed to query data from HDFS, but as the query engine, it can now query data from object storage, relational database management systems (RDMSs), NoSQL databases, and other systems. For that reason, the concept of a connector specific to each database allows Trino to interact with a resource using a predefined API while the connector knows how the access can be even more optimized.

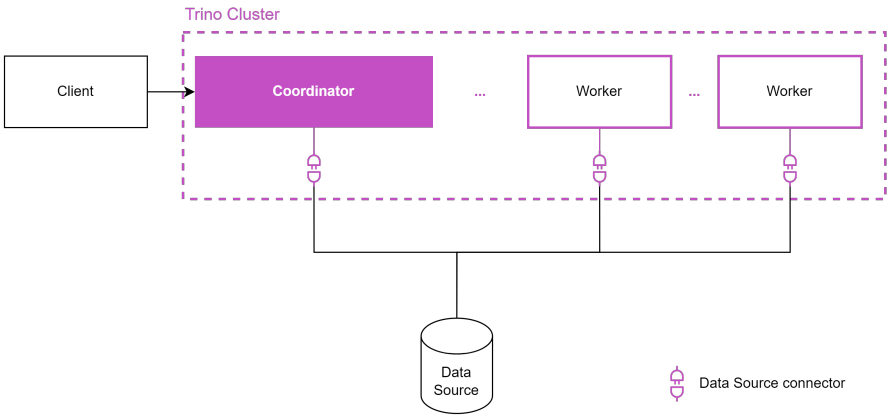


Figure 2.7: *Trino cluster.*

2.2 Related Work

This section will review some studies about scale processing architectures, like those in the previous chapter. These studies provide different architectural approaches and implementations in the context of SQL query engines while also demonstrating how they perform under various testing conditions and configurations.

2.2.1 On the performance of SQL scalable systems on Kubernetes: a comparative study

The paper [4] explores the impact in the performance of SQL on Hadoop systems, where the focus was on four representative systems, such as Apache Drill [25], Apache Hive [26], Apache Spark-SQL and Trino. The performance assessment consists of deploying these systems on a Hadoop cluster with Kubernetes and using the TPC-H benchmark.

The comparative methodology consists of executing queries a total of five times per SQL system to describe the ability of the engines to improve through optimization techniques. Each test of the systems was performed isolated. Another evaluation was made on the ability to scale to different volumes of data. Therefore, three different scale factors were considered for the benchmark: 10, 100, and 300GB. For each iteration, the tables will be created, and, at the end of every trial, the cluster is cleansed. The experiment was executed five times for each scale factor, the 22 queries of TPC-H [12].

The experimental setup is a kubernetes cluster with seven nodes. In terms of database configurations, Trino used the Hive metastore connector, along with the deployment of Hive where the TPC-H tables were created. For Spark, Python scripts were produced to create the tables in-memory. The Hive metastore used the JDBC driver to query the tables. Finally, for Drill a view was created for each table, to be queried.

The paper started by evaluating the performance of the four systems based on the mean and speed-up of the query execution times. The speed-ups are calculated as the highest execution time divided by the current system computing time, meaning that a speed-up of 1.0 indicates that the system is the slowest in the corresponding query. Therefore,

speedup values indicate how much faster a specific system performs relative to the slowest system for that query. The final statistics indicated Trino had the best execution time in 20 queries, whereas Drill and Spark had the fastest execution time in two queries. Looking at the values averaged across all 22 TPC-H queries presented in table 2.1, Trino is the fastest system.

Table 2.1: Mean execution time and speedup with respect to the slowest system, averaged across all 22 TPC-H queries (100 GB scale factor), extracted from paper [4].

	Drill	**Hive**	**Spark**	**Trino**
Speed-up	2.21	1.09	1.79	3.05
Mean	61.88	115.13	70.3	40.64

In the scalability side of the experiment, the objective of the authors was to determine whether the engines maintain the behavior that persists when scaling the data size down (10GB scale factor) and up (300GB scale factor). Comparing 10GB with 300GB data, the rank previously observed maintains for Drill, Spark, and Trino. As for the Hive, in the 10GB scale, the execution is frequently lower or almost equal execution time when compared to Spark, even so in the 300GB scale there is a degradation in the Hive execution time and an improvement for Spark. In conclusion, the results suggest that the overhead is not linear, and its influence on the total computing time is compensated when in a higher volume data situation.

By mentioning the study, the choice of integrating Trino is enforced as a fast-distributed SQL query engine capable of outperforming Spark and Hive.

2.2.2 Presto: A Decade of SQL Analytics at Meta

The paper [79] goes over the challenges faced by Meta [58] and how they improved Presto. Given the growth in data volume at Meta, maintaining query latency and scalability became even harder. Presto has supported production analytical at meta since 2003 for large-scale interactive, ad-hoc, and ETL workloads. Additionally, Meta's efforts in migrating all SparkSQL workloads to Presto to be the only SQL interface to the warehouse.

Therefore, due to the new SQL analytics challenges and demands from machine learning, privacy policy enforcement, and graph analytics, Presto addresses them with three perspectives:

- Latency and efficiency;
- Scalability and reliability;
- Requirements beyond data analytics, like SQL-like graph analytics, support for mutability, and user-defined types and functions.

The original architecture experiences latency bottlenecks due to I/O from external storage, which is disaggregated from compute engines. Additionally, Java-based workers are slower than native code due to limited memory control. Some storage connectors attempt to

reduce latency by dumping data into local memory or disks. However, collocated storage introduces data management overhead, eliminating the advantage of independently scaling compute and storage resources. Furthermore, as the amount of data processed by the system grows and elastic capacity is adopted, the limitations posed by the coordinator acting as a single point of failure and the absence of fault tolerance mechanisms for workers have become more pronounced. Also, the in-memory design limits the amount of data the system can hold. As a result, scalability and reliability started to be a concern.

Therefore, Meta improved Presto architecture as depicted in Figure 2.8, which supports multiple coordinators. Spark cluster leverages Spark as the runtime and Presto as the evaluation library for scalability. Materialized views are used in both setups to improve query performance and data mutability. Moreover, data can now be spilled to *temporary storage*, overcoming memory limitations, where the Presto cluster uses it for recoverability to materialize intermediate data and the Spark for shuffles.

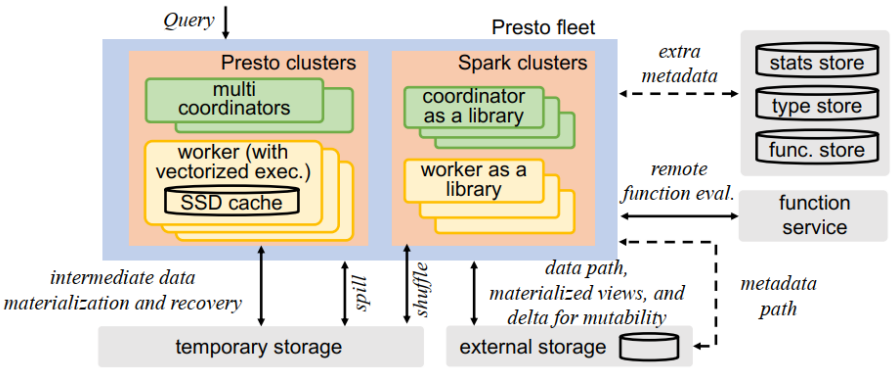


Figure 2.8: *Presto new Architecture. - image extracted from the paper [79]*

Four aspects were improved for latency improvements. Starting with caching data and metadata in three different levels: cache data in the workers local disk with cache locality so that queries are scheduled preferentially on workers that have the data cache; fragment result cache that translates to caching the result of leaf stages plan fragment without any filtering; and, finally metadata cache mutable (e.g. file indexes) on memory and immutable (e.g. schemas and file path) on the local flash of both coordinators and workers. The next improvement is native vectorized execution, given that Java cannot take advantage of modern CPU capabilities like SIMD (Single Instruction, Multiple Data) C++ workers were built to reduce CPU time by roughly 6500 seconds. Presto introduced Adaptive filtering, which stands out in filter reordering and filter-based lazy materialization, dropping more rows in the first filters so that the next filter has fewer rows and only materializes when all predicates match. Finally, materialized views and near real-time data support are added to improve latency, especially in subquery optimization. Presto attempts to match a materialized view with a sub-query of the received query. Subquery optimization improved by nearly 17 hours less CPU time from the original 24 hours, less 24 billion rows scanned, and less than 8 seconds of latency.

For efficiency improvements, the cost-based optimizer was upgraded to use external information to estimate the cost. All services, including Presto, are responsible for calculating

and publishing the partition statistics to the metadata store, which will be used later during planning. Cost-based optimizer improved the majority of the query's CPU efficiency even so, for others, increasing CPU. For the increased CPU queries, 83% of them lowered memory usage. Support for a history-based was added to leverage the precise execution statistics from the previously finished repeated queries to guide the planning of future repeated queries, giving an overall 10% CPU improvement for repeated queries. Even if the optimizer is missing statistics, adaptive execution allows the coordinator to leverage finished tasks to optimize the plan of downstream tasks.

The fact that the coordinator is a single point of failure is especially challenging for long-running queries. Therefore, Meta separates the queries and clusters' lifecycles to support multiple coordinators, introducing resource managers to manage the cluster, and the coordinators manage their queries while being independent of each other and without communicating with other coordinators.

The growth of machine learning and privacy-focused requirements also shifted from a static data warehouse to an open and mutable data lake setup. Furthermore, Graph extensions were added to Presto, such as a Presto SQL with graph querying language constructs and a graph query planner that incorporates graph-specific optimizations. Presto was compared against Apache Giraph [22], with queries with 10, 15, and 20 hops that showed an overall CPU gain of roughly 28% by using presto. For a query that calculates a "subgraph,"which is the set of reachable edges from a specific node within a larger graph, nearly a 13% CPU gain by using presto.

Generally, the latency improvements for both interactive and ad-hoc workloads are depicted in Figure 2.9, where the 95th percentile significantly improves to less than 10 seconds of latency.

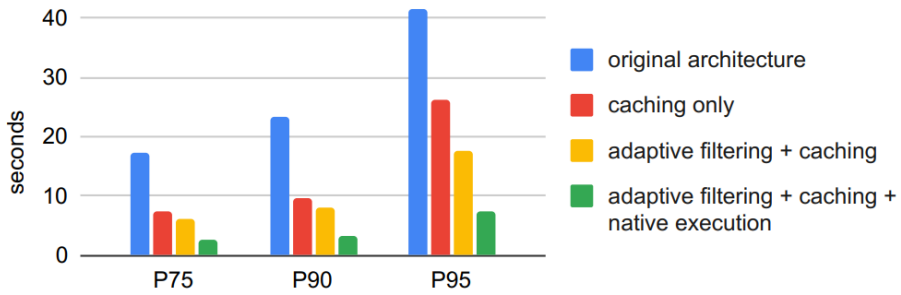


Figure 2.9: *Presto interactive workload latency comparison, from the paper [79].*

This inclusion is essential to understanding the challenges faced by Meta and the solutions implemented in a real-world application, and the possibility apply some of the solutions to our architecture that include Trino originally Presto.

2.3 Summary

This chapter overviews MPP technologies and their importance in large-scale processing architectures. The presented technologies are categorized into two main areas: general workload orchestrators and SQL workload processing systems. Argo is integrated into our

architecture for workload orchestration as a tool to manage and coordinate tasks efficiently in distributed environments. For SQL workload processing, Trino is used as a scalable query engine. Additional technologies are analyzed as inspiration and reference points for implementing and optimizing the architecture discussed in this thesis. Futhermore, relevant related work is reviewed to contextualize the contributions and evaluate alternative approaches.

3 System Architecture

3.1 Approach

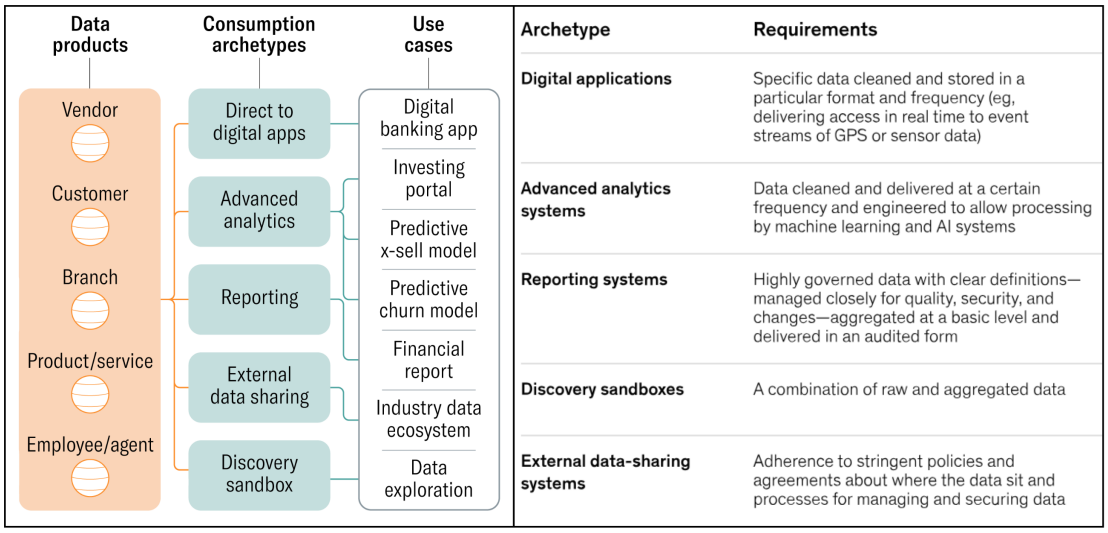
The architecture consists of a contrived MPP ecosystem, with different approaches that will be explored to achieve a symbiosis of multiple special purposed MPP systems. This final symbiosis will not contain all the technologies referenced here since the main objective is to achieve multidimensional optimizations by understanding distinct paradigms and finding the best fitting to the banking context. The process of optimizing the symbiosis of multiple specialized MPP systems involves addressing three key dimensions *:

- optimize data systems connections;
- find optimized ecosystem configurations;
- contrast the different data systems.

These dimensions are evaluated through benchmark execution, focusing on response time and resource usage as key metrics to assess and refine the architecture.

The fulfillment of modern banking needs, namely, efficient processing systems capable of handling the exponential dynamics of big data workload and workforce patterns, depends on the following scale processing desiderata: reliability and responsiveness, governability, and maintainability. The ultimate goal is to provide an agile platform for users and services to process relevant data from various data sources for applications with different data consumption archetypes, such as Figure 3.1. A consumption archetype describes what the data is used for.

Nevertheless, the focal point is to explore the Trino and SingleStore connection, the technologies used in production, by understanding and re-engineering the current implementation of the SingleStore plugin. Simultaneously, compare their performance with other systems that can work with SingleStore. The comparison is evaluated through a benchmarking framework focused on banking operations, accessing several relevant architecture metrics. The results will contribute to ongoing debates in the engineering community, such



Archetype	Requirements
Digital applications	Specific data cleaned and stored in a particular format and frequency (eg, delivering access in real time to event streams of GPS or sensor data)
Advanced analytics systems	Data cleaned and delivered at a certain frequency and engineered to allow processing by machine learning and AI systems
Reporting systems	Highly governed data with clear definitions—managed closely for quality, security, and changes—aggregated at a basic level and delivered in an audited form
Discovery sandboxes	A combination of raw and aggregated data
External data-sharing systems	Adherence to stringent policies and agreements about where the data sit and processes for managing and securing data

Figure 3.1: *Data applications, from Harvard Business Review [66] and Mckinsey [56].*

as data warehouse versus data lakehouse, ETL versus zero-ETL, and SQL Translytical 2.0 versus NoSQL[78].

3.2 Overview

The present work explores a symbiotic MPP ecosystem architecture integrating Trino as open data virtualize, Argo Wokflows as an open container orchestrator, Kafka [28] for Event Streaming, and five database paradigms:

- Lake Analytics DB;
- Translytical DB;
- Real-time Bid Data Analytics DB;
- Embedded Real-time Analytics DB;
- Multi-active Geo-transactional DB.

Figure 3.2 presents the architecture explained in this section. The architecture is incorporated on a federated domain approach [61], where each domain consumes data from the *insights hub* - Trino. Additionally, data ingestion from external data sources would be processed through the *event Hub*, Kafka, where the ultimate goal is to achieve real-time and batch processing. Even so, the study will focus on optimizing Trino and his data sources.

To orchestrate workloads Argo Workflows is used as a Kubernetes native solution, explored in chapter 2. Finally, Apache Flink is integrated into the architecture as a Change Data Capture (CDC) tool, enabling real-time, event-driven processing of database changes. Even so Apache Doris, Argo Workflows, and Apache Flink are integrated into the architecture, their role is purely exploratory and will not be put under test in the present study.

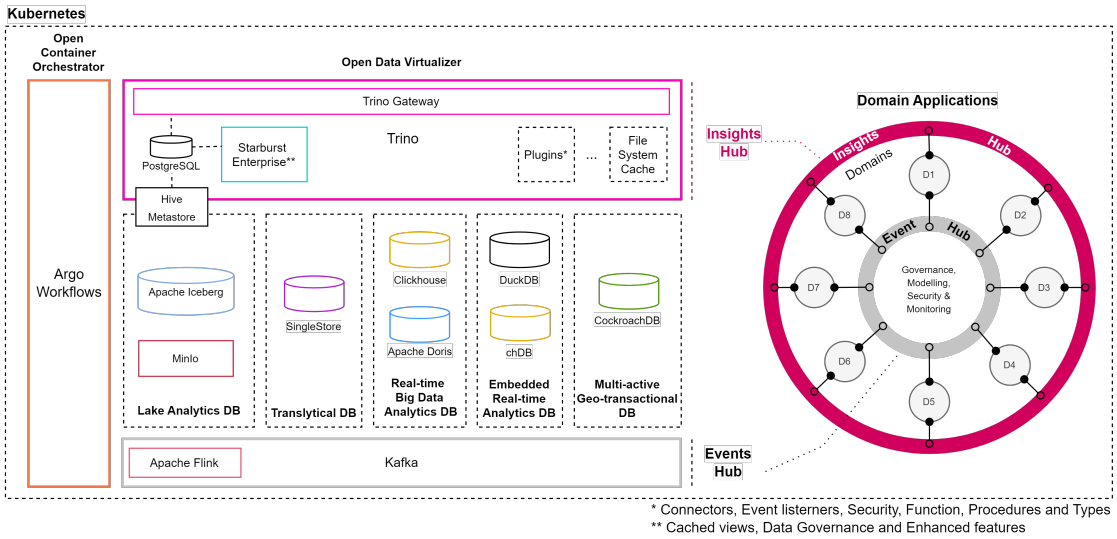


Figure 3.2: *Architecture overview.*

Trino is an open data virtualizer with an SQL interface capable of querying multiple data sources and managing massive workloads. Trino gateway [37] appears as a load balancer between Trino Clusters in a multi-cluster scenario. The enterprise version of Trino, Starburst Enterprise [74], can also link with the gateway. PostgreSQL database appears as a backend database to Trino Gateway, Starburst, and Hive-metastore [26].

Apache Iceberg is an open lakehouse tabular format and protocol that adds data warehouse-level capabilities to a traditional data lake. Iceberg brings database ACID functionality to object storage, instantiated in this architecture by MinIO-S3. It specifies a protocol that requires a catalog to record the file's location and distributed ACID information, enabling more efficient petabyte-scale data processing. The catalog is instantiated in a Hive-metastore and is an auxiliary to Trino when searching for data. MinIO is a high-performance, distributed object storage system that is S3 compatible.

In the different database paradigms, SingleStore [72] is the primary focus for optimization and comparison, as it forms the foundation of the architecture. SingleStore is a translytical [86], hybrid transactional/analytical processing [10, 64], multi-model, multi-workload, multi-timescale, multi-temperature distributed SQL database that offers high-throughput transactions and low latency analytics.

For Big data processing in Real-time, Clickhouse [8] and Apache Doris [29] databases stand out for OLAP workloads and their connection with S3 storage. In real-time processing, another paradigm is embedded databases, where a database is only instantiated when processing is needed. In our case, focusing on OLAP, DuckDB [33] and the embedded version of Clickhouse chDB [7] were added to the architecture.

Although Apache Doris wasn't included in the benchmarks, its promising results on other benchmarks [30] led us to add it to the architecture for future comparison.

CockroachDB [52] is focused on OLTP workloads such as SingleStore but changes the paradigm by being a geographically distributed database with high isolation transactions

various regions.

The compute and storage separation brings challenges since workload optimization can't be generic for every storage. Trino will need to communicate with the specific data source in a specialized way to maximize parallelism when executing a query, using the *connector* defined on the data source *catalog*. It is observed that the Trino plugin for SingleStore is not optimized, so the goal is to improve it. Furthermore, the purpose of having multiple data sources is to contrast them.

The following sections will detail each component of Figure 3.2, comprehensively explaining their roles and functions within the designed architecture.

3.3 Trino Gateway

Trino Gateway is a load balancer, proxy server, and configurable routing gateway for multiple Trino clusters. It can be crucial in managing and optimizing query routing, workload distribution, and system upgrades, offering numerous advantages in environments where multiple Trino clusters are utilized. Allows client tools to connect via a single connection URL, regardless of how many Trino clusters are running in the backend. Users don't need to worry about selecting the correct cluster or managing different connection points. The gateway distributes workloads across multiple clusters based on predefined rules, ensuring optimal query handling.

One of the key features of Trino Gateway is its ability to automatically route queries to the most appropriate Trino cluster based on a set of routing rules to decide which group of backends (trino clusters), called Routing Group, should handle the query. A routing group can have multiple backends, meaning the gateway needs to choose which handles the request using either an adaptive or round-robin strategy. The adaptive strategy sends a request to the least loaded member of the routing group, while the round-robin sends it to a random group.

Furthermore, managing resource groups across multiple Trino clusters is facilitated with the database resource group manager [36], where Trino clusters connect to a centralized resource groups table maintained in the Trino Gateway's database. Using Trino gateway with the database management avoids the need for cluster re-deployments when updating resource groups, enhancing operational efficiency.

3.4 Trino

Considering that Trino is the focus of the project due to his fundamental role as a virtualization engine, understanding its code structure is crucial to breaking down the complex workload management into its components so that it is possible to develop a detailed plan to deal with them. Trino's detailed architecture was already explored in chapter 2.

3.4.1 Git Code Structure

The Trino GitHub [39] is divided into three main packages: *client*, *plugin*, and *core*. It also contains other packages that provide libraries and testing. The code is implemented using Java programming language, with the Airlift [1] framework. Figure 3.3 illustrates the dependencies between all packages explored.

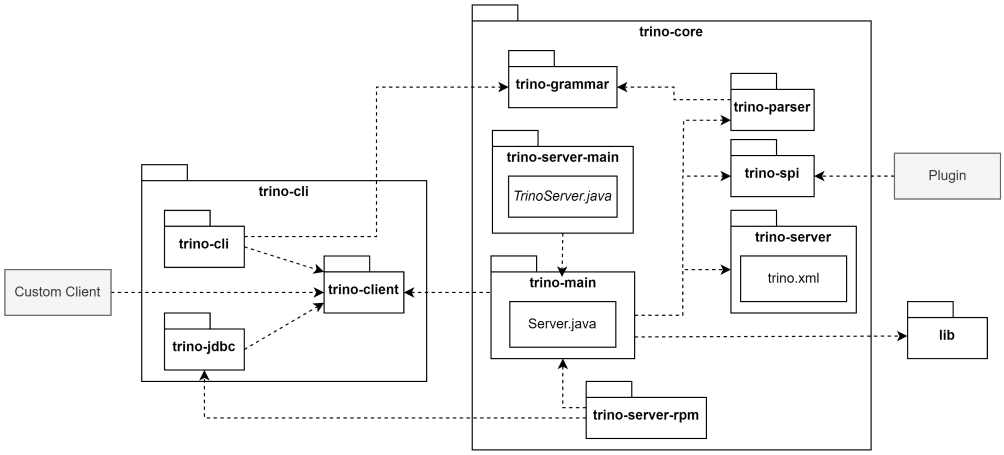


Figure 3.3: *Trino GitHub code structure analysis.*

Starting with the client packages (inside *trino-cli*) contain all the protocols necessary to connect to the Trino cluster. Hence, the Trino JDBC driver and Command Line Interface (CLI) use these same protocols to connect to Trino.

The core package is the backbone of Trino, where it contains code for both the coordinator and workers and also, the Service Provider Interface (SPI) that will reveal to be an important protocol of the engine. In full, the core package contains the following packages:

- **trino-grammar** - SQL grammar definition;
- **trino-main** - Implementation of all Trino feature;
- **trino-parser** - Query Parser, includes SQL, types, and exceptions;
- **trino-server-main** - Trino Server main class;
- **trino-server-rpm** - Red-hat Package Manager (RPM) package for Trino server;
- **trino-server** - Holder of *trino.xml*, contains all Maven [32] artifacts of Trino.
- **trino-spi** - SPI protocol.

The starting point is the *TrinoServer*, which will call the *Server*, present on *trino-main*, that is responsible for loading all configuration files and modules. The coordinator and worker have specific modules that distinguish them, meaning that, even though they require the definition of the class hierarchy from the same interface, they implement different logic. Thus, a configuration differentiates the coordinator from the worker. Furthermore, the server depends on the protocol defined on *trino-client* to receive queries.

The SPI defines a mission-critical protocol as the only connection between Trino and the external software. In other words, a connector implements the interfaces and methods defined by SPI, that benefit from the data source architecture to optimize data access.

Trino plugins provide additional capabilities, such as connectors, block encodings, types, functions, system access control, group providers, password authenticators, header authenticators, certificate authenticator, event listeners, resource group configuration managers, session property configuration managers, and exchange managers. Connectors are the source of data. They adapt your data source to the API expected by Trino. Furthermore, every Trino *catalog* references a data source with a connector. The package plugin contains all Trino plugins that are available.

3.5 SingleStore

SingleStore, or MemSQL, distributed architecture design is based on a few key principles, the first being high scalability, enabling storage capacity, and processing power scale at any time. It is designed for high availability by having replicas of every cluster partition that, in case of a node failure or even total cluster failure, can bring the cluster back online using those replicas. A partition is a subset of a database data, also known as a *shard*.

A cluster is divided into two tiers: aggregators and leaves. The aggregators receive queries and route them to leaves, aggregate intermediate results returned by the leaves, and return the final results to the client. There are two aggregator types, master and child, where the master is taking on additional duties of cluster monitoring and failover. As for the child aggregator, it doesn't have these additional cluster duties. Furthermore, a cluster always contains one master aggregator and may have none, one, or many child aggregators. Finally, a leave, also known as a leaf node, operates as a compute and storage node, given that it stores several partitions. Figure 3.4 offers a visualization of a SingleStore cluster described.

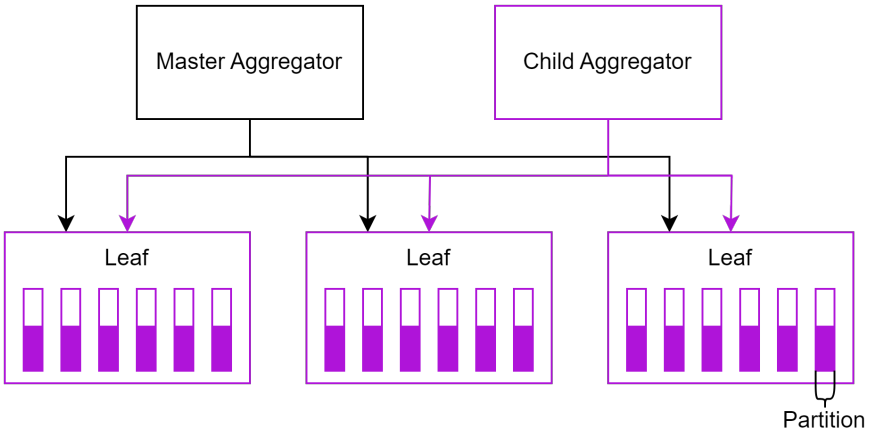


Figure 3.4: *SingleStore cluster overview.*

The data stored in tables can be one or two types:

- **Universal Storage (default)** - previously referred to as column store, recommended for all kinds of workloads. Supports OLTP and Hybrid Transactional and Analytical Processing (HTAP) workloads.
- **Rowstore** - stored in memory for high-performance OLTP.

Additionally, a table can either be sharded, distributed across partitions, or referenced, entirely replicated across all cluster nodes.

A cluster has at least one leaf node, typically more than one, and can rebalance partitions when adding another leaf without the restart of any node. The operation of rebalancing partitions is a re-distributing of data across the cluster so that data is more evenly distributed within the cluster.

A parametrized SQL query is immediately translated into a query shape for SingleStore to start processing it. A query shape refers to the structural and operational characteristics extracted from the original SQL query, which includes the patterns of joins, filters, aggregations, and how data flows through various stages of query execution. The first time SingleStore encounters a query shape, it will optimize and compile the query asynchronous for future invocations with the same "shape". The code generation process transforms the query shape into a SingleStore-specific representation tailored for the system (aggregators and leaves configuration), using SingleStore Plan Language - MemSQL Plan Language (MPL). After the code generation, the query is saved for future invocations in the plancache, where each node has its own. This cache is stored in-memory and on-disk (files on the node storage under the plancache directory). The way that it works is when a plan expires or is evicted from the in-memory, it is kept in-disk until expiration on the in-disk plancache (plan expiration limits), or the files are deleted. The plan is only recompiled for a certain query shape if that plan is present neither in-memory nor on-disk, meaning that it is re-loaded in-memory if present in the on-disk plancache.

SingleStore improves data parallelism while reading data with Flexible parallelism. The mechanism allows multiple cores on the same node to access the same database partition, by dividing a partition into sub-partitions. As a query runs on a leaf node, multiple cores working on behalf of the query can process different partition sub-partitions in parallel, which is highly beneficial for OLAP workloads.

Originally, code generation in SingleStore was in C++ and then compiled to machine code, impacting the first query (plan creation). Then, it was upgraded to the generation of Low-Level Virtual Machine (LLVM) bitcode and then compiled to internal MemSQL Bytecode (MBC), which is faster and enables more tailored optimizations at each level. Even so, to impact the first query even less while interpreting MBC on the first run, the LLVM is also compiling in the background.

The Workload Manager (WM) limits the execution of queries that require full distribution to ensure that they match the available system resources. The method improves overall query execution, given that it prevents surges of resource usage.

Each leaf node has a maximum of threads available for the aggregators to use. WM

distributes these threads among the aggregators depending on their load, meaning that at a point in time, one aggregator may have more threads available. Additionally, the WM also manages the number of connections established per leaf node, so it doesn't reach the configurable maximum and leaf memory.

WM also throttles the number of queries running on any aggregator by queuing the queries. First, queries are classified based on size into large(L), medium(M), or small(S). To ensure that no queue becomes saturated with a large query, SingleStore dynamically moves queries to other queues. Queries are evaluated on the resources they use, where each size has a maximum threshold. Small queries, due to their small resource utilization, are never queued. Figure 3.5 logically represents resource pools and the queuing and execution of the queries. If no resource pool is defined, SingleStore uses the default resource pool, which, if configured, can be chosen by the user or query.

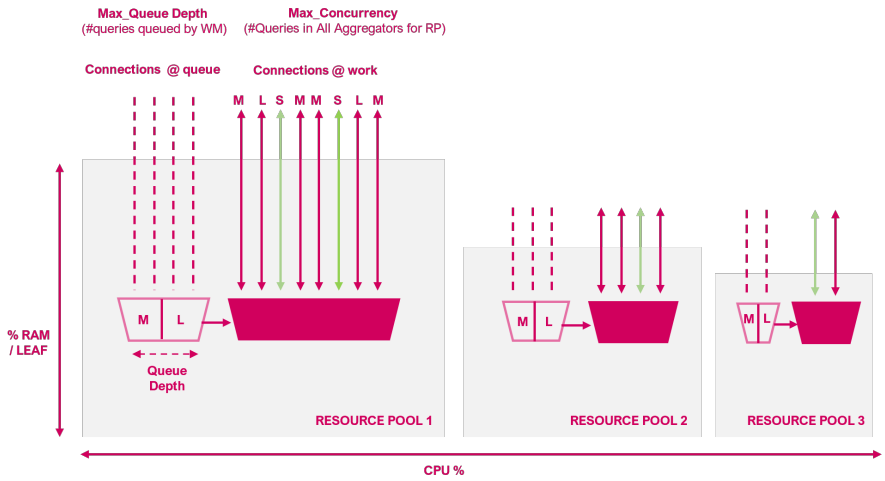


Figure 3.5: *SingleStore Resource Pools overview.*

The disk usage of the SingleStore node is split into four directories:

- *tracelogs* - where the engine writes all its diagnostic messages, being those messages either INFO, WARNING, ERROR, or FAILURE;
- *plancache* - on-disk plancache stores plans;
- *auditlog* - there are multiple logging levels, each providing limited or exhaustive information about user actions and database responses, useful for performing common information security tasks;
- *data* - has multiple directories under it, focusing on three main directories snapshots, blobs, and logs.

The directories snapshots and blobs store your data. The logs directory contains transaction logs for each partition. These files record all the changes made to the database before being committed to a snapshot.

3.6 Apache Iceberg

Apache Iceberg is a high-performance open table format containing several petabytes of data and works like a SQL table. Iceberg adds a metadata layer over an object store(S3/MinIO) that keeps track of table state modifications and an updated *snapshot* of the table.

A snapshot stores the files that belong to a table and their statistics. It has two levels of metadata, manifest list and manifest files. The manifest list acts as an index, based on the partition value ranges associated with the list of manifest files. The manifest file stores a list of data files with the corresponding partition data and column-level stats, using a persistent tree structure. Given that, Iceberg filters the manifests using the partition value ranges, it is possible to plan without reading all manifests, improving performance.

Iceberg was designed to solve correctness problems, improving reliability by always updating snapshots on write and delete. Snapshots reuse as much as possible of the previous version, to avoid the overhead of big writes. When committing a change, the path of the current table metadata file is replaced using an atomic operation. Hence, in concurrent writing the first to commit obligates the second to commit based on the updated snapshot.

Iceberg uses hidden partitioning, meaning that it automatically prunes out files that don't contain matching data, and you don't need to write queries for a specific partition layout. As a result, queries do not reference partition values directly so the partition spec can be updated using a metadata operation and does not eagerly rewrite files. Data sort order can be configured and later updated, but old data written with the previous order remains unchanged.

Iceberg uses metadata to keep track of the files in a snapshot, divided into manifest files and manifest lists. Manifest files store a list of data files, along with each data file's partition data and column-level stats. The manifest list stores the snapshot's list of manifests and the range of values for each partition field. Therefore, for fast scanning, the manifests are filtered using the partition value ranges in the manifest list, and then the data files are retrieved from the remaining manifest. Figure 3.6 visually explains how a table is stored in Iceberg. The table metadata file tracks the table schema, partitioning configuration, custom properties, and snapshots of the table contents. Any time a table state changes, a new metadata file is created, and the old one is replaced via an atomic swap. A snapshot represents a point-in-time view of the data in a table. It is one of the fundamental concepts used to track changes and enable features like time travel, data versioning, and incremental data processing. It references the manifest files, pointing to the actual data files. Each snapshot tracks all the data files via manifests that are part of the table at that moment. A new snapshot is created when changes are made, like appending new data, deleting, or updating records.

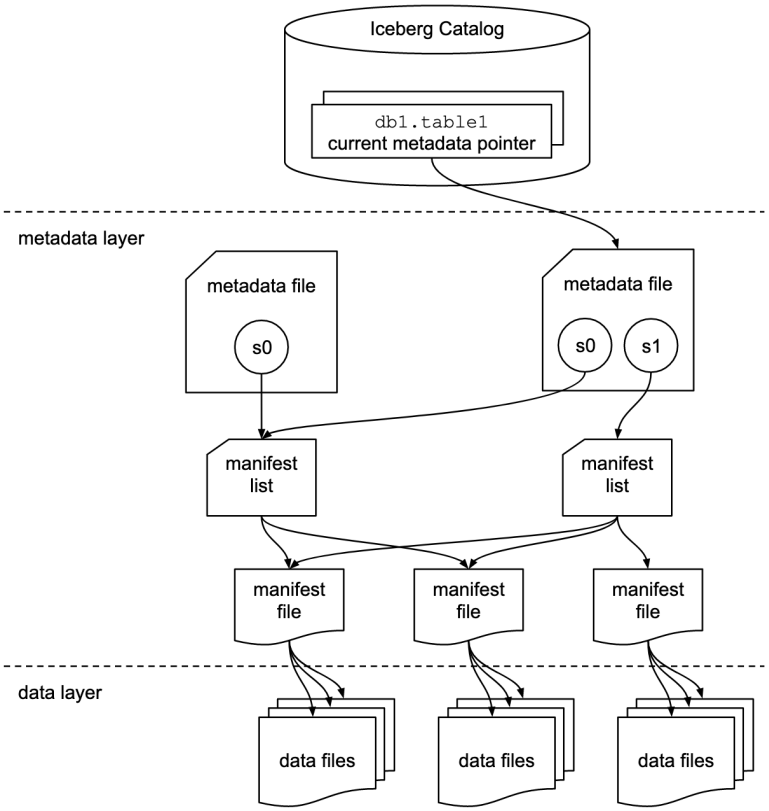


Figure 3.6: *Iceberg Files Overview, extracted from Apache Iceberg Documentation [27]. (s0=Snapshot0)*

3.7 Clickhouse

ClickHouse is a high-performance SQL Database Management System (DBMS) for OLAP, offered as open-source software. Initially built to perform a single task: filter and aggregate data as fast as possible. Therefore, several high-level decisions were made:

- Column-oriented storage - avoiding reading unnecessary columns to avoid expensive disk reads;
- Indexes - memory resident Clickhouse data structures enable the selection of the necessary column only;
- Data compression - storing different values of the same column together often translates to better compression ratios;
- Vectorized query execution - processing data in columns as they are stored leads to better CPU cache;
- Scalability - leverages all cluster resources to execute a single query.

Furthermore, Clickhouse focuses on low-level details, like automatically choosing the best GROUP BY implementation of 30+ possibilities for each query. This is also the case with algorithms used for sorting, where Clickhouse can choose the most performing.

The Clickhouse cluster consists of a collection of nodes running Clickhouse instances, called servers, that work together to store and process data.

The Clickhouse index was designed for massive data scales, meaning that instead of having a primary index for every table row, which is good for specific row locations, it uses a sparse index. The sparse index is possible because data is written part by part, where each part has one primary index entry, known as a mark, per group of rows, called a granule, and the primary key orders parts. Therefore, a sparse primary index is used to identify one or more granules that potentially match rows. Figure 3.7 illustrates the process of granule selection based on three file types:

- *primary.idx* - file contains the first row of every granule for each primary index., existing only one file stored in memory;
- *.mrk* - contains the offsets that point to the granule physical location on the bin files, existing one mark file per column;
- *.bin* - files that contain the actual data (all rows), existing one file per column.

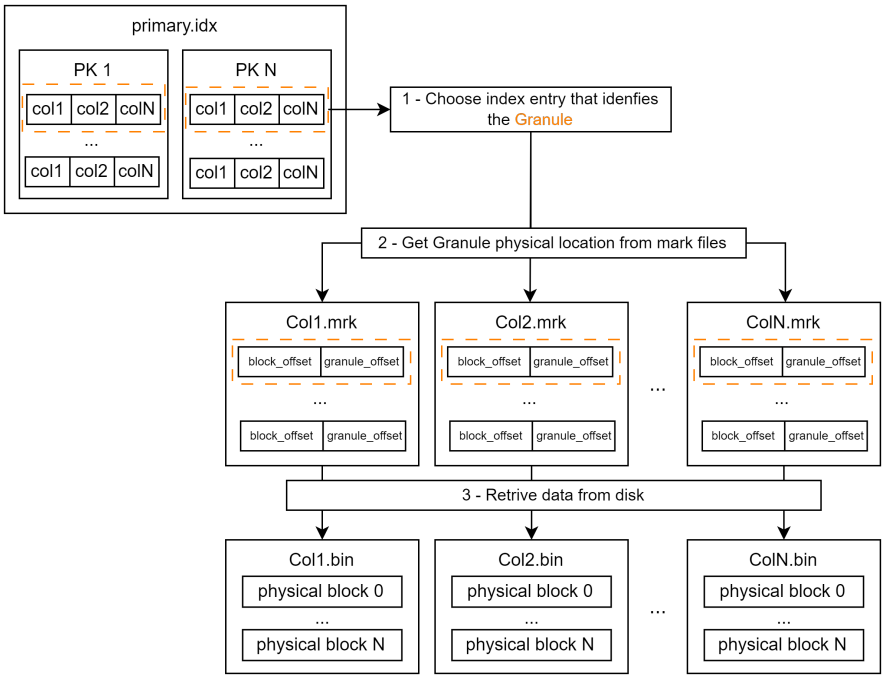


Figure 3.7: *Clickhouse sparse index - granule selection.*

3.8 DuckDB

The widespread acceptance of SQLite [43], OLTP database, shows a need for an OLAP in-process data management solution, such as DuchDB [65]. DuckDB is an open-source embeddable relational database management system built for OLAP workloads. It's accessible via a C/C++ API and is compatible with SQLite, Python, and others.

DuckDB uses columnar storage, which is particularly efficient for analytical queries where aggregation or filtering is often used. While traditional databases execute queries row by row, DuckDB uses vectorized query execution, where a vector is a large batch of values processed in one operation. Applying operations, such as aggregation, filtering, or calculations, to the entire vector simultaneously reduces the overhead of repeatedly switching between operations and data.

Furthermore, it offers an extension mechanism to define new data types, functions, file formats, and new SQL syntax. Some are already defined, like support for Parquet file format, JSON, and S3 protocols.

Regarding indexing, there are two types: zonemaps and Adaptive Radix Tree (ART) indexes. DuckDB automatically creates zonemaps for the columns of all general-purpose data types (e.g., INT, TIME), and works by storing only the minimum and maximum values for a set of data. As for the ART indexes can be created for a column implicitly with the PRIMARY KEY, FOREIGN KEY, and UNIQUE constraints or by executing "CREATE INDEX". ART indexes are beneficial for query performance when using index columns on the queries without jeopardizing the performance of join, aggregation, and sorting queries. This enables efficient constraint checking on changes but slows down changes to the affected column(s) because of index maintenance. Indexes are serialized to disk and deserialized lazily when the database is reopened. Operations using the index will only load the required parts of the index.

3.9 CockroachDB

CockroachDB is a distributed SQL DBMS key-value store built for OLTP workloads. It supports ACID (Atomicity, Consistency, Isolation, Durability) transactions across multiple regions, offering serializable isolation, the highest level of transaction isolation by default. Serializable isolation guarantees that transactions executing in parallel behave the same as if they had executed one at a time without any concurrency. Furthermore, its multi-region capabilities enable users to choose geographic regions by database, table, or row. In case of a node failure, it offers fault tolerance and full recovery of the node without interrupting service due to the replication between nodes, where replicas are automatically rebalanced to use all the available resources.

CockroachDB supports the PostgreSQL wire protocol [42] and most of its syntax. This signifies that the majority of PostgreSQL database tools and drivers migrate to CockroachDB without changing application code most of the time.

Indexing on CockroachDB automatically creates indexes with a primary key or, if there is no primary key, with a unique value per row. You can use secondary indexes, such as partial and spatial indexes, to further improve queries. A partial index is based on a predicate expression (WHERE) that evaluates rows and columns to true on the index's boolean and creates a sorted copy of the subset of rows without altering the table. As for spatial indexes, are to prevent from looking row-by-row. Also, storing columns not part of the index key is possible, so you don't need to join with the primary key index.

The keys and values are stored on disk as strings containing unrestricted byte values. Keys are encoded into a sentinel key that contains the table ID, primary key value, and column ID suffix, given that the value is the column value of the row. If the column is nullable, cockroachDB uses the absence of a key-value pair for a column to indicate NULL via a sentinel key with no column ID suffix. To represent secondary indexes, the index ID is added to the sentinel key, whereas the primary key is "primary". So the sentinel key would be as in the figure 3.8 and a practical example of it the figure 3.9, in the figures, the forward-slash is a visual separator.

Key	Value
/tableID/indexID/indexColumns[/columnID]	columnValue

Figure 3.8: *Cockroach Sentinel Key representation.*

SQL

```
CREATE TABLE test (
        key INT PRIMARY KEY,
        floatVal FLOAT,
        stringVal STRING
        )

INSERT INTO test
VALUES (10,4.5,"hello");

INSERT INTO test
VALUES (4,NULL,"hi");


CREATE INDEX foo ON test (stringVal)
```

Key	Value
/test/primary/4	∅
/test/primary/4/stringVal	"hi"
/test/primary/10	∅
/test/primary/10/floatVal	4.5
/test/primary/10/stringVal	"hello"
/test/foo/"hello"/4	∅
/test/foo/"hello"/10	∅

Nullable column indcation

Secondary Index

Figure 3.9: *Cockroach Sentinel Key example.*

3.10 Apache Flink

Apache Flink is a powerful, open-source stream processing framework that excels at handling both batch and real-time data streams. It is designed for distributed, stateful data

processing, enabling it to perform complex event-driven processing at scale. Flink offers different levels of abstraction for developing streaming/ batch applications, Figure 3.10, such as SQL, Table API, DataStream, and Stateful Stream Processing. Since our architecture is focused on SQL language throughout all MPP systems, the focus is on Flink SQL.

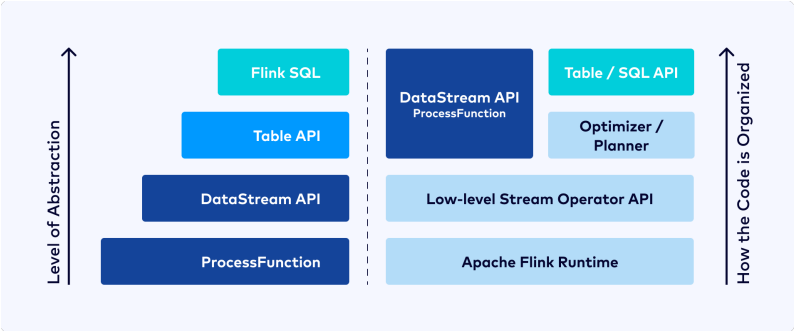


Figure 3.10: *Apache Flink Framework, from Confluent [11]*

Flink SQL allows users to write SQL queries to process batch and streaming data. It supports stream and batch processing through the same API, allowing historical and real-time data to be worked with. Additionally, enables time-based processing such as windowing and joins on event-time or processing-time, making it ideal for real-time analytics. Integrates with various data sources, including Kafka, relational databases (e.g., MySQL), and object stores (e.g. MinIO), providing flexibility in querying diverse data streams.

Flink uses stateful processing, enabling complex transformations, aggregations, and analytics over continuous data flows with exactly-once processing guarantees. This is accomplished through checkpointing, where it stores the state. Flink can be configured to store these Checkpoints on Minio server [85].

The presence of Flink in the architecture allows us to explore possibilities in real-time and batch processing from SingleStore to another database using Kafka.

3.11 Summary

In summary, the architecture presented in this chapter lays the foundation for exploring various database paradigms and comparing them, especially our core database, SingleStore. By focusing on multidimensional optimizations—ranging from optimizing system connections to exploring diverse ecosystem configurations and contrasting data systems, a research environment and benchmark were defined. Moving forward, the next chapter will focus on the practical implementation of this architecture, detailing the environment setup and the benchmark definitions. This environment will serve as the testing ground for the strategies outlined here, allowing for empirical evaluation of the system's performance under various conditions.

4 Environment and Benchmark: Setup and Evaluation

This chapter describes the implementation and deployment of the MPP systems described in the previous chapter, detailing the environment and the benchmark framework, reviewing each component's configurations, detailing the choices for Virtual Machines (VMs), networking considerations, and resource allocation, such as CPU, RAM, and storage requirements. As for the benchmark framework, we will explain how we designed it, focusing on banking operations.

4.1 Cloud Environment

The scale processing architecture is deployed on a Kubernetes (K8s) cluster hosted on the Azure Cloud [60] virtual machines. The Kubernetes cluster was deployed using a pure upstream version ("Vannila Kubernetes"), built from scratch. This approach ensures that we retain full control over the configuration, allowing us to fine-tune the environment to the specific requirements of our architecture while ensuring compatibility with any future updates or integrations.

In addition to the Kubernetes cluster, we also needed to deploy an Network File System (NFS) Server and a Workbench. The NFS server was chosen to streamline the volume allocation process across the cluster, so it facilitates the management and sharing of storage resources, ensuring that all components in the architecture have access to the necessary data, regardless of where they are deployed within the cluster. NFS provides a reliable and flexible solution for our distributed environment, making it easier to scale and manage storage as the system grows. The Workbench is designated as the environment for executing code that requires a Linux distribution for any code execution over the Kubernetes (K8s) cluster. It is the primary space for tasks such as benchmark execution, data generation, and other operations that require direct interaction with the underlying infrastructure. This setup is meant to ensure accurate performance measurements and seamless integration with every technology in the K8s cluster.

Therefore, within the same subnet, for the K8s cluster, eight nodes were needed, one master

and seven worker nodes, where the master node doesn't allow regular workloads only as the control plane is a tainted master node. For this reason, the master node requires fewer resources than the worker nodes. The NFS Server and the Workbench, due to the nature of their roles, also require fewer resources compared to other components in the architecture. In conclusion, the resources of each host are specified in Table 4.1. Regarding storage, all systems use dedicated premium SSD Locally Redundant Storage (LRS) disks supplied by the NFS server to store data. Trino file system cache uses temporary host storage, which doesn't need to be persisted. Resulting in three premium SSD LRS disks, high-performance managed disk offered by Microsoft Azure, attached to the NFS server, one with 128GB and two with 500GB capacity, the reasoning being having two disks is so we can decouple storage of SingleStore and S3 (main storage systems) into two separate disks of 500GB. Hence, 128 GB remains for Kafka, PostgreSQL, Trino catalogs, and other integrated databases, considering that Azure Cloud allows for disk expansion. All data disks were configured with XFS file system.

Table 4.1: Hosts resource allocation.

Designation	Quantity	Operative System	Azure Instance	vCPU	RAM (GiB)	Host Storage (GiB)
K8s - Master Node	1	Ubuntu Server	D4asv5	4	16	30
K8s - Worker Nodes	8	Ubuntu Server	D8sv5	8	32	30
NFS Server	1	Ubuntu Server	D4asv4	4	16	30
Workbench	1	Ubuntu Server	D4asv4	4	16	30

Figure 4.1 better illustrates the environment fully, also clarifying the Workbench role, which is to give access to the components installed in Kubernetes, either via a command-line interface (CLI), clients (e.g., DBeaver), or native user interfaces. Furthermore, it is used to execute the benchmark queries.

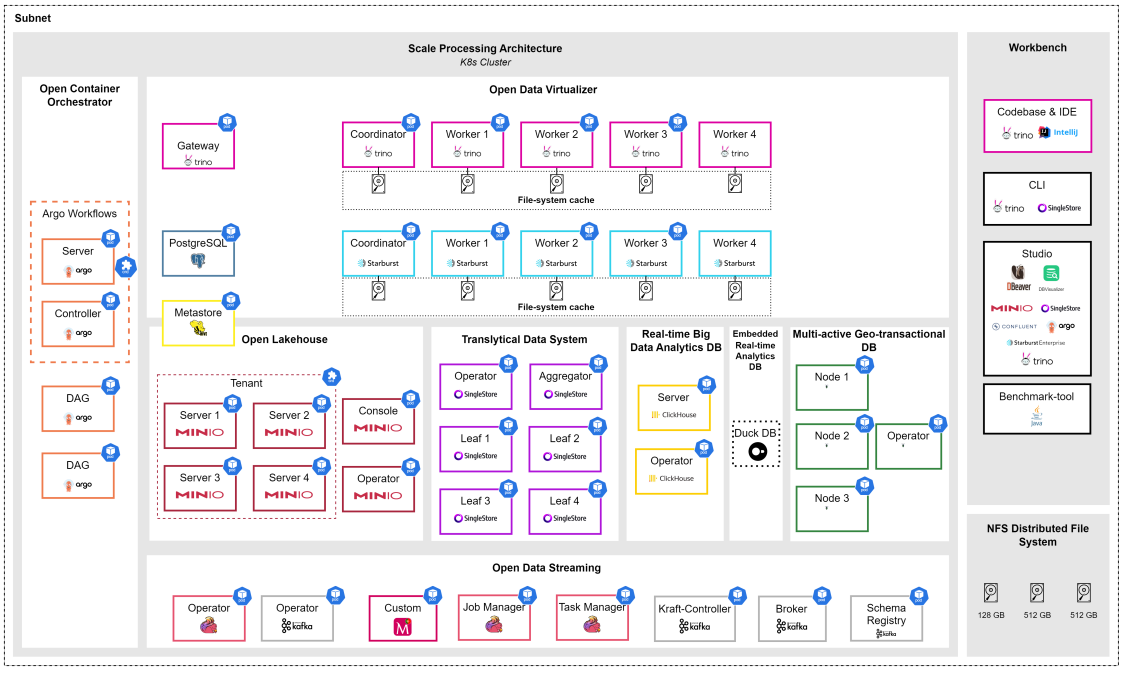


Figure 4.1: *SPA Environment.*

Considering all the MPP systems under test (SUTs) and the resources available on the

Kubernetes Cluster, Figure 4.2 details all the versions used during benchmark executions and resources allocated.

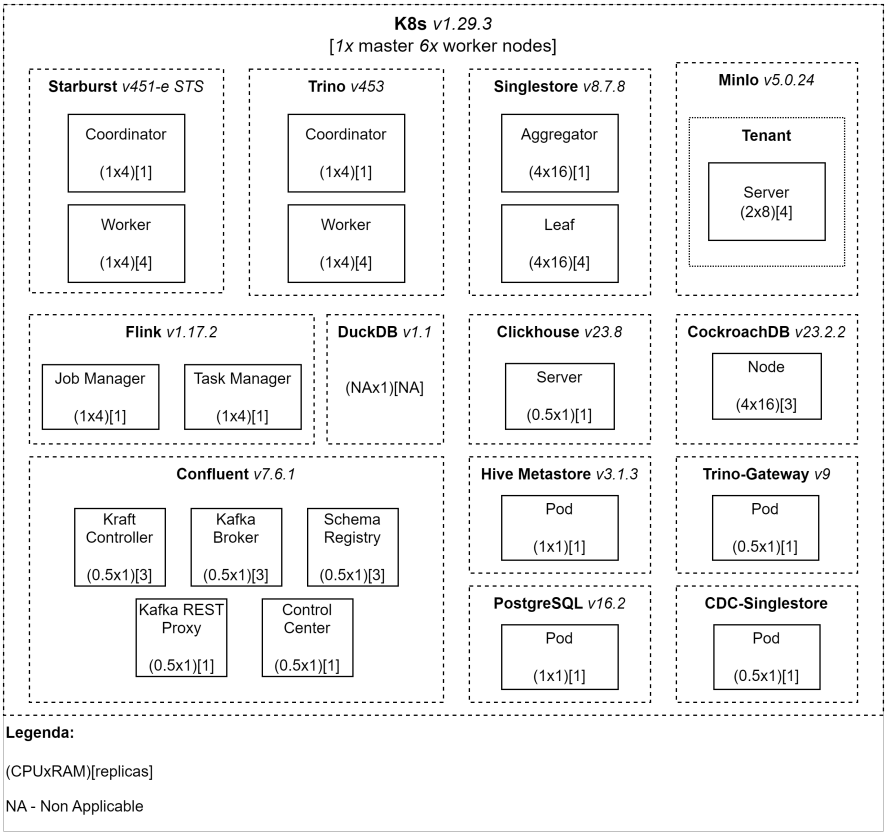


Figure 4.2: *SPA resource distribution.*

4.2 Streaming Experimental Setup

Data replication in real-time materialized views and stream pattern recognition is essential for building scalable, highly available, and performant systems to achieve a reactive architecture. Thus, it motivates a Proof of Concept (PoC) of a data pipeline using Flink SQL. The idea is to extract data from SingleStore (source), ingest it into Kafka using Trino, process it using Apache Flink, and then store it in Apache Iceberg (sink). Due to simplicity, data ingestion into Kafka was done using Trino, but it could be done directly to Kafka. The Figure 4.3 depicts the logic of the experiment.

To configure Flink, it was necessary to build an image for Flink with all the necessary packages to query data from both Kafka and Iceberg, which requires Hive and S3 object-storage packages.

The step of extracting data from SingleStore was done using the OBSERVE query [70], which returns a CDC (Change Data Capture) stream in table format. For every row, the relevant columns for the PoC are:

- Offset - current offset, where it can be used to keep the last processed;

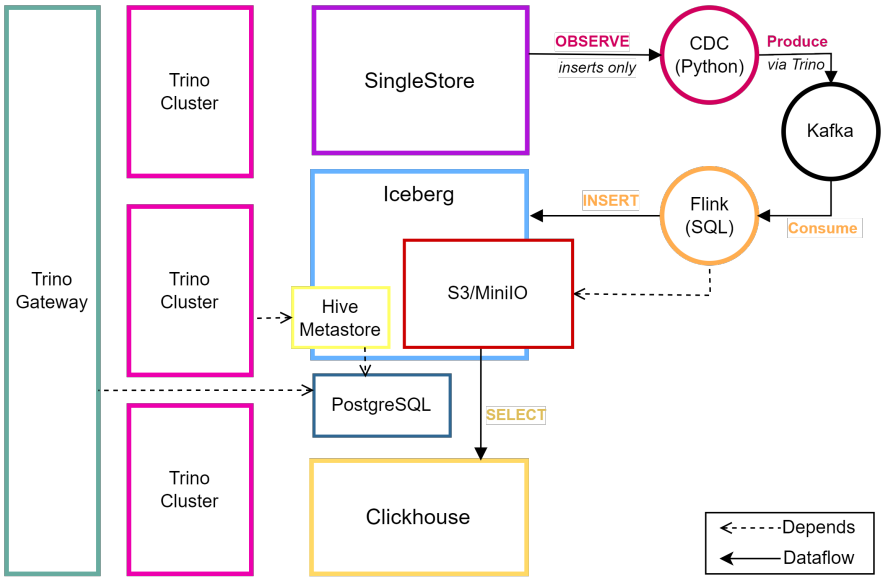


Figure 4.3: *Apache Flink - Proof of Concept*

- PartitionId - partition where the current record is written;
- Type - the type of change (e.g. INSERT, UPDATE, DELETE);
- TxId - unique ID of a transaction within a database;
- TxTimestamp - the commit timestamp of the record;
- InternalId - the internal identifier for tables that do not have primary keys, which allows the identification of records without a primary key. InternalId is unique across partitions.

The OBSERVE query is an always-running query executed in Python, based on the example provided by SingleStore Labs [71]. The rows extracted are then published on Kafka, and the processed offset is saved on an auxiliary table.

Once data is in Kafka, Flink SQL creates a job that extracts from Kafka and inserts it into Iceberg using the Flink Hive catalog. Figure 4.4 depicts the data flow and Flink SQL used. The Hive catalog is configured by loading the Hive configuration file into the pod storage. Using these catalogs, Flink SQL can query Iceberg data. For Kafka, a Flink SQL table corresponds to a Kafka topic. Finally, a continuous query [31] starts the job. A continuous query never terminates and produces dynamic results on the destination, which is Iceberg.

4.3 Benchmark

In data processing and decision support systems, evaluating the performance of database systems is crucial for understanding their capacity to handle complex workloads. The TPC-H benchmark [12] stands as a widely recognized standard for assessing such systems, specifically targeting decision support workloads that involve executing complex ad-hoc

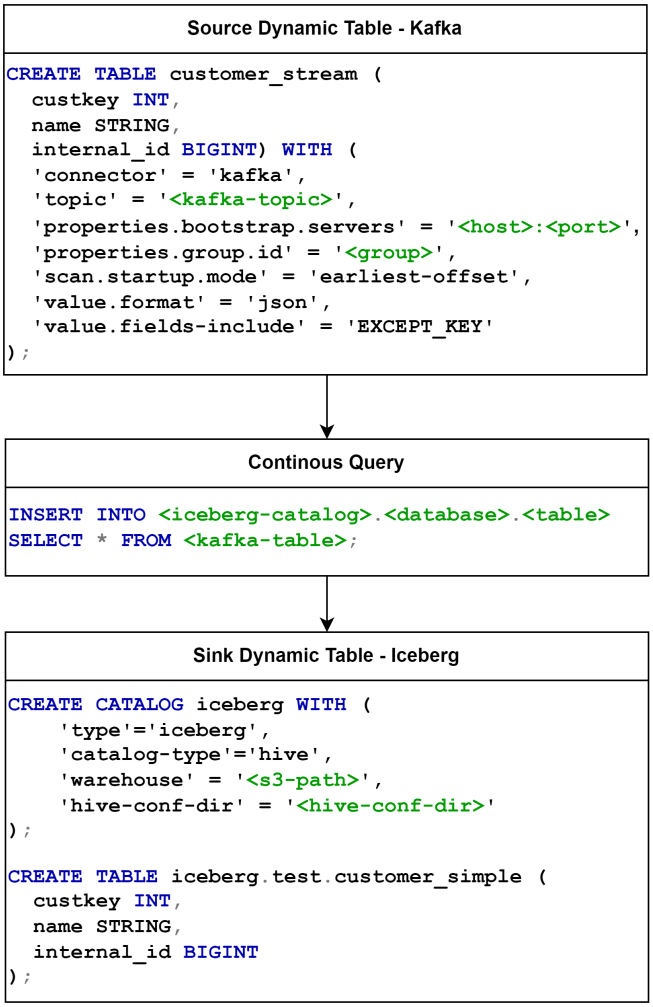


Figure 4.4: *Flink SQL - Proof of Concept*

queries on large datasets. TPC-H simulates real-world business environments with extensive data analysis, including data mining, reporting, and trend analysis. The TPC-H benchmark is centered around complex decision-support queries. It includes a Query Set comprising 22 unique queries simulating real-world business operations and a Query Stream, a sequence of these queries executed in a specific order to measure system performance under realistic workloads. The concepts are related as depicted in Figure 4.5.

Even so, the TPC-H benchmark doesn't quite match the banking paradigm, resulting in the creation of a benchmark focused on banking operations that aims to provide a structured and rigorous evaluation of database systems based on the TPC-H. Thus, the schema was framed to the banking context as Customers, Accounts, and Transactions, as depicted in Figure 4.6, inspired by the real dataset to approximate the data to real-world usage. Each table has a fixed number of rows, known as cardinality, because the database size is defined with reference to the scale factor 1GB (GigaBytes). The size of the scale factor is estimated by calculating the estimated data type size to obtain the corresponding number of rows needed to reach the size of 1GB, resulting in the table 4.2 containing the estimated database size. The scale factors planned are 1, 10, 30, 100, and 300 GB.

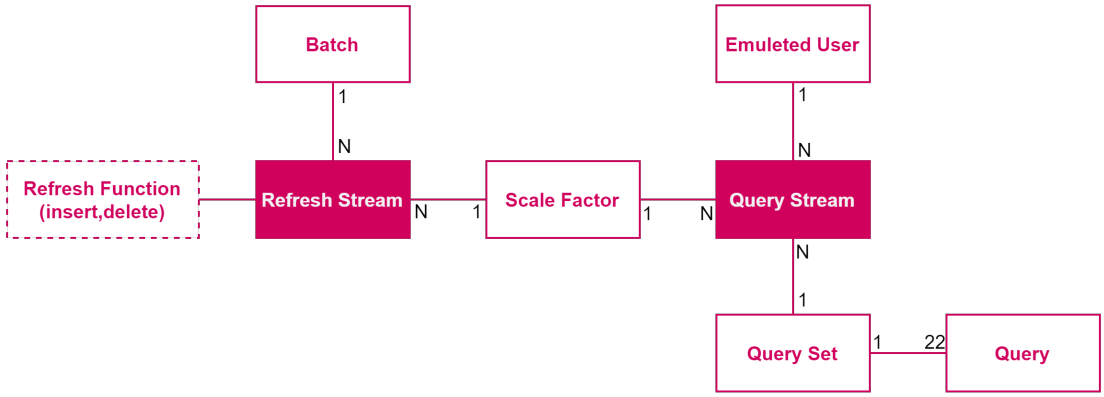


Figure 4.5: *TPC-H concepts relations.*

Table 4.2: Estimated database size - Scale Factor 1

Table Name	Cardinality (rows)	Length of Typical Row (bytes)	Typical Table Size (MB)
Customer	4101	528	2.0650177
Account	8202	1 392	10.88827515
Customer Account Relation	12 303	1056	12.3901062
Transactions	492 120	2072	972.4356079
Total	516 726	-	≈ 1000

Regarding indexing, Table 4.3 presents the indexing keys used in each database for the executions made. In each column referent to keys, there can be the column name, "ND"(Not Defined) or "NA"(meaning that the key is not available on that database). Two indexing configurations were tested for Iceberg: sorted and unsorted, due to the datalake storage being object storage (MinIo) and not a regular SSD. The objective of testing the data sorted is to see how it impacted the scan of range keys.

Table 4.3: Datasets Table Indexes.

Database System	Table Name	Primary Key	Order/Sort Key	Shard Key
SingleStore	Customer	ND	createdAt	customerId
SingleStore	Account	ND	createdAt	accountId
SingleStore	Customer Account Relation	ND	createdAt	id
SingleStore	Transactions	ND	transactionDateTime	customerId
Iceberg (Sorted)	Customer	ND	createdAt	NA
Iceberg (Sorted)	Account	ND	createdAt	NA
Iceberg (Sorted)	Customer Account Relation	ND	createdAt	NA
Iceberg (sorted)	Transactions	ND	transactionDateTime	NA
Iceberg (unsorted)	Customer, Account, Customer Account Relation and Transactions	ND	ND	NA
Clickhouse	Customer	ND	customerId, createdAt	NA
Clickhouse	Account	ND	accountId, createdAt	NA
Clickhouse	Customer Account Relation	ND	id, createdAt	NA
Clickhouse	Transactions	ND	customerId, transactionDateTime	NA
Cockroach	Customer	customerId	ND	NA
Cockroach	Account	accountId	ND	NA
Cockroach	Customer Account Relation	id	ND	NA
Cockroach	Transactions	transactionId	ND	NA
DuckDB	Customer	ND	ND	NA
DuckDB	Account	ND	ND	NA
DuckDB	Customer Account Relation	ND	ND	NA
DuckDB	Transactions	ND	ND	NA

4.3.1 Query Definitions

In this benchmark framework, several key variables define the queries to evaluate database query performance better. The queries always retrieve transactions for one or more customers. The dimension range and fact range will determine the scope of the query, respectively, the number of clients that the query includes, and the range of the days for

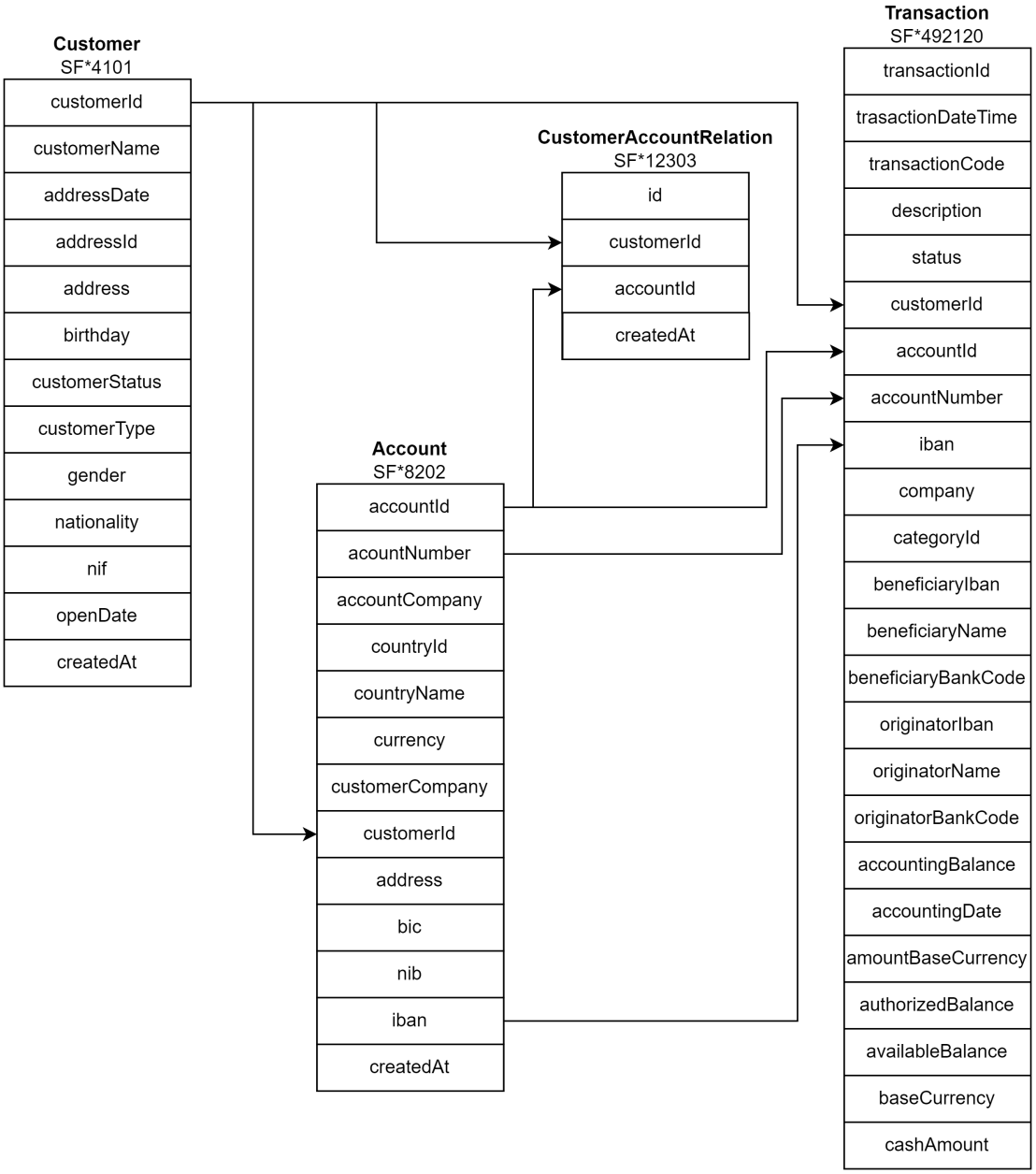


Figure 4.6: *Banking benchmark schema.*

the transactions. The dimension range differs from scale factor (SF) to scale factor, given that the number of customers varies. Other variables are the dimension and fact rules that apply business filters that affect previously established ranges. Finally, metrics (M) and arrangements (A) increase the query's complexity. The metrics are defined as a set that includes possible operations like "avg/sum", and "count", providing flexibility in how data is aggregated or analyzed. The arrangements set controls how the query results are structured, offering options such as "group by"or "order by"and allowing for various sorting and grouping configurations. As a result, the query at the Listing 4.1 is defined by variables at Listing 4.2. Given the query definition variables, for scale factor 1, a query set contains 20 queries based on dimension and fact range values. The rules, metrics, and arrangement increase the complexity of those 20 queries.

Listing 4.1: Query template.

```
SELECT <M> FROM singlestore.sf1.transaction t WHERE t.customerId<=<DIM_
  RANGE> AND t.transactionDateTime>=date'<date0>' AND t.transactionDateTime
  <=date '<date0>' + interval '<FACT_RANGE>' day <A>
```

Listing 4.2: Query definition variables

```
M = {m | m is metric} <=> {none,avg,sum,count}
A = {a | a is arrangement} <=> {none,group by,order by}
DIMENSION RANGE = {1,10,100,1000,4101} (SF=1)
FACT RANGE = {1,7,30,60} days
DIMENSION RULE = {none, selection rule 1, (...)}
FACT RULE = {none, selection rule 1, (...)}
```

4.3.2 Load and Performance Testing

Our benchmark is divided into Load Test and Performance Test, just like TPC-H. The load test is where all tables are populated, given that all data was generated according to predetermined characteristics. Each customer has three accounts, two of which are owned only by him, and the other are shared with another customer. When it comes to the transactions, for each account, 30 transactions per month were generated with random dates (Java-coded). In this case, we made queries for two months of transaction history, generating 60 per customer. The performance test is split into two parts, a power test and a throughput test, which differ in user concurrency, respectively, one and multiple users. In our benchmark, the concurrency was segmented into 4 (four) different levels:

- p1 - one user executing a query;
- p5 - five users executing a query sequentially 10 times;
- p10 - ten users executing a query sequentially 10 times;
- p50 - fifty users executing a query sequentially 10 times.

These concurrency levels will be executed for each query using all scale factors. The execution ensures that the 5 users start executing the query sequentially simultaneously so that the system has the maximum of x user querying concurrently, where x corresponds to the concurrency level.

5 Results

This chapter summarizes the key findings from the comprehensive benchmarking study conducted across multiple databases, evaluating their performance and overall suitability for modern data-intensive applications.

5.1 Baseline Benchmark: Results and Evaluation

5.1.1 Database comparison

The results are meant to be a baseline comparison between database systems, demonstrating that SingleStore, the primary database under evaluation, prevails due to its unique architecture and translytical paradigm. Even so, the other paradigms help us understand how other approaches can complement and configure SingleStore for our workloads. The database population for the benchmark execution was a scale factor of 1GB, with the variables of query definitions in the Listing 5.1.

Listing 5.1: Query definition variables applied.

```
M = none
A = none
DIMENSION RANGE = {1,10,100,1000,4101} (SF=1)
FACT RANGE = {1,7,30,60} days
DIMENSION RULE = none
FACT RULE = none
```

The systems under test (SUT) are Iceberg, SingleStore, Clickhouse, DuckDb, and CockroachDB, highlighted in blue in Figure 5.1. For SingleStore, Clickhouse, and Cockroach, queries are executed directly to the database, via Trino and Starburst. Iceberg will be tested using Trino and Starburst. Finally, DuckDB is tested using the JDBC driver for DuckDB. These are the database systems being evaluated.

Figure 5.2 is a graph with the average response time (left y-axis) and the errors (right y-axis) for the p10 execution, 10 concurrent users executing a single query sequentially 10 times. The results corresponding to queries (x-axis) are divided on the graph into sections representing clients' dimensions range (1, 10, 100, 1000, and 4101). Its purpose is to clarify

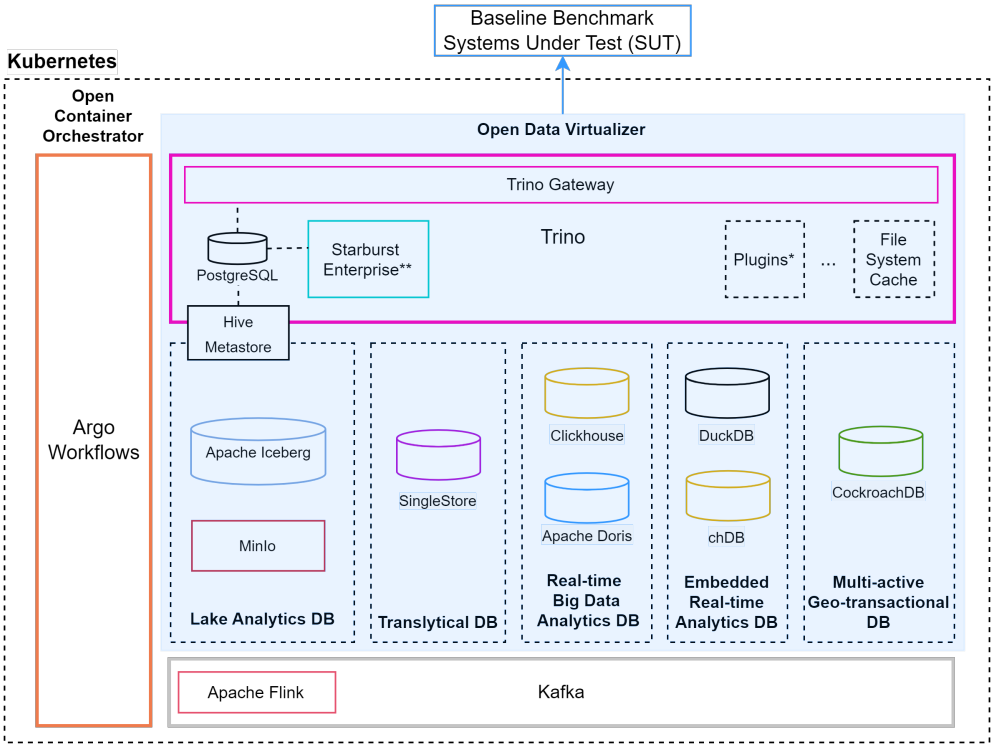


Figure 5.1: *Systems under test for presented results.*

sudden variations in response time that are correlated to the change of dimension range (clients) where, at the same time, the fact range (days) is reset (e.g. 1-60 represents 1 client with a query span of 60 days and 1000-7 represents 1000 clients with a query span of 7 days). The graph is then divided into four graphs: queries without virtualization, via Trino, via Starburst, and errors occurred overall executions, which correspond respectively to figures 5.3, 5.4, 5.5 and 5.6.

Starting by analyzing our reference database, SingleStore, without virtualization, is the overall winner except for the last four queries (dimension range of 4101 clients), where it starts to be contested, especially on query 20. Executing the queries via Starburst and Trino is significantly slower, ranging from 4 to 30 times slower for Starburst and from 0 to 51 times slower for Trino. It's important to point out that Trino queries mirror SingleStore behavior on a larger scale, and after query 18, Trino starts to converge to SingleStore response time. As for Starburst,it stops mirroring after query 14 spikes as fact range (days) increase for both dimensions ranges 1000 and 4101 clients.

CockroachDB starts by competing with Singletore for dimensions range of 1 and 10 clients, and in some cases, it is the fastest for 1 client dimension queries. This gain turns into a loss, growing exponentially from query 10 to 20, exactly when queries become more analytical (OLAP), including all the clients in the dataset. As for overhead, via Starburst and Trino, there existed a significant loss in response time relative to the direct access, which started to decrease as the dimension increased, becoming a gain in the 1000 and 4101 range. This means virtualization outperforms CockroachDB resource management in an analytical context, removing response time degradation on queries 15, 16, 18, 19 and outperforming

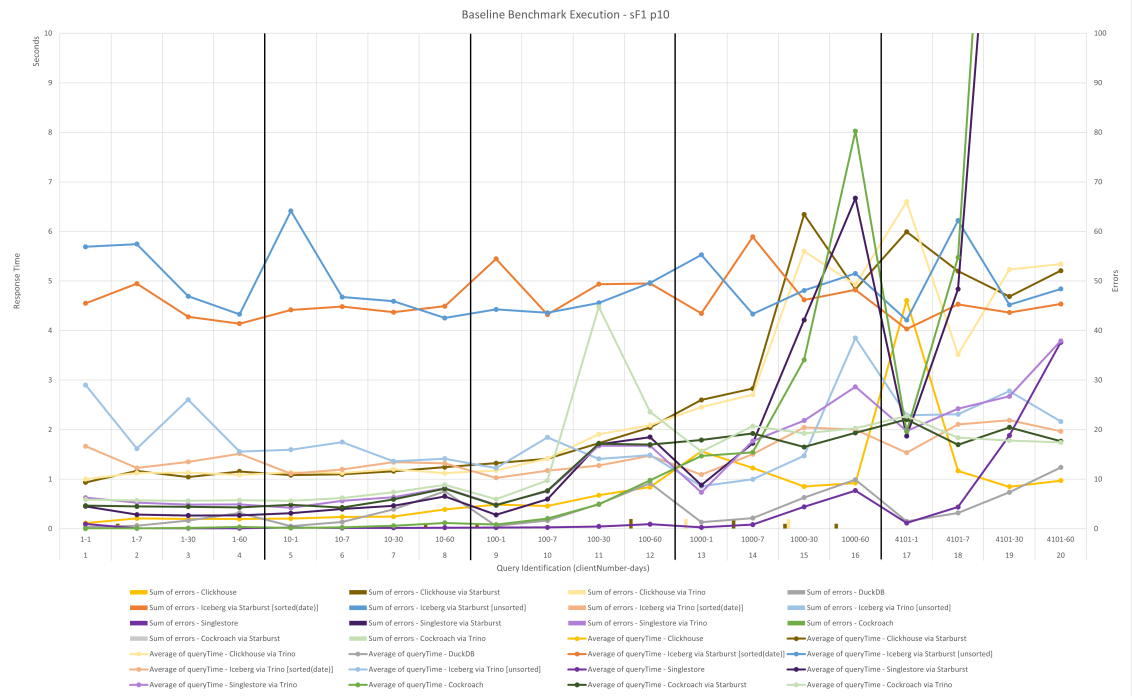


Figure 5.2: *Benchmark execution - sF1 p10.*

SingleStore on query 20, almost 2 times faster than CockroachDB direct access. As for Starburst versus Trino, their performance was very similar, except for queries 11 and 12, where Trino spiked.

Clickhouse, being a database focused on OLAP, is overall slower than SingleStore except for queries 19 and 20, respectively, with 55% and 74% gain in response time. It's almost 2 times faster for those queries, possibly indicating a change in the growth of the data being processed. In the graph, query 17 response time has a noticeable variation explained by the indexing and file organization. Knowing that the index is responsible for selecting the granules, in this case, it is defined as a client ID and transaction date, and the query filters only by the date and not by the client. This forces the engine to search all the granules for specific data. Such behavior doesn't extend to queries 18, 19, and 20, given that they select a bigger fact range. Further addressing the preceding queries would imply extending cluster memory limits. Starburst and Trino visualizations performed similarly relative to Clickhouse's direct access, ranging from 2 to 6 times slower.

DuckDB had the most consistent response time within each dimension range, where time increased as the Fact range increased (days). Being faster than SingleStore on the last dimension of 4101 clients, this dominance is contested only by Clickhouse on query 20. Additionally, the embedded database only surpassed the 1-second mark on query 20, which was still one of the best response times.

Iceberg was compared not only on the engine Starburst and Trino but also on the dataset configuration, sorted and unsorted. The sorted configuration has slightly better results overall, although not significant enough to determine who performs better. Comparing the engines, Trino is better than Starburst, from nearly 25% to 85% gain in response time.

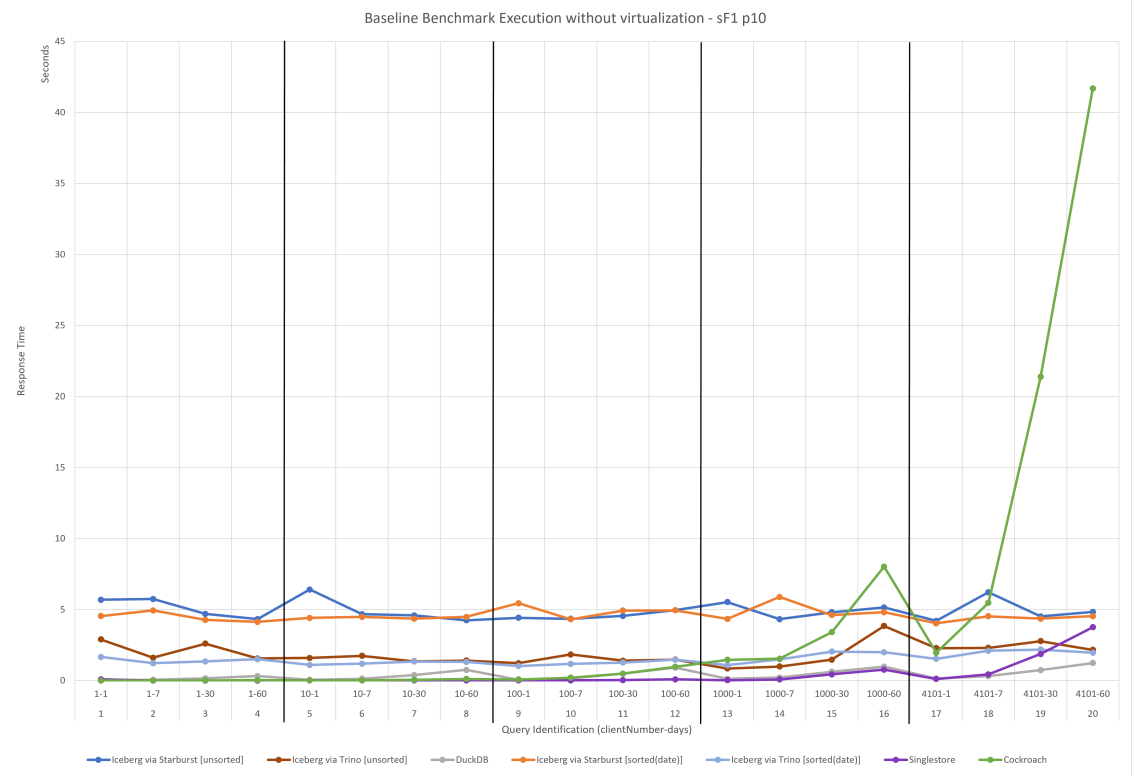


Figure 5.3: *Benchmark execution direct access (without virtualization) - sF1 p10.*

The p50 benchmark execution, Figure 5.7, demonstrates how the systems behave under more concurrent user access that resulted in a degradation of service in general, while SingleStore only started to lose performance after query 12, the remaining databases lost performance from the start.

Clickhouse lack of memory from p10 execution rapidly escalated, resulting in the server crashing instantly for direct access, and via Trino and Starburst, the server stood more load before crashing.

SingleStore via Starburst p50 had errors on queries 13 and 17 due to a coordinator crash. These match query response time degradation when compared to p10 results.

Iceberg Sorted also resulted in a coordinator crash for both Trino and Starburst, although more present on Trino for query 19, resulting on Trino's response time getting closer to Starburs's.

5.1.2 SingleStore Dominance

SingleStore's position as the top performer in this benchmark is due to its translytical capabilities and some advanced technical optimizations. For instance, the core engine is partially built in C++ for low-level resource control, allowing the database to handle high transactional volume and queries with real-time performance. By contrast, Java-based data systems have less control over machine-level resources.

Furthermore, the SingleStore data is stored in columnstore tables for this benchmark, where some query processing operations are executed directly on the encoded data. This data is

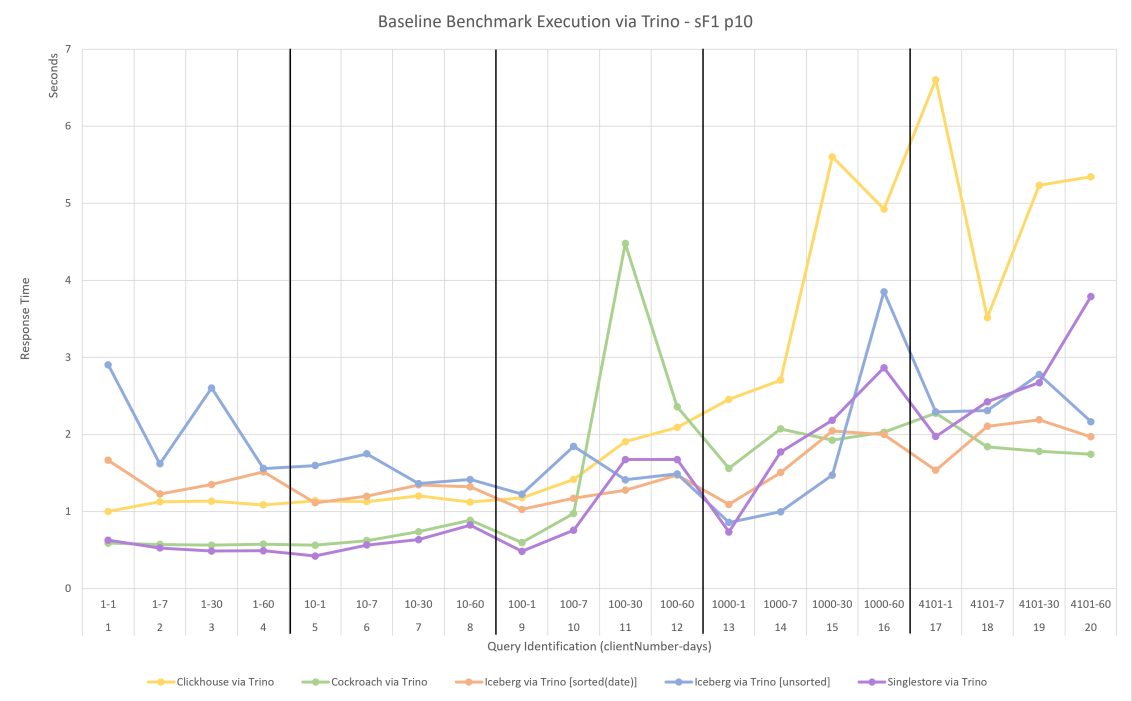


Figure 5.4: *Benchmark execution via Trino - sF1 p10.*

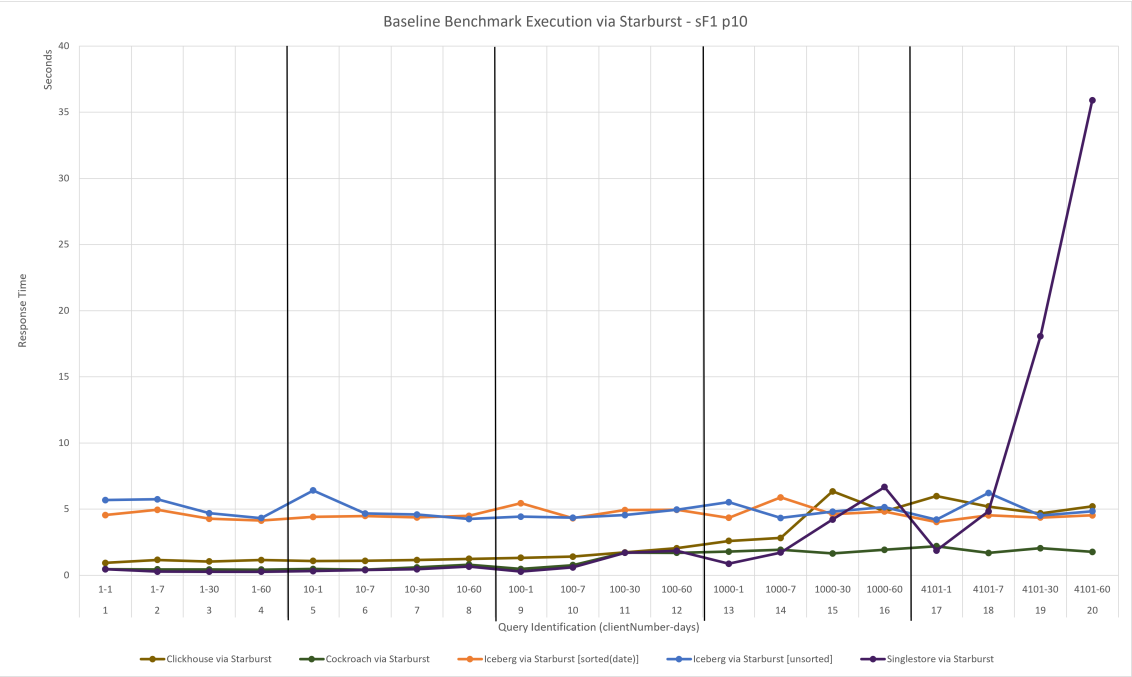


Figure 5.5: *Benchmark execution via Starburst - sF1 p10.*

processed using vectorized execution, where batches of data are algebraically aggregated without decompression at the CPU level. These loops are more efficient on modern CPUs than row-at-a-time processing, resulting in lower instruction count, better cache usage, and improved processor instruction pipeline efficiency. Furthermore, certain operations (e.g., filters and aggregations) can be executed through special techniques like Single-Instruction, Multiple-Data (SIMD) instructions on processors that support the Intel AVX2 instruction set. Although SIMD support is not required by SingleStore, even without it,

Figure 5.6: *Benchmark execution Errors - sF1 p10.*

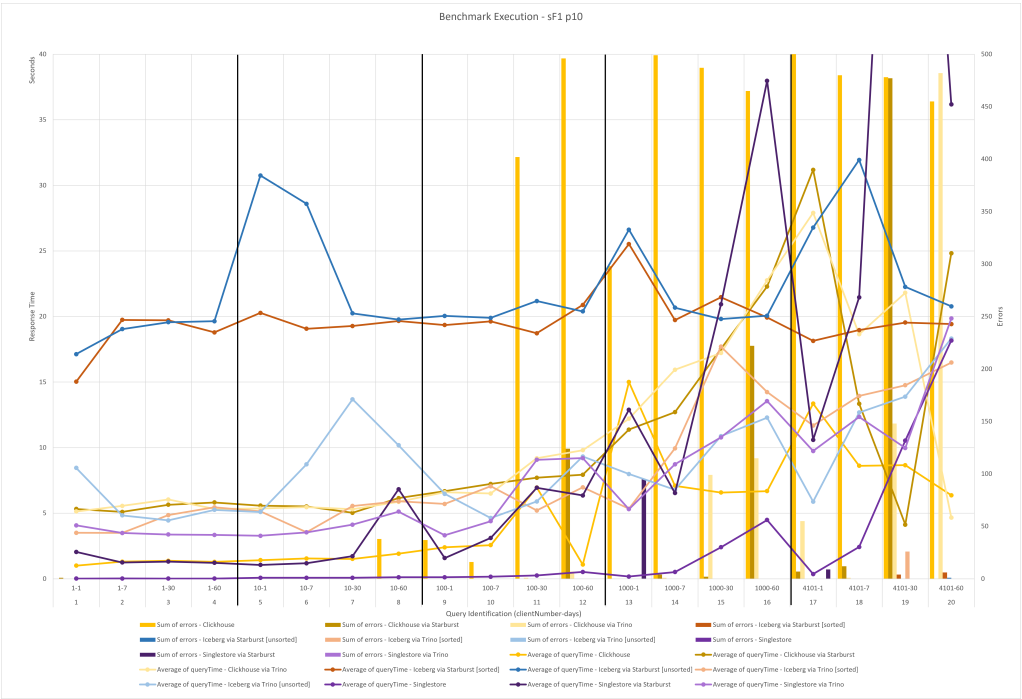


Figure 5.7: *Benchmark execution - sF1 p50.*

the performance of operations on encoded data speeds up to 30 times and gains nearly 2 to 3 times due to SIMD.

SingleStore parallel architecture allows the use of flexible parallelism instead of classic parallelism. This allows further subdivision of database partitions, enabling multiple cores working on behalf of the query to process different sub-partitions concurrently. In classic parallelism, one core can only access one partition at a time, not existing sub-partitions.

Finally, the Linux disk management cache provides a low-level mechanism to process columnstore disk-intensive tables at memory-comparable speeds.

5.1.3 Routing Mechanism

Trino gateway was tested as a routing mechanism to understand the cost of adding this extra layer using p10 concurrence. Figure 5.8 is a diagram with and without the SingleStore, Clickhouse, and Iceberg gateway. This indicates that routing to a Trino (or Starburst) cluster via Trino gateway has nearly no impact, meaning that it's a viable option to create special-purpose Trino Clusters that route based on workload type in our architecture.

Additionally, the gateway can route the queries to the least loaded backend (count-based) instead of round-robin by changing the module loaded and even creating a custom module.

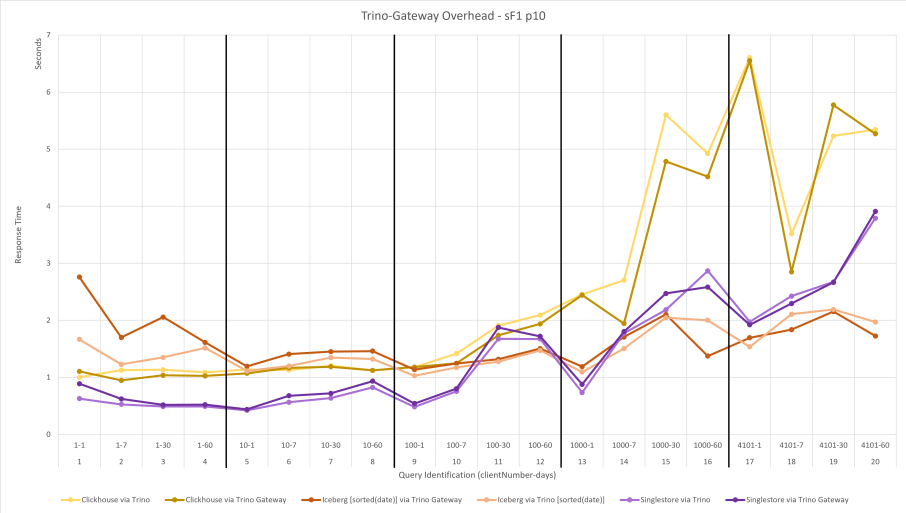


Figure 5.8: *Trino-gateway Overhead - sF1 p10.*

5.2 Summary

Despite its exploratory nature, this thesis has empirically compared five modern distributed data systems using an adapted version of a standard benchmarking methodology (TPC-H) and articulated the essential outline of a Scale Processing Architecture as a powerful enabler for Digital Advantage Leap.

Complementarily, this study reviewed in detail the available information on the history of modern and open data system paradigms providing a first systematic baseline benchmark for their comparison to the best of our knowledge.

This research has shed a contemporary light on the contentious issue of data paradigm selection (Lakehouse, DataWharehouse, RT Olap, Translytical, Streaming, Embedded, Geo-Transactional) and explored an alternative technological pathway for strategic de-risking and optionality.

Furthermore, the findings of this investigation provide empirical confirmation of the benefits of a translytical paradigm, especially for emerging software architectures endowed with reactive and intelligent multi-agency.

Ultimately, the project lays the foundation for continued advancements in the Virtualized/Translytical symbiotic paradigm, where its findings have the potential to influence

both theoretical exploration and practical implementation.

Moreover, the Kubernetes native laboratory provides the means for complex mechanism and configuration combinatorial testing, constituting a critical enabler for exploiting optimized configurations, accessing emerging technology and anticipating disruptive trends.

Noticeably, it turns up especially relevant for financial organizations struggling to navigate the balance between value, innovation, and risk on unprecedented seismic change and opportunity levels. (e.g. Gen AI [47])

Assuredly there are several thrusts and directions that the present work encourages natural expansion, as explained in the following sections.

6 Conclusions and Future Work

This section outlines the conclusions and several avenues for future research that could build upon the findings and methodologies presented in this thesis. Each proposed area aims to enhance our current architecture's scalability, efficiency, and applicability.

6.1 Conclusions

In conclusion, this thesis has empirically contrasted open data system paradigms, thereby providing a systematic account of their significance. It has also articulated the essential outline of a Scale Processing Architecture as a powerful enabler for the Digital Advantage Leap. Ultimately, the research lays the foundation for continued advancements in Trino/SingleStore symbiotic arrangements, where its findings have the potential to influence both theoretical exploration and practical implementation.

The present study also established an empirical framework and deployed an engineering laboratory to enhance diagnostic and design capabilities along four major axes:

- Benchmark Automation - Use of workbench to execute the benchmark tool, automating the process of running tests, gathering results, and managing configurations to ensure repeatability and consistency in the benchmarking process;
- Optimization and Control Design - fine-tuning configurations, adjusting workload distributions, and controlling performance to achieve optimal efficiency in the benchmark environment;
- Patterns and Practices Prototyping - exploration and testing of various system architecture patterns and best practices to identify effective configurations and approaches for large-scale data processing in the benchmarking context;
- Champion-Challenger Contrast - SingleStore as the champion, being challenged by alternative systems or configurations. The champion-challenger methodology compares the performance of a baseline system (SingleStore) against new or modified systems to identify improvements and validate new approaches.

The code and tools used as part of this research are available for public access and future development at the GitHub repository [54]. I encourage future researchers to explore and contribute to the project.

6.2 Future work

Our conviction is that the preceding achievements constitute a critical springboard for future research along the following directions.

One promising direction is to conduct benchmarking studies with larger scale factors, including larger data and query sets. This would involve testing the database systems under varying conditions, including different data sizes, workloads, and user scenarios. By doing so, we can gain insights into our approach's performance scalability and resilience while fine-tuning each system, thereby validating its applicability in real-world applications.

Optimization of our architecture would include the development of a custom Trino-SingleStore connector based on what was learned from the internals of SingleStore and other database systems. This connector would facilitate seamless data integration between Trino, a powerful distributed SQL query engine, and SingleStore, our best-performing database. Future work could focus on developing this connector to optimize query execution across both environments according to the workloads.

SingleStore can be further complemented with a data materialization mechanism for enhanced response time and reduced resource footprint. Hence, Apache Flink and Kafka servicing SQL continuous queries were rehearsed to fulfill these desiderata with good preliminary results. Future work would implicate performing tests at scale to validate such implementation.

The current results were complemented by an in-depth analysis of the Apache Doris architecture and available benchmarks, suggesting it to be a robust rival to Singlestore. It is an efficient and scalable analytics database that provides a seamless and natural architectural fit. More research is needed to determine the best coupling configuration.

Finally, graph databases are a rapidly growing branch of technology, particularly relevant for knowledge-intensive applications and other domains that require complex relationship modeling and pattern matching. By pursuing contrastive benchmarking on specialized systems for graph querying on either native (e.g., Neo4j [62]) or extension (e.g., Presto) form, we could devise patterns and practices for optimal incorporation.

Bibliography

[1] *Airlift*. URL: https://github.com/airlift/airlift (cit. on p. 31).

[2] M. M. Astrahan, M. W. Blasgen, D. D. Chamberlin, K. P. Eswaran, J. N. Gray, P. P. Griffiths, W. F. King, R. A. Lorie, P. R. McJones, J. W. Mehl, G. R. Putzolu, I. L. Traiger, B. W. Wade, and V. Watson. "System R: relational approach to database management". In: 1.2 (June 1976), 97–137. ISSN: 0362-5915 (cit. on p. 1).

[3] A. P. Authors. *Argo Project*. 2023. URL: https://argoproj.github.io/ (cit. on pp. 9, 10).

[4] C. Cardas, J. F. Aldana-Martín, A. M. Burguenõ-Romero, A. J. Nebro, J. M. Mateos, and J. J. Sánchez. "On the performance of SQL scalable systems on Kubernetes: a comparative study". In: (2022) (cit. on pp. 22, 23).

[5] D. D. Chamberlin and R. F. Boyce. "SEQUEL: A structured English query language". In: (1974) (cit. on p. 1).

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: (2006) (cit. on p. 3).

[7] I. ClickHouse. *chDB*. 2024. URL: https://clickhouse.com/chdb (cit. on p. 29).

[8] I. ClickHouse. *ClickHouse*. 2024. URL: https://clickhouse.com/ (cit. on p. 29).

[9] E. F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: (1970) (cit. on p. 1).

[10] F. Coelho, J. Paulo, R. Vilaça, J. Pereira, and R. Vilaça. "HTAPBench: Hybrid Transactional and Analytical Processing Benchmark". In: (Apr. 2017) (cit. on p. 29).

[11] I. Confluent. *Your Guide to the Apache Flink® Table API: An In-Depth Exploration*. 2014-2024. URL: https://www.confluent.io/blog/getting-started-with-apache-flink-table-api/ (cit. on p. 40).

[12] T. P. P. Council. *TPC-H Benchmark*. http://www.tpc.org/tpch/. Accessed: 2023-09-14. 2023 (cit. on pp. 22, 44).

[13] V. A. Data. *Volt DB*. 2009. URL: https://www.voltactivedata.com/ (cit. on p. 4).

[14] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: (2004) (cit. on p. 3).

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. "Dynamo: Amazon's Highly Available Key-value Store". In: (2007) (cit. on p. 3).

[16] I. Dremio. *Dreimo*. 2023. URL: https://docs.dremio.com/current/ (cit. on pp. 16, 17).

[17] A. S. Foundation. *Apache Nutch*. 2004-2023. URL: https://nutch.apache.org/ (cit. on p. 3).

[18] A. S. Foundation. *Apache Hadoop*. 2008-2023. URL: https://hadoop.apache.org (cit. on pp. 3, 11).

[19] A. S. Foundation. *Apache Hadoop Docs - HDFS Architecture*. 2008-2023. URL: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html (cit. on pp. 11, 13).

[20] A. S. Foundation. *Apache Hadoop Docs - YARN*. 2008-2023. URL: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html (cit. on pp. 11, 14).

[21] A. S. Foundation. *Apache Kafka*. 2011. URL: https://kafka.apache.org/ (cit. on p. 4).

[22] A. S. Foundation. *Apache Giraph*. 2011-2020. URL: https://giraph.apache.org/ (cit. on p. 25).

[23] A. S. Foundation. *Apache Lucene*. 2011-2024. URL: https://lucene.apache.org/ (cit. on p. 3).

[24] A. S. Foundation. *Apache Zookeeper*. 2020. URL: https://zookeeper.apache.org/ (cit. on p. 11).

[25] A. S. Foundation. *Apache Drill*. 2022. URL: https://drill.apache.org/ (cit. on p. 22).

[26] A. S. Foundation. *Apache Hive*. 2023. URL: https://hive.apache.org/ (cit. on pp. 22, 29).

[27] A. S. Foundation. *Apache Iceberg Documentation - Overview*. 2023. URL: https://iceberg.apache.org/spec/#overview (cit. on p. 36).

[28] A. S. Foundation. *Apache Kafka*. 2023. URL: https://kafka.apache.org/ (cit. on pp. 18, 28).

[29] A. S. Foundation. *Apache Doris*. 2024. URL: https://doris.apache.org/ (cit. on p. 29).

[30] A. S. Foundation. *Apache Doris - TPC-H Benchmark*. 2024. URL: https://doris.apache.org/docs/benchmark/tpch/ (cit. on p. 29).

[31] A. S. Foundation. *Apache Flink*. 2024. URL: https://nightlies.apache.org/flink/flink-docs-master/docs/dev/table/concepts/dynamic_tables/#dynamic-tables-amp-continuous-queries (cit. on p. 44).

[32] A. S. Foundation. *Apache Maven*. 2024. URL: https://maven.apache.org/ (cit. on p. 31).

[33] D. Foundation. *DuckDB*. 2024. URL: https://duckdb.org/ (cit. on p. 29).

[34] T. L. Foundation. *Kubernetes*. 2024. URL: https://kubernetes.io/ (cit. on p. 9).

[35] T. S. Foundation. *Trino*. URL: https://trino.io/ (cit. on p. 4).

[36] T. S. Foundation. *Trino - Database resource group manage*. URL: https://trino.io/docs/current/admin/resource-groups.html#database-resource-group-manager (cit. on p. 30).

[37] T. S. Foundation. *Trino Gateway*. URL: https://trinodb.github.io/trino-gateway/ (cit. on p. 29).

[38] S. Ghemawat, H. Gobioff, and S.-T. Leung. "The Google File System". In: (2003) (cit. on p. 3).

[39] *GitHub - Trino*. URL: https://github.com/trinodb/trino/tree/master (cit. on p. 31).

[40] Google. *GO*. 2023. URL: https://go.dev/ (cit. on p. 9).

[41] T. P. G. D. Group. *PostgreSQL*. 1996-2024. URL: https://www.postgresql.org/about/ (cit. on p. 2).

[42] T. P. G. D. Group. *Postgresql wire protocol*. 1996-2024. URL: https://www.postgresql.org/docs/current/protocol.html (cit. on p. 38).

[43] D. R. Hipp. *SQLite*. https://www.sqlite.org/. Accessed: 2024-09-30. 2024 (cit. on p. 37).

[44] IBM. *DB2*. URL: https://www.ibm.com/products/db2 (cit. on p. 2).

[45] IBM. *IBM*. 1996-2024. URL: https://www.ibm.com/us-en (cit. on p. 1).

[46] IBM. *2024 Global Outlook for Banking and Financial Markets*. 2024. URL: https://www.ibm.com/thought-leadership/institute-business-value/en-us/report/2024-banking-financial-markets-outlook (cit. on p. 6).

[47] IBM. *Scale Generative ai*. https://www.ibm.com/thought-leadership/institute-business-value/en-us/report/scale-generative-ai. 2024 (cit. on p. 56).

[48] Informix. *Informix*. 1980. URL: https://www.ibm.com/products/informix (cit. on p. 2).

[49] P. J.Sadalage and M. Fowler. *NoSQL Distilled*. Novatec, 2013 (cit. on p. 2).

[50] M. Kleppmann. *Designing Data-Intensive Applications*. O'Reilly, 2017 (cit. on pp. 2, 4).

[51] J. Kreps, N. Narkhede, and J. Rao. "Kafka: a Distributed Messaging System for Log Processing". In: (2011) (cit. on p. 4).

[52] C. Labs. *CockroachDB*. 2024. URL: https://www.cockroachlabs.com/ (cit. on p. 29).

[53] A. Lakshman and P. Malik. "Cassandra - A Decentralized Structured Storage System". In: (2009) (cit. on p. 3).

[54] A. Machado. *Thesis SPA - GitHub repo*. 2024. URL: https://github.com/afonsoamachado/thesis-SPA (cit. on p. 58).

[55] M. M. Matt Fuller and M. Traverso. *Trino: The Definitive Guide 2nd Edition*. O'Reilly, 2022 (cit. on p. 21).

[56] Mckinsey. *How to unlock the full value of data? Manage it like a product*. June 2022. URL: https://www.mckinsey.com/capabilities/quantumblack/our-insights/how-to-unlock-the-full-value-of-data-manage-it-like-a-product?utm_content=buffer93fb8&utm_medium=social&utm_source=linkedin.com&utm_campaign=buffer (cit. on p. 28).

[57] Mckinsey. *The digital Advantage*. June 2023. URL: https://www.mckinsey.com/featured-insights/sustainable-inclusive-growth/charts/the-digital-advantage (cit. on pp. 4, 6).

[58] Meta Platforms, Inc. *Meta Platforms, Inc.* https://about.meta.com/. Accessed: 2024-09-30. 2024 (cit. on p. 23).

[59] Microsoft. *Microsoft SQL Server*. URL: https://www.microsoft.com/pt-br/sql-server/ (cit. on p. 2).

[60] Microsoft. *Azure Cloud Services*. 2024. URL: https://azure.microsoft.com/en-us/free/cloud-services (cit. on p. 41).

[61] R. N. *Gartner Innovation Award 2023*. 2023 (cit. on p. 28).

[62] I. Neo4j. *Neo4j*. 2024. URL: https://neo4j.com/ (cit. on p. 58).

[63] M. Pato. *The ISELthesis LaTeX Template's Manual*. Instituto Superior de Engenharia de Lisboa (ISEL-IPL). 2024. URL: https://github.com/matpato/iselthesis (cit. on p. viii).

[64] M. Pezzini, D. Feinberg, N. Rayner, and R. Edjlali. "Hybrid Transaction/Analytical Processing Will Foster Opportunities for Dramatic Business Innovation". In: (Jan. 2014) (cit. on p. 29).

[65] M. Raasveldt and H. Mühleisen. "DuckDB: an Embeddable Analytical Database". In: (2019) (cit. on p. 37).

[66] H. B. Review. *A Better Way to Put Your Data to Work*. June 2022. URL: https://hbr.org/2022/07/a-better-way-to-put-your-data-to-work (cit. on p. 28).

[67] SAP. *SyBase*. URL: https://www.sap.com/products/acquired-brands/what-is-sybase.html (cit. on p. 2).

[68] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner. "Presto: SQL on Everything". In: *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 2019, pp. 1802–1813. DOI: 10.1109/ICDE.2019.00196 (cit. on pp. 3, 18, 19).

[69] K. Shvachko, H. Kuang, S. Radia, and R. Chansle. “The Hadoop Distributed File System”. In: (2010) (cit. on p. 3).

[70] SingleStore. *SinglesStore Documentation - Observe Query*. https://docs.singlestore.com/cloud/reference/sql-reference/data-manipulation-language-dml/observe/ (cit. on p. 43).

[71] SingleStore. *SinglesStore Labs - CDC out Examples*. https://github.com/singlestore-labs/cdc-out-examples/tree/main/python/cdc/ (cit. on p. 44).

[72] I. SingleStore. *SingleStore*. URL: https://www.singlestore.com/ (cit. on pp. 5, 29).

[73] A. N. Standards. *ANSI SQL*. URL: https://blog.ansi.org/sql-standard-iso-iec-9075-2023-ansi-x3-135/ (cit. on p. 18).

[74] I. Starburst Data. *Starburst*. 2023. URL: https://www.starburst.io/ (cit. on pp. 18, 29).

[75] M. Stonebraker. “The Case for Shared Nothing”. In: (1986) (cit. on p. 2).

[76] M. Stonebraker. *New SQL: An Alternative to NoSQL and Old SQL For New OLTP Apps*. 2011. URL: https://cacm.acm.org/blogs/blog-cacm/109710-new-sql-an-alternative-to-nosql-and-old-sql-for-new-oltp-apps/fulltext (cit. on p. 4).

[77] M. Stonebraker and U. Çetintemel. ““One Size Fits All”: An Idea Whose Time Has Come and Gone”. In: (2005) (cit. on p. 3).

[78] C. Strozzi. “NoSQL: A relational database management system”. In: (1998) (cit. on pp. 2, 28).

[79] Y. Sun, T. Meehan, R. Schlussel, W. Xie, M. Basmanova, O. Erling, A. Rosa, S. Fan, R. Zhong, A. Thirupathi, N. Collooru, K. Wang, S. Agarwal, A. Gupta, D. Logothetis, K. Xirogiannopoulos, A. Dutta, V. Gajjala, R. Jain, A. Palakuzhy, P. Pandian, S. Pershin, A. Saikia, P. Shankhdhar, N. Somanchi, S. Tailor, J. Tan, S. Viswanadha, Z. Wen, B. Chattopadhyay, B. Fan, D. Majeti, and A. Pandit. “Presto: A Decade of SQL Analytics at Meta”. In: 1.2 (June 2023). DOI: 10.1145/3589769. URL: https://doi.org/10.1145/3589769 (cit. on pp. 23–25).

[80] Teradata. *Teradata*. URL: https://www.teradata.com/ (cit. on p. 18).

[81] A. S. The Apache Software Foundation. *Apache Spark Documentation*. URL: https://spark.apache.org/ (cit. on pp. 3, 14).

[82] A. S. The Apache Software Foundation. *Apache Spark Documentation - Cluster Mode Overview*. URL: https://spark.apache.org/docs/latest/cluster-overview.html (cit. on p. 15).

[83] A. D. The Apache Software Foundation. *Apache Dolphin Scheduler Documentation*. 2019 - 2023. URL: https://dolphinscheduler.apache.org/en-us/docs/3.2.0 (cit. on p. 10).

[84] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. "Hive – A Petabyte Scale Data Warehouse Using Hadoop". In: (2010) (cit. on p. 3).

[85] R. Tiwari. *Stream Processing with Apache Flink and MinIO*. https://blog.min.io/stream-processing-with-apache-flink-and-minio/. Accessed: 2024-09-28. 2018 (cit. on p. 40).

[86] N. Yuhanna, A. Katz, M. Gualtieri, K. Monteverde, and J. Barton. "Translytical Architecture 2.0 Evolves To Support Distributed, Multimodel, And AI Capabilities". In: (Nov. 2023) (cit. on p. 29).

[87] M. Zaharia, M. Chowdhury, S. S. Michael J. Franklin, and I. Stoica. "Spark: Cluster Computing with Working Sets". In: (2010) (cit. on p. 3).

@is@a@figure