Project 1: Energy Data Analysis with Apache Spark

Alessandro Finocchi
University of study of Rome Tor Vergata
Rome, Italy
alessandro.finocchi@students.uniroma2.eu

***Overview*—This project focuses on the deployment and benchmarking of a containerized Big Data analytics platform, orchestrated via Docker Compose. The platform integrates five core components: Apache Nifi for data ingestion, Hadoop HDFS for scalable storage, Apache Spark for distributed processing, InfluxDB for time-series storage and Grafana for real-time data visualization. The primary objective is to develop two queries on a specified dataset, then systematically measure and compare their execution times under varying configurations which will consider data formats, Spark APIs and Spark horizontal scaling.**

***Index Terms*—Docker-compose, Nifi, HDFS, Spark, InfluxDB, Grafana**

I. Introduction

The goal of the project is to use the Apache Spark data processing framework to query historical data provided by the Electricity Maps datasets [1].
In order to accomplish that, a set of framework has been used to decouple each step of the big data pipeline, from the ingestion to the processing and storage, and this helped in achieving reliable performance benchmarking, ease of development, maintenance and performance.

II. Architecture Design

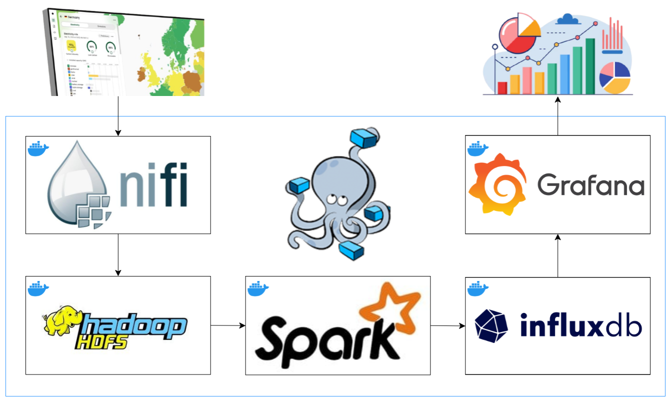


Fig. 1. Architecture

The overall system topology is the one in Figure 1: Nifi extracts and preprocesses the datasets, storing them on HDFS, which is read by Spark to compute queries, InfluxDB stores queries results and performances, and the obtained data is visible through Grafana.
Each framework is a service in the same network on top of the docker-compose orchestrator, in particular Spark and Hadoop have been defined as services in such a way that the desired number of Spark workers and Hadoop datanodes can be adjusted simply by changing a single parameter, in this way making the experiments with different scalability configuration it's easy and fast.
All the images come from the relative official repository except for the Hadoop one, which is a branch from [2].
The resource allocation and isolation has not been fully considered since the limited available hardware, the benchmarking will be evaluated with a minimum runlevel machine configuration and without any unneeded host process.

III. Data Ingestion

Starting from the data sources, the queries involved the Electricity-maps datasets [1] about the Italian and Sweden data from 2021 to 2024, for a total of 8 datasets: for ingesting the datasets Apache Nifi has been used, a system to process and distribute data that is easy to use thanks to its intuitive WebUI, yet powerful and reliable, and guarantees secure data transfer.

A. Nifi flow

The Nifi flow used is the one in Figure 2 and Figure 3: it manages the ingestion from the online repository of datasets all the way up to HDFS, with some preprocessing for aggregating the different files. The steps are the following:

1) GenerateLinks provides the 8 links of the datasets as a single flowfile
2) SplitLinks generates a set of flowfiles, each one with a single dataset link
3) ExtractLinks assigns to each flowfile the link in the text to the "url" property
4) DownloadDatasets invokes for each link the "url" property of the flowfile and downloads the dataset
5) FilterCols removes the columns not needed in the project from each downloaded dataset
6) RemoveHeader removes from each dataset the header
7) UnionDatasets append the 8 datasets to make them a single large dataset
8) Rename and Put branches allow to store on HDFS the file in the csv and parquet format

B. Dataset storage strategy

The decision to combine the files was driven by considerations related to HDFS block management: each original dataset was approximately 1 MB in size, which is significantly smaller

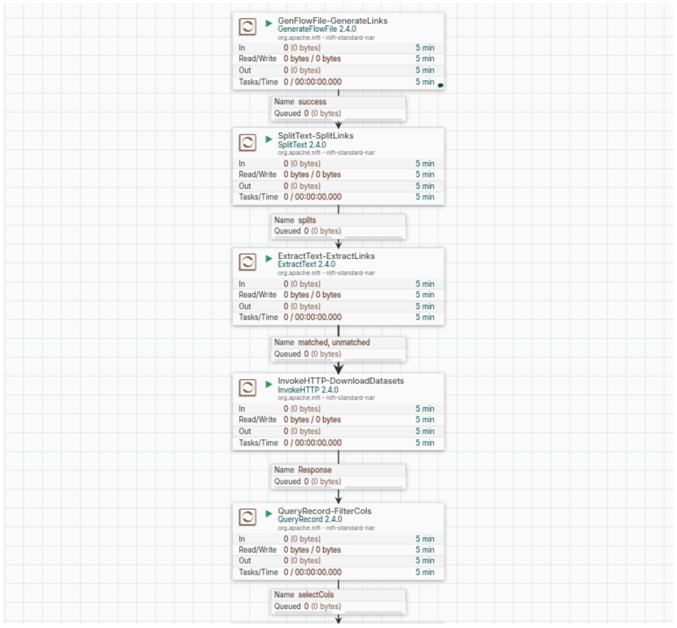


Fig. 2. Nifi flow pt.1

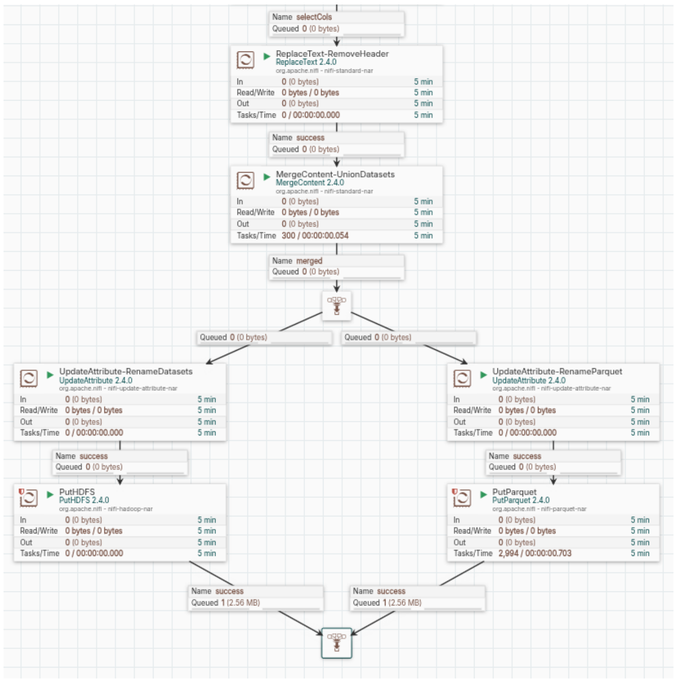


Fig. 3. Nifi flow pt.2

than the default HDFS block size of 128 MB. If each file were stored individually, it would result in a large number of blocks being allocated, which would be mostly underutilized, leading to a waste of storage resources. Instead, merging the datasets into one larger file in HDFS can better optimize block allocation, improving the overall storage efficiency and performance during the data processing tasks of the pipeline.

C. *Automating flow through REST API*

The flow is provided to the Nifi filesystem in its Dockerfile, and this allows to load it up since the container creation, so it will execute as soon as the cluster is built.
To have more control about the flow, REST API has been used: the file `nifi/runner.sh` provides a set of curl command for accessing sequentially the Nifi functionalities, from the access via JWT token up to process group status control, and the same functionalities has been implemented also programmatically in the `spark/src/deps/nifi_utils.py` file in order to allow programmatic control of the storage and ingeston status during Spark jobs.

IV. Storage and Persistence

A. *Hadoop HDFS*

The filesystem underneath the application is a cluster of Hadoop HDFS, a distributed filesystem for managing large files through parallelism, and it's been configured to be arbitrarily scalable at container-build time: indeed, as explained in the usage guide of the project repository, the number of namenodes is an easy configurable parameter. Query results can be visualized inside the HDFS directory `/data/results` after being processes.

B. *InfluxDB*

Additionally, the cluster has been integrated with InfluxDB too, a time-series database to store the query results: the reason for choosing InfluxDB was driven by the fact that the query results themselves are time-series data, so it was the most natural and efficient way to store them.

V. Data Processing

A. *Query 1*

Following the ingestion phase, the next step in our data pipeline was processing data through statistical queries.
In the development of the queries, great attention was paid to ensuring the decoupling of components, maintaining a clean yet modular code structure. Moreover, the two queries were implemented using multiple APIs (RDD, DataFrame, and SQL) and multiple file formats (csv and parquet) in order to analyse the performances for each approach. This structure provided ease of development of new queries, also facilitating a comparison of execution times across different methods.

B. *Spark-cluster configuration*

Similarly to the Hadoop cluster, the Spark cluster is easily configurable in terms of the number of workers by adjusting a simple parameter in the command used to launch the application: this allowed, during the performance studies, to try different configuration and see how time execution changed between them. Indeed, the optimal number of workers it's not trivial, one could think "the more the merrier", but actually if too many workers were used, the latency times would overcome on the execution time, thus resulting in worse performances.

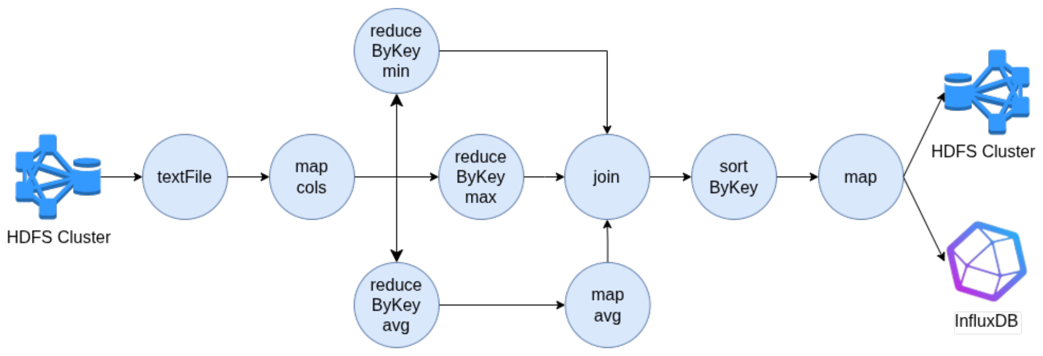


Fig. 4. Query 1 with rdds

C. The main application

The main application, which works as the entry point for the execution of the queries, is designed to accept command-line arguments for selecting the query number, API and file format to run the query. It dynamically imports the appropriate query module[1] based on the user's input, checks the availability of the dataset, and if it's not available triggers a Nifi flow to retrieve the data before proceeding execution.

D. Query structure

Thanks to the main application, every query can be structured to be code-minimal and focused on data-processing, in this way there is no need to pay attention to lots of details and creating a new one is fast and easy. Eventually, the results to store could differ from query to query, that's the reason why the responsibility to save the results has been designated to the query modules, so this aspect it's not fully decoupled. It is important to note that with the structure adopted by the queries, there was no need to use caching due to the absence of multiple actions for the same RDD, in this way Spark optimization have been fully exploited. Let's now go deeper in the implementations, only the queries with RDD APIs will be treated.

E. Query 1

The first query wants to compute the minimum, maximum and average values for the "*Carbon intensity gCO2eq/kWh (direct)*" and "*Carbon-free energy percentage (CFE%)*" dataset features for each year and country. Its DAG is shown in Figure 4:

1) it starts reading the dataset from the HDFS cluster using `textFile`
2) `maps` the columns in the format

   ((country, year), (int, cfe, 1))
3) aggregates by key for finding the maximum and minimum feature values with `reduceByKey`
4) aggregates by key for computing the sum of column values with `reduceByKey` and then divides by the number of instances for computing the average via `map`
5) unites all the maximum, minimum and average results through `join`
6) puts results in order via `sort`
7) puts columns in the requested order via `map`
8) lastly saves the results on both HDFS Cluster and InfluxDB

[1]The query module must respect some naming convention: query n with API <api> must be inside the python package "src/query$n$" and must be named "query$n$_<api>.py"

F. Query 2

The second query wants to compute the average values of "*Carbon intensity gCO2eq/kWh (direct)*" and "*Carbon-free energy percentage (CFE%)*" dataset features for each month and year (this result will be called *progress*), and at the same time take the top 5 and bottom 5 entries for each feature (this other result will be called *classification*). Its DAG is shown in Figure 5:

1) it starts reading the dataset from the HDFS cluster using `textFile`
2) `maps` the columns in the format

   ("year_month", (int, cfe, 1))
3) aggregates by key for computing the sum of column values with `reduceByKey` and then divides by the number of instances (third value) for computing the average via `map`
4) for computing progress (branch above):
   a) flattens the rdd with a `map`
   b) puts results in order by year and month via `sort`
5) for computing classification (branch below):
   a) takes the first 5 entries sorted in ascending/descending order for each feature using `takeOrdered`
   b) converts the result back to an RDD (since takeOrdered is an action) using `parallelize`
   c) unites all the results through `union`
   d) puts columns in the requested order via `map`
6) lastly, progress is saved in both InfluxDB and the HDFS cluster while classification is saved only in the HDFS cluster since it's not meant to be displayed in any chart.

VI. Results Evaluation

For visualizing results Grafana was used. In the file grafana/notes.txt there is a guide to use Grafana.

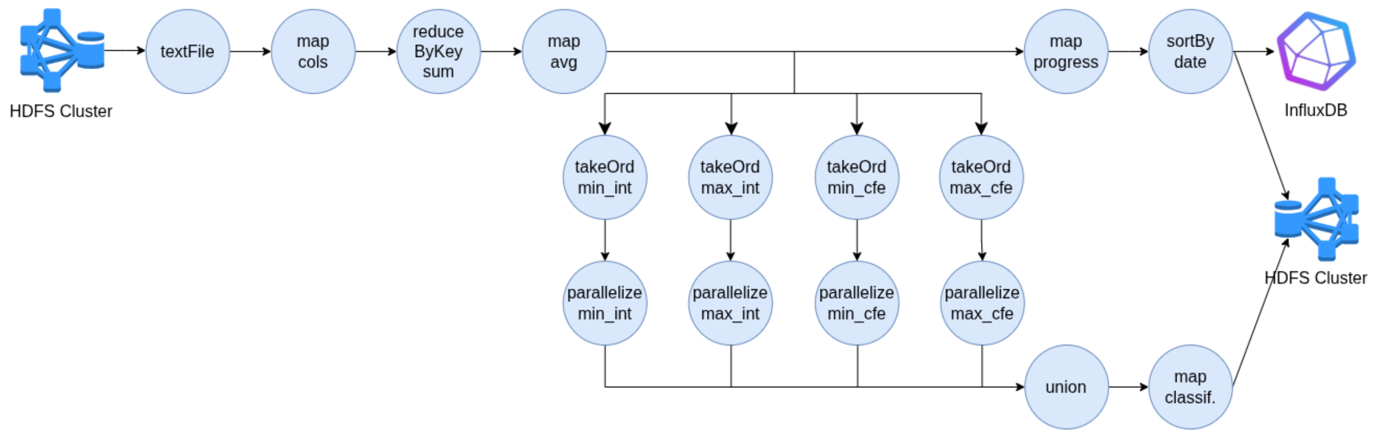
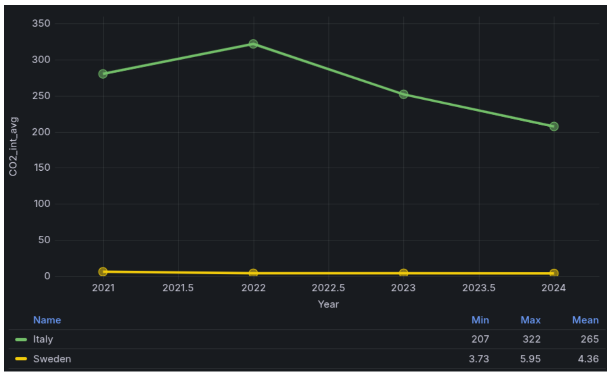
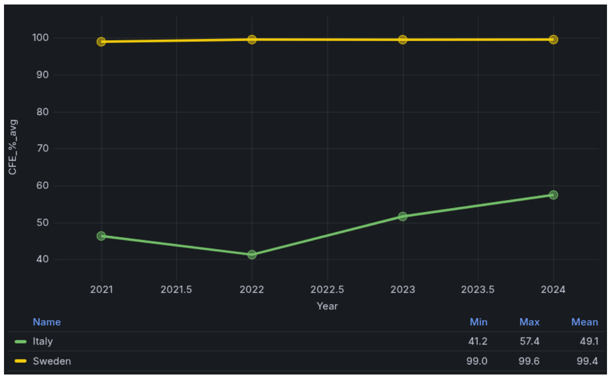


Fig. 5. Query 2 with rdds



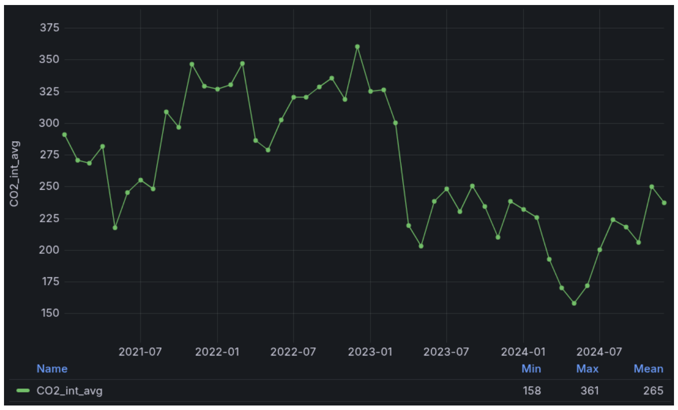
Name	Min	Max	Mean
Italy	207	322	265
Sweden	3.73	5.95	4.36

Fig. 6. Average CO2 intensity comparison



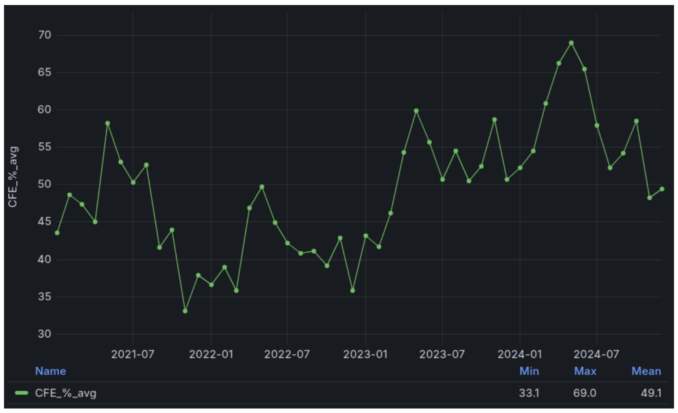
Name	Min	Max	Mean
Italy	41.2	57.4	49.1
Sweden	99.0	99.6	99.4

Fig. 7. Average CFE comparison



Name	Min	Max	Mean
CO2_int_avg	158	361	265

Fig. 8. Average italian CO2 intensity trend



Name	Min	Max	Mean
CFE_%_avg	33.1	69.0	49.1

Fig. 9. Average italian CFE trend

A. *Query 1*

The first query wanted to visually compare the Italy and Sweden trend on the yearly average of the two metrics. The results are shown in Figure 6 and Figure 7: as we can see, Italy is far behind Sweden in terms of both carbon intensity and carbon-free energy percentage usage.

B. *Query 2*

The second query wanted to visually evaluate the Italian trend of the monthly average of the two metrics. The results are show in Figure 8 and Figure 9: as shown in the figures, there appears to be a general downward movement from 2023 onwards in the CO2 intensity trend and a general upward movement from 2023 in the CFE trend, but the overall behaviour seems very unstable and the global improvement doesn't look that much significant.

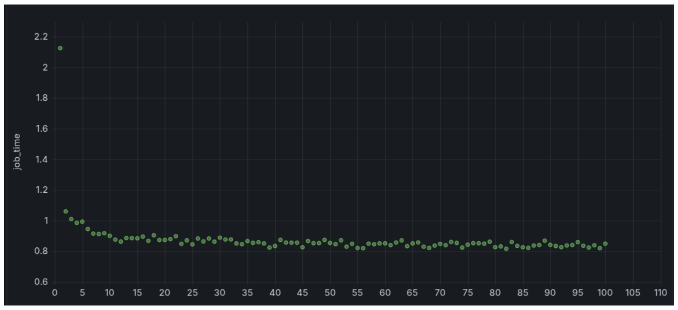


Fig. 10. Trend of execution time

TABLE I
QUERY 1 EXECUTION TIMES IN SECONDS

API-Format	min	max	avg	median	variance
Results with 1 Spark worker					
rdd-csv	0.816	2.13	0.875	0.854	0.0174
df -csv	0.395	4.43	0.524	0.444	0.25
sql-csv	0.394	5.16	0.546	0.446	0.336
df -parquet	0.432	5.60	0.578	0.499	0.284
sql-parquet	0.384	5.69	0.566	0.485	0.316
Results with 2 Spark worker					
rdd-csv	1.23	3.47	1.35	1.32	0.0495
df -csv	0.691	7.09	0.868	0.792	0.399
sql-csv	0.634	6.93	0.804	0.724	0.391
df -parquet	0.425	5.59	0.573	0.492	0.294
sql-parquet	0.376	5.68	0.566	0.488	0.325

C. Query performances

Query performances were evaluated starting before the dataset reading and ending after collecting the results: the queries were created with different configuration of API, file format and number of Spark workers. Each configuration have been executed for 100 times and the execution times (in seconds) of these runs can be seen at Table I and Table II.
As we can see, the maximum execution time is much greater than its mean value: the reason why it happens is that Spark apply resource usage optimization at runtime, which brings lower execution times as runs go on repeating. This optimization brings performances to increase since the second run: this is shown in Figure 10, where we can see the typical behaviour of the execution time which fastly converges and stabilize itself right after the second run[2].

VII. SECURITY ACCESS

In this project credentials have been securely managed to ensure the protection of sensitive data. To address this, two distinct approaches were adopted to manage the access credentials for InfluxDB and Nifi, which will be stored in specific files that can be excluded from VCS through a .gitignore, thus eliminating the chances of credential exposure[3].

[2]The image represent the experiment for query 1 with rdd APIs, csv file format and 1 Spark worker, but the same trend characterize all the other curves.

[3]For the project assignment they will not be excluded to simplify the delivery.

TABLE II
QUERY 2 EXECUTION TIMES IN SECONDS

API-Format	min	max	avg	median	variance
Results with 1 Spark worker					
rdd-csv	2.02	4.11	2.15	2.12	0.0436
df -csv	0.461	5.26	0.607	0.516	0.252
sql-csv	0.463	5.63	0.610	0.523	0.276
df -parquet	0.585	6.62	0.763	0.669	0.359
sql-parquet	0.591	6.55	0.751	0.665	0.351
Results with 2 Spark worker					
rdd-csv	2.82	5.02	3.00	2.98	0.0482
df -csv	0.856	7.58	1.07	0.970	0.450
sql-csv	0.816	7.82	1.08	0.981	0.482
df -parquet	0.591	6.49	0.760	0.668	0.343
sql-parquet	0.588	6.50	0.748	0.660	0.346

A. Using secrets for InfluxDB credentials

The access credentials for InfluxDB are stored in separate secret files, rather than being embedded directly in the source code or the docker-compose file. This approach reduces the risk of accidental exposure of credentials through source code control or unauthorized access to configuration files.

B. Using environment for Nifi credentials

Nifi credentials, on the other hand, are stored in the .env file. This file is used to securely manage environment variables without needing to include them in the source code or docker-compose file.

VIII. CONCLUSIONS AND FUTURE WORK

A. Key takeaways

From Table I and Table II we can clearly see that in this use case, where data are not many, using more Spark workers doesn't give any consistent advantage, indeed for the csv configuration having 2 Spark workers increases average execution times of about 60%-70%, while in the parquet configuration they are practically the same. Furthermore, dataframe and sql APIs show significant increasing of average performance, and even though rdd variance is the lower one, the dataframe and sql APIs look like the most preferable.

B. Future work

The project is easily extendable with new queries: it could be interesting to test the possible configurations with much larger data to understand the validity of the obtained results in real case scenarios.
Moreover, Docker Compose could be moved to Kubernetes for dynamic scaling if a query is particularly resource-eager and see if the loading times overcome the execution times.

REFERENCES

[1] Electricity Maps. Carbon Intensity Data. 2025. https://www.electricitymaps.com/.
[2] https://github.com/matnar/docker-hadoop.