SQL#TRINO

TBDM University Project.

Michele Benedetti
117318

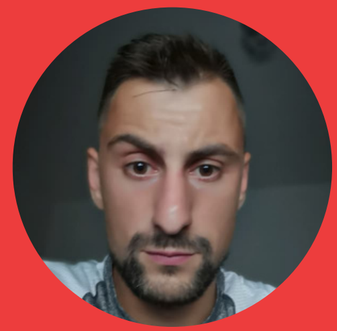
Flavio Pocari
123759

Armando Xheka
123579

Group Members

Michele
Benedetti

Flavio
Pocari



Armando
Xheka

Introduction

Part 01

Overview of the project

The **main** goal of **SQL#TRINO** project is to study the potentiality of using Kafka Broker and TrinoDB to analyze real-time data streaming.

The **objective** of the project is to define a prototype tool to analyze data in real-time by using Trino.

The implemented system comprises a docker-compose file configuring all the necessary images, such as **Zookeeper, Kafka, MongoDB, Mongo-Express, Kafdrop,** and **TrinoDB**.

It also includes a **Producer** script written in javascript and a **Jupyter** file with some examples of data analytics.

Scopes

Part 02

Known Problems

Scalability

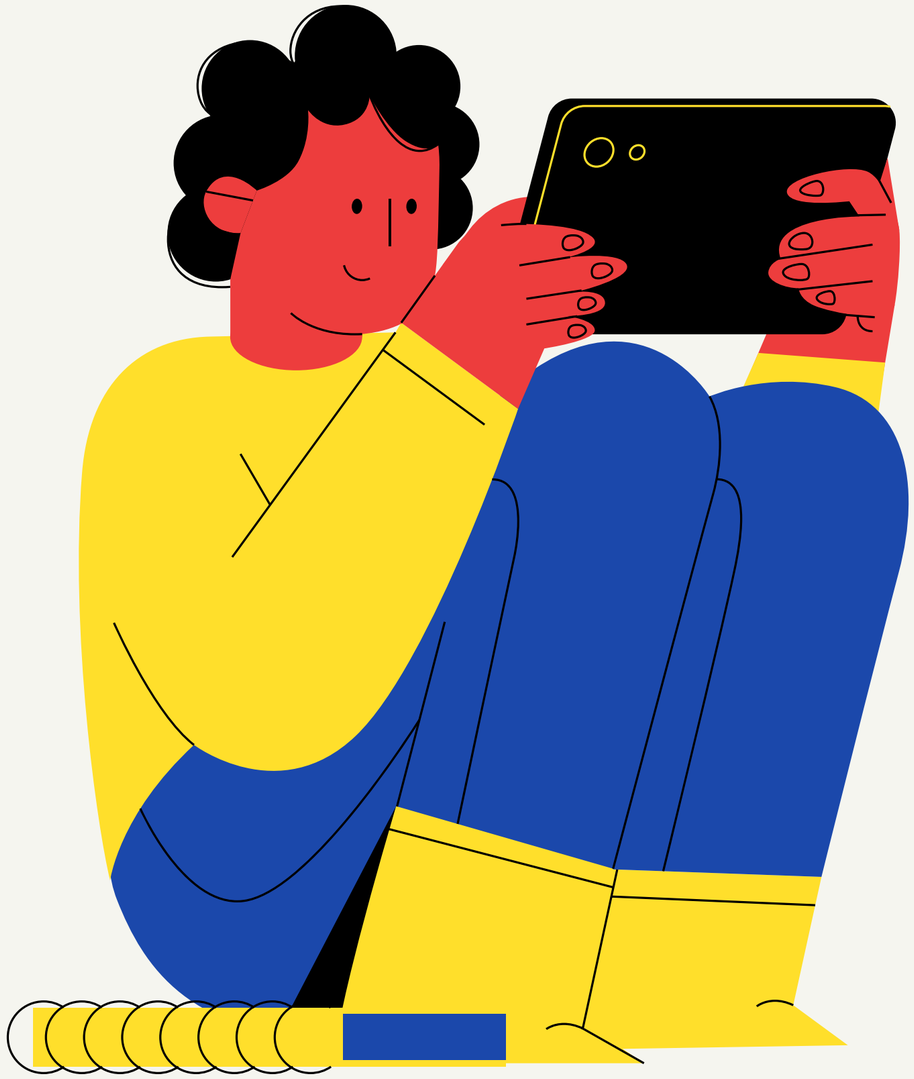
Common issue in big data systems is that they need to handle large and continuative amounts of data without compromising performance or reliability.

Mapping Schema

Most systems are not capable of mapping the structure from NoSQL databases and converting it into SQL tables to allow queries to be made.

Performance

With data growing in terms of volume and velocity, also difficulty of processing and analysing within a reasonable amount of time and resources became.



Technologies

Part 03

Technologies Overview

docker

MongoDB®

kafka

jupyter

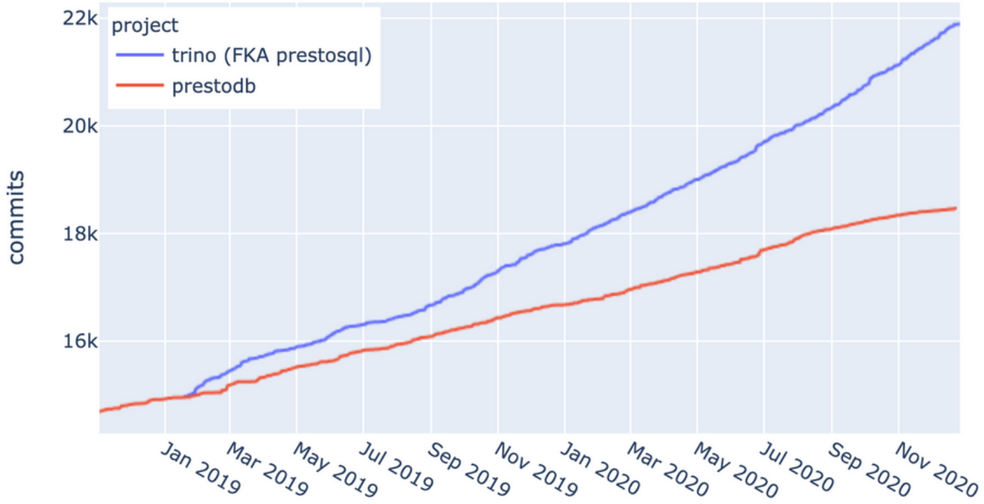
express

Git API

Trino! A fast query enigine.

The community and project velocity are growing and outpacing Presto by a lot despite the rebrand.

- It's a query engine, not a database (it doesn't store data).
- Born as a fork of PrestoDB once the founders quit Facebook (2019) with the initial name PrestoSQL.
- 2020 - Linux Foundation enforces Presto trademark and PrestoSQL was forced to rename to Trino (short for fast space particles called neutrinos).



Why Trino instead of Presto?

prestoDB

PERFORMANCE

COMMUNITY SUPPORT

BETTER INTEGRATION

1 Answer

Sorted by: Highest score (default)

1

I would recommend upgrading to Trino (formerly PrestoSQL) because the MongoDB connector (version >= 360) supports mapping fields to JSON type. This type mapping is unavailable in prestodb.

https://trino.io/download.html

Share Improve this answer Follow

answered Feb 15, 2022 at 7:50

ebyhr

1,497 • 9 • 15

Michele Benedetti 20 days ago

The _schema collection is automatically created also if a document is created outside of the trino-cli or mongosh? In my case i have a connector sync from Apache Kafka that constantly write data inside the DB

ebyhr 20 days ago

Even if the data is generated without Trino, the connector creates & updates the collection during the 1st read. (edited)

Trino Keywords

- **CLUSTER:** A Trino cluster has a coordinator and multiple workers who collaborate to access data sources configured in catalogs. Users connect to the coordinator with SQL queries. Query processing is stateful and parallelized across all workers using threads. The workload is orchestrated by the coordinator and spread parallel across all workers in the cluster. Each node runs Trino in one JVM instance, and processing is parallelized further using threads.

- **COORDINATOR:** It is the “brain” of a Trino installation and is also the node to which a client connects to submit statements for execution. The coordinator creates a logical model of a query involving a series of stages, which is then translated into a series of connected tasks running on a cluster of Trino workers.

- **WORKER:** When a Trino worker process starts up, it advertises itself to the discovery server in the coordinator, which makes it available to the Trino coordinator for task execution.

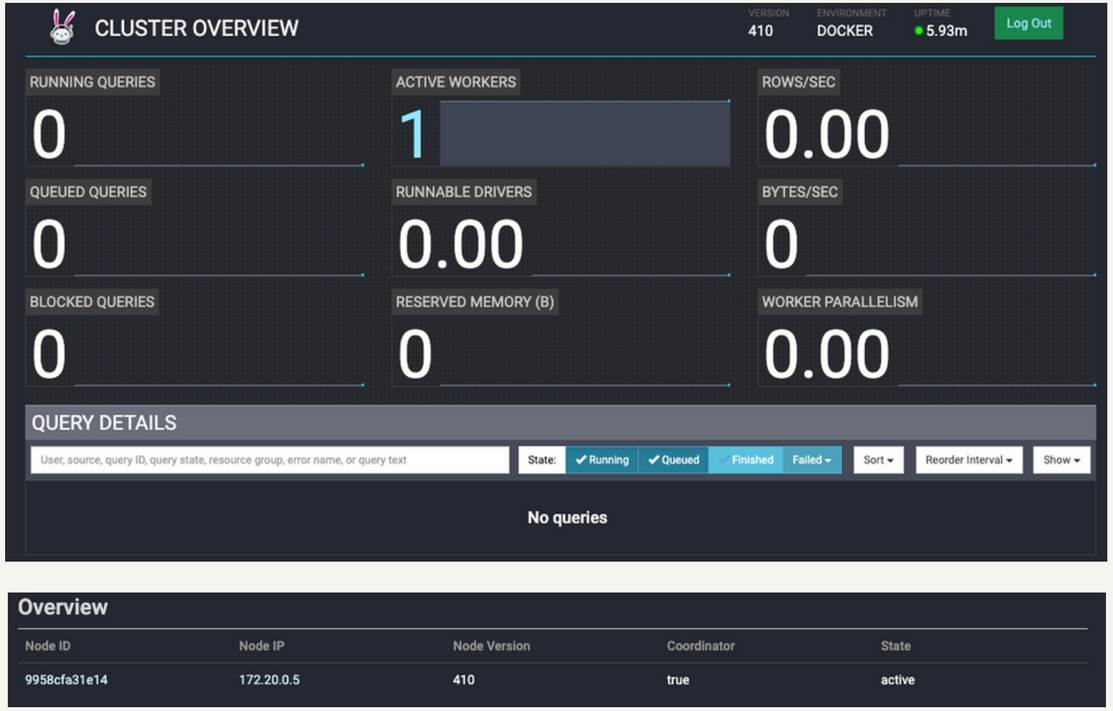
- **CONNECTOR:** Trino contains several built-in connectors such as a connector for JMX, a System connector which provides access to built-in system tables and Hive connector. It also allows configuring more catalogs in a single Trino cluster that uses different connectors, allowing one to query data from different clusters (e.g Mongo, Hive), even within the same SQL query.

How to configure Trino?

This used docker image provides an out-of-the-box **single node cluster** with the pre-installed catalogs.

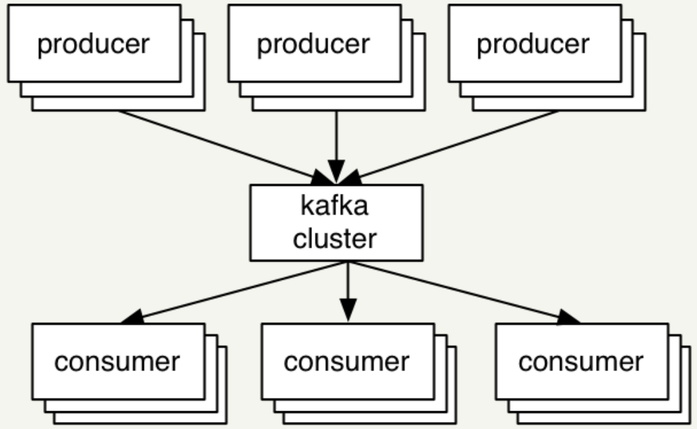
In order to configure the MongoDB connector inside Trino, we need to add a custom *mongodb.properties* file with the connector name and the connection-url of the MongoDB cluster.

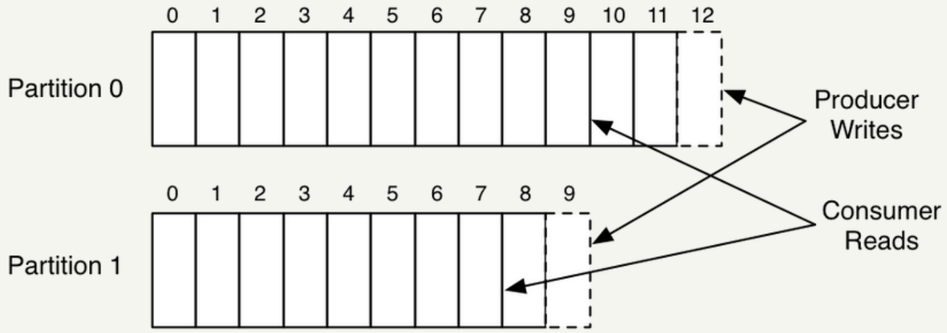
```
1   connector.name=mongodb
2   mongodb.connection-url=mongodb://root:example@mongo:27017
```



Once started, Trino adds a ***catalog*** named with the same name as the connector previously defined, it contains all the information about schemas and references of data sources. Thanks to the ***schema,*** Trino is able to organize data like documents into tables and, together with catalog, define a set of tables that can be queried.

Kafka & Kafdrop





In the implemented system, Kafka Broker and Zookeeper were used to realise a complex broker that collects and forwards messages through topics.

The major difference between a normal MQTT broker and Kafka is the capability of storing messages in order to make them available furthermore over time.

Our topics are created with a **partitions factor** of **6** and a **replication factor** of **1** (since we only have one cluster).

To keep track of what was happening inside our broker and to check all the configurations, Kafkdrop was used in order to have a more readable and user friendly interface.

All in one!

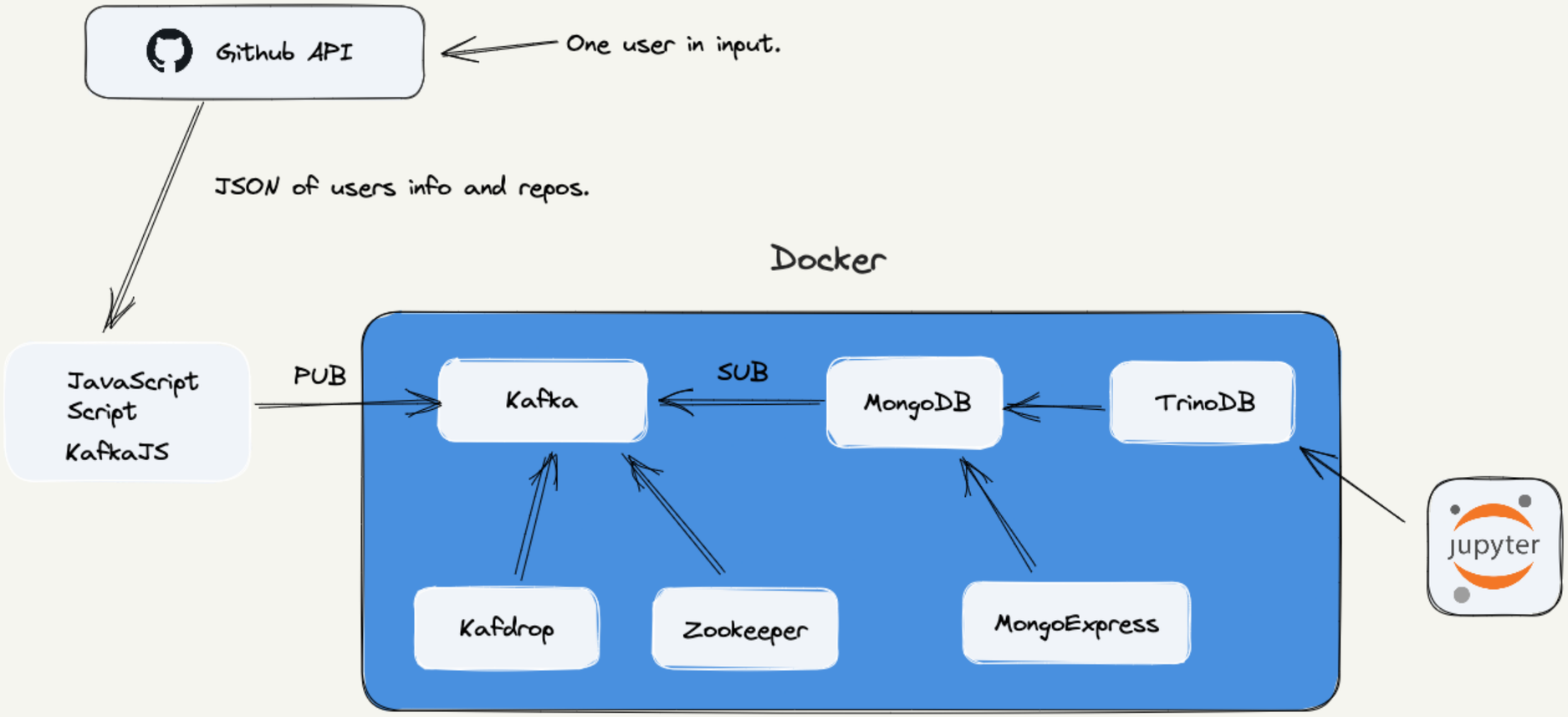
We used Docker for its portability, so the application runs better on the same machine like Linux and MacOS.

With Docker, we can run multiple containers and we can guarantee:

- **Resource** utilization: containers consume fewer resources.
- **Scalability**: easy to scale by adding or removing containers.
- **Portability**: containers can be easily moved between different environments.

Docker allows us to access the host's environment variables, so there is no need to enter the IP. We wrote a line in bash that allows us to take the IP of the host and make the docker file portable (except for wired connection).

```yaml
version: '3.8'
services:
  zookeeper:
    ...

  kafka:
    image: wurstmeister/kafka:latest
    ports:
      - "9092:9092"
    links:
      - zookeeper
    environment:
      KAFKA_ADVERTISED_HOST_NAME: $HOST_IP
      ...

  kafkadrop:
    image: obsidiandynamics/kafdrop
    ports:
      - "9000:9000"
    environment:
      ...
    depends_on:
      - kafka
      - zookeeper

  mongo:
    image: mongo
    restart: always
    ports:
      - 27017:27017
    environment:
      ...

  mongo-express:
    image: mongo-express
    restart: always
    ports:
      - 8081:8081
    environment:
      ...

  trinodb:
    image: trinodb/trino
    ports:
      - 8080:8080
    volumes:
      - ./etc:/etc/trino/catalog
```



System Architecture

—

The implemented system is composed of a Docker container, a Javascript Producer and a Jupyter Notebook to visualize GitHub data.

Implementation

Part 04

How work?

01 Clone Repository from Github

02 Docker Compose (run.sh)

03 Run KafkaJS client

04 Data Collection

05 Data Visualisation

Data Collection

Step 01

An array of users is passed, and then followers and following are taken and put into a *Set* in order to avoid duplicates.

Step 02

A request is made to the Github API for each user concerning the respective public repositories, and the main and most relevant information are taken for data analysis.

Step 03

The users or repositories are JSON, which are stored on MongoDB in the respective collections.

Data Transfer and Visualisation

Step 01

Through KafkaJS, the fetched information are sent to a Kafka Broker that is listening on a specific topic.

Step 02

Once a Topic receives message/s, it is sent into the desired Database by following the previously defined connector configuration file.

Step 03

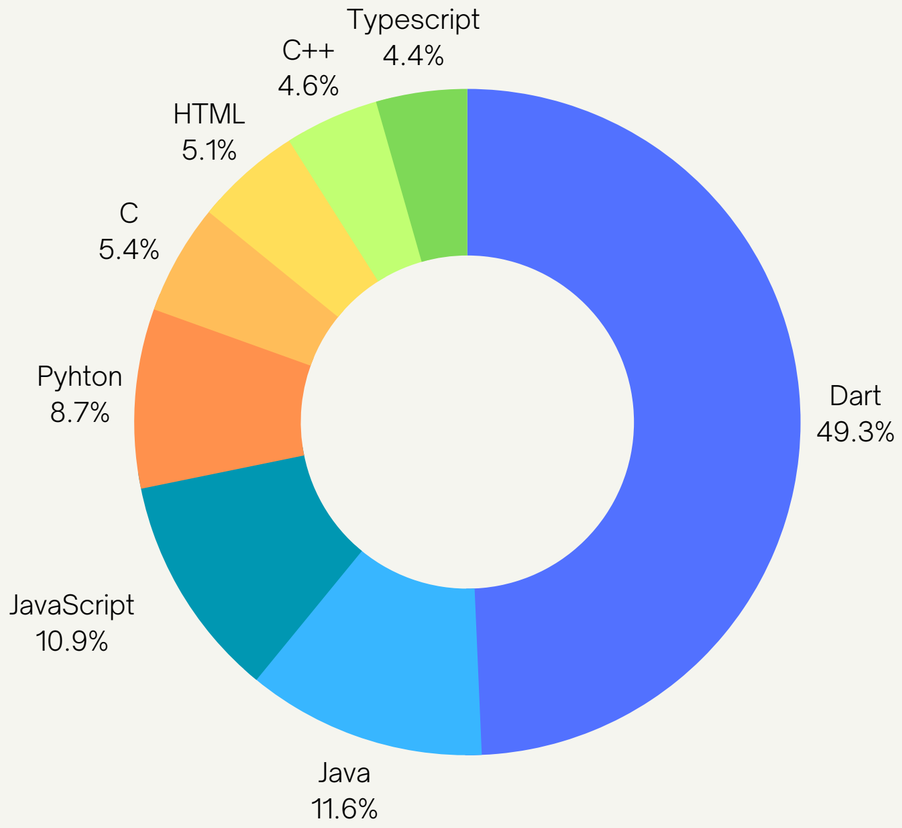
Now data are ready to be analysed via queries applicable through TrinoDB.

Step 04

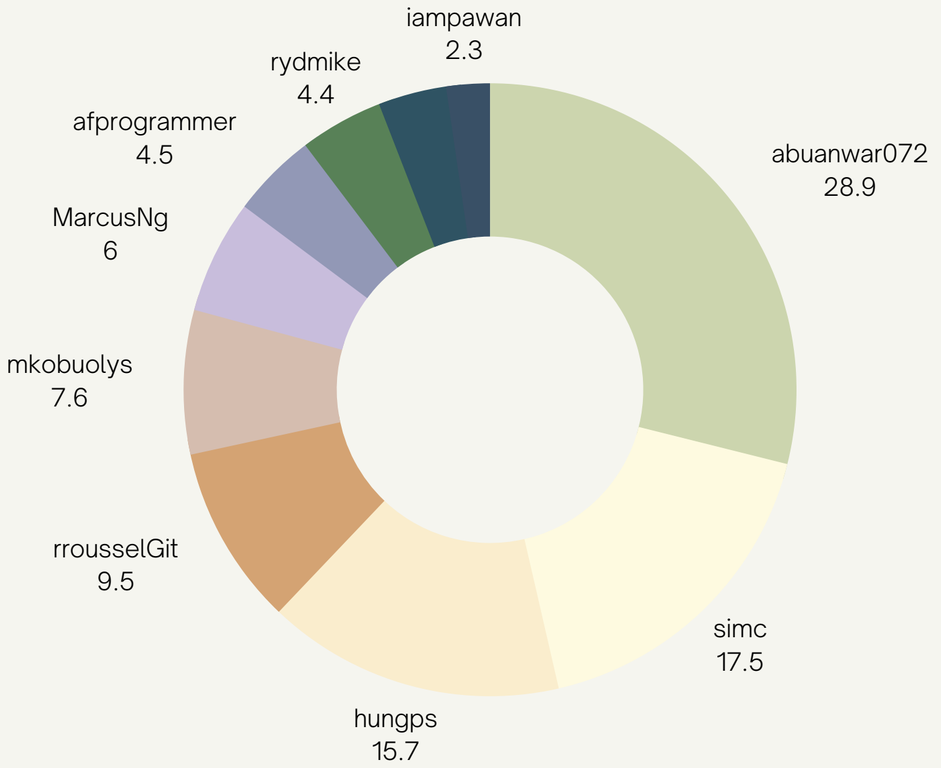
Jupyter Notebook is used to apply and gain the results of the queries into a visual graphic of the analysed data.

Results

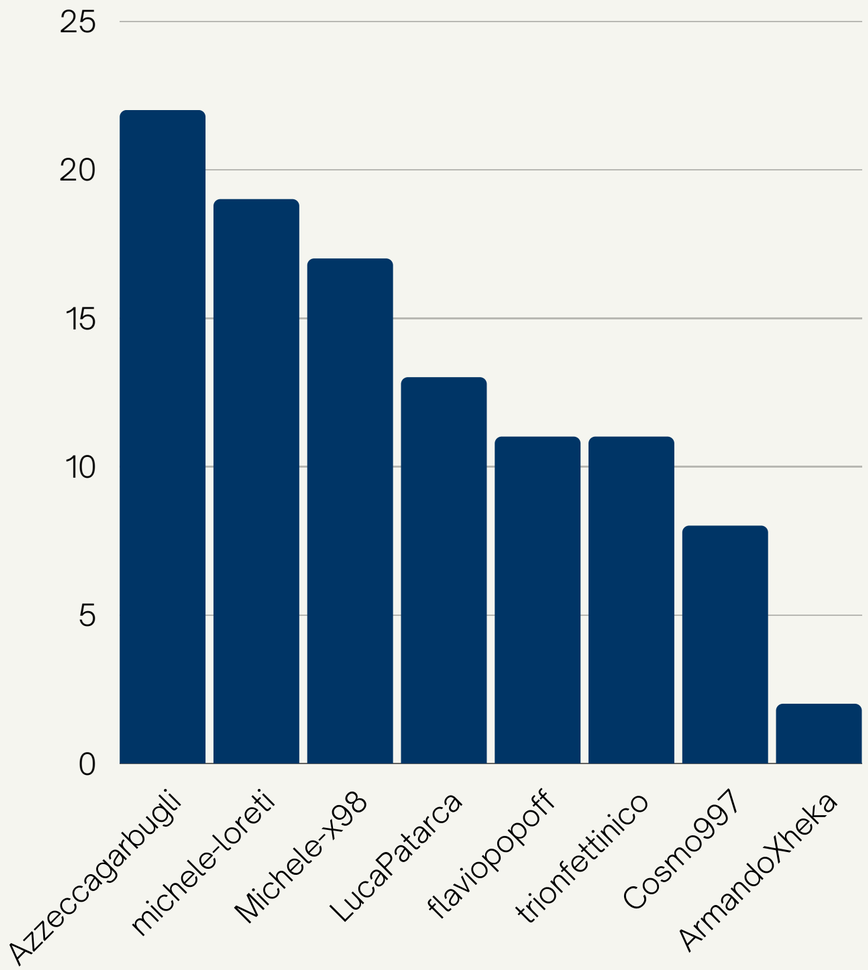
Part 05



Donut Chart of Most Used Language

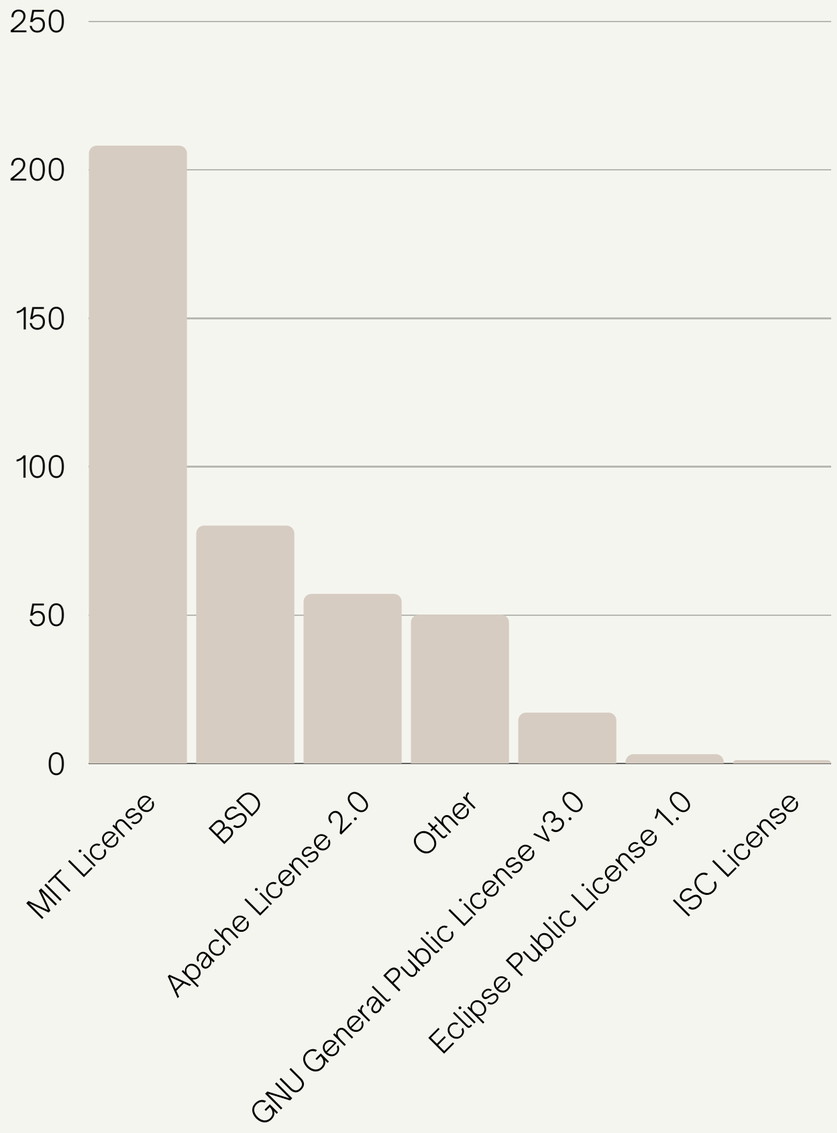


Donut Chart of Average Stars by User



Bar Chart of Repo Number by User.

Most license used for each repos.



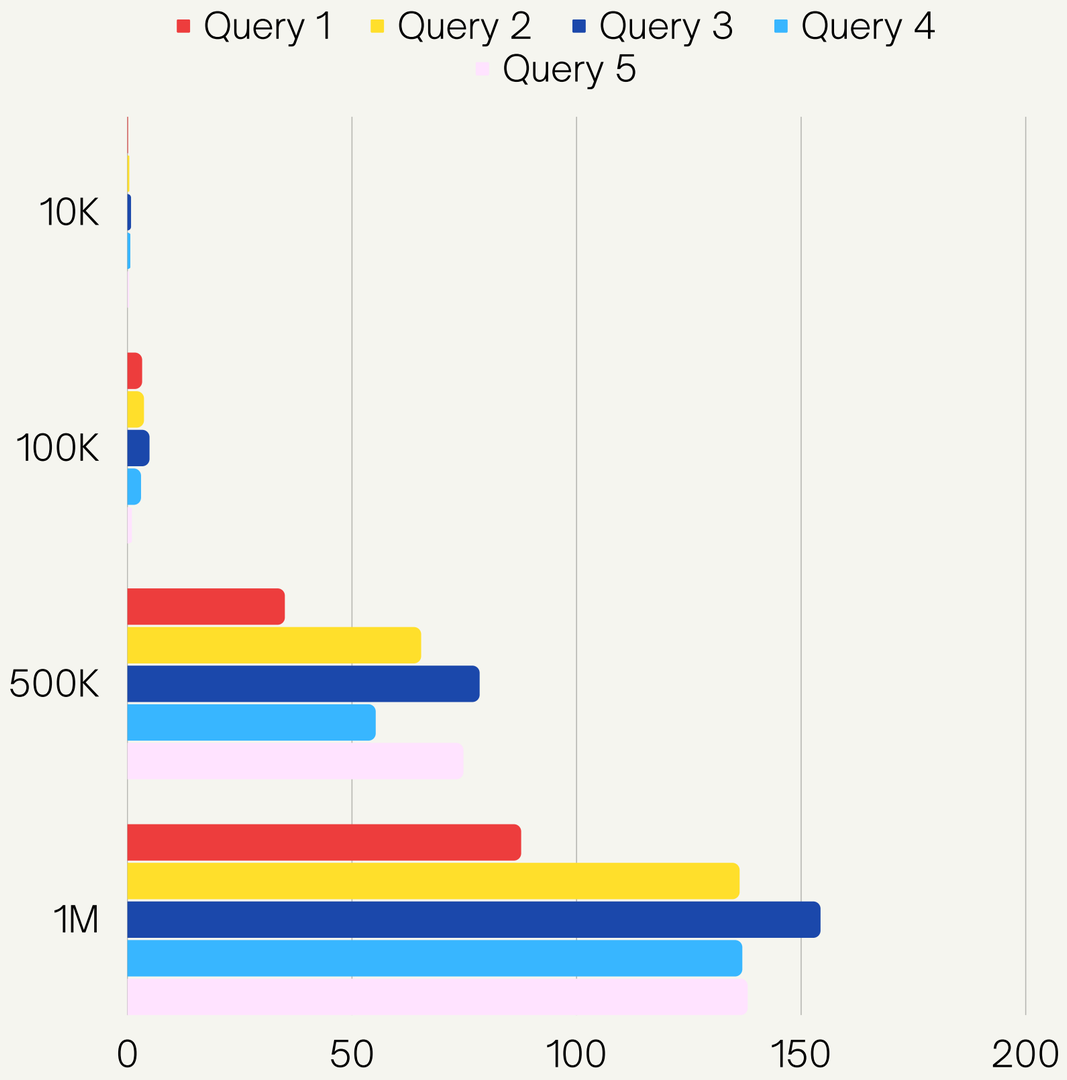
Query Performance

We performed queries on datasets containing the same data but with different sizes:

- ***10,000 elements.***
- ***100,000 elements.***
- ***500,000 elements.***
- ***1,000,000 elements.***

What surface from our analysis underlines the time needed to execute queries varied significantly depending on the amount of data involved, ranging from a time interval in the range of thousandths of a second to a time of just over two minutes in the case of a data sample consisting of one million elements.

Those performances are measured with only one Trino cluster composed of one worker that is a coordinator too.



Future Improvements

Part 06

Flexibility

Allow more flexibility over the Git API calls by asking which kinds of data the user wants to fetch and from which GitHub user.

Configuration

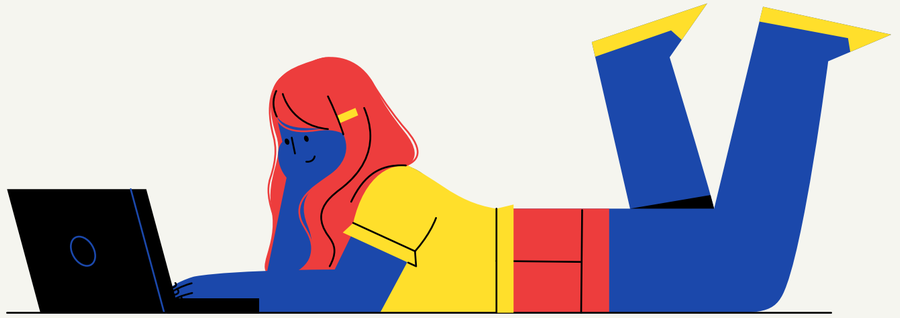
Allow selecting the number of replication for each topic by providing more than one broker node and test query performances with more workers nodes.

Multiple Database

Add more Database source built with differente technology and test the performance when Trino need to perform a query from different data source.

Conclusion

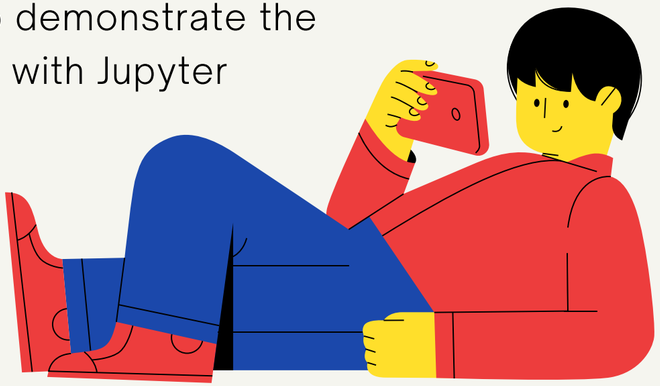
Part 07



In conclusion, the use of Kafka, Trino and Mongo enabled us to demonstrates the potential of this tools, and how efficiently and effectively manage large volumes of data.

The Kafka messaging system provides a reliable and scalable platform, the Trino query engine allows for flexible data analysis across a variety of data sources and Mongo allows for high-performance data storage and retrieval. These technologies form a robust and comprehensive solution for big data analysis.

Users can easily set up an environment for testing the connection between Kafka, Mongo and Trino, although, a simple producer is included in the project in order to demonstrate the complete flow of the application from data production to analysis with Jupyter



Contact

michele.benedetti@studenti.unicam.it
flavio.pocari@studenti.unicam.it
armando.xheka@studenti.unicam.it

Thanks for your attention.