

Infrastructures for Data Science Final Project

J. Galvis-Velandia - D. Yáñez-Oyarce

Description

The main objective of this project is the design and implementation of a data infrastructure architecture capable of integrating, processing, and jointly analyzing both static and temporal data extracted from Twitter.

To achieve this, the project leverages transmission and storage technologies such as **MQTT**, **Kafka**, **Druid**, **Hive**, and **Superset**, with the aim of enabling a data visualization environment that supports data-driven decision-making.

The development of this project is focused on exploring and applying various data infrastructure tools within a Dockerized environment.

It is recommended to read the assignment from this GitHub repository:[Press here](#)

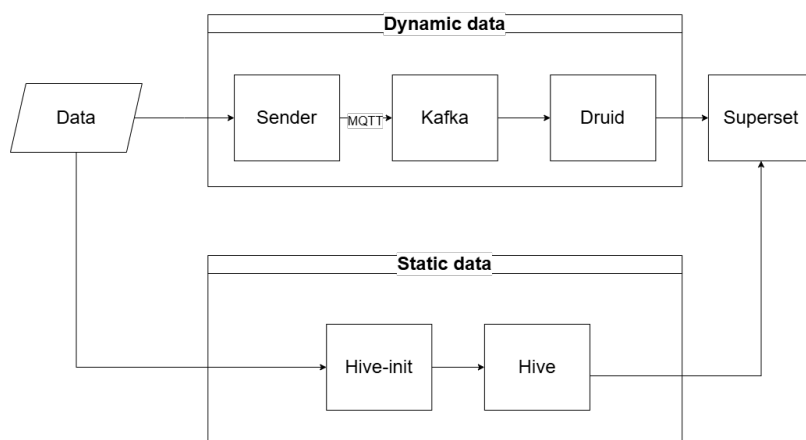


Figure 1: Service orchestration architecture for data transmission.

1 Pipeline - Temporal Data (tweets.json)

The objective of this part of the project is to build a data ingestion pipeline into data visualization tools such as *Superset*, simulating a real-time environment for temporal data from the file *tweets.json*. This file represents a dynamic data source to be processed as a stream, and contains multiple tweets posted by users at different intervals.

The pipeline enables users to make data-driven decisions by visualizing the information interactively through Business Intelligence tools.

1.1 Workflow

To achieve the objective, a distributed architecture has been designed to simulate a production-like environment using Docker containers. The data flow follows the steps described below.

1.1.1 MQTT

MQTT is a lightweight messaging protocol used to transmit data between a publisher and a subscriber.

A Python script (*sender.py*) is responsible for reading tweets from the `tweets.json` file and publishing them one by one to an *MQTT* topic named *tweets*.

1.1.2 MQTT Broker: Mosquitto

It acts as an intermediary between the publishers and subscribers of the messaging topic, receiving the messages published to the *tweets* topic and routing them to the respective subscribers.

1.1.3 Kafka Connect

Enables the integration of external systems with *Kafka*. It provides two types of connectors: one that allows data to be ingested into *Kafka* from external systems (in this case, *MQTT*), and another that allows data to be exported from *Kafka* to external systems (*Druid*).

1.1.4 Apache Kafka

A distributed real-time messaging platform designed to handle large volumes of data in a persistent and scalable way. It receives data from the *MQTT* messaging protocol, specifically from the topic named *tweets*.

1.1.5 Apache Druid

Druid is a real-time analytics database designed for fast querying over large volumes of data (especially temporal data). It is responsible for consuming messages from the *tweets* topic and structuring them for user-friendly analytical exploration.

To achieve this, **Druid** relies on the following components:

Component	Function
Broker	Receives queries from <i>Superset</i> and coordinates their execution.
Historical	Stores and queries historical data.
MiddleManager	Ingests data from sources like <i>Kafka</i> in real time.
Coordinator	Manages the distribution and availability of data across the cluster.
Router	Routes requests to the appropriate component.

Table 1: Druid components

1.1.6 Superset

It is an open-source Business Intelligence tool that allows users to visualize, explore, and analyze data interactively through a web interface. It supports connections to multiple databases, enables direct execution of SQL queries, and provides interactive charting capabilities. Thanks to its integration with tools like *Druid*, it is ideal for exploring large volumes of data.

In this project, it serves as the component that allows real-time visualization of tweets and enables analytical exploration of the data processed through the pipeline.

1.2 Procedure

In order to replicate the pipeline for processing and managing temporal data, follow these steps:

1. Open a terminal in the working directory and execute the following command:

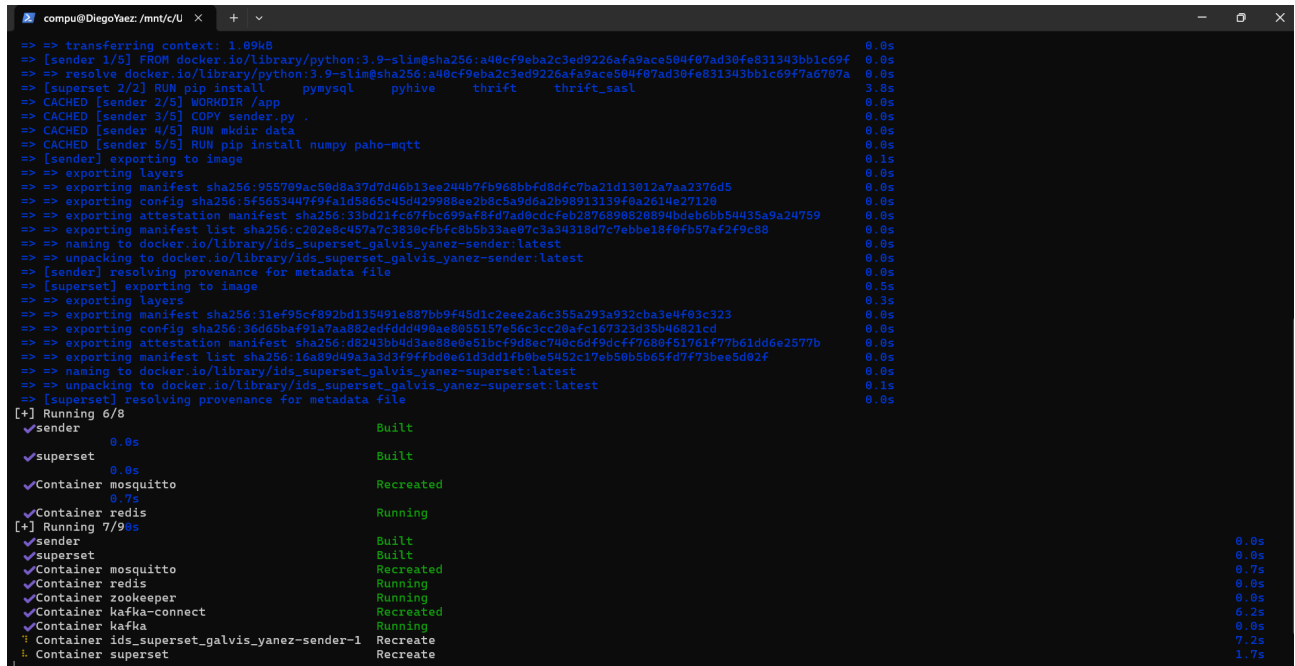
```
./start_pipeline.sh
```

The .sh script will start all the required services for the project using:

```
docker compose up --build -d
```

The following services will be launched:

- Zookeeper
- Kafka
- Kafka-connect
- Mosquitto
- Sender
- Redis
- Superset



```
compu@DiegoYanez: /mnt/c/U X + v
=> transferring context: 1.09kB
[sender 1/5] FROM docker.io/library/python:3.9-slim@sha256:a40cf9eba2c3ed9226afa9ace504f07ad30fe831343bb1c69f 0.0s
=> resolve docker.io/library/python:3.9-slim@sha256:a40cf9eba2c3ed9226afa9ace504f07ad30fe831343bb1c69f7a6707a 0.0s
[superset 2/2] RUN pip install pymysql pyhive thrift thrift_sasl 3.8s
=> CACHED [sender 2/5] WORKDIR /app 0.0s
=> CACHED [sender 3/5] COPY sender.py . 0.0s
=> CACHED [sender 4/5] RUN mkdir data 0.0s
=> CACHED [sender 5/5] RUN pip install numpy paho-mqtt 0.0s
[sender] exporting to image 0.1s
=> exporting layers 0.0s
=> exporting manifest sha256:955709ac50d8a37d7d46b13ee244b7fb968bbfd8dfc7ba21d13012a7aa2376d5 0.0s
=> exporting config sha256:5f5653447f9fald5865c45d429988ee2b8c5a9d6a2b90913139f0a2614e27120 0.0s
=> exporting attestation manifest sha256:33bd21fc67fbc699af8fd7ad0cdcfeb2876890820894bdeb6bb54435a9a24759 0.0s
=> exporting manifest list sha256:c202e8c457a7c3830cfbfc8b5b33ae07c3a34318d7c7ebba18f0fb57af2f9c88 0.0s
=> naming to docker.io/library/ids_superset_galvis_yanez-sender:latest 0.0s
=> unpacking to docker.io/library/ids_superset_galvis_yanez-sender:latest 0.0s
[sender] resolving provenance for metadata file 0.0s
[superset] exporting to image 0.5s
=> exporting layers 0.3s
=> exporting manifest sha256:31ef95cf892bd135491e887bb9f45d1c2ee2a6c355a293a932cba3e4f03c323 0.0s
=> exporting config sha256:36d65baf91a7aa882edfdd490ae8055157e56c3cc2aafc167323d35b46821cd 0.0s
=> exporting attestation manifest sha256:d8243bb4d3ae80e51bfc9d8ec748c6df9dcff7680f51761f77b61dd6e2577b 0.0s
=> exporting manifest list sha256:16a89d49a3a3d3f9ffbd0e61d3dd1fb0be5452c17eb50b5b65fd7f73bee5d02f 0.0s
=> naming to docker.io/library/ids_superset_galvis_yanez-superset:latest 0.0s
=> unpacking to docker.io/library/ids_superset_galvis_yanez-superset:latest 0.1s
[superset] resolving provenance for metadata file 0.0s
[+] Running 6/8
  ✓ sender Built 0.0s
  ✓ superset Built 0.0s
  ✓ Container mosquitto Recreated 0.7s
  ✓ Container redis Running 0.0s
[+] Running 7/9
  ✓ sender Built 0.0s
  ✓ superset Built 0.0s
  ✓ Container mosquitto Recreated 0.7s
  ✓ Container redis Running 0.0s
  ✓ Container zookeeper Running 0.0s
  ✓ Container kafka-connect Recreated 0.2s
  ✓ Container kafka Running 0.0s
  ? Container ids_superset_galvis_yanez-sender-1 Recreate 7.2s
  ! Container superset Recreate 1.7s
```

Figure 2: Docker Compose launching all services

After waiting for **Kafka Connect** to initialize, the **MQTT** → **Kafka** connector will be registered with the following configuration:

```
{
  "connector.class": "io.confluent.connect.mqtt.MqttSourceConnector",
```



```
"tasks.max": 1,
"mqtt.server.uri": "tcp://mosquitto:1883",
"mqtt.topics": "tweets",
"kafka.topic": "tweets",
"value.converter": "org.apache.kafka.connect.converters.ByteArrayConverter",
"confluent.topic.bootstrap.servers": "kafka:9092",
"confluent.topic.replication.factor": 1
}
```

```
Detailed connector status:
{
  "name": "mqtt-source",
  "connector": {
    "state": "RUNNING",
    "worker_id": "connect:8083"
  },
  "tasks": [
    {
      "state": "RUNNING",
      "id": 0,
      "worker_id": "connect:8083"
    }
  ],
  "type": "source"
}
```

Figure 3: MQTT connector registered

Once the connector has been successfully initialized, the first messages received in **Kafka** can be visualized:

```
docker exec -it kafka-connect \
  kafka-console-consumer \
    --bootstrap-server kafka:9092 \
    --topic tweets \
    --from-beginning \
    --property print.value=true \
    --max-messages 3
```

```
First messages received on Kafka topic «tweets»:
{"user_id": 17519984, "tweet": "You can do more for each other than the healthcare system can do for you. ~ @chamath #launch2013", "timestamp": "2025-06-21T12:06:40.684270Z"}
{"user_id": 1109585383589851140, "tweet": "Such a spectacular show with such wonderful characters, storylines, and writing deserves to be renewed @netflix\u2026 https://t.co/qytm75nLph", "timestamp": "2025-06-21T12:06:44.689085Z"}
{"user_id": 829196098472464387, "tweet": "@realDonaldTrump Nah...\ud83d\ude02 I'm pretty sure Silent Bob Mueller be like... https://t.co/ED0dD5XVUD", "timestamp": "2025-06-21T12:06:48.696693Z"}
Processed a total of 3 messages
```

Figure 4: Tweets appearing in Kafka topic

Visualizing the messages in the console confirms that **Kafka** is already receiving messages from the **tweets** topic, so we can proceed to start **Druid**:

```
docker run -d --name druid \
  --network kafka-net \
```



```
-p 8888:8888 -p 8082:8082 -p 8081:8081 \  
-e DRUID_XMS=512m -e DRUID_XMX=1g \  
-e DRUID_EXTENSIONS_LOADLIST='["druid-kafka-indexing-service"]' \  
fokkodriesprong/docker-druid
```

2. Once **Druid** has been initialized, we can access it at `localhost:8888`.

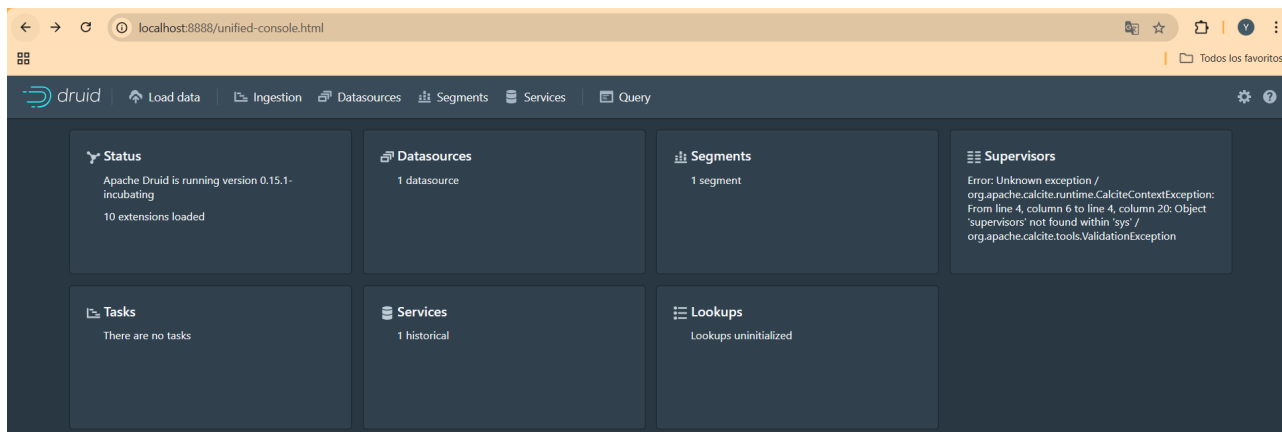


Figure 5: Druid Deployment

We navigate to the **Load Data** section and connect our data via Apache **Kafka**.

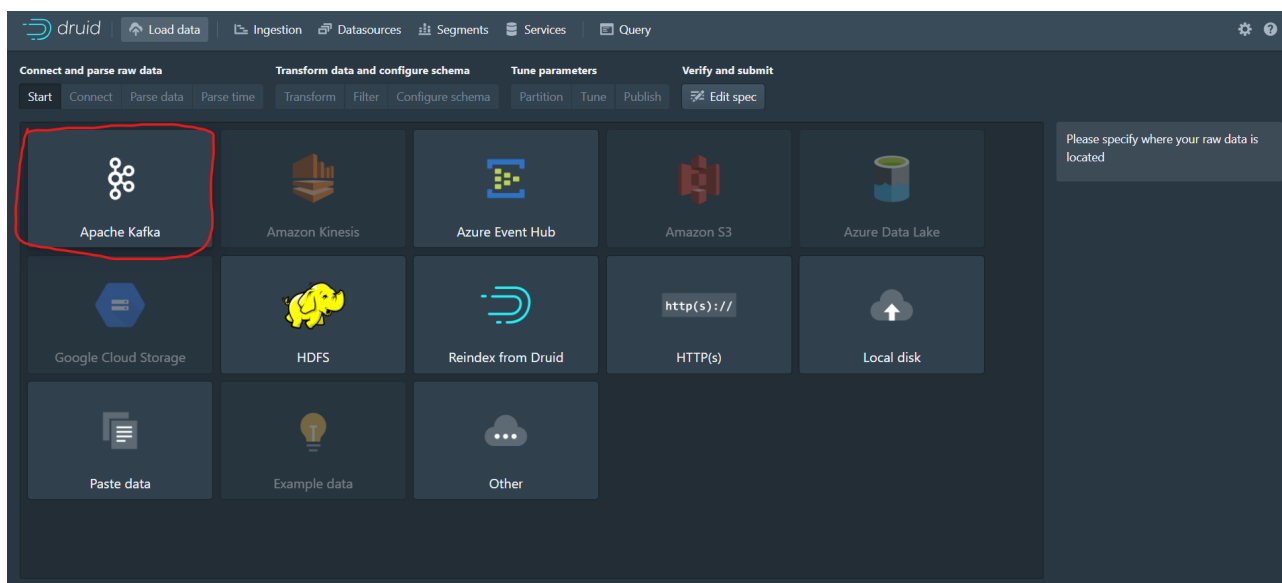


Figure 6: Ingestion from Kafka in Druid

We fill in the required fields for ingestion. In **Bootstrap servers**, we must specify the address of the **Kafka** broker that **Druid** should connect to in order to consume messages — in our case, `kafka:9092`. In **Topic**, we specify the Kafka topic from which to read the messages. Then, we click **Apply**.



Druid ingests raw data and converts it into a custom, indexed format that is optimized for analytic queries.

To get started, please specify what data you want to ingest.

[Learn more](#)

Bootstrap servers ⓘ

kafka:9092

Topic

tweets

Consumer properties ⓘ

```
{  
  "bootstrap.servers": "kafka:9092"  
}
```

Where should the data be sampled from?

Start of stream

Apply Cancel

Next: Parse data →

Figure 7: Bootstrap server and topic defined

3. We access **Superset** through localhost:8088 and fill in the username and password fields (admin - admin).

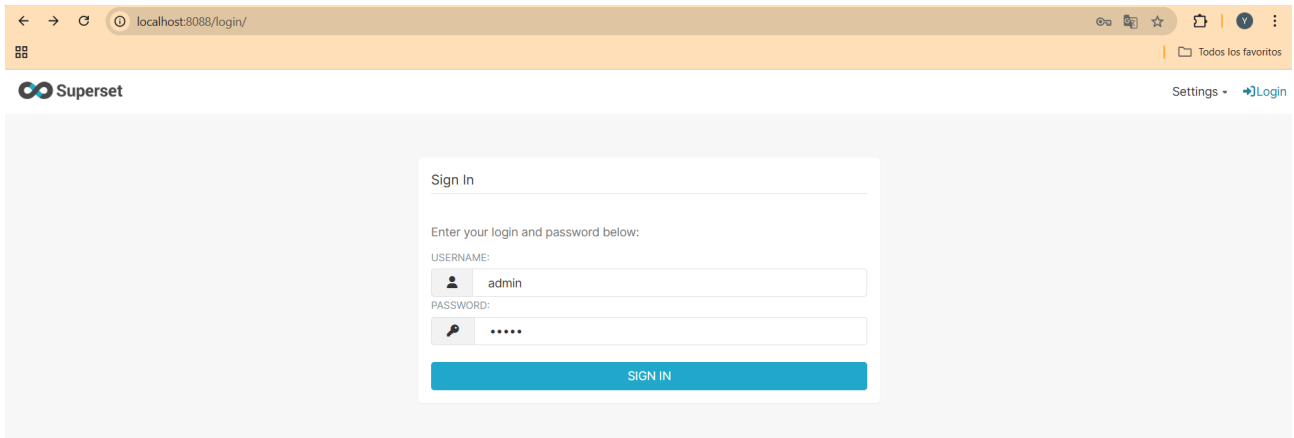


Figure 8: Superset login

We navigate to the **Databases** section, where we should be able to see our **tweets** data in order to generate visualizations.

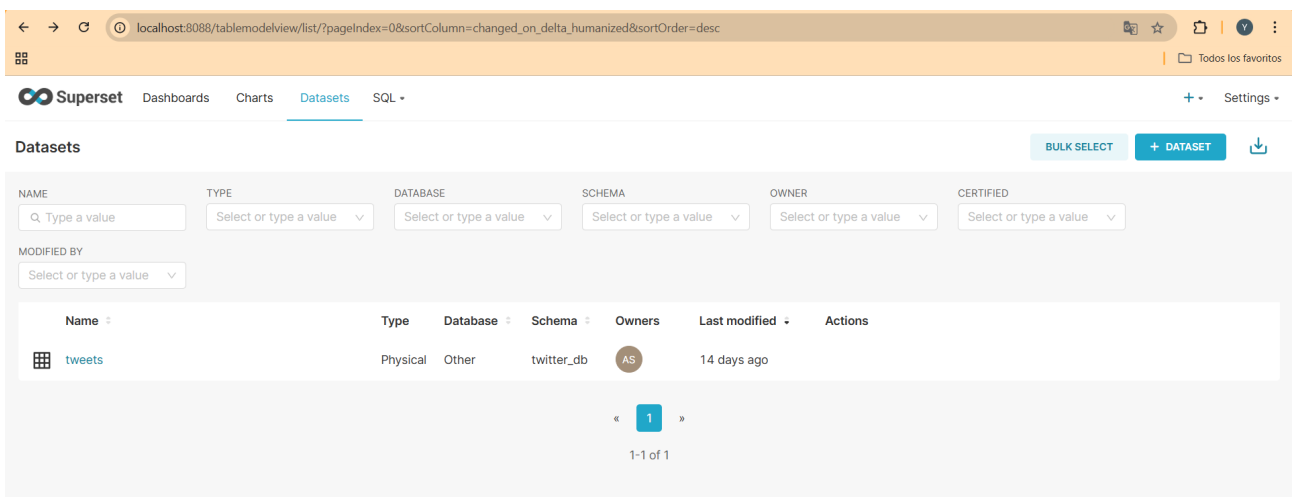


Figure 9: Tweets table for charting

NOTE: Although all services were successfully launched, we were unable to complete the ingestion into *Druid*. As a workaround, the dynamic data was loaded into *Hive* to enable visualizations.

2 Static Data (users1.json, edges1.json, mbti_labels.csv)

This part of the project focuses on the ingestion, transformation, and analysis of static data from local files: `users1.json`, `edges1.json`, and `mbti_labels.csv`. These files contain structured information about Twitter users, their following relationships, and their MBTI personality types, respectively.

The goal is to convert this information into analytical tables that can be queried and visualized through tools such as **Apache Superset**, using **Hive** as a distributed query engine.

To achieve this, an automated pipeline was implemented to prepare the data, load it into Hive, and make it available in Superset for further analysis.

2.1 Apache Hive

Apache Hive is a data warehouse system built on Hadoop that enables the analysis of large volumes of data using a SQL-like language called HiveQL. Hive translates these queries into MapReduce jobs, facilitating interaction with data stored in structured formats.

In this project, Hive acts as a repository and query engine for static data, allowing tools like Superset to easily explore and visualize it. Thanks to its SQL support, it is ideal for integrating processed data into analytical environments accessible to non-technical users.

Procedure

1. **Preprocessing with jq:** An automated script (`hive_pipeline.sh`) transforms JSON files into JSONL (JSON Lines) format for easier processing by Hive. It also makes adjustments to lists encoded as comma-separated strings.

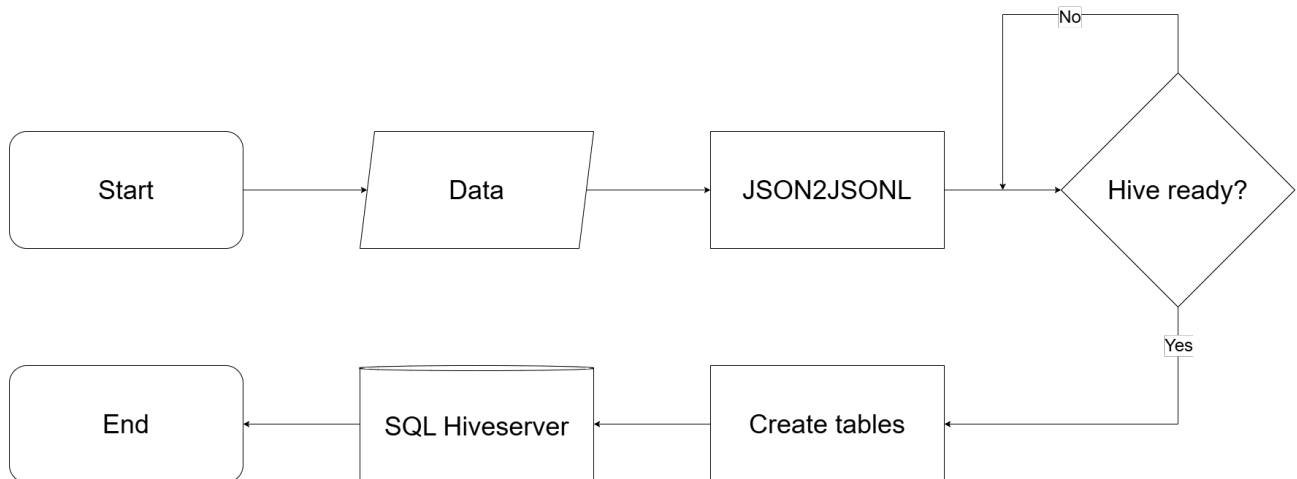


Figure 10: Pipeline start using Docker Compose

2. **Folder Organization:** Structured folders are created within `/workspace/data/` to facilitate the orderly ingestion of each dataset.
3. **Loading to Hive:** Once Hive is up and running, the `hive-init` container runs the load script, which creates the necessary tables in Hive and loads the transformed data.
4. **Access from Superset:** Finally, Superset connects to the Hive database and allows exploration and visualization of this static data.

To activate this section of the pipeline, run the following command:

```
docker-compose -f docker-compose-static.yml
```

Once the process runs successfully, access the network where the project is being deployed from a web browser. By default, it launches on `localhost:8088`. The login, by default, is defined as user `admin` and password `admin`. This can be modified in the `superset` service in `docker-compose-static.yml` if desired.

Once you log in, the Superset welcome page should appear. To connect to the Hive database:

- Click on **Settings** → **Database Connections**.
- Then, click on **CREATE DATABASE**.

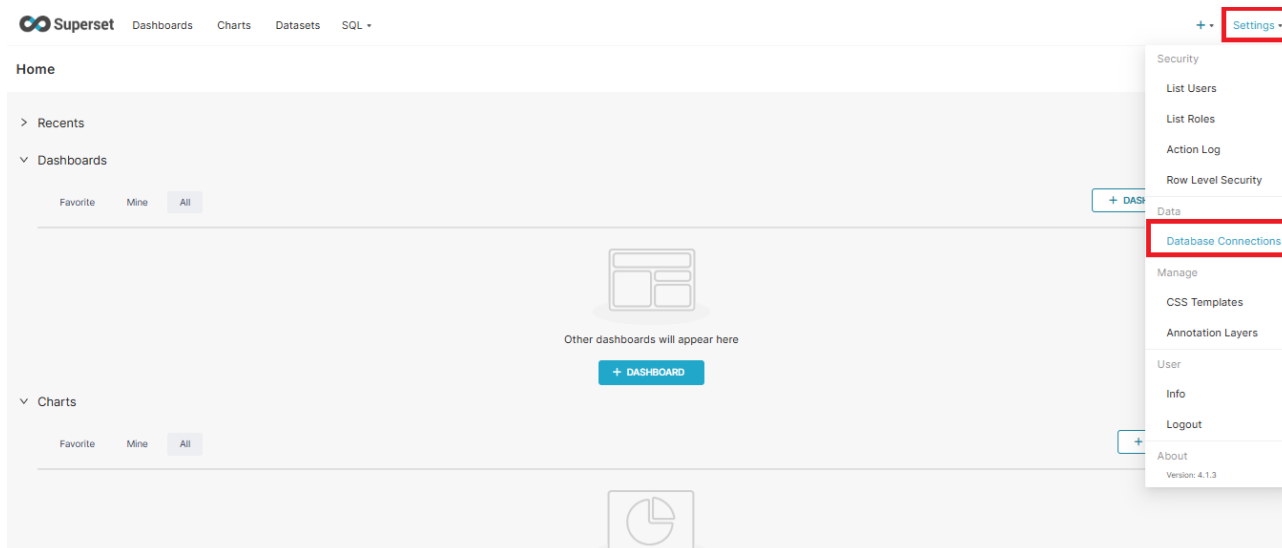


Figure 11: Superset welcome page

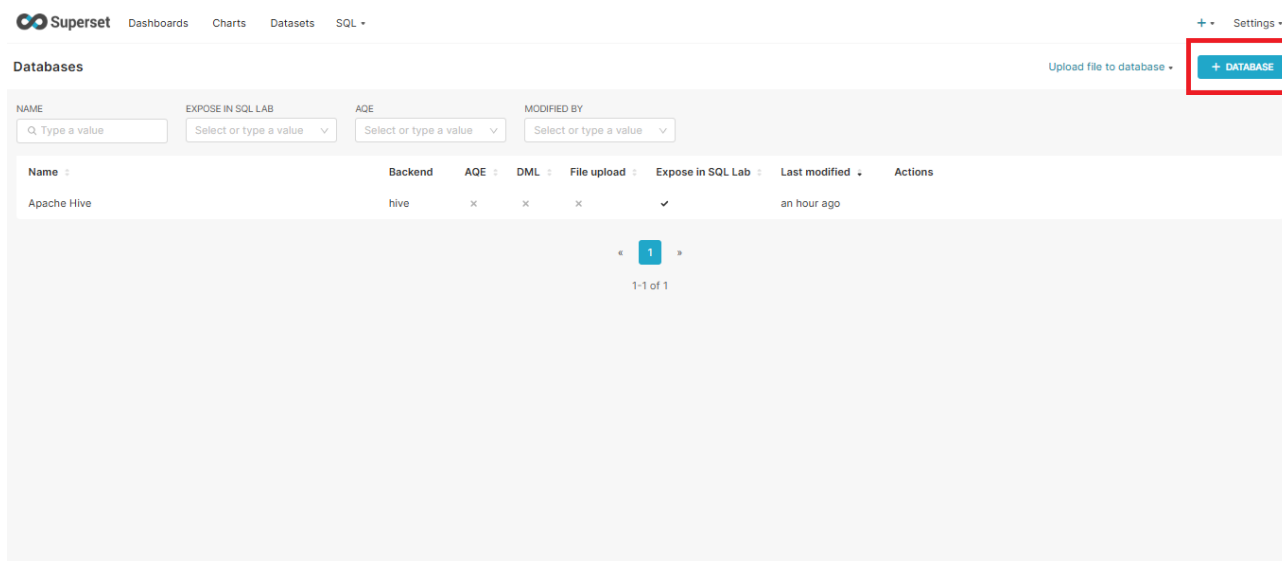


Figure 12: Creating a new database

A pop-up window will appear. From the drop-down list, select **Apache Hive**. Then, in the SQLAlchemy URI field, enter the following connection string:

```
hive://hive@hive:10000/default
```

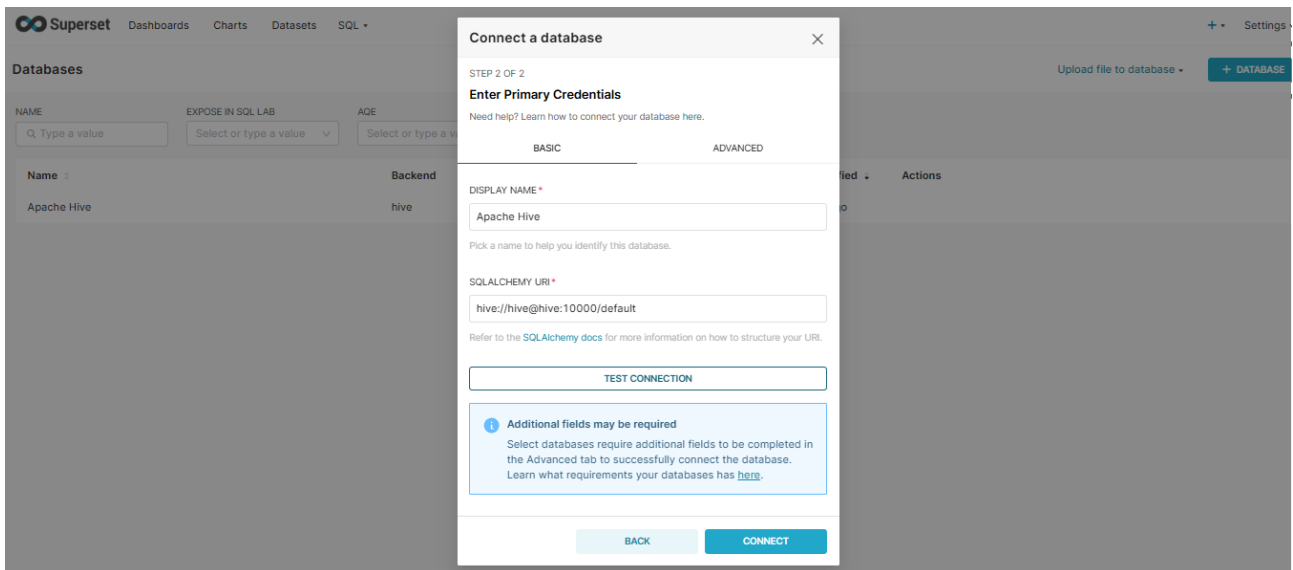


Figure 13: Hive SQLAlchemy connection setup

Click **Connect**, and Superset will now be connected to Hive.

To run SQL queries and process data, go to **SQL Lab** from the main menu:

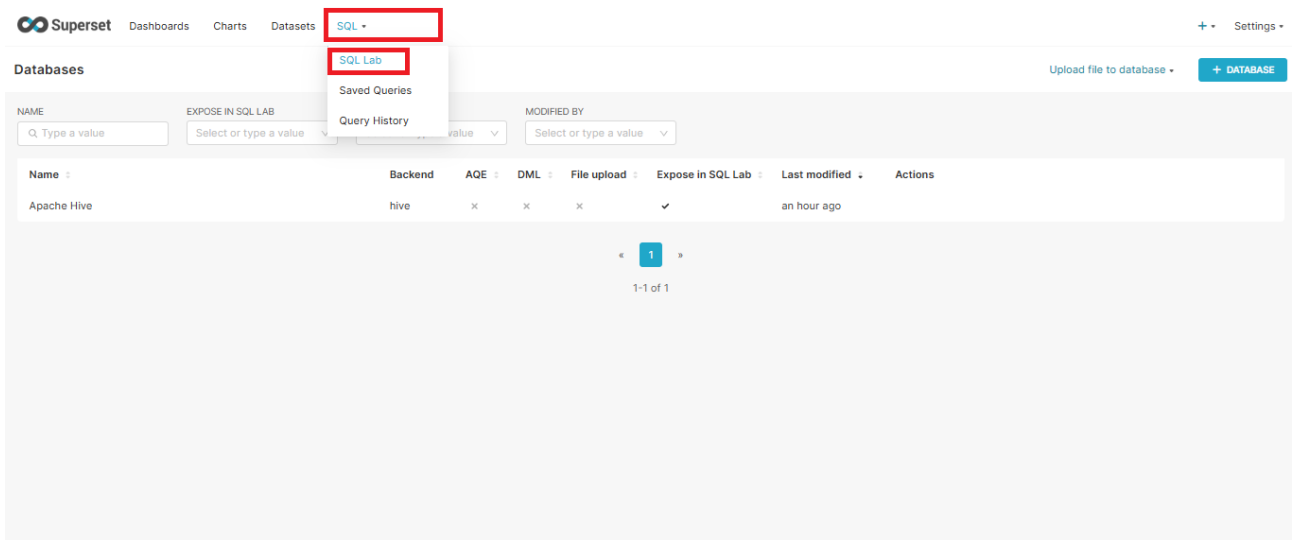


Figure 14: SQL Lab access

Select the database, then enter and execute your desired queries:

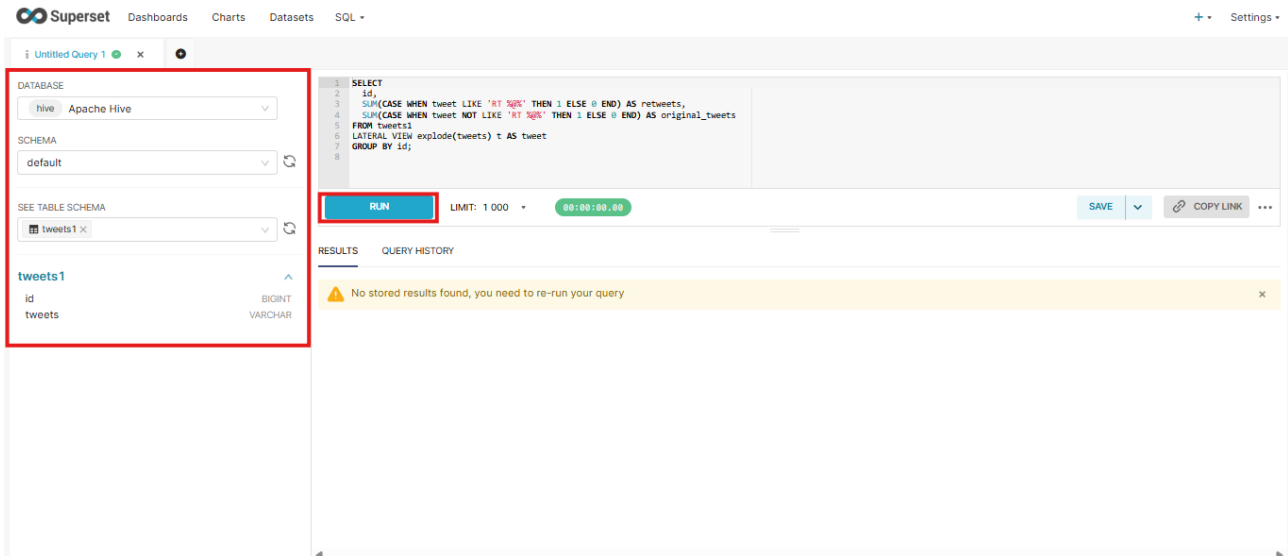


Figure 15: Executing queries in SQL Lab

3 Visualization

Once the temporal and static data processing pipeline has been executed, the data becomes available for visualization and analysis in *Superset*.

In this web-based visualization tool, it is possible to run *Hive-SQL* queries to explore, transform, and aggregate stored data, enabling the creation of interactive charts and custom dashboards that facilitate the analysis of relevant patterns and trends.

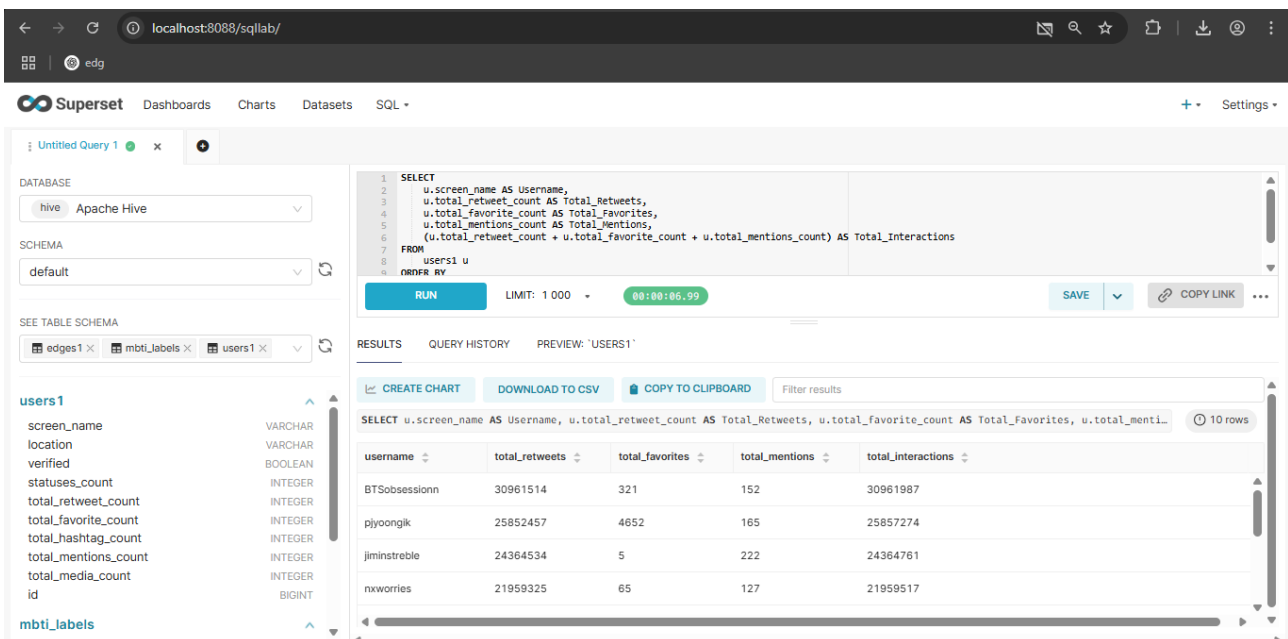


Figure 16: Hive SQL query

These queries allow us to explore and filter the database in order to generate visualizations that address key business questions. For example:

- Who are the users with the highest number of interactions?
- Which MBTI personality type generates the most content on Twitter?
- Which profile accumulates the most retweets, favorites, or mentions?
- What is the distribution of interactions across different personality types?

Below is a dashboard designed to answer this type of question:

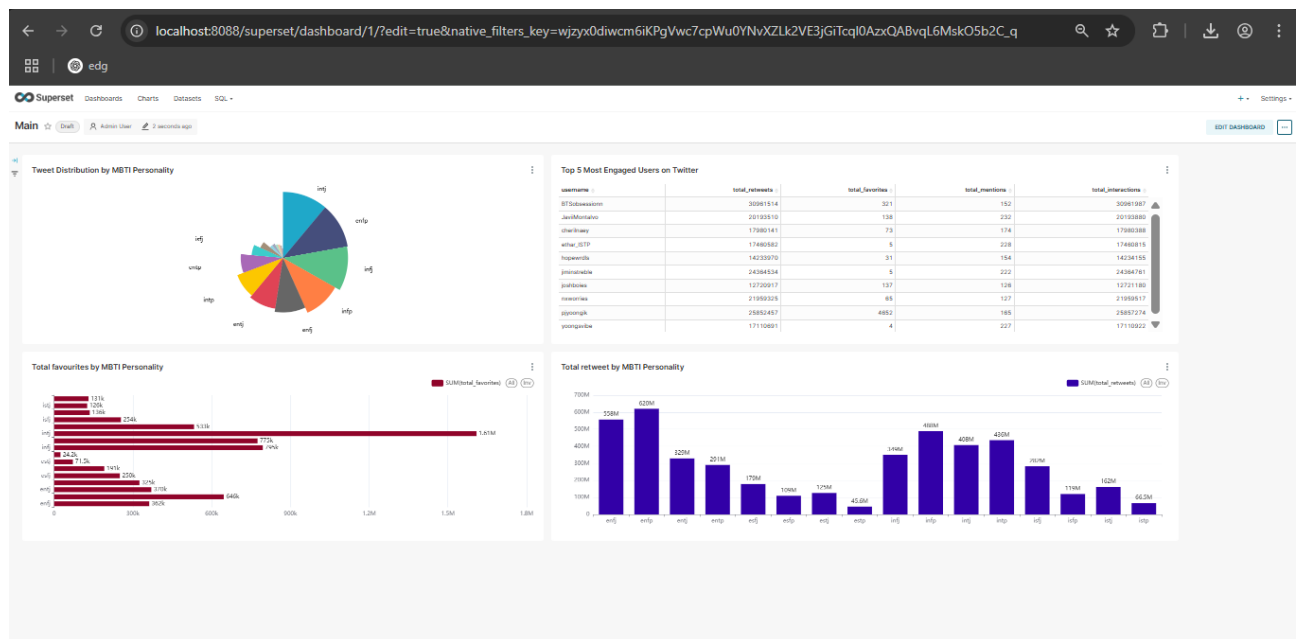


Figure 17: Superset dashboard

In the top left, a pie chart titled *Tweet Distribution by MBTI Personality* shows the proportion of tweets posted by users belonging to each of the 16 MBTI personality types. This chart reveals the most active profiles in terms of posting activity, highlighting **intj**, **enfp**, **infj**, and **infp** as the most frequent tweeters.

To its right, there is a table titled *Top 5 Most Engaged Users on Twitter*, sorted by total number of interactions. Each row displays the user's *screen name* along with the number of retweets, favorites, mentions, and the total of these metrics.

At the bottom left, a horizontal bar chart titled *Total Favorites by MBTI Personality* shows the total number of favorites received by each personality type. This metric reflects the appreciation received by tweets from different psychological profiles, with the **intj** personality type receiving the most likes. This outcome correlates with the first chart, as **intj** is also the most active in publishing. Still, it surpasses both **enfp** and **infj**, which have similar tweet volumes.

Finally, the vertical bar chart titled *Total Retweets by MBTI Personality* displays the total number of retweets grouped by MBTI type. This visualization helps identify which profiles generate the most reach on the platform. In this case, **enfp** stands out as the top performer, followed by **enfp**, **infj**, **infp**, and once again **intj**.

It is worth noting that these charts can still be improved—for instance, by sorting the bar charts in descending order to enhance visual impact—but the main objective here is to demonstrate successful data integration, interaction, and visualization in *Superset*.

The dashboard is shown again below in higher resolution, exported directly from Hive. The previous screenshot was included only to demonstrate that the service had been successfully deployed.

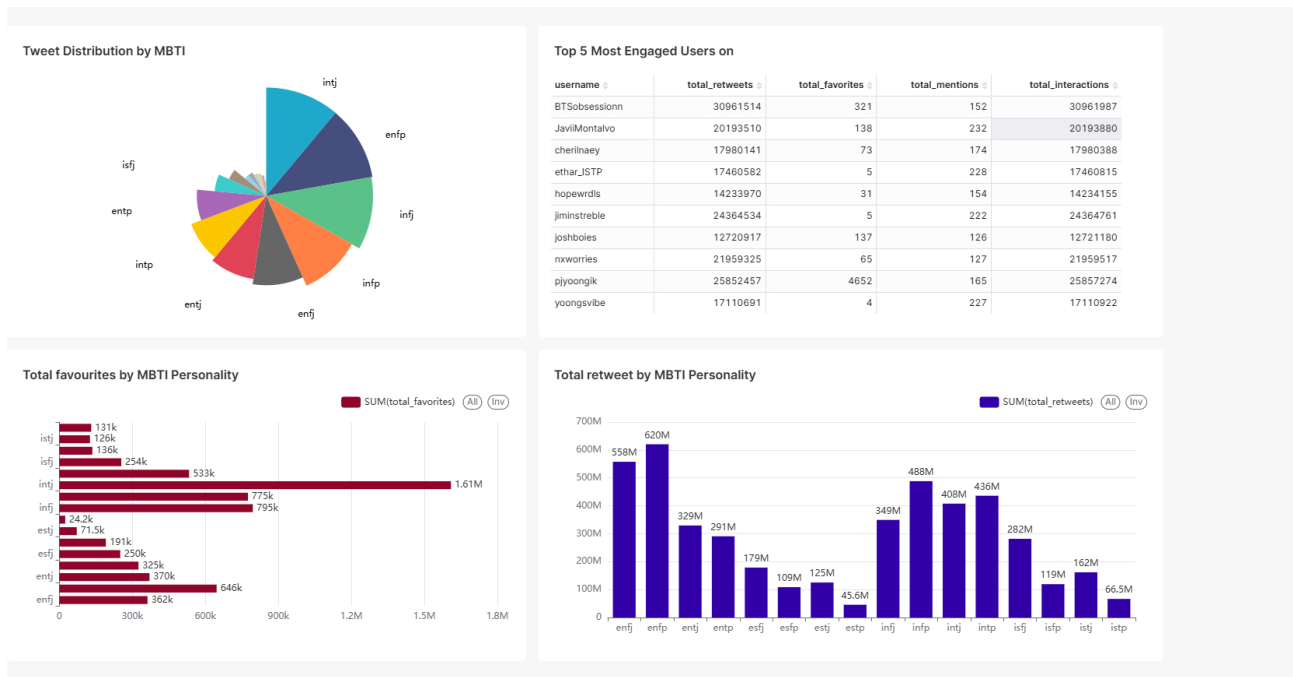


Figure 18: Superset dashboard with better resolution