MMINISTRY OF HIGHER EDUCATION
SCIENTIFIC RESEARCH AND INNOVATION

SULTAN MOULAY SLIMANE UNIVERSITY

NATIONAL SCHOOL OF APPLIED SCIENCES
KHOURIBGA

End of studies dissertation

for obtaining the

State engineer diploma

Sector : Computer Science and Data Engineering

Presented by :
Ziyad **RAKIB**

Participation in the migration from a monolithic architecture to a microservices architecture

Supervised by :
Abdelghani **Ghazdali**
Nidal **Lamghari**

Jury members :

Chairman's surname and first name	Entity	Chairman
Surname and first name of the examiner	Entity	Examiner
Surname and first name of the reporter	Entity	Reporter
Surname and first name of the reporter	Entity	Reporter

Academic Year : **2023**

Sommaire

Dedication

“

TO my mother, who showered me with her support and devoted me with unconditional love. You are for me an example of courage and continuous sacrifice. May this humble work bear witness to my affection, my eternal attachment and may it call upon me your continual blessing,

TO my father, no dedication can express the love, esteem, devotion, and respect I have always had for you. Nothing in the world is worth the efforts made day and night for my education and my well-being. This work is the fruit of your sacrifices that you made for my education and training,

TO my dear brothers, thank you for your love, support, and encouragement,

TO all my dear friends, for the support you have given me, I say

Thank you.

”

- Ziyad

Acknowledgements

First of all, I thank God Almighty for giving us the power to continue and overcome all the difficulties.

I would like to thank **Mr.Abdelghani Ghazdali**, head of the Computer Science and Data Engineering department who was able to ensure the smooth running of the coaching sessions from start to finish.

I would also like to express my sincere thanks to **Professor Nidal Lamghari**, my internal supervisor, for all the advice she gave me and her encouragement.

During this work, I would also like to express my sincere thanks and gratitude to all those who, through their teaching, support and advice, have contributed to the development of this project.

I would like to thank **Mr.Reda EL Mejad**, CEO and co-founder of Izicap and ENSA laureate, for the opportunity to spend my internship within this company.

I would like to thank my supervisor at Izicap, **Mr.Bilal SLAYKI** who patiently supervised me and spared no effort to provide me with the necessary explanations and valuable guidelines. His assistance and prominent advice have been a remarkable contribution to my growth. I also want to thank Nasr, Youness and Anas; the team with whom I worked and who generously are contributing to this work.

I would also like to thank the HR agent **Madame Sanaa ARROUCHE** for her good advice as she is always there to respond to our requests and answer our questions as well as assisting us to the best of her ability.

A special thank you to the members of the juries who honored me by accepting to judge this work and to give me the benefit of their comments and advice.

Finally, my thanks also go to the entire faculty and administrative staff of ENSA Khouribga for the effort they provide to guarantee us good training and to the administrative and technical team for all the services offered.

Résumé

Ce rapport fournit une analyse approfondie du projet complexe entrepris pour faire évoluer la source de données de Izicap. Le projet a nécessité la suppression de MariaDB et la mise en place de Delta Lake, une solution de stockage et de traitement de données hautes performances. De plus, l'architecture monolithique existante a été divisée en microservices utilisant Spring Boot comme backend avec un connecteur approprié pour Delta Lake, et les micro-frontends ReactJS comme frontend.

Le projet présentait de nombreux défis qui nécessitaient l'application de technologies et de stratégies avancées. Celles-ci comprenaient la migration des données, la garantie de la cohérence et de l'exactitude des données, la gestion de la complexité des systèmes distribués et l'intégration de diverses technologies et services. Le rapport décrit les différentes solutions développées pour relever ces défis, notamment l'utilisation d'algorithmes de traitement de données avancés, d'architectures informatiques distribuées et de la conteneurisation.

Malgré les défis rencontrés, le projet reste un travail en cours. Le rapport donne un aperçu des efforts en cours pour améliorer l'évolutivité, les performances et la fonctionnalité globale du projet.

Mots clés : Projet, Source de données, Delta Lake, Monolith, Microservice, Spring Boot, ReactJS, Micro-Frontend, Back-End, Front-End, Données, Évolutivité, Performance.

Abstract

This report provides an in-depth analysis of the complex project undertaken to scale the data source of Izicap. The project required the removal of MariaDB and implementation of Delta Lake, a high-performance data storage and processing solution. In addition, the existing monolithic architecture was divided into microservices utilizing Spring Boot as the backend with a suitable connector to Delta Lake, and ReactJS micro-frontends as the frontend.

The project presented numerous challenges that required the application of advanced technologies and strategies. These included data migration, ensuring data consistency and accuracy, managing the complexities of distributed systems, and integrating various technologies and services. The report outlines the various solutions developed to address these challenges, including the use of advanced data processing algorithms, distributed computing architectures, and containerization.

Despite the challenges encountered, the project remains a work in progress. The report provides insights into the ongoing efforts to improve the project's scalability, performance, and overall functionality.

Keywords: Project, Data Source, Delta Lake, Monolith, Microservice, Springboot, ReactJS, Micro-frontends, Back-End, Front-End, Data, Scalability, Performance.

ملخص

يقدم هذا التقرير تحليلاً متعمقًا للمشروع المعقد الذي تم إجراؤه لتوسيع نطاق مصدر بيانات الشركة. يتطلب المشروع إزالة MariaDB وتنفيذ Delta Lake ، وهو حل تخزين ومعالجة بيانات عالي الأداء. بالإضافة إلى ذلك ، تم تقسيم المونوليث الحالي إلى خدمات مصغرة باستخدام Spring Boot كواجهة خلفية مع موصل مناسب لـ Delta
Lake ، وواجهة ReactJS كواجهة أمامية. قدم المشروع العديد من التحديات التي تطلبت تطبيق التقنيات والاستراتيجيات المتقدمة. وشمل ذلك ترحيل البيانات ، وضمان اتساق البيانات ودقتها ، وإدارة تعقيدات الأنظمة الموزعة ، ودمج التقنيات والخدمات المختلفة. يحدد التقرير الحلول المختلفة التي تم تطويرها لمواجهة هذه التحديات ، بما في ذلك استخدام خوارزميات معالجة البيانات المتقدمة ، وبنى الحوسبة الموزعة ، والحاويات. على الرغم من التحديات التي واجهتها ، لا يزال المشروع قيد التنفيذ. يقدم التقرير رؤى حول الجهود الجارية لتحسين قابلية المشروع للتوسع والأداء والوظائف العامة

كلمات مفتاحية : المشروع، مصدر البيانات، Delta Lake ، المونوليث، الميكروسيرفس، Springboot ، ReactJS ، الميكروواجهات، الخلفية، الواجهة الأمامية، البيانات، القابلية للتوسع، الأداء. Delta Lake ، Springboot ، ReactJS ، الميكروواجهات.

Contents

List of Figures

List of Tables

Table 1: List of Abbreviations

API	Application programming interface
VSC	Visual Studio Code
DL	Delta Lake
REST	RepresEntational State Transfer
AWS	Amazon Web Services
GCP	Google Cloud Platform
AZR	Microsoft Azure
S3	Simple Storage Service
IAM	Identity and Access Management
MinIO	Minimal Object Storage
SQL	Structured Query Language
DB	Database
ACID	Atomicity, Consistency, Isolation, Durability
OLTP	Online Transaction Processing
OLAP	Online Analytical Processing
ReactJS	React JavaScript
SPA	Single Page Application
JS	JavaScript
CSS	Cascading Style Sheets
HTML	Hypertext Markup Language
UI	User Interface
API	Application Programming Interface
Spark	Apache Spark
Hadoop	Apache Hadoop
HDFS	Hadoop Distributed File System
YARN	Yet Another Resource Negotiator
MapReduce	MapReduce Programming Model
Metadata	Data about Data
ETL	Extract, Transform, Load
ELT	Extract, Load, Transform
CRM	Customer Relationship Management
RDBMS	Relational Database Management System
SPI	Server Provider Interface

General Introduction

Le monde des données continue de se développer à un rythme sans précédent. Cela a conduit au besoin de systèmes de stockage et de gestion de données efficaces capables de suivre cette croissance. Dans notre quête d'une meilleure solution, nous avons rencontré le problème de l'évolutivité avec notre système de gestion de base de données actuel, MariaDB. En conséquence, nous nous sommes lancés dans un projet visant à trouver une source de données évolutive qui répondrait à nos besoins. Après de nombreuses recherches et réflexions, nous avons décidé d'utiliser Delta Lake avec un stockage S3.

Cependant, nous avons rencontré plusieurs défis lors de la mise en œuvre de Delta Lake. L'un des principaux problèmes auxquels nous avons été confrontés était le manque de popularité et de documentation de Delta Standalone. Il s'est également avéré lent et ne pouvait pas être interrogé avec SQL. Par conséquent, nous avons opté pour Trino, un moteur de requête SQL distribué qui pourrait facilement interagir avec Delta Lake.

Un autre défi important était l'architecture monolithique de notre système, qui entravait sa flexibilité et son évolutivité. Par conséquent, nous avons décidé de passer aux microservices pour rendre notre système plus efficace et flexible.

Dans ce rapport, nous donnerons un aperçu de la société Izicap au chapitre 1, suivi d'une explication de Delta Lake et de ses défis de mise en œuvre au chapitre 2. Au chapitre 3, nous discuterons de Trino et de la manière dont il nous a aidés à surmonter les défis auxquels nous étions confrontés. avec le lac Delta.

Chapter 1

Izicap

Introduction

1.1 About

Izicap is pioneering the use of payment card data and turning it into powerful customer knowledge & business insights for merchants, allowing them to run their own loyalty programs and digital marketing campaigns. These marketing services, provided by acquirers using Izicap's SaaS solutions, quickly restore growth to merchant businesses by building their customers (the card-holders) spending and stickiness. Izicap's innovative card-linked CRM & Loyalty solution gives acquirers a competitive edge by monetising their payment transactions data, generating new sources of income and improving their retention capabilities. After having solidly established itself in France thanks to partnerships with Groupe BPCE and Crédit Agricole, Izicap partnered with Nexi, the leading acquirer & Fintech in Italy and has joined Mastercard's StartPath program with an aim to dramatically expand its worldwide reach. Izicap partners with leading payment solution providers such as Ingenico, Verifone, Poynt and PAX, and makes its card-linked CRM & Loyalty solution available on the most popular and innovative payments terminals

1.2 Organizational chart

Information on Izicap

Conclusion

Chapter 2

Delta Lake

Introduction

Data warehouses and data lakes are the most common central data repositories employed by most data-driven organizations today, each with its own strengths and tradeoffs. For one, while data warehouses allow businesses to organize historical datasets for use in business intelligence (BI) and analytics, they quickly become more cost-intensive as datasets grow because of the combined use of compute and storage resources. Additionally, data warehouses can't handle the varied nature of data (structured, unstructured, and semi-structured) seen today.

In this chapter, we will explore the key features of Delta Lake, how it works, and why it is a good choice for big data processing. We will also provide examples of how to use Delta Lake with other big data tools, such as Spark and Trino later on in this report.

2.1 Definition

Delta Lake is an open-source storage layer built atop a data lake that confers reliability and ACID (Atomicity, Consistency, Isolation, and Durability) transactions. It enables a continuous and simplified data architecture for organizations. A data lake stores data in Parquet formats and enables a lakehouse data architecture, which helps organizations achieve a single, continuous data system that combines the best features of both the data warehouse and data lake while supporting streaming and batch processing.

2.2 How Delta Lake Works

A Delta Lake enables the building of a data lakehouse. Common lakehouses include the Databricks Lakehouse and Azure Databricks. This continuous data architecture allows organizations to harness

the benefits of data warehouses and data lakes with reduced management complexity and cost. Here are some ways Delta Lake improves the use of data warehouses and lakes:

- **Enables a lakehouse architecture:** Delta Lake enables a continuous and simplified data architecture that allows organizations to handle and process massive volumes of streaming and batch data without the management and operational hassles involved in managing streaming, data warehouses, and data lakes separately.
- **Enables intelligent data management for data lakes:** Delta Lake offers efficient and scalable metadata handling, which provides information about the massive data volumes in data lakes. With this information, data governance and management tasks proceed more efficiently.
- **Schema enforcement for improved data quality:** Because data lakes lack a defined schema, it becomes easy for bad/incompatible data to enter data systems. There is improved data quality thanks to automatic schema validation, which validates DataFrame and table compatibility before writes.
- **Enables ACID transactions:** Most organizational data architectures involve a lot of ETL and ELT movement in and out of data storage, which opens it up to more complexity and failure at node entry points. Delta Lake ensures the durability and persistence of data during ETL and other data operations. Delta lake captures all changes made to data during data operations in a transaction log, thereby ensuring data integrity and reliability during data operations.

2.3 Delta Lake Architecture Diagram

Delta Lake is an improvement from the lambda architecture whereby streaming and batch processing occur parallel, and results merge to provide a query response. However, this method means more complexity and difficulty maintaining and operating both the streaming and batch processes. Unlike the lambda architecture, Delta Lake is a continuous data architecture that combines streaming and batch workflows in a shared file store through a connected pipeline.

The stored data file has three layers, with the data getting more refined as it progresses downstream in the dataflow:

- **Bronze tables:** This table contains the raw data ingested from multiple sources like the Internet of Things (IoT) systems, CRM, RDBMS, and JSON files.
- **Silver tables:** This layer contains a more refined view of our data after undergoing transformation and feature engineering processes.
- **Gold tables:** This final layer is often made available for end users in BI reporting and analysis or use in machine learning processes.

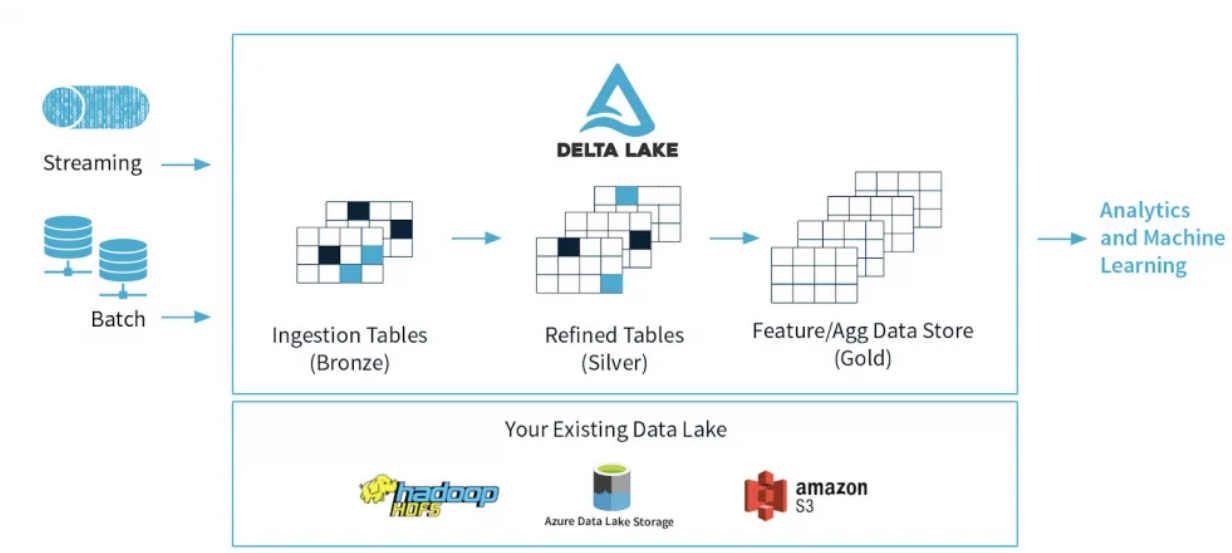

Figure 2.1: Delta Lake multi-hop architecture

2.4 Key benefits and features of Delta Lake

- **Audit trails and history:** In Delta Lake, every write exists as a transaction and is serially recorded in a transaction log. Therefore, any changes or commits made to the transaction log are recorded, leaving a complete trail for use in historical audits, versioning, or for time traveling purposes. This Delta Lake feature helps ensure data integrity and reliability for business data operations.

- **Time traveling and data versioning:** Because each write creates a new version and stores the older version in the transaction log, users can view/revert to older data versions by providing the timestamp or version number of an existing table or directory to the Sparks read API. Using the version number provided, the Delta Lake then constructs a full snapshot of the version with the information provided by the transaction log. Rollbacks and versioning play a vital role in machine learning experimentation, whereby data scientists iteratively change hyperparameters to train models and can revert to changes if needed.

- **Unifies batch and stream processing:** Every table in a Delta Lake is a batch and streaming sink. With Sparks structured streaming, organizations can efficiently stream and process streaming data. Additionally, with the efficient metadata handling, ease of scale, and ACID quality of each transaction, near-real-time analytics become possible without utilizing a more complicated two-tiered data architecture.

- **Efficient and scalable metadata handling:** Delta Lakes store metadata information in the transaction log and leverages Spark's distributed processing power to quickly process and efficiently read and handle large volumes of data metadata, thus improving data governance.

- **ACID transactions:** Delta Lakes ensure that users always see a consistent data view in a table or directory. It guarantees this by capturing every change made in a transaction log and isolating

it at the strongest isolation level, the serializable level. In the serializable level, every existing operation has and follows a serial sequence that, when executed one by one, provides the same result as seen in the table.

- **Data Manipulation Language operations:** Delta Lakes supports DML operations like updates, deletes, and merges, which play a role in complex data operations like change-data-capture (CDC), streaming upserts, and slowly-changing-dimension (SCD). Operations like CDC ensure data synchronization in all data systems and minimizes the time and resources spent on ELT operations. For instance, using the CDC, instead of ETL-ing all the available data, only the recently updated data since the last operation undergoes a transformation.
- **Schema enforcement:** Delta Lakes perform automatic schema validation by checking against a set of rules to determine the compatibility of a write from a DataFrame to a table. One such rule is the existence of all DataFrame columns in the target table. An occurrence of an extra or missing column in the DataFrame raises an exception error. Another rule is that the DataFrame and target table must contain the same column types, which otherwise will raise an exception. Delta Lake also use DDL (Data Definition Language) to add new columns explicitly. This data lake feature helps prevent the ingestion of incorrect data, thereby ensuring high data quality.
- **Compatibility with Spark's API:** Delta Lake is built on Apache Spark and is fully compatible with Spark API, which helps build efficient and reliable big data pipelines.
- **Flexibility and integration:** Delta lake is an open-source storage layer and utilizes the Parquet format to store data files, which promotes data sharing and makes it easier to integrate with other technologies and drive innovation.

2.5 Implementation

To use Delta Lake interactively within the Spark SQL, Scala, or Python shell, we need a local installation of Apache Spark. Depending on whether we want to use SQL, Python, or Scala, we can set up either the SQL, PySpark, or Spark shell, respectively.

Spark SQL Shell:

```
bin/spark-sql --packages io.delta:delta-core_2.12:2.3.0
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"
--conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

PySpark Shell:

1. Install the PySpark version that is compatible with the Delta Lake version by running the following:

```
pip install pyspark==<compatible-spark-version>
```

2. Run PySpark with the Delta Lake package and additional configurations:

```
pyspark --packages io.delta:delta-core_2.12:2.3.0
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"
--conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

Scala Shell:

Download the compatible version of Apache Spark by following instructions from Downloading Spark, either using pip or by downloading and extracting the archive and running spark-shell in the extracted directory.

```
bin/spark-shell --packages io.delta:delta-core_2.12:2.3.0
--conf "spark.sql.extensions=io.delta.sql.DeltaSparkSessionExtension"
--conf "spark.sql.catalog.spark_catalog=org.apache.spark.sql.delta.catalog.DeltaCatalog"
```

Create Table:

To create a Delta table, write a DataFrame out in the delta format. We can use existing Spark SQL code and change the format from parquet, csv, json, and so on, to delta.

1. **SQL:**

```
CREATE TABLE delta.`/tmp/delta-table` USING DELTA AS SELECT col1 as id FROM VALUES 0,1,2,3,4;
```

2. **Python:**

```
data = spark.range(0, 5)
data.write.format("delta").save("/tmp/delta-table")
```

3. **Scala:**

```
val data = spark.range(0, 5)
data.write.format("delta").save("/tmp/delta-table")
```

4. **Java:**

```
import org.apache.spark.sql.SparkSession;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

SparkSession spark = ...   // create SparkSession

Dataset<Row> data = spark.range(0, 5);
data.write().format("delta").save("/tmp/delta-table");
```

These operations create a new Delta table using the schema that was inferred from your DataFrame

Conclusion

Delta Lake is an important tool for big data processing, providing reliable data management and ensuring data integrity at scale. Its ACID transactions, schema enforcement, and data versioning

features make it a popular choice for companies that need to process large amounts of data with high accuracy and reliability.

By using Delta Lake, data engineers and data scientists can easily manage data quality, track data lineage, and collaborate on data analysis projects. With its seamless integration with other big data tools. Delta Lake provides a powerful solution for big data processing that can help companies gain insights from their data faster and more efficiently.

Chapter 3

Trino

Introduction

3.1 Définition

Trino is an open source distributed SQL query engine. It is a hard fork of the original Presto project created by Facebook. It lets developers run interactive analytics against large volumes of data. With Trino, organizations can easily use their existing SQL skills to query data without having to learn new complex languages. The architecture is quite similar to traditional online analytical processing (OLAP) systems using distributed computing architectures, in which one controller node coordinates multiple worker nodes.

3.2 How it works

Trino is a distributed system that runs on Hadoop, and uses an architecture similar to massively parallel processing (MPP) databases. It has one coordinator node working with multiple worker nodes. Users submit SQL to the coordinator which uses query and execution engine to parse, plan, and schedule a distributed query plan across the worker nodes. It supports standard ANSI SQL, including complex queries, joins aggregations, and outer joins.

Leveraging this architecture, the Trino query engine is able to process SQL queries on large amounts of data in parallel across a cluster of computers, or nodes. Trino runs as a single-server process on each node. Multiple nodes running Trino, which are configured to collaborate with each other, make up a Trino cluster.

The following figure displays a high-level overview of a Trino cluster composed of one coordinator and multiple worker nodes. A Trino user connects to the coordinator with a client, such as a tool using the JDBC driver or the Trino CLI. The coordinator then collaborates with the workers, which access the data sources.

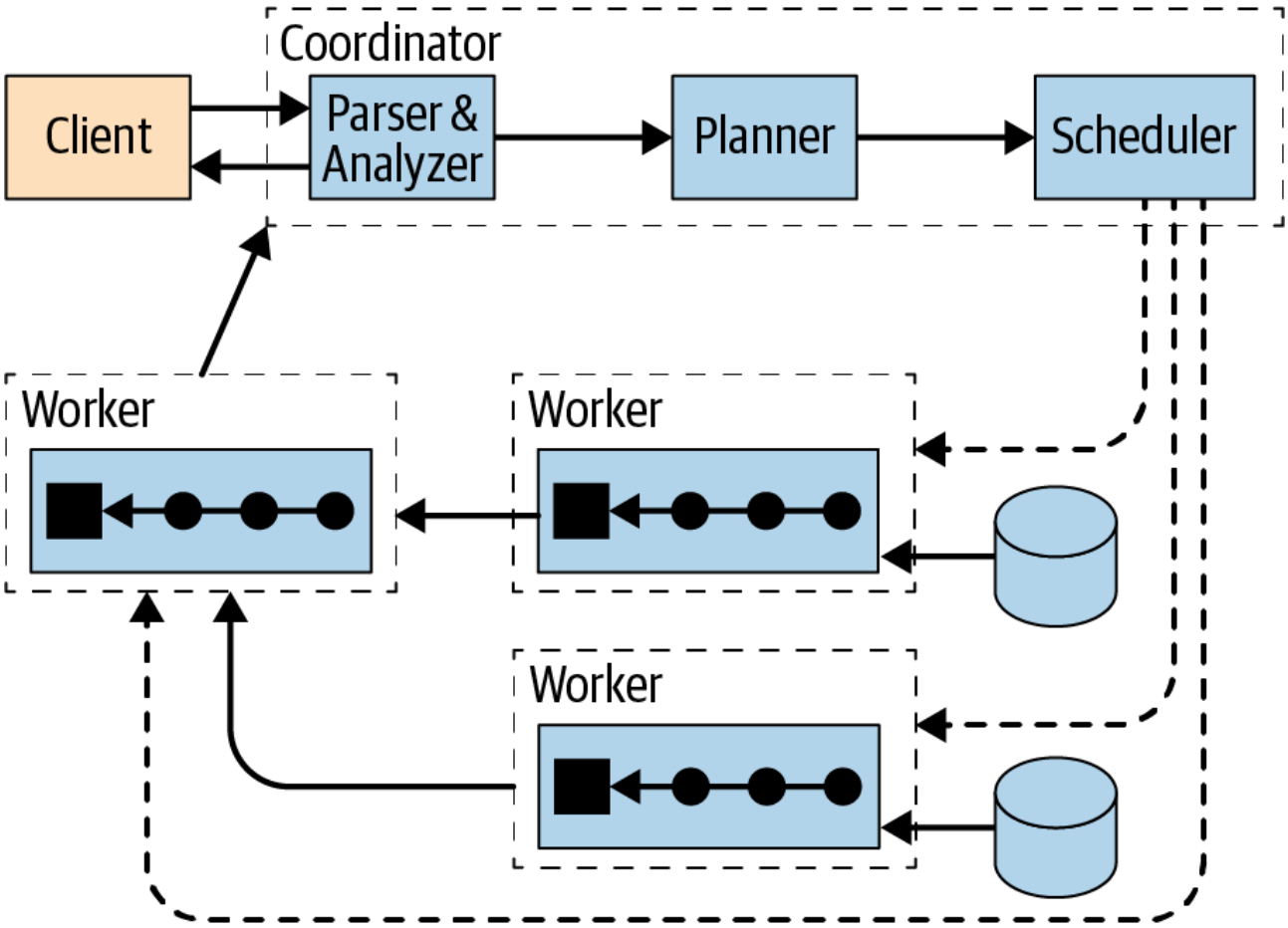

Figure 3.1: Trino architecture overview with coordinator and workers

1. A coordinator is a Trino server that handles incoming queries and manages the workers to execute the queries.
2. A worker is a Trino server responsible for executing tasks and processing data.
3. The discovery service typically runs on the coordinator and allows workers to register to participate in the cluster.
4. All communication and data transfer between clients, coordinator, and workers uses REST-based interactions over HTTP/HTTPS.

The following figure shows how the communication within the cluster happens between the coordinator and the workers, as well as from one worker to another. The coordinator talks to workers to assign work, update status, and fetch the top-level result set to return to the users. The workers talk to each other to fetch data from upstream tasks, running on other workers. And the workers retrieve result sets from the data source.

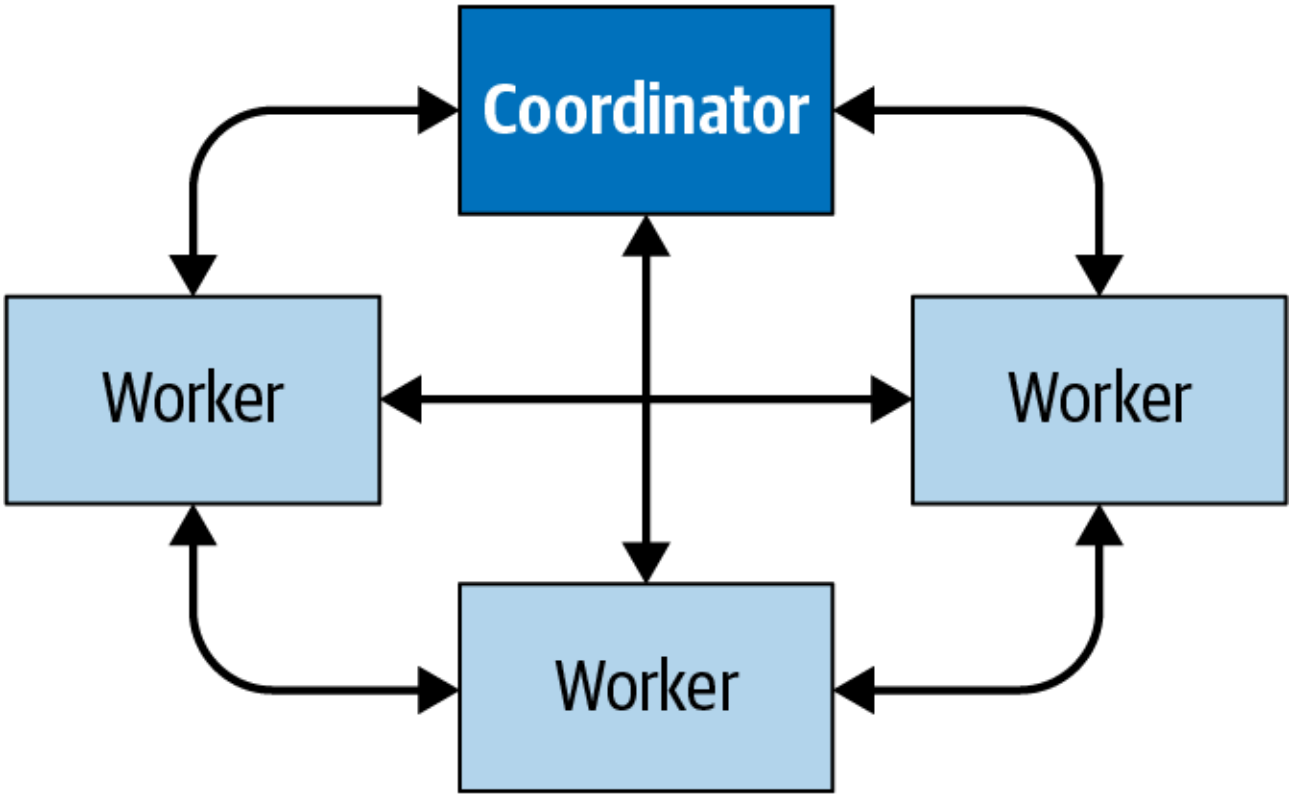

Figure 3.2: Communication between coordinator and workers in a Trino cluster

3.3 Coordinator

The Trino coordinator is the server responsible for receiving SQL statements from the users, parsing these statements, planning queries, and managing worker nodes. It's the brain of a Trino installation and the node to which a client connects. Users interact with the coordinator via the Trino CLI, applications using the JDBC or ODBC drivers, or any other available client libraries for a variety

of languages. The coordinator accepts SQL statements from the client such as SELECT queries for execution.

Every Trino installation must have a coordinator alongside one or more workers. For development or testing purposes, a single instance of Trino can be configured to perform both roles.

The coordinator keeps track of the activity on each worker and coordinates the execution of a query. The coordinator creates a logical model of a query involving a series of stages.

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

Coordinators communicate with workers and clients by using an HTTP-based protocol.

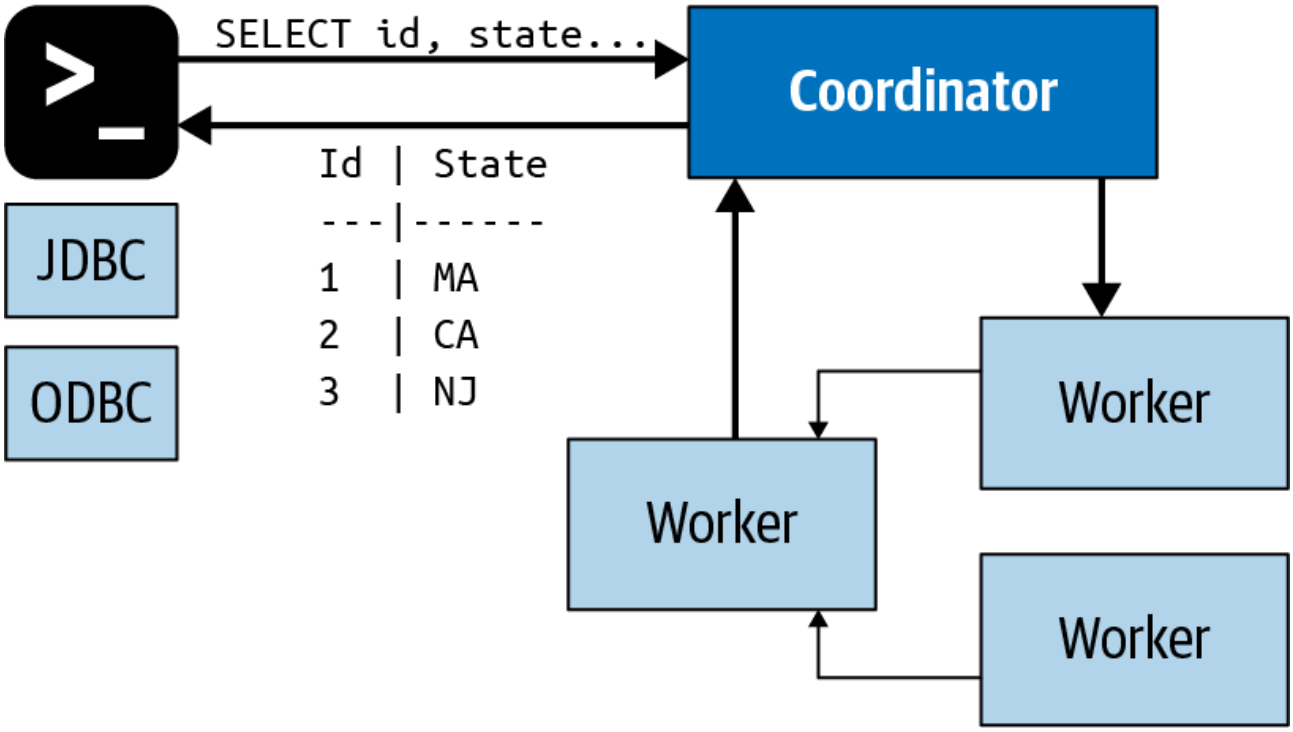

Figure 3.3: Client, coordinator, and worker communication processing a SQL statement

3.4 Workers

A Trino worker is a server in a Trino installation. It is responsible for executing tasks assigned by the coordinator and for processing data. Worker nodes fetch data from data sources by using connectors and then exchange intermediate data with each other. The final resulting data is passed on to the coordinator. The coordinator is responsible for gathering the results from the workers and providing

the final results to the client.

During installation, workers are configured to know the hostname or IP address of the discovery service for the cluster. When a worker starts up, it advertises itself to the discovery service, which makes it available to the coordinator for task execution.

Workers communicate with other workers and the coordinator by using an HTTP-based protocol.

The following figure shows how multiple workers retrieve data from the data sources and collaborate to process the data, until one worker can provide the data to the coordinator.

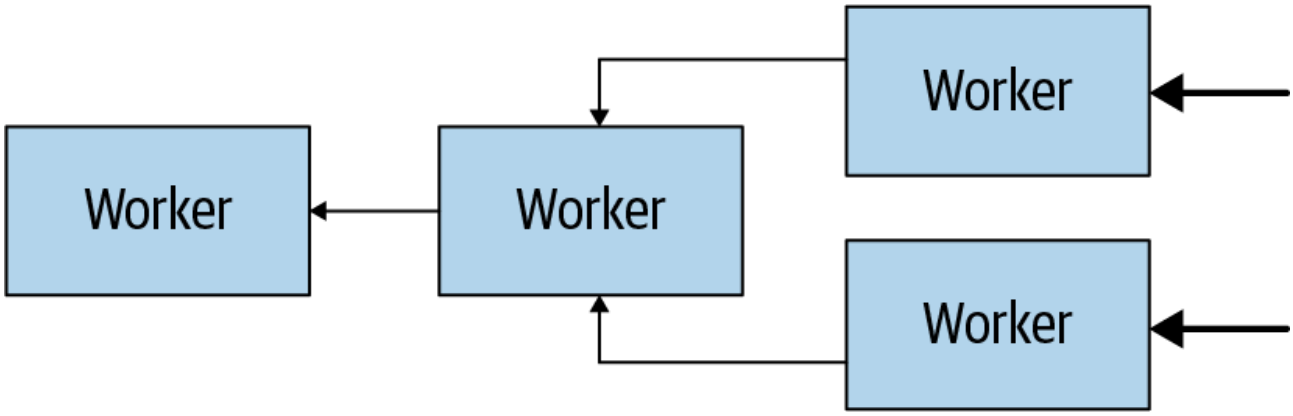

Figure 3.4: Workers in a cluster collaborate to process SQL statements and data

3.5 Connector-Based Architecture

At the heart of the separation of storage and compute in Trino is the connector-based architecture. A connector provides Trino an interface to access an arbitrary data source.

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

Trino provides a service provider interface (SPI), which is a type of API used to implement a connector. By implementing the SPI in a connector, Trino can use standard operations internally to connect to any data source and perform operations on any data source. The connector takes care of the details relevant to the specific data source.

At the heart of the separation of storage and compute in Trino is the connector-based architecture. A connector provides Trino an interface to access an arbitrary data source.

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

Trino provides a service provider interface (SPI), which is a type of API used to implement a connector. By implementing the SPI in a connector, Trino can use standard operations internally to connect to any data source and perform operations on any data source. The connector takes care of the details relevant to the specific data source.

- Operations to fetch table/view/schema metadata

- Operations to produce logical units of data partitioning, so that Trino can parallelize reads and writes
- Data sources and sinks that convert the source data to/from the in-memory format expected by the query engine

Trino provides many connectors to systems, here are the list of connects as of the writing of this report:

Connector Name	Documentation Link
Accumulo	https://trino.io/docs/current/connector/accumulo.html
Atop	https://trino.io/docs/current/connector/atop.html
BigQuery	https://trino.io/docs/current/connector/bigquery.html
Black Hole	https://trino.io/docs/current/connector/blackhole.html
Cassandra	https://trino.io/docs/current/connector/cassandra.html
ClickHouse	https://trino.io/docs/current/connector/clickhouse.html
Delta Lake	https://trino.io/docs/current/connector/delta-lake.html
Druid	https://trino.io/docs/current/connector/druid.html
Elasticsearch	https://trino.io/docs/current/connector/elasticsearch.html
Google Sheets	https://trino.io/docs/current/connector/googlesheets.html
Hive	https://trino.io/docs/current/connector/hive.html
Hudi	https://trino.io/docs/current/connector/hudi.html
Iceberg	https://trino.io/docs/current/connector/iceberg.html
Ignite	https://trino.io/docs/current/connector/ignite.html
JMX	https://trino.io/docs/current/connector/jmx.html
Kafka	https://trino.io/docs/current/connector/kafka.html
Kinesis	https://trino.io/docs/current/connector/kinesis.html
Kudu	https://trino.io/docs/current/connector/kudu.html
Local File	https://trino.io/docs/current/connector/localfile.html
MariaDB	https://trino.io/docs/current/connector/mariadb.html
Memory	https://trino.io/docs/current/connector/memory.html
MongoDB	https://trino.io/docs/current/connector/mongodb.html
MySQL	https://trino.io/docs/current/connector/mysql.html
Oracle	https://trino.io/docs/current/connector/oracle.html
Phoenix	https://trino.io/docs/current/connector/phoenix.html
Pinot	https://trino.io/docs/current/connector/pinot.html
PostgreSQL	https://trino.io/docs/current/connector/postgresql.html
Prometheus	https://trino.io/docs/current/connector/prometheus.html
Redis	https://trino.io/docs/current/connector/redis.html
Redshift	https://trino.io/docs/current/connector/redshift.html
SingleStore	https://trino.io/docs/current/connector/singlestore.html
SQL Server	https://trino.io/docs/current/connector/sqlserver.html
System	https://trino.io/docs/current/connector/system.html
Thrift	https://trino.io/docs/current/connector/thrift.html
TPCDS	https://trino.io/docs/current/connector/tpcds.html
TPCH	https://trino.io/docs/current/connector/tpch.html

Table 3.1: List of Trino Connectors and their documentation

Trino's SPI also gives you the ability to create your own custom connectors. This may be needed if you need to access a data source without a compatible connector. If you end up creating a connector, we strongly encourage you to learn more about the Trino open source community, use our help, and

contribute your connector. Check out "Trino Resources" for more information. A custom connector may also be needed if you have a unique or proprietary data source within your organization. This is what allows Trino users to query any data source by using SQL—truly SQL-on-Anything.

The following figure shows how the Trino SPI includes separate interfaces for metadata, data statistics, and data location used by the coordinator, and for data streaming used by the workers

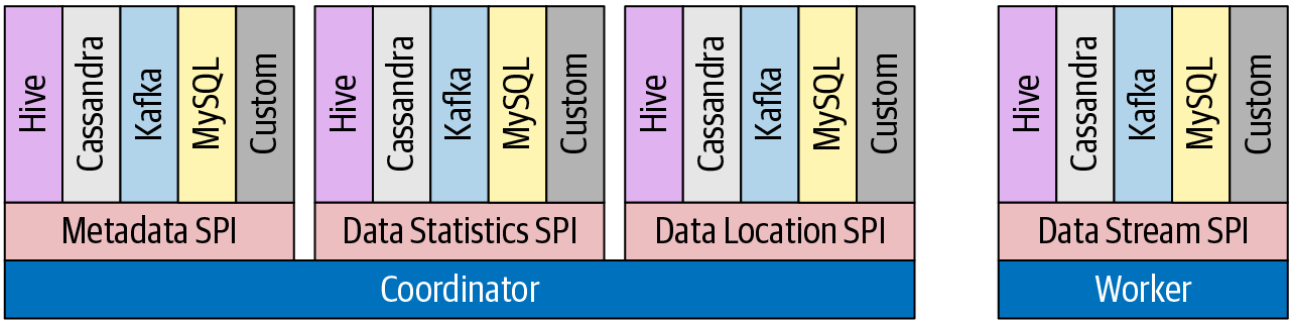

Figure 3.5: Overview of the Trino service provider interface (SPI)

Trino connectors are plug-ins loaded by each server at startup. They are configured by specific parameters in the catalog properties files and loaded from the plug-ins directory.

3.6 Catalogs, Schemas, and Tables

The Trino cluster processes all queries by using the connector-based architecture described earlier. Each catalog configuration uses a connector to access a specific data source. The data source exposes one or more schemas in the catalog. Each schema contains tables that provide the data in table rows with columns using different data types. You can find out more about catalogs, schemas, tables. Specifically in "Catalogs", "Schemas", and "Tables".

Conclusion

The Trino architecture has a coordinator receiving user requests and then using workers to assemble all the data from the data sources. Each query is translated into a distributed query plan of tasks in numerous stages. The data is returned by the connectors in splits and processed in multiple stages until the final result is available and provided to the user by the coordinator.

General Conclusion